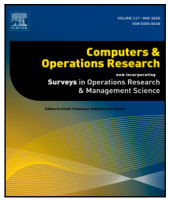
Computers & Operations Research 166 (2024) 106620

Contents lists available at ScienceDirect

Computers and Operations Research

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



Solving the Set Covering Problem with Conflicts on Sets: A new parallel GRASP

Francesco Carrabs [a], Raffaele Cerulli [a], Renata Mansini [b], Lorenzo Moreschini [b,*], Domenico Serra [a]

[a] University of Salerno, Department of Mathematics, Via Giovanni Paolo II 132, 84084, Salerno, Italy
[b] University of Brescia, Department of Information Engeneering, Via Branze 38, 25123, Brescia, Italy

ARTICLE INFO

Keywords:
Heuristics
Set Covering Problem
Conflicts
GRASP
Parallel algorithm

ABSTRACT

In this paper, we analyze a new variant of the well-known NP-hard Set Covering Problem, characterized by pairwise conflicts among subsets of items. Two subsets in conflict can belong to a solution provided that a positive penalty is paid. The problem looks for the optimal collection of subsets representing a cover and minimizing the sum of covering and penalty costs. We introduce two integer linear programming formulations and a quadratic one for the problem and provide a parallel GRASP (Greedy Randomized Adaptive Search Procedure) that, during parallel executions of the same basic procedure, shares information among threads. We tailor such a parallel processing to address the specific problem in an innovative way that allows us to prevent redundant computations in different threads, ultimately saving time. To evaluate the performance of our algorithm, we conduct extensive experiments on a large set of new instances obtained by adapting existing instances for the Set Covering Problem. Computational results show that the proposed approach is extremely effective and efficient providing better results than Gurobi (tackling three alternative mathematical formulations of the problem) in less than 1/6 of the computational time.

1. Introduction

In the classical Set Covering Problem (SCP) given a set of elements (e.g. regions into which a territory is partitioned) and a collection of subsets of such elements (e.g. facilities each one providing a service to a subset of regions), the problem looks for the optimal cover at minimum cost (the optimal set of plants to open so to cover the whole territory at the minimum set-up cost). The problem is known to be NP-hard.

A significant limitation of the SCP is its pursuit of an optimal cover without considering the interrelationships among the chosen subsets. This may lead to optimal solutions containing subsets that share a large number of elements which might not be practical in various scenarios. This paper introduces a generalization of the SCP dealing with pairwise conflicts among subsets. Differently from classic incompatibility constraints, two subsets in conflict can be jointly selected in a solution provided that a penalty is paid. The problem looks for a cover of the set of items that minimizes the sum of both covering and conflict costs. We name this problem the Set Covering Problem with Conflicts on Sets (SCP-CS). In our application the definition of the conflicts is related to the number of items jointly covered by a pair of subsets. More precisely, consider two subsets in conflict if they have more than a predefined number (called the *conflict threshold*) of elements in common.

Conflicts, also named incompatibility constraints, are commonly dealt with in different combinatorial problems such as knapsack problems (Coniglio et al., 2021; Hifi and Michrafy, 2007; Pferschy and Schauer, 2009), shortest path problems (Darmann et al., 2011), matching problems (Öncan and Kuban Altınel, 2018; Öncan et al., 2013), arc routing problems (Colombi et al., 2017), vehicle routing problems (Gendreau et al., 2016; Manerba and Mansini, 2016; Gobbi et al., 2023), bin packing problems (Ekici, 2021; Epstein et al., 2011; Sadykov and Vanderbeck, 2013), minimum spanning tree problems (Carrabs et al., 2019, 2021; Carrabs and Gaudioso, 2021) and maximum flow problems (Pferschy and Schauer, 2013; Şuvak et al., 2020). The term 'conflict' usually stands for an exclusionary constraint that forbids the simultaneous selection of a pair of items. Disjunctive constraints are frequently represented in terms of a conflict graph where vertices correspond to the items and edges encode the constraints. Whatever the considered combinatorial problem, a solution is feasible when it is conflict-free. In Jacob et al. (2019) the authors study a SCP variant

* Corresponding author.
E-mail addresses: fcarrabs@unisa.it (F. Carrabs), raffaele@unisa.it (R. Cerulli), renata.mansini@unibs.it (R. Mansini), lorenzo.moreschini@unibs.it (L. Moreschini), dserra@unisa.it (D. Serra).

https://doi.org/10.1016/j.cor.2024.106620
Received 15 December 2022; Received in revised form 31 January 2024; Accepted 11 March 2024
Available online 13 March 2024

where pairs of subsets are classified as incompatible and thus they cannot simultaneously belong to any feasible solution. They name the problem Conflict-free Set Covering Problem and analyze it in the context of approximation algorithms and parametrized complexity. In Saffari and Fathi (2022), the authors introduce several families of valid inequalities for the Conflict-free Set Covering Problem and work out some pre-processing procedures to further speed up the problem resolution. The work also suggests interesting application contexts such as wireless communication services management and multi-period staff-scheduling problems. Banik et al. (2020) face the SCP with conflicts in a geometric setting (Geometric Covering Problem with conflicts) where subsets are given by the intersection between geometric shapes and the universe of elements. As long as the conflict graph has bounded arboricity, they prove that if the Geometric Covering Problem is fixed-parameter tractable, then so is the conflict-free version, and if it admits a factor α-approximation then the same occurs for the conflict-free version.

Finally, for the sake of completeness, we also briefly present the literature that appeared in the last years concerning variants of the SCP including additional features such as profits and a budget constraint. In the partial SCP, it is either not necessary or not possible to cover all of the elements. Each element of the universe is associated with a profit whereas each subset has a cost. The problem aims to find a minimum cost collection of subsets such that the combined profit of the elements covered by the collection is at least equal to a predefined profit bound. In the prize-collecting version of partial SCP, if an element remains uncovered a penalty has to be paid. The problem looks for a collection of subsets such that the cost of selected subsets plus the penalties of uncovered elements is minimized (see Könemann et al. (2011) and references therein). In Bilal et al. (2014) the authors analyze a variant of the partial SCP where subsets are partitioned into groups. The selection of a subset in a solution implies the activation of its group and the payment of the cost associated with that group (negative profit). The objective is to maximize the total profit while it is not necessary to cover all the elements.

Motivation and contributions

SCP-CS finds application in different contexts including multi-period scheduling. In particular, we refer to a real-world application concerning the location of the radio base stations (RBS) in mobile communications. The larger the number of RBS opened, the higher the coverage of a geographic area, but also the higher the electromagnetic pollution that these antennas generate. The growth in the number of Internet users and applications that make massive use of network data exchange (online games, video streaming, telemedicine, lessons via videoconferencing, and the Internet of Things (IoT), just to name a few) has led to increased connectivity among devices of different types. Future communication networks will have to be implemented to create an infrastructure capable of simultaneously supporting various services and many users connected to those services (Attar et al., 2011; Guo et al., 2013). To support users and ensure high performance in densely populated urban areas, many small cells need to be located to provide good coverage. Nowadays people are more attentive and sensitive to the problem of possible dangers due to electromagnetic pollution generated by the use of these antennas (Saminathan et al., 2017; Sambo et al., 2015). Although no scientific evidence of a relationship between electromagnetic fields and human diseases has been provided, all precautionary measures against an uncontrolled rise in the electromagnetic field level must be taken. This choice is not only related to a general preventive principle but also to precise laws that require lowering electromagnetic emissions to a minimum value compatible with the quality of services (Cerri et al., 2002). In recent years, the interest in the reduction of electromagnetic pollution has also spread to the scientific community of optimization (Amaldi et al., 2002; Wang et al., 2013). In summary, the problem consists in ensuring the coverage of the selected areas with a good quality of service and limiting the overlapping of the signals among RBS to reduce the exposure to radiation of people living in the covered areas. To this end, we have set a penalty that is proportional to the number of signal intersections between two antennas so that a new RBS is installed only when strictly necessary. Notice that trying to completely remove the signal overlapping is not necessarily an optimal choice. It might lead to a reduction in the quality of the service and might generate coverage holes. For this reason, we introduce a conflict threshold k which defines the maximum number of overlapped signals that two antennas can have without paying a penalty.

The contribution of this paper is manifold. First of all, we provide the definition of a new and more realistic version of the SCP in which the conflict constraints are modeled as soft constraints and the concept of conflict depends on the structure of the available subsets. Moreover, we propose three mathematical formulations for the SCP-CS: a mixed integer linear program, a pure binary linear formulation, and a binary quadratic model comparing their performance when solved with a commercial solver. Introducing multiple models for the same problem proves advantageous, as the performance of state-of-the-art MIP solvers like Gurobi can vary significantly when applied to different models. Moreover, since the SCP-CS problem is a new problem, there are no benchmark instances and corresponding results that can be used to validate the effectiveness of new heuristic solution algorithms. Furthermore, to efficiently tackle the problem and obtain high-quality solutions in a limited amount of time, we introduce a novel parallel GRASP (Greedy Randomized Adaptive Search Procedure). To ensure an impartial and thorough investigation, we compare the outcomes of our parallel GRASP algorithm with those of Gurobi solving the three alternative formulations. The proposed parallel GRASP efficiently shares some common information among different processes that separately run the same version of a basic GRASP. Unlike parallel approaches commonly found in the literature, our method goes beyond a simple exchange of incumbent solutions among processes. We take it a step further by sharing intermediate computations, thereby eliminating redundant operations across threads. This approach results in an overall speedup of the procedure. The basic GRASP algorithm aims to enhance the incumbent solution in two distinct phases during each iteration. The initial phase involves generating a feasible solution using a combination of random and greedy rules, while the subsequent phase applies a local search procedure.

We obtained new benchmark instances by modifying existing ones originally designed for the standard SCP to accommodate conflicts and we run an extensive validation campaign by comparing the performance of the two linear models, the quadratic one (all implemented with the Gurobi solver) and the parallel GRASP. Computational results show that our parallel algorithm is extremely efficient and effective, always finding optimal or near-optimal solutions in a short amount of time.

The remainder of this paper is organized as follows. In Section 2, we introduce the formal problem definition and the three mathematical formulations. In Section 3 we present the parallel implementation of our heuristic approach. Computational results are reported in Section 4. Finally, conclusions are drawn in Section 5.

2. Problem definition and mathematical formulations

The SCP-CS is formalized as follows. Let $U = \{1, \dots, m\}$ be a finite set of elements (items) and $\mathcal{U} = \{U_j \subseteq U \mid j \in N\}$ be a collection of subsets of U with $N = \{1, \dots, n\}$. For each $j \in N$, we indicate as $c_j \in \mathbb{R}_0^+$ the cost of selecting the subset U_j and as $d_{jl} \in \mathbb{R}_0^+$ the cost that has to be paid if the subsets U_j and U_l ($j, l \in N$, $j \neq l$) are jointly selected. We indicate as B the set of all unordered pairs $\{j, l\}$ with $j, l \in N$ ($j \neq l$) and as $D \subseteq B$ the set of unordered pairs $\{j, l\}$ such that $d_{ij} > 0$ (the two subsets U_j and U_l are in conflict). In the following, we present two mathematical formulations for the SCP-CS: a linear formulation using only binary variables, and a binary quadratic formulation. We also introduce a mixed integer linear programming model for the special case related to the described application.

2.1. Binary linear programming formulation

To formulate the model, we introduce two sets of binary variables. The first set associates a binary variable x_j with each subset U_j ($j \in N$). Variable x_j takes value 1 if the subset is selected and zero otherwise. The second set consists of a binary variable y_{jl} ($j < l$) for each unordered pair $\{j,l\} \in D$ of subsets in conflict. The variable y_{jl} takes value 1 when both subsets are selected and zero otherwise. The mathematical formulation is as follows:

$$\textbf{(SCP-CS\_BLP)} \quad \min \sum_{j\in N} c_j x_j + \sum_{\{j,l\}\in D} d_{jl} y_{jl} \tag{1}$$

$$\sum_{j\in N} a_{ij} x_j \geq 1, \qquad \forall i \in U, \tag{2}$$

$$x_j + x_l \leq y_{jl} + 1, \qquad \forall \{j,l\} \in D, \tag{3}$$

$$x_j \in \{0,1\}, \qquad \forall j \in N, \tag{4}$$

$$y_{jl} \in \{0,1\}, \qquad \forall \{j,l\} \in D. \tag{5}$$

Constraints (2) are classical set covering constraints where the parameter a_{ij} is equal to 1 if $i \in U_j$ (element i is covered by subset U_j), and 0 otherwise. They ensure that each item belongs to at least one subset. Constraints (3) model the conflict: variable y_{jl} is forced to 1 when both subsets U_j and U_l ($\{j,l\} \in D$) are selected, i.e. when $x_j = x_l = 1$, and, thanks to the objective function minimization, it is guaranteed to be zero when at least one of x_j and x_l is zero. Since the size of D is bounded by $O(n^2)$, this model has $O(n^2)$ binary variables and $O(n^2 + m)$ constraints.

2.2. Binary quadratic programming formulation

To compare our heuristic algorithm with a wider range of traditional exact approaches, we also introduce a binary quadratic formulation where conflict constraints are removed and expressed as quadratic terms in the objective function. Nowadays, commercial solvers like Gurobi are highly efficient in solving binary programming problems with quadratic objectives and linear constraints. Research in this field is very active, as evidenced by recent contributions (see, for instance, Gusmeroli et al. (2022) and Rostami et al. (2023)).

Starting from the formulation of SCP-CS_BLP, the variables family $\{y_{jl}\}_{\{j,l\}\in D}$ is eliminated by incorporating constraints (3) into the objective function as quadratic terms.

$$\textbf{(SCP-CS\_BQP)} \quad \min \sum_{j\in N} c_j x_j + \sum_{\{j,l\}\in D} d_{jl} x_j x_l \tag{6}$$

$$\sum_{j\in N} a_{ij} x_j \geq 1, \qquad \forall i \in U, \tag{7}$$

$$x_j \in \{0,1\}, \qquad \forall j \in N. \tag{8}$$

The only set of constraints (7) is the classical family of set covering constraints. The conflict cost for a pair $\{j,l\} \in D$ of simultaneously selected subsets is tracked directly into the objective function where the product $x_j x_l$ is equal to 1 if and only if both subsets U_j and U_l are selected, and is equal to 0 otherwise. This model has exactly n binary variables, m constraints, and an objective function with $O(n^2)$ terms.

2.3. Special case

In the practical application described in Section 1, the conflict cost between two subsets depends linearly on the number of elements jointly covered by the two subsets and exceeding the threshold k, i.e.

$$d_{jl} = \gamma \max\left\{|U_j \cap U_l| - k,\ 0\right\}, \qquad \forall j,l \in N,\ j \neq l \tag{9}$$

where $\gamma \in \mathbb{R}^+$ is a unitary cost. In this special case, the previous general formulation (SCP-CS_BLP) can be rewritten using integer and binary variables as follows.

$$\textbf{(SCP-CS\_MILP)} \quad \min \sum_{j\in N} c_j x_j + \gamma \sum_{\{j,l\}\in B} w_{jl} \tag{10}$$

$$\sum_{j\in N} a_{ij} x_j \geq 1, \qquad \forall i \in U, \tag{11}$$

$$\left(\sum_{i=1}^{m} a_{ij} a_{il}\right)(x_j + x_l - 1) - k \leq w_{jl}, \ \forall \{j,l\} \in B, \tag{12}$$

$$x_j \in \{0,1\}, \qquad \forall j \in N, \tag{13}$$

$$w_{jl} \in \mathbb{N}, \qquad \forall \{j,l\} \in B. \tag{14}$$

The conflicts are represented by constraints (12). For every pair of subsets U_j and U_l where $j,l \in B$, an integer variable w_{jl} is defined. Thanks to objective function minimization, this variable takes the value $\max\left\{|U_j \cap U_l| - k, 0\right\}$ if both subsets are selected, and zero otherwise. Variables x_j ($j \in N$) have the meaning already explained. Since the number of integer variables grows as the square of the number of available subsets n, the model consists of $O(n^2)$ integer variables, $O(n)$ binary variables, and $O(n^2 + m)$ constraints.

It is worth noticing that integer variables are not necessary. Since variables w_{jl} ($\{j,l\} \in B$) only appear in constraints (12) where their value is bounded below by an integer quantity and in the objective function where the minimization of their costs forces them to take the smallest possible value, these variables can be defined as continuous. Problem complexity deflates to $O(n)$ binary variables and $O(n^2)$ continuous ones.

3. The parallel GRASP algorithm

GRASP (Greedy Randomized Adaptive Search Procedure) is an iterative heuristic approach. At each iteration, a possibly better solution is constructed through a two-phase procedure: in the first phase, an initial feasible solution is determined using a mix between random and greedy choices; then, in the second phase, a local search is applied to improve the current solution. At the end of each iteration, either the current (feasible) solution improves the incumbent one or it is discarded. Interested readers can refer to the initial work by Feo and Resende (1989) and the annotated bibliography by Festa and Resende (2002) for further details about the method and its variants.

The main contribution provided in this section is a parallel algorithm where common information (not limited to feasible solutions) is shared among different processes, each of them separately running the same single-process GRASP. The parallel execution allows the exploration of a much wider range of the search space within the same time constraint, leading to a significant improvement of the final solution.

Since the description of the final algorithm is quite complex and full of details, we decided to split it into three parts. Section 3.1 contains a high-level outline of the single-process GRASP for the SCP-CS problem, whereas in Section 3.2 we provide details on procedures, data structures, and time and space complexity. Finally, Section 3.3 explains how parallel processing has been exploited to speed up the algorithm. The complete source code written in Python is available at https://github.com/lmores/or-scp-cs-src.

Algorithm 1 Single-process GRASP: outline

```
function GraspStrategy(U, 𝒰, {c_j}_{j∈N}, {d_jl}_{{j,l}∈D}, f)   ▷ f is the objective function
    (W, w) ← (∅, 0)
    while ⋃_{j∈W} U_j ≠ U do                                    ▷ Phase 1: build initial solution
        Sort subsets in 𝒰_W in non-decreasing order w.r.t. θ_W
        Randomly select U_j among the first p subsets in 𝒰_W
        w ← w + |U_j \ ⋃_{j∈W} U_j| · θ_W(U_j)
        W ← W ∪ {U_j}
    end while
    while ∃ m ∈ M_W : f(m(W)) < w do                            ▷ Phase 2: improve current solution
        Pick m̄ ∈ argmin_{m∈M_W} {f(m(W))}
        (W, w) ← (m̄(W), f(m̄(W)))
    end while
    return (W, w)   ▷ Feasible solution and its cost built in a single GRASP iteration
end function
```

3.1. Single-process GRASP: the outline

This section is devoted to the description of the overall strategy we have developed. From now on, we will refer to a solution of the SCP-CS problem indicating a subset $W \subseteq N$ of the indexes of the available subsets and, with a slight abuse of terminology, we will say *"the subsets inside W"* when we actually mean *"the subsets whose index belongs to W"*.

The GRASP metaheuristic consists of the following two main phases. In the first phase, an initial solution $W \subseteq N$ is constructed. The process begins with an empty set and gradually adds subsets U_j ($j \in N$) one at a time until a feasible solution is obtained. When adding a new subset to W, the selection is made from the collection $\mathcal{U}_W$ of subsets that, if selected, cover at least one element not currently covered by the partial solution W, i.e.

$$\mathcal{U}_W := \left\{ U_j \in \mathcal{U} \;\middle|\; U_j \setminus \bigcup_{l \in W} U_l \neq \emptyset \right\}.$$

The choice of which subset to add is made by sorting all subsets in $\mathcal{U}_W$ by non-decreasing values of a greedy function $\theta_W : \mathcal{U}_W \to \mathbb{R}$ that measures the total increase of the objective function value associated with the selection of each subset in $\mathcal{U}_W$ as follows:

$$\theta_W : U_j \mapsto \frac{1}{|U_j \setminus \bigcup_{l \in W} U_l|} \left(c_j + \sum_{l \in W} d_{jl} \right).$$

Indeed, given a partial solution W, for each $U_j \in \mathcal{U}_W$, $\theta_W(U_j)$ provides the ratio between the current cost of U_j (comprising the costs of its conflicts with the subsets already included in the current partial solution W) and its cardinality excluding the items already covered by the current solution W. This heuristic is adaptive because the evaluation of each subset is updated at each iteration depending on the subsets already selected in the current partial solution. Once the subsets inside $\mathcal{U}_W$ have been ranked according to θ_W, the new subset to be added to W is chosen randomly among a predefined number $p \in \mathbb{N}^+$ of the most promising subsets, called the *Restricted Candidate List* (RCL). This probabilistic component allows us to obtain different solutions at the end of each iteration of the first phase of our GRASP algorithm.

The goal of the second phase is to iteratively enhance the current feasible solution W by exploring a neighborhood M_W of W in search of better solutions. To define M_W we introduce the concept of (r, s)-exchange move. Let $r, s \in \mathbb{N}$, we define a *(r, s)-exchange move* $\mathfrak{m}$ as a function that exchanges r subsets in the cover W with s subsets in $N \setminus W$ while preserving the feasibility. We choose

$$M_W := \{\mathfrak{m}(W) \mid \mathfrak{m} \text{ is an } (r, s)\text{-exchange move with } (r, s) \in (\{1\} \times \mathbb{N}) \cup (\mathbb{N} \times \{0, 1\})\}. \quad (15)$$

The neighborhood M_W contains all the moves that either remove exactly one subset from W ($r = 1$) and replace it with as many available subsets as needed to preserve the feasibility or insert at most one available subset ($s \in \{0, 1\}$) in W and remove from W potential subsets that are no longer needed to ensure the feasibility. Since the neighborhood M_W can be quite large, evaluating all the feasible solutions it contains may require an excessive amount of time. For this reason, in Section 3.2 we explain how we split M_W into three smaller neighborhoods that require an increasing computational burden to be searched. Our method explores the second neighborhood when the first one terminates in a local optimum and as soon as an improving solution is found in the second neighborhood, the search is restarted in the first one. The procedure moves to the last neighborhood only when no more improving solutions can be found in the preceding ones.

Algorithm 1 displays the pseudo-code of the procedure discussed so far. A naive implementation of this strategy would lead to a huge amount of time to execute even a single iteration of our algorithm. Since we aim to build and compare the outcomes of many iterations, in the next section we explain how we achieve an equivalent result in a much smaller amount of time.

Table 1
Summary of symbols adopted in parallel GRASP pseudo-code.

Symbol	Description
$p \in \mathbb{N}^+$	The size of the restricted candidate list
$W \subseteq N$	The indexes of the subsets in a (partial) cover of U
$S \subseteq N$	The indexes of the subsets that can be used during the first phase to build an initial cover (only meaningful for parallel two-stage GRASP, otherwise $S = N$)
$\mathcal{I}_i \subseteq N$	The indexes $j \in N$ such that $i \in U_j$ ($i \in U$)
$C_j \subseteq N$	The indexes $l \in N$ such that U_j ($j \in N$) is in conflict with U_l (i.e. $d_{jl} > 0$)
$R_j \in \mathbb{N}$	The current cost of U_j ($j \in N$) w.r.t. a (partial) cover W
$\mathcal{V}_i \subseteq N$	The indexes of the subsets of a (partial) cover W containing $i \in U$
$\mathcal{Q}_j \subseteq U$	The elements covered only by U_j in a (partial) cover W with $j \in W$
$\overline{U}_j \subseteq U$	The elements in U_j ($j \in N$) not yet covered by a (partial) cover W

3.2. Single-process GRASP

This section describes how the strategy outlined in Section 3.1 has been effectively implemented. In order to make the pseudo-code as readable as possible, we adopt the following conventions. Lowercase letters (a, b, c, ...) denote simple numerical values. Uppercase letters (A, B, C, ...) distinguish associative arrays containing simple numerical values, whereas uppercase letters with the calligraphic font ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ...) indicate associative arrays that contain at each position sets of numerical values. In both cases the value associated with a given key k is denoted placing k as subscript to the name of the data structure (A_k, B_k, C_k, ..., $\mathcal{A}_k$, $\mathcal{B}_k$, $\mathcal{C}_k$, ...).

Algorithm 2 Single-process GRASP

```
1: function GRASP(instance, i_max, t_max, S, sharedCache, incumbentCost)
2:     I ← BUILDINCIDENCESTRUCTURE(instance)
3:     C ← BUILDCONFLICTSTRUCTURE(instance)
4:     (W*, w*) ← (N, Σ_{j∈N} c_j + Σ_{{j,l}∈D} d_jl)
5:     (i, t) ← (0, 0)
6:     while (i < i_max) and (t < t_max) do
7:         (W, w) ← (∅, 0)
8:         (R, V, Q) ← INITSTATESTRUCTURES(instance)
9:         GRASPPHASE1(instance, C, I, W, w, R, V, Q, S, sharedCache)
10:        GRASPPHASE2(instance, C, I, W, w, R, V, Q)
11:        if w < w* then
12:            (W*, w*) ← (W, w)
13:            i ← 0
14:        else
15:            i ← i + 1
16:        end if
17:        t ← ELAPSEDTIME()
18:    end while
19:    return (W*, w*)
20: end function
```

Algorithm 2 provides the pseudo-code of the procedure run by each process and is thus at the core of our parallel algorithm. The method receives as input the data related to a problem *instance* (i.e. the set of elements U, the available subsets $\mathcal{U}$, the cost of each subset and the conflict costs) and the stopping rule parameters ($i_{\max} \in \mathbb{N}^+$ indicates the maximum number of iterations without improvement after which the algorithm terminates, $t_{\max} \in \mathbb{R}^+$ is the time limit assigned to the algorithm). Although the algorithm can run in a single process, we describe it by already including the data used by its parallel variant. In particular, *sharedCache* is the data structure that holds common information shared by all the processes and *incumbentCost* is the objective function value of the best solution found among all processes at any given time during the execution of the parallel variant of the algorithm. Finally, the set $S \subseteq N$ refers to the family of subsets available to build the initial cover during the first phase of GRASP. Since this data structure is meaningful only when a two-stage parallel variant of the algorithm is described in Section 3.3, we defer its detailed description and, in this section, we suppose that S equals N. In Table 1, we summarize the meaning of the symbols used in the pseudo-code.

The main loop of Algorithm 2 (Lines 6–18) ends as soon as one of the two stopping rules, controlled by $i_{\max}$ and $t_{\max}$, holds. Since we will often need to check which subsets contain a given element and which subsets are in conflict with a given one, at Lines 2–3 we initialize two data structures that allow to retrieve this information in constant time. BuildIncidenceStructure is a function that builds the array $\mathcal{I}$ that associates with each element $i \in U$ the subsets U_j containing i, i.e. $\mathcal{I}_i := \{j \in N \mid i \in U_j\}$. Note that infeasible instances can be easily detected by checking if any subset $\mathcal{I}_i$ ($i \in U$) is empty. Similarly, function BuildConflictStructure builds the associative array $\mathcal{C}$ that maps each index $j \in N$ to the set $\{l \in N \mid d_{jl} > 0\}$ ($j \in N$) of the subsets of U in conflict with U_j. At Line 4, the incumbent solution W^* and its objective function value w^* are initialized.

During each iteration of the main loop, we initialize an empty set W that will contain the cover we are going to build and we set its initial cost w equal to 0. Given a partial cover W, let us denote by $R_j = c_j + \sum_{l \in W} d_{jl}$, the *current cost* associated with each subset U_j ($j \in N$). If $j \notin W$, then R_j represents the increase of the objective function value if j were added to the current working cover W, whereas if $j \in W$, then R_j represents the decrease of the objective function value if j were removed from the current working cover W. We implement the current costs as an associative array named R. Since at the beginning of each iteration W is empty, the current cost of each subset U_j equals its subset cost (i.e. $R_j = c_j$ for each $j \in N$).

For each element $i \in U$, we denote by $\mathcal{V}_i := \{j \in W \mid i \in U_j\}$ the set of subsets U_j ($j \in W$) that cover i, and by $\mathcal{Q}_j = \{i \in U \mid \mathcal{V}_i = \{j\}\}$ the set of elements of U uniquely covered by subset U_j for a given $j \in W$. Data structures $\mathcal{V}$ and $\mathcal{Q}$ are implemented as associative arrays mapping each element $i \in U$ to $\mathcal{V}_i$ and each subset U_j ($j \in W$) to $\mathcal{Q}_j$. Since W is initially empty, each set inside the data structures $\mathcal{V}$ and $\mathcal{Q}$ is initially empty as well. Moreover, since they depend on W, whenever W changes they are updated accordingly. These data structures are mostly needed during the second phase of the algorithm when we must explore the neighborhood M_W. A priori, enforcing the feasibility of a solution would require computing the union of all subsets it contains and comparing it to U; the data structures $\mathcal{Q}$ and $\mathcal{V}$ make this operation less expensive as they incrementally track which subsets cover each item.

Lines 9–10 refer to the two phases of our algorithm. The function GraspPhase1 is in charge of building the working cover W to get an initial (possibly good) solution, whereas function GraspPhase2 tries to improve the initial solution exploring a sequence of neighborhoods and moving to a better cover (if any is found). When GraspPhase2 function completes, at Lines 11–16 the algorithm checks if the current working cover W improves the incumbent one W^* and if this is the case, the latter is replaced. Finally, the counter i, which tracks the number of iterations without improvement, and the variable t, which measures the time elapsed since the beginning of the execution, are updated. Line 19 returns the best cover found together with its cost when the stopping criterion terminates the main loop. In the following, we provide a detailed description of GraspPhase1 and GraspPhase2 functions.

Algorithm 3 Phase 1: initial solution

```
1: function GraspPhase1(instance, C, I, W, w, R, V, Q, S, sharedCache)
2:     U̅ ← InitializeCandidates(instance)
3:     Z ← BuildCandidateStructure(S, U̅, R)
4:     while U̅ ≠ ∅ do
5:         j ← PopRandomCandidate(Z)
6:         W ← W ∪ {j}
7:         U̅ ← Remove(U̅, j)
8:         if sharedCache_W ≠ null then
9:             (U̅, w, R, V, Q, Z) ← sharedCache_W
10:        else
11:            w ← w + R_j
12:            for l in C_j do
13:                R_l ← R_l + d_jl
14:                Z ← UpdateCandidateStructure(Z, l, R_l/|U̅_l|)
15:            end for
16:            T ← ∅
17:            for i in U_j do
18:                if |V_i| = 0 then
19:                    Q_j ← Q_j ∪ {i}
20:                else if |V_i| = 1 then
21:                    l ← Peek(V_i)
22:                    Q_l ← Q_l \ {i}
23:                    if |Q_l| = 0 then
24:                        T ← T ∪ {l}
25:                    end if
26:                end if
27:                V_i ← V_i ∪ {j}
28:            end for
29:            PruneWorkingCover(instance, C, I, W, w, R, V, Q, U̅, Z, T)
30:            for l in U̅ do
31:                if U̅_l \ U̅_j = ∅ then
32:                    U̅ ← U̅ \ l
33:                    Z ← RemoveCandidate(Z, l)
34:                else
35:                    U̅_l ← U̅_l \ U̅_j
36:                end if
37:            end for
38:            sharedCache_W ← (U̅, w, R, V, Q, Z)
39:        end if
40:    end while
41: end function

42: function PruneWorkingCover(instance, C, I, W, w, R, V, Q, U̅, Z, T)
43:    for j in T do
44:        if |Q_j| > 0 then
45:            W ← W \ {j}
46:            Q_j ← ∅
47:            w ← w − R_j
48:            for i in U_j do
49:                V_i ← V_i \ {j}
50:                if |V_i| = 1 then
51:                    l ← Peek(V_i)
52:                    Q_l ← Q_l ∪ {i}
53:                end if
54:            end for
55:            for l in C_j do
56:                R_l ← R_l − d_jl
57:                Z ← UpdateCandidateStructure(Z, l, R_l/|U̅_l|)
58:            end for
59:        end if
60:    end for
61: end function
```

Phase 1: building an initial solution

The pseudo-code of GraspPhase1 function is shown in Algorithm 3. The procedure is iterative. At each iteration, a new subset is randomly selected from the *Restricted Candidate List* (RCL) and used to populate the working cover W that is initially empty and its objective function value w is set to 0 (Line 1). At Line 2, the associative array $\overline{U}$ that contains the subsets to be selected to enter the current cover W is initialized. More precisely, each subset $\overline{U}_j$ is initialized with the subset U_j ($j \in N$). After each update of the current partial cover W, $\overline{U}_j$ will contain only the elements of U_j not yet covered by W.

The RCL contains the p most promising subsets obtained by ranking the candidates according to non-decreasing values of $\theta_W(U_j) = \frac{R_j}{|\overline{U}_j|}$. The ordering is adaptive since it depends on the value of R_j (that includes the cost of the conflicts between the subset U_j and the subsets that already belong to the current partial cover W) and the size of $\overline{U}_j$ (that contains only the elements of U_j not yet covered by the current partial cover W). To efficiently implement this strategy, we need a specialized data structure Z that during each iteration keeps track of the repeated changes of the values of the greedy function θ_W that determines the RCL (further details about the data structure Z can be found in Appendix). The while loop (Lines 4–40) is repeated until the selected collection of subsets provides a cover for U. At Lines 5–7, the algorithm randomly chooses a subset from the RCL (function PopRandomCandidate), adds it to the current cover W and removes it from $\overline{U}$. Next, at Line 8 the procedure checks if the current collection of subsets inside W has already been met in previous iterations. If this is the case, the procedure retrieves the previously computed information

from the *sharedCache* object and sets the updated values of all data structures (Line 9); otherwise, we update the data structure accordingly (Lines 11–38). This procedure is iterated until $\overline{\mathcal{U}}$ is empty, which implies that W is a cover of U. Each time the addition of a subset to the working cover W leads to a never-encountered partial cover, the algorithm works as follows.

1. The working cover cost w is increased by the current cost R_j of the selected subset j (Line 11).
2. The current cost of all subsets in conflict with j are increased and the structure Z is updated by the UpdateCandidateStructure function (Lines 12–15, where l is the index of a subset in conflict with j).
3. The subsets that have become redundant after the addition of subset U_j are removed from W (Lines 16–29). To achieve this goal we note that a subset U_l inside the current cover is redundant when all its elements are covered by at least another subset (i.e. when set $\mathcal{Q}_l$ is empty). To this aim, we iterate on elements i in U_j. If i was previously uncovered, U_j is recorded as the unique subset that covers i (Lines 18–19); otherwise, if i was previously covered by only one other subset l in the working cover, i.e. $\mathcal{V}_i$ contains exactly one index (the value returned by Peek($\mathcal{V}_i$)), we keep track that such subset is no more the only one that covers i (Lines 20–26). If this action makes U_l redundant, we add l to the list T of redundant subsets that may be eliminated (Lines 23–25). In any case, index j is added to the collection of the subsets in the working cover W that cover element i (Line 27). Finally, the PruneWorkingCover function actually removes from W some of the redundant subsets in T and updates all the other data structures accordingly (Line 29). Note that when T contains more than one redundant subset, they may have a non-empty pairwise intersection. In turn, this may imply that only some of them can be removed without uncovering some already covered elements. In this situation, making the best choice would require a cumbersome computational analysis of all the possibilities. However, since the goal of the current phase is to generate a feasible initial solution, we randomly remove as many redundant subsets as possible using a *first in first out* strategy and delegating any refinement to the second phase of the algorithm. The pseudo-code of the function PruneWorkingCover is provided at the end of Algorithm 3.
4. $\overline{\mathcal{U}}_j$ is subtracted from each subset $\overline{\mathcal{U}}_l$ inside $\overline{\mathcal{U}}$. If $\overline{\mathcal{U}}_j$ contains $\overline{\mathcal{U}}_l$ we remove l from $\overline{\mathcal{U}}$ and from Z, otherwise we replace $\overline{\mathcal{U}}_l$ with $\overline{\mathcal{U}}_l \setminus \overline{\mathcal{U}}_j$ (Lines 30–37).
5. Information about the new working cover W is saved into the *sharedCache* object for future reuse (Line 38).

At the end of this phase, the subsets contained in W provide a feasible solution with value w.

Phase 2: local search

The function GraspPhase2 employs a sequential local search strategy (pseudo-code is shown in Algorithm 4). Recall that, given a feasible solution W and $r, s \in \mathbb{N}$, we define a (r, s)-exchange move as a function that exchanges r subsets in the cover W with s subsets not belonging to W and preserves feasibility. We are interested in improving the current solution W looking for another cover inside the neighborhood M_W (defined in Eq. (15) by means of (r, s)-exchange moves with $r = 1$ or $s \in \{0, 1\}$). M_W can be written as the disjoint union of the following subsets:

$$M_W^0 = \{\mathfrak{m}(W) \mid \mathfrak{m}\ (r, 0)\text{-exchange move with } r \in \mathbb{N}\},$$
$$M_W^1 = \{\mathfrak{m}(W) \mid \mathfrak{m}\ (r, 1)\text{-exchange move with } r \in \mathbb{N}\},$$
$$M_W^2 = \{\mathfrak{m}(W) \mid \mathfrak{m}\ (1, s)\text{-exchange move with } s \in \mathbb{N},\ s \geq 2\},$$
$$M_W = M_W^0 \sqcup M_W^1 \sqcup M_W^2.$$

Algorithm 4 Phase 2: Local Search

```
1: function GraspPhase2(instance, C, I, W, w, R, V, Q)
2:     (W*, w*) ← (W, w)
3:     exit ← false
4:     repeat
5:         (W, w) ← ExploreM_W^0(instance, C, I, W, w, R, V, Q)
6:         if w < w* then
7:             (W*, w*) ← (W, w)
8:         else
9:             (W, w) ← ExploreM_W^1(instance, C, I, W, w, R, V, Q)
10:            if w < w* then
11:                (W*, w*) ← (W, w)
12:            else
13:                (W, w) ← ExploreM_W^2(instance, C, I, W, w, R, V, Q)
14:                if w < w* then
15:                    (W*, w*) ← (W, w)
16:                else
17:                    exit ← true
18:                end if
19:            end if
20:        end if
21:    until exit
22: end function
```

The three disjoint neighborhoods M_W^0, M_W^1, and M_W^2 require an increasing effort to be fully searched. We propose a local search strategy consisting of the following steps.

1. Explore M_W^0 by using a best improvement strategy (function ExploreM_W^0). This neighborhood contains all the feasible covers of U that can be obtained by removing one or more subsets from the current solution W. A necessary and sufficient condition to be able to remove a single subset U_j with index j from W is that the set of elements covered only by such a subset must be empty, i.e. $\mathcal{Q}_j = \emptyset$. Let $F \subseteq W$ be the set of removable subsets belonging to the current cover that satisfy this condition. Note that it may not be possible to remove all the subsets in F at once. We apply a procedure to determine the subfamily of F that once removed maintains the feasibility and produces the highest decrease of the objective function value.
2. Explore M_W^1 by using the best improvement strategy (function ExploreM_W^1). If at least one cover that improves the current one is found, update the best incumbent solution and restart the search from Step 1.
3. Explore M_W^2 (function ExploreM_W^2). This neighborhood is defined by the set of $(1, s)$-exchange moves with $s \geq 2$, namely, the moves that replace any subset in the current cover W with two or more external subsets while preserving feasibility. Fully exploring all these possibilities may require a large amount of time since restoring feasibility after the removal of a subset from a feasible solution amounts to solving a smaller SCP-CS instance. As a consequence, we explore this neighborhood only partially, starting from the most promising move and setting a time threshold $\tau \in \mathbb{R}^+$ to the evaluation of the moves that replace a given subset inside W (we forcefully stop the analysis when the time threshold is reached and move to the next one). More precisely, for each subset U_j ($j \in W$), we retrieve the set $\mathcal{Q}_j$ of elements covered only by U_j and, for each such an element, we sort its incident subsets by increasing current cost. We iterate over all possible Cartesian products of the subsets that cover the elements previously covered only by U_j and, if their addition to W improves the best value found so far, we save this move as the incumbent one. Once all subsets in W have been evaluated for removal, if we have found at least one cover improving the current one, we update W and restart from Step 1, otherwise, the local search ends.

The procedure iterates until the current cover W cannot be further improved and GraspPhase2 ends. To efficiently implement the above

procedure, we use information stored inside the associative arrays $\mathcal{V}$ and $\mathcal{Q}$. When removing a subset U_j ($j \in N$) from the current solution, a necessary and sufficient condition to restore the feasibility of the current cover W is to add a (possibly empty) collection of available subsets that covers the elements inside $\mathcal{Q}_j$. Data structures required during these steps are mainly needed to track the local changes of their global counterparts.

As far as time complexity is concerned, we observe that the size of the neighborhood in Step 1 depends on the cardinality of W which is at most $O(n)$ (although one may expect that on average $|W| \ll n$). To detect the best possible $(r,0)$-exchange move, during each iteration, the procedure goes through all the elements of a given subset and through all the subsets that have non-zero conflict costs with a such subset. Determining whether a subset can be added to the collection of removable subsets requires a time complexity of $O(s+t)$, where s is the maximum cardinality of a subset (bounded by m) and t is the maximum number of conflicts that involve a fixed subset in U (bounded by $n-1$). M_W^1 contains all covers that can be obtained by adding one external subset to W and removing the most convenient ones. Once the external subset is inserted, we find the best collection of subsets that can be removed preserving feasibility amounts to evaluate all possible $(r,0)$-exchange moves and we apply the same strategy adopted for M_W^0. Finally, as already described, when exploring neighborhood M_W^2 we set a time limit for the evaluation of each collection of moves that removes a given subset from W preserving feasibility.

3.3. Parallel GRASP

Parallel processing can greatly improve the outcome of the single-process algorithm. Due to GRASP's randomly-restart nature, the higher the number of restarts and thus of feasible solutions the algorithm is able to evaluate, the higher the chance it has to improve the value of the final solution. A naive application of parallel computing would be to spawn as many processes as the number of available CPUs on the machine where the program is executed, each one independently carrying out the single-process algorithm previously described. After the termination of all executions, the main procedure would gather the result of each process and select the best solution among all the available ones. However, since each process executes the same optimization steps, it is beneficial to save and share among all processes the information computed during each execution (especially for computationally intensive tasks). This prevents each process from performing exactly the same operations that others have done before. This strategy can be usefully adopted during the first phase of the single-process GRASP. When building the initial feasible solution the procedure starts from an empty collection and iteratively adds one subset randomly chosen from the RCL. Each time this action is carried out we are forced to update many data structures (namely $\overline{U}$, W, w, R, $\mathcal{V}$, $\mathcal{Q}$ and Z) to reflect the effect of the inclusion of a new subset inside the cover that is currently being built. However, the outcome after each iteration of the first phase of the algorithm solely depends on which subsets belong to the partial cover that we have constructed so far (regardless of their insertion order). This is a type of information that can easily be shared among all processes. Therefore, in the parallel implementation of our algorithm, once the new state of each data structure has been computed for a given (partial) cover W, it is stored in a hash table ($sharedCache$) available to all the processes. In this way, when any iteration of the first phase of the single-process GRASP needs to evaluate an already analyzed partial cover W, no additional computational effort is required. Note that this may come out to be particularly convenient when we need to retrieve the state of the data structures associated with partial covers containing a few (typically of high quality) subsets: they have a high probability of appearing more than once across different processes.

Algorithm 5 Parallel GRASP: change to Algorithm 2

```
11: if w < incumbentCost then
12:     (W*, w*, incumbentCost) ← (W, w, w)
13:     i ← 0
14: else
15:     i ← i + 1
16: end if
```

In order to reduce the overall running time, during the execution of the algorithm we also share among all workers the value of the best cover found at any given time. To this aim, we must replace Lines 11–16 in Algorithm 2 with the ones contained in Algorithm 5.

The benefits provided by multiprocessing are not limited to information sharing. The possibility to run the same instance multiple times in parallel has also been exploited to develop a *two-stage algorithm*. Performing the same non-deterministic task in multiple processes leads to (most likely) different final solutions. In our case, a solution consists of a cover of U that, although not optimal, is expected to contain at least some of the subsets that belong to an optimal solution. The union of all such solutions is likely to contain most (if not all) of the subsets needed to build an optimal cover. Therefore, at the end of the first stage, we collect the best solutions found by each process and during the first phase of the second stage, we restrict the candidate list to the collection of subsets selected in this way. This is the goal of the data structure S that appears in the signature of the function GRASP in Algorithm 2.

Algorithm 6 Parallel GRASP

```
1: function GraspCoordinator(instance, i_max, t_max, cpuCount)
2:     (workers, inbound, outbound, sharedCache, incumbentCost) ← InitMultiprocessing(cpuCount)
3:     for w in workers do
4:         Run(w, GraspWorker, (instance, i_max, t_max, inbound, outbound, sharedCache, incumbentCost))
5:     end for

6:     S ← ∅
7:     for i in 1, ..., |workers| do
8:         W ← Poll(outbound)                          ▷ Wait for Stage 1 completion
9:         S ← S ∪ W
10:    end for
11:    Clear(sharedCache)
12:    for i in 1, ..., |workers| do
13:        Put(inbound, S)
14:    end for

15:    W* ← N
16:    w* ← Σ_{j∈N} c_j + Σ_{{j,l}∈D} d_jl
17:    for i in 1, ..., |workers| do
18:        (W, w) ← Poll(outbound)                     ▷ Wait for Stage 2 completion
19:        if w < w* then
20:            W* ← W
21:            w* ← w
22:        end if
23:    end for

24:    return (W*, w*)
25: end function

26: function GraspWorker(instance, i_max, t_max, inbound, outbound, sharedCache, incumbentCost)
27:    (W, w) ← Grasp(instance, i_max, t_max, N, sharedCache, incumbentCost)
28:    Put(outbound, W)
29:    S ← Poll(inbound)
30:    (W, w) ← Grasp(instance, i_max, t_max, S, sharedCache, incumbentCost)
31:    Put(outbound, (W, w))
32: end function
```

An algorithm that leverages parallel processing needs a coordinator (at an outer layer) to handle and coordinate the work of each sub-process. The sole purpose of the function GRASPCOORDINATOR in Algorithm 6 (from now on the *main process*) is to orchestrate the execution of a pool of sub-processes (from now on the *workers*) in charge of running function GRASP (Algorithm 2). Function GRASPCOORDINATOR takes as input the instance data and the stopping rule parameters already discussed plus the integer value $cpuCount$, which represents the number of CPUs the algorithm can use during the execution. At Line 2, function INITMULTIPROCESSING is called to instantiate the $workers$, two queue-like data structures ($inbound$ and $outbound$) used to collect the individual

worker results at the end of the first and second stage of the algorithm, the associative array *sharedCache*, used to share among all workers the outcome of the most expensive computations, and the best incumbent objective function value *incumbentCost* found among all workers at a given instant of time. At Lines 3–5, Algorithm 6 calls the Run function that instructs each worker (the first argument) to execute the single-process GRASP (the second argument) on the necessary input data (the third argument). At Lines 6–10, Algorithm 6 gathers the results of the first stage of each worker (the best covers found) and computes their union, storing the result inside the array S. Note that the main process (the GraspCoordinator function) will block waiting for all workers to put their results inside the *outbound* queue (Line 8). Once all workers have completed the first stage, the method exits the loop and goes to Lines 12–14 where the union of all the best covers previously found is added to the *inbound* queue (shared with each worker). Adding such information inside the *inbound* data structure provides the signal to each worker to start running the second stage of parallel GRASP, using as available subsets for the first phase only those contained inside the set S. Eventually, at Lines 15–23, the algorithm waits for each worker to complete the execution of the second stage by querying the *outbound* queue. Once all results have been collected, the best one is returned as the final solution (Line 24).

3.4. Data structures, time and space complexity

In our single-process GRASP, we make use of various data structures. Most of the time, we need to store and retrieve information associated with an element or a subset (both represented by integers). Thus, the most frequently used data structures are associative arrays, more specifically arrays and hash maps. These well-established data structures offer efficient $O(1)$ complexity for retrieving, inserting, or updating entries.

However, in the first phase of our algorithm, there is a step that requires tracking the topmost p subsets ranked according to the value of $\frac{R_j}{|U_j|}$ (that depends on the current partial cover) as discussed in Section 3.2. This situation cannot be effectively handled using arrays or hash maps, since they do not provide efficient ways to sort their entries and to keep them sorted when the value of any of them changes. To address this issue we use a pair $Z = (H_{\max}, H_{\min})$ of key–value binary heaps. The former is a maximum key–value binary heap with a maximum size equal to p and at any given time during the execution of the algorithm, it holds the subsets that have the lowest current costs. The latter, instead, is a minimum key–value binary heap that contains all the remaining subsets (those with higher current costs). Whenever the current cost of any subset changes, we locate the heap that contains that subset with $O(1)$ time complexity and update its value with $O(\log_2(p))$ time complexity if it belongs to H_1 and $O(\log_2(n-p))$ complexity if it belongs to H_2. Finally, we compare the top elements of $H_{\max}$ and $H_{\min}$ (which can be both retrieved in $O(1)$) and, if the former is greater than the latter, we swap them. At the end of this procedure, $H_{\max}$ will contain the p subsets with the lowest current costs. One can easily check that in the worst-case scenario the time complexity of one iteration of the first phase of our algorithm is $O(n + m\log_2(n))$. Data structures involved in the second phase are just plain arrays and hash maps. As a final remark, we note that adding parallel processing to our algorithm has a space and time complexity that does not depend on the size of the input instance.

4. Experimental analysis

In this section, first, we describe the data set and how the algorithm parameters have been set, then we provide the results obtained solving all mathematical formulations using Gurobi and compare their performance with that of our parallel GRASP (using as conflict threshold both $k = 1$ and $k = 2$). The source code and all data sets are available at the following public GitHub repository: https://github.com/lmores/or-scp-cs-src.

The same machine has been used to execute all algorithms: an Intel Xeon Gold 6140M CPU 2.30 GHz with 8 physical cores and 16 logical cores paired with 64 GB of RAM running Microsoft Windows 10 Pro in a virtual machine.

4.1. Parameters setting

The parallel GRASP algorithm has been implemented using CPython 3.10. Since many of the implementation details of this parallel variant depend on the language chosen to code the algorithm, we specify that our architecture makes use of the tools provided by the `multiprocessing` module of the Python Standard Library and the interested reader can inspect the complete source code freely available at https://github.com/lmores/or-scp-cs-src. No other external software packages have been used. The main process in charge of handling the execution of all the subprocesses has been initialized with a number of workers equal to 15 (the number of available machine cores minus one used to handle the inter-process communication workload and to provide to the operating system a free thread to execute all the other programs).

Let u be the number of elements contained in each available subset counted with multiplicity, i.e. $u = \sum_{j \in N} |U_j|$. The average number of elements in a subset is, therefore, $\frac{u}{n}$ and we may expect that, on average, a feasible solution should be made up of a number of subsets equal to the total number of elements m divided by $\frac{u}{n}$. Therefore, a candidate value for the size p of the RCL could be $\frac{mn}{u}$. While this choice of p has proven effective for the majority of instances, there are specific cases where this value is either too high or too low. In the first case, creating an RCL with this size results in an excessively high variance in the quality of the subsets it contains. An example is offered by the instances family *scpcyc*, where each available subset contains either 3 or 4 items, while the set to be covered varies in size from 240 to 28 160. For these instances, we chose to limit the size of the RCL by $\sqrt{n}$ which depends solely on the number of available choices. In the second case (i.e. when the available subsets are nearly as large as the whole set to be covered), the size of the RCL happens to be too small to provide enough variance to escape from a local minimum. An example is provided by the instances family *scpe*, where many feasible solutions are made of only 3 subsets. In this case, we decided to bound by below the size of the RCL using a fixed value $p_{\min}$. The general rule adopted to set the value of p is therefore

$$p = \max\left\{p_{\min},\ \min\left\{\sqrt{n},\ \frac{mn}{\sum_{j \in N}|U_j|}\right\}\right\},$$

where we set $p_{\min} = 12$. Finally, parameters related to the stopping rules have been set to $i_{\max} = 50$ and $t_{\max} = 600$ s (equally divided between the first and second stage), whereas the threshold time for the evaluation of each subfamily of moves in M_W^2 has been set to $\tau = 0.01$ s.

To solve the mathematical formulations, we used Gurobi 9.5 without changing its default parameter settings except for the `TimeLimit` parameter which has been set to 1 h (in particular, the default value of the `MIPGap` parameter is 10^{-4}). As for parallel GRASP, all logical cores have been made available to Gurobi which decided independently how many of them to use during the optimization process.

4.2. Instances generation

Since the SCP-CS problem has not been previously studied in the literature, no benchmark instances exist. We have decided to generate them by adapting the instances for the SCP made available by Beasley in the OR-library (Beasley, 1990a,b). In Beasley's instances, the size and the number of overlaps between subsets are not high enough to determine a consistent number of conflicts even when $k = 1$ (the value $\max\left\{\max_{j,l \in N, j \neq l}\left\{|U_j \cap U_l| - k\right\}, 0\right\}$ is often very low or equal to zero).

Table 2
Benchmark instances: Set A.

Set-A				
Instance	$\|U\|$	$\|N\|$	#conf ($k=1$)	#conf ($k=2$)
scp41–3	200	334	8351	1908
scp42–3	200	334	8306	1848
scp43–3	200	334	8258	1791
scp44–3	200	334	8596	1977
scp45–3	200	334	8136	1789
scp46–3	200	334	8826	2055
scp47–3	200	334	7843	1711
scp48–3	200	334	8526	1860
scp49–3	200	334	8195	1807
scp410–3	200	334	7944	1793
scp51–3	200	667	33 962	8238
scp52–3	200	667	34 092	8047
scp53–3	200	667	34 153	8104
scp54–3	200	667	33 893	7913
scp55–3	200	667	32 490	7376
scp56–3	200	667	34 551	8355
scp57–3	200	667	34 832	8393
scp58–3	200	667	33 285	7784
scp59–3	200	667	33 094	7597
scp510–3	200	667	34 476	8321
scp61–3	200	334	49 416	40 825
scp62–3	200	334	49 916	41 855
scp63–3	200	334	49 827	41 583
scp64–3	200	334	49 404	40 723
scp65–3	200	334	49 830	41 733
scpa1–3	300	1000	139 239	49 629
scpa2–3	300	1000	139 174	49 455
scpa3–3	300	1000	139 351	49 665
scpa4–3	300	1000	139 104	50 086
scpa5–3	300	1000	139 255	49 635
scpb1–3	300	1000	485 873	457 826
scpb2–3	300	1000	486 022	458 275
scpb3–3	300	1000	485 654	457 218
scpb4–3	300	1000	485 276	456 577
scpb5–3	300	1000	485 595	457 456
scpc1–3	400	1334	347 270	152 214
scpc2–3	400	1334	345 511	150 739
scpc3–3	400	1334	345 621	151 469
scpc4–3	400	1334	346 553	150 846
scpc5–3	400	1334	346 137	151 744
scpd1–3	400	1334	883 245	867 920
scpd2–3	400	1334	883 605	869 019
scpd3–3	400	1334	883 254	868 280
scpd4–3	400	1334	883 136	868 196
scpd5–3	400	1334	883 471	868 775
scpe1–3	50	167	13 810	13 707
scpe2–3	50	167	13 837	13 756
scpe3–3	50	167	13 755	13 666
scpe4–3	50	167	13 818	13 707
scpe5–3	50	167	13 839	13 790

Table 3
Benchmark instances: Set B.

Set-B				
Instance	$\|U\|$	$\|N\|$	#conf ($k=1$)	#conf ($k=2$)
scpclr10–3	511	70	2415	2415
scpclr11–3	1023	110	5995	5995
scpclr12–3	2047	165	13 530	13 530
scpclr13–3	4095	239	28 441	28 441
scpcyc06–3	240	64	281	153
scpcyc07–3	672	150	806	443
scpcyc08–3	1792	342	2173	1230
scpcyc09–3	4608	768	5660	3267
scpcyc10–3	11 520	1707	14 370	8401
scpcyc11–3	28 160	3755	35 487	21 072

For this reason, for each instance in the OR-library we generated a new instance obtained by merging three consecutive subsets (as appearing inside Beasley's original instance) into a single one with a cost equal to the sum of the costs of the merged subsets. Thanks to this operation, the number of conflicts in each instance increased to a meaningful value for our purposes. As a consequence of this adaptation, the new instances contain a number of subsets that is about one-third of the number of subsets in the original ones. Moreover, to compute the conflict costs d_{jl} for each $\{j,l\} \in B$ we set

$$\gamma = \max\left\{\left\lceil \max_{j\in N} \frac{c_j}{|U_j|}\right\rceil, 1\right\}$$

in (9). Our goal is to generate complex instances where the conflict costs must have the same order of magnitude as the costs c_j ($j \in N$) associated with the selection of a single subset U_j. The rationale is that too-high conflict costs would simply imply the exclusion of the related pair of conflicting subsets from an optimal solution, while too-low conflict costs would essentially be negligible, actually reducing our SCP-CS to a traditional SCP. To distinguish the generated instances from the original ones we append the suffix '*-3*' to the name of the former ones (e.g., the name of the original instance '*scp42*' from the OR-library becomes '*scp42-3*').

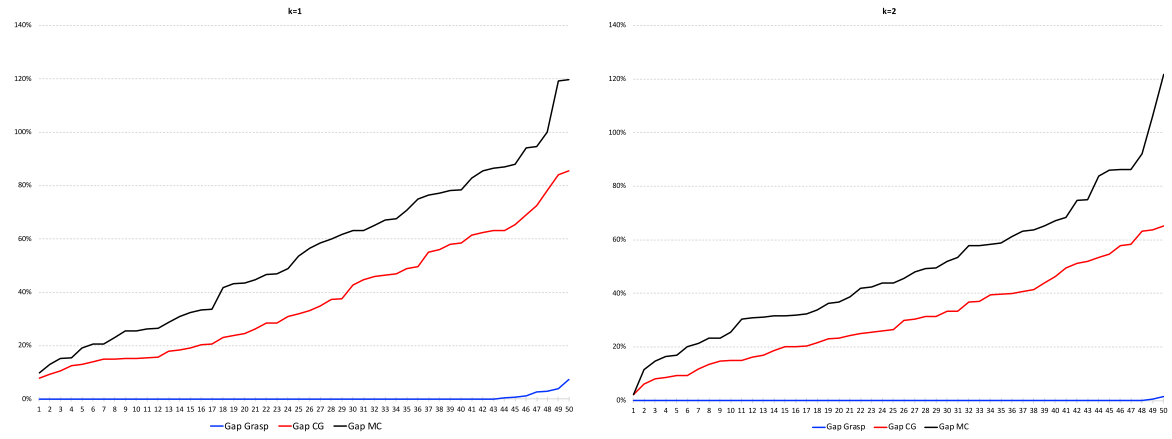
Table 4
Benchmark instances: Set C.

Set-C				
Instance	$\|U\|$	$\|N\|$	#conf ($k=1$)	#conf ($k=2$)
scpnre1–3	500	1667	1 388 611	1388611
scpnre2–3	500	1667	1 388 611	1388611
scpnre3–3	500	1667	1 388 611	1388611
scpnre4–3	500	1667	1 388 611	1388611
scpnre5–3	500	1667	1 388 611	1388611
scpnrf1–3	500	1667	1 388 611	1388611
scpnrf2–3	500	1667	1 388 611	1388611
scpnrf3–3	500	1667	1 388 611	1388611
scpnrf4–3	500	1667	1 388 611	1388611
scpnrf5–3	500	1667	1 388 611	1388611
scpnrg1–3	1000	3334	4 628 203	3574676
scpnrg2–3	1000	3334	4 626 736	3574156
scpnrg3–3	1000	3334	4 623 974	3574347
scpnrg4–3	1000	3334	4 630 196	3579650
scpnrg5–3	1000	3334	4 625 519	3575142
scpnrh1–3	1000	3334	5 555 947	5555602
scpnrh2–3	1000	3334	5 555 936	5555589
scpnrh3–3	1000	3334	5 555 946	5555566
scpnrh4–3	1000	3334	5 555 937	5555574
scpnrh5–3	1000	3334	5 555 925	5555576

Computational tests have been carried out on 80 instances partitioned into three sets. *Set-A*, consisting of 50 instances, has been obtained by adapting the instances proposed in Beasley (1987). *Set-B* contains 10 instances and has been obtained by modifying the instances proposed in Grossman and Wool (1997). Finally, *Set-C* has 20 instances obtained by modifying instances proposed in Beasley (1992). The main features of these three sets are summarized in Tables 2–4 where the first column shows the name of the instance, the second and third columns show the number of elements ($|U|$) and of subsets ($|N|$), whereas the last two columns show the number of pairs of subsets in conflict when $k = 1$ and $k = 2$, respectively.

4.3. Computational results: parallel GRASP vs adapted algorithms from the literature

Since SCP-CS modifies the classical SCP by introducing conflict costs, we adapted the classical SCP's algorithm by Chvátal (see Chvatal (1979)) to handle conflicts. We refer to this new algorithm as MC (Modified Chvátal). Moreover, we also developed an enhanced version of MC obtained by incorporating the Carousel Greedy framework (see Cerrone et al. (2017)). In the following, we provide a concise description of them. Both algorithms are much less effective than our parallel GRASP.



(a) (b)

Fig. 1. Parallel GRASP versus MC and CG: percentage gaps from the best-known values on *Set-A* instances.

- **Modified Chvátal (MC).** Original Chvátal's algorithm starts by placing the available subsets $U_j \subseteq U$ into a priority queue Q. Let $\rho(U_j)$ be the number of elements in U_j that are not yet covered by the currently selected subsets (at the beginning $\rho(U_j) = |U_j|$). With each subset U_j in Q we associate a priority equal to $\frac{c_j}{\rho(U_j)}$ (the lower the better). The algorithm builds a cover by iteratively selecting the subset U_j with the minimum ratio from the queue Q and adding it to the solution set C. The priority of the subsets in Q are updated at each iteration accordingly. This process continues until the chosen subsets provide a cover for U. We propose an adapted version of the algorithm, in which the weight for each subset instead of a pure c_j is modified taking into account the additional cost caused by the conflicts generated if U_j is added to C. It is worth noting that without this modification, the algorithm produces a percentage gap from the best-known values equal to 170%, at least.
- **Carousel Greedy (CG) algorithm for SCP-CS.** The CG approach is a versatile and generalized greedy framework designed to significantly enhance the performance of traditional greedy strategies. CG starts from an initial feasible solution generated by the domain-specific algorithm to be improved. This solution is refined by replacing prior selections, made by the greedy domain-specific algorithm, with new choices, thereby yielding a new feasible solution. CG requires as input an initial feasible solution along with two parameter values. The first parameter $\alpha \in \mathbb{N}$ is an integer that specifies the number of iterations, while the second parameter $\beta \in [0, 1]$ specifies the percentage of the initial solution that can be pruned and rebuilt. The CG implementation for the SCP-CS is based on the MC algorithm. After tuning, we found that the best results were obtained by setting $\alpha = 9$ and $\beta = 0.3$. The core steps of the algorithm are as follows.

  1. *Initialization:* invoke the MC algorithm to obtain an initial feasible solution built adding one available subset at a time; let it be $C = \left\{U_{i_1}, \ldots, U_{i_t}\right\}$.
  2. *Carousel start:* remove from C a percentage β of subsets starting from the last ones and obtaining a new (infeasible) solution $\overline{C} = \left\{U_{i_1}, \ldots, U_{i_s}\right\}$ $(s < t)$.
  3. *Iterative update:* iterate $\alpha \cdot |C|$ times, and at each iteration:
     (a) remove the oldest element from $\overline{C}$ (e.g., in the first iteration, remove U_{i_1}),
     (b) add a new subset to $\overline{C}$ using the greedy policy of MC.
  4. *Solution completion:* finally, invoke the MC algorithm on $\overline{C}$ to complete the solution (restoring the feasibility).

We verified the performance of MC and CG by comparing them with our parallel GRASP on the instances in Set A (where more optimal solutions are known) for both $k = 1$ and $k = 2$. The result of this comparison is shown in Fig. 1. To generate these charts we sort in non-decreasing order the percentage gaps from the best-known solution values (plotted on the y-axis), while the x-axis is related to the instances. The percentage gap is computed with the formula: $100 \times \frac{Obj - Best}{Best}$ where Obj is the solution value provided by the algorithm considered and $Best$ is the value of the best-known solution given by the minimum among the results of the three mathematical models, the parallel GRASP, MC, and CG.

As shown in Fig. 1, the parallel GRASP is by far the most effective among the three algorithms, with a percentage gap that is often equal to zero. In particular, for $k = 1$ it finds the best-known solution in 33 out of 40 instances, and in 38 instances for $k = 2$. On the other hand, MC never finds the best-known solution and its percentage gap ranges from 10% up to 120%, for $k = 1$, and from 2% up to 122%, for $k = 2$. The application of the Carousel Greedy framework on MC allows for improved effectiveness achieving a gap that ranges from 8% to 86%, for $k = 1$, and from 2% up to 65%, for $k = 2$. However, these results are not sufficient to make it competitive with the parallel GRASP algorithm which is, by far, more effective than the other developed methods.

4.4. Computational results: parallel GRASP vs mathematical models

In Table 5, we compare the performance of the two linear mathematical formulations (SCP-CS_BLP and SCP-CS_MILP) solved with Gurobi. Given a time limit of 1 h, Gurobi's execution terminates with one out of three possible statuses: `OPTIMAL` status indicates that Gurobi found an optimal solution within the time limit; `TIME LIMIT` status signifies that the imposed time limit was reached without finding an optimal solution; `MEMORY LIMIT` status indicates that the procedure ran out of memory before reaching the time limit. In the latter two cases, the incumbent feasible solution (if any) is returned. When the algorithm terminates with an `OPTIMAL` status, we denote the corresponding solution value with an asterisk (*). In the event of a `MEMORY LIMIT` status, we report the computational time in italics. Notice that Table 5 is partitioned into two parts, one devoted to the comparison of the two models for $k = 1$ and the other for $k = 2$.

As Gurobi allows us to tune the percentage of the total runtime spent in heuristic procedures (rather than advancing in the exploration of the branch and bound tree), we conducted additional computational experiments by running Gurobi with different settings for the `MIPFocus` parameter. When this parameter is set to 0 (the default) Gurobi strikes a balance between finding new feasible solutions and proving that the incumbent solution is optimal, whereas when set to 1 Gurobi

Table 5
Computational results for SCP-CS_BLP and SCP-CS_MILP.

GUROBI																
Instance	k = 1								k = 2							
	SCP-CS_BLP				SCP-CS_MILP				SCP-CS_BLP				SCP-CS_MILP			
	Mode	Obj	ObjC (#conf)	Time	Mode	Obj	ObjC (#conf)	Time	Mode	Obj	ObjC (#conf)	Time	Mode	Obj	ObjC (#conf)	Time
Set-A																
scp41–3	D	2037*	350 (7)	253.69	D	2037*	350 (7)	200.33	D	1108*	50 (1)	1.50	D	1108*	50 (1)	1.47
scp42–3	D	1977*	700 (14)	81.81	D	1977*	700 (14)	84.44	D	1209*	100 (2)	22.47	D	1209*	100 (2)	1.42
scp43–3	D	2583*	500 (5)	1314.28	D	2583*	500 (5)	1428.64	D	1113*	0 (0)	0.86	D	1113*	0 (0)	0.94
scp44–3	D	2543*	686 (7)	1033.85	D	2543*	686 (7)	824.51	D	1192*	98 (1)	2.64	D	1192*	98 (1)	2.62
scp45–3	D	2247*	413 (7)	400.11	D	2247*	413 (7)	363.55	D	1279*	59 (1)	2.00	D	1279*	59 (1)	1.78
scp46–3	D	2602*	666 (9)	1550.40	D	2602*	666 (9)	1535.76	D	1302*	74 (1)	3.23	D	1302*	74 (1)	3.22
scp47–3	D	2128*	590 (8)	778.80	D	2128*	590 (8)	731.50	D	1116*	118 (2)	2.36	D	1116*	118 (2)	2.30
scp48–3	D	2647*	567 (7)	2220.35	D	2647*	567 (7)	1756.22	D	1149*	0 (0)	1.69	D	1149*	0 (0)	1.53
scp49–3	D	2604*	564 (6)	705.27	D	2604*	564 (6)	968.81	D	1398*	0 (0)	4.83	D	1398*	0 (0)	4.83
scp410–3	D	2501*	290 (5)	903.23	D	2501*	290 (5)	911.93	D	1404*	0 (0)	2.81	D	1404*	0 (0)	2.31
scp51–3	D	1532	304 (4)	3601.05	D	1615	76 (1)	3600.84	D	618*	0 (0)	4.94	D	618*	0 (0)	5.39
scp52–3	H	1446	73 (1)	3601.46	D	1446	73 (1)	3606.41	D	602*	0 (0)	1.19	D	602*	0 (0)	1.20
scp53–3	D	1506	300 (3)	3600.72	D	1423	0 (0)	3602.14	D	627*	0 (0)	5.14	D	627*	0 (0)	5.36
scp54–3	H	1391	294 (3)	3600.70	D	1360	196 (2)	3601.83	D	546*	0 (0)	2.14	D	546*	0 (0)	2.09
scp55–3	D	1375	201 (3)	3601.73	D	1375	201 (3)	3602.23	D	528*	0 (0)	0.78	D	528*	0 (0)	0.81
scp56–3	D	1410	91 (1)	3601.16	D	1410	91 (1)	3601.00	D	511*	0 (0)	0.56	D	511*	0 (0)	0.47
scp57–3	D	1551	0 (0)	3601.06	D	1551	0 (0)	3601.41	D	764*	0 (0)	7.58	D	764*	0 (0)	7.73
scp58–3	D	1574	77 (1)	3600.43	H	1612	0 (0)	3602.34	D	650*	0 (0)	3.45	D	650*	0 (0)	3.34
scp59–3	D	1518	200 (2)	3601.19	D	1535	100 (1)	3601.18	D	660*	0 (0)	3.23	D	660*	0 (0)	3.34
scp510–3	H	1361	0 (0)	3601.23	D	1547	178 (2)	3601.01	D	642*	0 (0)	1.86	D	642*	0 (0)	1.87
scp61–3	H	4833	3572 (63)	3602.29	D	4846	3078 (74)	3601.16	H	2169	1216 (47)	3600.78	H	2286	1501 (40)	3600.80
scp62–3	H	4808	3344 (83)	3602.87	H	4310	3439 (86)	3603.81	H	2518	1330 (43)	3601.23	H	2740	2223 (56)	3601.02
scp63–3	H	5709	3975 (71)	3601.94	D	6227	4150 (69)	3601.12	H	2925	1625 (43)	3600.57	D	3103	2000 (41)	3600.66
scp64–3	H	4061	3260 (79)	3602.47	D	4905	3880 (84)	3601.08	H	2128	1180 (40)	3600.83	D	2905	1980 (52)	3600.78
scp65–3	D	4508	3145 (73)	3601.05	D	4636	3230 (78)	3601.02	H	2319	1224 (47)	3600.68	D	2590	1496 (46)	3600.67
scpa1–3	H	3717	1248 (24)	3602.71	D	4784	2080 (34)	3601.44	D	963	52 (1)	3602.77	D	963	52 (1)	3603.54
scpa2–3	H	4037	1357 (22)	3601.89	D	4395	1947 (26)	3601.40	D	1048	59 (1)	3601.59	H	1048	59 (1)	3601.04
scpa3–3	H	3459	1150 (22)	3601.29	D	4189	1500 (26)	3600.96	D	910	0 (0)	3604.48	D	910	0 (0)	3604.49
scpa4–3	H	4025	1496 (22)	3607.18	D	4242	952 (14)	3601.74	D	946	0 (0)	3602.87	D	946	0 (0)	3603.09
scpa5–3	H	4216	670 (10)	3603.66	D	5000	2278 (31)	3600.82	D	894*	0 (0)	2010.28	D	894*	0 (0)	697.61
scpb1–3	H	9378	9126 (148)	3600.11	D	10 857	8853 (115)	3600.13	H	6045	5564 (172)	3600.43	D	6697	5772 (140)	3600.14
scpb2–3	D	10 581	9408 (235)	3600.12	D	10 581	9408 (235)	3600.11	D	6109	5432 (178)	3600.17	D	6109	5432 (178)	3600.15
scpb3–3	D	9324	8772 (160)	3600.13	D	9324	8772 (160)	3600.13	D	5864	5112 (182)	3600.82	H	4802	4440 (154)	3600.50
scpb4–3	D	9576	8820 (257)	3600.11	H	9232	9024 (152)	3600.10	D	5145	4596 (151)	3600.42	D	5145	4596 (151)	3600.46
scpb5–3	D	10 922	9758 (217)	3600.12	D	10 922	9758 (217)	3600.13	D	6178	5614 (174)	3600.11	D	6178	5614 (174)	3600.14
scpc1–3	H	14 664	11600 (86)	3602.17	D	17 478	13100 (96)	3600.94	H	1613	0 (0)	3606.77	H	1585	0 (0)	3602.22
scpc2–3	D	7473	4785 (111)	3601.70	D	7473	4785 (111)	3602.05	H	1391	99 (3)	3601.92	H	1596	33 (1)	3601.72
scpc3–3	H	10 559	7350 (106)	3603.30	D	10 699	7100 (107)	3602.40	H	1583	150 (3)	3602.28	H	1553	150 (3)	3603.92
scpc4–3	D	7997	5206 (107)	3601.81	D	7997	5206 (107)	3601.83	H	1596	152 (4)	3601.78	D	1636	190 (5)	3629.82
scpc5–3	D	10 557	7150 (105)	3601.31	D	10 557	7150 (105)	3601.92	H	1514	0 (0)	3601.94	H	1370	50 (1)	3601.77
scpd1–3	H	12 058	11106 (306)	3600.16	D	12 601	10845 (136)	3600.22	D	8882	8181 (241)	3600.21	D	8882	8181 (241)	3600.23
scpd2–3	D	11 682	10746 (304)	3600.25	D	11 682	10746 (304)	3600.20	D	9258	8361 (279)	3600.21	D	9258	8361 (279)	3600.23
scpd3–3	D	11 049	9432 (152)	3600.22	D	11 049	9432 (152)	3600.21	H	8530	7785 (233)	3600.14	H	9177	8784 (183)	3600.12
scpd4–3	D	13 535	12320 (120)	3600.39	D	13 535	12320 (120)	3600.22	D	13 738	12586 (231)	3600.18	D	13 738	12586 (231)	3600.21
scpd5–3	D	13 176	12366 (329)	3600.22	D	13 176	12366 (329)	3600.24	D	8403	7659 (237)	3600.18	D	8403	7659 (237)	3600.20
scpe1–3	D	26*	17 (3)	72.41	D	26*	17 (3)	71.23	D	23*	14 (3)	69.64	D	23*	14 (3)	50.59
scpe2–3	D	28*	19 (3)	94.54	D	28*	19 (3)	93.20	D	25*	16 (3)	63.08	D	25*	16 (3)	71.44
scpe3–3	D	24*	18 (1)	33.94	D	24*	18 (1)	23.81	D	23*	11 (5)	42.39	D	23*	11 (5)	54.05
scpe4–3	D	25*	16 (3)	88.23	D	25*	16 (3)	88.88	D	22*	13 (3)	30.17	D	22*	13 (3)	45.31
scpe5–3	D	28*	19 (3)	85.77	D	28*	19 (3)	85.64	D	25*	16 (3)	73.84	D	25*	16 (3)	73.78
Set-B																
scpclr10–3	D	1926*	1893 (55)	412.10	D	1926*	1893 (55)	271.49	D	1871*	1838 (55)	331.91	D	1871*	1838 (55)	258.75
scpclr11–3	D	4215	4182 (55)	3600.43	H	4330	4294 (66)	3600.62	D	4034	3998 (66)	3600.26	H	3446	3413 (55)	3600.80
scpclr12–3	H	9236	9200 (66)	3600.78	H	9986	9950 (66)	3601.45	D	7728	7692 (66)	3600.23	H	8612	8579 (55)	3600.83
scpclr13–3	H	28 175	28130 (105)	3600.47	H	17 926	17890 (66)	3603.27	H	27 325	27283 (91)	3600.25	H	21 299	21260 (78)	3601.56
scpcyc06–3	D	126*	42 (27)	1.23	D	126*	42 (27)	2.41	D	99*	15 (9)	0.83	D	99*	15 (9)	0.72
scpcyc07–3	D	335*	136 (74)	2273.54	D	335*	136 (74)	1942.80	D	250*	49 (42)	1688.08	D	250*	49 (38)	1246.73
scpcyc08–3	D	911	434 (183)	3600.78	D	919	442 (176)	3600.22	H	660	177 (163)	3600.41	H	659	188 (160)	3600.31
scpcyc09–3	D	2332	1234 (485)	3600.83	H	2594	1481 (773)	3601.59	D	1678	565 (453)	3600.27	D	1721	608 (476)	3600.16
scpcyc10–3	D	6755	4214 (1966)	3600.70	H	7603	4985 (2428)	3600.99	D	4462	1966 (1136)	3600.58	D	4441	1909 (1376)	3600.38
scpcyc11–3	H	19 380	13500 (6627)	3600.54	H	19 744	13792 (6640)	3600.80	H	12 694	6826 (4059)	3600.42	H	13 718	7542 (5123)	3600.33
Set-C																
scpnre1–3	D	7424	7149 (78)	3600.35	H	4233	3675 (36)	3603.10	D	7047	5724 (55)	3600.29	H	4773	4344 (45)	3602.90
scpnre2–3	D	9054	8826 (78)	3618.44	H	4905	4536 (45)	*2079.32*	H	7553	6192 (55)	3600.95	H	5210	4854 (45)	*1546.80*
scpnre3–3	D	8211	7440 (105)	3600.28	H	4680	4158 (36)	3603.01	D	6581	6213 (78)	3601.35	H	5625	5142 (45)	*2072.90*
scpnre4–3	D	7230	7158 (78)	3601.18	H	4221	3699 (36)	3603.44	D	6327	6273 (66)	3600.33	H	4856	4449 (45)	3603.74
scpnre5–3	D	7488	6288 (55)	3600.30	H	5139	4695 (45)	*2787.62*	D	7323	6123 (55)	3600.32	H	4530	4152 (45)	3603.19
scpnrf1–3	D	2145	1608 (15)	3601.12	H	1835	1556 (15)	3603.78	D	2798	2777 (21)	3601.01	H	1807	1718 (15)	*1360.77*
scpnrf2–3	D	2324	1748 (15)	3601.03	H	1888	1522 (15)	3602.90	H	2214	1919 (21)	3601.50	H	1968	1722 (15)	3603.83
scpnrf3–3	D	2241	1836 (15)	3601.09	H	1813	1777 (15)	3602.89	H	2082	1878 (15)	3601.17	H	1892	1655 (15)	*2459.28*
scpnrf4–3	D	2484	2370 (21)	3601.12	H	1696	1666 (15)	3603.45	H	2206	1657 (15)	3601.24	H	1829	1730 (15)	*1921.07*
scpnrf5–3	H	1829	1793 (15)	3601.19	H	1445	1250 (10)	3602.91	H	1814	1778 (15)	3601.36	H	1708	1630 (15)	*1765.84*
scpnrg1–3	D	33 276	32210 (1125)	3600.98	D	33 276	32210 (1125)	3601.00	H	15 738	15140 (736)	3600.60	D	21 735	21240 (794)	3601.28
scpnrg2–3	D	34 828	34560 (1039)	3601.02	H	34 092	33830 (980)	3600.60	D	15 524	14020 (770)	3600.75	D	15 524	14020 (770)	3600.75
scpnrg3–3	D	28 168	26620 (1104)	3601.02	D	28 168	26620 (1104)	3601.07	D	17 639	15906 (771)	3600.69	D	17 639	15906 (771)	3600.82
scpnrg4–3	D	30 665	28740 (1324)	3600.96	H	30 425	29950 (1041)	3600.60	D	17 604	15940 (819)	3600.72	D	17 604	15940 (819)	3601.21
scpnrg5–3	D	26 931	24960 (1234)	3600.95	D	26 931	24960 (1234)	3601.07	D	15 825	15270 (755)	3602.36	D	15 825	15270 (755)	3601.19
scpnrh1–3	D	19 313	15684 (253)	3601.29	H	18 114	17799 (300)	3601.52	D	20 124	20019 (351)	3602.75	H	17 637	17307 (325)	3600.78
scpnrh2–3	H	18 537	14442 (253)	3600.79	D	18 537	14442 (253)	3601.48	D	19 527	19383 (351)	3601.27	H	16 860	16581 (276)	3600.84
scpnrh3–3	D	17 982	17889 (300)	3601.29	D	17 982	17889 (300)	3601.40	H	17 082	16989 (300)	3600.97	H	17 082	16989 (300)	3600.77
scpnrh4–3	D	16 749	14646 (253)	3601.27	D	16 749	14646 (253)	3601.37	D	16 815	16716 (300)	3601.39	D	16 815	16716 (300)	3601.69
scpnrh5–3	D	17 361	17268 (300)	3601.26	H	15 654	15564 (276)	3600.78	D	16 461	16368 (300)	3601.31	H	14 967	14877 (276)	3600.83
Avg		**7652.49**		**2944.94**		**7385.54**		**2904.29**		**5003.62**		**2350.54**		**4813.15**		**2197.07**
#Opt				**18**				**18**				**29**				**29**

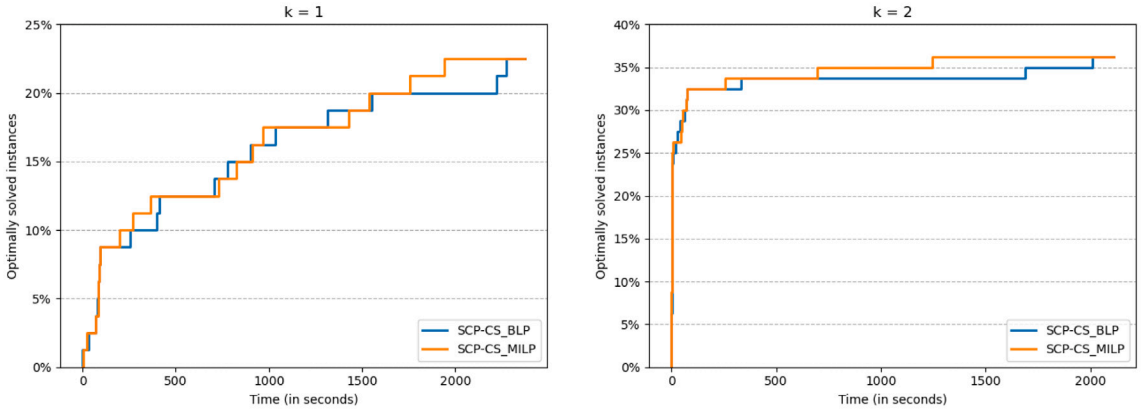


Fig. 2. Performance comparison between SCP-CS_BLP and SCP-CS_MILP for $k = 1$ (a) and $k = 2$ (b).

focuses on finding feasible solutions quickly (by default Gurobi spends only 5% of runtime on solving heuristics). For each linear model and instance, we run Gurobi with both settings: column *Mode* shows for each case the best-performing configuration between the Default (D) or the Heuristic (H) one. Column *Obj* provides the solution value, while column *ObjC* shows the sum of all conflict costs, and, in brackets, it reports the corresponding number of conflicts *#conf*. Column *Time* contains the computational time in seconds. Finally, the last two lines of the table report the average objective and time values (*Avg*), and the number of optimal solutions (*#Opt*) found by the two models out of all instances, respectively. Analyzing these last two lines, we observe that both models find the same number of optimal solutions for $k = 1$ and $k = 2$. In particular, for $k = 1$, they find 18 optimal solutions out of 80 instances while, for $k = 2$, they provide 29 optimal solutions. Moreover, for $k = 2$, the computational time decreases by $\sim 20\%$ for SCP-CS_BLP and by $\sim 25\%$ for SCP-CS_MILP. Thus, instances with $k = 2$ seem to be easier to solve than instances with $k = 1$: this is most likely due to the lower number of conflicts (cf. Tables 2–4). From the results in the *Avg* row, we deduce that SCP-CS_MILP is, on average, faster than SCP-CS_BLP and that it provides better solutions when both models do not find an optimal one. However, the `MEMORY LIMIT` status occurs only for SCP-CS_MILP (in particular when $k = 2$) and this indicates that the growth of its search tree is faster than the one of SCP-CS_BLP. Upon a closer examination of the result table, it becomes evident that the instances in *Set-C* are more challenging for both models. Notably, neither model succeeds in finding optimal solutions for this set. This difficulty can likely be attributed to the high number of conflicts within these instances. It is important to highlight that when we set the `MIPFocus` parameter equal to 1, Gurobi manages to improve the quality of the best solution in many instances. Nevertheless, this improvement comes at the expense of significantly higher memory consumption, occasionally reaching the machine's memory limit of 64 GB before the one-hour time limit. Despite these advancements, our parallel GRASP algorithm consistently outperforms Gurobi in the majority of the cases.

In Fig. 2, we show the results reported in Table 5 in a way that better highlights the performance of the linear models. In these charts, the horizontal axis reports the computational time in seconds and the vertical one shows the percentage of optimally solved instances within that time. More precisely, a (x, y) point on this plot shows the percentage of optimally solved instances (y value) in less than or equal to x seconds. This implies that the faster the growth of a curve, the better the performance. The blue curve is associated with SCP-CS_BLP model, whereas the orange one corresponds to SCP-CS_MILP. Fig. 2(a) certifies the similar performance of the two models that we have already observed. However, we note that, overall, around 23% of the instances are solved to optimality by SCP-CS_BLP within 2280 s, while SCP-CS_MILP reaches the same result in 1950 s. This means that SCP-CS_MILP is $\sim 15\%$ faster than SCP-CS_BLP in reaching the optimal solution. This trend is even more clear in Fig. 2(b). Here, the number of optimal solutions found is equal to 36% and most of them are found by both models in less than 75 s. However, the performance gap between the two models for $k = 2$ is more evident as SCP-CS_MILP reaches the highest percentage in about 1250 s while SCP-CS_BLP requires around 2000 s to achieve the same result. To summarize, the effectiveness of the two models is the same, but SCP-CS_MILP is, on average, more efficient than SCP-CS_BLP.

In Table 6, we compare the solutions found by our parallel GRASP with the ones found by the three mathematical models. Also in this case the table is partitioned into two parts providing results for $k = 1$ and $k = 2$, respectively. In particular, under the headings *ILP* and *BQP*, we report the best (possibly optimal) solution found by Gurobi within the time limit of 1 h, when solving the two linear models and the quadratic one, respectively. Optimal solution values are emphasized by a trailing asterisk. Columns *ObjBest* and *ObjMed* report the best and the median objective values of parallel GRASP out of 10 runs for each instance. The next four columns provide statistics about the best solution out of the ten runs. Column *ObjC* shows the sum of all conflict costs and the corresponding number of conflicts *#conf* while column *Ttb* reports the time to best that represents the time required to find the best solution provided as output. The column *Time* shows the total computational time in seconds. This value is underlined when parallel GRASP reaches the time limit of 300 s during Stage 1. Finally, the *Gap* column provides the percentage gap between *ObjB* and the best value found when solving all three mathematical models. This percentage value is computed as $100 \times \frac{ObjBest - \min\{ILP,\ BQP\}}{\min\{ILP,\ BQP\}}$. In each row, the best value is marked in bold. At the bottom of the table, the *Avg* row reports the average values of the computational time and of the percentage gap, while *#Best* shows how many times parallel GRASP finds a solution that is better than or equal to the one found by the three models solved with Gurobi. Finally, the *AvgRSD* row contains the average value of the relative standard deviation of the objective value computed using the results of the 10 runs. The results in row *#Best* show that, for $k = 1$, parallel GRASP finds a solution better than or equal to the best one found by Gurobi in 72 out of 80 instances. In the remaining 8 instances the percentage gap is lower than 7.3%. For 51 instances parallel GRASP provides a solution strictly better than the best one with an average improvement of $\sim 15\%$. Similar results are observed for $k = 2$ where for 74 instances the solution provided by parallel GRASP is the best one (the value is strictly better in 41 instances with an average improvement of 14.56%); the percentage gap in the remaining 4 instances is lower than 4.4%. It is worthy of note that, for $k = 2$, parallel GRASP always finds the optimal solution in the 30 instances

Table 6
Gurobi vs. parallel GRASP: computational results.

Parallel GRASP																
Instance	k = 1								k = 2							
	ILP	QBP	ObjBest	ObjMed	ObjC (#conf)	Ttb	Time	Gap	ILP	QBP	ObjBest	ObjMed	ObjC (#conf)	Ttb	Time	Gap
Set-A																
scp41–3	**2037***	**2037***	2097	2123.00	650 (13)	68.83	110.11	2.95%	**1108***	**1108***	**1108**	1108.00	50 (1)	3.25	69.81	0.00%
scp42–3	**1977***	**1977***	**1977**	2000.00	700 (14)	66.77	105.20	0.00%	**1209***	**1209***	**1209**	1209.00	50 (1)	4.52	89.06	0.00%
scp43–3	**2583***	**2583***	**2583**	2613.00	500 (5)	87.81	134.53	0.00%	**1113***	**1113***	**1113**	1113.00	0 (0)	1.14	74.20	0.00%
scp44–3	**2543***	**2543***	**2543**	2639.00	686 (7)	166.52	209.64	0.00%	**1192***	**1192***	**1192**	1192.00	98 (1)	3.03	89.83	0.00%
scp45–3	**2247***	**2247***	**2247**	2247.00	413 (7)	74.94	111.58	0.00%	**1279***	**1279***	**1279**	1279.00	59 (1)	51.00	81.56	0.00%
scp46–3	**2602***	**2602***	**2602**	2688.50	666 (9)	91.50	133.59	0.00%	**1302***	**1302***	**1302**	1302.00	74 (1)	60.56	93.78	0.00%
scp47–3	**2128***	**2128***	**2128**	2128.00	590 (8)	2.00	96.92	0.00%	**1116***	**1116***	**1116**	1116.00	118 (2)	5.08	78.34	0.00%
scp48–3	**2647***	**2647***	2663	2777.00	972 (10)	84.67	129.61	0.60%	**1149***	**1149***	**1149**	1149.00	0 (0)	8.52	97.16	0.00%
scp49–3	**2604***	**2604**	**2604**	2684.00	564 (6)	16.14	109.52	0.00%	**1398***	**1398***	**1398**	1398.00	0 (0)	53.25	95.34	0.00%
scp410–3	**2501***	**2501***	**2501**	2527.00	290 (5)	102.95	147.56	0.00%	**1404***	**1404***	**1404**	1404.00	0 (0)	11.17	96.50	0.00%
scp51–3	**1532**	1637	**1532**	1665.00	304 (4)	199.44	273.38	0.00%	**618***	**618***	**618**	618.00	0 (0)	13.47	137.75	0.00%
scp52–3	**1446**	**1446***	**1446**	1499.00	73 (1)	30.66	189.63	0.00%	**602***	**602***	**602**	602.00	0 (0)	5.17	105.80	0.00%
scp53–3	**1423**	**1423***	1479	1502.50	300 (3)	152.00	235.36	3.94%	**627***	**627***	**627**	627.00	0 (0)	14.55	144.83	0.00%
scp54–3	**1360**	1392	**1360**	1535.00	196 (2)	89.91	263.18	0.00%	**546***	**546***	**546**	546.00	0 (0)	75.80	121.81	0.00%
scp55–3	**1375**	**1375***	1382	1428.00	335 (5)	143.06	217.92	0.51%	**528***	**528***	**528**	528.00	0 (0)	2.91	105.14	0.00%
scp56–3	**1410**	**1410**	**1410**	1458.00	91 (1)	172.99	247.16	0.00%	**511***	**511***	**511**	511.00	0 (0)	6.50	96.35	0.00%
scp57–3	**1551**	**1551**	1571	1650.00	100 (1)	200.74	276.17	1.29%	**764***	**764***	**764**	764.00	0 (0)	77.41	137.03	0.00%
scp58–3	1574	1611	**1517**	1631.00	231 (3)	129.86	197.42	−3.62%	**650***	**650***	**650**	661.00	0 (0)	140.85	201.06	0.00%
scp59–3	**1518**	**1518**	1557	1664.50	100 (1)	216.12	285.26	2.57%	**660***	**660***	**660**	660.00	0 (0)	8.63	133.24	0.00%
scp510–3	1361	**1338***	1435	1467.50	356 (4)	95.67	182.38	7.25%	**642***	**642***	**642**	664.00	0 (0)	23.30	142.72	0.00%
scp61–3	4833	**2930**	**2930**	2930.00	2166 (63)	0.70	53.08	0.00%	2169	1807	**1733**	1733.00	969 (30)	31.20	58.72	−4.10%
scp62–3	4310	3437	**3361**	3469.50	2394 (59)	37.45	60.38	−2.21%	2518	2147	**2037**	2037.00	1482 (52)	28.30	52.86	−5.12%
scp63–3	5709	4840	**4105**	4492.50	3125 (70)	39.95	61.78	−15.19%	2925	2498	**2355**	2544.00	1375 (37)	25.45	50.20	−5.72%
scp64–3	4061	3447	**3211**	3286.50	2460 (63)	30.31	55.11	−6.85%	2128	**1899**	**1899**	1899.00	940 (32)	33.81	60.94	0.00%
scp65–3	4508	3345	**3181**	3260.00	2091 (60)	54.81	78.20	−4.90%	2319	1994	**1960**	1986.00	1071 (36)	43.16	70.00	−1.71%
scpa1–3	3717	3136	**2893**	3048.00	936 (17)	45.27	389.85	−7.75%	**963**	976	978	1000.00	52 (1)	343.19	454.07	1.56%
scpa2–3	4037	3291	**2934**	3250.00	826 (14)	338.52	489.32	−10.85%	**1048**	1127	1053	1103.50	0 (0)	141.39	417.33	0.48%
scpa3–3	3459	3728	**2647**	2790.50	1150 (23)	361.96	514.08	−23.47%	910	**907**	**907**	919.00	50 (1)	204.44	338.07	0.00%
scpa4–3	4025	3601	**2701**	3031.50	884 (13)	539.39	601.24	−24.99%	**946**	**946***	**946**	951.50	0 (0)	257.06	374.28	0.00%
scpa5–3	4216	3930	**3069**	3160.50	737 (10)	455.65	588.59	−21.91%	**894***	**894***	**894**	895.00	0 (0)	43.00	321.89	0.00%
scpb1–3	9378	4972	**4320**	4724.50	3666 (100)	150.86	255.21	−13.11%	6045	2975	**2863**	3005.00	1963 (83)	290.55	406.12	−3.76%
scpb2–3	10 581	5101	**4810**	5079.00	4074 (108)	198.02	287.22	−5.70%	6109	3663	**3044**	3350.00	2016 (77)	138.06	268.08	−16.90%
scpb3–3	9324	4678	**4099**	4384.50	3372 (99)	159.67	251.77	−12.38%	4802	3094	**2643**	2892.00	2052 (82)	246.63	349.38	−14.58%
scpb4–3	9232	4782	**3885**	4370.00	3084 (108)	42.98	233.85	−18.76%	5145	3129	**2470**	2865.50	1776 (69)	88.52	308.71	−21.06%
scpb5–3	10 922	5120	**4542**	5044.50	3570 (104)	156.44	257.60	−11.29%	6178	3185	**2944**	3068.00	2156 (77)	264.19	388.07	−7.57%
scpc1–3	14 664	9677	**7437**	7944.50	3900 (35)	525.47	601.88	−23.15%	1585	1766	**1583**	1663.00	0 (0)	573.22	601.44	−0.13%
scpc2–3	7473	4506	**3995**	4309.00	2211 (61)	88.92	580.51	−11.34%	1391	1470	**1249**	1351.00	264 (7)	423.39	601.52	−10.21%
scpc3–3	10 559	5767	**4611**	5002.00	2050 (39)	453.91	601.57	−20.05%	1553	1746	**1343**	1426.00	100 (2)	484.02	601.41	−13.52%
scpc4–3	7997	5791	**4457**	4695.00	1900 (44)	394.38	566.19	−23.04%	1596	1543	**1385**	1494.00	304 (8)	488.61	592.34	−10.24%
scpc5–3	10 557	5673	**5239**	5493.50	3100 (56)	355.77	601.83	−7.65%	1370	1533	**1295**	1451.50	50 (1)	281.63	592.41	−5.47%
scpd1–3	12 058	6412	**5340**	5733.50	4662 (101)	109.91	407.39	−16.72%	8882	5634	**4104**	4345.50	3330 (125)	317.49	494.85	−27.16%
scpd2–3	11 682	6192	**5520**	5929.00	4923 (116)	263.38	391.35	−10.85%	9258	4485	**4213**	4483.50	3456 (124)	276.17	461.47	−6.06%
scpd3–3	11 049	6324	**5572**	5987.50	4833 (131)	29.03	333.22	−11.89%	8530	4890	**4468**	4596.00	3357 (112)	89.42	440.39	−8.63%
scpd4–3	13 535	9939	**7873**	8784.50	7210 (115)	81.22	378.18	−20.79%	13 738	7102	**6209**	6431.00	5390 (132)	509.02	602.33	−12.57%
scpd5–3	13 176	6588	**5718**	5993.00	4887 (143)	72.97	376.96	−13.21%	8403	4740	**4287**	4554.00	3609 (136)	412.71	544.11	−9.56%
scpe1–3	**26***	**26***	**26**	27.00	17 (3)	1.52	9.25	0.00%	**23***	**23***	**23**	24.00	14 (3)	0.89	10.13	0.00%
scpe2–3	**28***	**28***	**28**	28.00	19 (3)	0.09	7.39	0.00%	**25***	**25***	**25**	25.00	16 (3)	0.50	7.98	0.00%
scpe3–3	**24***	**24***	**24**	24.00	18 (1)	0.05	5.89	0.00%	**23***	**23***	**23**	23.00	17 (1)	0.05	6.03	0.00%
scpe4–3	**25***	**25***	**25**	25.00	16 (3)	0.12	8.39	0.00%	**22***	**22***	**22**	22.00	13 (3)	0.09	8.50	0.00%
scpe5–3	**28***	**28***	**28**	28.00	19 (3)	0.44	7.19	0.00%	**25***	**25***	**25**	25.00	16 (3)	0.06	7.12	0.00%
Set-B																
scpclr10–3	**1926***	**1926***	**1926**	1926.00	1893 (55)	1.03	21.69	0.00%	**1871***	**1871***	**1871**	1871.00	1838 (55)	0.22	20.61	0.00%
scpclr11–3	4215	3649	**3501**	3501.00	3468 (55)	1.03	27.39	−4.06%	**3446**	**3446**	**3446**	3446.00	3413 (55)	0.80	27.70	0.00%
scpclr12–3	9236	7455	**6429**	6429.00	6396 (55)	21.31	40.45	−13.76%	7728	8142	**6374**	6374.00	6341 (55)	21.66	41.05	−17.52%
scpclr13–3	17 926	20 357	**13 267**	14 122.50	13234 (55)	48.64	79.23	−25.99%	21 299	21 107	**13 212**	14 033.00	13179 (55)	3.64	71.66	−37.40%
scpcyc06–3	**126***	**126***	**126**	126.00	42 (27)	0.47	13.80	0.00%	**99***	**99***	**99**	99.00	15 (9)	0.61	15.70	0.00%
scpcyc07–3	**335***	**335**	**335**	335.00	136 (74)	1.23	41.36	0.00%	**250***	**250**	**250**	253.00	49 (42)	33.00	50.13	0.00%
scpcyc08–3	911	947	**908**	911.00	434 (204)	6.83	599.41	−0.33%	**659**	663	677	689.50	191 (147)	366.02	602.08	2.73%
scpcyc09–3	2332	2654	**2317**	2338.00	1222 (488)	402.47	602.13	−0.64%	**1678**	1762	1751	1781.00	611 (463)	448.85	602.19	4.35%
scpcyc10–3	6755	6420	**5728**	5794.00	3280 (1250)	456.13	602.21	−10.78%	**4441**	4594	4525	4586.50	1931 (1397)	568.99	602.23	1.89%
scpcyc11–3	19 380	23 938	**13 727**	14 107.50	8334 (3030)	569.43	602.96	−29.17%	12 694	14 388	**11 391**	11 534.50	5565 (3942)	288.55	603.00	−10.26%
Set-C																
scpnre1–3	4233	4369	**3711**	3893.00	3546 (36)	333.32	581.88	−12.33%	4773	4296	**3648**	3860.00	3204 (36)	5.16	465.45	−15.08%
scpnre2–3	4905	4316	**3693**	4000.50	3372 (36)	263.22	495.97	−14.43%	5210	4473	**3449**	3876.00	3090 (36)	7.81	452.55	−22.89%
scpnre3–3	4680	4912	**3826**	3998.50	3510 (36)	137.64	591.86	−18.25%	5625	4595	**3549**	3848.00	3357 (36)	465.75	566.35	−22.76%
scpnre4–3	4221	4692	**3726**	3881.50	3264 (36)	157.58	574.05	−11.73%	4856	4095	**3558**	3832.50	3330 (45)	145.59	556.71	−13.11%
scpnre5–3	5139	4696	**3660**	3898.00	3348 (36)	297.39	566.87	−22.06%	4530	4536	**3579**	3768.50	3234 (36)	123.27	565.82	−20.99%
scpnrf1–3	1835	1621	**1185**	1232.00	1098 (10)	19.02	559.89	−26.90%	1807	1546	**1175**	1234.50	1088 (10)	78.53	603.60	−24.00%
scpnrf2–3	1888	1607	**1243**	1243.00	1090 (10)	34.19	570.86	−22.65%	1968	1602	**1227**	1233.00	1116 (10)	483.97	608.15	−23.41%
scpnrf3–3	1813	1564	**1184**	1205.00	1088 (10)	1.83	561.74	−24.30%	1892	1556	**1174**	1195.00	1078 (10)	41.28	576.44	−24.55%
scpnrf4–3	1696	1602	**1183**	1183.00	1105 (10)	2.20	560.21	−26.15%	1829	1533	**1173**	1173.00	1095 (10)	357.62	563.22	−23.48%
scpnrf5–3	1445	1590	**1226**	1251.00	1138 (10)	78.03	588.70	−15.16%	1708	1353	**1216**	1245.50	1128 (10)	204.60	603.42	−10.13%
scpnrg1–3	33 276	**13 234**	13 302	13 616.50	10560 (567)	240.19	609.41	0.51%	15 738	**6110**	6151	6424.00	3930 (285)	392.78	615.22	0.67%
scpnrg2–3	34 092	13 663	**13 110**	13 644.00	10500 (582)	48.30	607.63	−4.05%	15 524	7114	**6272**	6780.50	4010 (274)	575.94	613.44	−11.84%
scpnrg3–3	28 168	14 737	**14 287**	14 946.00	12078 (603)	65.88	615.52	−3.05%	17 639	7779	**6947**	7464.50	4686 (284)	587.18	608.29	−10.70%
scpnrg4–3	30 425	14 053	**13 673**	13 983.50	11330 (614)	38.61	613.43	−2.70%	17 604	6653	**6589**	7023.00	4900 (340)	348.42	613.66	−0.96%
scpnrg5–3	26 931	15 077	**13 529**	14 169.50	10520 (589)	90.75	610.50	−10.27%	15 825	8521	**6493**	7002.50	4080 (277)	344.52	614.15	−23.80%
scpnrh1–3	18 114	12 252	**9315**	9856.50	8559 (190)	129.91	616.55	−23.97%	17 637	11 370	**8574**	9247.00	7956 (153)	160.64	627.48	−24.59%
scpnrh2–3	18 537	12 972	**9101**	9788.50	7974 (153)	14.45	613.57	−29.84%	16 860	10 887	**8880**	9147.00	8019 (190)	9.11	623.06	−18.43%
scpnrh3–3	17 982	12 348	**9294**	9696.00	8511 (171)	12.13	617.05	−24.73%	17 082	10 951	**8833**	9175.50	7971 (190)	140.39	625.81	−19.34%
scpnrh4–3	16 749	12 000	**9644**	9784.50	8727 (171)	100.91	616.32	−19.63%	16 815	10 599	**8688**	9150.00	8112 (171)	117.17	627.57	−18.03%
scpnrh5–3	15 654	12 408	**9461**	9762.50	8814 (208)	80.34	620.76	−23.75%	14 967	11 307	**9070**	9249.50	7701 (190)	103.78	629.44	−19.78%
Avg							**339.18**	**−9.28%**							**325.98**	**−7.32%**
#Best/Opt			**72**								**74**					
AvgRSD			**2.31%**								**1.56%**					

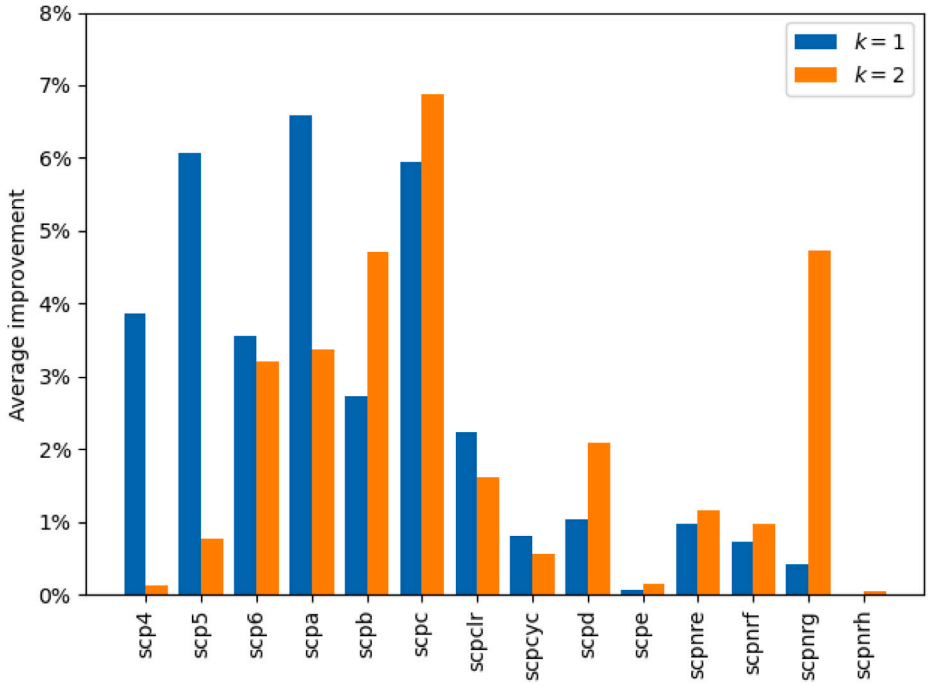


Fig. 3. Average improvement of the incumbent solution value before and after the second stage of parallel GRASP.

where this solution is known. Moreover, the average percentage gaps equal to −9.28% for $k = 1$ and −7.32% for $k = 2$ further highlight the effectiveness of our algorithm. Finally, the standard deviation values show that our algorithm is also stable with an average relative standard deviation equal to 2.31% and 1.56%, respectively. The results in the *Gap* column confirm that the hardest instances to solve using the mathematical formulations are the ones in *Set-C*. Indeed, the average percentage gap computed on these instances is equal to −16.77% for $k = 1$, and to −17.36% for $k = 2$. Nevertheless, even for parallel GRASP, these instances consistently prove to be the most demanding, frequently resulting in the algorithm reaching the time limit. Regarding Gurobi's solutions, it is worth highlighting the distinct performance of the quadratic model, which impressively outperforms the other two integer linear programming models on the *Set-C* instances, thus providing a better bound for the GRASP evaluation. Finally, regarding the computational time, parallel GRASP requires on average less than 340 s for $k = 1$ and less than 326 s for $k = 2$. In 62% of the instances the algorithm stops before reaching the time limit of 600 s.

4.5. Computational results: performance evaluation of parallel GRASP components

We conclude this section by providing some charts that allow us to better understand the contribution of the different components of the parallel GRASP algorithm and to highlight how the structural differences of the test instances at hand reflect on the internal workload. Fig. 3 shows the performance impact of the second stage in parallel GRASP (when GRASP is rerun restricting the subsets taken into account during the first phase) that consistently improves the value of the final solution. Each bar represents the percentage improvement of the average solution value computed on the instances of a given family (each instance has been run ten times). The family that benefits the most by the second stage is the *scpc* family that has an average improvement of ∼ 7%. A notable exception is given by the *scpe* and *scpnrh* families, for both $k = 1$ and $k = 2$, where the improvement of the final solution during the second stage is low.

Fig. 4 quantifies the benefit yielded from the adoption of the shared cache during the first phase of parallel GRASP (see Section 3.2). For each family of instances, each bar of the chart represents the average percentage number of times when parallel GRASP has been able to retrieve data from the shared cache avoiding further computations (each instance has been run ten times). The most relevant result is observed again for the *scpe* family on which the parallel GRASP algorithm reuses previously computed information ∼60% of times. For all other families, this value ranges between 1% and 30% and for half of the families this value is equal to at least 10%. It is important to note that whenever the algorithm retrieves the required information from the shared cache, it can substitute a procedure with a time cost of $O(n + m \log_2 n)$ with a straightforward memory read, which incurs a cost of $O(1)$.

Figs. 5(a) and 5(b) show how many times the main loop of GraspPhase1 and GraspPhase2 functions have been repeated on average in order to complete a single iteration of the parallel GRASP algorithm. Recall that one iteration of the main loop in the first phase of the parallel GRASP amounts to the addition of a subset to the current partial cover, while one iteration of the main loop of the second phase of the algorithm corresponds to searching the neighborhood of the current solution for a better cover. Each bar of the figure shows the average number of times that a given phase has been repeated across all instances belonging to a given family (each instance has been run ten times). The particularly high values of the *scpcyc* family in Figs. 5(a) and 5(b) are due to the fact that each available subset belonging to the instances of this family has at most 4 elements and the total number of elements in U ranges from 240 in instance *scpcyc06-3* up to 28 160 in instance *scpcyc11-3*.

4.6. Computational results: parallel GRASP on the standard Set Covering Problem

As SCP is a special case of the SCP-CS for the case where conflict costs are all zero, we also tested the performance of our parallel GRASP on benchmark instances for the SCP available in the OR library (Beasley, 1990a) for which the optimal solution values are known. We are interested in checking the effectiveness of our algorithm by comparing its solution values with the optimal ones. It is important to highlight that no particular expedient or modification has been applied to the original version of the parallel GRASP to adapt it to this

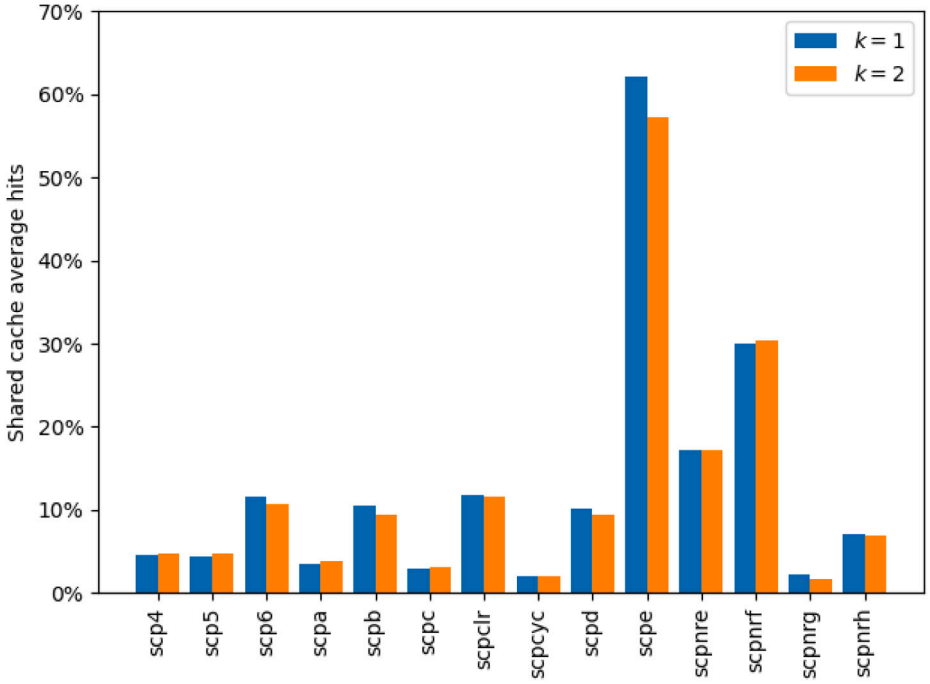


Fig. 4. Average number of times precomputed data has been retrieved from the shared cache.

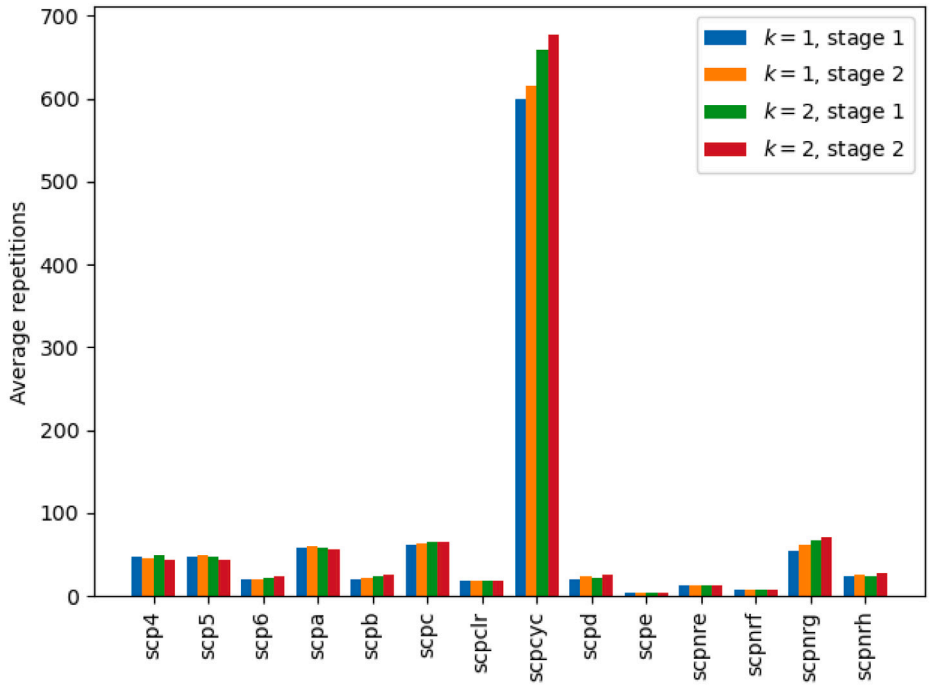
Table 7
Parallel GRASP on standard Set Covering Problem instances from OR library (Beasley, 1990a).

Parallel GRASP on SCP instances														
Instance	Opt	ObjBest	ObjMed	Gap	Instance	Opt	ObjBest	ObjMed	Gap	Instance	Opt	ObjBest	ObjMed	Gap
scp41	429	429	429	0.00%	scp51	253	253	253	0.00%	scp61	138	138	138	0.00%
scp42	512	512	512	0.00%	scp52	302	302	304	0.00%	scp62	146	146	146	0.00%
scp43	516	516	516	0.00%	scp53	226	226	226	0.00%	scp63	145	145	145	0.00%
scp44	494	494	494	0.00%	scp54	242	242	242	0.00%	scp64	131	131	131	0.00%
scp45	512	512	512	0.00%	scp55	211	211	211	0.00%	scp65	161	161	161	0.00%
scp46	560	560	560	0.00%	scp56	213	213	213	0.00%					
scp47	430	430	430	0.00%	scp57	293	293	293	0.00%					
scp48	492	492	492	0.00%	scp58	288	288	288	0.00%					
scp49	641	641	641	0.00%	scp59	279	279	279	0.00%					
scp410	514	514	514	0.00%	scp510	265	265	265	0.00%					
scpa1	253	254	256	0.40%	scpb1	69	69	69	0.00%	scpc1	227	229	233	0.88%
scpa2	252	252	257	0.00%	scpb2	76	76	76	0.00%	scpc2	219	223	226	1.83%
scpa3	232	234	235	0.86%	scpb3	80	80	80	0.00%	scpc3	243	245	251	0.82%
scpa4	234	235	237	0.43%	scpb4	79	79	79	0.00%	scpc4	219	223	225	1.83%
scpa5	236	237	237	0.42%	scpb5	72	72	72	0.00%	scpc5	215	215	219	0.00%
scpd1	60	60	61	0.00%	scpe1	5	5	5	0.00%					
scpd2	66	66	67	0.00%	scpe2	5	5	5	0.00%					
scpd3	72	72	73	0.00%	scpe3	5	5	5	0.00%					
scpd4	62	62	63	0.00%	scpe4	5	5	5	0.00%					
scpd5	61	61	61	0.00%	scpe5	5	5	5	0.00%					

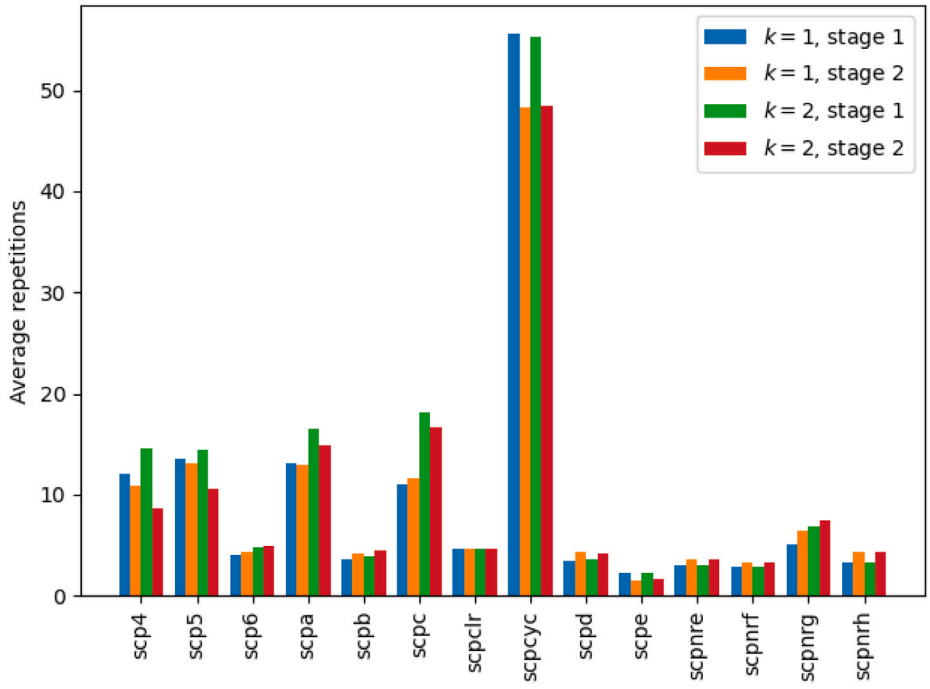
special case. For this reason, its efficiency results in being penalized by all the steps concerning the conflicts that are performed in vain. Although not directly comparable in terms of computational time with specialized state-of-the-art heuristic methods (see Lan et al. (2007)) that find optimal solutions in a few seconds, our method performs very well. Table 7 shows the results obtained running parallel GRASP ten times for each instance using the same settings described in Section 4.1. Column *Instance* reports the name of the original instance from the OR library, column *Opt* shows the value of the optimal solution from the literature, columns *ObjBest* and *ObjMed* contain the best solution value and the median value found by parallel GRASP out of the 10 trials, respectively. Finally, the *Gap* column shows the percentage gap between the optimal solution value and the one reported in the *ObjBest* column. Parallel GRASP comes out to be effective also for the SCP, finding the optimal solution in 42 out of 50 instances. Moreover, its average percentage gap is equal to 0.15% with a peak of 1.83% that occurs only twice. It is worth noting that for each instance the values of columns *ObjBest* and *ObjMed* are very close, certifying the stability of GRASP.

5. Conclusions

In this paper, we propose a new variant of the Set Covering Problem that introduces conflicts among subsets. Two subsets in conflict can belong to the same solution provided that a cost (proportional to the number of items that exceeds such a threshold) is paid. We provide three mathematical models for the problem: a pure binary linear formulation, a quadratic model, and a mixed integer linear program (the



(a) Average number of repetitions of Phase 1.



(b) Average number of repetitions of Phase 2.

Fig. 5. Average number of repetitions of phase 1 (a) and phase 2 (b) in the parallel GRASP algorithm.

latter tailored to our specific application where two subsets are in conflict when they share a number of elements exceeding a given threshold.). Moreover, we develop a novel parallel GRASP algorithm that exploits information sharing for the most demanding tasks. Unlike classical parallel implementations that transfer only the incumbent solutions among threads, our algorithm also shares intermediate computations and this allows us the removal of redundant operations across threads. This approach results in an overall speedup of the procedure. The parallel GRASP (tested with a time limit of 600 s) is very effective, frequently outperforming Gurobi when solving three different mathematical formulations by using the same number of processors and a larger amount of time (1 h).

Starting from the definition of SCP-CS introduced in this work, future research topics may include, for example, the study of the

special case when the conflict threshold k is equal to 0 (i.e. when any pair of overlapping subsets is in conflict) and the evaluation of other meaningful criteria to set the cost of conflicts. Specifically, one can consider a problem's variant that minimizes the total overlap or the maximum overlap of a single element, employing a classical min–max objective function.

CRediT authorship contribution statement

Francesco Carrabs: Data curation, Formal analysis, Validation, Visualization, Writing – original draft, Writing – review & editing, Conceptualization. **Raffaele Cerulli:** Formal analysis, Project administration, Supervision, Writing – original draft, Conceptualization. **Renata Mansini:** Formal analysis, Methodology, Project administration, Supervision, Writing – original draft, Writing – review & editing, Conceptualization. **Lorenzo Moreschini:** Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing, Conceptualization. **Domenico Serra:** Data curation, Formal analysis, Validation, Visualization, Writing – original draft, Writing – review & editing, Conceptualization.

Data availability

We share the link to data and source code hosted on GitHub.

Appendix. GRASP phase 1: the Z data structure

Algorithm 7 Phase 1: the $Z = (H_{\max}, H_{\min})$ data structure

```
1: function PopRandomCandidate(Z)
2:     r ← RandomInteger(0, |H_max| − 1)
3:     (j, v_j) ← HeapPop(H_max, r)
4:     if |H_min| > 0 then
5:         (l, v_l) ← HeapPop(H_min, 0)
6:         H_max ← HeapPush(H_max, l, v_l)
7:     end if
8:     return j
9: end function

10: function UpdateCandidateStructure(Z, j, v_j)
11:     if j ∈ H_max then
12:         HeapUpdate(H_max, j, v_j)
13:     else
14:         HeapUpdate(H_min, j, v_j)
15:     end if
16:     (a, v_a) ← HeapPeek(H_max)
17:     (b, v_b) ← HeapPeek(H_min)
18:     if v_a > v_b then
19:         HeapPop(H_max, 0)
20:         HeapPop(H_min, 0)
21:         HeapPush(H_max, b, v_b)
22:         HeapPush(H_min, a, v_a)
23:     end if
24:     return Z
25: end function
```

This appendix provides details about the data structure Z employed during the first phase of the parallel GRASP algorithm. The main goal of this data structure is to ease the computation of the restricted candidate list (RCL) during the construction of a feasible solution W. The most computationally expensive task is by far constructing the collection $\mathcal{U}_W$ and sorting it by non-decreasing values of θ_W. The key observation to achieve this in an efficient way is to observe that the construction of a feasible solution W is pursued incrementally by expanding a partial solution. Each time a new subset U_j ($j \in N \setminus W$) is added to W, the modification of $\mathcal{U}_W$ can be incrementally computed taking into account only the subsets already present inside W that have a non-zero conflict cost with U_j, whereas all the other ones (the most part) will be unaffected. Similarly, after the addition of U_j to W, the values computed by $\theta_W(U_l)$ will change only for those subsets U_l ($l \in N \setminus W$) conflicting with U_j, leaving unmodified the values associated to all the others. Therefore, our approach aims to take advantage of this observation by using the right data structure (a specialized pair of binary heaps) to incrementally track the changes to $\mathcal{U}_W$ and θ_W during the construction of a feasible solution W. Determining the RCL essentially requires to keep track of the current costs of the subsets inside $\mathcal{U}_W$ in order to pick one at random among the top p most promising ones. For this reason, the data structure Z is implemented as a pair of heaps $Z = (H_{\max}, H_{\min})$. The former is a maximum key–value binary heap with fixed size p and the latter is a minimum key–value binary heap with unbounded size. Each entry in both these data structures is a key–value pair: keys are the indexes of the available subsets $U_j \in \mathcal{U}_W$ and their associated values are those provided by the ranking function θ_W. The heap structure is computed with respect to the value of each key–value pair. Alongside the array that stores the heap as a binary tree, we also keep a hash map that maps each key to the current position of the key–value pair inside the array.

Whenever a key–value pair $(U_j, \theta_W(U_j))$ is inserted in Z, we store it inside either $H_{\max}$ or $H_{\min}$ enforcing the following conditions: each value inside $H_{\max}$ must not be greater than any value stored inside $H_{\min}$ and pairs cannot be inserted inside $H_{\min}$ if $H_{\max}$ has not reached its size limit of p entries. In this way, we are able to perform on the data structure $Z = (H_{\max}, H_{\min})$ all the tasks required by our use case in sublinear time. Since both heaps are stored in an array, picking one item at random among the most promising p ones simply requires generating a random integer number r in $[0, |H_{\max}| - 1]$ and popping the element in position r from the array, preserving the heap structure. This can be achieved in $O(\log_2 p)$ time. Similarly, when W is modified, updating the cost of a subset U_j inside $\mathcal{U}_W$ is as simple as establishing whether U_j belongs to $H_{\max}$ or $H_{\min}$ and changing its position inside the heap according to its new current cost. When doing this we must handle an important case: if U_j is inside $H_{\min}$ and reaches the top of the heap after the update, we may need to swap the elements at the top of the two heaps in order to maintain the separation property.

Algorithm 7 shows the pseudo-code of the methods in Algorithm 3 that modify the data structure Z. All functions whose name begins with "Heap" behave as in the traditional implementation of the heap data structure; their description can be found in most textbooks (e.g. Cormen et al. (2022)) or in the source code available at https://github.com/lmores/or-scp-cs-src/blob/master/utils/heaps.py.

References

Amaldi, E., Capone, A., Malucelli, F., Signori, F., 2002. Umts radio planning: Optimizing base station configuration. In: IEEE Vehicular Technology Conference. 56, pp. 768–772.

Attar, A., Li, H., Leung, V.C.M., 2011. Green last mile: How fiber-connected massively distributed antenna systems can save energy. IEEE Wirel. Commun. 18 (5), 66–74.

Banik, A., Panolan, F., Raman, V., Sahlot, V., Saurabh, S., 2020. Parameterized complexity of geometric covering problems having conflicts. Algorithmica 82 (1), 1–19.

Beasley, J., 1987. An algorithm for set covering problem. European J. Oper. Res. 31 (1), 85–93.

Beasley, J.E., 1990a. Or-library. Website. URL http://people.brunel.ac.uk/mastjjb/jeb/orlib/scpinfo.html.

Beasley, J.E., 1990b. Or-library: Distributing test problems by electronic mail. J. Oper. Res. Soc. 41 (11), 1069–1072.

Beasley, J., 1992. A lagrangean heuristic for set covering problems. In: Combinatorial Optimization. Springer, Berlin, Heidelberg, pp. 325–326.

Bilal, N., Galinier, P., Guibault, F., 2014. An iterated-tabu-search heuristic for a variant of the partial set covering problem. J. Heuristics 20 (2), 143–164.

Carrabs, F., Cerrone, C., Pentangelo, R., 2019. A multiethnic genetic approach for the minimum conflict weighted spanning tree problem. Networks 74 (2), 134–147.

Carrabs, F., Cerulli, R., Pentangelo, R., Raiconi, A., 2021. Minimum spanning tree with conflicting edge pairs: a branch-and-cut approach. Ann. Oper. Res. 298 (1), 65–78.

Carrabs, F., Gaudioso, M., 2021. A lagrangian approach for the minimum spanning tree problem with conflicting edge pairs. Networks 78 (1), 32–45.

Cerri, G., Leo, R.D., Micheli, D., Russo, P., 2002. Reduction of electromagnetic pollution in mobile communication systems by an optimized location of radio base stations. In: Proceedings of the XXVIIth URSI General Assembly. Vol. 864, pp. 1–4.

Cerrone, C., Cerulli, R., Golden, B., 2017. Carousel greedy: A generalized greedy algorithm with applications in optimization. Comput. Oper. Res. 85, 97–112.

Chvatal, V., 1979. A greedy heuristic for the set-covering problem. Math. Oper. Res. 4 (3), 233–235.
Colombi, M., Corberán, Á., Mansini, R., Plana, I., Sanchis, J.M., 2017. The directed profitable rural postman problem with incompatibility constraints. European J. Oper. Res. 261 (2), 549–562.
Coniglio, S., Furini, F., San Segundo, P., 2021. A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. European J. Oper. Res. 289 (2), 435–455.
Cormen, T., Leiserson, C., Rivest, R., Stein, C., 2022. Introduction to Algorithms, fourth ed. MIT Press.
Darmann, A., Pferschy, U., Schauer, J., Woeginger, G.J., 2011. Paths, trees and matchings under disjunctive constraints. Discrete Appl. Math. 159 (16), 1726–1735.
Ekici, A., 2021. Bin packing problem with conflicts and item fragmentation. Comput. Oper. Res. 126.
Epstein, L., Favrholdt, L.M., Levin, A., 2011. Online variable-sized bin packing with conflicts. Discrete Optim. 8 (2), 333–343.
Feo, T., Resende, M., 1989. A probabilistic heuristic for a computationally difficult set covering problem. Oper. Res. Lett. 8 (2), 67–71.
Festa, P., Resende, M., 2002. Grasp: An annotated bibliography. In: Essays and Surveys in Metaheuristics. Wiley-Blackwell, pp. 32–45.
Gendreau, M., Manerba, D., Mansini, R., 2016. The multi-vehicle traveling purchaser problem with pairwise incompatibility constraints and unitary demands: A branch-and-price approach. European J. Oper. Res. 248 (1), 59–71.
Gobbi, A., Manerba, D., Mansini, R., Zanotti, R., 2023. Hybridizing adaptive large neighborhood search with kernel search: a new solution approach for the nurse routing problem with incompatible services and minimum demand. International Transactions in Operational Research 30 (1), 8–38.
Grossman, T., Wool, A., 1997. Computational experience with approximation algorithms for the set covering problem. European J. Oper. Res. 101 (1), 81–92.
Guo, W., Wang, S., Chu, X., Zhang, J., Chen, J., Song, H., 2013. Automated small-cell deployment for heterogeneous cellular networks. IEEE Commun. Mag. 51 (5), 46–53.
Gusmeroli, N., Hrga, T., Lužar, B., Povh, J., Siebenhofer, M., Wiegele, A., 2022. Biqbin: A parallel branch-and-bound solver for binary quadratic problems with linear constraints. ACM Trans. Math. Softw. 48 (2).
Hifi, M., Michrafy, M., 2007. Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. Comput. Oper. Res. 34 (9), 2657–2673.
Jacob, A., Majumdar, D., Raman, V., 2019. Parameterized complexity of conflict-free set cover. In: van Bevern, R., Kucherov, G. (Eds.), Computer Science – Theory and Applications. pp. 191–202.
Könemann, J., Parekh, O., Segev, D., 2011. A unified approach to approximating partial covering problems. Algorithmica 59 (4), 489–509.
Lan, G., DePuy, G.W., Whitehouse, G.E., 2007. An effective and simple heuristic for the set covering problem. European J. Oper. Res. 176 (3), 1387–1403.
Manerba, D., Mansini, R., 2016. The nurse routing problem with workload constraints and incompatible services. IFAC-PapersOnLine 49 (12), 1192–1197.
Öncan, T., Kuban Altınel, İ., 2018. A branch-and-bound algorithm for the minimum cost bipartite perfect matching problem with conflict pair constraints. Electron. Notes Discrete Math. 64, 5–14.
Öncan, T., Zhang, R., Punnen, A., 2013. The minimum cost perfect matching problem with conflict pair constraints. Comput. Oper. Res. 40 (4), 920–930.
Pferschy, U., Schauer, J., 2009. The knapsack problem with conflict graphs. J. Graph Algorithms Appl. 13, 233–249.
Pferschy, U., Schauer, J., 2013. The maximum flow problem with disjunctive constraints. J. Combinat. Optim. 26 (1), 109–119.
Rostami, B., Errico, F., Lodi, A., 2023. A convex reformulation and an outer approximation for a large class of binary quadratic programs. Oper. Res. 71 (2), 471–486.
Sadykov, R., Vanderbeck, F., 2013. Bin packing with conflicts: A generic branch-and-price algorithm. INFORMS J. Comput. 25 (2), 244–255.
Saffari, S., Fathi, Y., 2022. Set covering problem with conflict constraints. Comput. Oper. Res. 143, 105763.
Sambo, Y.A., Héliot, F., Imran, M.A., 2015. A survey and tutorial of electromagnetic radiation and reduction in mobile communication systems. IEEE Commun. Surv. Tutor. 17 (2), 790–802.
Saminathan, B., Tamilarasan, I., Murugappan, M., 2017. Energy and electromagnetic pollution considerations in arof-based multi-operator multi-service systems. Photonic Netw. Commun. 34 (2), 221–240.
Şuvak, Z., Altınel, İ., Aras, N., 2020. Exact solution algorithms for the maximum flow problem with additional conflict constraints. European J. Oper. Res. 287 (2), 410–437.
Wang, X., Jiang, Z., Gao, S., 2013. An enhanced difference method for multi-objective model of cellular base station antenna configurations. Commun. Netw. 05, 361–366.