Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DCT-LNS: Integration of The Large Neighborhood Search Algorithm into the DCT

Ana Paula M Tarchetti

Undergraduate thesis submitted presented as partial requirement
for completion of the Bachelor's Degree in Computer Science

Supervisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DCT-LNS: Integration of The Large Neighborhood Search Algorithm into the DCT

Ana Paula M Tarchetti

Undergraduate thesis submitted presented as partial requirement
for completion of the Bachelor's Degree in Computer Science

Prof. Dr. Rodrigo Bonifácio de Almeida (Supervisor)
CIC/UnB

Profa. Dra. Edna Dias Canedo
CIC/UnB

Dr. Luís Henrique Vieira Amaral
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordinator of the Bachelor's Degree in Computer Science

Brasília, May 25, 2021

Dedicatory

I dedicate this work to everyone who supported me in this undergraduate journey, my family, friends and teachers.

Acknowledgments

Initially, I would like to thank all the time and patience that my supervisor made available to help me in this work. I would also like to thank my colleagues: Luís Amaral, João Neves, and Marcos César, each of them did extremely important work that made up our platform. Finally, I would like to thank the past researchers who have been working on this same theme over the years and helped in the construction of an initial knowledge base that was extremely helpful.

Abstract

This study operates in the area of Software Quality in order to help overcoming the problems related to the lack of documentation and architecture planning of a software system. This can be done by recovering the system architecture, using automatic module clustering tools. Thus, was presented the integration of one of the multiple automatic module clustering tools that compose the Draco Clustering Tool (DCT). This tool is based on a mono-objective implementation, in which only the Modularization Quality (MQ) parameter is considered to perform the clustering of the software's modules. This implementation was done by means of the evolutionary metaheuristic algorithm Large Neighborhood Search (LNS). In this context, two comparative analyses were performed. The focus of the first analysis was on the existing multi-objective tool of *Draco Clustering Tool* and showed that it can be considered scalable. Furthermore, in this first analysis it was also possible to see how effective the multi-objective tool from DCT is in terms of runtime and memory usage in relation to another multi-objective tool available in the literature. Finally, this first analysis also shows the better performance of the mono-objective tools in terms of all the metrics used in the study. Taking this into consideration, the motivation for this study arose, which is the integration of another mono-objective tool into the DCT, which was evaluated in the second comparative analysis done by this study, in which it was possible to see that, the integrated tool: (i) is also scalable, possessing in a cubic time complexity, (ii) showed better performance compared to the other tool that also used LNS in terms of the runtime and memory usage metrics, and (iii) also shows good performance compared to other tools with other algorithm in terms of all metrics except runtime when in comparison with the *Bunch* tool. This last item is a possibility for improvement to be explored in future work.

Keywords: Software Module Clustering, Multi-Objective, Mono-Objective, Evolutionary Algorithms

Resumo

Este estudo atua na área de Qualidade de Software no sentido de aulixiar na superação dos problemas relacionados à falta de documentação e de planejamento da arquitetura de um sistema de *software*. Isso é feito por meio da recuperação da aquitetura do sistema, utilizando-se de ferramentas automáticas de clusterização de módulos. Sendo assim, foi apresentado a integração de uma das múltiplas ferramentas automáticas de clusterização de módulos que compõem a *Draco Clustering Tool (DCT)*. Essa ferramenta em questão se baseia em uma implementação mono-objetiva, na qual se considera apena o parâmetro de Qualidade de Modularização para realizar a clusterização dos módulos de um *software*. Essa implementação foi feita por meio do algoritmo meta-heurístico evolucionário *Large Neighborhood Search (LNS)*. Nesse contexto, foram realizadas duas análises comparativas. O foco da primeira análise foi a ferramenta multi-objetiva já existente da *Draco Clustering Tool (DCT)* e mostrou que ela pode ser considerada escalável. Além disso, nessa primeira análise também foi possível perceber a eficácia da ferramenta multi-objetiva da DCT em termos de tempo de execução e uso de memória com relação a outra ferramenta mult-objetiva disponível na literatura. Por fim, essa primeira análise também mostra o melhor desempenho das ferramentas mono-objetivas em termos de todas as métricas utilizadas no estudo. Levando isso em consideração, surgiu a motivação deste estudo, que é a integração de mais uma ferramenta mono-objetiva na DCT, que foi avaliada na segunda análise comparativa feita por este estudo, na qual foi possível perceber que, a ferramenta integrada: (i) também é escalável, possindo uma complexidade de tempo cúbica, (ii) mostrou melhor desempenho em comparação com a outra ferramenta que usa o *LNS* em termos das métricas de tempo de execução e uso de memória e (iii) também mostra um bom desempenho em comparação com outras ferramentas em termos de todas as métricas, exceto o tempo de execução quando em compação com a ferramenta *Bunch*. Esse último item constitui uma possibilidade de melhoria para ser explorada em trabalhos futuros.

Palavras-chave: Clusterização de módulos, Multi-objetivo, Mono-objetivo, Algoritmos Evolucionários

Contents

List of Figures

List of Tables

Acronyms

CLI Command Line Interface.

DCT Draco Clustering Tool.

GA Genetic Algorithm.

HD-LNS Heuristic Design LNS.

LNS Large Neighborhood Search.

MDG Module Dependency Graph.

MQ Modularization Quality.

SMC Software Module Clustering.

Chapter 1

Introduction

With increasing complexity of modern software, there is an increased demand for automated tools to support the maintainability and scalability of those systems, Dahiya et al. [1]. Fundamental contributions to this subject include, for instance, the introduction of the automated Software Module Clustering (SMC) tool by Mitchell and Mancoridis [2]. This appliance began with the purpose to offer techniques to reveal the structure of a software system by grouping its modules into clusters. They based their algorithm on the principle of "low coupling and high cohesion". The input of the algorithm is a set of modules and dependencies between them. Typically, these modules correspond to files (or classes, in object-oriented programming languages), and the dependencies correspond to function/method calls or variables/fields access. While this structure of modules and dependencies are common, other representations are useful too, such as methods/fields as modules and co-change history as the dependencies [3].

Revealing the software structure by using SMC tools can help to overcome complications related to misleading or insufficient documentation. The problem concerning documentations is accurately comprehended in Lethbridge et al. [4], which is a study consisting of interviews with software engineers, and the general answers about documentation were the following: (i) documentation is frequently out of date, (ii) often poorly written, (iii) challenging in terms of finding useful content and (iv) has a considerable untrustworthy fraction. In this context, it becomes very meaningful the chase for computational mechanisms such as SMC so that the documentation gap could be filled, hence, making it possible to support the six main aspects of software development pointed out by Garlan [5]: understanding, reuse, construction, evolution, analysis, and management. Besides software structure recovering, SMC techniques can also be used as a preprocessing phase for the following activities: (a) recommend or reveal alternative decompositions, (b) recommend refactorings in order to conform to some alternative decomposition, and (c) detect anomalies in the software design. It is possible to mention two frameworks

that provide a compilation of tools that perform some of these aforementioned activities by using SMC as the preprocessing phase. The first is the recently published, ARCADE workbench [6], which is a work of several researchers that offers a five stages implementation that supports the recovery of software system architectures, and the evaluation and visualization of the architectural change and decay. The second work, named Draco, is also the result of the work of several researchers [3, 7], and is similar to ARCADE in terms of goals. This present study is part of the Draco project and will focus on evaluating and trying to improve the Draco SMC stage.

1.1 Motivation

Past researches have proposed many alternative SMC approaches [8, 9, 10, 11, 12, 13, 14], however, they failed to provide publicly available tools that use multi-objective genetic algorithms in their designs. One of the primary benefits of multi-objective algorithms is that they output a set of best solutions in contrast with mono-objective where there is only one "best" solution. The problem with pursuing only one solution is that we have to chose between conflicting objectives. For example, it is hard to chose between a solution with better cohesion or other with better coupling; i.e. for a SMC tool to find the best solution among several candidate solutions they have to decide about questions like "which is better: coupling or cohesion?" [8, 15].

However, in general, multi-objective algorithms can be more time consuming than the mono-objective options, this could be implied after a previous published article about a research conducted by a group of academics that includes the present author [16]. Several parts of this published article are presented in Chapter 3.

1.2 Proposed solution

With that in mind, this study presents the Draco Clustering Tool (DCT), a public tool that belongs to the Draco project and performs automated SMC using both multi-objective and mono-objective evolutionary algorithms. The study was separated in two phases. The first phase proposes a comparative analysis between the original DCT implementations (multi and mono objectives) and other implementations found in the literature. The second phase proposes a new mono-objective implementation based on the Large Neighborhood Search (LNS) algorithm, Pisinger et al. [17], and also performs a comparative analysis, now considering the original mono-objective implementation of the DCT, the new proposed algorithm for the DCT and other mono-objectives implementations found in the literature.

1.3 Document Structure

The present study is organized as follows: background and related work is provided in Chapter 2, Chapter 3 (i) exposes details about the original DCT, (ii) sets up the first empirical study, regarding the original DCT and other implementations and (iii) presents the results of the first empirical study. Chapter 4 (a) exposes details about the new proposed mono-objective implementation for the DCT based on the LNS algorithm, (b) sets up the second empirical study, regarding only mono-objective algorithms, and (c) presents the results of the second empirical study. Finally, Chapter 5 concludes the paper.

Chapter 2

Background and Related Work

The re-engineering process in large scale software projects requires appropriate and scalable techniques. With the focus on Software Module Clustering (SMC) techniques, the work of Anquetil and Lethbridge [18], for instance, compares different strategies for using SMC as a software remodularization recommender. More recently, Maqbool and Babri [19] investigate the use of hierarchical clustering algorithms for architecture recovering.

Praditwong et al. [8] proposed to represent the SMC problem as a multi-objective search problem. They formulated the problem representing separately several different objectives (including cohesion and coupling). The rationale of this proposal is that it is not always possible to capture the relative importance of some desirable properties (for example, it is hard to decide if cohesion is more important than coupling or vice-versa).

Candela et al. [15], investigated which properties developers consider relevant for a high-quality software remodularization. To be able to compare different properties, they had to use a multi-objective genetic algorithm to compute the software module clusters. Accordingly, they presented to the developers several recommendations of remodularization, and investigated which property (e.g. cohesion or coupling) the developers regard most. This kind of study was only possible by using a multi-objective SMC tool.

Pinto et al. [20] propose the use of mono-objective evolutionary algorithms within the Software Module Clustering Problem. Their work uses the Iterated Local Search algorithm in order to perform the module clusterization. This algorithm is found alongside the algorithm used in this study in the Handbook of Metaheuristics [21]. In addition to that, the article [20] also performed a comparative analysis between their solution and other genetic algorithms and their implementation outperformed the best configuration for the genetic algorithms in 24 out of 40 instances and using only a fraction of the computing effort. This drove attention to the possibility of using other evolutionary algorithms besides the genetic algorithms.

Other works are also worth mentioning here, because they provide different SMC

implementations. First, M. Barros discusses the effects of using the Modularization Quality (MQ) metric as an extra objective on a multi-objective SMC tools [22]. Second, Monçores et al. present a large study addressing a heuristic based on the mono-objective *Large Neighborhood Search (LNS)* algorithm, applied to SMC problems [23]. Both works published tools that were explored in this study. Finally, in a recent work concerning the Draco project, [7] was leveraged a multi-objective software module clustering tool to produce a set of alternative decompositions of a software.

Lastly, it is also worth mentioning again the ARCADE workbench [6], already mentioned in the previous chapter. The ARCADE is a framework that provides a lot of tools regarding the support for recovery of software systems' architectures, and for evaluating architectural change and decay. The interesting part that can be implied from this related work is that they use 10 integrated Software Module Clustering (SMC) tools to compose their framework, that means that it's important to have several options of SMC tool in a framework like this. Therefore, it's possible to validate the need for the integration of a further implementation in the Draco Clustering Tool (DCT).

Chapter 3

Draco Clustering Tool

Draco Clustering Tool (DCT) is a Command Line Interface (CLI) tool, that reads a *Module Dependency Graph (MDG)* [2] from the standard input and writes a clustered graph represented as a DOT[1] file in the standard output. It was implemented in Go[2] programming language, and is publicly available.[3]

3.1 Design Principles

The currently main use case of the tool is to run experiments involving multi-objective SMC computation. Accordingly, the following principles guided the design of DCT:

- **An easy to use interface.** While a Graphical User Interface potentially could be more intuitive, it makes experiments automation more difficult;
- **Minimal memory usage.** DCT users might want to run the tool in parallel, so its memory consumption must be minimal;
- **Runtime efficiency.** Similarly, the time spent running a experiment must be minimal;
- **Extensible.** To experiment with multiple scenarios, it must be possible to replace portions of the clustering algorithm or to tune its parameters values;
- **Standard formats.** To make comparisons of DCT with other tools easier, DCT must adopt well-known file formats, both for input (MDG) and output (DOT);

[1] `https://graphviz.org/doc/info/lang.html`
[2] `https://golang.org`
[3] `https://github.com/project-draco/tools/tree/master/clustering`

In order to address these principles, Go was chosen as the programming language. Go programs are compiled ahead of time to native machine code, therefore compiled programs can execute efficiently. Furthermore, this property makes the use of CLI tools more convenient, since they would not require a virtual machine to run. In addition, the extensibility principle was addressed using Go interfaces. For instance, the presence of a Go interface to abstract the random number generator (see more details bellow).

3.2 Genetic Algorithm Characterization

In DCT was used the definition of the SMC problem as a multi-objective optimization problem, using the same set of objects recommended by Praditwong et al. [8]. The input is a MDG represented by a graph $G = (V, E)$ from a set of modules V and a set of dependencies $E \subseteq V \times V$; and the output is a set of solutions. A solution is a partition of a MDG that corresponds to a set of clusters. Although the design of DCT uses a multi-objective genetic algorithm (GA) [24] to compute optimal partitions, it is also possible to extend DCT to use mono-objective algorithms.

3.2.1 Individuals and Genetic Operators

To use a genetic algorithm, it is necessary to precisely define the concept of *individuals* and *fitness functions* for the problem domain. A typical GA executes as follows:

1. It first generates an initial population (i.e., a set of individuals) randomly;
2. It repeatedly produces a new population, by (a) selecting individuals from the previous population using the fitness values and (b) combining them using the genetic operators *crossover* and *mutation*;
3. It proceeds until a stop condition is met.

In DCT, each GA component (e.g., the fitness function or the crossover operator) is defined as Go interfaces, which enables the replacement for other implementations. The default implementations of these interfaces are specified next.

The default DCT implementation relies on the multi-objective genetic algorithm NSGA-II [25], responsible to implement the selection operator of the GA. When using multi-objective GAs, each individual has a vector of fitness values [24]. To compare two individuals, the concept of *Pareto Dominance* was used: a vector v dominates another vector u if no value v_i is smaller than the value u_i, and at least one v_j is greater than u_j [24] (this applies to optimizations where the goal is to maximize the objective values, if the goal is the opposite, we must invert the comparisons).

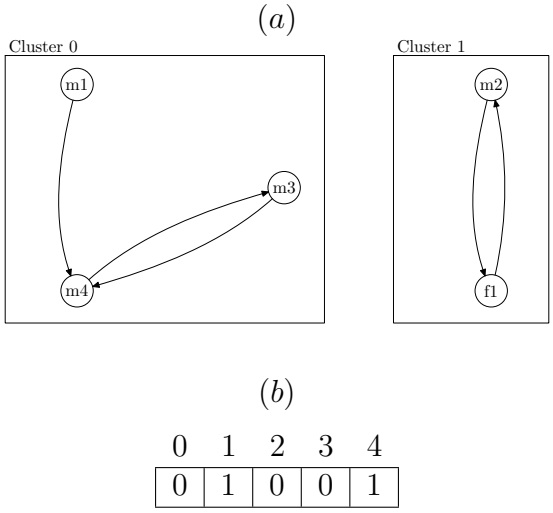

Figure 3.1: Individual representation.

As such, the individuals were represented as a mapping from a module to the cluster it belongs to (typically a module represents a file or a class). Technically, an individual is an array where each position corresponds to a module, and each value corresponds to a cluster. Two different modules belong to the same cluster when they refer to the same value. Figure 3.1-(a) illustrates this representation, showing four modules (m1, m2, m3, m4, f1). All modules belong to the cluster C_0, except for m2 that belongs to the cluster C_1 (together with module f1).

Differently from previous works [26, 27, 23], DCT saves computer's main memory since the array is codified as a binary string (i.e., as a sequence of bits), as we can see in Figure 3.1-(b). The maximum number of clusters is set to $\frac{|V|}{2}$, and each element of the array occupies $\left\lceil \log_2 \frac{|V|-1}{2} \right\rceil$ bits of the binary string—where V is the set of vertices of the MDG. Previous works represent the individual as an array of "integers" [26, 27, 23], which could place a toll on today processors that take 64 bits. For example, if we have a MDG with 10,000 vertices, one element of the array will occupy 13 bits, while the state of the art would occupy 64 bits.

The genetic operators transform the population through successive generations, maintaining the *diversity* and *adaptation* properties from previous generations. In this work, the *one-point crossover operator* was used, which takes two binary strings (parents) and a random index as input, and produces two new binary strings (offspring) by swapping the parents' bits after that index. For example, if we have the parent binary strings $p_1 = 101010$ and $p_2 = 001111$, and an index $i = 1$, the offspring will be $c_1 = 101111$ and $c_2 = 001010$. In addition to that, a *mutation operator* was used that can flip any bit of the individual's binary string at a specified probability. That is, given a mutation

probability p and a binary string $s = b_1 b_2 \ldots b_n$, we produce a random number $0 \leq r_i < 1$ for each bit b_i, flipping b_i in the cases where $r_i < p$. For example, if we have a binary string $s = 10011$, a mutation probability $p = 0.1$, and a sequence of random numbers $r = (0.9, 0.3, 0, 0.6, 0.5)$, the algorithm will produce a *mutant binary string* $s' = 10111$. In DCT was used the Xorshift algorithm in order to generate random numbers; which is a known fast algorithm [28]. To the best of our knowledge, no other SMC tool uses this algorithm.

3.2.2 Fitness Evaluation and Parameters

As mentioned before, the GA was setup to optimize the following five objectives [8]:

- maximize *Modularization Quality (MQ)*;
- maximize intra-edge dependencies;
- minimize inter-edge dependencies;
- maximize number of clusters;
- minimize the difference between the maximum and the minimum number of source-code entities in a cluster.

MQ was defined by Mitchell and Mancoridis [26] as follows:

$$MQ = \sum_{i=1}^{k} CF_i$$

$$CF_i = \begin{cases} \dfrac{\mu_i}{\mu_i + \frac{1}{2} \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \mu_i > 0 \\ 0 & \mu_i = 0. \end{cases}$$

In this equation, k is the number of clusters, μ_i is the number of edges within the i^{th} cluster, and $\varepsilon_{i,j}$ is the number of edges between the i^{th} and the j^{th} clusters.

With relation to the parameters, their values were chosen similarly to Candela et al [15]. As such, given a software module graph $G = (V, E)$, and $n = |V|$, it was defined the parameters population size (PS), maximum number of generations (MG), crossover probability (CP), and mutation probability (MP) are as follows:

- $PS = \begin{cases} 2n & \text{if } n \leq 300 \\ n & \text{if } 300 < n \leq 3000 \\ n/2 & \text{if } 3000 < n \leq 10000 \\ n/4 & \text{if } n > 10000 \end{cases}$

- $MG = \begin{cases} 50n & \text{if } n \leq 300 \\ 20n & \text{if } 300 < n \leq 3000 \\ 5n & \text{if } 3000 < n \leq 10000 \\ n & \text{if } n > 10000 \end{cases}$
- $CP = \begin{cases} 0.8 & \text{if } n \leq 100 \\ 0.8 + 0.2(n - 100)/899 & \text{if } 100 < n < 1000 \\ 1 & \text{if } n \geq 1000 \end{cases}$
- $MP = \frac{16}{100\sqrt{n}}$

In summary, DCT is a *full-fledged* SMC tool written in the Go programming language, which (a) uses the multi-objective NSGA-II algorithm as default implementation, (b) it is also possible to extend DCT to use mono-objective algorithms, (c) employs a simple CLI to ease the execution of experiments, and (d) explores two optimization techniques: binary strings to represent individuals and the Xorshift random number generator algorithm.

3.3 Empirical Assessment Settings

This empirical assessment aims to evaluate the performance of DCT for clustering software systems of different sizes and complexities. Two experiments were conducted. The first compares the performance of DCT against one software clustering tool that runs in a multi-objective mode (Heuristic Design NSGA-II [27]). The second compares the performance of DCT and HD-NSGA-II against two software clustering tools that use a mono-objective strategy (Bunch [2] and Heuristic Design LNS [23]). It is important to point out that, although many research studies on software clustering are available in the literature, most of these publications do not provide tools available for use.

We investigate the following questions in our study:

(a) How does the complexity of the systems affect DCT performance?

(b) How does the DCT performance compare to the performance of multi-objective tools (HD-NSGA-II)?

(c) How does the performance of multi-objective tools (DCT and HD-NSGA-II) compare to the performance of mono-objective tools (Bunch and HD-LNS)?

The multi-objective algorithm of DCT must explore a solution space of exponential complexity. As such, answering the first research question allows us to understand if DCT could be used to cluster software systems in real settings. Answering the second

research question, allows us to understand the performance of DCT in comparison with another NSGA-II implementation. Finally, regarding the last research question, it is still unclear to what extent the use of multi-objective algorithms compromise the performance of publicly available SMC tools. Answering the last research question allows us to better estimate the effect of using a multi-objective algorithm to cluster software systems.

We leveraged three **metrics** to answer these research questions:

1. **TS** is the elapsed time in seconds to cluster each studied system;
2. **MMC** is the Maximum Memory Consumption (in KB) necessary to cluster each studied system; and
3. **MQ** is a metric for estimating the Modularization Quality of the clusters [29, 2].

We ran Bunch and HD-LNS tools with their default settings. On the other hand, HD-NSGA-II was not concluding the process even on small systems. To reduce the number of evaluations, we set the parameters *population size* and *maximum number of generations* to $2n$ and $4n$, respectively, where n is the number of vertices on the MDG. The default values of these parameters are $10p$ and $200p$, where p is the *package count*. The definition of *package* used in HD-NSGA-II corresponds to a `package` in the Java programming language. Furthermore, it was necessary to write a tool to convert MDGs to the proprietary file format used by HD-NSGA-II. Finally, we ported the HD-NSGA-II and HD-LNS implementations to Java libraries and implemented a command line tool to execute both of them.[4]. We hope that this decision could help other researchers to experiment with these tools.

We used the `time` Linux tool to compute the first two metrics. To calculate the MQ metric we considered the outcomes of the clustering tools (Bunch, Heuristic Design, and DCT). We used a dataset of 17 MDGs in our study. These MDGs come from a convenient sample population of open source systems we used in a previous research work [7]. These systems are from different domains and range from small to medium size systems (in terms of lines of code). Moreover, we set 48h as the maximum execution time. Table 3.1 presents some characteristics of these systems.

We executed our experiments using an Intel(R) Xeon(R) E-2124 CPU @ 3.30GHz with 32 GB of RAM, running a Linux Ubuntu distribution (18.04.4 LTS).

[4] `https://github.com/project-draco/cms_runner`

Table 3.1: Projects used in the empirical assessments

System	Modules	Deps.	KLOC	Commits
React Native Framework	190	1006	48	7842
Storm distributed realtime system	388	3249	213	7451
Bigbluebutton web conf. system	497	3661	82	13420
Minecraft Forge	501	3403	72	5498
CAS - Enterprise Single Sign On	513	1718	87	6268
Atmosphere Event Driven Framework	658	3523	41	5748
Druid analytics data store	668	2648	297	7452
Liquibase database source control	716	3981	77	5360
Kill Bill Billing & Payment Platform	767	5422	139	5361
Actor Messaging Platform	768	7452	157	8772
The ownCloud Android App	833	3389	36	5329
Hibernate Object-Relational Mapping	836	2935	628	7302
jOOQ SQL generator	851	4118	133	5022
LanguageTool Style/Grammar Checker	871	1931	75	19121
Bazel build system	965	3813	375	7258
H2O-3 - Machine Learning Platform	1586	27725	143	19336
Jitsi communicator	2557	6742	326	12420

3.4 Results of The Empirical Assessment

In this section we highlight the main findings of our empirical study and provide answers to the research questions we introduced in Section 3.3.

3.4.1 How does the complexity of the systems affect the DCT performance?

To answer this research question, we first considered the complexity of the MDGs (in terms of number of modules) as a model of the log of the elapsed time (TS) to compute the clusters. That is, we expressed this model as $log(TS) \approx Modules$. Considering the adjusted R^2, this model indicates that we can explain 88.87% of the TS variance as an exponential function on the number of modules. This exponential model better explains this variance, in comparison to a quadratic model ($R^2 = 0.73$) and a linear model ($R^2 = 0.38$), as shown in Table 3.2.

Table 3.2: R^2 scores of the **TS** variance models for DCT

model	R^2 score
exponential	88.87%
quadratic	73%
linear	38%

In practice, DCT finds a cluster solution to a small system with 190 modules and 48 KLOC in `00:01:57` (React Native Framework), to a medium size system with 767 modules and 139 KLOC in `00:23:49` (Kill Bill Billing & Payment Platform), and to a large system with 2557 modules and 326 KLOC in `08:30:07` (Jitsi communicator). That is, although we confirmed the exponential cost necessary for DCT to compute the clusters (as a function on the number of modules), we argue that it can still be used in

practice, particularly for small and medium size systems. For larger systems, DCT might find a solution in an interval that goes from a couple of hours to a few days (for extra large systems). So, regarding our first question, we found that:

> Our empirical assessment suggests that we can predict the time necessary for DCT to compute a cluster using an exponential formula on the system's number of modules.

> In the longest scenario of our experiment, DCT found a cluster in 08:30:07 for a system with more than 2500 modules. We argue that this is still a reasonable time for running a SMC reengineering task on a large system using a multi-objective approach.

3.4.2 How does the DCT performance compare to the performance of multi-objective tools (HD-NSGA-II)?

Our goal by answering this question is to understand how DCT compares to another multi-objective SMC tool. Nonetheless, HD-NSGA-II only concluded the execution for seven (out of the 17 projects we considered in our study) within our maximum time threshold (48 hours). Considering only these seven projects, we realized a substantial benefit on the DCT speed-up, ranging from 2.13x to 221x (see Table 3.3).

Table 3.3: Comparison of the elapsed time to generate the clusters (considering the multi-objective tools DCT and HD-NSGA-II).

System	DCT (TS)	HD-NSGA-II (TS)	Speed-up
React Native	117	249	2.13x
Storm	228	12448	54.60x
Big Blue Button	442	36264	82.05x
Minecraft Forge	579	54691	94.46x
CAS Single Sign On	335	39963	119.29x
Atmosphere	970	90954	93.77x
Druid	741	164428	221.90x

Regarding the other metrics (MMC and MQ), DCT improved memory consumption up to 2x (minimum gain of 1.8x — see Table 3.4) and slightly decreased the MQ metrics in six out of the seven cases (see Table 3.5). Specifically, DCT presents a significant reduction on the time necessary to compute the clusters, in comparison to the HD-NSGA-II tool; however, we observed a slight reduction in the quality of the clusters. In the worst case, (Atmosphere project), DCT found a cluster with MQ = 69.64; while HD-NSGA-II found a cluster with MQ = 95.70. Altogether, we answer our second research question as follows:

Our assessment reveals that DCT scales better than HD-NSGA-II, finishing the clusterization process of the Druid tool in 741 seconds (while HD-NSGA-II needed 164 428 seconds). Considering larger projects, HD-NSGA-II did not finish the analysis within our maximum time threshold.

We observed that HD-NSGA-II clusters are slightly better than the clusters produced by DCT

Table 3.4: Comparison of the memory usage generating the clusters (considering the multi-objective tools DCT and HD-NSGA-II).

System	DCT (MB)	HD-NSGA-II (MB)	Improv.
React Native	91	188	2.07
Storm	218	463	2.12
Bigbluebutton	282	511	1.81
Minecraft Forge	320	595	1.86
CAS - Enterprise Single Sign On	300	528	1.76
Atmosphere	416	741	1.78
Druid	396	713	1.80

Table 3.5: Comparison of the clusters' MQ (considering the multi-objective tools DCT and HD-NSGA-II).

System	DCT (MQ)	HD-NSGA-II (MQ)	Improv.
React-native	39.27	33.57	1.17
Storm	60.40	66.60	0.91
Big Blue Button	71.15	79.31	0.90
Minecraft Forge	87.94	92.76	0.95
CAS - Enterprise Single Sign On	92.77	99.07	0.94
Atmosphere	69.64	95.70	0.73
Druid	122.65	128.00	0.96
Average			0.94

3.4.3 How does the performance of multi-objective tools (DCT and HD-NSGA-II) compare to the performance of mono-objective tools (Bunch and HD-LNS)?

The boxplots in Figure 3.2 show the performance of the tools (DCT, HD-NSGA-II, Bunch, and HD-LNS), considering execution time (TS), memory consumption (MMC), and modularization quality (MQ). One could observe that multi-objective SMC implementations requires much more time to compute the clusters. In the worst scenario, DCT requires `00:40:48` while Bunch required `00:00:04`, and HD-LNS requires `00:02:57` in the same comparison.

Regarding memory consumption, the Bunch tool achieved the best performance, with an average memory consumption of ∼126MB; while HD-LNS achieved an average consumption of ∼546MB. Considering the impact on the MQ metric, Figure 3.2 shows a

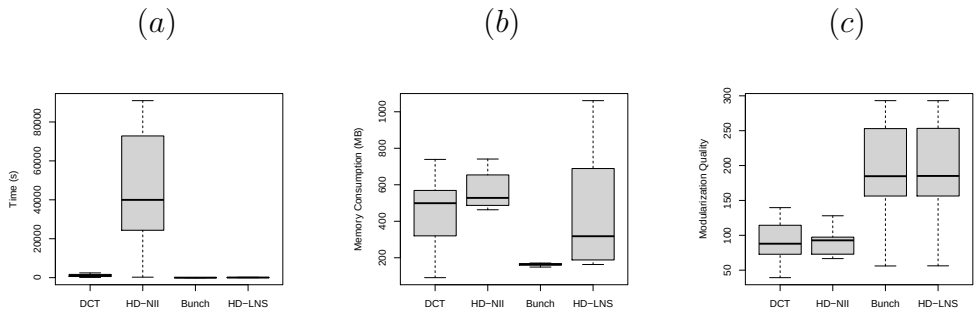

Figure 3.2: Performance comparison of SMC tools. (a) Compares TS, (b) compares MMC, and (c) compares MQ. We removed the outliers in the boxplots.

(median) decreasing of 44% on the modularization quality of the clusters from multi-objective SMC tools. Differently, the mono-objective tools preserve the average quality of the clusters. Altogether, we answer our second research question as follows.

> The use of multi-objective SMC implementations brings a negative impact on both performance and modularization quality, in comparison with the multi-objective tools we used in our research. That is, on average, we found a central tendency of (a) increasing in 400x the time necessary to compute the cluster and (b) decreasing in 44% the modularization quality.

> Comparing to HD-LNS, Bunch brings significant improvements in two metrics (on average): time necessary to compute the clusters (up to 20x) and maximum memory consumption (up to 2x).

Due to the issues raised above, came the opportunity for the integration of yet another mono-objective implementation into the DCT platform, which is the main motivation of this study. In the next chapter this new implementation will be described in more detail.

Chapter 4

The LNS extension for the DCT

This chapter presents the DCT-LNS, the new mono-objective implementation that this study proposed to compose the DCT. This implementation is based on the Large Neighborhood Search (LNS) algorithm brought by Pisinger et al. [17] alongside with other evolutionary algorithm in the Handbook of metaheuristics [21]. It is important to mention that this algorithm was first used in the context of SMC tools by Monçores et al. [23], which served as the knowledge base for the implementation that was carried out. The tool developed by Monçores et al. [23] in their research is the Heuristic Design LNS (HD-LNS). This tool will be part of the empirical assessment explained in more details further on.

4.1 Design Principles

DCT-LNS is also a Command Line Interface (CLI) tool, that reads a *Module Dependency Graph (MDG)* [2] from the standard input and writes a clustered graph represented as a DOT[1] file in the standard output. It was implemented in Go[2] programming language as well, and is publicly available.[3] A typical invocation of the tool looks like this:

```
$ ./lns < software.mdg > software.dot
```

This new implementation tries to follow the DCT design guidelines that was mentioned earlier. The choice regarding the programming language still relies on the fact that Go programs are compiled instead of being interpreted. Moreover, Go provides an easy interface to deal with parallel programming. Such features can significantly increase the runtime speed. Besides that, it is important to point out that the two optimization techniques implemented by the DCT: (a) binary strings to represent individuals and (b)

[1] `https://graphviz.org/doc/info/lang.html`
[2] `https://golang.org`
[3] `https://github.com/project-draco/DCT-mono-obj`

Xorshift random number generator algorithm, have not yet been explored by the DCT-LNS.

The DCT-LNS used the definition of the SMC problem as a mono-objective optimization problem. The same set of objects recommended by Praditwong et al. [8] used by the DCT was also used for DCT-LNS. Furthermore, in DCT-LNS, the MQ metric was used as the single objective to be reached. The input is a MDG represented by a graph $G = (V, E)$ from a set of modules V and a set of dependencies $E \subseteq V \times V$; and the output is a unique solution. This output solution is a partition of a MDG that corresponds to a set of clusters with the best MQ found.

4.2 Large Neighborhood Search (LNS) Algorithm

A typical LNS executes as follows:

(a) It first generates an initial cluster set, randomly or not (See "**Step 1**" in Figure 4.1);

(b) It repeatedly produces a new cluster set, by: (a) removing some of the vertices from its cluster (b) placing the removed vertices back into the cluster set, considering the cluster that returns the best MQ (See "**Step 2**" in Figure 4.1);

(c) It proceeds until a stop condition is met, for example a threshold of no improvement iterations (See "**Step 3**" in Figure 4.1);

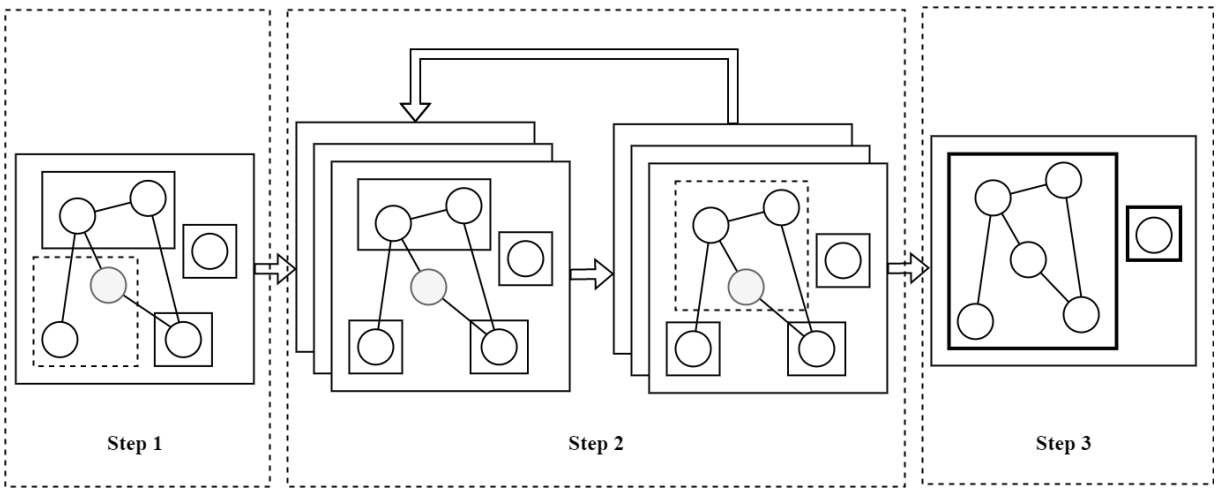

Figure 4.1: Illustration of a typical LNS execution

The DCT-LNS implementation of the LNS algorithm follows the best configurations found on the empirical study conducted by Monçores et al. [23], as follows:

1. The initial cluster set generation could be (i) randomly generated or (ii) agglomerative, the study [23] found out that the agglomerative method, despite of being slightly slower produced solutions with better MQs. The process of the agglomerative method starts with each vertex belonging to its own cluster, then we compute the MQ for each possible merge between 2 of the remaining clusters. With that, the merge with the best MQ is done. The process continues until only one cluster remains. At the end, the set of clusters with the best MQ that was visited in the process is returned.

2. The removal process of the vertices from its cluster could be (i) by choosing random vertices to be removed, (ii) choosing a random cluster and removing up to six vertices out of it or (iii) choosing based on the difference between the cluster of a random vertex in the current solution and in the best solution found so far. The study [23] found out that choosing random vertices to be removed was the best method, specifically with a 10% removal rate.

3. The placing back process of the vertices could be in the various ways explained in the study [23]. The study found out that the best method is a joint between two methods described by the author in the study. However the DCT-LNS use only one of the methods proposed, which is the method with lower complexity. This chosen method for DCT-LNS is a greedy method that chooses a random vertex from the set of removed vertices and place it on the cluster that provides the best MQ, also being possible the creation of a new cluster in this process, this continues until there is no more vertices on the removed vertices set.

4. The stop condition is based on a no-improvement iterations threshold. The study [23] found out that the best threshold to use was 1000, so that is the threshold that the DCT-LNS adopted.

In summary, DCT-LNS is a SMC tool to be integrated into the DCT, written in the Go programming language, which (a) uses the mono-objective LNS algorithm, (b) employs a simple CLI to ease the execution of experiments, and (c) follows the algorithm configuration proposed by a previous LNS applied to SMC study, Monçores et al. [23].

4.3 Empirical Assessment Settings

This empirical assessment aims to evaluate the performance of DCT-LNS for clustering software systems of different sizes and complexities. Two experiments were conducted.

The first compares the performance of DCT-LNS against one software clustering tool that runs in mono-objective mode Heuristic Design LNS (HD-LNS) [23].

We investigate the following questions in our study:

(a) How does the complexity of the systems affect DCT-LNS performance?

(b) How does the DCT-LNS performance compare to the performance of a mono-objective tool with same algorithm HD-LNS?

(c) How does the performance of DCT-LNS compare to the performance of other mono-objective tools: Bunch and the old DCT mono-obj algorithm?

The implementation of the LNS algorithm made in DCT-LNS must explore a solution space of cubic complexity. As such, answering the question (a) allows us to understand if the runtime of DCT-LNS could be estimated when used to cluster software systems in real settings. Answering question (b) allows us to understand the performance of the DCT-LNS in comparison with another LNS implementation. Finally, the answer to the question (c) will help us to understand the performance of the DCT-LNS in comparison with other tools available.

The metrics taken into consideration were the same as in the DCT empirical assessment:

1. **TS** is the elapsed time in seconds to cluster each studied system;
2. **MMC** is the Maximum Memory Consumption (in KB) necessary to cluster each studied system; and
3. **MQ** is a metric for estimating the Modularization Quality of the clusters [29, 2].

Bunch was executed from the CLI interface that the authors have provided [2]. HD-LNS was also executed from the CLI, but it was necessary to port the Java libraries and implement a CLI tool to make this possible, as mentioned in Chapter 3. The projects `time` Linux tool was used to compute **TS** and **MMC**. The **MQ** metric is printed by the tools on the standard output. A data set of 16 MDGs was used in the study. These MDGs come from a convenient sample population of open source systems we used in a previous research work [7], as mentioned in Chapter 3. These systems are from different domains and range from small to medium size systems (in terms of lines of code). Moreover, we set 2h as the maximum execution time. Table 4.1 presents some characteristics of these systems.

All experiments are available online[4] for replication within a Docker[5] environment. We executed our experiments using a server with an Intel(R) Xeon(R) CPU @ 2.30GHz with 32 GB of RAM, running a Linux Debian distribution (Debian GNU/Linux 10 (buster)), and for each tool and each MDG the execution was repeated five times and the metrics means were collected.

Table 4.1: Projects used in the empirical assessments

System	Modules	Deps.	KLOC	Commits
React Native Framework	190	1006	48	7842
Storm distributed realtime system	388	3249	213	7451
Bigbluebutton web conf. system	497	3661	82	13420
Minecraft Forge	501	3403	72	5498
CAS - Enterprise Single Sign On	513	1718	87	6268
Atmosphere Event Driven Framework	658	3523	41	5748
Druid analytics data store	668	2648	297	7452
Liquibase database source control	716	3981	77	5360
Kill Bill Billing & Payment Platform	767	5422	139	5361
Actor Messaging Platform	768	7452	157	8772
The ownCloud Android App	833	3389	36	5329
Hibernate Object-Relational Mapping	836	2935	628	7302
jOOQ SQL generator	851	4118	133	5022
LanguageTool Style/Grammar Checker	871	1931	75	19121
Bazel build system	965	3813	375	7258
Jitsi communicator	2557	6742	326	12420

4.4 Results of The Empirical Assessment

In this section we highlight the main findings of our empirical study and provide answers to the research questions we introduced in Section 4.3.

4.4.1 How does the complexity of the systems affect DCT-LNS performance?

To answer this research question, we considered the complexity of the MDGs and elapsed time (TS) to compute the clusters. Then, the following models were tested: linear, quadratic, cubic and exponential. For each model the R^2 score was calculated. The cubic model was the expected best fit, and indeed was the best fit with a rate equal to 99,99%. The quadratic model was the second best (R^2 = 99,89%), followed by the linear model (R^2 = 86,83%), finally, the exponential model presented the worst fit ($R^2 = 83,46\%$). (See Table 4.2).

In practice, DCT-LNS finds a cluster solution to a small system with 190 modules and 48 KLOC in `00:00:00.59` (React Native Framework), to a medium size system with 767 modules and 139 KLOC in `00:00:35.64` (Kill Bill Billing & Payment

[4] `https://github.com/project-draco/SCM_performance_Analysis`
[5] `https://www.docker.com/`

Table 4.2: R^2 scores of the **TS** variance models for DCT-LNS

model	R^2 score
exponential	83,46%
cubic	99,99%
quadratic	99,89%
linear	86,83%

PLATFORM), and to a large system with 2557 modules and 326 KLOC in `00:33:11.23` (JITSI COMMUNICATOR). That is, although we confirmed the cubic cost necessary for DCT-LNS to compute the clusters (as a function on the number of modules), we argue that it can still be used in practice, and it takes less than an hour even for large systems. So, regarding our first question, we found that:

> Our empirical assessment suggests that we can predict the time necessary for the DCT-LNS to compute a cluster using an cubic formula on the system's number of modules.

> In the longest scenario of our experiment, DCT-LNS found a cluster in `00:33:11.23` for a system with more than 2500 modules. We argue that this is a great time for running a SMC reengineering task on a large system using this mono-objective approach.

4.4.2 How does the DCT-LNS performance compares to the performance of mono-objective tool with same algorithm HD-LNS?

Our goal by answering this question is to understand how DCT-LNS compares to another mono-objective LNS SMC tool. Considering the 16 systems in the experiment, it was possible to see a benefit on the DCT-LNS speed-up for all systems, ranging from 1.44x to 5.32x (see Table 4.3). Regarding the other metrics (MMC and MQ), DCT-LNS improved memory consumption up to 110.1x (minimum gain of 6.87x — see Table 4.4) and for the MQ metric both tools performed equally for all the systems (see Table 4.4). Altogether, we answer our second research question as follows:

> Our assessment reveals that DCT-LNS, in comparision with HD-LNS, is optimized regarding the elapsed time (TS) and the memory usage (MEM). However, in terms of Modularization Quality (MQ), there was no difference between the performance of the tools.

Table 4.3: Comparison of the elapsed time to generate the clusters (considering the mono-objective LNS tools DCT-LNS and HD-LNS).

System	DCT-LNS (TS)	HD-LNS (TS)	Speed-up
React Native Framework	0.59	1.16	1.97x
Storm distributed realtime system	4.82	14.28	2.96x
Bigbluebutton web conf. system	9.6	18.62	1.94x
Minecraft Forge	13.91	35.96	2.59x
CAS - Enterprise Single Sign On	8.61	19.88	2.31x
Atmosphere Event Driven Framework	28.02	97.84	3.49x
Druid analytics data store	19.14	54.89	2.87x
Liquibase database source control	29.28	47.68	1.63x
Kill Bill Billing & Payment Platform	36.01	96.97	2.69x
Actor Messaging Platform	39.38	123.83	3.14x
The ownCloud Android App	38.71	205.89	5.32x
Hibernate Object-Relational Mapping	39.78	76.14	1.91x
jOOQ SQL generator	41.29	154.6	3.74x
LanguageTool Style/Grammar Checker	58.96	85.06	1.44x
Bazel build system	80.34	281.44	3.5x
Jitsi communicator	1993.61	3711.5	1.86x

Table 4.4: Comparison of the MQ and (MEM) metrics (considering the mono-objective LNS tools DCT-LNS and HD-LNS).

System	DCT-LNS (MQ)	HD-LNS (MQ)	Improv.	DCT-LNS (MEM)	HD-LNS (MEM)	Improv.
React Native	56.22	56.22	1.0	14995.0	103041.0	6.87
Storm.	115.67	115.7	1.0	16685.0	264452.0	15.85
Bigbluebutton.	119.6	119.6	1.0	18776.0	170310.0	9.07
Minecraft Forge	154.59	154.61	1.0	17758.0	382812.0	21.56
CAS.	156.33	156.33	1.0	17711.0	180866.0	10.21
Atmosphere.	185.08	185.1	1.0	18794.0	183831.0	9.78
Druid analytics	193.37	192.66	1.0	18147.0	411647.0	22.68
Liquibase.	183.36	183.36	1.0	19720.0	185210.0	9.39
Kill Bill Billing.	214.66	214.66	1.0	21288.0	648191.0	30.45
Actor Platform	182.25	182.27	1.0	23016.0	188594.0	8.19
OwnCloud Android	171.63	171.93	1.0	20255.0	427341.0	21.1
Hibernate Object.	253.25	253.33	1.0	18371.0	1197090.0	65.16
jOOQ SQL generator	213.75	213.75	1.0	19324.0	198582.0	10.28
LanguageTool.	293.3	293.02	1.0	17163.0	1222302.0	71.22
Bazel build system	279.15	279.23	1.0	20258.0	1200883.0	59.28
Jitsi communicator	745.43	745.95	1.0	23040.0	2536610.0	110.1

4.4.3 How does the performance of DCT-LNS compare to the performance Other mono-objective tools: Bunch and the old DCT mono-obj algorithm?

The boxplots in Figure 4.2 show the performance of the tools (DCT-LNS, Bunch, and Other DCT mono-objective implementation), considering the Log of the execution time (Log TS) for better visualization, memory consumption (MMC), and Modularization Quality (MQ). For this experiment, only 15 MDGs were considered out of the original 16, because the other DCT mono-objective implementation could not complete the clustering

for the *Jitsi communicator* system before the time limit that was setup at the begging of the empirical assessment. One could observe that the Other DCT mono-objective implementation requires much more time to compute the clusters. In the worst scenario, DCT-LNS requires `00:01:20.34` while Bunch required `00:00:03.34`, and HD-LNS requires `00:36:11.4` on the same comparison.

Regarding memory consumption, the DCT-LNS tool achieved the best performance in all the cases (See Figure 4.2). Considering the impact on the MQ metric, Figure 4.2 shows that DCT-LNS and Bunch perform noticeably better than the Other DCT mono-objective implementation, and that DCT-LNS is slightly better than Bunch in all the systems. Altogether, we answer our second research question as follows.

> The DCT-LNS has remarkable results regarding the MQ and MEM metric. However, it is worse than Bunch in terms of elapsed time (TS), although it still executes in a feasible time.

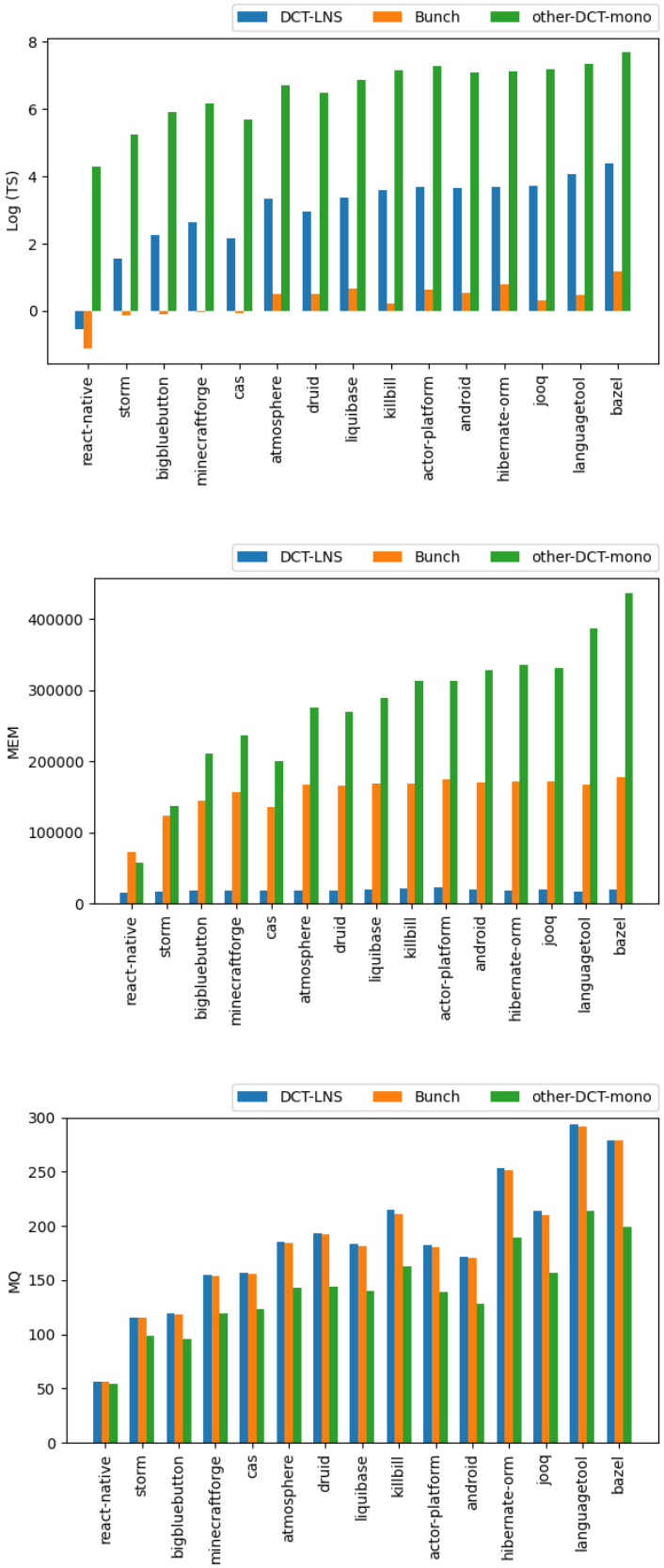

Figure 4.2: Performance comparison of SMC tools. (a) Compares TS, (b) compares MMC, and (c) compares MQ.

Chapter 5

Final Remarks

With this present work it was possible to enhance the knowledge about automatic techniques for software architecture recovery, more specifically, with respect to SMC tools. From the empirical assessments that were performed it was possible to deduce that the DCT tools, both mono-objective and multi-objective, are tools with good performance. The advantage of the DCT tools is that they are available online for the academic and even business community.

More in detail, the empirical assessments presented the DCT tools as scalable, as the integrated mono-objective tool showed a cubic time complexity and the multi-objective tool showed an exponential time complexity. This exponential time complexity of the multi-objective tool was a concern at first, but the tests using real systems validated that the usage of this tool was still possible within a few hours for large systems and within a few days for extra large systems, considering that the size of the system was based on the number of modules. Moreover, the empirical assessments also presented a better performance of the multi-objective tool in comparison with other available multi-objective tool in terms of the execution time and the memory usage. The integrated mono-objective tool showed a good performance comparison as well, since it had a higher efficiency than the other available tool that uses the same algorithm (LNS) in terms of execution time and memory usage. On both DCT tools, the MQ metric had an equal or slightly worse results than the other tools that were being compared. With that said, it's possible to conclude that the initial purpose of this study has been accomplished, which was evaluating and extending the DCT with good results.

Bibliography

[1] Dahiya, S. S., J. K. Chhabra, and S. Kumar: *Use of genetic algorithm for software maintainability metrics' conditioning*. In *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pages 87–92, 2007. 1

[2] Mitchell, B. S. and S. Mancoridis: *On the automatic modularization of software systems using the bunch tool*. IEEE Transactions on Software Engineering, 32(3):193–208, March 2006, ISSN 1939-3520. 1, 6, 10, 11, 16, 19

[3] Oliveira, Marcos César de, Rodrigo Bonifácio, Guilherme N. Ramos, and Márcio Ribeiro: *Unveiling and reasoning about co-change dependencies*. In Fuentes, Lidia, Don S. Batory, and Krzysztof Czarnecki (editors): *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016*, pages 25–36. ACM, 2016. https://doi.org/10.1145/2889443.2889450. 1, 2

[4] Lethbridge, T. C., J. Singer, and A. Forward: *How software engineers use documentation: the state of the practice*. IEEE Software, 20(6):35–39, 2003. 1

[5] Garlan, David: *Software architecture: a roadmap*. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101, 2000. 1

[6] Schmitt Laser, Marcelo, Nenad Medvidovic, Duc Minh Le, and Joshua Garcia: *Arcade: an extensible workbench for architecture recovery, change, and decay evaluation*. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1546–1550, 2020. 2, 5

[7] Oliveira, Marcos César de, Davi Freitas, Rodrigo Bonifácio, Gustavo Pinto, and David Lo: *Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings*. Journal of Systems and Software, 158:110420, 2019, ISSN 0164-1212. http://www.sciencedirect.com/science/article/pii/S0164121219301943. 2, 5, 11, 19

[8] Praditwong, Kata, Mark Harman, and Xin Yao: *Software module clustering as a multi-objective search problem*. IEEE Trans. Softw. Eng., 37(2):264–282, March 2011, ISSN 0098-5589. 2, 4, 7, 9, 17

[9] Huang, Jinhuang, Jing Liu, and Xin Yao: *A multi-agent evolutionary algorithm for software module clustering problems*. Soft Computing, 21(12):3415–3428, 2017. 2

[10] Chhabra, Jitender Kumar *et al.*: *Many-objective artificial bee colony algorithm for large-scale software module clustering problem.* Soft Computing, 22(19):6341–6361, 2018. 2

[11] Prajapati, Amarjeet and Jitender Kumar Chhabra: *Madhs: Many-objective discrete harmony search to improve existing package design.* Computational Intelligence, 35(1):98–123, 2019. 2

[12] Sun, Jiaze and Beilei Ling: *Software module clustering algorithm using probability selection.* Wuhan University Journal of Natural Sciences, 23(2):93–102, 2018. 2

[13] Bishnoi, M. and P. Singh: *Modularizing software systems using pso optimized hierarchical clustering.* In *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, pages 659–664, 2016. 2

[14] Singh, V.: *Software module clustering using metaheuristic search techniques: A survey.* In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 2764–2767, 2016. 2

[15] Candela, Ivan, Gabriele Bavota, Barbara Russo, and Rocco Oliveto: *Using cohesion and coupling for software remodularization: Is it enough?* ACM Trans. Softw. Eng. Methodol., 25(3):24:1–24:28, June 2016, ISSN 1049-331X. `http://doi.acm.org/10.1145/2928268`. 2, 4, 9

[16] Tarchetti, Ana Paula M, Luís Amaral, Marcos C Oliveira, Rodrigo Bonifácio, Gustavo Pinto, and David Lo: *Dct: An scalable multi-objective module clustering tool.* In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 171–176. IEEE, 2020. 2

[17] Pisinger, David and Stefan Ropke: *Large neighborhood search.* In *Handbook of metaheuristics*, pages 399–419. Springer, 2010. 2, 16

[18] Anquetil, Nicolas, Cédric Fourrier, and Timothy C. Lethbridge: *Experiments with clustering as a software remodularization method.* In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 235–, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0303-9. 4

[19] Maqbool, Onaiza and Haroon Babri: *Hierarchical clustering for software architecture recovery.* IEEE Trans. Softw. Eng., 33(11):759–780, November 2007, ISSN 0098-5589. 4

[20] Pinto, Alexandre Fernandes, Adriana Cesário de Faria Alvim, and Márcio de Oliveira Barros: *Ils for the software module clustering problem.* XLVI Simpósio Brasileiro de Pesquisa Operacional. Salvador:[sn], pages 1972–1983, 2014. 4

[21] Glover, Fred W and Gary A Kochenberger: *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006. 4, 16

[22] Oliveira Barros, Márcio de: *Evaluating modularization quality as an extra objective in multiobjective software module clustering.* In *International Symposium on Search Based Software Engineering*, pages 267–267. Springer, 2011. 5

[23] Monçores, Marlon C, Adriana CF Alvim, and Márcio O Barros: *Large neighborhood search applied to the software module clustering problem.* Computers & Operations Research, 91:92–111, 2018. 5, 8, 10, 16, 17, 18, 19

[24] Goldberg, David E: *E. 1989. genetic algorithms in search, optimization, and machine learning.* Reading: Addison-Wesley, 1990. 7

[25] Deb, K., A. Pratap, S. Agarwal, and T. Meyarivan: *A fast and elitist multiobjective genetic algorithm: Nsga-ii.* IEEE Transactions on Evolutionary Computation, 6(2):182–197, Apr 2002, ISSN 1089-778X. 7

[26] Mitchell, Brian S. and Spiros Mancoridis: *On the automatic modularization of software systems using the bunch tool.* IEEE Trans. Softw. Eng., 32(3):193–208, March 2006, ISSN 0098-5589. 8, 9

[27] Barros, Marcio de Oliveira: *An analysis of the effects of composite objectives in multiobjective software module clustering.* In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1205–1212, 2012. 8, 10

[28] Marsaglia, George: *Xorshift rngs.* Journal of Statistical Software, Articles, 8(14):1–6, 2003, ISSN 1548-7660. `https://www.jstatsoft.org/v008/i14`. 9

[29] Mancoridis, Spiros, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner: *Bunch: A clustering tool for the recovery and maintenance of software system structures.* In *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*, page 50. IEEE Computer Society, 1999. `https://doi.org/10.1109/ICSM.1999.792498`. 11, 19