Computers and Operations Research 91 (2018) 92–111

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor

Large Neighborhood Search applied to the Software Module Clustering problem

Marlon C. Monçores[1,*], Adriana C.F. Alvim[2], Márcio O. Barros[2]

UNIRIO - Universidade Federal do Estado do Rio de Janeiro, Centro de Ciências Exatas e Tecnologia. Av. Pasteur, 458, Urca, Rio de Janeiro, RJ 22290-240, Brasil

ARTICLE INFO

Article history:
Received 3 December 2016
Revised 8 October 2017
Accepted 9 October 2017
Available online 10 October 2017

Keywords:
SBSE
Software Module Clustering
Large Neighborhood Search

ABSTRACT

The Software Module Clustering problem seeks to distribute the modules comprising a software system into clusters to maximize cluster cohesion and minimize coupling between clusters. Metaheuristics based on local search have been successfully applied to find good solutions for this problem, outperforming more-complex, population-based heuristics. In this paper, we present a heuristic based on Large Neighborhood Search to address the Software Module Clustering problem. We also perform what is to our knowledge the largest experimental study addressing this problem to date, involving 124 instances of varying size and complexity and comparing our proposed algorithm to the heuristic that has found the best results for the problem so far. Our proposed algorithm outperformed the state-of-the-art heuristic on 93 out of 124 instances with 95% confidence level. We also report new upper bounds on the MQ value for 44 instances and evaluated the relative goodness of the solutions obtained by our proposed algorithm for 89 instances. Considering the 77 instances for which optimal solutions are proven, the proposed algorithm found the optimal solution for 30 instances (39%). Additionally, thirteen developers participate in a study focused on the distribution of the modules comprising a software project into clusters. We used JodaMoney as our study object and we compared the characteristics of the solutions generated by its authors, the best solution generated by LNS_SMC and the solution generated by each of the 13 subjects. LNS_SMC solution performs better than inexperienced subjects ones in issue resolution prediction but performs worse than the solutions proposed by experienced subjects and also the author's solution. For the prediction of concomitant changes, the LNS_SMC solution was outperformed by both subjects and authors solutions.

1. Introduction

Software systems are created by writing text files called source code *modules*. Software systems are usually comprised of many modules, which are compiled together to create the software's executable files. A module may interact with other modules to provide the functions that the software offers to its users. Such interaction happens if, for instance, module A calls a function, accesses a variable, refers to a constant, or uses any programming language construct that is defined in module B. Due to these references, we say that module A *depends on* module B, in the sense that if module B is missing, module A won't work properly.

Software Module Clustering (SMC) is the problem of arranging software modules into larger, container-like structures called *clusters*. Modules need to be distributed into clusters because large software systems may involve hundreds or thousands of modules and there must be some higher-level organization to help developers find the modules responsible for a given task performed by the system. In fact, it has been suggested that proper distribution of modules into clusters aids in identifying modules responsible for a given functionality (Larman, 2002), provides easier navigation among software parts (Gibbs et al., 1990), and enhances comprehension (Larman, 2002; McConnell, 2004). Therefore, proper module distribution makes software systems easier to develop and maintain. On the other hand, experimental studies have shown a strong correlation between poor module distributions and the presence of faults in software systems (Briand et al., 1999).

* Corresponding author.
E-mail addresses: marlon@moncores.com (M.C. Monçores), adriana@uniriotec.br (A.C.F. Alvim), marcio.barros@uniriotec.br (M.O. Barros).

[1] Student at the Postgraduate Information Systems Program, Federal University of the State of Rio de Janeiro.

[2] Fellow at the Postgraduate Information Systems Program, Federal University of the State of Rio de Janeiro.

https://doi.org/10.1016/j.cor.2017.10.004
0305-0548/© 2017 Elsevier Ltd. All rights reserved.

The quality of a module distribution is usually addressed by two metrics: cohesion and coupling (Yourdon and Constantine, 1979). *Cohesion* measures the strength of dependencies among modules in the same cluster, while *coupling* refers to the strength of dependencies among modules in different clusters (Praditwong, 2011). *High cohesion* is a desirable property, as developers aim to create clusters enclosing modules that depend on each other in hopes that such dependencies are due to working together to implement one feature of the software. *Low coupling* is also a welcome property, as modules from a given cluster should require services from as few modules in other clusters as possible to perform their duties. Thus, the SMC problem can be restated as finding module distributions where modules that have many dependencies among each other are placed in the same cluster (high cohesion), while modules that share little or no dependencies are placed in different clusters (low coupling). Thus, the SMC problem can be considered a combinatorial optimization problem for which, between all possible distribution from modules to clusters, we aim to select one that maximize a given quality metric.

Mancoridis et al. (1998) were the first authors to use a search-based approach to find good solutions for the SMC problem. They propose a measure used to calculate the quality of a module distribution according to a balance between cohesion and coupling. This objective function, called MQ (*Modularization Quality*), guides a local search algorithm that traverses the solution space in search of high-quality module distribution. Several works have used MQ to evaluate clustering modules after the seminal work of Mancoridis et al. (1998). Many of these works used MQ as a mono-objective function (Barros, 2013; Doval et al., 1999; Köhler et al., 2013; Kramer et al., 2016; Mahdavi et al., 2003; Mancoridis et al., 1999; Mitchell, 2002; Pinto, 2014; Praditwong, 2011). Other works have used multi-objective optimization techniques considering MQ along with other objectives, including compliance with version control history, semantic and structural similarity, and so on (Kumari and Srinivas, 2013; Mkaouer et al., 2014; Praditwong et al., 2011). In this paper we perform all optimizations using MQ as fitness function, we analyze the properties of the solutions found by optimization and perform an experimental study with human subjects to address whether solutions proposed by these software developers would show similar properties to the one presented by optimization.

Local search algorithms seem well-suited for the SMC problem. The Hill Climbing search used by Mancoridis et al. (1998), for instance, has not been outperformed by genetic algorithms (Doval et al., 1999), and has only been consistently outperformed by a multi-objective genetic algorithm (Praditwong, 2011), though with significant cost in terms of processing time. More-complex metaheuristic-based algorithms evolving a single solution (instead of a population) (Lourenço et al., 2010; Pisinger and Ropke, 2010) may be applied to optimization problems to improve the ability of simple heuristics (such as local search) to avoid local optima while maintaining their simplicity and performance. One of those methods, the *Iterated Local Search* metaheuristic (ILS), was recently applied to the SMC problem and outperformed genetic algorithms and simpler local search algorithms (Pinto et al., 2014; Pinto, 2014).

In this work, we describe a heuristic based on Large Neighborhood Search (LNS) (Pisinger and Ropke, 2010), originally outlined in Monçores et al. (2015), to find good solutions for the SMC problem. The LNS metaheuristic uses the concept of destroying and rebuilding parts of a solution in order to evolve and improve this solution. We used the MQ objective function to guide the search and evaluate the quality of a given solution. We report the results of experimental studies involving up to 124 instances, the largest experiment addressing the SMC problem to date. The usual studies in the SCM literature use up to 40 instances, the largest of which have about 470 modules; meanwhile, the largest instance in our dataset has 1161 modules. This dataset contains both instances generated by us from open-source projects (28), as well as instances made available by researchers who have studied the SMC problem (96).

To evaluate the performance of our LNS-based heuristic, we compare it with the solutions produced by ILS. ILS was chosen because it found the best results for several instances of the SMC problem when compared with other non-exact, mono-objective algorithms addressing the same problem (Pinto et al., 2014). We coded the ILS proposed by Pinto et al. (2014) to compare its results on the same instances and running the same number of optimization cycles as LNS. The results show that LNS outperforms ILS for 93 out of 124 instances with 95% confidence, most frequently on the largest instances in our dataset. In addition, LNS requires reasonable processing time, on the same order of magnitude as local search algorithms, and orders of magnitude faster than population-based algorithms, such as genetic algorithms. Finally, we evaluate the relative goodness of solutions obtained by LNS with respect to optimal solutions and upper bounds on the MQ value, provided by Kramer et al. (2016) and Kramer (2017), for 89 instances.

Our evaluation is based on the MQ fitness function, despite the fact that some authors have found that using MQ as the driver for optimization may lead to solutions that are not accepted by developers (Bavota et al., 2012; Glorie et al., 2009; de Oliveira Barros et al., 2015). To compensate for this limitation, we designed and executed a human study in an effort to validate the results found by our algorithm. We highlight the following four major contributions of this paper:

- We propose and evaluate the use of the LNS metaheuristic to find solutions for the SMC problem, showing that this algorithm outperforms other mono-objective algorithms applied to the same problem while keeping the simplicity and high performance that characterize local search algorithms;
- We present the largest, reusable experiment addressing the SMC problem to date. The experiment consists of more than 120 instances, some having more than 1000 modules. These instances, along with our results, are available for other researchers interested in investigating solutions to the SMC problem;
- We present the design and results of a human-based study on the arrangement of classes into clusters for a small software project, comparing its results to the distribution used by the authors of such software and the clustering found by optimization. We found out that if the software structure resembles the roles played by its classes, then the SMC problem tends to lead to distributions which are relevant to developers;
- We report new upper bounds on the MQ value for 44 instances, provided by Kramer (2017).

This paper is divided into eight sections, beginning with this introduction. Next, we provide background information required to understand our proposed solution and the evaluation that was carried out. The third section describes the proposed LNS method. The fourth section presents a set of experimental studies that were designed to find the best parameter values of the LNS method to address the SMC problem. Section 5 presents the evaluation of the LNS method. Section 6 present a human-based study that addresses some limitations presented by the clusterings found by our algorithms. Section 7 presents related work and, finally, Section 8 briefly concludes the paper. The appendix presents detailed computational results, including new upper bounds on the MQ value provided by Kramer (2017).

2. Background

This section provides background information for the reader to understand the SMC problem, the measures used to evaluate the quality of a clustering, a preprocessing algorithm that may reduce the size of an instance of the problem, and an iterated local search procedure applied to the problem.

2.1. SMC as a graph partition problem

The SMC problem can be modeled as a *graph partition problem*, which is NP-Hard (Garey and Johnson, 1979), disregarding idiosyncrasies related to software or programming language issues. Usually, the dependencies among modules comprising a software system are modeled as a graph called the Module Dependency Graph (MDG) (Mancoridis et al., 1998). This graph is the input to the SMC problem, which can then be seen as a graph partition problem. The MDG is a directed graph where the *modules* are represented by *nodes, dependencies* between modules are represented by *edges* between these nodes, and each *cluster* is represented by a *partition* of the graph.

Taking an MDG as input, the SMC problem can be formally defined as follows (Köhler et al., 2013). Let $G=(V,E)$ be an MDG, where V is the set of n nodes (modules) and $E=\{(u,v)|u,v\in V\}$ is the set of dependencies between those modules. The SMC problem consists in finding a partitioning of G into k clusters C_1, C_2, ... ,C_k, that split V into V_1, V_2, ... ,V_k components having $\cup_{i=1}^{k}V_i=V$; $V_i\cap V_j=\emptyset$ for $1\le i, j\le k$; $i\ne j$ and $V_i\ne\emptyset$ for $1\le i\le k$. Each node in V_i is associated to cluster C_i for $1\le i\le k$.

Each edge in an MDG may have an associated *weight*, representing the relevance of some dependency which exists between the two modules. In case a *weight* is missing, it is assumed to be equal to 1. The assignment of weights can depend on various kinds of dependencies that may exist between two modules (inheritance, master-detail, association, and so on). Developers may assign weights to various types of dependencies according to their perspectives on the relevance of each kind of dependency. The overall weight for the edge between two modules is calculated by adding all the dependency weights between these modules (Mitchell, 2002). Despite using some instances with weighted dependencies in this work, the definition of how these weights are assigned to different types of dependency is outside the scope of this work.

As any non-trivial set of modules might allow more than one partitioning satisfying the properties stated above, there must exist one or more measures to calculate the *quality* of a partitioning. If such measures are available, an *optimization procedure* can be applied to the problem of finding high-quality solutions. Mancoridis et al. (1998) define three functions for calculating the quality of a clustering. Those functions have many similarities and are collectively called *Modularization Quality* (or simply MQ). MQ represents a trade-off between inter-connectivity and intra-connectivity in an MDG representing a software project. *Intra-connectivity* is a measure of the connection between modules placed in the same cluster, and *inter-connectivity* is a measure of the connection between modules placed in different clusters. The MQ function rewards clusters having high intra-connectivity (high cohesion) and low inter-connectivity (low coupling). This trade-off is established by subtracting the average inter-connectivity from the average intra-connectivity (Mitchell, 2002). Many authors (Barros, 2012; Doval et al., 1999; Köhler et al., 2013; Kumari and Srinivas, 2013; Mahdavi et al., 2003; Mancoridis et al., 1999; 1998; Mitchell, 2002; Praditwong, 2011; Praditwong et al., 2011; Semaan and Ochi, 2007) use MQ as the objective function to address the SMC problem.

For each cluster C_i, $1 \le i \le k$, the *cluster factor* (CF_i) of cluster C_i is presented in Eq. (1) (Mitchell, 2002). Given an MDG partitioned into k clusters, MQ is calculated by summing the cluster factor (CF) for each cluster. Let μ_i be the sum of weights of the internal edges in cluster C_i and $\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ be the sum of weights of the edges that connect clusters C_i and C_j in each direction, respectively. We assume that all edges without explicitly defined weights have weight 1. We also assume that self-dependencies do not exist, as they should be interpreted as internal dependencies. Eq. (1) shows the calculations required to compute the MQ function for a given partitioning.

$$MQ=\sum_{i=1}^{k}CF_i$$
$$CF_i=\begin{cases}0 & \text{if } \mu_i=0\\ \frac{2\mu_i}{2\mu_i+\sum_{j=1,j\ne i}^{k}(\varepsilon_{i,j}+\varepsilon_{j,i})} & \text{otherwise}\end{cases}\qquad(1)$$

Considering MQ as the quality measure of a partitioning of the SMC, the objective is to maximize MQ. Being a sum of cluster factors, MQ is particularly well-suited to local search. If a module is moved from one cluster to another, only the two clusters involved in the exchange have their cluster factor values changed. All other clusters maintain their cluster factors, since their respective μ and ε are unaffected by the move. So, it is not necessary to fully recalculate the MQ value for every move operation: it can be calculated incrementally, updating only the values of the clusters involved in the changes (Mitchell, 2002). This incremental update operation has complexity equal to $O(1)$.

MQ is not universally accepted as a good objective function to find good solutions for the SMC. de Oliveira Barros et al. (2015) found that optimizing the distribution of modules into cluster using MQ leads to solutions having a large number of cluster, each with just a few modules. The same happens for EVM, an alternative objective function used to guide an optimizer searching for solutions of the SMC problem. According to Bavota et al. (2012) the automated methods used to find a system decomposition always led to solutions heaving a very high number of clusters, mostly composed of few or only one module. In the same way Hall et al. (2012) use a method called SUMO which allows user to interact in the modularization process, the authors conclude that the quality of the results increase consistently as more input from the developer is provided.

2.2. Preprocessing reduction procedure

Köhler et al. (2013) proposed a preprocessing procedure which aims to reduce the size of a given instance of the SMC. The procedure is based on the following theorem:

Theorem 1. *Let $G=(V,E)$ be the undirected weighted graph given as input for the SMC. Let $u\in V$ be a node with degree equal to one and $v\in V$ be adjacent to u. Then in the optimal solution of the SMC, u and v are assigned to the same cluster.*

Proof. cf. Köhler et al. (2013, page 121, Theorem 4.1) □

To apply the reduction procedure, one needs to transform the MDG into the non-directed and weighted graph $G=(V,E)$. If the MDG is non-weighted, we first set weights on all edges, assigning unitary weight to previously non-weighted edges. Next, considering a weighted MDG, opposite edges are merged to allow removing direction from all edges. The weight of the new “merged” edge is calculated as the sum of the weights of the two original edges. Let u be a node in V with degree equal to 1 and let v be the single node adjacent to u. The preprocessing algorithm removes u from V and adds a self-dependency (v, v) to E whose weight is the same as the removed dependency ($c_{vv}=c_{uv}$). Fig. 1 shows an example of

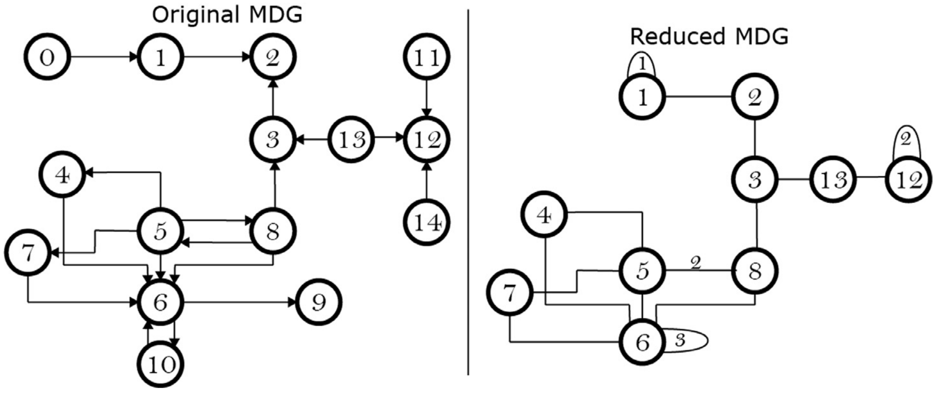

Fig. 1. Example of an MDG before and after the graph shrinkage preprocessing algorithm.

an MDG before and after the application of the reduction preprocessing procedure.

We used the above procedure to reduce the size of the original SMC instance. Then we applied our proposed method to heuristically solve the new reduced instance. Then, the solution to the reduced instance is transformed into a solution to the original instance, by adding all nodes that were eliminated from V and assigning them to the same cluster to which their adjacent nodes are assigned. We note that the solution quality of the reduced graph is the same as that of the original one (Köhler et al., 2013).

2.3. Local Search and Neighborhood

The use of a local search algorithm for a combinatorial optimization problem presupposes the definition of a neighborhood. This can be formulated as follows: let I be an instance of a combinatorial optimization problem, X its finite set of feasible solutions and $c : x \to \mathbb{R}$ a function that maps a solution to its cost. In this section, we assume that the combinatorial optimization problem is a minimization problem, that is, the objective is to find a solution x^* such that $c(x^*) \leq c(x) \forall x \in X$. A *neighborhood* N is a mapping which defines for each solution $x \in X$ a set $N(x) \subseteq X$ of solutions that are in some sense "close" to x. A solution x is said to be a *local optimum* (or *local minimum*) with respect to a neighborhood N if $c(x) \leq c(x') \forall x' \in N(x)$.

A basic version of a local search algorithm is *iterative improvement*. It starts with a given initial solution and searches its neighborhood for a better solution, that is, a solution with lower cost in the case of a minimization problem. If such a solution is found, it replaces the current solution, and the search continues. Otherwise, the algorithm returns the current solution, which is a local optimum. This iterative improvement strategy is also known as *iterative descent* or *Hill Climbing* (HC). One technique to escape from a local optimum solution is simply to reinitialize the search process whenever a local optimum is found, which is known as a *random restart* strategy.

2.4. Iterated Local Search

Mancoridis et al. (1998) propose applying a HC search with random restarts to the SMC problem. The neighborhood suggested by the authors is defined by all reachable solutions that can be obtained by moving one module to a different (or new) cluster. However, restarting from a completely random solution discards relevant characteristics of an already good solution that had been captured by the local attractor.

The *Iterated Local Search* (ILS) (Lourenço et al., 2010) metaheuristic addresses this problem by looking slightly beyond the neighborhood whenever a local search is stuck in a local optimum. It starts by applying a local search on an initial, random solution. Once a local optimum is reached, a perturbation operator is applied to the solution. As a result, a new solution is generated and becomes the initial solution of the next iteration of the local search. The perturbation operator produces something in between a neighboring solution and a completely random one. The intention is to take the best known solution found thus far, and push it away from the local attractor while preserving some good characteristics of the solution.

Pinto (2014) and Pinto et al. (2014) applied an ILS-based heuristic to the SMC problem, namely ILS_SMC. They generated initial solutions using the Agglomerative Hierarchical Clustering Method proposed by Hansen and Jaumard (1997). A Hill Climbing algorithm using best improvement selection was used as the local search component of the algorithm. The perturbation operator randomly selects 10% of the modules and moves each one of them to a randomly and independently selected cluster. Experiments performed on 40 instances show that ILS_SMC outperforms genetic algorithms using different solution representations in both solution quality and required processing time.

3. Proposed solution

This section presents the Large Neighborhood Search metaheuristic according to Pisinger and Ropke (2010) and discusses its application to the SMC problem.

3.1. Large Neighborhood Search

The Large Neighborhood Search (LNS) metaheuristic was first proposed by Shaw (1998) as a member of the class of Very Large Scale Neighborhood Search (VLSN) algorithms. According to Ahuja et al. (2002), for a search algorithm to be considered as a VLSN its neighborhood must grow exponentially with the instance size, or the neighborhood must simply be too large to be exhaustively searched. Intuitively, a search in a very large neighborhood should lead towards solutions with better quality than a search in small neighborhoods. However, in practice, small neighborhoods can provide similar or even higher-quality solutions when embedded in heuristics, simply because the search can be more efficiently performed. Thus, the success of a VLSN based heuristic is not guaranteed only by the size of its neighborhood (Pisinger and Ropke, 2010).

In LNS the neighborhood is implicitly defined by the methods used to iteratively *destroy* and *repair* the current solution. At each iteration a `destroy` method takes a valid solution and removes some of its components, creating a temporary, invalid solution. Then, a `repair` method takes that temporary solution and reinserts the elements that had been removed, generating another valid solution (Pisinger and Ropke, 2010). If the solution generated by the `repair` method is better than the previous best known solution, that solution is stored as the current best known solution. At the end of all iterations, the best solution found is returned.

LNS has a simple structure and its pseudo-code is shown in Algorithm 1. The algorithm needs three control variables and four functions. The control variables are: x, used to store the current solution; x^t, used to store a temporary solution obtained after the execution of the `destroy` and `repair` methods and x^b, which stores the best known solution. The functions are: *destroy*(x), responsible for the partial destruction of the solution *x; repair*(x), responsible for the repairing process; *accept*(x^t, x), responsible for verifying if the temporary solution x^t will be accepted as the current solution x; and *cost*(x), responsible for calculating the quality of solution x.

Algorithm 1 Large Neighborhood Search Algorithm.

```
1:  $x \leftarrow$ a feasible solution
2:  $x^b \leftarrow x$
3:  repeat
4:      $x^t \leftarrow repair(destroy(x))$
5:      if $accept(x^t, x)$ then
6:          $x \leftarrow x^t$
7:      end if
8:      if $cost(x^t) > cost(x^b)$ then
9:          $x^b = x^t$
10:     end if
11: until stop condition is met
12: return $x^b$
```

Typically, the `destroy` method contains a stochastic component so that each invocation of the method will destroy different parts of the solution. The neighborhood $N(x)$ of a solution x is defined as all other solutions that can be reached by first calling the `destroy` method and then the `repair` method. The size of the neighborhood $N(x)$ is defined as $max\{|N(x)|: x \in X\}$.

As an example of a neighborhood size, consider a percentage or *degree of destruction* equal to 15% and a solution with 100 components. There are $C(100, 15) = 100!/(15! \times 85!) = 2.5 \times 10^{17}$ different ways to destroy the solution and, for each partially destroyed solution generated, there are many different ways to rebuild it. (Of course this process can yield many equivalent solutions.) (Pisinger and Ropke, 2010).

Removing fewer elements implies a smaller neighborhood. With small neighborhoods, the search might rapidly converge to a local minimum of low quality. Removing more elements of the solution leaves the search freer to navigate the search space and find better solutions, although the processing time to find these solutions may be longer. One can also use an approach where the size of the neighborhood changes during the search, for instance, increasing the neighborhood size if the search reaches stagnation at some point. The name Large Neighborhood Search is explained by the fact that the neighborhood size can be large.

3.2. Large Neighborhood Search applied to the SMC problem

When developing heuristics based on a metaheuristic, it is necessary to make specific choices for the free components defined in the metaheuristic. LNS has the following free components: the *initial solution*, the `destroy` method, the `repair` method, the *acceptance test* and the *stopping criterion*. For all methods described below, we are given set N with n modules.

We used two constructive methods for building feasible initial solutions for SMC:

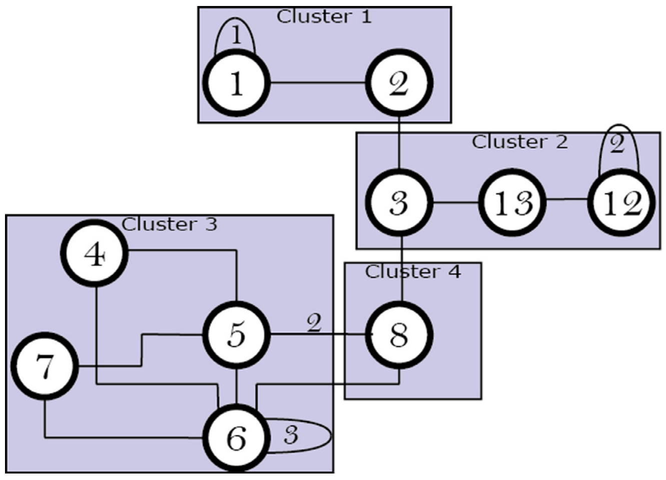

Fig. 2. Example of a solution that will be processed by different destruction methods.

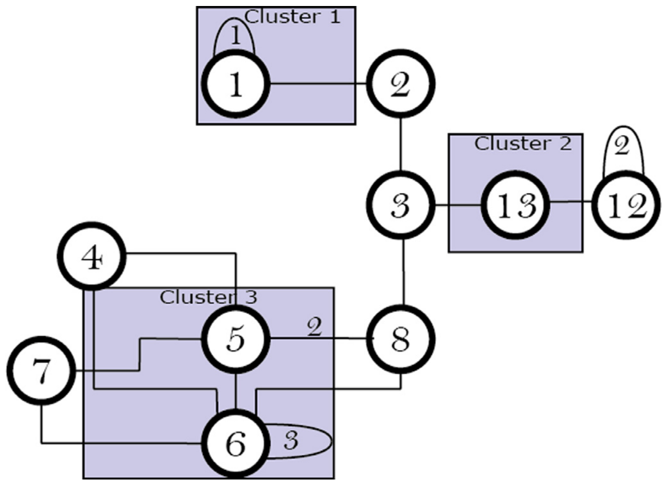

Fig. 3. Result of applying the DR destruction method upon the solution presented in Fig. 2. Modules 2, 3, 4, 7, 8, and 12 were randomly picked for removal.

- Constructive Random (CR): creates a random solution by generating, for each module $i \in N$, a random number k between 1 and n, and assigning i to cluster k.
- Constructive Agglomerative MQ (CAMQ): this is an iterative process that starts with a solution where each module is in a different cluster. At each iteration it evaluates all possible merge between two clusters and select the one with the resulting best MQ. It continues the process until all modules belong to the same cluster. The process returns the best visited solution.

The `destroy` method removes elements from a solution, turning a feasible solution into a partial one. The most important choice when implementing the `destroy` method is the percentage of destruction, controlled by the *degree of destruction* parameter (`degree_d`). The minimum value for this parameter is 0 (no module is removed from the solution) and the maximum value is 1 (all modules are removed from the solution). We proposed three `destroy` methods, which will be discussed using Fig. 2 as an example of a solution that is processed by a destruction method.

- Destructive Random (DR): removes $m \le n$ randomly selected modules from a solution. The complexity of this method is $O(m)$. Fig. 3 depicts the results of applying the DR method to Fig. 2, using $m = 6$.

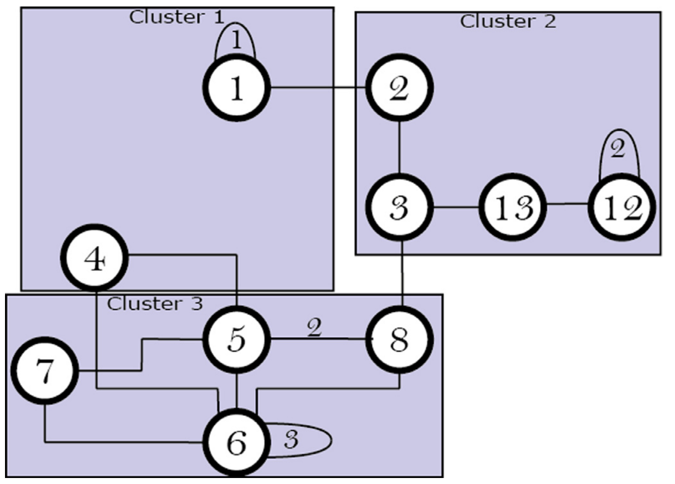

Fig. 4. Example of a hypothetical best solution based on Fig. 2 that was visited up to a given moment in the search.

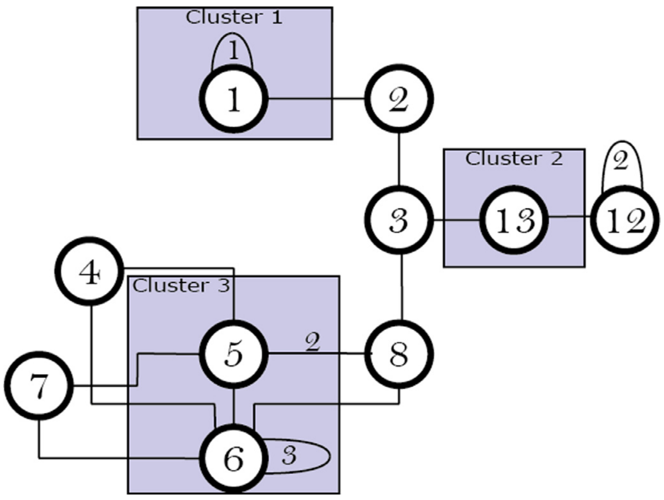

Fig. 5. Results of executing the DDBS destruction method upon the solution represented in Fig. 2 having Fig. 4 as the best solution known so far. Modules 2, 4 and 8 were removed because they are in different clusters in the best solution. Modules 3, 7 and 12 were randomly chosen to complete the removal process.

- Destructive Difference from the Best Solution (DDBS): given the best solution found so far, randomly removes $m \leq n$ modules whose cluster in the current solution differs from the cluster in the best solution. If fewer than m modules belong to different clusters, the remaining modules to be removed are randomly selected from the set of modules placed in the same cluster. The complexity of this method is $O(m)$. Fig. 4 depicts a hypothetical best solution visited up to the present moment in the search, while Fig. 5 shows the results of applying the DDBS method upon Fig. 2 using $m = 6$.
- Destructive Random Cluster (DRC): randomly selects a cluster and removes $m \leq n$ randomly selected modules from it. If the cluster is emptied during this process, another cluster is randomly selected until m modules have been removed from the solution. The complexity of this method is $O(m)$. Fig. 6 depicts an application of the DC method to the solution presented in Fig. 2, using $m = 6$.

The `repair` methods are responsible for turning a partially destroyed solution back into a feasible one. Pisinger and Ropke (2010) suggest using greedy algorithms for solution repair, since an optimal `repair` method may repeatedly generate the

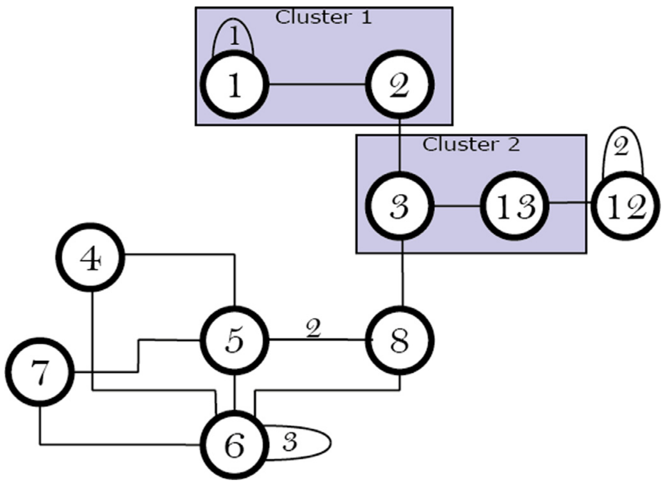

Fig. 6. Results of executing the DC destruction method upon the solution presented in Fig. 2. The method first chose Cluster 3 and removed all of its four modules. Then, it chose Cluster 4 and removed its single module. Finally, it chose Cluster 2 and randomly selected module 12 to be removed.

same solution and some diversity may allow the LNS algorithm to escape from local attractors. We proposed the following four alternative `repair` methods:

- Repair Greedy Best Improvement Random (RGBIR): at each iteration, randomly chooses one of the m modules removed by the `destroy` method. Next, checks all clusters, considering creating a new one, and assigns the current module to the cluster that leads to the highest MQ. The complexity of this method is $O(m \times n)$.
- Repair Greedy Worst Improvement Random (RGWIR): at each iteration, randomly chooses one of the m modules removed by the `destroy` method. Next, checks all clusters, considering creating a new one, and assigns the current module to the cluster that leads to the worst MQ improvement. If no cluster improves the MQ, the least worst cluster is selected. The complexity of this method is also $O(m \times n)$.
- Repair First Improvement Random (RFIR): at each iteration, randomly chooses one of the m modules removed by the `destroy` method. Next, checks the clusters ordered by their respective index, also considering creating a new cluster, and assigns the current module to the first cluster that improves the MQ. If no cluster improves the MQ, the least worst cluster is selected. Again, the complexity of this method is $O(m \times n)$.
- Repair Greedy Best Improvement (RGBI): at each iteration, checks the insertion of all removed modules in all clusters, also considering creating a new cluster, and chooses the module/cluster assignment that produces the best MQ. The complexity of this method is $O(m \times n^2)$.

As can be seen in Algorithm 1, the repaired solution x^t (line 4) must pass through an acceptance test (line 5) that determines whether or not it should replace the best known solution. The proposed heuristic only accepts solutions that improve MQ.

Finally, we selected two alternative stopping criteria for the algorithm: (i) to stop after a certain number of iterations are executed, controlled by parameter `max_iterations`; and (ii) to stop after a number of iterations are completed without improvement to the quality of the best known solution, controlled by parameter `no_improvement_max_threshold`.

Given the number of alternatives for each free component of our LNS for the SMC, we have to decide on which methods should be used and which values should be assigned to their parameters. The next section presents a sequence of experimental studies that

Table 1
Comparison between the initial solution construction methods. CAMQ clearly outperforms the CR method.

Value	Points	Time (s)	% Points
CAMQ *	7455	0.74	78.65%
CR	2024	0.58	21.35%

Table 2
Comparison between different repair methods. RGBI scored more points, but required 30 times the processing time of RGBIR, which is also a competitive method in terms of points.

Value	Points	Time (s)	% Points
RGBI *	15817	1.98	53.93%
RGBIR *	11853	0.07	40.41%
RFIR	1659	0.02	5.66%
RGWIR	0	0.06	0.00%

Table 3
Comparison between destroy methods. DR scored more points, closely followed by DDBS.

Value	Points	Time (s)	% Points
DR *	20283	1.77	50.10%
DDBS	19975	1.78	49.34%
DRC	226	1.30	0.56%

Table 4
Comparison between the values to be assigned to the destruction degree parameter. The value 0.10 represents a good balance between number of "victories" (i.e., higher point scores) and processing time required.

Value	Points	Time (s)	% Points
0.15	14646	0.44	14.23%
0.10 *	14163	0.21	13.76%
0.20	13470	0.75	13.09%
0.25	11914	1.15	11.58%
0.30	10772	1.63	10.47%
0.35	9892	2.21	9.61%
0.40	8530	2.85	8.29%
0.45	7449	3.50	7.24%
0.50	6297	4.17	6.12%
0.05	5787	0.06	5.62%

were planned to determine the best parameter values for the LNS algorithm when applied to the SMC problem.

4. Configuring the proposed approach

In order to make the ideal choices for the LNS components and the values of their parameters, we designed and executed two sets of experiments described in Sections 4.1 and 4.2. All experiments were executed on a computer with an Intel Core i7-2600 CPU 3.40GHz processor, 4GB DDR3 RAM memory, running Windows 7 and the program encoding the experiment. The algorithms were implemented in Java 7 and compiled using JDK 1.8.0_20-b26. For the purpose of the experiments, we selected 18 instances that were previously used in other works (Barros, 2012; Köhler et al., 2013; Kumari and Srinivas, 2013; Mitchell, 2002; Pinto, 2014; Praditwong et al., 2011). These instances vary in size (from 20 to 293 modules) and complexity (from 57 to 1745 dependencies).

4.1. Configuring the LNS free components

The first experimental study aimed to evaluate different choices for the free components of the LNS metaheuristic presented in Section 3.2. For this purpose, we use the term *experiment parameter* to designate a component of the LNS, or a parameter of the algorithms used. We also use the term *parameter value* for any possible choice for a given component, or any given values that can be assumed by a parameter. The possible choices for the components of the LNS algorithm are the methods described in Section 3.2, while the possible choices for the `degree_d` parameter are values in the interval [0, 1].

The experiment was conducted as follows: first, an experiment parameter was chosen to be studied. Next, all other parameters had their values randomly selected within the possible choices available for each parameter. Then, the search was performed once for each possible value of the parameter being studied and we took note of the maximum MQ found by this search, as well as the time required to complete the search. After all parameter values were evaluated for the parameter under study, we awarded one point for each value of this parameter which had produced the maximum MQ found. If all parameter values produced the same MQ, none of these values were awarded any point for the round. To reduce the effect of stochastic components on the parameters, or of a specific choice of parameter values, this procedure was repeated 2000 times for each instance, re-sampling the values of the other parameters at each run. After setting a given experiment parameter, its value was used in subsequent experiments, reducing the number of experimental parameters to have their values sampled. Each search in this process was executed for 500 iterations.

Tables 1 to 4 have similar structure. The "Points" column displays the number of times a parameter value achieved the best result ("victory") in comparison to other values for the same parameter. The second column, called "Time (s)", shows the average processing time required to perform the search. Finally, the "%% Points" column shows the percentage of points that a given parameter value received. An asterisk next to the parameter's value means the value was selected and maintained for the following experiments. On the other hand, values without an asterisk were discarded.

The first experiment parameter analyzed was the initial solution construction method. Table 1 shows that the CAMQ method found the best solutions more frequently than the CR algorithm. On average, for every 100 runs examined, CAMQ found the best solution in 78 of them.

The second experiment parameter analyzed was the `repair` method. This parameter is responsible for choosing the algorithm used to repair solutions. Table 2 shows that the RGBI method received about 54%% of the total points, while RGBIR received another 40%%. Both methods were kept for the next round of experiments: RGBI was selected because it received the most points, whereas RGBIR was selected because it is 30 times faster than RGBI and is also competitive in terms of points.

The third experiment parameter analyzed was the `destroy` method. Table 3 presents the data obtained in the experiment. Both the DR and DDBS methods received about 50%% of the total points and took approximately the same time to execute. DR received slightly more points and was therefore selected for the last experiment in this sequence.

Finally, we analyzed the destruction degree parameter. This parameter is responsible for the number of modules that are removed from the solution at each iteration. It is represented as a percentage of the number of modules in the instance. To avoid bias, we picked the values used in the work of Ropke and Pisinger (2006). Thus, the values ranged from 0.05 to 0.5 using gaps of 0.05. Table 4 presents the data obtained in the experiment. The values 0.15 and 0.10 achieved a roughly similar number

Table 5
The values chosen for the experiment parameters after the first set of experiments.

Experiment parameter	Values
Initial solution construction method	CAMQ
Repair method	RGBIR, RGBI
Destroy method	DR
Destruction degree	0.1

of points. Although the value 0.15 received a few more points than 0.10, it was twice as expensive in terms of processing time. Thus, a value of 0.10 was selected for the destruction degree parameter.

Table 5 displays the values chosen for each parameter after the first set of experiments. Interestingly, the solution construction methods achieve better results when they use non-random components (CAMQ and RGBI), while the `destroy` method is dependent on random components.

4.2. Evaluating LNS Configurations

The second experimental study investigated the stopping criterion component. Unlike the first study, which evaluated each experiment parameter in isolation, the second study compares various *configurations* of the algorithm, where a given configuration assigns a value for each parameter. This study evaluates the chain effect of a set of values for the components of the algorithm. The search was performed 100 times for each of 18 instances.

Four types of configurations were created. The first, called *FIXED*, has all experimental parameters fixed, and the search stops when a maximum number of iterations is reached. The second configuration type, named *MIXED*, uses two `repair` methods: RGBIR and RGBI. The search starts using one of the `repair` methods, and when a maximum number of iterations without improving the best known solution value (controlled by the `no_improvement_threshold` parameter) is reached, it flips to the other `repair` method. The search will flip back to the first method if it reaches `no_improvement_threshold` cycles and the second method made some improvement to its starting solution. If no improvement was observed, the search stops.

The third configuration type, *MIXED NO RESET*, is similar to the *MIXED* configuration but stops after the second `repair` method is executed, regardless of any improvement. The last configuration type, *MIXED RESET 0.5*, starts with a given `repair` method and flips to the other when it reaches `no_improvement_threshold`. Once the second `repair` method reaches `no_improvement_threshold`, the search flips back to the first `repair` method if any improvement has occurred, but this time the maximum number of iterations without improvement is halved. The search stops when both `repair` methods are executed without improving the best solution. Table 6 shows the configurations that were analyzed. The table contains the values chosen for each parameter and configuration.

Fig. 7 shows the average MQ obtained by each LNS configuration. To calculate this value, we first calculated the average MQ value for each instance over 100 evaluation cycles and then calculated the average of these values. Fig. 8 shows the average time required for the search under each LNS configuration following the same procedure.

Table 7 shows in matrix form a comparison of the MQ values obtained for each pair of configurations listed in the first row and column of the matrix. The comparison was made by calculating the sum of the relative difference between the mean MQ values obtained by both methods for each instance, that is, $\sum_{i=1}^{18}(MQ_{i,row} - MQ_{i,column})/MQ_{i,row}$, where $MQ_{i,\ row}$ is the MQ value

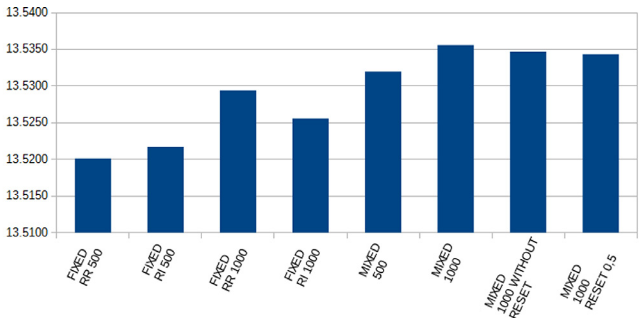

Fig. 7. Average MQ obtained by LNS configurations.

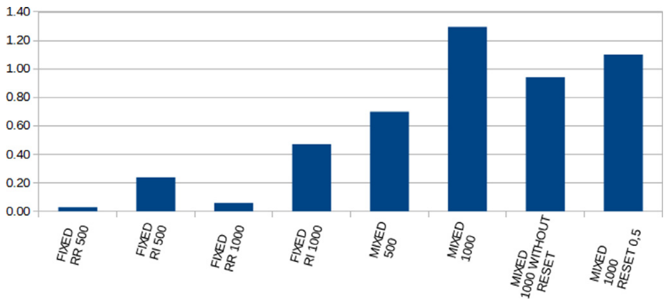

Fig. 8. Average time incurred by LNS configurations.

obtained for the *i*th instance using the method listed on the row and $MQ_{i,\ column}$ is the MQ value obtained for the *i*th instance using the method presented in the column. Thus, each cell in the matrix represents the value obtained by comparing the configuration listed in the row versus the configuration listed in the column. Positive values mean the configuration displayed in the row was better, while negative values mean the configuration displayed in the column was better.

The MIXED 1000 setting was the only one which obtained favorable results compared to all other settings. It also achieved the highest average MQ for 13 out of 18 instances (72%%). However, this better solution quality came with a great computational cost. Focusing on solution quality, we chose MIXED 1000 as the default configuration for the LNS algorithm applied to the SMC problem, hereafter known as *LNS_SMC*. This configuration is used in the next section, which evaluates the results produced by LNS as compared to other algorithms addressing the SMC problem.

5. Evaluation of the proposed approach

To evaluate LNS_SMC, we used 96 instances collected from several works that addressed the SMC problem (Barros, 2012; Köhler et al., 2013; Kumari and Srinivas, 2013; Mancoridis et al., 1998; Mitchell, 2002; Pinto, 2014; Praditwong, 2011; Praditwong et al., 2011), and 28 instances that we built from open-source systems. All 124 instances were used in a study that compared LNS_SMC to ILS_SMC (Pinto, 2014), which to our knowledge is the best mono-objective algorithm that has been applied to the SMC problem.

To facilitate analysis, the 124 instances were classified into categories according to their size. We used the *k-means* clustering method to group these instances, allowing the clustering method to create five categories. *K-means* created categories containing, respectively, 64, 29, 18, 9, and 4 instances. As the last two categories were very small in comparison to the remaining ones, we decided to join them into a single category. Table 8 shows the category name, number of instances, and minimum and maximum number of modules for each category. The largest instance in our dataset has 1161 modules and more than 5700 dependencies. The

Table 6
LNS configurations under evaluation.

Experiment Parameter	FIXED RR 500	FIXED RI 500	FIXED RR 1000	FIXED RI 1000	MIXED 500	MIXED 1000	MIXED 1000 NO RESET	MIXED 1000 RESET 0.5
Initial solution construction	CAMQ	CAMQ	CAMQ	CAMQ	CAMQ	CAMQ	CAMQ	CAMQ
Repair method	RGBIR	RGBI	RGBIR	RGBI	RGBIR + RGBI	RGBIR + RGBI	RGBIR + RGBI	RGBIR + RGBI
Destroy method	DR	DR	DR	DR	DR	DR	DR	DR
Destruction degree	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
max_iterations	500	500	1000	1000	–	–	–	–
no_improvement_threshold	–	–	–	–	500	1000	1000	1000

Table 7
Pair-to-pair comparison between LNS configurations studied.

	FIXED RI 500	FIXED RR 1000	FIXED RI 1000	MIXED 500	MIXED 1000	MIXED 1000 NO RESET	MIXED 1000 RESET 0.5
FIXED RR 500	−0.20%	−1.46%	−1.05%	−1.70%	−2.59%	−2.23%	−2.12%
FIXED RI 500	–	−1.26%	−0.85%	−1.50%	−2.39%	−2.03%	−1.92%
FIXED RR 1000	–	–	0.41%	−0.24%	−1.12%	−0.76%	−0.66%
FIXED RI 1000	–	–	–	−0.65%	−1.54%	−1.18%	−1.07%
MIXED 500	–	–	–	–	−0.88%	−0.52%	−0.42%
MIXED 1000	–	–		–	–	0.35%	0.46%
MIXED 1000 NO RESET	–	–	–	–	–	–	0.11%

Table 8
Instances categorized by number of modules.

Category	# of Modules	# of Instances
Small (S)	From 2 to 74	64
Medium (M)	From 79 to 182	29
Large (L)	From 190 to 377	18
Very Large (VL)	From 413 to 1161	13

Table 9
Percentage reduction in modules and dependencies for each instance category. It can be observed that smaller instances were more affected by the preprocessing step than large instances.

Instance category	Modules reduced (%)	Dependencies reduced (%)
Small	22.90%	23.38%
Medium	19.09%	20.41%
Large	11.06%	13.39%
Very Large	9.19%	15.34%
Average	**18.85%**	**20.39%**

instances were grouped according to their original size, that is, before the preprocessing step that reduces instance size was applied.

Table 9 shows the average percentage reduction in modules and dependencies that were removed by the preprocessing step discussed in Section 2.2. Surprisingly, smaller instances showed the largest reduction, both in size and complexity, while the largest instances showed the smallest percentage reduction in size over the instance dataset.

The experiments reported in the next subsections were performed to answer the following research question:

- RQ1: Does LNS_SMC produce solutions with higher MQ and in shorter time than ILS_SMC?
  Both heuristics were executed for all instances. We compared mean solution quality and average processing time obtained by each heuristic, for each instance in our dataset.

5.1. Comparison between LNS_SMC and ILS_SMC

To answer RQ1, we compared the results obtained by LNS_SMC to the results obtained by the ILS_SMC heuristic proposed by Pinto (2014). The source code for ILS_SMC is available online.[3] Originally, the ILS_SMC algorithm did not accept weighted MDGs or self-dependencies. So, for the execution of the study reported in this section we updated the source code accordingly.

A hundred independent runs were executed for each algorithm and for each instance to account for the stochastic components present in both algorithms. Then, statistical inference tests and standardized effect-size measures were applied to determine whether the results obtained by LNS_SMC could be considered significantly different from those produced by ILS_SMC. The R statistical tool (v3.1.0) was used to calculate all statistics.

We used the non-parametric Wilcoxon–Mann–Whitney test at 95%% confidence level as inference test (Feltovich, 2003). This test compares the medians of two independent samples, indicating whether these samples have significantly different medians or if the observed difference can be attributed to chance. It does not require that the samples resemble the normal distribution or present similar variance. For each instance in our dataset, we used the statistical inference test to determine if LNS_SMC found solutions with significantly higher median MQ than ILS_SMC, awarding one “victory” to our proposed algorithm. On the other hand, if ILS_SMC found solutions with significantly higher median MQ than LNS_SMC, we awarded one victory to the competing heuristic. Otherwise, it was considered to be a tie between the algorithms, and no victory was awarded.

Standardized effect-size measures, such as the non-parametric Vargha and Delaney $\hat{A}_{12}$ statistics (Vargha and Delaney, 2000), assess the magnitude of improvement in a pair-wise comparison. Given a measure M for observations collected after applying treatments A and B, $\hat{A}_{12}$ measures the probability that treatment A yields higher M value than B. If both treatments are equivalent, then $\hat{A}_{12} = 0.5$; otherwise, the value of $\hat{A}_{12}$ indicates the frequency of improvement. For instance, $\hat{A}_{12} = 0.8$ denotes that higher results are obtained 80%% of the time with A.

Table 10 displays a summary of the results produced by our experimental study comparing the quality of solutions found by the LNS_SMC and ILS_SMC heuristics. Detailed average results over 100 executions on an instance basis are presented in Appendix A, which also shows optimal values for 77 instances and upper bounds for 12 instances.

For the 64 instances in the Small category, LNS_SMC produced solutions with significantly higher median MQ than ILS_SMC for 36 instances, ILS_SMC found significantly better solutions for 13 instances, and the algorithms reached a tie for the remaining 15 instances. For the 29 instances in the Medium category, LNS_SMC

[3] https://github.com/cmsp/ils.git.

Table 10
Number of victories for each algorithm in the pairwise comparison of solution quality over 100 optimization cycles for our instances, along with average differences in results and effect-size measures.

Instance category	LNS_SMC	ILS_SMC	Ties	Mean gap	Mean ES
Small	36	13	15	22.3%	68%
Medium	26	1	2	14.4%	87%
Large	18	0	0	1.9%	94%
Very Large	13	0	0	2.7%	99%

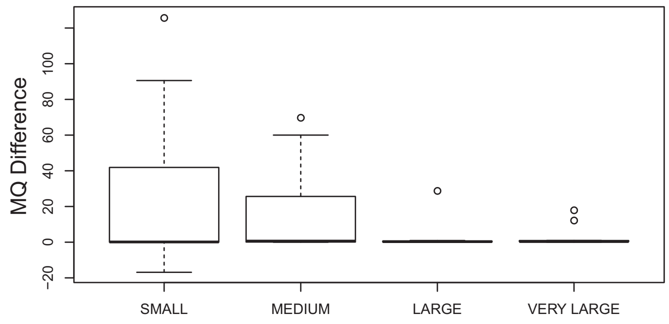

Fig. 9. Comparison between the MQ of the solutions found by the LNS_SMC and ILS_SMC heuristics.

Table 11
Comparison between LNS_SMC and ILS_SMC in terms of processing time for the 124 instances in our dataset.

Category	LNS_SMC was faster	ILS_SMC was faster
Small	51	13
Medium	22	7
Large	2	16
Very Large	0	13

found significantly better solutions for 26 instances, was outperformed by ILS_SMC in one instance, and two ties were observed. Finally, LNS_SMC found significantly better solutions than ILS_SMC for 30 out of 31 instances from the Large and Very Large categories, reaching a tie for the remaining instance.

Table 10 also shows the mean gap and effect-size for this comparison. The gap is calculated on an instance basis and averaged over the category. For any given instance, the gap is calculated as $\frac{\mu_{LNS}-\mu_{ILS}}{\mu_{ILS}}$, where μ_{LNS} represents the mean solution value produced by LNS_SMC for the instance, and μ_{ILS} represents the corresponding mean solution value produced by ILS_SMC.

Fig. 9 displays the distribution of the gap between LNS_SMC and ILS_SMC for each category of instances. Almost all values on the box-plots are positive, meaning that LNS_SMC beats ILS_SMC for almost all instances (as shown in Table 10). One can notice the differences between the median and mean values for each category: while the medians are close to 0%%, the means represent a much larger improvement due to outliers. Meanwhile, the observed effect-sizes in Table 10 support the conclusion that LNS_SMC finds better solutions than ILS_SMC more frequently for medium, large and very large instances. Our proposed algorithm seems well-suited for the SMC problem, especially if applied for large instances.

Fig. 10 displays the average processing time for each instance. The x-axis is arranged according to the number of modules in each instance after preprocessing its MDG, while the y-axis shows the computational time required to complete the search using log_{10} scale. Table 11 summarizes the comparison of processing time between LNS_SMC and ILS_SMC. The second column shows the number of instances for which LNS_SMC was faster, while the third one shows the number of instances for which ILS_SMC was faster.

Fig. 11 compares the mean processing times for LNS_SMC (t_{LNS}) and ILS_SMC (t_{ILS}). The chart shows the distributions of the increase in processing time required by LNS_SMC, that is, the ratio $\frac{t_{LNS}-t_{ILS}}{t_{ILS}}$. We can notice that for categories Small, Medium and Large the time difference is almost zero, meaning that both algorithms have close execution times for those instance. However, for Very Large instances, the execution times required by LNS_SMC are much higher than for ILS_SMC.

Overall, we observe that LNS_SMC can find good solutions for Medium, Large, and Very Large instances, though it consumes more processing time than ILS_SMC. We believe the extra processing time required by LNS_SMC might not be a problem for most practical purposes, even for large instances.

5.2. Characteristics of solutions found by LNS_SMC

This section discusses the characteristics of the best solution found by LNS_SMC for each instance. According to Lanza and Marinescu (2010), Java programs have on average 17 modules per cluster. The total of all modules across all 124 instances (after preprocessing) is 16313 modules, whereas the best solutions found by LNS_SMC total to 6061 clusters. Therefore, these solutions have on average 2.7 modules per cluster. This number is below the ratio found in real-world projects and is compatible with the ratio found by de Oliveira Barros et al. (2015) while studying the SMC problem for Apache Ant. Fig. 12 shows a box-plot depicting the modules-to-cluster ratio for each instance category. We observe that for 80%% of the instances, the ratio is between one to four modules per cluster. The mean and median obtained are 2.78 and 2.76 modules per cluster, respectively.

Praditwong et al. (2011) define two multi-objective formulations for the SMC problem (Section 7), namely *MCA* and *ECA*. The MCA formulation considers, among other factors, the number of single-module clusters (to be minimized), while ECA considers the difference between the largest and smallest cluster (also to be minimized). Although LNS_SMC did not consider these factors, we calculated their values for the best solutions found. They are presented in Table 12 for each instance category. We can observe that more than 73%% of the solutions have at least one single-module cluster. These solutions also present a large number of single-module clusters: on average, 13.6%% of the clusters have a single module. Furthermore, the average difference in the number of modules between the largest and the smallest cluster as a percentage of the total number of modules (last column) decreases rapidly as the instances grow in size.

5.3. Discussion

The small average number of classes per cluster, the large number of single-module clusters, and the small average difference in the number of modules between the largest and the smallest cluster suggest that the best solutions found by LNS_SMC may not lead to module distributions aligned to developers' expectations and useful for software projects. But if the results generated by LNS_SMC cannot help developers to reorganize the modules comprising their systems into clusters, what is the use of solving the SMC problem and, consequently, of our proposed algorithm? We believe there are several answers to this question.

First, the exercise of solving the SMC problem using optimization has lent support to the argument that the coupling/cohesion mantra dictated by theoretical software design and serving as the basis for the MQ fitness function may not be enough to capture the intentions of software developers (de Oliveira Barros et al., 2015). In consequence, we need deeper knowledge about design to find new objective functions to drive optimization. Without optimization, we might still be applying limited changes to increase

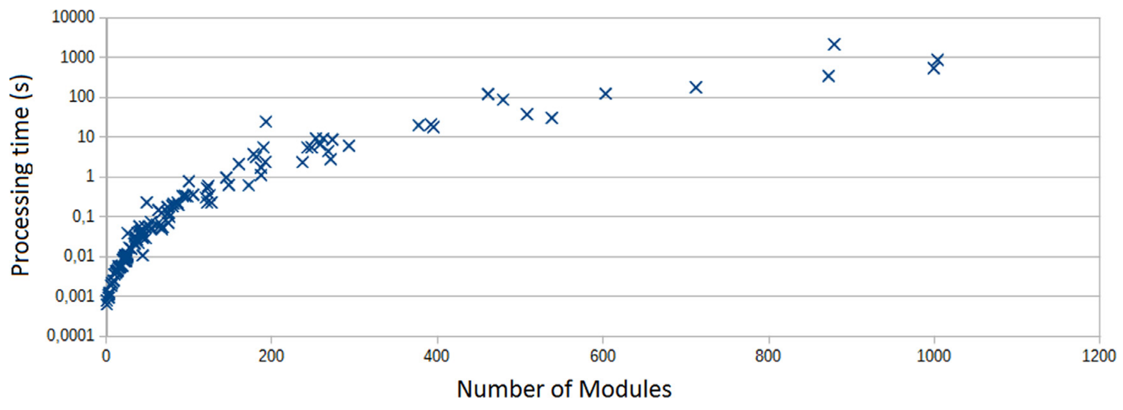

Fig. 10. Average processing time according to instance size after the MDG reduction preprocessing step.

Table 12
Summary of the characteristics of best solutions, according to instance categories.

Instance category	% Instances with single- module clusters	% Single- module clusters	% Difference in size
Small	54.69%	14.78%	14.56%
Medium	89.66%	19.59%	6.85%
Large	94.44%	11.35%	3.97%
Very Large	100.00%	12.48%	2.07%
TOTAL	**73.39%**	**13.63%**	**9.85%**

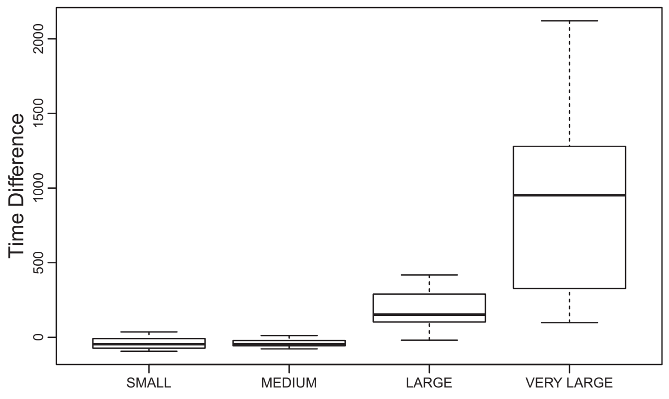

Fig. 11. Comparison between the processing time of the solutions found by the LNS_SMC and ILS_SMC heuristics.

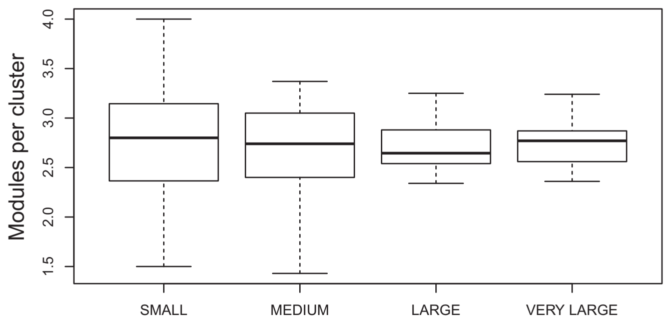

Fig. 12. Ratio between the total number of clusters and the total number of modules in each category.

cohesion and/or reduce coupling in software systems, believing that successive application of such procedures would provide ever-increasing benefits to the design of the software.

Next, although the logical organization of modules into clusters proposed by the optimization process may not entirely represent the desires of developers, it may point them to weak spots in their own distributions, providing guidance for reconsidering some of their decisions. Optimization may also benefit from guidance from developers, accepting input from them about pairs of modules that should be placed in different clusters or in the same cluster. Having this feedback may even provide researchers with data about where current measures fail to capture developers' intentions, pointing to future research directions.

Finally, the module distributions found by LNS_SMC may represent different perspectives of the relationships between modules, such as modules that are related to the same errors/changes to the source-code base or modules that change together frequently. These perspectives may not represent the logical distribution of modules into clusters. For instance, developers may distribute modules according to stereotypes or system layers, accepting the crisscross of dependencies between clusters as a price to be paid to bring together modules that require the same skills to build. Thus, the distributions proposed by LNS_SMC may be used by recommendation systems to show other aspects of the structure of the software aside from the logical distribution of modules into clusters.

5.4. Threats to validity

To allow the reproducibility of the study reported in this section we made a series of design decisions: (i) we selected problem instances that were already used in other studies; (ii) we selected an objective function (MQ) that has also been extensively used in former studies to allow comparing the results obtained through our algorithms; and (iii) the source code and instances used in the experiment were made available online,[4] along with all data resulting from the execution of the experiment. In addition, to account for the randomness of stochastic search procedures, all algorithms were executed several times for each instance: 2,000 times while selecting the free components of our approach and 100 times while evaluating the approach itself.

[4] https://bitbucket.org/marlonmoncores/unirio_lns_cms.

6. Human-based study

To determine whether the ideas discussed in Section 5.3 can be observed in practice, we invited a group of developers to participate in a study focusing the distribution of the modules comprising a software project into clusters. The object of our study is JodaMoney,[5] a Java library comprised of 20 core classes that stores, manipulates, and represents monetary amounts. We selected JodaMoney because it was used in the experiment reported in Section 5 and it is small enough for the subjects to understand the relationships and roles played by its classes in the two-hour session they were willing to invest in our study.

6.1. Subject characterization

We have contacted 20 software developers with mixed experience in object-oriented programming and design. Thirteen of these subjects accepted to participate in the proposed study. These developers work or have worked in the software development industry, both in private and public held companies. Most of these companies are based in Brazil, but one subject works for a Canadian company, while a second one works for a company based in the United States.

Before running the study, we sent a characterization questionnaire to all subjects. We aimed to separate the subjects into experienced and inexperienced developers to analyze whether the results of our study might be influenced by subjects having more or less experience. To that purpose, we measured their programming and design skills by proposing two questions for each criterion. These questions would be answered in a three-point Likert-scale representing low, medium, or high experience. On programming, we asked how many years the subject has worked as a programmer (up to 5 years, from 6 to 10 years, or more than 10 years) and about his/her experience in object-oriented programming (low, medium, or high). Concerning design, we asked about the experience on interpreting class diagrams (low, medium, or high) and experience with software architecture (low, medium, or high).

For each criterion, we observed whether there was agreement on the answers to both questions and assigned the agreed upon experience level to the subject. One last question was added to the characterization questionnaire to address academic formation. Whenever disagreement arose while evaluating a given criterion, the answer to the question addressing academic formation was used to resolve the dispute. We observed a broad range of academic formation in our subjects, though all of them have Computer Science background. Two subjects were DSc students, three were MSc students, two completed a Master degree program, two completed a *latu-sensu* post-graduate course, two completed graduation, and the last two were undergraduate students working as interns for their companies. Subjects holding a Master degree were considered experienced and if their answers to the questions addressing a given criterion were in disagreement, their classification drifted to the higher rank. On the other end, undergraduate and graduate students were given the lower rank if a disagreement appeared in their assessments.

Regarding programming, an agreement on the answers was reached for only 5 out of 13 subjects. Three subjects reported less than five years of programming practice and medium experience with object-oriented programming (OOP), being ranked as low-experienced programmers. Two subjects reported six to ten years working as programmers and high experience with OOP and were ranked as high-experienced. An MSc student reported working as a programmer for more than ten years, but only medium experience with OOP and was ranked as a medium-experienced programmer. A subject holding an MSc degree reported working six to ten years with OOP and was ranked as a high-experienced programmer. Finally, a DSc student reported working for less than five years with programming, but having high experience with OOP and was ranked as high-experienced. We ended up with eight high-experienced, two medium-experienced, and three low-experienced programmers.

Regarding design, an agreement on the answers was reached for 9 out of 13 subjects. An undergraduate subject reported medium experience in reading class diagrams and low experience on software architecture and was ranked as a low-experienced designer. A subject holding an MSc degree reported low experience in interpreting class diagrams but high experience in software architecture and was ranked as a high-experienced designer. Another subject holding an MSc degree reported high experience in reading class diagrams, but medium experience on software architecture and was ranked as a high-experienced designer. Finally, a DSc student reported medium experience in reading class diagrams and low experience in software architecture and was classified as a medium-experienced designer. In the end, we had four high-experienced, four medium-experienced, and five low-experienced designers.

Finally, subjects were classified as experienced or inexperienced according to the former criteria. Consistent with expectations, whenever a subject was given a high score in design, he/she also received a high score in programming. Thus, high-experienced programmers with at least medium design experience were classified as experienced subjects, while the remaining ones were classified as inexperienced. Overall, we had seven experienced and six inexperienced subjects. The analyses presented in the next subsections take these groups into consideration.

6.2. Solution characteristics

After filling the characterization questionnaire, the subjects received a class diagram depicting JodaMoney's core classes, their attributes, and relationships, along with a brief description of the project. We then asked the subjects to group the classes into as many clusters as they thought were necessary for the project. No further information was provided to the subjects, though they could afford enough time to search for more information on the Internet.

The solutions proposed by the subjects were all different among themselves and also different from the solution proposed by LNS_SMC and JodaMoney's developers. LNS_SMC proposed distributing the classes into nine clusters, the largest one having four classes and the smallest one having a single class. JodaMoney's authors used only two clusters, each holding 10 classes.

The average number of clusters proposed by our subjects was 6.08 (median=6). Experienced subjects proposed using on average 6.14 clusters, while inexperienced ones proposed using 6 clusters (the median was 6 for both groups). Two experienced subjects proposed the smallest solutions using three clusters, while another experienced subject proposed the larger distribution using 11 clusters. Experienced subjects used on average 2.0 single module clusters (median=2), while inexperienced ones used 1.2 such clusters (median=1). The average difference between the largest cluster and the smallest one for experienced subjects was 4.7 classes (median=4), while for inexperienced subjects it was 5.2 classes (median=5). The largest cluster, comprised of 12 classes, was proposed by an experienced subject.

We observe the lack of a clear agreement on the distribution of classes into clusters, even for such a small project as JodaMoney. Both experienced and inexperienced subjects used more clusters than it would be suggested by the literature, though the number of clusters used by JodaMoney's developers is more consis-

[5] http://www.joda.org/joda-money/

tent with such literature. Moreover, both groups of subjects used roughly the same number of single class clusters (inexperienced subjects seemed slightly more prone to the recommendation of avoiding such structures and not concentrating many classes in a single cluster).

Only 5 out of 13 subjects proposed a hierarchical decomposition of classes into clusters, while the remaining eight subjects opted for a flat distribution, with no cluster containing another cluster. Two out of these five subjects were experienced. This is an interesting aspect because providing a flat distribution of classes into clusters is a known limitation of the proposed formulation of the SMC problem, one that is inherited in LNS_SMC. However, at least for small projects, this does not seem to be a concern for many developers.

6.3. Predictions on issue resolution

JodaMoney currently uses the GitHub platform to keep its source-code base. The platform offers an issue tracking system where bugs and requests for new features can be reported and followed by the library's users as developers work on them. The platform also offers some integration between the version control system on which the source code is stored and the issue tracking system. This integration allows marking an issue as resolved by referring to its identifier in a commit to the version control system. Such *issue-related-commits* allow researchers to find which source-code modules were changed to resolve an issue.

As of May 2017, JodaMoney has 50 issues registered in its issue tracking system: 41 resolved issues and 9 open issues. Twenty-three issues were resolved through a commit to the version control system. Thirteen of these issues involved at least one Java file (a class among the core classes comprising JodaMoney) that was not a test case. Six issues (#16, #24, #31, #35, #47 and #58) involved two or more classes. We used the classes involved in these issues to determine whether the solution found by LNS_SMC would be a better predictor for classes that might be related to the same issues than the solution proposed by JodaMoney's authors or by our subjects.

All six issues involved classes pertaining to the same cluster in the solution proposed by JodaMoney's developers. On the other hand, only four of these issues involved classes that pertain to the same cluster in LNS_SMC's solution. The *BigMoney* and *CurrencyUnit* classes were changed together to solve issue #3, but pertain to different clusters in the solution found by optimization. Similarly, the *CurrencyUnit, CurrencyUnitDataProvider* and *DefaultCurrencyUnitDataProvider* classes were changed together as part of issue #24's resolution, but are not part of the same cluster in the solution proposed by LNS_SMC.

Experienced subjects proposed solutions that encompass classes that were changed together to resolve on average 4.6 issues (median=5), while inexperienced subjects brought together classes that were changed to resolve on average 3.6 issues (median=4). Thus, we observe that solutions found by experienced developers perform better than the solution found by optimization as a predictor of classes that might change together to resolve issues. On the other hand, the solution found by LNS_SMC performs a bit better than those proposed by inexperienced subjects. This provides some evidence that coupling and cohesion may be associated with changes related to the same purpose, but further experiments are needed to provide stronger evidence.

6.4. Predictions on concomitant changes

JodaMoney has been developed over the last eight years since its first commit to the version control system on August 19, 2009. The log of the version control system records 282 commits, 255 of which performed by the most active developer engaged in the project (Stephen Colebourne). After him, one developer has performed three commits, six developers performed two commits, and twelve developers performed a single commit. Half of the 282 commits involve one or more Java classes and 129 of them involve at least one Java class that is not a test case. Fifty-nine commits involve two or more Java classes among the 20 core classes in JodaMoney's design.

Table 13
Percentile of pairs of classes that were changed together and reside on the same cluster for each solution. Bold face values cannot be attributed only to chance.

JodaMoney authors	LNS_SMC	Experienced subjects	Inexperienced subjects
56%	13%	**49%**	**32%**
		37%	**30%**
		37%	**28%**
		32%	**22%**
		30%	20%
		18%	18%
		9%	
56%	13%	30%	25%

We identified 919 pairs of classes that were changed together in the 59 commits comprising more than one core class. The *Money* and *BigMoney* classes were changed together more often, participating in 29 out of the 59 commits. To check if a given solution for the SMC problem is a good predictor for classes that would be changed together, we counted how many of the 919 pairs reside in the same cluster in solutions provided by JodaMoney's authors, LNS_SMC, and our subjects. We also verified whether the number of pairs appearing together is larger than what would be expected due to the number of clusters on each solution and applied a χ^2 inference test at 99%% confidence level to determine whether being larger than expected could be attributed only to chance.

Table 13 presents the results we have found. The first and second columns present the percentile of pairs of classes that were changed together and reside on the same cluster for the clustering used by JodaMoney's authors and for the solution proposed by LNS_SMC, respectively. The third column presents the same information for all solutions proposed by our experienced subjects, one per line. The last column presents the same information for our inexperienced subjects.

The solution proposed by JodaMoney's authors is the best predictor for classes that change together and, despite having only two clusters, the number of correct predictions cannot be attributed to chance alone. The solution found by optimization, on the other hand, is the worst predictor and the number of correct predictions might be due to chance. Solutions proposed by experienced subjects perform relatively well, except for a poor solution rating at 9%% of correct predictions. Solutions proposed by inexperienced subjects perform below the bar of those proposed by experienced developers, but still make on average twice as many correct predictions as the solution found by optimization.

6.5. Comparison among solutions

Given that the solution proposed by JodaMoney's authors was the best predictor for concomitant changes and issue-related changes, next we examine how close are the solutions found by optimization and proposed by our subjects to the one used by JodaMoney's authors using the MojoFM metric (Wen and Tzerpos, 2004). MojoFM(A,B) measures the distance between two clustering solutions, A and B, by calculating the number of modules that must be moved from one cluster to another and the number of clusters that must be merged to transform a given solu-

Table 14
MojoFM for solutions proposed by our subjects and found by optimization compared to the solution proposed by JodaMoney's authors.

LNS_SMC	Experienced subjects	Inexperienced subjects
61%	78%	78%
	78%	78%
	67%	72%
	61%	67%
	61%	67%
	56%	61%
	39%	
61%	63%	70%

tion A into B. It varies from 0 to 1, a larger number denoting two solutions that represent roughly the same clustering and a number close to zero representing two solutions that are very different from each other.

Table 14 presents the MojoFM values for the solution found by optimization and those proposed by our subjects when compared to the solution proposed by JodaMoney's authors. We observe that inexperienced subjects provided the closest solutions to the reference one, followed closely by the experienced subjects who are kept back by an outlier solution that performs well below the remaining ones (average grows to 67%% if this solution is discarded).

Most importantly, LNS_SMC's solution departs more from the distribution of classes to clusters proposed by the authors than the solutions provided by 8 out of 13 subjects. So, if we assume that the clustering provided by the authors is a good solution because of the most active developer's experience, as well as its ability to predict classes that are changed together during the development life-cycle, the solution found by LNS_SMC cannot be considered a good solution from a software developer's perspective.

6.6. Post-hoc analysis by subjects

On completing the distribution of JodaMoney's classes into clusters, subjects were asked to fill a post-hoc questionnaire which asked for the rationale they have used to perform the task and if any potentially useful information was lacking. Both questions were optional and only 9 out of 13 subjects provided answers for them.

We had only two clear answers to the rationale used to distribute classes into clusters, one pointing to coupling between classes and another stating that all exception classes were grouped in the same package (hinting on grouping based on class stereotype). The second question provided richer answers: 9 out of 13 subjects stated that the class diagram and project description were not enough to perform the task, five being experienced and four inexperienced subjects.

Seven subjects asked for more information about the classes, stating that the role played by some of these classes could not be inferred by their names, attributes, and relationships; two subjects asked for sequence diagrams, which capture dynamic relationships among classes, as they call each other's methods to perform some task; and, finally, one subject asked for a description of the requirements of the library under analysis.

The huge interest in class descriptions suggests that subjects were trying to understand the objectives of the classes to find a suitable distribution among clusters. JodaMoney's classes can be roughly divided into four groups: *model* classes, that represent monetary amounts and currencies; *provider* classes, that allow instantiating model classes; *parser/printer* classes, that allow parsing strings containing monetary units and printing monetary amounts into strings, according to a given style and a format; and *exception* classes, that represent exceptions that might be raised by algorithms processing monetary units. These groups are overlapping and subject to interpretation: for instance, one may say that a parsing exception is both an exception and a parser/printer class.

JodaMoney's authors separated parser/printer classes into a cluster and model/provider classes into a second one, distributing exceptions accordingly to the position of the classes that raise them. The solution found by LNS_SMC also separated parser/printer classes from the remaining ones, but the optimizer used five clusters to group these classes instead of a single cluster, each of these clusters having little cohesion from a semantic point-of-view. Eleven subjects created a cluster having at least half of the classes dedicated to parsing and printing, leading to the conclusion that they have used class names to infer their purpose and decided to bring those classes together. It helps that seven of these parser/printer classes are strongly connected among themselves and disconnected from the remaining classes in the class diagram.

JodaMoney's authors and four subjects (three experienced, one inexperienced) placed provider classes in the same cluster that holds the classes they instantiate. The solution found by LNS_SMC placed one provider class along with its related class and the remaining two provider classes in a separate cluster; this was also the solution adopted by two subjects, both experienced. Two subjects placed the three provider classes in the same cluster, separately from all other classes.

Five subjects grouped all exceptions in the same cluster (two experienced subjects and three inexperienced). Two subjects separated exceptions into two clusters: one for a formatting-related exception and the other for two currency-related exceptions. Both subjects used hierarchical clustering to place the exceptions under clusters holding the classes that implement the actions that might trigger them. Thus, subjects identified the purpose of exception classes due to naming conventions (exception classes are usually named using the *Exception* suffix) and separated them into a cluster conveying classes that are not structurally cohesive (do not depend on each other), but have the same purpose (they are semantically related).

6.7. Discussion

Based on the data presented in the former subsections, we conclude that at least for the JodaMoney project the solution found by optimization is a poor predictor both of concomitant changes and changes related to new features and issues. The solution proposed by JodaMoney's authors, on the other hand, is a good predictor for both kinds of changes. In addition, 8 out of 13 solutions proposed by the subjects that participated in our study were closer to this reference solution (in terms of moving classes to other clusters and merging clusters) than the solution found by LNS_SMC.

Further examination of the solutions proposed by the authors and the subjects shows that they frequently take into account the roles played by the classes comprising the software, building clusters with classes that perform similar roles. Developers seem to look for a kind of *semantic cohesion* instead of the structural cohesion outlined by usage dependencies among classes and often suggested as good practice in design. Even if they use more than one cluster to hold all classes that perform a given role, developers do not frequently mix these classes with those playing other roles. In addition, coupling seems to be a secondary criterion to select among different class distributions that represent similar levels of semantic cohesion.

So what is the use of the coupling/cohesion mantra and of an optimization algorithm based on the design principle that suggests maximizing cohesion and minimizing coupling? Besides arguing that the former evidence was brought to light in no small part due to the existence of such algorithm, we observe that there are certain situations in which the algorithm finds suitable distributions of classes into clusters. Taking Sullivan's motto that *"form ever follows function"* in architecture (Sullivan, 1896), we posit that the SMC problem formulation leads to solutions that can be useful to developers if the structural dependencies present in the software (that is, the usage dependencies among classes used to measure cohesion and coupling) bind classes that perform similar roles.

The relation between structural and semantic properties can be observed in certain (parts of) software, but it does not apply to every class in the source code. Taking JodaMoney as an example, on one hand we have the dependencies binding the *parser/printer* classes. If submitted to a clustering procedure, such as LNS_SMC, this *Bauhaussian* part of the library results in clusters that might be reasonable to developers, even if the number of cluster is exaggerated for the number of classes involved. On the other hand, exception classes are seldom bound by any kind of dependency, even though they have similar functions. Thus, they are scattered across several clusters, each exception class being attracted to the cluster comprising the classes that raise the particular exception.

Therefore, if *form* (defined here by the structural dependencies) follows *function* (we drop the *ever* in favor of the *if*), the SMC problem formulation will lead to solutions that merge classes that are both structurally and semantically related. To which extent such phenomena happens in software systems other than JodaMoney is unknown and further studies are required to discover.

6.8. Threats to validity

The study reported in this section has some limitations that may threaten the validity of its results.

First, the study is based on a single software project that is used to evaluate the usefulness of solutions for the SMC problem that were found through different means. JodaMoney is a small software library, comprised of 20 core classes, and may not be representative of the properties of the average software project. However, the size of the project is reasonable if we take into account that subjects were not expected to invest more than two hours in the study. Moreover, we were able to gather enough information to discuss the limitations of SMC even from this small project.

Besides being a small project, JodaMoney had 50 issues registered in its issue tracking system and 282 commits registered in its version control system by the time we collected information for the study. The number of issues related to two or more classes was eventually reduced to six, and the number of commits related to two or more Java classes was reduced to fifty-nine. These are relatively small numbers for the average software project and therefore may be biased towards the design chosen by JodaMoney's developers.

A second limitation refers to the number of subjects participating in the study. While only thirteen subjects participated in the study, they were all active software developers working for companies in Brazil or abroad. Therefore, we had a small, but representative and diversified sample of subjects. On the other hand, many subjects asked for more information after performing the experiment and we cannot rule out the possibility that their answers would be different if they were given such data.

7. Related work

Local search has shown to be a reliable class of algorithms to address the SMC problem. Mancoridis et al. (1998) were the first to present the SMC problem as an optimization problem. They developed the Bunch tool,[6] which implements a local search algorithm that finds close-to-optimal solutions for instances of the SMC problem. Their work was extended by Mitchell (2002), who tested several alternatives to the Hill Climbing search driving Bunch.

The building blocks technique (Mahdavi et al., 2003) involves performing multiple Hill Climbing searches and identifying modules that are associated to the same cluster in a percentage of the best solutions found by the independent searches. Then, these modules are fixed in a solution and a new series of Hill Climbing searches is performed to find the best allocation for the remaining modules. The authors compared the technique with random restart Hill Climbing and found that it generates better solutions for both weighted and non-weighted MDGs, particularly for large graphs.

Barros (2013) introduces the concept of incremental search-based software engineering – ISBSE. The distribution of modules to clusters made by the software development team is taken as an initial solution for a local search and a degree of disturbance is applied. The optimization is made incrementally, i.e., several optimization rounds are performed over progressively improved solutions. After each round, developers can analyze the generated solution and discard some changes made by the optimizer. The next iteration will not make changes that have already been discarded.

Besides local search, deterministic algorithms have been used to solve small instances of the SMC problem. Mancoridis et al. (1998) used exhaustive search to solve instances with up to 15 modules. Li (1994) modeled the SMC problem as a series of equations with binary values and used Mixed-Integer Linear Programming (MILP) along with branch-and-bound to find the optimal solution for small instances. Köhler et al. (2013) also used a MILP formulation to find the optimal solution for 34 MDGs with up to 29 modules. The best exact approach for the SMCP is the work of Kramer et al. (2016), which proposes a Dantzig-Wolfe decomposition of one of the Mixed Integer Linear Programming (MILP) formulations proposed in Köhler et al. (2013) and solve it by using two column generation techniques (the standard column generation (CG) and a new approach called Staged Column Generation (SCG)), leading to a branch-and-price algorithm. The CG approaches for the SMCP either find and prove the optimal solution value or find an upper bound value on the MQ value. To perform computational experiments, they considered a set of 45 MDG instances, with up to 62 modules. Recently, Kramer (2017) made experiments with the standard column generation (CG) presented in Kramer et al. (2016) considering a set of 44 MDGs instances, presenting optimal solution value for 32 instances and new upper bounds for the remaining 12 instances. Considering both works, there are 89 instances for which CG approaches was able to provide an upper bound on the solution value.

Genetic algorithms were also used to find solutions for the SMC problem. Doval et al. (1999) compared the solutions found by a genetic algorithm to those produced by local search, concluding that the former was unable to outperform Hill Climbing searches. Later, Praditwong (2011) compared the results of genetic algorithms using two solution representations, namely *Group Number Encoding* (GNE) and *Grouping Genetic Algorithms* (GGA). GGA outperformed GNE in nine out of ten weighted MDGs. On the other hand, GNE outperformed GGA in five out of seven non-weighted graphs. The author concludes that there is no single representation for SMC solutions that is better for all cases.

As the MQ objective function models a balance between two opposing metrics, some authors have attempted to use multi-objective genetic algorithms to evaluate whether better solutions could be found if such metrics were analyzed separately. Praditwong et al. (2011) define two multi-objective formulations to the SMC problem: the *Maximizing Cluster Approach* (MCA) and the *Equal-Size cluster Approach* (ECA). MCA seeks to maximize the sum of the intra-edges of all clusters, minimize the sum of the inter-

[6] https://www.cs.drexel.edu/~spiros/.

edges of all clusters, maximize the number of clusters, maximize the MQ value, and minimize the number of clusters having a single module. ECA uses the first four functions of MCA and minimizes the difference between the maximum and minimum number of modules in all clusters. Experimental analysis showed that a Hill Climbing search could outperform MCA on non-weighted MDGs, but the latter outperformed the former for weighted graphs. Meanwhile, ECA outperformed local search and MCA for both types of MDG. Overall, the authors show that multi-objective genetic algorithms can find solutions with better MQ than local search, though with a significant increase in processing time.

Mkaouer et al. (2014) propose modeling SMC as a many-objective problem. The authors use seven objective functions in their formulation: minimizing the number of modules per cluster, minimizing the number of clusters, maximizing cluster cohesion, minimizing cluster coupling, minimizing changes to the distribution provided by developers, minimizing code changes needed to move modules from one cluster to another, and maximizing the consistency with change history. As implied in the functions, the work goes beyond the relationships between modules and considers source code changes, version control history, and distributions provided by developers, in order to propose evolved solutions for the problem. The authors used NSGA-III (Deb and Jain, 2014) to find solutions for a set of instances under this formulation. They compared their method to state-of-the-art remodularization approaches and concluded that their approach significantly outperforms, on average, existing approaches according to both a quantitative and a qualitative (human-oriented) evaluation.

Kumari and Srinivas (2013) use a genetic based multi-objective hyper-heuristic, called Multi-objective Hyper-heuristic Evolutionary Algorithm - MHypEA, to find solutions for the SMC problem. The heuristic to be used is determined by a weighting schema. Initially, all heuristics have the same weight. At each iteration, the weights are adjusted depending on the performance shown by each heuristic. The MHypEA produces better solutions and is 20 times faster than the multi-objective genetic algorithm proposed by Praditwong et al. (2011).

Semaan and Ochi (2007) present a hybrid evolutionary algorithm based on a genetic heuristic which works in conjunction with Hill Climbing and Path Relinking. In Path Relinking, two solutions are compared (usually, the best known solution and the current solution). The goal is to transform one solution into another through the generation of intermediate solutions. Eventually, some of the intermediate solutions may be better than the starting solutions (Semaan and Ochi, 2007). The authors found the best results when the search used all three proposed methods. Although the use of the three techniques together produced a better solution, the processing time was increased. On the other hand, the configuration using only Hill Climbing achieved similar results with reduced processing time (Semaan and Ochi, 2007).

8. Conclusions and future work

In this work we proposed and evaluated a heuristic based on Large Neighborhood Search for the SMC problem (LNS_SMC). The core features of an LNS-based heuristic are its `destroy` and `repair` methods which implicitly define the neighborhood of a given solution. In addition, an LNS-based heuristic has three other free components to be defined: the initial solution, the acceptance test and the stopping criterion. A robust computational study was made with 18 benchmark instances to define the choices for all five components of the final version of LNS_SMC. The study evaluated three alternatives for the `destroy` method, four alternatives for the `repair` method, two strategies to generate the initial solution, and ten distinct values for the destruction degree parameter, leading to almost 700,000 independent runs of LNS_SMC. Another feature of the proposed heuristic is the use of a preprocessing reduction procedure available in the literature.

We compared the results obtained by LNS_SMC with those obtained by ILS_SMC, the heuristic that found the best results for the SMC problem so far, considering 124 instances ranging from 2 to 1161 modules. Statistical inference tests were used whenever possible to prevent conclusions due to chance. Effect-size measures were also used to calculate the relative improvement of our algorithm over the state-of-the-art one. LNS_SMC improved the mean solution found by ILS_SMC (Pinto, 2014) in 93 out of 124 instances, showing the robustness and performance of the algorithm.

We expect to study applications of the results found by LNS_SMC in the next steps of our research. As close-to-optimal solutions have shown properties that might inhibit their automatic use as module distributions, we expect to analyze whether they can help developers as part of recommendation systems, either to guide distributing modules or to support predictive impact analysis, finding modules related to issues or finding modules that are changed together.

Acknowledgments

We thank two anonymous referees for several valuable comments that contributed to improve the final presentation of this paper. In addition, the authors are grateful to Hugo H. Kramer who provided new upper bounds on the MQ value for a set of 44 instances.

Appendix A. Detailed experiments results for SMCP instances

This appendix presents detailed results obtained by computational experiments on software module clustering problem instances. For the set of 124 instances, presented in Table 8 of Section 4.2, we present detailed computational results in Tables A.1 and A.2.

Table A.1presents results for Column Generation approaches (Kramer, 2017; Kramer et al., 2016), LSN_SMC (Section 3) and ILS_SMC (Section 5.1) for 111 instances from categories “Small”, “Medium” and “Large” instances. Kramer et al. (2016) reported optimal solution values for 45 instances and Kramer (2017) reported optimal solution values for 32 instances and new upper bounds for 12 instances. In addition, Kramer (2017) reported the following ERRATA: “in Table 2 from Kramer et al. (2016), Dual Bound for instance SPDB should be 5.58974 instead of2.58974”. For category “Very Large”, for which there is no known upper bound, Table A.2 presents results for LSN_SMC and ILS_SMC. All computational experiments with LNS_SMC and ILS_SMC were performed on a Intel Core i7-2600 CPU 3.40 GHz with 4GB DDR3 of RAM memory. Algorithms LNS_SMC and ILS_SMC were coded in Java 7 and compiled with JDK 1.8.0_20-b26.

The first column of Table A.1 indicates the instance name. Columns “Orig” and “Pre” show, respectively, the number of modules ($|V|$) and the number of dependencies ($|E|$) in the original MDGs and in the MDGs after preprocessing. Column “CG” shows the upper bound of the MQ value obtained by column generation approaches, which, in case of proven optimality, is indicated by the symbol “*”, and time to find the upper bound value. In addition, next to the MQ value, we indicate, using superscript *a*, *b* or *c*, the reference of upper bound value, which is detailed in the three rows below the table. The next four columns indicate the average solution value and the average time in seconds over 100 executions, respectively, for methods LNS_SMC and ILS_SMC. Table A.1 is divided in three groups related to the instances categories.

The first column of Table A.2 indicates the instance name. Columns “Orig” and “Pre” show, respectively, the number of modules ($|V|$) and the number of dependencies ($|E|$) in the original

Table A1
Detailed computational results of best exact approaches (CG) (Kramer, 2017; Kramer et al., 2016), LNS_SMC and best heuristic approach ILS_SMC, for 111 instances from categories "Small", "Medium" and "Large" instances.

Instance	Orig		Pre		CG			LNS_SMC		ILS_SMC	
	|V|	|E|	|V|	|E|	MQ		time	MQ	s.	MQ	s.
Small instances											
squid	2	2	1	1	1.00000[c]	*	0s	1.00000	0.00	1.00000	0.00
small	6	5	3	5	1.83333[a]	*	0s	1.52381	0.00	1.83333	0.00
compiler	13	32	13	32	1.50649[a]	*	0s	1.49683	0.00	1.50649	0.00
random	13	32	12	23	2.44092[a]	*	0s	2.44092	0.00	2.44092	0.00
regexp	14	20	9	18	2.44704[a]	*	0s	2.44704	0.00	1.47592	0.00
jstl	15	20	14	15	5.00000[c]	*	1s	5.00000	0.00	5.00000	0.00
lab4	15	18	10	14	3.40000[a]	*	0s	3.40000	0.00	3.40000	0.00
netkit-ping	15	15	1	1	1.00000[a]	*	0s	1.00000	0.00	1.00000	0.01
nss_ldap	15	16	3	4	1.00000[a]	*	0s	0.97634	0.00	0.89721	0.01
nos	16	52	15	50	1.67748[a]	*	0s	1.65652	0.01	1.67433	0.01
lslayout	17	43	17	43	1.86130[a]	*	1s	1.81710	0.01	1.86130	0.00
boxer	18	29	12	29	3.10110[a]	*	0s	3.10110	0.00	3.10110	0.00
netkit-tftpd	18	23	6	11	1.14421[a]	*	0s	1.14421	0.00	0.95243	0.01
sharutils	19	36	14	30	2.54921[a]	*	0s	2.53381	0.00	1.85256	0.01
mtunis	20	57	20	57	2.31446[a]	*	0s	2.28564	0.01	2.31446	0.01
spdb	21	33	7	8	5.58974[c]	*	0s	5.58974	0.00	5.58974	0.01
xtell	22	57	14	44	2.00523[a]	*	0s	2.00523	0.00	1.18318	0.02
bunch	23	62	15	45	2.40600[a]	*	0s	2.40260	0.01	2.40564	0.01
ispell	24	103	23	97	2.36393[a]	*	3s	2.33231	0.01	2.35053	0.01
netkit-inetd	24	25	4	6	1.31206[a]	*	0s	1.31206	0.00	1.26011	0.02
nanoxml	25	64	23	62	3.81733[c]	*	16s	3.81705	0.01	3.81733	0.01
ciald	26	64	22	62	2.85126[b]	*	2s	2.84592	0.01	2.84717	0.01
jodamoney	26	102	26	85	2.74888[c]	*	33s	2.74888	0.01	2.74888	0.01
modulizer	26	66	18	57	2.75790[a]	*	1s	2.75790	0.01	2.75790	0.01
bootp	27	75	19	55	2.19853[a]	*	0s	2.19853	0.01	1.77014	0.02
jxlsreader	27	73	25	73	3.60330[c]		19s	3.59208	0.01	3.59666	0.01
sysklogd-1	28	74	22	65	1.71050[a]	*	2s	1.68703	0.01	1.08230	0.03
telnetd	28	81	19	62	1.84749[a]	*	1s	1.84749	0.01	1.32381	0.03
crond	29	112	25	90	2.30300[a]	*	2s	2.30300	0.01	1.71147	0.03
netkit-ftp	29	95	24	76	1.76800[a]	*	2s	1.75623	0.01	0.77652	0.03
rcs	29	163	28	155	2.27754[a]	*	24s	2.22608	0.02	2.23982	0.02
seemp	30	61	21	43	4.65359[c]	*	8s	4.65359	0.01	4.65359	0.01
dhcpd-2	31	122	26	104	3.49437[a]	*	1s	3.48894	0.01	2.68472	0.03
cyrus-sasl	32	100	21	77	3.25182[a]	*	0s	3.25182	0.01	2.51672	0.03
tcsh	32	105	23	86	1.21410[a]	*	3s	1.19454	0.01	0.64741	0.05
micq	33	156	26	116	2.16799[a]	*	1s	2.13288	0.01	1.11917	0.04
apache_zip	36	86	30	74	5.76634[c]	*	17s	5.76634	0.02	5.76493	0.02
star	36	89	36	89	3.83208[b]	*	11s	3.82497	0.03	3.81835	0.02
bison	37	179	36	167	2.70390[a]	*	23s	2.68281	0.03	2.68864	0.03
cia	38	636	34	167	3.75067[c]	*	1m43s	3.73064	0.02	3.57836	0.06
stunnel	38	97	25	78	2.52605[a]	*	2s	2.52605	0.01	2.00197	0.04
minicom	40	257	35	198	2.57585[a]	*	18s	2.57585	0.03	1.60281	0.08
mailx	41	331	38	244	3.24023[a]	*	12s	3.23952	0.04	2.22615	0.09
dot	42	255	40	248	2.84698[c]	*	1h13m	2.83787	0.06	2.83573	0.05
screen	42	292	35	208	2.26010[c]	*	3m30s	2.24550	0.02	1.35065	0.09
slang	45	242	42	184	4.67942[a]	*	7s	4.67903	0.04	3.24461	0.08
slrn	45	323	37	231	2.39276[a]	*	19s	2.38358	0.03	1.53312	0.10
net-tools	48	183	44	157	4.31590[a]	*	24s	4.29318	0.03	3.40884	0.07
graph10up49	49	1650	49	1054				1.25200	0.23	1.25299	0.43
wu-ftpd-1	50	230	44	187	2.44373[a]	*	23s	2.41242	0.03	1.34585	0.10
joe	51	540	44	318	3.34109[a]	*	39s	3.31855	0.04	1.84666	0.18
hw	53	51	15	28	8.49672[c]	*	3s	8.49672	0.01	8.49672	0.63
imapd-1	53	298	40	211	3.62499[a]	*	18s	3.62499	0.04	3.06581	0.12
wu-ftpd-3	54	278	48	222	3.39527[a]	*	23s	3.34020	0.03	1.97553	0.12
udt-java	56	227	54	210	5.28294[c]	*	11m2s	5.28282	0.05	5.28057	0.06
javaocr	58	155	43	126	9.02417[c]	*	1m7s	9.02066	0.05	9.00564	0.05
dhcpd-1	59	571	55	398	3.98710[a]	*	1m20s	3.96083	0.05	2.55276	0.23
icecast	60	650	51	419	2.75436[a]	*	1m12s	2.74742	0.06	1.99026	0.28
pfcda_base	60	197	54	168	7.33944[c]		2m4s	7.33187	0.07	7.33103	0.06
servletapi	61	131	47	122	9.88232[c]	*	1m6s	9.74747	0.05	9.80916	0.05
php	62	191	39	148	5.32421[a]	*	11s	5.32421	0.02	3.30435	0.12
bunch2	65	151	42	111	7.72506[c]	*	46s	7.71149	0.04	7.70552	0.07
forms	68	270	64	224	8.32579[c]	*	6m9s	8.32579	0.06	8.32579	0.09
jscatterplot	74	232	68	171	10.74190[c]	*	2m15s	10.74143	0.05	10.74087	0.09
Medium instances											
jxlscore	79	330	67	305	9.80785[c]		1h38m	9.79502	0.13	9.79084	0.12
elm-2	81	683	75	541	3.82964[c]	*	5h39m	3.79995	0.12	2.37492	0.34
jfluid	81	315	75	276	6.57961[c]	*	58m11s	6.57960	0.07	6.57910	0.14
grappa	86	295	76	249	12.70540[c]	*	7m12s	12.70536	0.10	12.69812	0.13
elm-1	88	941	81	736				4.30567	0.21	2.82003	0.48

(continued on next page)

Table A1 (*continued*)

Instance	Orig		Pre		CG			LNS_SMC		ILS_SMC	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	MQ		time	MQ	s.	MQ	s.
gnupg	88	601	74	420	7.15850 [c]	*	49m34s	7.13058	0.18	5.17002	0.38
inn	90	624	80	485	8.02398 [c]	*	1h2m	8.01629	0.18	6.44343	0.34
bash	92	901	86	662	5.93593 [c]		1h46m	5.88429	0.23	4.27161	0.47
jpassword	96	361	87	332	10.70680 [c]	*	1h13m	10.62576	0.19	10.61359	0.18
bitchx	97	1653	92	1075	4.38631 [c]		46h6m	4.35862	0.33	2.56038	0.95
junit	99	276	61	193	11.09010 [c]	*	2m38s	11.09003	0.06	11.09009	0.17
xntp	111	729	95	534	8.36145 [c]		2h15m	8.34925	0.33	6.64592	0.54
acqCIGNA	114	188	75	157	16.59710 [c]	*	4m6s	16.51660	0.12	16.45496	0.23
bunch_2	116	364	98	355	13.62040 [c]	*	1h5m	13.60757	0.32	13.57594	0.23
exim	118	1255	105	891				6.61602	0.36	5.46376	0.89
xmldom	118	209	67	159	10.92360 [c]	*	2m18s	10.92356	0.05	10.85657	0.22
cia++	124	369	63	309	15.47280 [c]	*	13m34s	15.47239	0.15	15.37372	0.26
tinytim	129	564	122	499	12.54530 [c]		29m12s	12.51998	0.51	12.50784	0.36
mod_ssl	135	1095	123	752	10.11770 [c]		6h55m	10.10195	0.59	7.97054	0.89
jkaryoscope	136	460	127	345	18.98710 [c]		26m49s	18.98674	0.23	18.97142	0.34
ncurses	138	682	120	487	11.83180 [c]	*	52m58s	11.82627	0.30	9.88493	0.64
gae_plugin_core	139	375	87	259	17.33700 [c]	*	5m35s	17.33576	0.19	17.27774	0.34
lynx	148	1745	100	1211				4.97140	0.78	3.59568	1.83
javacc	153	722	145	663				10.69303	0.96	10.61752	0.54
lucent	153	103	66	74	59.94880 [c]	*	4s	59.94877	0.06	59.93766	0.21
javageom	171	1445	160	1339				14.09949	2.10	14.07396	0.95
incl	174	360	122	329				13.61224	0.23	13.61269	0.51
jdendogram	177	583	148	447	26.08430 [c]	*	53m18s	26.06752	0.63	26.06631	0.56
xmlapi	182	413	125	358	19.10240 [c]		1h18m	19.09631	0.35	18.97819	0.55
Large instances											
jmetal	190	1137	178	1123				12.53937	3.73	12.42464	0.89
graph10up193	193	9190	193	7552				2.24651	24.39	2.23143	8.21
dom4j	195	930	181	876				19.21559	3.18	19.12779	0.82
nmh	198	3262	190	2473				9.40220	5.49	7.27407	3.42
pdf_renderer	199	629	172	497				22.35717	0.62	22.34373	0.77
jung_graph_model	207	603	187	532	32.0334 [c]		2h0m	32.02566	1.71	31.96997	0.71
jung_visualization	208	919	192	837				21.83006	2.38	21.77989	0.90
jconsole	220	859	187	663	26.5195 [c]	*	4h54m	26.51841	1.11	26.50565	0.97
pfcda_swing	248	885	237	801				29.07201	2.32	28.97777	1.14
jml-1.0b4	267	1745	262	1671				17.55401	9.23	17.45366	1.93
jpassword2	269	1348	248	1171				28.59214	5.50	28.49965	1.66
notelab-full	293	1349	273	1233				29.64456	8.66	29.45818	1.87
poormans CMS	301	1118	253	1009				34.23546	9.24	34.14020	1.79
log4j	305	1078	258	944				32.50388	6.50	32.45713	1.85
jtreeview	320	1057	268	830				48.12804	4.45	48.10833	1.97
bunchall	324	1339	243	1250				16.97494	5.45	16.87101	2.41
jace	338	1524	271	1209				26.82142	2.79	26.81593	2.91
javaws	377	1403	293	1107	38.3405 [c]		58h36m	38.31677	6.06	38.27501	2.99

[a] Result obtained with best column generation approach presented in (Kramer et al., 2016), extract from Table 2, and executed on a 2.5 GHz Intel Core i5-3210M PC with 6 GB RAM.

[b] Result obtained with best column generation approach presented in (Kramer et al., 2016), extract from Table 3, and executed on a 2.5 GHz Intel Core i5-3210M PC with 6 GB RAM.

[c] Result obtained with standard column generation (SCG) presented in (Kramer et al., 2016), executed on a Intel Core i7-4790 3,60 GHz with 16 GB RAM, extract from Personal Comunication (Kramer, 2017).

Table A2
Detailed computational results for LNS_SMC and best heuristic approach ILS_SMC for "Very large" instances.

Very Large instances								
Instance	Orig		Pre		LNS_SMC		ILS_SMC	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	MQ	s.	MQ	s.
swing	413	1513	377	1478	45.36454	19.91	44.93944	3.22
lwjgl-2.8.4	453	1976	392	1790	37.13882	20.80	37.00750	4.87
res_cobol	470	7163	461	5690	15.97610	120.76	15.97329	12.45
ping_libc	481	2854	395	1802	51.85827	17.86	46.02614	9.00
y_base	556	2510	479	2166	58.30854	87.04	58.11031	7.32
krb5	558	3793	508	2494	54.59182	37.07	46.10499	13.39
apache_ant_taskdef	626	2421	538	2108	66.63475	30.46	66.52446	9.37
itextpdf	650	3898	603	3555	58.65491	122.89	58.32766	11.68
apache_lucene_core	738	3726	712	3330	76.87917	176.11	76.58993	14.68
eclipse_jgit	909	5452	872	4904	86.54838	343.10	86.29530	24.87
linux	916	13493	876	11059	54.63761	2153.65	54.14175	49.51
apache_ant	1085	5329	999	4881	102.74301	539.80	102.40967	33.06
ylayout	1161	5770	1004	4990	111.03204	874.21	110.88459	39.37

Table A3
Relative goodness of LNS_SMC method against best exact CG approaches, for 89 instances grouped by their categories.

Category	LNS_SMC × CG (gap)
Small	−0.00708
Medium	−0.00213
Large	−0.00030

MDGs and in the MDGs after preprocessing. The next four columns indicate the average solution value and the average time in seconds over 100 executions, respectively, for methods LNS_SMC and ILS_SMC.

We note, in Table A.1, that CG approaches were able to find and prove the optimality for 77 instances. Considering these 77 instances, LNS_SMC found the optimal value for 30 instances (39%%): 25 “Small” instances and five “Medium” instances for which |gap| is smaller than 0.00001 (*jfluid, grappa, junit, xmldom*, and *lucent*). ILS_SMC found optimal value for 18 instances (23.4%%): 17 “Small” instances and one “Medium” instance (*junit*, with |gap| smaller than 0.00001). Considering CG approaches, the biggest instance solved to optimality is *jconsole* with 187 vertices after reduction within almost five hours of processing time. The “Large” instance *javaws*, with 292 vertices after reduction, took the longest processing time, equivalent to 58 h. For this instance, LNS_SMC produced a solution with gap equal to −0.00062 in 6.06 s.

For category “Small”, with 64 instances, LNS_SMC found better solutions than ILS_SMC for 38 instances (59.4%%) and found the same value for 12 instances (18.75%%): *squid, random, jstl, lab4, netkit-ping, boxer, spdb, jodamoney, modulizer, seemp, hw*, and *forms*. The 14 instances for which ILS_SMC found better solutions than LNS_SMC are: *small, compiler, nos, lslayout, mtunis, bunch, ispell, nanoxml, ciald, jxlsreader, rcs, bison, graph10up49*, and *servletapi*. For “Medium” instances, LNS_SMC found better results for 27 instances (93.1%%) if compared to ILS_SMC, the two exception instances being *junit* and *incl*. For categories “Large” and “Very Large” (Table A.2), LNS_SMC found better solutions than ILS_SMC for all 31 instances.

Finally, in Table A.3 we summarize average results from Table A.1 and evaluate the relative goodness of the solutions obtained by LNS_SMC with respect to upper bounds on the MQ value obtained by Column Generation (CG) approaches, for 89 instances. The gap was obtained by averaging the ration $\frac{MQ_{LNS}-MQ_{CG}}{MQ_{CG}}$ for all instances within a group. Negative gap values for MQ mean that CG found better quality solutions than LNS_SMC. We note that the smallest group gap was obtained for “Large” instances (−0.00030), followed by “Medium” instances (−0.00213). The biggest group gap obtained for “Small” instances was mainly due to the poor performance of LNS_SMC method for instance *small* (gap equal to −0.16883), with only six vertices. If this instance is discarded, the gap for small instances is reduced to −0.00447.

In this document we have reported the best known MQ values for 124 instances which was found by either CG, ILS_SMC or the proposed method LNS_SMC. We hope that the above detailed computational results will be useful to future researchers that aim to propose new approaches to find solutions for the software module clustering problem and wish to compare their results with the technical literature.

References

Ahuja, R.K., Ergun, O., Orlin, J.B., Punnen, A.P., 2002. A survey of very large-scale neighborhood search techniques. Discrete Appl. Math. 123 (1–3), 75–102.
Barros, M.O., 2012. An analysis of the effects of composite objectives in multiobjective software module clustering. In: Proceedings of the 2012 Conference on Genetic and Evolutionary Computation. ACM, Philadelphia, pp. 1205–1212.
Barros, M.O., 2013. An experimental study on incremental search-based software engineering. In: Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE) 2013, St. Petersburg, Russia, August 24–26, 2013., pp. 34–49. St.Petersburg
Bavota, G., Carnevale, F., De Lucia, A., Di Penta, M., Oliveto, R., 2012. Putting the developer in-the-loop: an interactive GA for software re-modularization. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 75–89.
Briand, L. C., Morasca, S., Basili, V. R., 1999. Defining and validating measures for object-based high-level design.
Deb, K., Jain, H., 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. IEEE Trans. Evol. Comput. 18 (4), 577–601.
Doval, D., Mancoridis, S., Mitchell, B.S., 1999. Automatic clustering of software systems using a genetic algorithm. In: Proceedings of the Software Technology and Engineering Practice, 1999 (STEP '99), pp. 73–81. Pittsburgh
Feltovich, N., 2003. Nonparametric tests of differences in medians: comparison of the Wilcoxon-Mann-Whitney and robust rank-order tests. Exp. Econ. 6 (3), 273–297.
Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences), 1 W. H. Freeman.
Gibbs, S., Casais, E., Nierstrasz, O., Pintado, X., Tsichritzis, D., 1990. Class management for software communities. Commun. ACM 33 (9), 90–103.
Glorie, M., Zaidman, A., van Deursen, A., Hofland, L., 2009. Splitting a large software repository for easing future software evolution&mdash;an industrial experience report. J. Softw. Maint. Evol. 21 (2), 113–141.
Hall, M., McMinn, P., Walkinshaw, N., 2012. Supervised software modularisation. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM). IEEE Computer Society, Washington, DC, USA, pp. 472–481.
Hansen, P., Jaumard, B., 1997. Cluster analysis and mathematical programming. Math. Program. 79 (1–3), 191–215.
Köhler, V., Fampa, M., Araújo, O., 2013. Mixed-integer linear programming formulations for the software clustering problem. Comput. Optim. Appl. 55 (1), 113–135.
Kramer, H. H., 2017. Private Communication. June.
Kramer, H.H., Uchoa, E., Fampa, M., Köhler, V., Vanderbeck, F., 2016. Column generation approaches for the software clustering problem. Comput. Optim. Appl. 64 (3), 843–864.
Kumari, A.C., Srinivas, K., 2013. Software module clustering using a fast multiobjective hyper-heuristic evolutionary algorithm. Int. J. Appl. Inf.Syst. 5 (6), 12–18. Published by Foundation of Computer Science, New York, USA.
Lanza, M., Marinescu, R., 2010. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag, Berlin.
Larman, C., 2002. Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process. Prentice Hall PTR.
Li, H.-L., 1994. A global approach for general 0–1 fractional programming. Eur. J. Oper. Res. 73 (3), 590–596.
Lourenço, H.R., Martin, O.C., Stützle, T., 2010. Iterated local search: framework and applications. In: Gendreau, M., Potvin, J.-Y. (Eds.), Handbook of Metaheuristics, 146. Springer US, New York, USA, pp. 363–397. chapter 12
Mahdavi, K., Harman, M., Hierons, R.M., 2003. A multiple hill climbing approach to software module clustering. In: Proceedings of the International Conference on Software Maintenance, 2003 (ICSM 2003). IEEE Computer Society, Amsterdam, pp. 315–324.
Mancoridis, S., Mitchell, B.S., Chen, Y., Gansner, E.R., 1999. Bunch: a clustering tool for the recovery and maintenance of software system structures. In: Proceedings of the International Conference on Software Maintenance, 1999 (ICSM '99), pp. 50–59. Oxford
Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y., Gansner, E.R., 1998. Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of the 6th International Workshop on Program Comprehension, 1998 (IWPC '98), Ischia, pp. 45–52.
McConnell, S., 2004. Code complete. Developer Best Practices. Pearson Education.
Mitchell, B.S., 2002. A Heuristic Search Approach to Solving the Software Clustering Problem. Drexel University, Philadelphia, PA, USA Ph.d. thesis.
Mkaouer, W., Kessentini, M., BECHIKH, S., DEB, K., OUNI, A., 2014. Many-objective software remodularization using NSGA-III. ACM Trans. Softw. Eng. Method.
Monçores, M.C., Alvim, A.C.F., Barros, M.O., 2015. Large neighborhood search for the software module clustering problem. In: Proceedings of the 11th Metaheuristics International Conference.
de Oliveira Barros, M., de Almeida Farzat, F., Travassos, G.H., 2015. Learning from optimization: a case study with apache ant. Inf. Softw. Technol. 57, 684–704.
Pinto, A., Alvim, A.C.F., Barros, M.O., 2014. ILS for the software module clustering problem. In: Simpósio Brasileiro de Pesquisa Operacional, pp. 1972–1983. Salvador
Pinto, A.F., 2014. Uma Heurística Baseada em Busca Local Iterada Para o Problema de Clusterização de Módulos de Software,. PPGI/UNIRIO, Rio de Janeiro, RJ, Brasil Master's thesis.
Pisinger, D., Ropke, S., 2010. Large neighborhood search. In: Gendreau, M., Potvin, J.-Y. (Eds.), Handbook of Metaheuristics. Springer US, New York, USA, pp. 399–419. chapter 12
Praditwong, K., 2011. Solving software module clustering problem by evolutionary algorithms. In: 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), Nakhon Pathom, pp. 154–159.
Praditwong, K., Harman, M., Yao, X., 2011. Software module clustering as a multiobjective search problem. IEEE Trans. Softw. Eng. 37 (2), 264–282.

Ropke, S., Pisinger, D., 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transp. Sci. 40 (4), 455–472.

Semaan, G.S., Ochi, L.S., 2007. Algoritmo evolutivo para o problema de clusteriza-Ã§ão em grafos orientados. Simpósio de pesquisa operacional da marinha, 2007 (SPOLM2007), Rio de Janeiro.

Shaw, P., 1998. Using constraint programming and local search methods to solve vehicle routing problems. In: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming. Springer-Verlag, Pisa, pp. 417–431.

Sullivan, L.H., 1896. The tall office building artistically considered. Lippincott's Mag. 403–409.

Vargha, A., Delaney, H.D., 2000. A critique and improvement of the cl common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. 25 (2), 101–132.

Wen, Z., Tzerpos, V., 2004. An effectiveness measure for software clustering algorithms. In: Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004., pp. 194–203.

Yourdon, E., Constantine, L., 1979. Structured design: fundamentals of a discipline of computer program and systems design. Yourdon Press computing series. Prentice Hall.