The Journal of Systems & Software 190 (2022) 111349

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

An efficient heuristic algorithm for software module clustering optimization☆

Javier Yuste, Abraham Duarte, Eduardo G. Pardo *

Universidad Rey Juan Carlos, C/Tulipán s/n, Móstoles, 28933, Madrid, Spain

ARTICLE INFO

Article history:
Received 8 February 2022
Received in revised form 6 April 2022
Accepted 20 April 2022
Available online 27 April 2022

Keywords:
Software Module Clustering
Search-Based Software Engineering
Heuristics
Modularization Quality
Maintainability

ABSTRACT

In the lifecycle of software projects, maintenance tasks usually entail 75% of the total costs, where most efforts are spent in understanding the program. To improve the maintainability of software projects, the code is often divided into components, which are then grouped in different modules following good design principles, lowering coupling and increasing cohesion. The Software Module Clustering Problem (SMCP) is an optimization problem that looks for maximizing the modularity of software projects in the context of Search-Based Software Engineering. In the SMCP, projects are often modeled as graphs. Therefore, the SMCP can be interpreted as a graph partitioning problem, which is proved to be NP-hard. In this work, we propose a new heuristic algorithm for software modularization, based on a Greedy Randomized Adaptive Search Procedure with Variable Neighborhood Descent. We present a three-fold categorization of neighborhoods for the SMCP and leverage domain-specific information to filter unpromising solutions. Our proposal has been successfully tested over a dataset of real software projects, outperforming the previous state-of-the-art approach in terms of Modularization Quality in very short computing times. Therefore, it could be integrated in software development tools to improve the quality of software projects in real time.

1. Introduction

The lifecycle of a software project contains different activities, such as requirements definition, system design, software development, and maintenance (International Organization for Standardization, 2017a). During the maintenance process, software projects are modified with the aim of fixing bugs, improving the code, preventing potential failures, and adapting the product to changes in the environment (Bakota et al., 2012). Frequently, up to 75% of the total costs of a software project are spent on the maintenance process (Chen et al., 2017). In this phase, most efforts are dedicated to comprehending the existent software. Moreover, the larger the project, the harder it is to understand its code base. As systems grow larger and more complex over time, their code becomes increasingly difficult to understand. To ease the understanding of software systems, the code is frequently divided into different components which interact with each other. The concept of a software component can be understood in different ways, such as: a class, a source code file, or even a folder. Moreover, the terms "module" and "component" are frequently used interchangeably or understood as subelements of one another depending on the context. Although the relationship between these terms is not yet standardized (International Organization for Standardization, 2017b), in this paper we will refer to atomic units (e.g., classes and source code files) as components and to groups of components (e.g., folders and packages) as modules.

In ISO 24765:2017, modularity is defined as the set of *"software attributes that provide a structure of highly independent components"* (International Organization for Standardization, 2017b). Therefore, the modularization of a software project is the process of organizing its components in different modules that are highly independent, facilitating the understanding of each part separately. Traditionally, the modularity of a program has been measured by its coupling and its cohesion. Thus, the objective of this process is to arrange the project such that it has low coupling (components from different modules are loosely connected) and high cohesion (components in the same module are highly related). This modularization process presents different benefits, such as easing the comprehension of the system or reducing the appearance of bugs in the code (Briand et al., 1999; Gibbs et al., 1990; Larman, 2012; McConnell, 2004).

Search-Based Software Engineering (SBSE) is a research area that focuses on tackling traditional Software Engineering tasks by reformulating them as optimization problems, allowing researchers and practitioners to employ automatic processes that improve the quality of software systems and reduce their costs

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail addresses: javier.yuste@urjc.es (J. Yuste), abraham.duarte@urjc.es (A. Duarte), eduardo.pardo@urjc.es (E.G. Pardo).

https://doi.org/10.1016/j.jss.2022.111349
0164-1212/

(Colanzi et al., 2020). Optimization problems are usually determined by two features. On the one hand, a huge number of different solutions and, on the other hand, a method to compare them and select the best one. Then, optimization is defined as the search process that looks for the best possible solution to an optimization problem, generally in a limited time (Duarte et al., 2007).

In this context, the Software Module Clustering Problem (SMCP) has been formulated as an optimization problem, which objective is to maximize the modularity of software systems, thus reducing the costs associated with the maintenance phase. Moreover, this problem has been proved to be $\mathcal{NP}$-hard (Brandes et al., 2007). For the SMCP, we can find several variants of the problem depending on the particular quality definition considered as the objective function to optimize. However, the most frequently studied objective function is Modularization Quality (MQ), presented by Mancoridis et al. (1998). This function calculates the quality of a solution based on a balance between coupling and cohesion. In this tradeoff, the coupling of the architecture must be minimized, while the cohesion of the different modules must be maximized. For the MQ evaluation metric, the best identified method in the related literature is the Large Neighborhood Search (LNS) procedure introduced by Monçores et al. (2018), which was favorably compared with previous approaches.

In this paper, to address the SMCP, we present an efficient and effective Greedy Randomized Adaptive Search Procedure (GRASP) (Feo and Resende, 1995), where the improvement phase is based on a Variable Neighborhood Descent (VND) method (Mladenović and Hansen, 1997). To evaluate the quality of the proposed solutions, we use the Modularization Quality (MQ) objective function. In particular, we study the most recent MQ measurement, the TurboMQ formulation, as presented by Mitchell and Mancoridis (2002a). We then compare the results obtained by our proposal with those obtained by the LNS algorithm proposed in Monçores et al. (2018). To perform a fair comparison, we consider a set of 124 instances made publicly available by the authors of that work. This dataset contains instances with up to 1161 components and 11722 dependencies. We show that our approach produces solutions of equal or better quality for 116 out of the 124 tested instances. Furthermore, according to the Wilcoxon Signed-Rank test, the obtained results are statistically significant with a p-value lower than 0.01.

The rest of this paper is organized as follows. First, in Section 2, we review the previous work related to the SMCP. Then, in Section 3, we introduce the formulation of the SMCP and the model of the instances. In Section 4, we describe the algorithmic proposal and detail its different components. In Section 5, we present some advanced strategies to improve the efficiency of the proposed algorithm. In Section 6, we describe the experiments performed to evaluate the proposal and discuss the results. Finally, we identify some threats to validity in Section 7 and present our conclusions in Section 8.

2. Related work

As far as we know, the first attempt to deal with the SMCP was presented in Mancoridis et al. (1998), where the authors introduced a new objective function named Modularization Quality (MQ). This function calculates the quality of a solution based on a balance between coupling and cohesion. With the goal of accelerating the computation of the objective function, Mitchell and Mancoridis (2002b) described two variants of MQ, denoted as BasicMQ and TurboMQ. These metrics have been extensively used in the related literature (see, for instance, Huang et al. (2017), Praditwong (2011), and Prajapati and Chhabra (2018)).

Other approaches in the state-of-the-art literature have considered a multi-objective perspective to evaluate the quality of solutions for the SMCP. Praditwong et al. (2010) described two different proposals: Equal-size Cluster Approach (ECA) and Maximizing Cluster Approach (MCA). Both of them consider 5 different objectives, being MQ one of them. ECA and MCA were further analyzed by Barros (2012), conducting a thorough experimentation to determine whether it is better to use MQ as one of the objective functions or not (considering in the latter case the Evaluation Metric Function (EVM)). More recently, new proposals with a multi-objective approach were introduced (Mkaouer et al., 2015; Kumari and Srinivas, 2016). Similarly, a recent work proposed the utilization of different metrics in a cooperative clustering technique to improve software modularization results (Naseem et al., 2013).

A different approach was described by Bavota et al. (2012), where a software engineer is involved in the decision process. Specifically, the expertise of the engineer is considered as a way of evaluating the quality of the solutions found, making the remodularization process an interactive procedure. Similarly, Chong and Lee (2017) proposed the use of an automated approach to derive clustering constraints of the analyzed software based on graph theory analysis to help improve semisupervised software modularization.

Independently of the considered approach, the time needed to find and evaluate all possible solutions rapidly increases as the input grows (Brandes et al., 2007). Therefore, approximate search-based algorithms are more suitable for the SMCP than exact methods (Harman et al., 2012; Ramirez et al., 2019), since the former ones are usually faster. In this context, most of the recent search-based algorithms proposed for the family of SMCP problems are based on evolutionary approaches. In Praditwong (2011), results of different Genetic Algorithms with two encoding representations (Group Number Encoding and Grouping Genetic Algorithms) were compared. In Chhabra (2017) and Prajapati and Chhabra (2019), the authors proposed two algorithms based on Harmony Search. Other works have proposed the use of algorithms based on swarm intelligence, such as Artificial Bee Colony (Chhabra, 2018), Ant Colony Optimization (Varghese R et al., 2019), or Particle Swarm Optimization (Prajapati and Chhabra, 2018).

In contrast to the aforementioned approaches, some authors have proposed trajectory-based algorithms. In Pinto et al. (2014), the authors proposed an algorithm based on Iterated Local Search, which was favorably compared with other previous population-based algorithms. This approach was later improved by an algorithm based on LNS (Monçores et al., 2018). Despite these two proposals, trajectory-based metaheuristics, based on the improvement of a single solution, have been little explored in this context, and some authors have highlighted the need for a deeper and richer exploration of search-based alternatives to evolutionary strategies (Ramirez et al., 2019). Furthermore, we have identified that multistart procedures have not been explored to construct initial solutions to the SMCP. In addition, we miss a comprehensive criterion to define neighborhood structures for the problem, which are the core of trajectory-based search algorithms.

3. Background

In this paper, we deal with the SMCP using heuristic optimization strategies. In Section 3.1, we present the definition of the SMCP and some background information about the optimization strategies proposed to tackle the problem. Particularly, to overcome the limitations identified in the related literature, we propose the use of the GRASP multistart constructive procedure (see Section 3.2) and a VND improvement method to explore several neighborhoods (see Section 3.3). On the one hand, GRASP

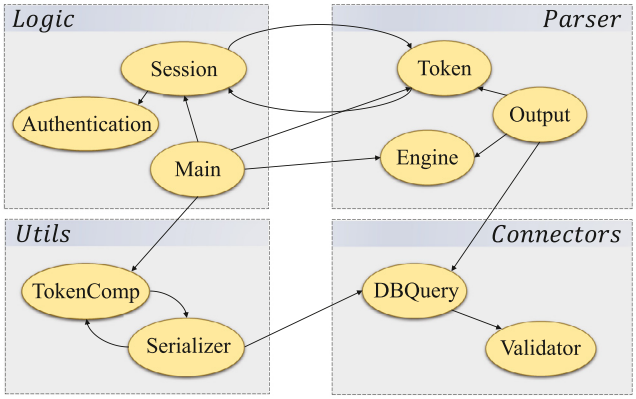


Fig. 1. Representation of a software project in an MDG. The components of the system are represented as vertices of a graph and the dependencies among components are represented as edges.

constructs promising solutions for the SMCP following a multi-start strategy finding a tradeoff between greediness and randomness. On the other hand, VND provides a method for exploring neighborhoods in a deterministic way. In this sense, a previous key step is the definition of complementary neighborhoods to be explored within the VND.

3.1. Software module clustering problem

The SMCP is an optimization problem where the objective is to find the best modular organization of a software project. That is, the aim is to form groups of components that are highly related among them but loosely connected to components from other modules. Frequently, the SMCP is modeled as a graph partitioning problem. In that model, the software project is represented by a directed weighted graph, referred in the literature as a Module Dependency Graph (MDG). Then, the SMCP consists of clustering the vertices of an MDG in groups (modules) to maximize its modularity. In Fig. 1, we show the modeling of a fictitious software project. There, we depict the classes and their relations in a dependency diagram, along with their organization in different software packages (i.e., *Logic, Parser, Utils,* and *Connectors*).

As it was aforementioned, we use the MQ objective function to evaluate the quality of the solutions of our approach, following other works in the area. This metric is evaluated by considering, first, the Modularization Factor (MF_i) of each module i. Specifically, given a solution x to the SMCP which has k modules, with $1 \leq i \leq k$, the Modularization Factor for each module i is computed as follows:

$$MF_i = \begin{cases} 0, & \text{if } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \varepsilon_i}, & \text{if } \mu_i > 0 \end{cases} \tag{1}$$

where μ_i is the sum of weights of the internal edges in module i and ε_i is the sum of weights of the edges between module i and other modules. It is worth mentioning that the number of modules k is not known *a priori* and different solutions for the same software project might have a different value of k.

Then, given a solution x represented by an MDG with k modules, the value of the objective function MQ is calculated by adding the Modularization Factor of each module in x. In mathematical terms:

$$MQ(x) = \sum_{i=1}^{k} MF_i. \tag{2}$$

Let us illustrate how we can compute the MQ value for the example shown in Fig. 1. First, we compute the value of MF for each of the 4 modules in the solution. In the case of the *Logic* module, $\mu_{Logic} = 2$, since there are two edges connecting components of the same module (lets call them internal edges) and $\varepsilon_{Logic} = 5$, because there are 5 edges connecting components of the module with other modules (lets call them external edges). Then, $MF_{Logic} = 2 \cdot 2/(2 \cdot 2 + 5) = 0.44$. We proceed in a similar way obtaining $MF_{Parser} = 4/(4 + 5) = 0.44$, $MF_{Utils} = 4/(4 + 2) = 0.67$, and $MF_{Connectors} = 2/(2 + 2) = 0.5$. Therefore, $MQ = MF_{Logic} + MF_{Parser} + MF_{Utils} + MF_{Connectors} = 2.05$.

It is worth mentioning that the value of each MF_i (and, therefore, of MQ) is directly proportional to the sum of internal dependencies and inversely proportional to the sum of external dependencies of the module i. Therefore, the MQ function benefits software organizations with high cohesion and low coupling.

Finally, the objective of the SMCP is to find a solution $x^\star$ among the set of all possible solutions X that maximizes Eq. (2). In mathematical terms:

$$x^\star = \arg\max_{x \in X}(MQ(x)). \tag{3}$$

3.2. Greedy Randomized Adaptive Search Procedure

The Greedy Randomized Adaptive Search Procedure (GRASP), originally presented by Feo and Resende (1995), is a multistart metaheuristic for combinatorial optimization problems. The general schema of GRASP is divided into two phases that are performed at each iteration of the algorithm: construction and improvement. The former builds a feasible solution based on a greedy function, which is dependant on the tackled problem. However, to ensure some diversity in the constructed solutions at each iteration, some randomness is introduced in the elections of the greedy function. Then, in the latter phase, the constructed solution is typically improved using a local search procedure. However, this procedure can be replaced by other search metaheuristics (Martín-Santamaría et al., 2022; Gil-Borrás et al., 2020). The best solution over all iterations is kept as the final result of the method.

The constructive method builds an initial solution from scratch, adding elements to a partial solution sequentially. At each iteration, the elements that can be added to the partial solution are ordered in a Candidate List (CL), from best to worst. From this CL, a subset of elements is picked to become part of a Restricted Candidate List (RCL), which is formed by the best candidates of the CL. Then, a candidate is chosen at random from the RCL and added to the partial solution. This RCL is a key component of the GRASP metaheuristic. The number of candidates that are picked from the CL depends on a threshold value (i.e., a user-defined parameter) which balances the tradeoff between greediness and randomness. Then, the candidates are evaluated with a greedy criterion, and only those with a score above the threshold are added to the RCL. Commonly, the threshold is calculated as follows:

$$th = c_{min} + \alpha \cdot (c_{max} - c_{min}) \tag{4}$$

where c_{min} and c_{max} are the minimum and maximum values, respectively, of the candidates in the CL according to the greedy criterion, and α is a search parameter in the range [0,1]. As it can be observed, if $\alpha = 0$, then the decision is completely random (all candidates from the CL will be part of the RCL); and if $\alpha = 1$, then the decision is completely greedy (only the best candidate of the CL is added to the RCL).

To summarize, the GRASP methodology is a metaheuristic that combines a construction phase with a local search procedure. The objective of this metaheuristic is to provide different promising initial solutions for the local search, allowing the latter to

explore different promising regions in the search space of the optimization problem. To fulfill this objective, the constructive method is guided by a greedy function (which ensures that the constructed solution is promising) that allows the selection of candidates that are not the best ones (providing some diversity to the set of initial solutions). As stated in Talbi (2009): "The larger is the variance between the solutions of the construction phase, the larger is the variance of the obtained local optima. The larger is the variance between the initial solutions, the better is the best found solution".

3.3. Variable neighborhood search

In local search procedures, an initial solution is iteratively improved by applying small perturbations. The common search concept for local search procedures is the definition of neighborhood structures. A neighborhood $N(x)$ is defined as the set of solutions that are close to a starting point x. More precisely, this set is defined as the set of solutions that can be reached by applying a particular movement (e.g., in the context of the SMCP, a movement could consist of deleting one vertex from one module and inserting the vertex in another module). Neighborhood structures are typically traversed with a heuristic procedure (local search) which finds the best solution in the neighborhood (i.e., the local optimum), that is, a solution that has better quality than all its neighbors. However, a local optimum for a neighborhood N_1 might not be a local optimum for another neighborhood N_2. This fact encourages the exploration of different neighborhoods.

Variable Neighborhood Search (VNS) was originally proposed by Mladenović and Hansen (1997) as a general framework for solving hard combinatorial and global optimization problems, becoming one of the most successful strategies in the area (Cavero et al., 2022; Lozano-Osorio et al., 2022). The basic idea behind this metaheuristic is the exploitation of the fact that the exploration of different neighborhoods might generate different local optima and that the global optimum for a problem must be a local optimum in all possible neighborhoods by definition. Therefore, exploring different neighborhoods results in solutions that are closer to the global optimum.

The Variable Neighborhood Descent (VND) strategy is a particular version of VNS which proposes the definition of a set of different neighborhoods for a particular solution and its exploration in a deterministic way. First, a set of neighborhoods for the problem is defined. Then, the first neighborhood of the initial solution is explored. If the neighborhood does not contain any solution better than the current one, VND proceeds to explore the next neighborhood. When a better neighbor is found within any neighborhood, the current solution is replaced by that neighbor and VND starts the exploration of the first neighborhood again. This process is repeated until the current solution is a local optimum in all the defined neighborhoods.

As it can be derived from the description of the VND procedure, the order of the neighborhoods is important, since the first neighborhood will usually be explored more times than the last one. A common strategy is to order the neighborhoods in increasing order of complexity, so that large neighborhoods are explored fewer times than small neighborhoods, reducing the time consumption of the algorithm.

4. Algorithmic proposal

In this section, we describe the proposed algorithm, which is based on a GRASP methodology (Feo and Resende, 1995). We illustrate our approach in Algorithm 1. As it can be observed, the proposed algorithm receives two input parameters: the MDG of the software project that must be modularized and the maximum number of iterations to be performed. First, the size of the MDG is reduced in a preprocessing step to obtain an equivalent MDG (step 2), which is described in Section 4.1. Then, the procedure enters in a loop (steps 5 to 10) for the predefined number of iterations *maxIter*. In step 6, an initial solution for this iteration is constructed. As it is customary in a GRASP schema, the construction phase is based on a greedy function, which we detail in Section 4.2. Then, the initial solution is improved in step 7. In this case, the typical improvement phase based on a local search procedure is replaced by a metaheuristic procedure (Variable Neighborhood Descent, VND), which we describe in Section 4.3. Therefore, in each iteration of the GRASP algorithm, we build a solution (using a greedy, randomized, and adaptive procedure) and then improve it with a VND method. Finally, the best solution found during the whole process is returned as the best solution found (step 11).

Algorithm 1 Algorithmic proposal

```
1:  procedure GRASP(MDG,maxIter)
2:      MDG' ← PREPROCESS(MDG)                          ▷ Section 4.1
3:      bestSolution ← ∅
4:      iter ← 0
5:      while iter ≤ maxIter do
6:          s ← CONSTRUCT(MDG')                         ▷ Section 4.2
7:          s' ← IMPROVE(s)                             ▷ Section 4.3
8:          bestSolution ← SELECT_BEST_SOLUTION(bestSolution,s')
9:          iter ← iter + 1
10:     end while
11:     return bestSolution
12: end procedure
```

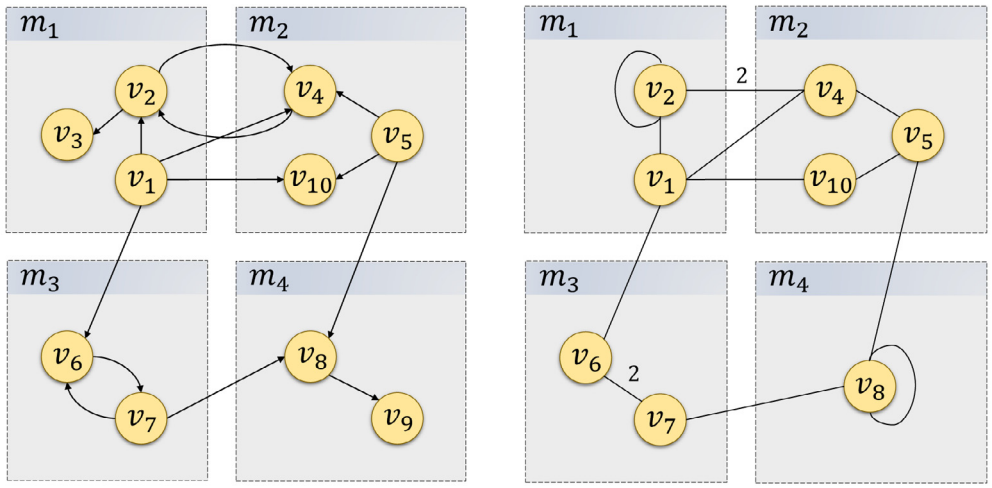
4.1. Preprocessing reduction procedure

Given an MDG that represents a solution to a software project, Köhler et al. (2013) proposed the following theorem, which is used as the foundation of the reduction procedure:

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

Based on the previous theorem, Monçores et al. (2018) proposed a method to reduce the size of the input MDG, obtaining an equivalent MDG of smaller size. In this method, the directions of the edges are firstly eliminated, transforming the MDG into an undirected weighted graph. Then, for each pair of adjacent vertices (u, v) connected by more than one edge, we replace all connecting edges with a single one, which weight is calculated as the sum of weights of the removed edges. In addition, each vertex $u \in V$ with a single adjacent vertex v is deleted from the MDG and a self-loop edge is added to v, with the same weight as the edge that connected u and v. This step is repeated until no vertices with a single connection remain in the graph. The resulting graph is formally defined as a 3-tuple (V, E, W), being V the set of vertices, E, the set of edges, and W, the weight associated to each edge. Notice that the MQ of the reduced graph is equal to the MQ of the original MDG (Köhler et al., 2013). Therefore, a partition of the reduced MDG can be used to produce a solution for the SMCP.

Let us illustrate how to use this method with an example. In Fig. 2(a), we depict a simplified version of the MDG depicted in Fig. 1. Specifically, modules *Logic*, *Parser*, *Utils*, and *Connectors* have been renamed as m_1, m_2, m_3, and m_4, respectively. Similarly, the label of each vertex has been replaced with a number. For



(a) Solution obtained after relabeling the module and component names.

(b) Solution obtained after applying the preprocessing reduction procedure.

Fig. 2. Resulting MDG after applying the preprocessing reduction procedure to the graph of Fig. 1.

example, in module m_1, components *Main*, *Session*, and *Authentication* have been relabeled as v_1, v_2, and v_3, respectively. In Fig. 2(b), we represent the MDG after applying the reduction procedure. As it can be observed, the directions of the edges have been removed, edges connecting the same pair of vertices have been joined and weighted (the weight of not numbered edges is equal to 1), and vertices with a single connection have been eliminated and modeled as self-loop edges. The resulting graph has two vertices and two edges less than the original MDG.

4.2. Construction phase

In the context of the SMCP, we start from an empty solution (there are not any vertices or modules). First, a vertex is randomly selected and included in a new module. Then, we propose a criterion that evaluates the 'urgency' of incorporating each vertex to the solution under construction. This criterion is based on a balance between the number of adjacent vertices that are already included in the solution and the number of adjacent vertices which are not yet included. Following the randomized greedy strategy proposed by GRASP, one of the most 'urgent' vertices, according to the previous criterion, is selected and added to the solution. The vertex is placed in an already existent module or in a new empty one, depending on which one maximizes the objective function for the current partial solution. The process is repeated until all vertices are included in the solution.

Let us introduce some basic terminology to properly define the greedy criterion used to evaluate each candidate vertex. Given a preprocessed graph $MDG = (V, E, W)$, where each edge $(u, v) \in E$ (with $u, v \in V$) has associated a weight $w_{u,v} \in W$, we define $S = (S_1, S_2, \ldots, S_k)$ as a partial solution, where S_i (with $1 \leq i \leq k$) identifies the set of vertices in module i. The set of vertices not included in S is then $Q = V \setminus \cup_{i=1}^{k} S_i$. Then, it trivially holds that a partial solution verifies that $\sum_{i=1}^{k} |S_i| < |V|$.

Then, the greedy function g which evaluates the urgency of every candidate vertex $u \in Q$ is defined as follows:

$$g(u) = \underbrace{\max\left(\sum_{v \in S_i} w_{u,v},\ 1 \leq i \leq k\right)}_{\text{(a)}} - \underbrace{\sum_{v \in Q} w_{u,v}}_{\text{(b)}}. \tag{5}$$

Particularly, g computes, for every vertex u: (a) the maximum number of adjacent vertices to u that are included in a particular module S_i of the partial solution; and (b) the number of adjacent vertices to u which are still not part of the solution. Then, it returns the value obtained as the difference between (a) and (b).

The construction phase of the GRASP procedure is presented in Algorithm 2. The procedure receives an MDG representing the instance. It starts with an empty solution (step 2), and adds all candidate vertices from the MDG to a data structure named Candidate List (CL) (step 3). Then, it calculates the value of the search parameter α as a random value in [0, 1] (step 4) which will be used in every iteration of the construction. Next, the procedure enters in a loop, while there are vertices in the CL. In each iteration of the loop, the procedure evaluates the candidate vertices in the CL with the greedy function g and calculates a threshold *th* (step 6) depending on the value of the best candidate, $\max(g(u))$, the worst candidate, $\min(g(u))$, and the α parameter, as introduced in Section 3.2. Next, a Restricted Candidate List (RCL) is constructed with the vertices which score a value of g greater than *th* (step 7). Then, one vertex is selected at random from the RCL and added to the module which maximizes the value of the objective function for the current partial solution (step 8). Finally, the assigned vertex is removed from the CL. This process is repeated until all candidate vertices have been assigned. When the partial solution is completed, it is returned (line 12).

Algorithm 2 Constructive procedure

```
1:  procedure CONSTRUCTIVE(MDG)
2:      solution ← ∅
3:      CL ← V
4:      α ← RANDOM(0.0,1.0)
5:      while CL ≠ ∅ do
6:          th ← min(g(u)) + α · (max(g(u)) − min(g(u))) | u ∈ CL
7:          RCL ← {u ∈ CL | g(u) ≥ th}
8:          vertex ← RANDOM(RCL)
9:          solution ← ADD(vertex, solution)
10:         CL ← CL \ vertex
11:     end while
12:     return solution
13: end procedure
```

It is worth mentioning that the construction process involves two decisions at each iteration of the construction: the vertex that will be introduced next in the solution and the module where the selected vertex will be included in. The ADD procedure, introduced at step 9 of Algorithm 2, determines the second decision by

trying the insertion of the selected vertex, not only in any existent module, but also in an empty new module. In addition, to foster the creation of promising new modules, the procedure tries the insertion of the selected vertex in a new module together with each of the vertices already assigned to the partial solution, since a single-vertex module does not have internal edges its *MF* value would be zero. After trying the aforementioned insertions, the best one (in terms of MQ) is performed.

4.3. Variable neighborhood descent

In this paper, we propose the use of Variable Neighborhood Descent (VND) as the improvement phase of the GRASP procedure. In Algorithm 3, we present the pseudocode of our VND procedure. The method receives a solution (x) and the maximum number of neighborhoods to explore (k_{max}). The variable k determines the neighborhood to be explored in the next iteration, and it is initialized to 1 (step 2). Then, the procedure iterates (steps 3 to 11) until the last neighborhood (k_{max}) is explored without finding an improvement. In each iteration, a local search procedure explores the neighborhood N_k and returns the best solution found in that neighborhood (step 4). If the quality of that solution is better than the quality of the previous best overall solution, then the best solution is updated (step 6) and k is set to 1 (step 7). Otherwise, the value of k is simply increased (step 9). Notice that whenever a new best solution is found, we need to reset k to 1 to guarantee the complete exploration of the considered neighborhoods, since the neighborhood structure depends on both the initial solution for starting the search and the defined movement that can be applied. Then, a new best solution in one of the neighborhoods represents a new starting point for the rest of the neighborhoods. Consequently, the solution returned by the method is a local optimum within all the neighborhoods explored.

Algorithm 3 VND

```
1:  procedure VND(x, k_max)
2:      k ← 1
3:      while k ≤ k_max do
4:          x' ← LocalSearch(N_k, x)
5:          if MQ(x') > MQ(x) then
6:              x ← x'
7:              k ← 1
8:          else
9:              k ← k + 1
10:         end if
11:     end while
12:     return x
13: end procedure
```

In this paper, we propose different neighborhoods for the SMCP to be explored with the VND procedure introduced in Algorithm 3. Particularly, we have found that any neighborhood in the context of the SMCP can be classified in one of the following categories, depending on the type of movement performed:

- **Neighborhoods defined by movements which do not alter the number of modules**. The exploration of the neighborhoods defined by this type of movement intends to move vertices between existent modules (e.g., insertion of a vertex in a new module, interchange of two vertices in different modules, etc.).
- **Neighborhoods defined by movements which increase the number of modules**. The exploration of the neighborhoods defined by this type of movement intends to create new modules in the solution (e.g., split of a module in two; extraction of a vertex from a module, etc.).
- **Neighborhoods defined by movements which reduce the number of modules**. The exploration of the neighborhoods defined by this type of movement intends to move all vertices from an existent module to other modules, thus reducing the number of modules in the solution (e.g., merge of two modules, destruction of a module, etc.).

Specifically, we propose four different neighborhoods for the SMCP: two in the category which does not alter the number of modules (N_1 and N_2), one in the category which increases the number of modules (N_3), and one in the category which reduces the number of modules (N_4). Next, we describe each of them in detail. The order of the neighborhoods is considered as a search parameter of the algorithm, and it will be explored in Section 6.2.2.

4.3.1. Insert

The first neighborhood (N_1) is defined by an insertion movement, which is a classical move in graph partitioning problems. An insertion consists of moving a vertex from one module to another one. We denote this operation as $x' \leftarrow Insert(x, v, m, m_j)$, where x represents the solution prior to the movement, x' is the solution after performing the insertion of vertex v, m represents the original module that contained v before the move, and m_j is the module where v is inserted. In Fig. 3, we represent an example of the *Insert* operation of the vertex v_1, originally belonging to module m_1 in x, into module m_2, obtaining the new solution x'. Notice that the vertices affected by the move are represented in orange color with a dashed border line. Then, given a solution x, we define the neighborhood N_1 as the set of solutions that can be reached by applying the *Insert* operation to any vertex v in the solution and any other module but its current one. More formally:

$$N_1(x) = \{x' \leftarrow Insert(x, v, m, m_j) : \\ \forall\, v \in V,\ \forall\, m_j \in M \,/\, v \in m,\ m \neq m_j\},$$

where M is the set of all modules of the solution and $1 \leq j \leq |M|$. Given a graph with $|V|$ vertices and $|M|$ modules, the size of the complete neighborhood is $|V| \cdot (|M| - 1)$.

4.3.2. Swap

The second neighborhood (N_2) is defined by a swap movement, which is also a classical move in graph partitioning problems. A swap consists of selecting two vertices from different modules and interchanging them. We denote this operation as $x' = Swap(x, v_i, v_j, m_k, m_l)$, where x represents the solution prior to the movement, x' is the solution after performing the swap, v_i is a vertex belonging to module m_k, and v_j is a vertex belonging to module m_l. In Fig. 4, we represent an example of the $Swap$ operation of vertices v_1 and v_4 in x obtaining the new solution x'. Notice that the affected vertices are represented in orange color with a dashed border line.

Then, given a solution x, we define the neighborhood N_2 as the set of solutions that can be reached by applying the $Swap$ operation to any pair of vertices v_i, v_j belonging to different modules. More formally:

$$N_2(x) = \{x' \leftarrow Swap(x, v_i, v_j, m_k, m_l) : \forall v_i, v_j \in V \\ / \ v_i \in m_k \wedge v_j \in m_l \wedge m_k, m_l \in M \wedge m_k \neq m_l\},$$

where M is the set of all modules of the solution, $1 \leq i, j \leq |V|$, and $1 \leq k, l \leq |M|$. For a graph with $|V|$ vertices, in the worst case, the size of the complete neighborhood is $|V| \cdot \frac{(|V|-1)}{2}$. Notice that the size of the neighborhood is actually a bit smaller since vertices are not swapped with vertices in the same module.

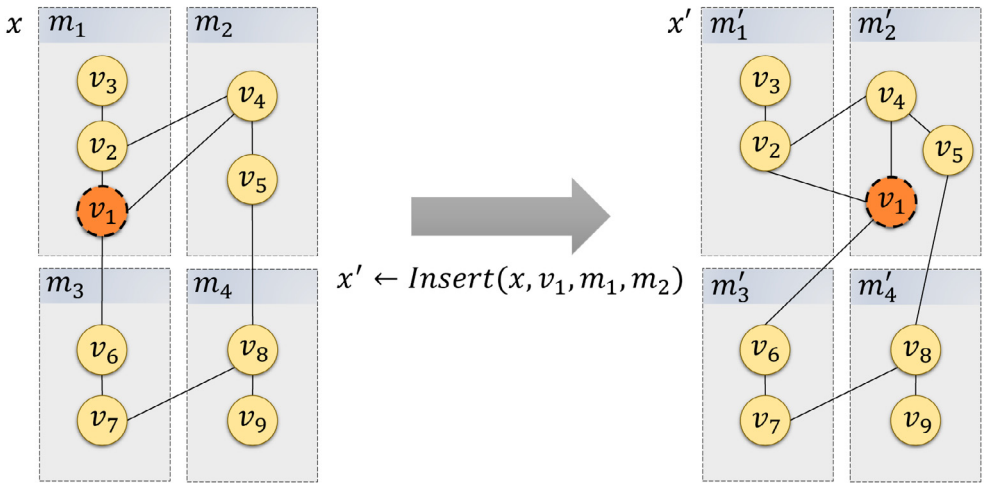


Fig. 3. Example of the *Insert* operation of the vertex v_1 in module m_2. On the left side, we present the solution x before applying the *Insert* operation. On the right side, we depict the solution x', obtained after inserting vertex v_1 into module m_2. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

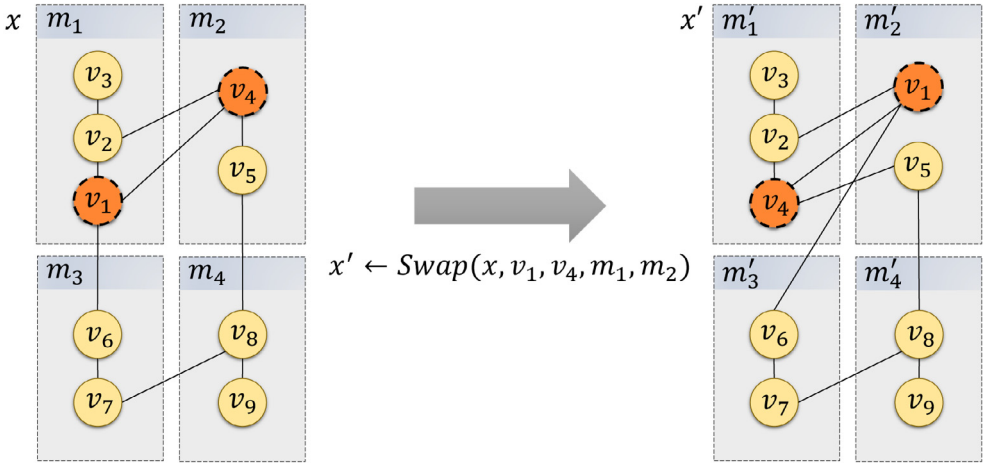


Fig. 4. Example of the *Swap* operation of vertices v_1 and v_4. On the left side, we present the solution x before applying the *Swap* operation. On the right side, we depict the solution x', obtained after applying the *Swap* operation in solution x. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.3.3. Extract

The third neighborhood (N_3) is defined by two different extraction operations, where vertices are extracted from their current module and inserted in a newly created one. The first extraction movement involves two vertices of the solution, and it is denoted as $x' = Extract\text{-}2(x, v_i, v_j)$, where x represents the solution prior to the movement, x' is the solution after performing the extraction of v_i and v_j, and v_i and v_j are two vertices in V. In Fig. 5, we represent the *Extract*-2 operation of vertices v_1 and v_2, which belong to module m_1, to a new module m_5. Notice that the vertices affected by the operation are represented in orange color with a dashed border line. In this case, both extracted vertices belong to the same original module (m_1). However, they could have been assigned to different modules.

The second extraction movement involves three vertices of the solution and it is denoted as $x'' = Extract\text{-}3(x, v_i, v_j, v_k)$, where x represents the solution prior to the movement, x' is the solution after performing the extraction of v_i, v_j, and v_k, and v_i, v_j, and v_k are three vertices in V. In Fig. 6, we represent the *Extract*-3 operation of vertices v_1, which belongs to module m_1, v_2, which also belongs to module m_1, and v_6, which belongs to module m_3, to a new module m_5. Vertices affected by the operation are represented in orange color with a dashed border line.

Then, given a solution x, we define the neighborhood N_3 as the set of solutions that can be reached by applying either the *Extract*-2 or the *Extract*-3 operation. More formally:

$$N_3(x) = \{x' \leftarrow Extract\text{-}2(x, v_i, v_j) \cup x'' \leftarrow Extract\text{-}3(x, v_i, v_j, v_k) : \forall\, v_i, v_j, v_k \in V \,/\, v_i \neq v_j \neq v_k\}.$$

For a graph with $|V|$ vertices, the size of the complete neighborhood is $(|V| \cdot (|V| - 1)) + (|V| \cdot (|V| - 1) \cdot (|V| - 2))$. Notice that other extraction movements involving more than three vertices could be defined. However, we limit the number of extractions to three in order to maintain a reduced number of neighborhoods to speed up the analysis of the code.

4.3.4. Destroy

The last neighborhood (N_4) is defined by an operation which eliminates one of the existent modules and inserts its vertices, one by one, into other modules. We denote this operation as $x' \leftarrow Destroy(x, m, O, D)$, where x represents the solution prior to the movement, x' is the solution after the movement, m is the module to be removed, O is the ordered list of vertices that belong to m, and D is the ordered list of destination modules where the vertices in O will be inserted. Therefore, $O = \{v_1, v_2, \ldots, v_k\}$ and $D = \{m_1, m_2, \ldots, m_k\}$. Notice that there is an order correspondence between the elements in O and the elements in D in such a way that the first vertex in O is inserted in the first module in D, and so on. In Fig. 7, we represent the *Destroy* operation of the

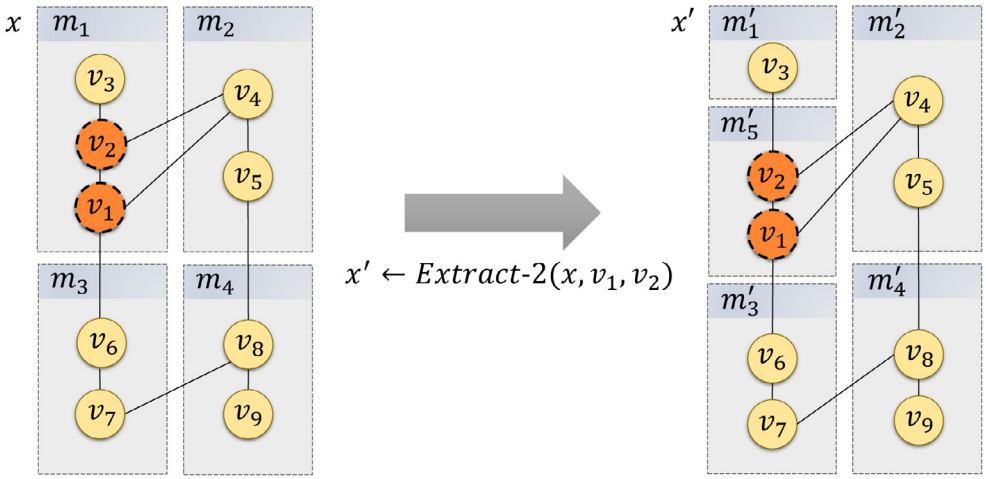


Fig. 5. Example of the *Extract*-2 operation of vertices v_1 and v_2. On the left side, we present the solution x before applying the *Extract*-2 operation. On the right side, we depict the solution x', obtained after applying the *Extract*-2 operation in solution x. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

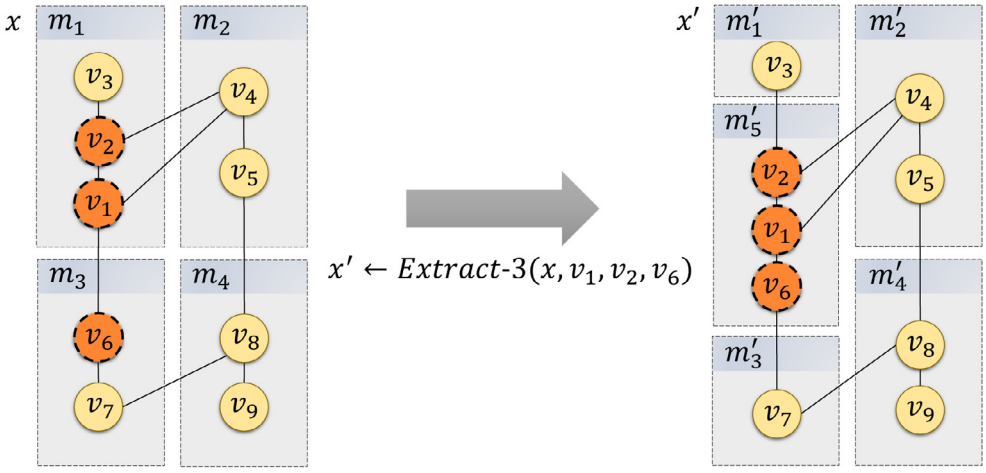


Fig. 6. Example of the *Extract*-3 operation of vertices v_1, v_2, and v_6. On the left side, we present the solution x before applying the *Extract*-3 operation. On the right side, we depict the solution x', obtained after applying the *Extract*-3 operation in solution x. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

module m_3 and the reinsertion of its vertices v_6 and v_7 in modules m_1 and m_4, respectively. Notice that the vertices affected by the operation are represented in orange color with a dashed border line.

Then, given a solution x, we define the neighborhood N_4 as the set of solutions that can be reached by applying the *Destroy* operation over any module in the solution. More formally:

$$N_4(x) = \{x' \leftarrow Destroy(x, m, O, D_j) : \forall\, m \in M \,/\, |O| = |D|, m \notin D\},$$

where M is the set of all modules of the solution, O contains all vertices in the removed module m, and D_j is a particular list of modules, where every item of the list can be any module in M (including repetitions) but m.

For a graph with $|M|$ modules and $|V|$ vertices, in the worst case, the size of the complete neighborhood would be $|M| \cdot (|M| - 1)^{|D|}$, with $|D| \leq |V|$.

5. Advanced strategies

In this section, we present some strategies to improve the efficiency of the proposed method. In Section 5.1, we describe an efficient evaluation of the objective function. Then, in Section 5.2, we describe a strategy to reduce the size of the explored neighborhoods.

5.1. Efficient evaluation after a movement

The objective function studied in this work (MQ) is computed as the sum of the Modularization Factor (*MF*) of each module i. Therefore, when performing any move operation, the *MF* of those modules which are not altered after the move does not change. Then, if we evaluate a given solution and store the *MF* values of its modules, we only need to evaluate and update the *MF* value of the modules affected by the move. For instance, in Fig. 8, we represent the insertion of vertex v_1 from module m_1 into module m_2. We depict the solution before and after the aforementioned move. In both figures, we note the *MF* of each module and we highlight in bold type font the *MF* values that need to be computed.

The objective of this strategy is to save time when evaluating a solution after a move. In the example of Fig. 8, we avoided the reevaluation of MF in half of the cases. It is easy to see that the benefits of this strategy are usually increased with respect to the increase in the number of modules.

5.2. Reduction of the size of the neighborhoods

We propose here a general strategy to reduce the size of the neighborhoods introduced in Section 4.3, avoiding the

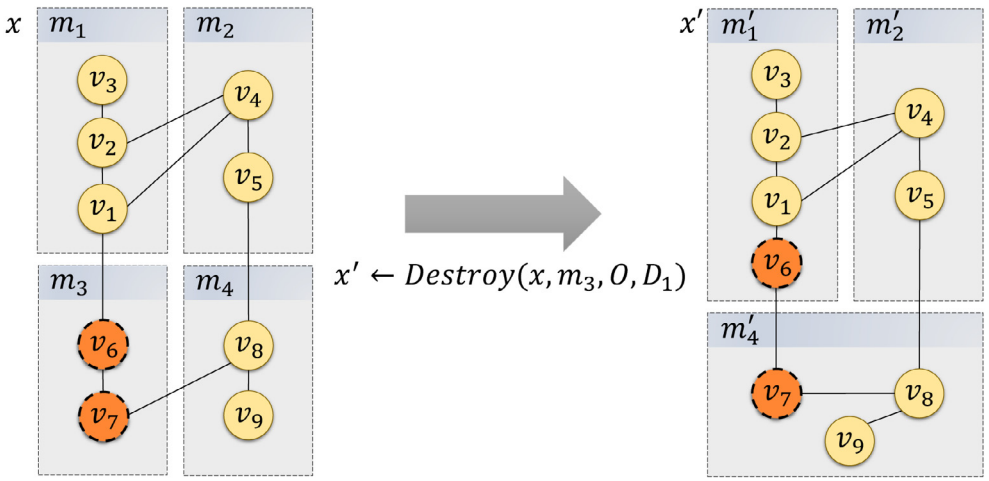


Fig. 7. Example of the *Destroy* operation of module m_3 from solution x. On the left side, we present the solution x before applying the *Destroy* operation. On the right side, we depict the solution x', obtained after destroying module m_3 from solution x and reinserting its vertices v_6 and v_7 in modules m_1 and m_4, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

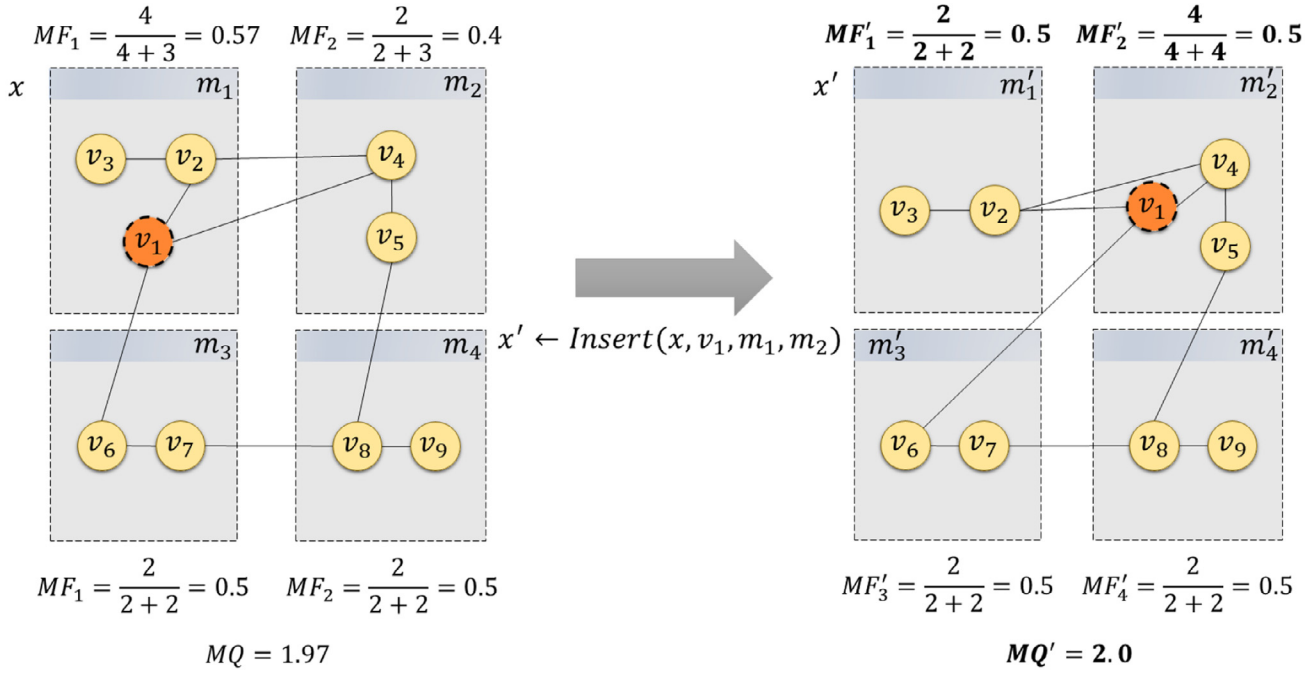


Fig. 8. Efficient evaluation of a solution x' after an *Insert* operation of the vertex v_1 in module m_2, in a given solution x that had been previously evaluated. On the left side, we depict the solution x. On the right side, we present the solution x', obtained after performing the insertion. The affected calculations of MF and MQ are highlighted in bold font.

exploration of unpromising solutions. This strategy is based on the following theorem introduced in Köhler et al. (2013):

Theorem 2. *Let $G(V, E)$ be an input MDG to the SMCP and let $A(v)$ be the set of adjacent vertices of $v \in V$. Suppose that in the optimal solution of the SMCP, all vertices in $A(v)$ are assigned to at most two different modules m_i and m_j. Then, v is either assigned to m_i or to m_j.*

In other words, in the optimal solution of the SMCP, every vertex will always be placed in a module where there is at least one of its adjacent vertices.

Therefore, we can reduce the size of any proposed neighborhood by considering only the solutions obtained as a result of move operations which ensure that the affected vertices are placed in modules where there exists at least one of their adjacent vertices.

6. Experimental results

In this section, we describe the results obtained in the experiments performed in this research. First, in Section 6.1, we describe the dataset used in the experiments. Then, in Section 6.2, we illustrate the contribution of our proposal by performing some preliminary experiments. Next, in Section 6.3, we compare the results of the proposed approach with the best previous state-of-the-art method. Finally, we discuss the obtained results in Section 6.4.

6.1. Dataset

To evaluate the proposed approach, we use a set of 124 instances introduced in a previous work (Monçores et al., 2018). In that work, the instances were divided into four different categories according to their size: 64 small instances (up to 68

Table 1
Instances of the preliminary dataset. For each instance, we present the number of vertices, the number of edges, and the density of the graph.

Instance	$\|V\|$	$\|E\|$	D
regexp	14	20	10.99%
jxlsreader	27	73	10.40%
star	36	89	7.06%
cia	38	185	13.16%
net-tools	48	183	8.11%
wu-ftpd-1	50	230	9.39%
joe	51	540	21.18%
jscatterplot	74	232	4.29%
mod_ssl	135	1095	6.05%
gae_plugin_core	139	375	1.95%
nmh	198	3262	8.36%
jtreeview	320	1057	1.04%
lwjgl-2.8.4	453	1976	0.97%
apache_ant_taskdef	626	2421	0.62%

vertices), 29 medium instances (from 69 to 182 vertices), 18 large instances (from 190 to 377 vertices), and 13 very large instances (from 413 to 1161 vertices). From this dataset, we have selected 14 instances at random to perform the preliminary experiments, following a similar distribution in terms of size: 8 instances from the small category, 2 instances from the medium category, 2 instances from the large category, and 2 instances from the very large category. In Table 1, we show the instances that we randomly selected for the preliminary dataset. Particularly, we present the number of vertices, the number of edges, and the density of each instance. As it can be observed, the smallest instance has 14 vertices and 20 edges, while the largest instance has 626 vertices and 2421 edges.

6.2. Preliminary experiments

In this section, we illustrate the contribution of the strategies introduced in this paper and experimentally adjust some parameters of our algorithm. Each experiment is presented in a different subsection starting with the research question that we are trying to answer with that experiment.

6.2.1. Contribution of each neighborhood

This experiment is devoted to answer the following research question: what is the contribution of each proposed neighborhood to the quality of the solutions?

To answer the previous question, we explored their performance in isolation. Particularly, for each neighborhood, we designed a multistart procedure combining the GRASP constructive algorithm with a local search which explores each of the neighborhoods separately, following a first improvement strategy. Then, each of the four methods was run for 100 iterations per instance, and we reported the average improvement of the local search with respect to the initial solution in the 100 iterations. To perform a fair comparison, we ensured that the exploration of any of the neighborhoods started from the same initial solution in each of the iterations. With this experiment, we are able to see which local search procedure (i.e., which neighborhood) produces a larger benefit given a particular initial solution provided by GRASP.

In Table 2, we report, for each neighborhood: the average improvement of the solutions compared to the initial solutions (△ Avg. MQ), the average execution time consumed (CPUt (s)), the average effect-size of the improvement compared to the initial solutions (ES), and the average number of modules in the final solutions (Avg. $|M|$). Notice that we report the increment of MQ, instead of the final MQ, to better determine the contribution of each neighborhood. Additionally, we report the effect-size, which represents the magnitude of improvement in a pairwise comparison, calculated using the Vargha–Delaney test (Vargha and Delaney, 2000; Arcuri and Briand, 2014). A result equal to 50% means that the exploration of a given neighborhood had no effect on the quality of the final solution. In contrast, a result equal to 100% means that the exploration of a given neighborhood achieved better results in all cases.

Table 2
Contribution of the exploration of each neighborhood to the quality of the solutions provided by the constructive procedure in the reduced dataset.

Neighborhood	△ Avg. MQ	CPUt (s)	ES	Avg. $\|M\|$
N_1	0.6258	<0.01	95.47%	47.76
N_2	0.0207	<0.01	52.25%	48.76
N_3	0.4663	0.04	92.88%	51.99
N_4	0.1133	0.01	66.32%	44.14

Table 3
Comparison of the results obtained with the GRASP+VND algorithm with different orderings of the proposed neighborhoods in the reduced dataset.

Order	Avg. MQ	Dev. (%)	# Best	CPUt (s)	Avg. $\|M\|$
N_1, N_3, N_4	15.9545	<0.01%	12	6.23	47.21
N_1, N_4, N_3	15.9549	<0.01%	11	5.78	47.21
N_3, N_1, N_4	15.9537	0.02%	9	10.36	47.50
N_3, N_4, N_1	15.9541	<0.01%	10	15.37	47.36
N_4, N_1, N_3	15.9542	<0.01%	9	9.33	47.00
N_4, N_3, N_1	15.9641	<0.01%	10	15.41	47.50

As it can be observed, the N_1 (defined by the operation *Insert*) and N_3 (defined by the operation *Extract*) neighborhoods were the ones that further improved the solutions, with an average improvement of 0.63 and 0.47, respectively, and an effect-size greater than 90%. However, the extract neighborhood was also the most time consuming strategy. The N_4 neighborhood (defined by the operation *Destroy*) offered a modest average improvement of 0.11, with an effect-size of 66.32%. Finally, N_2 (defined by the operation *Swap*) contributed to a poor quality improvement of 0.02, with a negligible effect-size close to 50%. Therefore, we decided to configure the VND approach with only the three neighborhoods that contributed the most: N_1, N_3, and N_4.

6.2.2. Order of the neighborhoods

This experiment is devoted to answer the following research question: what is the best order of the neighborhoods for the VND procedure?

VND traverses a predefined list of neighborhoods in a deterministic way. After the exploration of each neighborhood, if an improvement has been made, the method returns to the first neighborhood of the list and starts exploring each neighborhood again. Otherwise, VND changes to the next neighborhood of the list and explores it. This process is repeated until all the neighborhoods have been explored consecutively without improvement. Therefore, the order in which neighborhoods are arranged is important, since the first neighborhood is usually explored more times than the last one.

In this experiment, we study all possible orders of the three neighborhoods selected in the previous experiment (N_1, N_3, and N_4) within the VND. Particularly, we have run each VND configuration for 100 iterations per instance, ensuring that all procedures start from the same initial solutions. In Table 3, we report, for each configuration: the average quality of the best solutions found per instance (Avg. MQ), the average deviation to the best solution found in the experiment (Dev. (%)), the number of best solutions found (# Best), the average CPU time per instance (CPUt (s)), and the average number of modules in the best solutions (Avg. $|M|$). As it can be observed, there was not much difference among the different VND configurations in terms of quality. However, some configurations resulted to be considerably faster than others. Among all the VND configurations, we selected the ordering N_1, N_3, N_4 for the following experiments, since it found the best solution for 12 of the instances in the reduced dataset in a reasonable amount of time.

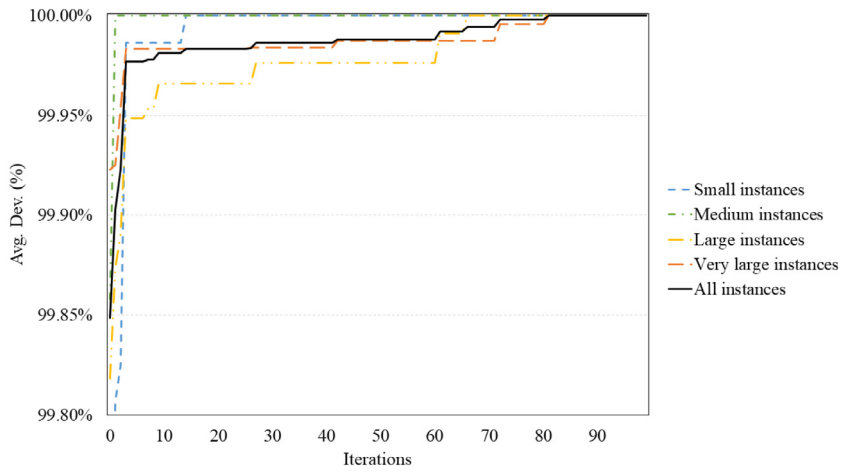


Fig. 9. Average deviation (%) of the best solution found by the algorithm at each iteration to the best solution found after 100 iterations for each instances of the preliminary dataset.

6.2.3. Stopping criterion

This experiment is devoted to answer the following research question: what is the best stopping criterion for the GRASP+VND method in this particular problem?

Since the proposed algorithm is a multistart procedure, the stopping criterion of this proposal is based on the number of iterations. Therefore, we have performed a preliminary experiment to determine the minimum number of iterations that we should let the algorithm run (i.e., avoiding iterations which produce a marginal improvement). In particular, we have executed the algorithm with the configuration set in previous preliminary experiments (i.e., three neighborhoods ordered as follows: N_1, N_3, N_4) for 100 iterations per instance, reporting the quality of the best solution found at the end of each iteration.

In Fig. 9, we represent the average deviation, in percentage, from the best solution found at each iteration to the best solution found for all instances (black solid line). Moreover, we depict the average deviation for the small (blue dashed line), medium (green dash–dot line), large (yellow long dash–dot–dot line), and very large (orange long dash line) instances of the preliminary dataset. As it can be observed, the average quality for the small and medium instances is not further improved after iteration 15. In contrast, the average quality of solutions for large and very large instances still increases until iteration 81. However, this improvement is relatively small ($<0.05\%$) for all instances after the first 10 iterations. To find a balance between quality and CPU running time, we have selected to stop our procedure after 20 iterations per instance.

6.2.4. Influence of the advanced strategies

This experiment is devoted to answer the following research question: what is the influence of the advanced strategies in the performance of the algorithm?

In order to test the impact of the advanced strategies described in Section 5, we compare here the results obtained with the algorithm with and without them. To this end, we executed the proposed algorithm with three different configurations. First, we ran the algorithm with none of the advanced strategies (denoted as None in Table 4). Then, we ran the algorithm employing the efficient evaluation of movements described in Section 5.1 (denoted as EE in Table 4). Finally, we ran the algorithm incorporating the efficient evaluation of movements and the reduction of the size of the neighborhoods described in Section 5.2 (denoted as EE+RN in Table 4). Notice that the three configurations inherited the settings described in previous sections. That is, the stopping criterion was set at 20 iterations and the neighborhoods were explored in the following order: N_1, N_3, N_4. Since the differences between the variants are related to the improvement phase, the three variants started from the same initial solutions for each instance.

Table 4
Comparison of the efficiency improvement of different advanced strategies.

Method	Avg. MQ	Dev. (%)	# Best	CPUt (s)	Avg. $\|M\|$
None	15.9528	0.01%	12	147,632.44	47.36
EE	15.9528	0.01%	12	182.31	47.36
EE+RN	15.9519	<0.01%	11	1.71	47.71

In Table 4, we report the results of this experiment. As can be observed, all configurations were able to reach solutions of similar quality. However, the computational time differs greatly between them. Particularly, the implementation of an efficient evaluation of movements decreased the computational average time from 147,632.44 s to only 182.31 s (saving 99.88% of the original time). When the efficient evaluation is combined with the reduction of the size of the neighborhoods, the computational time is further reduced to 1.71 s (saving an additional 99.06% of the 182.31 s).

Notice that the differences in terms of quality between the first two configurations and the third one are explained by the difference in size of the explored neighborhoods. Particularly, since we are considering a reduced neighborhood and a first improvement exploration strategy (i.e., the first move which improves the current solution is performed), the search pattern is different. Therefore, the solutions found by the exploration of those neighborhoods might be also different. This fact explains the differences observed in the results obtained by EE+RN in Table 4 (a slightly smaller deviation, but one best solution less).

6.3. Comparison with the state-of-the-art procedure

Our final experiments are devoted to compare the performance of our best proposed algorithm (GRASP+VND) with the performance of the LNS algorithm proposed in Monçores et al. (2018). Let us summarize that the final configuration of our GRASP+VND has been executed for 20 iterations per instance. In each iteration, the α value of the GRASP procedure is set at random. The VND is configured with neighborhoods N_1, N_3, N_4, and the method includes both advanced strategies: efficient evaluation of the objective function after a move and reduction of the neighborhood size. On the other hand, the LNS procedure has been executed using the original code and configurations provided by Monçores et al. (2018), which are publicly available at https://bitbucket.org/marlonmoncores/unirio_lns_cms.

Table 5
Comparison of the results of the algorithm presented in this work (GRASP+VND) and the state of the art (LNS Monçores et al., 2018).

Category	Method	Avg. MQ	Dev. (%)	σ	# Best	CPUt (s)	Avg. $\|M\|$
Small	GRASP+VND	**3.4757**	**0.00%**	0.00%	**64**	0.09	9.95
	LNS (Monçores et al., 2018)	3.4603	0.68%	2.14%	35	**0.02**	9.83
Medium	GRASP+VND	**12.9489**	<0.01%	<0.01%	**27**	0.40	37.34
	LNS (Monçores et al., 2018)	12.9270	0.26%	0.33%	10	**0.33**	37.10
Large	GRASP+VND	**24.9043**	**0.02%**	0.05%	**14**	28.20	83.22
	LNS (Monçores et al., 2018)	24.8868	0.12%	0.17%	5	**4.95**	82.78
Very large	GRASP+VND	**63.1797**	<0.01%	0.01%	**11**	**235.77**	237.85
	LNS (Monçores et al., 2018)	63.1029	0.15%	0.17%	2	338.87	231.62
All	GRASP+VND	**15.0611**	<0.01%	0.02%	**116**	**28.95**	50.89
	LNS (Monçores et al., 2018)	15.0374	0.44%	1.57%	52	36.33	50.05

Both algorithms were run over the whole dataset of 124 instances on the same computer (AMD EPYC 7282 @ 2795 MHz with 8 cores and 8 GB RAM). The operating system was Microsoft Windows 10 Pro 10.0.19042 x64. All methods were implemented with Java OpenJDK 15.0.2.

In Table 5, we report the results obtained by both algorithms for all the instances of the dataset, organized in groups (small, medium, large, and very large). In particular, we report the average quality of the best solutions found per instance (Avg. MQ), the average deviation from the best solution found in the experiment (Dev. (%)), the standard deviation (σ), the number of best solutions found (# Best), and the average CPU time per instance (CPUt (s)). Notice that the standard deviation is computed with respect to the Dev. (%) value per instance. We do not provide the standard deviation of the Avg. MQ value since it varies significantly depending on the size of the instance. As it can be observed, the quality of the solutions reached by GRASP+VND is higher than the quality of the solutions found by LNS. On average, GRASP+VND is able to obtain solutions closer to the best one found in the experiment (with a deviation <0.01%) than LNS (with a deviation of 0.44%). Additionally, GRASP+VND was able to find better solutions for 116 out of 124 instances, while LNS reached the best solution only for 52 instances. Moreover, if we do not consider the 44 instances in which there is a tie (i.e., both algorithms achieve a solution of the same quality), GRASP+VND reaches the best solution in 72 out of 80 instances, while LNS reaches the best solution for 8 of them. Finally, GRASP+VND was on average almost 8 s faster than LNS. This improvement is particularly notable for the very large instances, where GRASP+VND is almost 103 s faster than LNS on average. We include the individual results for each instance in the Appendix to facilitate future comparisons.

To identify significant performance differences between both algorithms, we performed the Wilcoxon's signed rank test over the obtained results. Particularly, we applied the two-tailed version with a significance level of 0.01. Our null hypothesis is that there is no difference in the quality of the solutions obtained by the compared algorithms. In contrast, the alternative hypothesis is that the solutions obtained by one algorithm are better than the solutions obtained by the other one. The test resulted in a p-value smaller than 0.01. Thus, we can confidently reject the null hypothesis. Therefore, we can conclude that the difference in quality between the solutions reached by the GRASP+VND algorithm and the solutions reached by the LNS (Monçores et al., 2018) algorithm is statistically significant with $p < 0.01$.

6.4. Discussion of the results

Analyzing the results presented in previous sections by group of instances (i.e., organized by size depending on the number of vertices) we cannot identify a clear pattern in the behavior of the compared methods. However, although the proposed approach (GRASP+VND) is faster, on average, than the previous state-of-the-art algorithm, we noticed that it takes a significant amount of time to process some particular instances such as *graph10up193* or *graph10up49*, while it is very fast in other instances such as *ylayout*, *apache_ant*, or *eclipse_jgit*, despite of being very large. This behavior might be partially explained by some properties of the instances. Particularly, we identified that our method performs better in sparse graphs, which are frequent in the SMCP context.

Performing a closer analysis, in Section 5.2, we introduced an advanced strategy to reduce the size of neighborhoods by considering only "adjacent moves" (i.e., those where every affected vertex is placed in a module where there exists at least one of its adjacent vertices). Then, the number of neighbor solutions that exist in a given neighborhood is directly related to the degree of vertices in the graph. The lesser the number of edges, the fewer the number of solutions to be explored in the neighborhood. On the contrary, when observing the previous best proposal in the state of the art, we noticed that LNS does not reduce the size of neighborhoods based on the number of edges, but it partially explores the neighborhoods depending on the number of vertices/modules. Therefore, the introduced strategy presents an advantage over the previous state-of-the-art for sparse graphs, where there exists a low number of edges compared to the number of vertices. In contrast, this advantage disappears for dense graphs, where the number of edges per vertex is high. Observing the density of the graphs in the considered dataset, it varies between 0.43% and 21.67%, with the only two exceptions of *graph10up49*, with a density of 70.15%, and *graph10up193*, with a density of 24.80%.

As far as the time is concerned, if we observe in the detailed results per instance the time needed to process large and very large instances (where the differences in time consumption start to be noticeable), GRASP+VND is faster in 28 out of the 31 instances. Furthermore, GRASP+VND is able to find better solutions for some of the largest instances, such as *ylayout* and *apache_ant*, in only 0.03% of the time needed by LNS.

From the practical perspective of software developers, the time needed to receive insights about the quality of their code is essential to let them refactor the code as soon as possible. Therefore, we do believe that our approach might be beneficial for general software since, according to Mitchell and Mancoridis (2008), in the case of the SMCP, dependencies between components often result in sparse graphs, with densities that vary between 1.77% and 21.52%.

7. Threats to validity

As it is customary in the literature related to SBSE (de Oliveira Barros and Dias-Neto, 2011; Feldt and Magazinius, 2010), in this section we discuss some limitations identified related to our research that threaten the validity of the results presented in this paper.

First, empirical experiments are highly dependent on the data used for testing. Although we used a set of 124 real-world instances previously collected in related works (Monçores et al., 2018), the promising empirical results obtained might be limited by either the size of the dataset or the particular instances conforming it. Nevertheless, the instances used are publicly available online, which allows better reproducibility and future extensions.

Additionally, the validity of our results should be taken into account considering some known shortcomings in the interpretation of the MQ metric (Mancoridis et al., 1998). Here, we assume that an improvement in the MQ value of a software project implies an improvement in the maintainability of the project. The discussion of the validity of the underlying objective function subjected to optimization in the SMCP is beyond the scope of this work. Instead, we focus our attention on the design of good algorithmic strategies for the SMCP (i.e., categories of neighborhoods and promising areas in the search space) without regard to the particular objective function used.

However, previous assumptions about the quality of a solution based on the MQ value are not necessarily true for a developer, and further validation by human experts could be desirable. In this context, results of the same quality might produce different software organizations, and even solutions with a lower MQ score might be more attractive from the practical point of view of a software developer. Our algorithmic proposal includes a stochastic key component (α) which determines the randomness/greediness of the constructed solution, providing different software organizations in different executions of the algorithm. This fact can be seen as an opportunity rather than as a weakness, since the algorithm could be configured to suit the needs of each developer. Furthermore, although the method would try to maximize the MQ value, developers could examine the solution provided in terms of understandability and run the algorithm again to obtain an alternative proposal, in case they are not satisfied with the solution provided.

Finally, we would like to analyze to what extent our findings are valid for other related contexts. First, we believe that the categorization of neighborhoods presented in this work is independent of either the objective function studied or the underlying optimization problem. Therefore, it could be extended to other clustering problems. Second, we present a strategy to reduce the size of neighborhoods and explore promising regions in the search space, which is valid for the SMCP independently of the objective function studied. However, as discussed previously, this strategy is built on the assumption that instances are sparse, which seems valid for the SMCP, where the density of software projects often varies in the range [0.43%–24.80%].

8. Conclusions

In this paper, we have proposed an optimization algorithm for the Software Module Clustering Problem. The proposed method is based on a GRASP schema, where the intensification phase is replaced by a VND strategy. We favorably compare the proposed algorithm (GRASP+VND) with the best previous proposal on the state of the art, which is based on a Large Neighborhood Search strategy. The latter is, to the best of our knowledge, the best mono-objective algorithm for the SMCP that studies the MQ objective function. We compared the two approaches on a dataset of 124 real instances made available by the community, and our approach reached the best solution in 116 instances, outperforming LNS (Monçores et al., 2018) in 72 instances. These results turned out to be statistically significant according to the Wilcoxon signed rank test.

In this research, we have identified that, in the context of the SMCP, we can classify any neighborhood into three different categories: neighborhoods defined by movements that do not alter the number of modules, neighborhoods defined by movements that increase the number of modules, and neighborhoods defined by movements that reduce the number of modules. Therefore, algorithms based on local search could benefit from exploring neighborhoods in different categories, deriving different possible strategies with the ability to maintain, increase, or reduce the number of modules in a given solution. Following this idea, in this work, we have configured a VND method with three neighborhoods, one per each identified category: N_1, defined by an insertion move (maintain); N_3, defined by an extraction move (increase); and N_4, defined by a destruction move (reduce). We believe that this categorization could also be interesting for other graph partitioning problems, in which vertices are organized in clusters in a similar fashion.

As we illustrate in this work, exploration strategies can be further benefited by an efficient implementation of highly repeated operations, such as a fast evaluation of the objective function after a move. Additionally, we would like to highlight the importance of reducing, as much as possible, the size of the explored neighborhoods by identifying the promising areas of the search space, and thus reducing the computing time needed for its exploration. Particularly, in the context of the SMCP, we were able to combine these two strategies and reduce the execution time by 5 orders of magnitude with respect to the original implementation, showing the importance of leveraging domain-specific knowledge about the problem to configure optimization algorithms.

In the context of software development, it is important not only to provide high-quality solutions but also to find them in a short computing time. The small consumption time shown by the proposed method allows it to be easily incorporated in Integrated Development Environments, thus providing developers with suggestions to improve the organization of their code in real time. This will help developers to improve the software quality of their projects, either by implementing the suggested remodularizations or by learning from them. Moreover, this solution can be implemented in software repositories as a part of a quality gate to ensure the quality of the contributed software.

Finally, search-based software engineering can be seen as a learning tool to evaluate the behavior of Software Engineering concepts. In this sense, we believe that the proposed method allows for a closer examination of the modularity concept in general and of the MQ metric in particular when taken to the extreme. This closer evaluation might allow developers to observe current limitations in the coupling–cohesion paradigm and to improve the quality of their projects.

CRediT authorship contribution statement

Javier Yuste: Conceptualization, Investigation, Data curation, Writing – original draft, Software. **Abraham Duarte:** Supervision, Formal analysis, Writing – review & editing, Funding acquisition. **Eduardo G. Pardo:** Conceptualization, Methodology, Investigation, Validation, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has been partially funded by the Spanish Government (Refs. PGC2018-095322-B-C22 and PID2021-125709OA-C22) and by the Comunidad de Madrid (Ref. P2018/TCS-4566), cofinanced by European Structural and Investment Funds (ESF and ERDF). The opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect those of any of the funders.

Appendix. Detailed results

In Tables Tables A.6–A.9, we include the individual results obtained in the final experimentation presented in Section 6.3. For each instance, we include the number of vertices ($|V|$), the number of edges ($|E|$), and the density (D). Additionally, for each algorithm we include the score (MQ) and CPU time (CPUt (s)). To ease future comparisons, the instances have been sorted in ascending order, according to their number of vertices, and separated in four groups (small, medium, large, and very large).

Table A.6
Detailed results for small instances.

Instance	Original			Reduced		GRASP+VND		LNS (Mançores et al., 2018)	
	$\|V\|$	$\|E\|$	D	$\|V\|$	$\|E\|$	MQ	CPUt (s)	MQ	CPUt (s)
squid	2	2	100.00%	1	1	**1.0000**	<0.01	**1.0000**	<0.01
small	6	5	16.67%	3	5	**1.8333**	<0.01	1.5238	<0.01
compiler	13	32	20.51%	13	32	**1.5065**	0.02	1.4968	<0.01
random	13	30	19.23%	12	23	**2.4409**	<0.01	**2.4409**	<0.01
regexp	14	20	10.99%	9	18	**2.4470**	<0.01	**2.4470**	0.02
jstl	15	20	9.52%	14	15	**5.0000**	<0.01	**5.0000**	<0.01
lab4	15	18	8.57%	10	14	**3.4000**	<0.01	**3.4000**	<0.01
netkit-ping	15	15	7.14%	1	1	**1.0000**	<0.01	**1.0000**	<0.01
nss_ldap	15	16	7.62%	3	4	**1.0000**	<0.01	0.9763	<0.01
nos	16	52	21.67%	15	50	**1.6775**	<0.01	1.6565	<0.01
lslayout	17	43	15.81%	17	43	**1.8613**	<0.01	1.8171	0.02
boxer	18	29	9.48%	12	29	**3.1011**	<0.01	**3.1011**	0.02
netkit-tftpd	18	23	7.52%	6	11	**1.1442**	0.02	**1.1442**	0.02
sharutils	19	36	10.53%	14	30	**2.5492**	0.02	2.5338	<0.01
mtunis	20	57	15.00%	20	57	**2.3145**	<0.01	**2.3145**	0.02
spdb	21	17	4.05%	7	8	**5.5897**	<0.01	**5.5897**	0.02
xtell	22	57	12.34%	14	44	**2.0052**	<0.01	**2.0052**	0.02
bunch	23	62	12.25%	15	45	**2.4060**	<0.01	2.4026	<0.01
ispell	24	103	18.66%	23	97	**2.3639**	0.02	2.3323	<0.01
netkit-inetd	24	25	4.53%	4	6	**1.3121**	<0.01	**1.3121**	<0.01
nanoxml	25	64	10.67%	23	62	**3.8173**	<0.01	**3.8173**	<0.01
ciald	26	64	9.85%	22	62	**2.8513**	<0.01	2.8459	<0.01
jodamoney	26	102	15.69%	26	85	**2.7489**	0.02	**2.7489**	<0.01
Modulizer	26	66	10.15%	18	57	**2.7579**	<0.01	**2.7579**	0.02
bootp	27	75	10.68%	19	55	**2.1985**	0.01	**2.1985**	<0.01
jxlsreader	27	73	10.40%	25	73	**3.6025**	0.02	3.5921	0.02
sysklogd-1	28	74	9.79%	22	65	**1.7105**	<0.01	1.6870	0.02
telnetd	28	81	10.71%	19	62	**1.8475**	<0.01	**1.8475**	<0.01
crond	29	112	13.79%	25	90	**2.3030**	0.03	**2.3030**	0.02
netkit-ftp	29	95	11.70%	24	76	**1.7680**	<0.01	1.7562	<0.02
rcs	29	163	20.07%	28	155	**2.2775**	0.06	2.2262	0.02
seemp	30	61	7.01%	21	43	**4.6536**	<0.01	**4.6536**	<0.01
dhcpd-2	31	122	13.12%	26	104	**3.4944**	0.02	3.4889	0.02
cyrus-sasl	32	100	10.08%	21	77	**3.2518**	0.02	**3.2518**	<0.01
tcsh	32	105	10.58%	23	86	**1.2141**	0.02	1.1945	0.02
micq	33	156	14.77%	26	116	**2.1680**	0.03	2.1329	<0.01
apache_zip	36	86	6.83%	30	74	**5.7663**	0.02	**5.7663**	0.02
star	36	89	7.06%	36	89	**3.8321**	0.02	3.8264	0.03
bison	37	179	13.44%	36	167	**2.7039**	0.08	2.6797	0.03
cia	38	185	13.16%	34	167	**3.7507**	0.06	3.7289	0.03
stunnel	38	97	6.90%	25	78	**2.5261**	<0.01	**2.5261**	0.02
minicom	40	257	16.47%	35	198	**2.5758**	0.05	**2.5758**	0.03
mailx	41	331	20.18%	38	244	**3.2402**	0.13	**3.2402**	0.05
dot	42	255	14.81%	40	248	**2.8460**	0.07	2.8373	0.05
screen	42	292	16.96%	35	208	**2.2601**	0.07	2.2455	0.02
slang	45	242	12.22%	42	184	**4.6794**	0.07	**4.6794**	0.05
slrn	45	323	16.31%	37	231	**2.3928**	0.10	2.3843	0.03
net-tools	48	183	8.11%	44	157	**4.3159**	0.03	4.2869	0.02
graph10up49	49	1650	70.15%	49	1054	**1.2588**	3.83	1.2503	0.19
wu-ftpd-1	50	230	9.39%	44	187	**2.4437**	0.04	2.4068	0.02
joe	51	540	21.18%	44	318	**3.3411**	0.17	**3.3411**	0.05
hw	53	51	1.85%	15	28	**8.4967**	<0.01	**8.4967**	<0.01
imapd-1	53	298	10.81%	40	211	**3.6250**	0.08	**3.6250**	0.03
wu-ftpd-3	54	278	9.71%	48	222	**3.3953**	0.07	3.3405	0.03
udt-java	56	227	7.37%	54	210	**5.2829**	0.10	**5.2829**	0.05
javaocr	58	155	4.69%	43	126	**9.0242**	0.01	**9.0242**	0.06
dhcpd-1	59	571	16.69%	55	398	**3.9871**	0.16	3.9609	0.05
icecast	60	650	18.36%	51	419	**2.7544**	0.27	2.7474	0.06
pfcda_base	60	197	5.56%	54	168	**7.3328**	0.04	**7.3328**	0.05
servletapi	61	131	3.58%	47	122	**9.8823**	0.03	9.7578	0.05
php	62	191	5.05%	39	148	**5.3242**	0.02	**5.3242**	0.02
bunch2	65	151	3.63%	42	111	**7.7251**	0.03	**7.7251**	0.06
forms	68	270	5.93%	64	224	**8.3258**	0.06	**8.3258**	0.06
jscatterplot	74	232	4.29%	68	171	**10.7419**	0.03	**10.7419**	0.06

Table A.7
Detailed results for medium instances.

Instance	Original			Reduced		GRASP+VND		LNS (Monçores et al., 2018)	
	$\|V\|$	$\|E\|$	D	$\|V\|$	$\|E\|$	MQ	CPUt (s)	MQ	CPUt (s)
jxlscore	79	330	5.36%	67	305	**9.7968**	0.10	9.7864	0.17
elm-2	81	683	10.54%	75	541	**3.8296**	0.32	3.8000	0.22
jfluid	81	315	4.86%	75	276	**6.5796**	0.06	**6.5796**	0.08
grappa	86	295	4.04%	76	249	**12.7054**	0.09	**12.7054**	0.11
elm-1	88	941	12.29%	81	736	**4.3171**	0.61	4.2893	0.17
gnupg	88	601	7.85%	74	420	**7.1585**	0.21	7.1300	0.20
inn	90	624	7.79%	80	485	**8.0240**	0.34	8.0180	0.20
bash	92	901	10.76%	86	662	**5.9359**	0.40	5.8776	0.33
jpassword	96	361	3.96%	87	332	**10.7068**	0.14	10.5942	0.13
bitchx	97	1653	17.75%	92	1075	**4.3860**	1.23	4.3517	0.45
junit	99	276	2.84%	61	193	**11.0901**	0.06	**11.0901**	0.06
xntp	111	729	5.97%	95	534	**8.3614**	0.33	8.3455	0.34
acqCIGNA	114	179	1.39%	75	157	**16.5971**	0.33	16.5166	0.28
bunch_2	116	364	2.73%	98	355	**13.6204**	0.11	13.6075	0.61
exim	118	1255	9.09%	105	891	**6.6260**	0.78	**6.6260**	0.75
xmldom	118	209	1.51%	67	159	**10.9236**	0.03	**10.9236**	0.05
cia++	124	369	2.42%	63	309	**15.4728**	0.11	15.4724	0.16
tinytim	129	564	3.42%	122	499	**12.5400**	0.20	12.4937	0.44
mod_ssl	135	1095	6.05%	123	752	**10.1174**	0.54	10.1086	0.66
jkaryoscope	136	460	2.51%	127	345	**18.9871**	0.11	**18.9871**	0.23
ncurses	138	682	3.61%	120	487	**11.8318**	0.19	11.8245	0.27
gae_plugin_core	139	375	1.95%	87	259	**17.3370**	0.08	**17.3370**	0.22
lynx	148	1745	8.02%	100	1211	**4.9912**	2.38	4.9717	0.53
javacc	153	722	3.10%	145	663	**10.7138**	0.39	10.6255	0.66
lucent	153	103	0.44%	66	74	**59.9488**	<0.01	**59.9488**	0.05
JavaGeom	171	1445	4.97%	160	1339	14.1008	2.22	**14.1034**	1.44
incl	174	360	1.20%	122	329	**13.6396**	0.08	13.6146	0.20
jdendogram	177	583	1.87%	148	447	**26.0843**	0.15	26.0569	0.42
xmlapi	182	413	1.25%	125	358	19.0954	0.13	**19.0980**	0.27

Table A.8
Detailed results for large instances.

Instance	Original			Reduced		GRASP+VND		LNS (Monçores et al., 2018)	
	$\|V\|$	$\|E\|$	D	$\|V\|$	$\|E\|$	MQ	CPUt (s)	MQ	CPUt (s)
jmetal	190	1137	3.17%	178	1123	**12.6398**	1.62	12.5520	3.69
graph10up193	193	9190	24.80%	193	7552	**2.2492**	484.29	2.2471	11.72
dom4j	195	930	2.46%	181	876	**19.2268**	2.75	19.2060	2.94
nmh	198	3262	8.36%	190	2473	**9.4166**	7.85	9.3990	7.16
pdf_renderer	199	629	1.60%	172	497	**22.3628**	0.18	22.3573	0.55
Jung_graph_model	207	603	1.41%	187	532	**32.0320**	0.32	32.0292	1.09
jung_visualization	208	919	2.13%	192	837	**21.8608**	0.55	21.8183	3.41
jconsole	220	859	1.78%	187	663	**26.5195**	0.33	**26.5195**	1.14
pfcda_swing	248	885	1.44%	237	801	**29.0991**	0.45	29.0950	1.75
jml-1.0b4	267	1745	2.46%	262	1671	**17.5971**	2.20	17.5601	11.80
jpassword2	269	1348	1.87%	248	1171	**28.6024**	1.09	28.5946	6.00
notelab-full	293	1349	1.58%	273	1233	**29.7366**	1.37	29.6454	7.20
Poormans_CMS	301	1118	1.24%	253	1009	**34.2478**	0.79	34.2338	5.56
log4j	305	1078	1.16%	258	944	32.5023	0.68	**32.5127**	4.25
jtreeview	320	1057	1.04%	268	830	48.1222	0.48	**48.1280**	4.38
bunchall	324	1339	1.28%	243	1250	16.9696	1.33	**16.9850**	6.47
JACE	338	1524	1.34%	271	1209	26.7554	0.74	**26.8140**	3.56
javaws	377	1403	0.99%	293	1107	**38.3373**	0.62	38.2648	6.47

Table A.9
Detailed results for very large instances.

Instance	Original			Reduced		GRASP+VND		LNS (Monçores et al., 2018)	
	$\|V\|$	$\|E\|$	D	$\|V\|$	$\|E\|$	*MQ*	CPUt (s)	*MQ*	CPUt (s)
swing	413	1513	0.89%	377	1478	45.3688	5.87	**45.3779**	19.72
lwjgl-2.8.4	453	1976	0.97%	392	1790	37.1433	2.17	**37.1500**	19.91
res_cobol	470	7163	3.25%	461	5690	**16.0509**	20.99	15.9510	115.85
ping_libc	481	2854	1.24%	395	1802	**51.9359**	1.78	51.8601	11.50
y_base	556	2510	0.81%	479	2166	**58.3401**	3.92	58.3370	56.11
krb5	558	3793	1.22%	508	2494	**54.6419**	4.13	54.6067	55.58
apache_ant_taskdef	626	2421	0.62%	538	2108	**66.7152**	4.28	66.6521	18.17
itextpdf	650	3898	0.92%	603	3555	**58.7029**	62.92	58.6110	115.63
apache_lucene_core	738	3726	0.69%	712	3330	**77.0329**	25.70	76.8654	110.71
eclipse_jgit	909	5452	0.66%	872	4904	**86.5955**	77.95	86.5954	382.80
linux	916	11722	1.40%	876	11059	**54.7541**	2822.70	54.5814	2328.94
apache_ant	1085	5329	0.45%	999	4881	**102.8853**	18.11	102.7229	634.15
ylayout	1161	5770	0.43%	1004	4990	**111.1698**	14.51	111.0273	536.21

References

Arcuri, A., Briand, L., 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250.

Bakota, T., Hegedus, P., Ladanyi, G., Kortvelyesi, P., Ferenc, R., Gyimothy, T., 2012. A cost model based on software maintainability. In: IEEE International Conference on Software Maintenance, ICSM. pp. 316–325.

Barros, M.d.O., 2012. An analysis of the effects of composite objectives in multiobjective software module clustering. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, pp. 1205–1212.

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. In: International Symposium on Search Based Software Engineering. Springer, pp. 75–89.

Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., Wagner, D., 2007. On modularity clustering. IEEE Trans. Knowl. Data Eng. 20 (2), 172–188.

Briand, L.C., Morasca, S., Basili, V.R., 1999. Defining and validating measures for object-based high-level design. IEEE Trans. Softw. Eng. 25 (5), 722–743.

Cavero, S., Pardo, E.G., Duarte, A., 2022. A general variable neighborhood search for the cyclic antibandwidth problem. Comput. Optim. Appl. 1–31.

Chen, C., Alfayez, R., Srisopha, K., Boehm, B., Shi, L., 2017. Why is it important to measure maintainability and what are the best ways to do it? In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017. Institute of Electrical and Electronics Engineers Inc., pp. 377–378.

Chhabra, J.K., 2017. Harmony search based remodularization for object-oriented software systems. Comput. Lang. Syst. Struct. 47, 153–169.

Chhabra, J.K., 2018. Many-objective artificial bee colony algorithm for large-scale software module clustering problem. Soft Comput. 22 (19), 6341–6361.

Chong, C.Y., Lee, S.P., 2017. Automatic clustering constraints derivation from object-oriented software using weighted complex network with graph theory analysis. J. Syst. Softw. 133, 28–53.

Colanzi, T.E., Assunção, W.K., Vergilio, S.R., Farah, P.R., Guizzo, G., 2020. The symposium on search-based software eengineering: Past, present and future. Inf. Softw. Technol. 127, 106372.

Duarte, A., Pantrigo, J.J., Gallego, M., 2007. Metaheurísticas. Dykinson, Madrid.

Feldt, R., Magazinius, A., 2010. Validity threats in empirical software engineering research-an initial survey.. In: Seke. pp. 374–379.

Feo, T.A., Resende, M.G., 1995. Greedy randomized adaptive search procedures. J. Global Optim. 6 (2), 109–133.

Gibbs, S., Casais, E., Nierstrasz, O., Pintado, X., Tsichritzis, D., 1990. Class management for software communities. Commun. ACM 33 (9), 90–103.

Gil-Borrás, S., Pardo, E.G., Alonso-Ayuso, A., Duarte, A., 2020. GRASP With variable neighborhood descent for the online order batching problem. J. Global Optim. 78 (2), 295–325.

Harman, M., Mansouri, S.A., Zhang, Y., 2012. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45 (1), 1–61.

Huang, J., Liu, J., Yao, X., 2017. A multi-agent evolutionary algorithm for software module clustering problems. Soft Comput. 21 (12), 3415–3428.

International Organization for Standardization, 2017a. ISO/IEC/IEEE 12207:2017 Systems and software engineering — software life cycle processes.

International Organization for Standardization, 2017b. ISO/IEC/IEEE 24765:2017 Systems and software engineering — vocabulary.

Köhler, V., Fampa, M., Araújo, O., 2013. Mixed-integer linear programming formulations for the software clustering problem. Comput. Optim. Appl. 55 (1), 113–135.

Kumari, A.C., Srinivas, K., 2016. Hyper-heuristic approach for multi-objective software module clustering. J. Syst. Softw. 117, 384–401.

Larman, C., 2012. Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development. Pearson Education India.

Lozano-Osorio, I., Martínez-Gavara, A., Martí, R., Duarte, A., 2022. Max-min dispersion with capacity and cost for a practical location problem. Expert Syst. Appl. 116899.

Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y.-F., Gansner, E.R., 1998. Using automatic clustering to produce high-level system organizations of source code. In: Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242). IEEE, pp. 45–52.

Martín-Santamaría, R., Sánchez-Oro, J., Pérez-Peló, S., Duarte, A., 2022. Strategic oscillation for the balanced minimum sum-of-squares clustering problem. Inform. Sci. 585, 529–542.

McConnell, S., 2004. Code Complete. Pearson Education.

Mitchell, B.S., Mancoridis, S., 2002a. Using heuristic search techniques to extract design abstractions from source code. In: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, pp. 1375–1382.

Mitchell, B.S., Mancoridis, S., 2002b. A Heuristic Search Approach to Solving the Software Clustering Problem. Drexel University Philadelphia, PA, USA.

Mitchell, B.S., Mancoridis, S., 2008. On the evaluation of the bunch search-based software modularization algorithm. Soft Comput. 12 (1), 77–93.

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A., 2015. Many-objective software remodularization using NSGA-III. ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (3), 1–45.

Mladenović, N., Hansen, P., 1997. Variable neighborhood search. Comput. Oper. Res. 24 (11), 1097–1100.

Monçores, M.C., Alvim, A.C.F., Barros, M.O., 2018. Large neighborhood search applied to the software module clustering problem. Comput. Oper. Res. 91, 92–111.

Naseem, R., Maqbool, O., Muhammad, S., 2013. Cooperative clustering for software modularization. J. Syst. Softw. 86 (8), 2045–2062.

de Oliveira Barros, M., Dias-Neto, A.C., 2011. 0006/2011-Threats to validity in search-based software engineering empirical studies. RelaTe-DIA.

Pinto, A.F., de Faria Alvim, A.C., de Oliveira Barros, M., 2014. ILS For the software module clustering problem. In: XLVI Simpósio Brasileiro de Pesquisa Operacional. Salvador:[Sn]. pp. 1972–1983.

Praditwong, K., 2011. Solving software module clustering problem by evolutionary algorithms. In: 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE). IEEE, pp. 154–159.

Praditwong, K., Harman, M., Yao, X., 2010. Software module clustering as a multi-objective search problem. IEEE Trans. Softw. Eng. 37 (2), 264–282.

Prajapati, A., Chhabra, J.K., 2018. A particle swarm optimization-based heuristic for software module clustering problem. Arab. J. Sci. Eng. 43 (12), 7083–7094.

Prajapati, A., Chhabra, J.K., 2019. MaDHS: MAny-objective discrete harmony search to improve existing package design. Comput. Intell. 35 (1), 98–123.

Ramirez, A., Romero, J.R., Ventura, S., 2019. A survey of many-objective optimisation in search-based software engineering. J. Syst. Softw. 149, 382–395.

Talbi, E.-G., 2009. Metaheuristics: From Design to Implementation, Vol. 74. John Wiley & Sons.

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.

Varghese R, B.G., Raimond, K., Lovesum, J., 2019. A novel approach for automatic remodularization of software systems using extended ant colony optimization algorithm. Inf. Softw. Technol. 114, 107–120.

Javier Yuste is a researcher at Universidad Rey Juan Carlos (Madrid, Spain) where he is part of the Group for Research in Algorithms For Optimization (GRAFO). His research interests focus on the applicability of Artificial Intelligence (AI) techniques to solve software quality problems. In particular, his research focuses on the automatic optimization of the quality of software systems through the use of metaheuristic algorithms.

Abraham Duarte is a Full Professor of Computer Science at Universidad Rey Juan Carlos (Madrid, Spain) where he leads the Group for Research in Algorithms For Optimization (GRAFO). His research interests include the proposal of heuristic and metaheuristic algorithms for solving hard optimization problems in many operation-management areas such as logistics, supply chains, telecommunications, decision-making under uncertainty and simulated systems.

Eduardo G. Pardo is an Associate Professor at Universidad Rey Juan Carlos (Madrid, Spain). He got his Ph.D. in heuristic optimization and he is co-author of many journal papers related to this topic. Additionally, Eduardo is founding member of the Group for Research in Algorithms For Optimization (GRAFO) at Universidad Rey Juan Carlos. Among his research interests he includes the development and application of efficient Artificial Intelligence techniques to real-world problems, where we can find problems related to software.