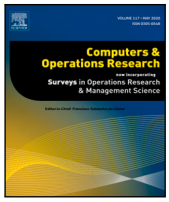
Computers & Operations Research 165 (2024) 106584

Contents lists available at ScienceDirect

Computers and Operations Research

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



General Variable Neighborhood Search for the optimization of software quality

Javier Yuste, Eduardo G. Pardo *, Abraham Duarte

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

ARTICLE INFO

Keywords:
Variable Neighborhood Search
Software maintainability
Search-Based Software Engineering
Software Module Clustering
Heuristic

ABSTRACT

In the area of Search-Based Software Engineering, software engineering issues are formulated and tackled as optimization problems. Among the problems within this area, the Software Module Clustering Problem (SMCP) consists of finding an organization of a software project that minimizes coupling and maximizes cohesion. Since modular code is easier to understand, the objective of this problem is to increase the quality of software projects, thus increasing their maintainability and reducing the associated costs. In this work we study a recently proposed objective function named Function of Complexity Balance (FCB). Since this problem has been demonstrated to be $\mathcal{NP}$-hard, we propose a new heuristic algorithm based on the General Variable Neighborhood Search (GVNS) schema to tackle the problem. For the GVNS, we propose six different neighborhood structures and categorize them into three different groups. Then, we analyze their contribution to the results obtained by the algorithm. In order to improve the efficiency of the proposed approach, we leverage domain-specific information to perform incremental evaluations of the objective function and to explore only areas of interest in the search space. The proposed algorithm has been tested over a set of real world software repositories, achieving better results than the previous state-of-the-art method, a Hybrid Genetic Algorithm, in terms of both quality and computing times. Furthermore, the relevance of the improvement produced by our proposal has been corroborated by non-parametric statistical tests.

1. Introduction

Search-Based Software Engineering (SBSE) is a research area where software engineering tasks are formulated and tackled as optimization problems. In this area, there exists a wide array of problems of different nature, including both technical and non-technical tasks (Catal et al., 2016), such as reducing the cognitive complexity of software systems (Saborido et al., 2022), identifying modules that are faulty (Thirumoorthy et al., 2022; Manchala and Bisi, 2022), building automated tests (Sahin and Akay, 2016), or scheduling software updates for connected cars (Andrade et al., 2019). Nevertheless, in spite of the aforementioned variety of tackled issues, the intention of the research in SBSE is to raise the quality of software projects and reduce their associated costs (Colanzi et al., 2020).

In the context of SBSE research, the optimization problem known as the Software Module Clustering Problem (SMCP) consists of maximizing the modularity of software systems. As specified by the International Organization for Standardization (ISO, 2017), the modularity of a system is the set of "*software attributes that provide a structure of highly independent components*". In software engineering, modularity has traditionally been measured as a balance between cohesion (dependencies between artifacts that belong to the same module) and coupling (dependencies between artifacts in different modules). In particular, a system is often considered well organized/modularized if it presents high cohesion and low coupling. In this work, we use the term "artifact" referring to atomic units and the term "module" referring to sets of artifacts.

In software systems, a good modular organization facilitates the comprehension of the code. During the lifecycle of software projects, most of the efforts made by developers are dedicated to understanding the existent software. Moreover, modularization has truly interest in large projects, which is usually the case of real software, where the size of the project makes the process of understanding the code much harder. By maximizing the modularity of software systems, the SMCP aims to enhance the overall quality of the code, facilitate the understanding of the system, and reduce the costs associated with its development and maintenance (Briand et al., 1999; Gibbs et al., 1990; Larman, 2012). However, decomposing a system into modules is not a trivial task (Fakhoury et al., 2019; Mkaouer et al., 2014), and there exist previous works that have tackled this problem (Mancoridis

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

https://doi.org/10.1016/j.cor.2024.106584
Received 13 July 2023; Received in revised form 1 December 2023; Accepted 7 February 2024
Available online 8 February 2024

et al., 1998; Mitchell and Mancoridis, 2006). Indeed, the optimization of software modularity is being used as a tool to refine and improve the concept of modularity (de Oliveira Barros and de Almeida Farzat, 2013; de Oliveira Barros et al., 2015). Furthermore, given the $\mathcal{NP}$-hard nature of the problem and the size of large projects, it is almost humanly impossible to easily enhance a given organization. Therefore, the goal of the approach proposed in this paper is to help software developers to enhance the quality of their software, by automatizing the identification of more suitable organizations of their code.

In this work, we propose a General Variable Neighborhood Search (GVNS) (Mladenović and Hansen, 1997) to tackle the SMCP studying the Function of Complexity Balance (FCB), a novel objective function introduced by Mu et al. (2020). Our approach is compared with the results obtained by the Hybrid Genetic Algorithm (HGA) proposed by the same authors (Mu et al., 2020), which is, as far as we know, the current state-of-the-art algorithm for the problem. To compare the algorithms, we use a set of 124 real-world software systems, which were published by Monçores et al. (2018). Finally, we showcase that the proposed approach finds the best known solutions for 124 out of the 124 instances in the dataset. Moreover, the results are shown to be statistically significant, with a p-value lower than 0.00001, according to the Wilcoxon Signed-Rank test.

The remainder of this work is structured as follows. In Section 2, we present a literature review of the SMCP, followed by its definition in Section 3. Then, in Section 4, we detail the proposed algorithm. In Section 4.1, we describe the neighborhoods proposed. Next, in Section 5, we describe two advanced strategies to enhance the efficiency of our algorithm. In Section 6, we detail the performed experiments and discuss the obtained results. Finally, in Section 7, we present our conclusions.

2. Literature review

As far as we know, the first to study the SMCP were Mancoridis et al. (1998). They introduced an objective function known as Modularization Quality (MQ), which computes the quality of a solution taking into account the cohesion and coupling of the architecture. In addition, only considering cohesion and coupling would result in a trivial, useless solution where every artifact is located in the same module. For this reason, MQ implicitly takes into account the number of modules. Two variants of MQ, named BasicMQ and TurboMQ, were later formulated by the same authors to accelerate the evaluation of the objective function (Mitchell and Mancoridis, 2002a,b). These metrics have been widely used in the literature to address the SMCP from its original appearance (Huang et al., 2017; Praditwong, 2011; Prajapati and Chhabra, 2018).

Although MQ and its associated variants have been widely used in the area, some authors highlight the existence of several concerns in the cohesion-coupling paradigm with respect to the MQ metric (Mancoridis et al., 1998; de Oliveira Barros et al., 2015). Due to these concerns, some alternatives have been proposed in the literature. In this sense, the Function of Complexity Balance (FCB) was recently proposed (Mu et al., 2020), which tries to overcome some of the issues highlighted in MQ. As reported by the authors (Mu et al., 2020), traditional metrics impose a restriction on extreme coupling, but not on extreme cohesion. Since high cohesion has traditionally been regarded as desirable, previous state-of-the-art metrics have not limited its value. This has led optimization algorithms to obtain unreasonable solutions, where there exist very complex modules. One of the main objectives of FCB is to avoid the issue of over-cohesiveness, benefiting the distribution of components from very complex modules to other modules. At the same time, indirectly, the goal of FCB is to reduce the number of isolated modules within the solution. Unlike optimizing architectural quality as a whole, indirectly minimizing the maximal cohesion of individual modules forces components to be distributed to isolated clusters.

In contrast to traditional metrics, some authors have proposed a multi-objective approach, analyzing different desirable and conflicting properties of good modular organizations. In this sense, Praditwong et al. (2010) introduced two different approaches, the Equal-size Cluster Approach (ECA) and the Maximizing Cluster Approach (MCA), each one including five conflicting objectives. Interestingly enough, both approaches considered MQ as one of the objectives. Later, Barros (2012) proposed replacing MQ in ECA and MCA with a previously known metric named Evaluation Metric Function. Another related proposal was introduced by Mkaouer et al. (2015), where the authors considered a problem with seven different objectives.

Other strategies found in the literature include works based on semi-supervised approaches, where some constraints are imposed on top of clustering algorithms for the SMCP. For instance, Bavota et al. (2012) considered introducing the expertise of a software engineer for the evaluation of the solutions, while Chong and Lee (2017) leveraged graph theory to analyze the software and construct clustering constraints.

Regardless of the objective function used, as the tackled projects grow in size, evaluating all the possible solutions becomes impractical. Indeed, the SMCP has been proven to be $\mathcal{NP}$-hard (Brandes et al., 2007). Consequently, in general, exact methods are not suitable for tackling the SMCP in real contexts, since the exploration of all possible solutions might be very time consuming. On the contrary, search-based algorithms, which usually achieve solutions of high quality (not necessarily optimal) in short times, seem to be more suitable (Harman et al., 2012). In this context, evolutionary approaches, a subset of bioinspired search-based algorithms, have been prominently popularized for tackling the SMCP (Ramirez et al., 2019). Praditwong (2011) compared two encoding representations for Genetic Algorithms (GAs). Chhabra (2017) and Prajapati and Chhabra (2019) extended the Harmony Search framework to design two algorithms. Additional works studied the application of algorithms based on swarm intelligence to the SMCP, proposing an algorithm based on Artificial Bee Colony (Chhabra, 2018) and a Particle Swarm Optimization algorithm (Prajapati and Chhabra, 2018). Similarly, Gee Varghese et al. (2019) designed a method based on the Ant Colony Optimization framework. In a recent work, Mu et al. (2020) proposed the use of an HGA, which leverages domain-specific heuristics to improve the efficacy of GAs for the remodularization of software architectures. As an alternative to population-based strategies, some researchers have studied algorithms which progressively improve a single solution through small modifications until a local optimum is found. In this regard, Pinto et al. (2014) proposed an Iterated Local Search algorithm and Monçores et al. (2018) proposed another one based on Large Neighborhood Search (LNS) to study the MQ objective function. The LNS approach used an agglomerative constructive method to build high-quality initial solutions. Then, the initial solutions were improved using destroy and repair procedures. Recently, a method for the SMCP based on a Greedy Randomized Adaptive Search Procedure (GRASP) combined with a Variable Neighborhood Search (VNS) was introduced and compared with the previous LNS approach for the MQ metric (Yuste et al., 2022). In this case, the GRASP approach used a semi-greedy constructive approach, followed by a VNS procedure to improve the initial solutions. In both cases, the LNS and GRASP methods relied on a constructive procedure to build high-quality initial solutions considering the MQ objective function.

3. Software module clustering problem

The goal of the SMCP is to organize a software project such that its modularity is maximized. To achieve this objective, a method is needed to compare solutions and find the best one. Moreover, in order to enable comparisons, there needs to be a common representation of the solutions. In this problem, software is frequently modeled in a graph structure, where vertices represent artifacts and edges represent dependencies between artifacts. Furthermore, software artifacts are

usually grouped into packages or modules which, in this case, would be represented as a group of vertices. More formally, a software is modeled by an undirected weighted graph $G = (V, E, W)$, where V is the set of vertices, E is the set of edges between vertices, and W is the set of weights associated to the edges in E. Specifically, each edge is defined as a 2-tuple $(u, v) \in E$, with $u, v \in V$, and it receives a weight $w_{u,v} \in W$. In the case of unweighted graphs, it is assumed that edges have associated a weight equal to 1.

In the SMCP literature, this structure is commonly known as a Module Dependency Graph (MDG). Then, a solution for this problem is a clustering of the MDG that represents the software architecture. That is, a solution consists of a set M of disjoint subsets of V, such that $M = \{m_1, m_2, \ldots, m_k\}$, where k represents the number of modules ($1 \leq k \leq |V|$) and each m_i, with $1 \leq i \leq k$, is a disjoint subset of V. Notice that there exist two trivial solutions for the problem: (i) a solution where every vertex is assigned to the same module ($k = 1$) and (ii) a solution where each vertex forms an isolated module ($k = |V|$).

In this work, we study the SMCP based on the FCB objective function, which was proposed by Mu et al. (2020). For a given solution x, this function is defined as:

$$FCB(x) = \frac{C + \max_{m_i \in M}(d_{m_i})}{T}, \tag{1}$$

where C refers to the coupling of the project, measured as the sum of the weights of the edges that connect different modules. Notice that the weight between any pair of non-adjacent vertices is zero. Mathematically,

$$C = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{\substack{u \in m_i \\ \wedge v \in m_j}} w_{u,v}. \tag{2}$$

d_{m_i} is the cohesion of module m_i, measured as the sum of the weights of the edges between artifacts within module m_i. Mathematically,

$$d_{m_i} = \sum_{u,v \in m_i} w_{u,v}. \tag{3}$$

Finally, T is the sum of the weights of all edges of the entire architecture. As can be observed, T is a constant value that is independent of the particular clustering. Mathematically,

$$T = \sum_{e=(u,v) \in E} w_{u,v}. \tag{4}$$

Therefore, the objective of this problem is to find a solution $x^\star$ from the set of all feasible solutions X that minimizes Eq. (1), such that:

$$x^\star = \arg\min_{x \in X}(FCB(x)). \tag{5}$$

Finally, let us exemplify the evaluation of the FCB value with the small synthetic MDG in Fig. 1. In that figure, we present the modular organization of a software project (i.e., a solution x) with 11 artifacts ($v_1, v_2, \ldots, v_{11}$) and 3 modules (m_1, m_2, m_3). As aforementioned, vertices of the graph represent artifacts of the system and edges represent dependencies between artifacts. In this representation, we can also observe that some connections are produced between pairs of artifacts assigned to the same module, but others between pairs of artifacts belonging to different modules. Let us assume that the weight of each edge is equal to one. In order to evaluate FCB, we first compute the value of the coupling of the architecture. In this case, $C = 4$, since there exist four different edges in the solution that connect vertices belonging to different modules. Then, we proceed to compute the cohesion of each module in the example. Regarding module m_1, its cohesion is $d_1 = 2$, since there exist two pairs of vertices that are connected inside m_1. In a similar fashion, we can obtain the cohesion values for the rest of the modules: $d_2 = 3$ and $d_3 = 4$. Therefore, the maximum cohesion of any module is $d_3 = 4$. Next, we can calculate the value of the constant T as the sum of all edges in the solution, which results in $T = 13$. Finally, we can compute the quality of the solution x as depicted in the figure, resulting in $FCB(x) = 0.62$.

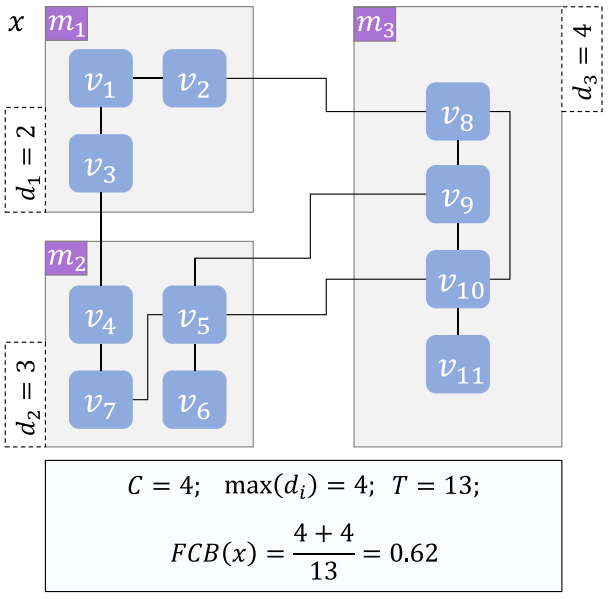


Fig. 1. Representation of a solution x for a software project and evaluation of its FCB value.

4. Algorithmic proposal

In order to tackle the SMCP, we propose an algorithm based on the Variable Neighborhood Search (VNS) methodology, a general framework for solving optimization problems. This framework was first introduced by Mladenović and Hansen (1997) and it is based on the idea of performing systematic/stochastic changes in the neighborhood structure of a search algorithm. Here, a neighborhood $N(x)$ is conceived as the set of alternative solutions that can be obtained by performing a specified move to a starting point x. A move can be defined as an operation that, applied to a feasible solution, results in another feasible solution. Neighborhood structures are usually explored with a local search, a procedure that performs moves upon the incumbent solution to find better solutions. If none of the solutions in the neighborhood is better than a solution x, then x is said to be a local optimum.

Derived from the main ideas within the VNS framework, there exist several schemes. Among the classic and best-known ones, we can find Basic VNS, Reduced VNS, General VNS or Variable Neighborhood Descent, among others. These variants together with the main ideas behind the VNS framework can be found in Hansen et al. (2010, 2017). Even though it was originally thought for single-objective optimization problems using a single thread, it has been extended and it is possible to find parallel (Duarte et al., 2016) and multi-objective (Duarte et al., 2015) implementations of VNS, among other extensions (Brimberg et al., 2023; Granata and Sgalambro, 2023; Mladenović et al., 2022; Nadar et al., 2023).

To select the most suitable variant of VNS for the SMCP, we analyzed the most common strategies. Although all of them are based on the same ideas, they differ in the way they perform the changes in the neighborhood structure (i.e., stochastic, deterministic, or both). Reduced VNS implements the ideas of VNS in a shake procedure, which has a stochastic behavior. On the contrary, VND explores several neighborhoods in a deterministic way. Basic VNS extends Reduced VNS by performing an exhaustive exploration of a neighborhood within a local search procedure, in addition to the stochastic exploration performed by the shake procedure. Finally, GVNS extends VND by introducing a shake procedure which introduces diversification in the search through a stochastic exploration. In this paper, we have selected

the GVNS scheme to tackle the SMCP since, as it will be discussed in Section 4.1, we need to explore several neighborhoods belonging to different categories that complement each other.

The pseudocode of the GVNS procedure proposed in this paper is presented in Algorithm 1. This method receives three variables: an initial solution (x), the maximum size of the perturbation to be performed by the shake procedure (k_{max}), and the maximum allowed time for the algorithm to run (t_{max}). In this case, the starting solution is built by a random constructive procedure. Initially, the algorithm enters an outer loop (steps 3–16) until the maximum time t_{max} is reached. At each iteration of the loop, k, which determines the size of the perturbation performed by the shake procedure, is set to 1 (step 4). Then, the procedure enters into an inner loop (steps 5–14) where the solution is first shaken (step 6) and then improved by a VND procedure (step 7). The shake procedure randomly performs a set of moves within a given neighborhood (i.e., it performs a random exploration of the neighborhood). The amount of moves is dictated by the value of k. Then, the VND explores two or more neighborhoods with a local search procedure (i.e., it performs a deterministic exploration of the neighborhoods) and returns the best solution found with respect to all neighborhoods explored. Finally, the neighborhood change operation is applied (steps 8 to 13). If the new solution found (x'') is better than the previous best one (x), then the new solution becomes the best overall solution (step 9) and k is reset to 1 (step 10). Notice that the quality of both solutions is evaluated with the function FCB (step 8), which evaluates a solution as defined in Eq. (1). If that is not the case, then variable k is incremented (step 12). Once the maximum allowed time is reached, the best overall solution found is returned (step 17).

Algorithm 1 Algorithmic proposal

```
1: GVNS (x, k_max, t_max)
2: t = CPUTime();
3: while (t < t_max) do
4:    k = 1;
5:    while (k ≤ k_max) do
6:       x' = Shake(x, k);
7:       x'' = VND(x');
8:       if (FCB(x'') < FCB(x)) then
9:          x = x'';
10:         k = 1;
11:      else
12:         k = k + 1;
13:      end if
14:   end while
15:   t = CPUTime();
16: end while
17: return x;
```

As mentioned above, the initial solution for the GVNS procedure is built at random. In particular, the construction is performed as follows. First, $|V|$ different empty modules are created. That is, we create one empty module per vertex in the MDG. Then, we assign each vertex in the solution to a random module m_i. The module is chosen following a uniform distribution from the set of existing modules. Finally, once every vertex belongs to a module in the solution, the modules that do not contain any vertices are removed from the solution. The resulting initial solution is then improved by the GVNS procedure.

The general design of a GVNS procedure has to be particularized for the specific problem tackled by defining the set of neighborhood structures that will be used either in the deterministic exploration performed by the VND component or in the stochastic exploration within the shake method. In the following section, we proceed to describe the neighborhood structures proposed for the GVNS procedure.

4.1. Neighborhood structures

In this work, we study six distinct neighborhoods for the SMCP following the categorization presented in a previous work (Yuste et al., 2022), where three categories were proposed. It is important to notice that many neighborhoods have been previously proposed for related optimization problems. Particularly, in Yuste et al. (2022), four neighborhoods were explored for the MQ objective function, while in Monçores et al. (2018) three destroy and four repair methods were proposed for the same objective function. In this paper, among the six neighborhood structures studied, two can be considered as classical neighborhood structures for combinatorial optimization problems, two neighborhoods were previously proposed for the MQ objective function, and two are novel neighborhoods designed for the FCB. The proposed neighborhoods are classified in one of the tree categories, according to the impact that the move that defines them has on the number of modules. The first category contains those neighborhoods defined by moves that are not designed to modify the number of modules. The second set of neighborhoods contains neighborhoods defined by moves that increment the number of modules of the solution. Finally, the neighborhoods defined by moves that decrease the number of modules fall in the third category. Specifically, among the six neighborhoods proposed, N_1 and N_2 belong to the first category, N_3 and N_4 belong to the second category, and N_5 and N_6 belong to the third category. The proposed neighborhoods will be empirically analyzed to consider its inclusion in the deterministic or in the random exploration. Particularly, in Section 6.2.1, we study the contribution of the neighborhoods and select some of them to be included in the VND procedure. Furthermore, in Section 6.2.2 we empirically determine the order in which the selected neighborhoods are explored. It is worth mentioning that the deterministic exploration of the selected neighborhoods will be performed using a local search procedure based on a first improvement strategy. In Section 6.2.3 we determine which neighborhood is more suitable to be explored within the shake procedure to complement the neighborhoods selected for the VND. Notice that the neighborhood explored within the shake procedure is stochastically traversed. Next, we describe each of the neighborhoods in detail.

4.1.1. N_1: Insertions

The first neighborhood proposed (N_1) is defined by an insertion operator, which is a classic move in heuristic optimization (Cavero et al., 2022; Yuste et al., 2022). This operator involves relocating a vertex from its current module to a different one. This operator is defined as $x' \leftarrow \texttt{Insert}(x, v, m, m_i)$, where x is the incumbent solution before applying the move, x' denotes the resulting solution, m is the module that contains v originally, and m_i is the module that contains v after the move. More formally:

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

where M represents the set of modules in x and $1 \leq i \leq |M|$. Therefore, the number of possible moves to consider for a given solution is equal to $|V| \cdot (|M| - 1)$.

4.1.2. N_2: Swaps

The second neighborhood (N_2) is characterized by a swap operator, which is also a classic move in heuristic optimization (Gil-Borrás et al., 2021; Yuste et al., 2022). This operator consists of interchanging two vertices located in two different modules. This operator is defined as $x' \leftarrow \texttt{Swap}(x, v_i, v_j, m_k, m_l)$, where x is the incumbent solution, x' is the resulting solution, v_i is a vertex located in module m_k, and v_j is a vertex located in module m_l. More formally:

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

where M represents the set of modules in x, $1 \leq k, l \leq |M|$, and $1 \leq i, j \leq |V|$. Thus, the number of moves to consider for a given solution is less than $|V| \cdot \frac{(|V|-1)}{2}$, since vertices in the same module cannot be swapped.

4.1.3. N_3: Splits

The third neighborhood (N_3) is characterized by a split operator, which consists of dividing a module into halves. This operator is denoted as $x' \leftarrow \texttt{Split}(x, m, m_1, m_2)$, where x is the solution before applying the move that contains the module m and x' is the solution that results after the move that contains the new modules m_1 and m_2 instead of m. Furthermore, m_1 contains half of the vertices that belonged to m, and m_2 contains the other half. More formally:

$$N_3(x) = \{x' \leftarrow \texttt{Split}(x, m, m_1, m_2) : \forall\, m \in M \; / \; m_1 \cup m_2 = m,\ |m_1| = \lfloor \frac{|m|}{2} \rfloor \wedge |m_2| = \lceil \frac{|m|}{2} \rceil \},$$

where M represents the set of modules in the solution x. Notice that all the possible distributions of the vertices into the resulting modules m_1 and m_2 are considered within the neighborhood structure, as long as both modules are equal in size (or almost equal in the case of an odd number of vertices). Therefore, the size of the entire neighborhood is $\sum_{i=1}^{|M|} \frac{|m_i|!}{\lfloor |m_i|/2 \rfloor ! (|m_i| - \lfloor |m_i|/2 \rfloor)!}$.

4.1.4. N_4: Extractions

The fourth neighborhood (N_4) is defined by an extraction operator. As defined in Yuste et al. (2022), this operator selects a combination of vertices from one or more modules and inserts them into a new empty module. This operator is denoted as $x' \leftarrow \texttt{Extract}(x, l, u, L)$, where x is the original solution, x' is the solution that results from the application of the operator, and L is any possible set of vertices. The size $|L|$ of the set ranges between l and u, which indicate the minimum and maximum possible size of the new empty module. As a result, the solution x' will have a new module that contains all the vertices of L. More formally:

$$N_4(x) = \{x' \leftarrow \texttt{Extract}(x, l, u, L) : \forall\, L \subseteq V \; / \; 1 \leq l \leq |L| \leq u \leq |V|\}.$$

Notice that all possible combinations of $|L|$ vertices are considered, for any number of vertices as long as $l \leq |L| \leq u$. Therefore, the size of this neighborhood is $\sum_{n=l}^{u} \frac{|V|!}{n!(|V|-n)!}$.

4.1.5. N_5: Merges

The fifth neighborhood (N_5) is defined by a merge operator, where two modules are combined into one. We define this operator as $x' \leftarrow \texttt{Merge}(x, m_i, m_j)$, where x is the incumbent solution, x' is the resulting solution, and m_j is the module that will be merged into module m_i. That is, m_i will contain all the vertices in $m_i \cup m_j$. More formally:

$$N_5(x) = \{x' \leftarrow \texttt{Merge}(x, m_i, m_j) : \forall\, m_i, m_j \in M \; / \; i \neq j\},$$

where M represents the set of modules in x. Then, the size of the entire neighborhood is $|M| \cdot \frac{|M|-1}{2}$.

4.1.6. N_6: Destructions

The last neighborhood (N_6) is defined by a destruction operator, as defined in Yuste et al. (2022). This operator removes one module, and the affected vertices are relocated into other existing modules. We define this operator as $x' \leftarrow \texttt{Destroy}(x, m, O, D)$, where x is the incumbent solution, x' is the resulting modularization, m is the module that will be destroyed, O is an array of the vertices that are contained in m, and D is an array of modules. Both O and D are ordered, and there exists a correspondence between the two lists such that the ith vertex in O will be placed in the ith module in D. More formally:

$$N_6(x) = \{x' \leftarrow \texttt{Destroy}(x, m, O, D) : \forall\, m \in M \; / \; m \notin D,\ |O| = |D| = |m|\},$$

where M represents the set of modules in x. Notice that all modules are considered to be destroyed. Moreover, for each module, all possible distributions of their vertices into other modules are considered. Therefore, the size of the neighborhood is $(|M| - 1)^{|D|} \cdot |M|$, with $|D| \leq |V|$.

5. Advanced strategies

Here, we describe two strategies to enhance the performance of the most time-consuming subroutines of the proposed algorithm: the local search procedures within the VND. Local search procedures need to perform many moves in the search space and evaluate the solution that results after each move. Therefore, to enhance their performance, we first introduce an efficient computation of the quality of the solution (see Section 5.1). Then, we detail a strategy to explore only promising areas of the search space (see Section 5.2).

5.1. Efficient computation of the objective function

The FCB objective function (see Eq. (1)) is calculated as the sum of the coupling and the maximum cohesion of the solution, divided by a constant value. Therefore, an initial naive idea is to calculate this value only once, as has been proposed for other metrics (Yuste et al., 2022). Furthermore, analyzing the purpose of the division, we noticed that it is devoted to normalize the value of the function in $(0, 1]$, allowing comparison of the modularity of different software projects. However, different solutions for the same project (as is the case with the search procedure) can be successfully compared without performing this division. Therefore, we can compare the quality of different solutions for a given MDG with the following simplified function:

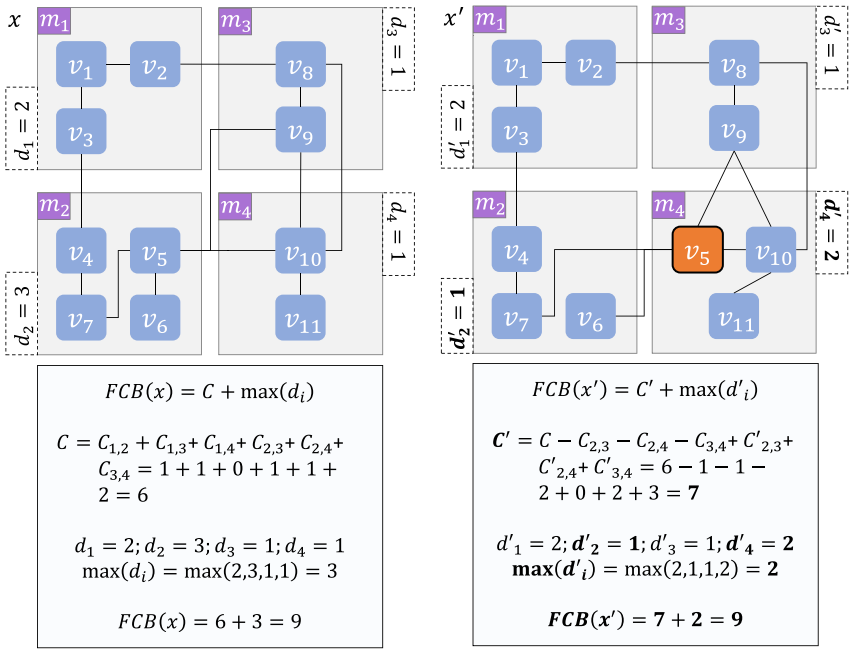
$$FCB(x) = C + \max_{m_i \in M} (d_{m_i}). \tag{6}$$

The resulting objective function is computed as the sum of the coupling between the modules of the architecture and the maximum cohesion of any module. However, these values do not need to be calculated from scratch each time a move operation is performed. Instead, if the values of the cohesion of each module and the coupling between each pair of modules of the starting solution are stored, we will only need to update the cohesion and coupling values of the affected modules after a move operation.

In Fig. 2, we represent an example of the use of these strategies. On the left side, we represent a solution x that has already been evaluated. Therefore, the cohesion of each module and the coupling between each pair of modules are already known. Specifically, let $d_1 = 2$ be the cohesion of module m_1 (similarly, $d_2 = 3$, $d_3 = 1$, and $d_4 = 1$). Furthermore, let $c_{1,2} = 1$ be the coupling between modules m_1 and m_2 (similarly, $c_{1,3} = 1$, $c_{1,4} = 0$, $c_{2,3} = 1$, $c_{2,4} = 1$, and $c_{3,4} = 2$). We also represent, under the solution, the detailed evaluation of the objective function. On the right part, we represent a modified solution x', obtained by moving the vertex v_5 to module m_4. In this case, we also represent the detailed evaluation of the FCB objective function, but highlighting in bold type font only the affected values after the move. As it can be observed the cohesion of modules m_1 and m_3 has not changed. Therefore, we only need to update the cohesion of the affected modules: m_2 and m_4. Similarly, we can avoid recalculating the coupling of the whole architecture from scratch, since only the coupling between modules m_2 and m_4 and m_3 and m_4 is affected.

5.2. Reduction of neighborhoods

Here, we extend a strategy presented in Yuste et al. (2022) to explore only promising solutions, thus reducing the size of the search space explored. This strategy is built on top of a theorem introduced by Köhler et al. (2013). There, the authors state that in the best solution for this problem, all vertices will be connected to at least one vertex within the same module. That is, for each vertex, there will be at least one adjacent vertex located in the same module. Thus, we can focus on exploring only promising solutions, obtained by performing move operations that place a vertex in a module where there is at least one adjacent vertex.



(a) Solution x. (b) Solution $x' = \texttt{Insert}(\texttt{x}, \texttt{v}_5, \texttt{m}_2, \texttt{m}_4)$.

Fig. 2. Example of the evaluation of two solutions x and x' before and after an `Insert` operation of the vertex v_5 into module m_4 in solution x.

6. Experimental results

Here, we present some experimental results performed to configure the proposed approach. First, we outline the dataset used for the experimentation in Section 6.1. Then, we present some preliminary experiments needed to configure the search parameters of our proposal in Section 6.2. Finally, in Section 6.3, we perform a comparison of the results obtained by our proposal with the results obtained by the state-of-the-art method.

6.1. Dataset

In order to evaluate the performance of the proposed method, we employ a set of 124 previously published instances (Monçores et al., 2018). This dataset is made up of real-life software projects of varying sizes, comprising projects with up to 1161 artifacts and 11 722 dependencies. For the preliminary experiments, we selected 14 instances at random to form a reduced dataset. Particularly, the instances contained in the preliminary experiments are: apache_ant_taskdef, gae_plugin_core, javacc, joe, jscatterplot, jtreeview, jxlsreader, lwjgl-2.8.4, mod_ssl, net-tools, nmh, regexp, star, and wu-ftpd-1.

6.2. Preliminary experiments

In this section, we perform some preliminary experiments to adjust the search parameters of our approach or to illustrate the behavior of the proposed strategies. The objective is to configure the best possible variant of our algorithm to tackle the problem studied in this paper. Each configuration of the tested algorithms is executed just once. However, since the initial solutions are built at random, in order to perform a fair comparison, we ensure that all the configurations tested in each experiment start from the same initial solution.

6.2.1. Contribution of each neighborhood

In Section 4, we proposed six neighborhoods for the SMCP. In particular, we identified three different categories of neighborhoods and proposed two neighborhoods per category. However, including all the neighborhoods in a VND procedure might result in a very heavy method. Furthermore, some of the neighborhoods previously defined are very large and should be bounded in order to find a balance between performance and time consumption. In particular, we have bounded N_3, N_4 and N_6. In the case of N_3 we only explore $|V|$ solutions. Each of them is constructed by dividing the vertices of a module considering their adjacency. In particular, for each module, we consider the vertices in the module one by one as a candidate seed. The seed is inserted into a new empty module together with half of the vertices from the same module that have the strongest dependencies towards the seed in terms of adjacency. The remaining vertices are inserted into another new empty module. Since all vertices are considered as possible seeds, the size of this neighborhood is $|V|$. Similarly, we bound N_4 by fixing the parameters l and u to 2 and 3 vertices, respectively. Finally, we also bound the size of N_6. This neighborhood structure considers the destruction of every module in the solution and the insertion of its vertices into other modules. However, we do not consider all the possible insertions when a module is destroyed. Instead, each vertex is inserted only in the module that has the most dependencies towards that vertex. Therefore, the size of N_6 in practice is equal to the number of modules $|M|$.

Here, we analyze the benefits of exploring each neighborhood structure in isolation. The aim is to identify the most promising neighborhoods in each category. To this end, we explore each neighborhood in isolation using a local search procedure. In the local search, a first improvement strategy is followed. To execute a reasonable comparison, we constructed a random solution for each of the tested instances and provided the same solution to each of the six local search procedures. Each method stopped its execution when no further improvements were found in the explored neighborhood, reporting the CPU time and the quality of the solution found in terms of FCB.

In Table 1, we summarize the results obtained by each local search method, together with the results provided by the random initial constructions. In particular, we report the category of the neighborhood structure (Category), the average value of the objective function (O.F.), the deviation towards the objective value of the best solution (Dev. (%)), the number of instances for which the best solution is obtained (# Best), the duration of the search process (CPUt (s)), and the average

Table 1
Comparison of the results obtained by exploring different neighborhoods in isolation.

Category	Method	O.F.	Dev. (%)	# Best	CPUt (s)	Avg. Improv.
–	Constructions	0.9524	53.90%	0	0.04	–
1	Insertions (N_1)	**0.6753**	**7.58%**	**7**	3.33	204.07
1	Swaps (N_2)	0.8425	36.67%	0	0.36	54.07
2	Splits (N_3)	0.9494	53.44%	0	0.06	0.71
2	Extractions (N_4)	**0.8862**	**43.47%**	0	0.94	39.21
3	Merges (N_5)	0.8059	29.31%	0	0.06	15.64
3	Destructions (N_6)	**0.6824**	**8.29%**	**7**	0.11	26.07

number of improving moves that have been performed during the exploration of the neighborhood structure (Avg. Improv.). As it is illustrated, the neighborhoods N_1 and N_6 achieved the best results, with a quality of 0.6753 and 0.6824 on average, respectively. Moreover, these neighborhoods obtained the best results for 7 instances. Furthermore, in both cases, the exploration of the neighborhood structures resulted in a higher number of improvements than the exploration of the other neighborhood tested in the same category. However, N_1 was also one of the most time-consuming neighborhood structures to explore. Therefore, we selected both neighborhoods, N_1 and N_6, to be part of the algorithm.

Finally, to complement the previous neighborhoods, we selected the best neighborhood in the second category: N_4. One of the main ideas of VNS is to explore multiple neighborhood structures, and it is desirable that neighborhoods included in the method complement each other. In this case, exploring at least one neighborhood structure from each category gives flexibility to the algorithm. Despite neighborhoods from the second category perform poorly in isolation, without neighborhood structures of the second category, the algorithm would not be able to increase the number of modules, but only to maintain or reduce it. Thus, we configured the VND procedure with three neighborhoods: N_1, N_4, and N_6.

6.2.2. Order of the neighborhoods

Within the VNS methodology, the VND variant is characterized by its exploration of the neighborhoods in a deterministic way. If an improvement is made while exploring a neighborhood, the procedure is reset to explore the list of neighborhoods from the beginning once again. Otherwise, if no improving solution is found in the current neighborhood, the VND continues exploring the next neighborhood in the list. This procedure is performed repeatedly until it is unable to improve the current solution. Therefore, as each neighborhood is often explored more times than the following ones, the order in which they are structured might be impactful on the performance of the algorithm in either time or quality.

Here, we investigate all possible combinations of the three neighborhoods that were selected in Section 6.2.1 to be part of the algorithm: N_1, N_4, and N_6. We have explored the three neighborhoods in a VND procedure, in different orders, but ensuring that each of the six methods began its exploration from the same set of starting solutions, which were built at random.

In Table 2, we report the results obtained for each ordering of the considered neighborhoods (i.e., for each VND configuration). As illustrated, each VND method performed differently. Furthermore, in some cases, the differences among the methods in CPU time or deviation are remarkable. In particular, we observed that the combinations that start by exploring N_6 are faster, whereas those that explore N_1 before N_6 are slower. Taking into account the quality of the solutions obtained, the last ordering (N_6, N_4, N_1) performed the best, with an average score of 0.5971. Moreover, by exploring the neighborhoods in this order, the method obtained the best results for 7 out of 14 instances in a very short computing time. Correspondingly, we configured the VND component of the proposed GVNS to explore neighborhoods N_6, N_4, and N_1, in that order.

Table 2
Comparison of the results obtained by a VND procedure when exploring the selected neighborhoods in all possible orders.

Order	Avg. O.F.	Avg. dev. (%)	# Best	Avg. CPUt (s)
N_1, N_4, N_6	0.6055	6.93%	3	1.94
N_1, N_6, N_4	0.6053	6.89%	3	2.85
N_4, N_1, N_6	0.6163	8.40%	3	2.29
N_4, N_6, N_1	0.6045	5.96%	4	0.89
N_6, N_1, N_4	0.6093	7.53%	4	**0.15**
N_6, N_4, N_1	**0.5971**	**4.88%**	**7**	0.18

6.2.3. Comparison of shake procedures

The shake procedure is used by the GVNS method to diversify the search by choosing a random solution after performing k moves in a predefined neighborhood. Therefore, this method is highly dependent on the neighborhood chosen, but also on the neighborhoods included in the VND method within the GVNS. In this paper, we study three different shake procedures, one for each of the three categories of neighborhoods introduced in Section 4. Particularly, we have selected the neighborhoods that were discarded in Section 6.2.1 to configure the three shake procedures: $Shake_1$ uses N_2, $Shake_2$ uses N_3, and $Shake_3$ uses N_5. To perform a fair comparison, we executed three GVNS algorithms configured with the VND selected in Section 6.2.2 and each of the shake procedures proposed. We also tried different values of k_{max} (10, 20, and 30) and k-step was set to one (i.e., k is increased in one unit per iteration).

Since the neighborhoods considered for the shake procedures are considerably different (an operation in N_2 affects two vertices, an operation in N_3 affects half of the vertices of a module, and an operation in N_5 affects all the vertices of two modules), the magnitude of the perturbation performed in the solution, given the same value of k, might also be different. Taking this into account, at each step of the algorithm, the number of `Swap` moves performed by $Shake_1$ is $\max(k, (|V| * k)/100)$. The number of `Split` moves performed by $Shake_2$ is $\min(\max(k, (originalNumberOfModules * k)/100), originalNumberOfModules)$. Finally, the number of `Merge` moves performed by $Shake_3$ is $\min(\max(k, (originalNumberOfModules * k)/100), originalNumberOfModules)$.

In Table 3, we present, for each configuration and maximum value of k: the neighborhood used in the shake procedure (Shake), the average quality of the best solution obtained for each instance (Avg. O.F.), the average deviation from the best solution found in the experiment (Avg. dev. (%)), the number of instances for which the best solution was obtained (# Best), and the average execution time consumed (Avg. CPUt (s)). Notice that the comparisons between the shake procedures were made independently for each maximum value of k (i.e., deviation and best solutions were calculated separately for $k = 10$, $k = 20$, and $k = 30$). As it can be observed, the configuration with the shake based on swap moves was the one that obtained the best results, beating the other configurations in terms of average quality, average deviation, and number of best solutions found for any of the maximum values of k tried in the experiment. Therefore, we select the $Shake_1$ procedure, based on the N_2 (`Swap` operator) to configure the shake component of our GVNS.

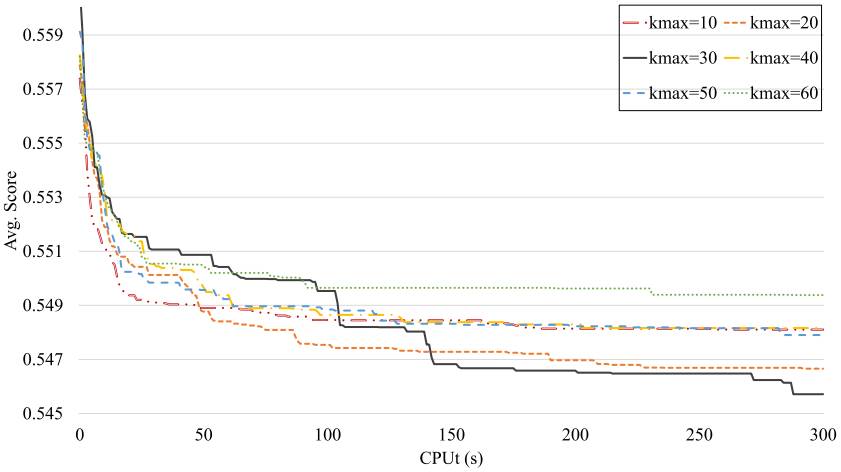


Fig. 3. Comparison of the average quality of the best solution found for the instances in the reduced dataset over time with different values of k_{max}.

Table 3
Comparison of a GVNS algorithm with different shake procedures and different values of k_{max}. Each configuration was executed for one iteration and started from the same initial solutions.

k_{max}	Avg. O.F.			Avg. dev. (%)			# Best			Avg. CPUt (s)		
	10	20	30	10	20	30	10	20	30	10	20	30
$Shake_1$	0.57	**0.56**	**0.56**	**0.37**	**0.15**	**0.15**	**12**	**13**	**13**	3.79	4.86	6.31
$Shake_2$	0.59	0.59	0.59	5.31	5.63	5.63	3	2	2	1.92	1.72	2.04
$Shake_3$	0.57	0.57	0.57	1.41	1.73	1.73	5	3	3	8.29	18.56	31.46

6.2.4. Maximum value of k

In order to tune the k_{max} parameter of GVNS, we performed an experiment to analyze the performance of the method for different values of k_{max} over time. In particular, we configured the GVNS to use the shake procedure based on swap moves and explore neighborhoods N_6, N_4, and N_1 within the VND component in the specified order, setting the maximum running time to 300 s per configuration. Notice that, at each iteration of the algorithm, the number of moves performed by the shake procedure is equal to $\max(k, (|V| * k)/100)$. Therefore, the size of the perturbation is related to the size of the graph. In Fig. 3, we represent the average quality of the best solutions obtained for each instance at any instant during a time period of 300 s, for different values of k_{max}: 10, 20, 30, 40, 50, and 60. As it can be observed, depending on the time horizon considered, different values of k_{max} were the most beneficial. For instance, the method configured with $k_{max} = 10$ achieved the best results in the first 50 s, while the method configured with $k_{max} = 20$ was the best configuration in the time interval between 50 and 100 s. Finally, the method configured with $k_{max} = 30$ was the best one after 120 s. In this sense, the configuration of the algorithm should be set depending on the particular running context. Although $k_{max} = 10$ allowed the algorithm to improve the solutions faster than other values, $k_{max} = 30$ achieved better long-term performance. Therefore, we decided to set the value of k_{max} to 30 for the configuration of the approach.

6.2.5. Stopping criterion

In search-based optimization strategies, the stopping criterion is often related to a maximum number of iterations, a maximum execution time, a maximum number of iterations without improvement, or a combination of them. In this work, we propose a stopping criterion based on a maximum number of consecutive iterations without improvement, allowing the algorithm to avoid further processing when the procedure stagnates. Here, we configure the algorithm to stop the execution after 5, 10, 15, and 20 iterations without improving the solution. The rest of the algorithm is configured with the parameters selected in previous experiments: a swap shake procedure, $k_{max} = 30$, and a VND approach that explores neighborhoods N_6, N_4, and N_1, in that order.

Table 4
Comparison of the results obtained with different configurations of the stopping criterion. In particular, the algorithm has been set to stop after 5, 10, 15, or 20 iterations without improvement.

Iterations without improvement	Avg. O.F.	Avg. dev. (%)	# Best	Avg. CPUt (s)
5	0.5509	0.27%	10	30.42
10	0.5501	0.09%	12	54.57
15	0.5496	0.00%	14	87.07
20	0.5496	0.00%	14	99.95

We present the obtained results for each configuration of the stopping criterion (Iterations without improvement) in Table 4. As expected, increasing the number of maximum consecutive iterations without improvement allows the algorithm to further explore the search space, resulting in solutions of better quality at the expense of additional time. However, the relative improvement obtained between consecutive configurations decreases as the number of maximum iterations increases. In fact, we barely observe an improvement in terms of quality by increasing the number of maximum iterations without improving from 10 to 15, and the improvement is non-existent after increasing this number from 15 to 20. Therefore, we decided to configure the stopping criterion of the proposed algorithm to halt the search after 15 consecutive iterations without improving the solution.

6.2.6. Influence of the advanced strategies

In this section, we test the influence of the advanced strategies on the overall performance of the proposed procedure. The results of this experiment are summarized in Table 5. All the results reported in the table are obtained with the configuration of our GVNS obtained from previous experiments. That is, the shake procedure utilizes the swap neighborhood, $k_{max} = 30$, and the VND is configured to explore the neighborhoods N_6, N_4, and N_1, in that order. The halting criterion is set to stop after 15 consecutive iterations without improving the solution.

Particularly, in Table 5, we report four different rows depending on the advanced strategies used by the algorithm. In the first row, denoted as None, we report the results obtained by the algorithm without any

Table 5
Comparison of the outcomes obtained with different advanced strategies: None, efficient evaluation (EE), and reduction of the size of neighborhoods (RN).

Advanced strategies	Avg. O.F.	Avg. CPUt (s)
None	0.5762	89 479.66
EE	0.5762	509.63
RN	0.5703	250.59
EE+RN	0.5703	6.56

of the advanced strategies. In the second row, denoted as EE (Efficient Evaluation) we report the results obtained by using the strategy introduced in Section 5.1. In the third row, denoted as RN (Reduction of the size of Neighborhoods), we report the results obtained by the algorithm using the strategy introduced in Section 5.2. Finally, in the fourth row, denoted as EE+RN, we report the results obtained by the algorithm using the two previous strategies simultaneously. As it can be observed, each advanced strategy, EE and RN, used in isolation, is able to reduce the time consumption of the algorithm in two orders of magnitude, from an average time per instance of 89479.66 s to 509.63 and 250.59 s, respectively. In addition, when combining both of them, the time consumption of the algorithm is further reduced to an average running time per instance of 6.56 s, i.e., four orders of magnitude less than the time needed by the algorithm without any of the proposed strategies.

In the results reported in Table 5 it is also noticeable that there are variations in the average FCB value between the two first configurations of the table and the other two. This is due to the addition of the strategy which reduces the size of the neighborhoods. Since we are exploring the neighborhoods with a first improvement approach (i.e., the first neighbor found that is better than the current solution is selected), reducing the size of the neighborhoods often results in a different search pattern. Thus, the solutions obtained can be different. This pattern has also been reported in previously related research (Yuste et al., 2022).

Finally, we would like to note that for this experiment we only used the ten smallest instances from the reduced dataset. This is due to the fact that the time needed by the version of the method without any of the advanced strategies for the largest instances was inacceptable for practical purposes. However, the larger the size of the instance considered, the greater the benefits of the proposed strategies.

6.3. Final experiments

Finally, once we have configured the search parameters of the proposed algorithm, we perform an experiment to compare our proposal with the best algorithm known in the literature (i.e., the HGA proposed by Mu et al. (2020)) on a set of 124 real software projects published in the literature (Monçores et al., 2018). The outline of the final configuration of our GVNS method is the following: the algorithm runs until 15 consecutive iterations are completed without improving the solution; the shake procedure utilizes the swap neighborhood; $k_{max} = 30$; the VND is configured to explore neighborhoods $N_6, N_4,$ and N_1, in that order; and the algorithm incorporates the strategies introduced in Section 5.

All experiments were run on a Microsoft Windows 10 Pro 10.0.19042 x64 operating system, with an AMD EPYC 7282 @ 2795 Mhz CPU with 8 cores and 8 GB RAM. The proposed method was implemented in Java 17.0.1. In addition, we implemented it using the Metaheuristic Optimization framewoRK (MORK) project (Martín-Santamaría et al., 2022). Unfortunately, the original implementation of the HGA algorithm is not public. Therefore, we implemented the HGA algorithm as described by Mu et al. (2020) in both Java and Matlab (R2021b Update 1). Nevertheless, we found that our implementation in Matlab (as originally proposed by the authors) was more efficient than our implementation in Java, since it took advantage of the fast calculation of matrix operations available in the platform, which is an important issue in the design of the algorithm. Therefore, in the comparison, we used the implementation made in Matlab.

Table 6
Comparison of the results obtained with the method proposed in this research, GVNS, and the best known algorithm, HGA (Mu et al., 2020).

Size of instances	Method	O.F.	Dev.	# Best	CPUt (s)	p-value
Small (64)	GVNS	**0.6448**	**<0.01%**	**64**	**4.29**	
	HGA	0.7234	14.10%	12	27.56	<0.001
Medium (29)	GVNS	**0.5312**	**<0.01%**	**29**	**23.89**	
	HGA	0.7215	46.67%	0	538.31	<0.001
Large (18)	GVNS	**0.5075**	**<0.01%**	**18**	**103.73**	
	HGA	0.7555	54.05%	0	7,629.85	<0.001
Very large (13)	GVNS	**0.4901**	**<0.01%**	**13**	**1 084.09**	
	HGA	0.7842	65.97%	0	254,173.61	0.001
All (124)	GVNS	**0.5821**	**<0.01%**	**124**	**136.51**	
	HGA	0.7340	32.96%	12	27,894.91	<0.001

In Table 6, we present the outcomes obtained by the two algorithms, HGA and GVNS, for all instances, divided in four different groups, following the same distribution given by Monçores et al. (2018): instances with less than 79 vertices (Small), instances with less than 190 vertices (Medium), instances with less than 400 vertices (Large), and instances with more than 400 vertices (Very large). Specifically, we outline the average FCB value (O.F.), the average deviation from the best solution found in this experiment for each instance by any of the methods compared (Dev.), the number of instances for which the obtained solution was better or equal to the solution obtained by the other method (# Best), the average execution time consumed (CPUt (s)), and the p-value obtained for each group of instances according to the Wilcoxon's Signed Rank Test (p-value). As can be noticed, the solutions obtained by the GVNS approach present a better quality on average than the solutions obtained by the HGA method. Moreover, GVNS obtained solutions that were closer to the best ones found (with less than a 0.01% of deviation) than HGA (with a deviation of 32.96% on average). Furthermore, GVNS obtained the best results for 124 out of 124 instances, whereas HGA obtained the best results for 12 of the smallest instances in the dataset. Finally, it can be seen that GVNS was three orders of magnitude faster than HGA. In addition, the results are statistically significant, according to the Wilcoxon's signed rank test, with $p < 0.001$. In Table B.8 of Appendix B, we report the detailed results of the Wilcoxon's signed rank test.

7. Conclusions and future research

In this work, we have presented an algorithm based on the GVNS methodology for the SMCP. The proposed algorithm includes both a diversification phase and an intensification phase. In the diversification phase, a shake procedure perturbs the solution to escape from local optima. In the intensification phase, a VND procedure finds a local optimum within several neighborhood structures. Since the diversification phase is performed by a shake procedure, the method does not need to rely on a randomized constructive procedure to introduce diversification into the search process, in contrast to other methods in the literature (Yuste et al., 2022). For this reason, the initial solutions for the GVNS approach are built at random. Moreover, the proposed GVNS schema introduces some adaptiveness in the shake procedure, by adapting the size of the perturbation depending on the size of the instance at hand.

The proposed algorithm has been used to study a novel quality metric for software modularization: FCB. To improve the efficiency of the proposed approach for the FCB objective function, two advanced strategies have been included in the proposal: (i) an efficient computation of the quality of the solutions and (ii) a reduction of the size of the explored neighborhoods.

Table A.7
Best solutions found by each algorithm for all instances in the dataset.

Instance	$\|V\|$	$\|E\|$	GVNS		HGA (Mu et al., 2020)	
			FCB	CPUt (s)	FCB	CPUt (s)
squid	2	2	1.0000	0.09	1.0000	5.42
small	6	5	0.6000	0.08	0.6000	5.45
random	13	30	0.6400	0.22	0.6400	19.27
compiler	13	32	0.6875	0.38	0.7813	5.32
regexp	14	20	0.6892	0.25	0.6892	35.73
netkit-ping	15	15	1.0000	0.15	1.0000	15.43
nss_ldap	15	16	0.9957	0.18	0.9957	18.37
lab4	15	18	0.5000	0.19	0.5000	6.81
jstl	15	20	0.3333	0.16	0.3333	8.94
nos	16	52	0.6600	0.49	0.7000	21.07
lslayout	17	43	0.6977	0.48	0.7442	21.92
netkit-tftpd	18	23	0.8824	0.36	0.8824	29.08
boxer	18	29	0.6552	2.19	0.7931	6.00
sharutils	19	36	0.6250	0.45	0.6827	13.22
mtunis	20	57	0.6140	0.65	0.6140	25.53
spdb	21	17	0.5000	0.21	0.5000	20.34
xtell	22	57	0.6309	0.68	0.6779	12.93
bunch	23	62	0.6400	3.06	0.7600	7.88
netkit-inetd	24	25	0.9588	0.25	0.9588	27.39
ispell	24	103	0.7010	2.21	0.7526	13.36
nanoxml	25	64	0.5968	1.52	0.6452	28.46
ciald	26	64	0.6250	1.85	0.6563	7.62
Modulizer	26	66	0.6032	3.14	0.6825	30.06
jodamoney	26	102	0.6000	1.12	0.6824	19.95
jxlsreader	27	73	0.5890	1.06	0.6986	15.82
bootp	27	75	0.6780	3.77	0.7186	14.44
sysklogd-1	28	74	0.6914	1.34	0.7543	30.20
telnetd	28	81	0.6908	1.27	0.7127	40.04
netkit-ftp	29	95	0.7058	1.73	0.7394	59.01
crond	29	112	0.6815	2.44	0.7124	16.55
rcs	29	163	0.7097	3.09	0.7613	33.53
seemp	30	61	0.4800	0.60	0.6600	17.27
dhcpd-2	31	122	0.6174	1.97	0.6946	18.34
cyrus-sasl	32	100	0.6104	2.08	0.6835	15.35
tcsh	32	105	0.8240	2.62	0.8522	28.55
micq	33	156	0.6799	3.55	0.6933	42.58
apache_zip	36	86	0.5135	3.68	0.7703	22.78
star	36	89	0.5393	1.10	0.6404	20.53
bison	37	179	0.6527	5.54	0.7784	11.45
stunnel	38	97	0.6532	2.48	0.6937	76.60
cia	38	185	0.5717	3.86	0.6550	22.63
minicom	40	257	0.6903	5.35	0.7424	63.58
mailx	41	331	0.6540	7.25	0.7473	47.91
dot	42	255	0.6935	6.83	0.7661	13.18
screen	42	292	0.7092	9.84	0.7465	49.33
slang	45	242	0.6659	4.76	0.7474	35.01
slrn	45	323	0.6957	7.97	0.7740	36.81
net-tools	48	183	0.6000	3.65	0.7582	58.62
graph10up49	49	1 650	0.7410	49.89	0.7467	18.97
wu-ftpd-1	50	230	0.7161	16.51	0.8086	29.04
joe	51	540	0.6182	9.33	0.7640	32.22
hw	53	51	0.4188	0.55	0.7145	19.23
imapd-1	53	298	0.6567	9.29	0.7424	25.32
wu-ftpd-3	54	278	0.7038	12.56	0.7418	35.61
udt-java	56	227	0.5762	4.16	0.7143	35.04
javaocr	58	155	0.5420	1.67	0.5573	38.80
dhcpd-1	59	571	0.6315	13.67	0.7370	41.84
pfcda_base	60	197	0.5680	5.42	0.7456	54.93
icecast	60	650	0.6887	25.06	0.7764	19.33
servletapi	61	131	0.4173	1.88	0.6378	39.56
php	62	191	0.5679	4.16	0.7475	57.25
bunch2	65	151	0.4531	2.53	0.7031	38.12
forms	68	270	0.5378	6.65	0.7200	27.05
jscatterplot	74	232	0.3988	2.91	0.6705	56.04
jxlscore	79	330	0.5728	6.80	0.6731	43.51
jfluid	81	315	0.6129	9.43	0.7384	60.46
elm-2	81	683	0.6801	20.17	0.7665	38.39
grappa	86	295	0.5198	5.26	0.5952	33.93
gnupg	88	601	0.5246	6.83	0.7550	51.10
elm-1	88	941	0.6466	34.97	0.7768	48.09
inn	90	624	0.6664	22.31	0.7474	63.58
bash	92	901	0.6633	57.03	0.7641	385.52
jpassword	96	361	0.5479	7.73	0.7126	157.63
bitchx	97	1 653	0.6792	61.49	0.7449	394.22
junit	99	276	0.4798	7.33	0.7309	296.47
xntp	111	729	0.5793	34.08	0.7876	197.56
acqCIGNA	114	179	0.4255	5.61	0.7500	365.20
bunch_2	116	364	0.4944	10.58	0.7346	640.21
xmldom	118	209	0.4322	4.12	0.5678	221.17
exim	118	1 255	0.6786	42.52	0.7645	294.76
cia++	124	369	0.5629	21.99	0.7635	440.95
tinytim	129	564	0.4671	12.79	0.7605	737.20
mod_ssl	135	1 095	0.5782	42.71	0.7487	510.18
jkaryoscope	136	460	0.3937	8.04	0.7615	457.68
ncurses	138	682	0.5397	29.31	0.7515	2 170.12
gae_plugin_core	139	375	0.4966	8.40	0.7905	320.07
lynx	148	1 745	0.6599	93.02	0.7598	863.54
lucent	153	103	0.1515	1.21	0.7071	2 365.04
javacc	153	722	0.4766	39.44	0.7421	547.10
JavaGeom	171	1 445	0.6006	62.34	0.7519	916.86
incl	174	360	0.4306	12.31	0.5083	640.17
jdendogram	177	583	0.4326	17.91	0.7696	1 482.01
xmlapi	182	413	0.4121	7.11	0.5000	868.31
jmetal	190	1 137	0.6280	43.31	0.7520	1 885.94
graph10up193	193	9 190	0.7381	681.49	0.7484	1 589.80
dom4j	195	930	0.5597	67.89	0.7452	1 409.25
nmh	198	3 262	0.6099	151.77	0.7662	4 452.61
pdf_renderer	199	629	0.3499	28.99	0.7117	4 259.32
Jung_graph_model	207	603	0.4262	29.38	0.7327	3 227.33
jung_visualization	208	919	0.4589	77.92	0.7533	2 163.71
jconsole	220	859	0.5378	22.49	0.7467	3 451.13
pfcda_swing	248	885	0.3720	41.29	0.7516	6 203.74
jml-1.0b4	267	1 745	0.5583	202.57	0.7684	4 977.25
jpassword2	269	1 348	0.6197	58.32	0.7589	9 372.19
notelab-full	293	1 349	0.4503	71.27	0.7583	9 827.90
Poormans_CMS	301	1 118	0.4961	61.43	0.7636	10 991.71
log4j	305	1 078	0.4182	52.35	0.7402	18 238.81
jtreeview	320	1 057	0.4350	58.28	0.7719	13 297.17
bunchall	324	1 339	0.5153	76.08	0.7661	5 652.04
JACE	338	1 524	0.4375	51.05	0.7621	12 743.43
javaws	377	1 403	0.5245	91.32	0.8026	23 593.95
swing	413	1 513	0.4677	101.84	0.7500	32 474.25
lwjgl-2.8.4	453	1 976	0.4374	256.97	0.7547	49 643.19
res_cobol	470	7 163	0.7017	3364.78	0.7539	29 593.82
ping_libc	481	2 854	0.4200	590.50	0.7960	12 184.89
y_base	556	2 510	0.4508	230.70	0.7755	75 474.69
krb5	558	3 793	0.4304	1179.04	0.7535	124 381.28
apache_ant_taskdef	626	2 421	0.5100	940.36	0.7872	222 383.90
itextpdf	650	3 898	0.5705	463.67	0.8334	126 596.26
apache_lucene_core	738	3 726	0.3633	277.59	0.7546	247 894.46
eclipse_jgit	909	5 452	0.5594	782.56	0.8131	546 848.53
linux	916	11 722	0.5912	3818.92	0.6988	292 383.92
apache_ant	1085	5 329	0.5101	1280.62	0.8686	891 821.18
ylayout	1161	5 770	0.3585	805.64	0.8557	652 576.52

(*continued on next page*)

(*continued*).

This method has been favorably compared with the best known method that tackles the FCB metric, an HGA presented by Mu et al. (2020). The comparison has been made on a set of 124 instances obtained from real-world software projects, published by Monçores et al. (2018). The results showed that the proposed algorithm outperformed the state-of-the-art algorithm in 112 instances. Moreover, according to the Wilcoxon's signed rank test, these results are statistically significant.

As it has been illustrated in this work, optimization algorithms can largely benefit from an efficient implementation of frequent functions, such as the computation of the objective function. In particular, we believe that trajectory-based metaheuristics present the advantage of partially reevaluating solutions after a move, avoiding unnecessary recalculations. Furthermore, we would like to point out the usefulness of identifying promising areas in the search space, consequently reducing the size of the neighborhoods and shortening the time necessary for their exploration. In this context, the combination of these strategies reduced the time consumption of the search process by four orders of

magnitude. This finding highlights the relevance of bringing domain-specific knowledge of the tackled problem into the design of the optimization algorithms.

In the context of software engineering, the proposed method can help software developers to enhance the quality of their code, either by incorporating this tool in Integrated Development Environments to receive suggestions in real time or by incorporating the method in software repositories to enforce a minimum modularization quality of the contributed software. Therefore, in order to be interesting for practical use, there is no doubt that the method must be able to provide high-quality solutions in a short computing time. In this regard, the proposed method is fast enough to be integrated in the software development lifecycle. Moreover, it can be configured to meet the particular needs of software developers by adjusting the stopping criterion.

Finally, following the directions given by de Oliveira Barros et al. (2015), we believe that Search-Based Software Engineering can be effectively used as a learning tool to investigate common concepts in software engineering. Accordingly, we consider that the proposed method, by providing better solutions than the best available algorithm in the state of the art in terms of the FCB objective function, can allow software practitioners and researchers to thoughtfully inspect the concept of modularity and the FCB metric.

In future work, it would be interesting to analyze and compare the performance of state-of-the-art algorithms for different objective functions in the SMCP literature. This analysis could help identify common parts and differences between the proposed quality metrics, in addition to possible unexplored transference of search strategies between different variants of the SMCP.

CRediT authorship contribution statement

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

Data availability

Data will be made available on request.

Acknowledgments

This research has been partially supported by grants: PID2021-125709OA-C22, PID2021-126605NB-I00, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way ofmaking Europe"; grant CIAICO/2021/224 funded by Generalitat Valenciana; grant M2988 funded by "Proyectos Impulso de la Universidad Rey Juan Carlos 2022"; and "Cátedra de Innovación y Digitalización Empresarial entre Universidad Rey Juan Carlos y Second Episode" (Ref. ID MCA06).

Appendix A. Detailed results for each instance.

In Table A.7, we present the results obtained in the comparison with the state of the art described in Section 6.3. We include, for each instance used, the number of vertices ($|V|$), the number of edges ($|E|$), and the quality (FCB) and computation time (CPUt (s)) of each algorithm. To facilitate comparisons, we have ordered the instances from smallest to largest, depending on the number of nodes.

Table B.8
Detailed results of the Wilcoxon's signed rank test for each group of instances and for all the instances in the dataset.

Size of instances	Ranks	Avg. rank	Z	*p*-value
Small (64)	GVNS < HGA: 52 GVNS > HGA: 0 GVNS = HGA: 12	26.50 0.00 –	−6.275	<0.001
Medium (29)	GVNS < HGA: 29 GVNS > HGA: 0 GVNS = HGA: 0	15.00 0.00 –	−4.703	<0.001
Large (18)	GVNS < HGA: 18 GVNS > HGA: 0 GVNS = HGA: 0	9.50 0.00 –	−3.724	<0.001
Very large (13)	GVNS < HGA: 13 GVNS > HGA: 0 GVNS = HGA: 0	7.00 0.00 –	−3.180	0.001
All (124)	GVNS < HGA: 112 GVNS > HGA: 0 GVNS = HGA: 12	56.50 0.00 –	−9.186	<0.001

Appendix B. Detailed results of the Wilcoxon's signed rank test.

In Table B.8, we present the detailed results of the Wilcoxon's signed rank test, for each group of instances and for all the instances in the dataset. Note that the objective of the studied problem is to minimize the value of the objective function. Therefore, GVNS < HGA indicates instances for which the solution found by GVNS was better than the solution found by the HGA approach.

References

Andrade, C.E., Byers, S.D., Gopalakrishnan, V., Halepovic, E., Poole, D.J., Tran, L.K., Volinsky, C.T., 2019. Scheduling software updates for connected cars with limited availability. Appl. Soft Comput. 82, 105575.

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.

Brimberg, J., Salhi, S., Todosijević, R., Urošević, D., 2023. Variable Neighborhood Search: The power of change and simplicity. Comput. Oper. Res. 155, 106221.

Catal, C., Bayrak, C., Nassif, A.B., Polat, K., Akbulut, A., 2016. Soft computing in software engineering. Appl. Soft Comput. 49, 953–955.

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

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., Pardo, E.G., Mladenovic, N., 2015. Multi-objective variable neighborhood search: an application to combinatorial optimization problems. J. Global Optim. 63 (3), 515–536.

Duarte, A., Pantrigo, J.J., Pardo, E.G., Sánchez-Oro, J., 2016. Parallel variable neighbourhood search strategies for the cutwidth minimization problem. IMA J. Manag. Math. 27 (1), 55–73.

Fakhoury, S., Roy, D., Hassan, A., Arnaoudova, V., 2019. Improving source code readability: Theory and practice. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, pp. 2–12.

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

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., 2021. A heuristic approach for the online order batching problem with multiple pickers. Comput. Ind. Eng. 160, 107517.
Granata, D., Sgalambro, A., 2023. A hybrid modified-NSGA-II VNS algorithm for the Multi-Objective Critical Disruption Path Problem. Comput. Oper. Res. 160, 106363.
Hansen, P., Mladenović, N., Moreno Perez, J.A., 2010. Variable neighbourhood search: methods and applications. Ann. Oper. Res. 175 (1), 367–407.
Hansen, P., Mladenović, N., Todosijević, R., Hanafi, S., 2017. Variable neighborhood search: basics and variants. EURO J. Comput. Optim. 5 (3), 423–454.
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.
ISO, 2017. ISO/IEC/IEEE 24765:2017 Systems and Software Engineering — Vocabulary. International Organization for Standardization.
Köhler, V., Fampa, M., Araújo, O., 2013. Mixed-integer linear programming formulations for the software clustering problem. Comput. Optim. Appl. 55 (1), 113–135.
Larman, C., 2012. Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development. Pearson Education India.
Manchala, P., Bisi, M., 2022. Diversity based imbalance learning approach for software fault prediction using machine learning models. Appl. Soft Comput. 124, 109069.
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., Cavero, S., Herrán, A., Duarte, A., Colmenar, J.M., 2022. A practical methodology for reproducible experimentation: an application to the Double-row Facility Layout Problem. Evol. Comput. 1–35.
Mitchell, B.S., Mancoridis, S., 2002a. A Heuristic Search Approach to Solving the Software Clustering Problem. Drexel University, Philadelphia, PA, USA.
Mitchell, B.S., Mancoridis, S., 2002b. 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., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Trans. Softw. Eng. 32 (3), 193–208.
Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., Ó Cinnéide, M., 2014. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. pp. 1263–1270.
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.
Mladenović, N., Todosijević, R., Urošević, D., Ratli, M., 2022. Solving the capacitated dispersion problem with variable neighborhood search approaches: From basic to skewed VNS. Comput. Oper. Res. 139, 105622.
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.
Mu, L., Sugumaran, V., Wang, F., 2020. A hybrid genetic algorithm for software architecture re-modularization. Inf. Syst. Front. 22 (5), 1133–1161.
Nadar, R.A., Jha, J., Thakkar, J.J., 2023. Adaptive variable neighbourhood search approach for time-dependent joint location and dispatching problem in a multi-tier ambulance system. Comput. Oper. Res. 159, 106355.
de Oliveira Barros, M., de Almeida Farzat, F., Travassos, G.H., 2015. Learning from optimization: A case study with Apache Ant. Inf. Softw. Technol. 57, 684–704.
de Oliveira Barros, M., de Almeida Farzat, F., 2013. What can a big program teach us about optimization? In: Search Based Software Engineering: 5th International Symposium, SSBSE 2013, St. Petersburg, Russia, August 24-26, 2013. Proceedings 5. Springer, pp. 275–281.
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].
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.
Saborido, R., Ferrer, J., Chicano, F., Alba, E., 2022. Automatizing software cognitive complexity reduction. IEEE Access 10, 11642–11656.
Sahin, O., Akay, B., 2016. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. Appl. Soft Comput. 49, 1202–1214.
Thirumoorthy, K., et al., 2022. A feature selection model for software defect prediction using binary Rao optimization algorithm. Appl. Soft Comput. 131, 109737.
Yuste, J., Duarte, A., Pardo, E.G., 2022. An efficient heuristic algorithm for software module clustering optimization. J. Syst. Softw. 190, 111349.