A new combinatorial branch-and-bound algorithm for the Knapsack Problem with Conflict Graph

Stefano Coniglio

University of Southampton, Southampton, United Kingdom
School of Mathematical Sciences
s.coniglio@soton.ac.uk

Fabio Furini

Istituto di Analisi dei Sistemi ed Informatica "A. Ruberti"
Consiglio Nazionale delle Ricerche (IASI-CNR), Roma, Italy
f.furini@iasi.cnr.it

Pablo San Segundo

Universidad Politécnica de Madrid (UPM), Madrid, Spain
Center of Automation and Robotics (CAR), Madrid, Spain
pablo.sansegundo@upm.es

Abstract

We study the Knapsack Problem with Conflict Graph (KPCG), a generalization of the Knapsack Problem in which a conflict graph specifies pairs of items (vertices of the graph) which cannot be simultaneously selected in a solution. The KPCG asks for determining a maximum-profit subset of items of total weight no larger than the knapsack capacity which do not violate any of the item conflicts. In this work, we propose a novel combinatorial branch-and-bound algorithm for the KPCG based on an n-ary branching scheme. Our algorithm effectively combines different procedures for pruning the branch-and-bound nodes based on different relaxations of the KPCG. Key to the algorithm is its high pruning potential and the low computational effort that it requires to process each branch-and-bound node. An extensive set of experiments carried out on the benchmark instances typically used in the literature shows that, for edge densities ranging from 0.1 to 0.9, our algorithm is faster by up to two orders of magnitude than the state-of-the-art method and by up to several orders of magnitude than a state-of-the-art mixed-integer linear programming solver.

Keywords: Knapsack Problem with Conflict Graph, Maximum Weighted Clique Problem, Branch-and-Bound algorithm.

1. Introduction

Given a set V of n items with a positive integer profit p_i and a positive integer weight w_i, for all $i \in V$, and an integer capacity c, the classical *Knapsack Problem* (KP) asks for identifying a subset of items of maximum profit whose total weight is no larger than c. The KP is one of the most studied problems in combinatorial optimization: it is known to be

$\mathcal{NP}$-hard (albeit weakly) since the seminal work of Karp [26], and very effective algorithms are available for its solution [33]. For a comprehensive survey on algorithms and applications, we refer the reader to [27, 32].

In this paper, we study a generalization of the KP known as *Knapsack Problem with Conflict Graph* (KPCG). Given a simple undirected (conflict) graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, where the sets of vertices and edges represent, respectively, knapsack items and conflicts between pairs of them (i.e., $\{i, j\} \in E$ if and only if items i and j cannot be simultaneously contained in a solution), the KPCG asks for a maximum-profit *conflict-free* subset of items whose total weight does not exceed the knapsack capacity. If the item weights are ignored, the KPCG admits the Maximum Weighted Independent Set Problem (MWISP) as a special case. As the MWISP is strongly $\mathcal{NP}$-hard and inapproximable in polynomial time to within any polynomial factor unless $\mathcal{P} = \mathcal{ZPP}$ [24], the same hardness result applies to the KPCG. This shows that the problem is considerably harder than the KP.

The KPCG plays an important role in the Operations Research literature. It arises as pricing subproblem when solving with branch-and-price algorithms both the Bin Packing Problem (BPP) (with a Ryan-Foster branching scheme) [51] and the Bin Packing Problem with Conflicts (BPPC) (with any branching scheme) [4]. Besides its role as a subproblem, the KPCG is of interest on its own right. Real-world applications arise in any context where a subset of items has to be selected subject to a capacity (or budget) constraint and a set of incompatibility/conflict constraints prevent the decision maker from selecting certain pairs of items. This is the case, for instance, of profit-maximization scheduling problems with a single machine, in which G is a graph with vertices corresponding to tasks and conflicts represent pairs of tasks which cannot be executed in parallel (due to, e.g., requiring the same nonpreemptible resource).

1.1. Notation

In the remainder of the paper, we adopt both terms "vertex" and "item" to identify the members of V, preferring the former when referring to the conflict-graph aspect of the KPCG and the latter when referring to its knapsack aspect. We will use the word *node* only when referring to a node of the branch-and-bound tree.

Throughout the paper, we call a subset of vertices a *clique* if every pair of its vertices are neighbors. We denote by $\overline{G} = (V, \overline{E})$ the complement graph of G, where $\overline{E}$ is the set of nonedges of G, i.e., $\overline{E} = \{e = \{i, j\} \text{ with } i, j \in V : e \notin E\}$. We call a subset of vertices an *independent set* (or *stable set*) if it forms a clique in $\overline{G}$. For each $i \in V$, we denote its *neighborhood* by $N(i) := \{j \in V : \{i, j\} \in E\}$ and its *antineighborhood* by $\overline{N}(i) := \{j \in V : \{i, j\} \in \overline{E}\}$. For each $i \in V$, $\delta(i) \subseteq E$ denotes the subset of edges incident with i. For each $U \subseteq V$, we denote by $G[U] = (U, E[V])$ the graph *induced* by U, where $E[U]$ corresponds to the subset of edges incident with two vertices in U.

Given a subset of items $I \subseteq V$, we denote their total profit and weight by:

$$p(I) = \sum_{i \in I} p_i \quad \text{and} \quad w(I) = \sum_{i \in I} w_i.$$

Notice that, for I to be a feasible solution to the KPCG, it must be an independent set and it must satisfy $w(I) \leq c$. Given any $\hat{V} \subseteq V$ and $\hat{c} \in \mathbb{R}^+$ with $\hat{c} \leq c$, we denote by $KPCG(\hat{V}, \hat{c})$

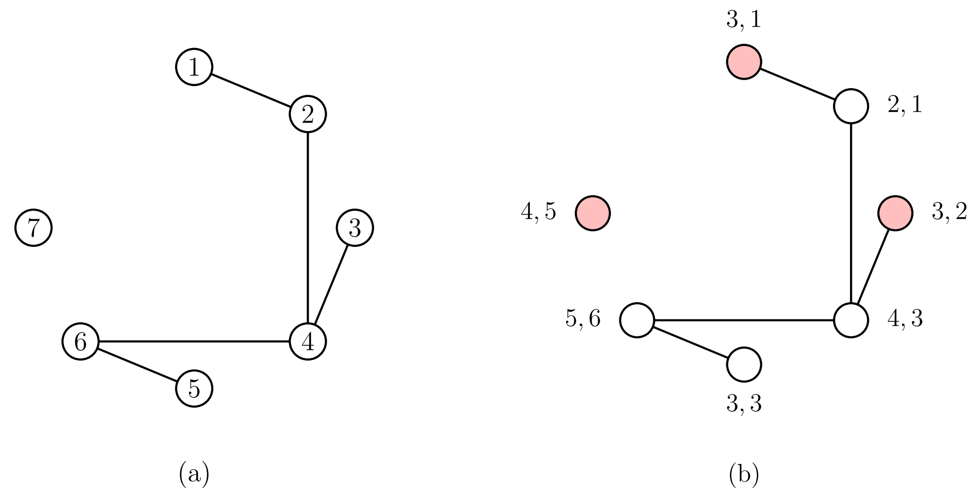
the optimal solution value of the KPCG on an instance with vertex set $\hat{V}$, capacity $\hat{c}$, and conflicts corresponding to the edges of $G[\hat{V}]$.

Throughout the paper, we assume that the vertices of G are sorted in nonincreasing order of profit-over-weight ratio. That is, we assume that the vertex set $V = \{1, 2, \ldots, n\}$ is ordered so to satisfy the following relationship:

$$\frac{p_j}{w_j} \geq \frac{p_{j+1}}{w_{j+1}} \qquad j = 1, \ldots, n-1. \tag{1}$$

1.2. Illustrative Example

Figure 1 reports an illustration of a KPCG instance with 7 vertices and 5 conflicts ($n = 7$ and $m = 5$) and capacity $c = 8$. Part (a) depicts the conflict graph G along with the indices of the vertices. Part (b) reports the profit and weight of each vertex separated by a comma (first the profit and then the weight). The vertices in the independent set $I = \{1, 3, 7\}$, highlighted in red, correspond to an optimal KPCG solution (of value $KPCG(V, 8) = 10$).



- $n = 7 = |V|, \;\; m = 5 = |E|, \;\; c = 8$
- $\frac{p_1}{w_1} = \frac{3}{1} \geq \frac{p_2}{w_2} = \frac{2}{1} \geq \frac{p_3}{w_3} = \frac{3}{2} \geq \frac{p_4}{w_4} = \frac{4}{3} \geq \frac{p_5}{w_5} = \frac{3}{3} \geq \frac{p_6}{w_6} = \frac{5}{6} \geq \frac{p_7}{w_7} = \frac{4}{5}$

Figure 1: (a) A conflict graph G with $n = 7$ vertices (items) and $m = 5$ edges (conflicts); the numbering corresponds to the indices of the vertex set. (b) The profit and weight of each vertex are reported separated by a comma; the independent set $I = \{1, 3, 7\}$ (whose vertices are highlighted in red) is an optimal KPCG solution, of value $KPCG(V, 8) = 10$ and total weight equal to 8.

1.3. Contributions and outline of the paper

In this work, we propose a novel combinatorial branch-and-bound algorithm for solving large instances of the KPCG which employs an n-ary branching scheme based on the "branching and pruned set" approach. Our method effectively combines the two main aspects of the problem, its capacity (or KP) aspect and its conflict (or MWISP) aspect, by relying on different bounding procedures for pruning the individual branch-and-bound nodes as well as for reducing the number of child nodes that are generated during the branching phase.

The paper is structured as follows. Section 2 summarizes previous works on the KPCG and on closely related problems. Section 3 introduces three Integer Linear Programming (ILP) formulations for the KPCG. The new branch-and-bound algorithm that we propose in this paper is presented in Section 4. The bounding procedures used to reduce the size of the branching tree are described in Section 5. The implementation details and an illustration of the execution of the algorithm are reported in Section 6. The results of an extensive set of experiments carried out to compare our algorithm to the state-of-the-art approaches and to assess the relevance of its components are reported in Section 7. Finally, Section 8 draws some concluding remarks and highlights directions for future works.

2. Previous and related works

The KPCG is originally introduced in [53], where the authors propose a local-search and a branch-and-bound method, the latter based on the Lagrangian relaxation obtained by dualizing the conflict constraints. In [21], a method is proposed which combines a primal heuristic with a fixing procedure based on the lower bound provided by the former to reduce the problem size, after which the (reduced) problem is solved via an ILP-based branch-and-bound algorithm. A reactive-search method is proposed in [20], whereas [1] introduces a heuristic based on an integer programming formulation, which also relies on the notion of local branching [14]. In[22] and [19], a scatter-search and an iterative rounding search-based heuristic are proposed, respectively. The state-of-the-art exact algorithm for solving large instances of the KPCG is presented in [4]. It consists of a branch-and-bound algorithm based on binary branching, relying on a fathoming procedure which combines a bounding technique often used for solving the MWISP, see, e.g, [18], with the KP aspect of the KPCG.

Some work has been carried out for the case where the graph G has a special topology. In particular, [36] presents pseudopolynomial-time algorithms capable of producing exact solutions when G is either a chordal graph or a graph with bounded tree width, and illustrates how to extend such algorithms to obtain fully polynomial-time approximation schemes. The same paper shows, in contrast to the MWISP, which is polynomial-time solvable on perfect graphs [17], that the KPCG is strongly $\mathcal{NP}$-hard when G is perfect. This property does not hold for every perfect graph though, since, as shown in [40], the KPCG is weakly $\mathcal{NP}$-hard when G is an interval graph.

As we mentioned, the KPCG arises as pricing subproblem when solving with branch-and-price algorithms the BPP and the BPPC (adopting, for the former, a Ryan-Forster branching scheme). In this context, [13] proposes a two-step algorithm which first applies a greedy heuristic and then, if no column with negative reduced cost has been found, resorts to solving the KPCG with a state-of-the-art ILP solver. In [11], the KPCG is tackled by solving it directly using an ILP formulation. In [40], an exact approach based on a depth-first branch-and-bound algorithm is proposed and, for the case where G is an interval graph, a pseudopolynomial-time algorithm based on dynamic programming. In [51], an exact (exponential-time in the worst case) label-setting algorithm is proposed which relies on fathoming and dominance rules to reduce the search space. To the best of our knowledge, the latter constitutes the state of the art when tackling the special type of KPCG instances which arise when solving the BPP/BPPC with a branch-and-price method. As the authors of [51] state, their algorithm does not compete with the one proposed in [4] when tested on the (harder) instances used in the latter work.

Notwithstanding the large amount of works on the KP, see [33] for the state-of-the-art method, as well as the surveys [27, 32], such works are highly problem-specific and often difficult to extend to more general problems. Differently, many of the works on the MWISP or on the *Maximum Weighted Clique Problem* (MWCP), which is obtained from the MWISP by complementing G, have a general enough structure that lends itself well to extensions. For surveys on these problems, we refer the reader to [52]. While many approaches based on mathematical programming are known for solving the MWISP/MWCP, see [35, 6, 39, 2, 38, 15, 3, 7, 16, 8], the most efficient methods known in the literature for these problems and their variants are based on combinatorial branch and bound, see [50, 42, 18, 34, 23, 48, 46, 47]. In particular, for large instances, [47] and [46] are the state-of-the-art exact algorithms for the cases with weights on the vertices and on the edges, respectively.

3. Integer Linear Programming Formulations for the KPCG

Before introducing our novel branch-and-bound algorithm, we present three Integer Linear Programming (ILP) formulations for the KPCG. We will compare them to our algorithm from a computational standpoint in Section 7.

Letting, for each item $i \in V$, the variable $x_i \in \{0, 1\}$ take value 1 if and only if vertex i is part of the solution, a natural ILP formulation for the KPCG reads as follows:

$$\max_{x \in \{0,1\}^{|V|}} \sum_{i \in V} p_i x_i \tag{2}$$

$$\sum_{i \in V} w_i x_i \le c \tag{3}$$

$$x_i + x_j \le 1 \qquad \{i, j\} \in E. \tag{4}$$

The objective function (2) corresponds to the total profit of the chosen items. Constraint (3) imposes that the total item weight be no larger than the knapsack capacity c. Constraints (4) impose that, from each edge $\{i, j\} \in E$ of the conflict graph G, at most one vertex among i and j be selected. We refer to this formulation as ILP_1.

A reformulation, originally introduced in [9] for the MWCP, featuring fewer constraints than ILP_1 can be obtained by replacing Constraints (4), whose cardinality is $m = O(n^2)$, by the following constraints, whose cardinality is only n:

$$\sum_{j \in N(i)} x_j \le |N(i)|(1 - x_i), \qquad i \in V. \tag{5}$$

Constraints (5) prevent any vertex in the neighborhood of a vertex $i \in V$ from being selected if i is contained in the solution (this has to be the case since all vertices in $N(i)$ are in conflict with i). We refer to this formulation as ILP_2. Note that, while featuring fewer constraints than ILP_1, the Linear Programming (LP) relaxation of ILP_2 is no tighter than that of ILP_1. This is because Constraints (5) can be obtained by linear combination of Constraints (4) (as, for each $i \in V$, the constraint among Constraints (5) corresponding to i is obtained by linearly combining with unit weights the $N(i)$ constraints in (4) corresponding to all edges $\{i, j\} \in \delta(i)$).

A third ILP formulation, originally proposed in [4], considers a collection $\tilde{\mathscr{C}}(V)$ of cliques of G covering all the edges in E (i.e., such that, for each edge $\{i, j\}$ of E, both i and j belong to some clique $C \in \tilde{\mathscr{C}}(V)$). We refer to this formulation as ILP_3. To obtain it, it suffices to replace Constraints (4) by:

$$\sum_{i \in C} x_i \leq 1 \qquad\qquad C \in \tilde{\mathscr{C}}(V). \tag{6}$$

Constraints (6) impose that no more than a single vertex be selected from each clique $C \in \tilde{\mathscr{C}}(V)$. The LP relaxation of ILP_3 is at least as tight as that of ILP_1 as, for each of Constraints (4) in ILP_1, ILP_3 contains a constraint among those in (6) which features more variables in the left-hand side while having the same right-hand side, thus being at least as tight. Different heuristic procedures can be used for creating $\tilde{\mathscr{C}}(V)$, among which the one proposed in [4].

4. A new combinatorial branch-and-bound algorithm

In this section, we describe our new combinatorial branch-and-bound algorithm for solving the KPCG. The algorithm is based on an n-ary branching scheme relying on the notion of "branching and pruned sets", which we define in what follows. In particular, we rely on several effective bounding techniques to reduce the size of the branching tree by pruning its nodes.

The nodes of the branch-and-bound tree are obtained by adding to the current solution one item at a time in a recursive fashion, avoiding any conflicting items and guaranteeing that the capacity constraint be satisfied. Given a node of the tree, let $\hat{I} \subseteq V$ be the independent set corresponding to the partial solution associated with it. Let also LB be the value of the incumbent solution I_{inc}. If, at any node, $p(\hat{I}) > LB$ holds, both I_{inc} and LB are updated.

We denote by $\hat{V}$ the set of vertices not in $\hat{I}$ with the property that, if any vertex $i \in \hat{V}$ is individually added to $\hat{I}$, the resulting set $\hat{I} \cup \{i\}$ remains an independent set. Formally, $\hat{V}$ corresponds to the intersection of the antineighborhoods of the vertices in $\hat{I}$, i.e.:

$$\hat{V} := \bigcap_{i \in \hat{I}} \overline{N}(i). \tag{7}$$

Notice that, by definition, $\hat{V}$ contains all vertices which are potential candidates for branching. In particular, $\hat{V}$ is used to derive the branching and pruned sets, as we will describe later. Throughout our algorithm, the elements of $\hat{V}$ preserve the order introduced in (1) for V. With each set $\hat{V}$, we associate a *residual capacity* defined as:

$$c(\hat{V}) := c - w(\hat{I}). \tag{8}$$

Such quantity corresponds to the knapsack capacity that is still available for selecting vertices from $\hat{V}$ after those in $\hat{I}$ have been chosen. At each node of the branching tree, we update the definition of $\hat{V}$ by removing from it any vertex $i \in \hat{V}$ with weight larger than the residual capacity (i.e., with $w_i > c(\hat{V})$).

Our branching scheme relies on partitioning the vertex set $\hat{V}$ into two subsets: the *branching set* B and the *pruned set* P. [1] The overarching idea is to determine a set $P \subseteq \hat{V}$ of vertices such that the incumbent solution cannot be improved by adding any feasible subset of P to it. For a given set P, the set B is defined as P's complement, i.e., $B := \hat{V} \setminus P$. Since at least one item from B is necessary to improve the incumbent solution, we can avoid branching on any of the vertices in P at the current node of the tree. As a consequence, the larger P the smaller the number of child nodes of a given branch-and-bound node.

After P and, consequently, B, have been determined, we carry out a $|B|$-ary branching operation, creating a tree node per vertex $i \in B$ by adding i to $\hat{I}$. The effective procedure we adopt to construct a "small" branching set B and, therefore, a "large" pruned set P is described in the next subsection.

4.1. Constructing the branching and pruned sets B and P: the PARTITION *procedure*

In principle, the largest pruned set P and the corresponding smallest branching set B can be obtained by solving the following problem

$$P := \arg\max_{\bar{P} \subseteq \hat{V}} \left\{ |\bar{P}| : LB - p(\hat{I}) \geq KPCG(\bar{P}, c(\hat{V})) \right\} \quad \text{and} \quad B := \hat{V} \setminus P \tag{9}$$

which, unfortunately, can be computationally very hard. [2] Aiming at an efficient branch-and-bound method, it is therefore crucial to design an efficient heuristic procedure for constructing the set P.

The procedure we propose in this paper, which we refer to as `PARTITION`, is constructive and relies on an efficient method for computing an upper bound $UB(P)$ on $KPCG(P, c(\hat{V}))$ and for updating it alongside P. In particular, the procedure operates in a *greedy* fashion, maintaining the following invariant:

$$LB - p(\hat{I}) \geq UB(P), \tag{10}$$

which, as it is clear, implies $LB - p(\hat{I}) \geq KPCG(P, c(\hat{V}))$. For each node of the branch-and-bound tree, `PARTITION` starts by letting $B := \hat{V}$ and $P := \emptyset$. It then, iteratively, takes every vertex $j \in B$ into account and checks if, by adding j to P (and removing it from B), the following condition is satisfied:

$$LB - p(\hat{I}) \geq UB(P \cup \{j\}).$$

If this is the case, we let $P := P \cup \{j\}$. If not, j remains in B and the next vertex in B is

[1]Similar branching schemes are used in the majority of the most effective combinatorial branch-and-bound algorithms for the MWISP/MWCP and their variants, see, e.g., [25, 28, 29, 30, 41, 42, 43, 44, 45, 47].

[2]Indeed, the problem is a $\Sigma_2^{\mathcal{P}}$-hard bilevel programming problem. This can be proved by following an argument similar to the one used in [46] in the context of the maximum edge-weighted clique problem.

examined. The procedure stops when all vertices in $\hat{V}$ have been examined.

The upper bound $UB(P)$ we rely on for constructing P is similar to those often used in branch-and-bound algorithms for the MWISP/MWCP, see, e.g, [18, 29, 47]. It is obtained by considering a relaxation of the formulation ILP_3 with vertex set P which features only a subset $\mathscr{P}(P)$ of the collection of cliques in $G[P]$ forming a partition of the vertices, dropping the capacity constraint in (3) as well as any integrality constraints on the variables. The bound $UB(P)$ is provided by the following primal-dual pair of LPs:

$$\max_{x \geq 0} \left\{ \sum_{i \in P} p_i \, x_i : \sum_{i \in C} x_i \leq 1, \;\; C \in \mathscr{P}(P) \right\} = \min_{\pi \geq 0} \left\{ \sum_{C \in \mathscr{P}(P)} \pi_C : \pi_{C(i)} \geq p_i, \;\; i \in P \right\},$$

where, for all $i \in P$, $C(i)$ is the unique clique in $\mathscr{P}(P)$ containing item i. The unique optimal solution to the dual is obtained by letting $\pi^*_C := \max_{i \in C}\{p_i\}$, which leads us to the following *clique partition* upper bound:

$$UB_C(P) := \sum_{C \in \mathscr{P}(P)} \pi^*_C. \tag{11}$$

We remark that, differently from our algorithm, the works in [18, 29, 47] consider a clique cover of P, rather than a partition. While employing a cover could result in a tighter upper bound, it would come at a higher computational cost, as it would require the generation of a (much) larger number of cliques and it would not allow for a closed-form computation of the bound.

In our algorithm, the `PARTITION` procedure constructs $\mathscr{P}(P)$ iteratively alongside the corresponding upper bound $UB_C(P)$. When it halts, P corresponds to the union of the vertices of the cliques that have been created. The procedure, which is inspired by the greedy sequential independent-set coloring algorithm proposed in [42] for the MWCP with unit weights, takes as input the set $\hat{V}$, the solution value LB of the incumbent, and the total profit $p(\hat{I})$ of the solution $\hat{I}$ that corresponds to the current branch-and-bound node. Key to `PARTITION` is the notion of budget, defined as:

$$\texttt{budget} := LB - p(\hat{I}) - UB_C(P), \tag{12}$$

which we use for checking whether an item $i \in B$ can be added to P without violating the invariant (10).

The `PARTITION` procedure builds the cliques in $\mathscr{P}(P)$ one at a time. When building clique C (starting from the empty set), it examines each vertex $i \in B$ and checks whether $C \cup \{i\}$ is a clique. If this is the case, it further tests whether, with the value of π^*_C that would be obtained by adding i to C, the value of `budget` would be nonnegative. If both conditions hold, vertex i is removed from B and added to C and P. If not, vertex i remains in B and the next vertex is examined. The procedure halts when either (i) no additional vertices in B can be added to P without resulting in a negative `budget` (which implies that no more vertices can be added to C without violating the invariant) or (ii) $P = \hat{V}$. Whenever the procedure finds that adding a vertex $i \in B$ to C would result in a negative `budget`, i is discarded as, by construction of the algorithm, such vertex would lead to a negative `budget` even if it were added to one of the cliques the procedure is yet to construct.

The `PARTITION` procedure considers the vertices in $\hat{V}$ in reverse profit-over-weight order (i.e., in reverse order w.r.t. the order in (1)). This way, P is more likely to contain vertices with low profit-over-weight ratio, leaving those with a higher ratio in the set B. As we branch by adding the vertices in B to the current solution $\hat{I}$ one at a time, with this choice we are more likely to obtain solutions with high profit at the earlier stages of the algorithm.

5. Bounding procedures for pruning individual nodes

In our algorithm, we consider different bounds for pruning individual nodes of the branch-and-bound tree. They are of two families. The first family is based on the KP relaxation of the KPCG that is obtained by dropping the conflict constraints. The second family relies on the *Multiple-Choice Knapsack Problem* (MCKP), a special case of the KPCG in which the conflict graph is a collection of pairwise disjoint cliques. [3] The MCKP has been widely studied in the literature, see, e.g., [37, 49], and Ch. 11 of [27].

We remark that, in spite of the KPCG being strongly $\mathcal{NP}$-hard, all the bounds we consider correspond to weakly $\mathcal{NP}$-hard problems (the KP and the MCKP) and to relaxations of either of them which are computable in polynomial time. As better explained in the following, in our algorithm we compute the polynomial-time bounds at each branch-and-bound node, while we compute the weakly $\mathcal{NP}$-hard ones only once (at the root node) in such a way that they are valid in every node of the branch-and-bound tree, and store them in lookup-tables (see Sections 5.1.1 and 5.2.1).

All the upper bounds that we introduce in the following are valid for $KPCG(\hat{V}, c(\hat{V}))$ (the instance of the KPCG corresponding to the current branch-and-bound node with vertex set $\hat{V}$ and residual capacity $c(\hat{V})$). A branch-and-bound node with vertex set $\hat{V}$ can be pruned whenever the upper bound on $KPCG(\hat{V}, c(\hat{V}))$ is less or equal than $LB - p(\hat{I})$ as, if $KPCG(\hat{V}, c(\hat{V})) \leq LB - p(\hat{I})$, no subset of vertices in $\hat{V}$ suffices to create a feasible solution better than the incumbent.

We introduce the following notation, which will be used in the remainder. For a given set $\hat{V}$ of cardinality $\ell := |\hat{V}| \leq n$ and with items respecting the order in (1), let α be a function which, given the position $j \in \{1, \ldots, \ell\}$ of an item in $\hat{V}$, returns its index $\alpha_j(\hat{V})$ in V. An example of $\hat{V}$ and its mapping is provided in Figure 4. Given $\hat{V}$, we define $\check{V}$ as the superset of $\hat{V}$ obtained by considering all the items in V of index, in V, greater than or equal to $\alpha_1(\hat{V})$ (which corresponds to the first item in $\hat{V}$). Notice that there are as many sets $\check{V}$ as the number of items n, equal to $\{j, \ldots, n\}$ for all $j \in V$.

5.1. KP-based upper bounds

Consider the KP relaxation of the KPCG obtained by ignoring the conflict graph, namely the following upper bound:

$$UB_{KP}(\hat{V}) := \max_{x \in \{0,1\}^{|\hat{V}|}} \Big\{ \sum_{i \in \hat{V}} p_i \, x_i : \sum_{i \in \hat{V}} w_i \, x_i \leq c(\hat{V}) \Big\}. \tag{13}$$

[3]Notice that, while, traditionally, the MCKP asks for selecting exactly one item per clique, the problem asking for selecting *at most* an item per clique can be reduced to the traditional one by adding a dummy item to each clique with 0 profit and 0 weight.

At each branch-and-bound node, to avoid computing the bound (13) for the associated vertex set $\hat{V}$ (which would be too time consuming), we consider two strategies that are in line the previous works [4, 40].

5.1.1. *KP-based lookup-table upper bound*

The first strategy derives an upper bound on (13) by considering a further relaxation of the associated KP problem where, rather than $\hat{V}$, its superset $\check{V}$ is considered. The upper bounds $UB_{KP}(\check{V})$ of (13) associated with all possible sets $\check{V}$ (i.e., with all possible sets $\{j, \dots, n\}$, $j \in V$) and all possible capacity values $s = \{0, \dots, c\}$ can be precomputed at the root node of the branch-and-bound tree in $O(n\,c)$ via the classical Dynamic Programming (DP) algorithm for the KP, see, e.g., [32], and stored in a lookup-table. The DP algorithm relies on the following recursive equation:

$$f(j, s) := \max\{f(j + 1, s), f(j + 1, s - w_j) + p_j\},$$

where $f(j, s)$ is the optimal solution value of a restriction of the problem to the items in $\{j, \dots, n\}$ and a capacity of $s \le c$. The recursion starts with $j = n$ and proceeds backwards (the next item of the recursion being $j = n - 2$). The function $f(j, s)$ returns value 0 if $j = n + 1$ for all $s \ge 0$ and it returns $-\infty$ if $s < 0$. By accessing the lookup-table in position $j = \alpha_1(\hat{V})$ and $s = c(\hat{V})$, we can recover the following *KP-based lookup-table upper bound* in $O(1)$:

$$UB_{L_1}(\hat{V}) := f(\alpha_1(\hat{V}), c(\hat{V})). \tag{14}$$

5.1.2. *Martello-Toth upper bound*

The second strategy considers the vertex set $\hat{V}$ and it relaxes the integrality of the variables. For this purpose, we rely on the classical *Martello-Toth upper bound* valid for the KP [31], which improves over the *Dantzig upper bound* (see, e.g., [32]). Let t be the position of the *critical item* in $\hat{V}$, i.e., the position in $\{1, \dots, |\hat{V}|\}$ taken by the first (w.r.t. the order in (1)) item in $\hat{V}$ with the property that the total weight of the items in the set $\{\alpha_1(\hat{V}), \dots, \alpha_t(\hat{V})\}$ exceeds the residual capacity $c(\hat{V})$ while the total weight of those in the set $\{\alpha_1(\hat{V}), \dots, \alpha_{t-1}(\hat{V})\}$ does not. Let $\bar{c}(\hat{V})$ be the difference between the residual capacity $c(\hat{V})$ and the capacity consumed by all the items preceding the critical one, namely: $\bar{c}(\hat{V}) := c(\hat{V}) - \sum_{j=1}^{t-1} w_{\alpha_j(\hat{V})}$. The Martello-Toth bound reads:

$$UB_{MT}(\hat{V}) := \max\left\{UB_0(\hat{V}), UB_1(\hat{V})\right\}, \tag{15}$$

where

$$UB_0(\hat{V}) := \sum_{j=1}^{t-1} p_{\alpha_j(\hat{V})} + \left\lfloor \bar{c}(\hat{V}) \frac{p_{t+1}}{w_{t+1}} \right\rfloor \quad \text{and} \quad UB_1(\hat{V}) := \sum_{j=1}^{t-1} p_{\alpha_j(\hat{V})} + \left\lfloor p_t - (w_t - \bar{c}(\hat{V})) \frac{p_{t-1}}{w_{t-1}} \right\rfloor.$$

$UB_0(\hat{V})$ is an upper bound obtained when the critical item t (we assume $t > 1$ and $t < |\hat{V}|$) is not part of the solution, whereas $UB_1(\hat{V})$ is an upper bound for the case where it is. The upper bound $UB_{MT}(\hat{V})$ can be computed very efficiently, in $O(|\hat{V}|)$ time.

We remark that, while $UB_{L_1}(\check{V})$ is defined for a superset of $\hat{V}$ and retains the integrality of the variables, $UB_{MT}(\hat{V})$ is defined for $\hat{V}$ and considers continuous variables. Therefore, the two bounds do not dominate each other.

5.2. MCKP-based bounds

Let $\mathscr{P}(\hat{V})$ be a clique partition of $\hat{V}$. We consider the following MCKP relaxation of the KPCG which provides the following upper bound:

$$UB_{MCKP}(\hat{V}) := \max_{x \in \{0,1\}^{|\hat{V}|}} \left\{ \sum_{i \in \hat{V}} p_i\, x_i : \sum_{i \in \hat{V}} w_i\, x_i \leq c(\hat{V}),\ \sum_{i \in C} x_i \leq 1,\ C \in \mathscr{P}(\hat{V}) \right\}. \tag{16}$$

As solving the MCKP in (16) at each branch-and-bound node (associated with the corresponding vertex set $\hat{V}$) can be computationally expensive, we consider different relaxations. Before resorting to them, though, we check whether the problem in (16) can be solved in closed form the set $\hat{V}$ at hand.

Let, for each $C \in \mathscr{P}(\hat{V})$, $i(C) := \arg\max_{i \in C}\{p_i\}$ be the index of an item of maximum profit in C (breaking ties by smallest weight). If $\{i(C)\}_{C \in \mathscr{P}(\hat{V})}$ is a feasible solution, i.e., if it satisfies the capacity constraint, the optimal value of the MCKP instance in (16) is $\sum_{C \in \mathscr{P}(\hat{V})} p_{i(C)}$. If the condition is not satisfied, we resort to two upper bounds which we now introduce.

5.2.1. MCKP-based lookup-table bound

Consider the superset $\check{V} = \{j, \ldots, n\}$ of $\hat{V}$ as done in Section 5.1.1, and a clique partition $\mathscr{P}(\check{V})$ of $\check{V}$. The upper bound $UB_{MCKP}(\check{V})$ on (16) associated with all sets $\check{V}$ and all possible capacity values $s \in \{0, \ldots, c\}$ can be computed in $O(n\,c)$ via a well-known DP algorithm for the MCKP, see, e.g., [36]. The algorithm is based on the following recursive equation:

$$g_j(s, \ell) := \max\left\{ \max_{i \in C_\ell : w_i \leq s} \{g_j(s - w_i, \ell - 1) + p_i\}, g_j(s, \ell - 1) \right\}, \tag{17}$$

where $g_j(s, \ell)$ is the optimal solution value of the restriction of the MCKP obtained by considering the first ℓ cliques ($\ell \leq |\mathscr{P}(\check{V})|$), their items, and a capacity of $s \leq c$. The recursion starts with $\ell = 1$ and proceeds onwards (the next clique of the recursion being $\ell + 1$). The function $g_j(s, \ell)$ returns 0 if $\ell = 0$ for all $s \geq 0$.

Differently from the recursion of Section 5.1.1, where the lookup-table can be entirely constructed in $O(n\,c)$ by simply examining the items in reverse order of $\check{V}$, in this case we need to run the DP algorithm for the MCKP n times, once for each item $j \in V$. This results in an $O(n^2\,c)$ algorithm. Thus, we obtain the following *MCKP-based lookup-table upper bound*:

$$UB_{L_2}(\hat{V}) := g_{\alpha_1(\hat{V})}(c(\hat{V}), h), \tag{18}$$

where $h := |\mathscr{P}(\check{V})|$, i.e., the number of cliques of the clique partitioning $\mathscr{P}(\check{V})$ which is uniquely determined by $\alpha_1(\hat{V})$. For this reason, as for $UB_{L_1}(\hat{V})$, $UB_{L_2}(\hat{V})$ only depends on $\alpha_1(\hat{V})$ and on $c(\hat{V})$. An example of the construction of this lookup-table is given in Section 6. By nature, since $UB_{L_2}(\hat{V})$ considers the internal conflicts in each of the cliques which in $UB_{L_1}(\hat{V})$ are entirely ignored, it is stronger than the latter for any clique partition.

5.2.2. *LP-relaxation-based upper bound*

Consider the dual of the LP relaxation of the MCKP as defined in (16):

$$UB_{MCKP-LP}(\hat{V}) := \min_{\pi,\beta\geq 0}\left\{\sum_{C\in\mathscr{P}(\hat{V})}\pi_C + c(\hat{V})\,\beta:\ \ \pi_{C(i)} + w_i\,\beta \geq p_i,\ \ i\in\hat{V}\right\}, \tag{19}$$

where the dual variables π and β are associated with the primal constraints of the MCKP. This dual problem asks for covering the weight p_i of each item $i \in \hat{V}$ by either the variable $\pi_{C(i)}$ associated with the unique clique $C(i)$ containing i or by the variable β or by both. Intuitively, β acts as a "super" clique covering each item proportionally to its weight but having an objective function cost which, rather than unitary as for the π variables, is equal to the capacity $c(\hat{V})$.

Algorithms for solving either the dual problem (19) or its primal are known in the literature. They rely on the notion of *slope*. Considering the items in each clique $C \in \mathscr{P}(\hat{V})$ by nondecreasing weight and ignoring any dominated items (see [27] for further details), a slope is associated with each pair of nondominated items j and $j-1$ in each clique C and is defined as follows:

$$s_j := \frac{p_j - p_{j-1}}{w_j - w_{j-1}},$$

with $s_j = \frac{p_j}{w_j}$ if j is the first item of the clique.

In what follows, we sketch the algorithm to solve the primal problem which has inspired the new bound that we propose in this section. The key idea, also see [27], is to start with an empty solution and create a set of metaitems, one per slope. These metaitems have as profit the difference of two consecutive items of a clique, and as weight the difference of the weights. The algorithm then follows the standard Dantzig algorithm for the KP problem (see [32]), inserting these metaitems in the knapsack in order of nonincreasing profit-over-weight ratio and halting when the capacity is fully saturated. The effect of taking a metaitem is that, by definition of its profit and weight, a swap is (implicitly) performed in the solution to the original problem (defined over items, rather than metaitems), according to which the $j-1$-th item is replaced with the j-th of the corresponding slope. The algorithm runs in $O(|\hat{V}|\log|\hat{V}|)$, since it requires to sort all the items as well as all the slopes.

Consider now Figure 2, where we report the slopes associated with the clique partition $C_1 = \{1,2\}$, $C_2 = \{3,4\}$, $C_3 = \{5,6\}$, and $C_4 = \{7\}$ of the instance reported in Figure 1. In C_1, there is a unique metaitem associated with the slope $s_1 = \frac{3}{1}$ (since item 2 is dominated). In C_2, there are two metaitems associated with the slopes $s_3 = \frac{3}{2}$ and $s_4 = \frac{4-3}{3-2}$. In C_3, there are two metaitems associated with the slopes $s_5 = \frac{3}{3}$ and $s_6 = \frac{5-3}{6-3}$. Finally, in C_4 there is a single metaitem associated with the slope $s_7 = \frac{4}{5}$ (which is not depicted in Figure 2 as C_4 is a singleton).

After sorting, the Dantzig algorithm attempts to insert the metaitems in the knapsack in the following order: $s_1 \prec s_3 \prec s_4 \prec s_5 \prec s_7 \prec s_6$. It halts after examining the metaitem with slope s_7, by which the capacity of 8 units is fully saturated, adding $\frac{1}{5}$ of this metaitem to the solution. It then determines the optimal value $\beta^* = \frac{4}{5}$ as the slope associated with the metaitem 7, which leads to the following value of the π variables: $\pi^*_{C_1} = \frac{11}{5}$, $\pi^*_{C_2} = \frac{8}{5}$, $\pi^*_{C_3} = \frac{3}{5}$, and $\pi^*_{C_4} = 0$. The primal solution is $x^*_1 = 1$, $x^*_4 = 1$, $x^*_5 = 1$, and $x^*_7 = \frac{1}{5}$, with all the other variables set to 0. The optimal solution value is $\frac{54}{5}$.

In the literature, algorithms such as [10, 54] are known for solving the dual (19) in $O(|\hat{V}|)$ by determining the optimal value β^* of β (which can be shown to be equal to the slope of the metaitem that is taken fractionally by the Dantzig algorithm). Such value is then used to compute the optimal values π^* of π in closed form. These algorithms, however, can be time consuming in practice, especially if used at every node of the branch-and-bound tree—see Section 7, where we report the computational results obtained with a version of our algorithm modified to use the state-of-the-art implementation of the dual algorithm described in [37].

To reduce the computational effort, we propose a greedy heuristic which constructs a good-quality dual solution very efficiently. The algorithm is inspired by the exact primal algorithm we just illustrated but, differently from it, it does not require any sorting nor any pre-processing phase in which dominated items are removed.

Specifically, our heuristic computes a (potentially suboptimal) value $\bar{\beta}$ for β defined as the ratio $p_{\bar{j}}/w_{\bar{j}}$ of what we call the *MCKP-critical item* $\bar{j} \in \hat{V}$. This item is obtained as follows. The procedure keeps track of a *clique profit* $\bar{p}(C)$ for each clique in $\mathscr{P}(\hat{V})$ as well as of a *total residual capacity* $\bar{w}$. Initially, the clique profits are all set to zero and the total residual capacity $\bar{w}$ is set to $c(\hat{V})$. Next, we iterate over the items in $\hat{V}$ according to the order in (1). Let $j \in \hat{V}$ be the current item under consideration, and let $C(j)$ be the (unique) clique containing it. If $p_j \leq \bar{p}(C(j))$, item j is discarded and the algorithm proceeds to examine the next item. Otherwise, if $w_j \leq \bar{w}$, the clique profit $\bar{p}(C(j))$ is set to p_j and the total residual capacity $\bar{w}$ is updated (reduced) as follows:

$$\bar{w} := \bar{w} - \big(p_j - \bar{p}(C(j))\big)\,\frac{w_j}{p_j}.$$

Note that the reduction corresponds to the fraction of the item weight which is necessary to cover the difference in profit between p_j and $\bar{p}(C(j))$. If $w_j > \bar{w}$, i.e., if item j exceeds the residual capacity, the algorithm halts and item j becomes the MCKP-critical item. Note that such item always exists since, if the MCKP relaxation as defined in (16) admits an optimal solution in which the capacity is not saturated, we always solve it to optimality in closed form as described in Section 5.2.

Once the value $\bar{\beta}$ is computed, and in order to obtain a complete feasible dual solution, we determine the values $\bar{\pi}$ of the π variables as follows:

$$\bar{\pi}_C := \max\left\{0, \max_{i\in C}\left\{p_i - \bar{\beta}\, w_i\right\}\right\} \qquad C \in \mathscr{P}(\hat{V}). \tag{20}$$

This leads us to the following *partition-based upper bound*:

$$UB_p(\hat{V}) := \left\lfloor \sum_{C\in\mathscr{P}(\hat{V})} \bar{\pi}_C + \bar{\beta}\; c(\hat{V}) \right\rfloor. \tag{21}$$

While the computation of this bound requires the same time, $O(\hat{V})$, as the dual algorithms, it is extremely faster in practice and it produces very tight upper bounds for the instances we consider (see Section 7).

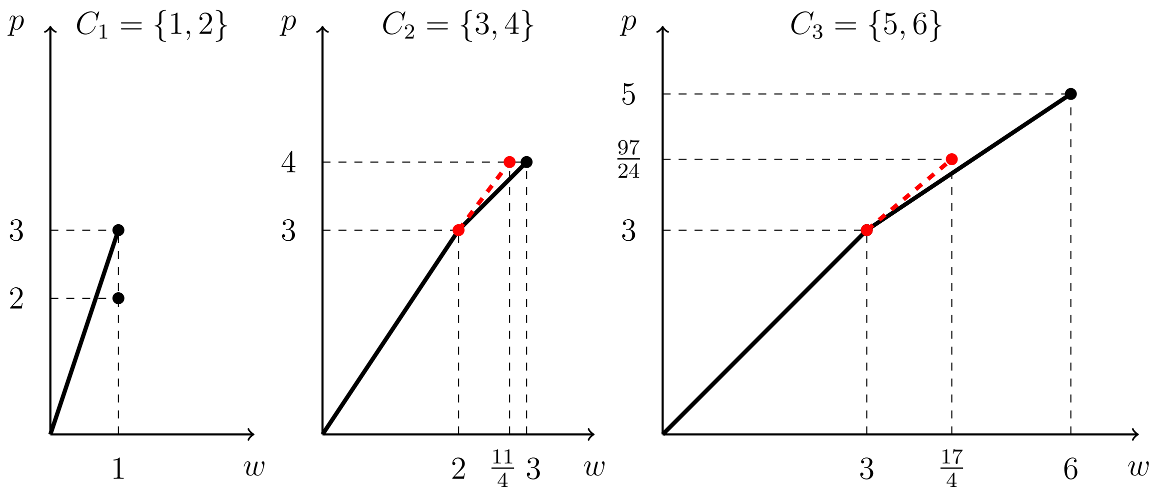


Figure 2: Demonstration of our greedy bounding procedure on the example of Figure 1.

We now demonstrate the way our new bound (21) is computed using the same example depicted in Figure 1. Initially, the clique profits are set to 0 and the total residual capacity $\bar{w}$ is set to 8 units. The procedure examines the items according to the order in (1), so that item 1 is selected first, resulting in a clique profit $\bar{p}(C_1) = 3$ and in a reduction of the total residual capacity by w_1, i.e., $\bar{w} = 8 - 1 = 7$. After item 2 is discarded (as its profit is smaller than p_1), the procedure considers item 3, which results in $\bar{p}(C_2) = 3$ and $\bar{w} = 7 - (3 - 0)\,\frac{2}{3} = 5$. The next item to be processed is item 4, which results in $\bar{p}(C_2) = 4$ and $\bar{w} = 5 - (4 - 3)\,\frac{3}{4} = \frac{17}{4}$. Item 5 is then added, resulting in $\bar{p}(C_3) = 3$ and $\bar{w} = \frac{5}{4}$. At this point, the total residual capacity is strictly smaller than the weight of item 6. Thus, item 6 becomes the MCKP critical item with $\bar{\beta} = \frac{5}{6}$ and $\bar{\pi}_{C_1} = \frac{13}{6}$, $\bar{\pi}_{C_2} = \frac{3}{2}$, $\bar{\pi}_{C_3} = \frac{1}{2}$, and $\bar{\pi}_{C_4} = 0$. The resulting bound is $UB_p(\hat{V}) = \frac{65}{6}$, which is very close to the optimal solution value $\frac{54}{5}$ obtained with the optimal value of β, equal to $\beta^* = \frac{4}{5}$.

Figure 2 illustrates the behavior of our greedy procedure. In the part of the figure that refers to C_1, after items 1 and 2 are considered we can read the values $\bar{p}(C_1) = 3$ and $\bar{w} = 1$. In the central part of the figure concerning C_2, we can read the clique profit $\bar{p}(C_2) = 4$ and infer the total residual capacity $\bar{w} = 7 - \frac{11}{4} = \frac{17}{4}$, which is obtained after examining items 3 and 4. Finally, in the rightmost part we can see that the capacity is fully saturated after taking $\frac{3}{4}$-th of item 6, which leads to a clique profit of $\bar{p}(C_3) = \frac{97}{24}$. Finally, the profit-over-weight ratio of item 6 becomes $\bar{\beta}$.

6. Outline of the algorithm and demonstration

In this section, we provide an outline of our new combinatorial branch-and-bound algorithm, which we refer to as `CFS` (the acronym corresponds to the initials of the last names of the three authors of this work). This algorithm consists of two phases: (*i*) a preprocessing phase in which we compute an initial feasible solution and we reduce the size of the input graph by *pegging* some of the vertices (see Section 6.1); (*ii*) a branch-and-bound procedure described in the previous sections which finds an optimal solution to the problem (see

Section 6.2 for the implementation details).

6.1. Preprocessing phase and heuristic algorithm

Before running the branch-and-bound procedure, `CFS` computes a heuristic solution by resorting to two simple and fast heuristics. The first one is a greedy heuristic which considers the items in nonincreasing order of profit-over-weight ratio and adds them to the solution only if they do not conflict with the previously-added ones and as long as the capacity is not exceeded. We run this heuristic n times, forcing, each time, one of the items in V to be in the solution and storing the best solution found. The second heuristic is a diving heuristic guided by the LP relaxation of ILP_2 (which can be computed very efficiently, see Table 4). Starting from the empty solution, the method iteratively solves the LP relaxation and checks whether the first (in order of profit-over-weight ratio) fractional item can be added to the current solution without violating any constraints. If this is the case, the item is added to the current solution, whereas, if not, it is forced out of it. The method stops when no more fractional items are available, thus finding a feasible heuristic solution.

After two two heuristics have been run, the initial lower bound LB is then set to the value of the best solution found by either of them. Then, we compute the MCKP-based lookup-table described in Section 5.2.1. These bounds are then used in every branch-and-bound node of the tree.

The algorithm performs then an additional phase with the goal of pegging some items, i.e., either eliminating them from the instance or forcing them to be in the solution of the problem. For each item $i \in V$, we consider the following *first pegging condition*:

$$LB \geq UB(N(i), c - w_i) + p_i, \tag{22}$$

where as upper bound we adopt the one which, among the ones we introduced, is the tighest on the problem instance restricted to the items in $N(i)$ and to a capacity of $c - w_i$. If the condition is satisfied, we drop item i from V since no feasible solutions containing it can improve on the incumbent (of value LB). For each item $i \in V$, we also consider the following *second pegging condition*:

$$LB \geq UB(V \setminus \{i\}, c), \tag{23}$$

where as upper bound we adopt the one which is the tightest on the problem instance restricted to the items in $V \setminus \{i\}$ and to a capacity of c. If the condition is satisfied, we drop item i from V and add it to the (partial) root-node solution $\hat{I}$ (since any solution which can improve on the incumbent must contain it).

6.2. Implementation details

During the exploration of the branching tree, `CFS` examines the vertex set $\hat{V}$ induced by the branching operations (see Section 4), attempting to prune its node by relying on the upper bounds proposed in Section 5. More precisely, it first computes the bound $UB_{L_2}(\hat{V})$ (18) (which can be read from the MCKP-based lookup-table). Next, it computes the bound $UB_{MT}(\hat{V})$ (15). If both bounds are insufficient to prune the node, the `PARTITION` procedure is called (ignoring the condition on the `budget` which would halt the procedure if the introduction of any extra nodes were to make it negative) to compute a clique partition of the entire set of vertices $\hat{V}$. Finally, we compute the bound $UB_P(\hat{V})$ (21) based on the obtained partition.

If all these attempts fail, we resort to the branching operations described in Section 4.1, which are based on the branching and pruned sets B and P constructed by the `PARTITION` procedure. The nodes of the branch-and-bound tree are explored in a depth-first fashion and, naturally, the recursion stops whenever the branching set B is empty.

Finally, let us mention that our implementation relies on a very efficient bitstring representation to encode the conflict graph G as well as the sets of vertices $\hat{V}$ and the clique partitions.

6.3. Demonstration of the branch-and-bound algorithm `CFS`

We now illustrate in detail the operations carried out by the `CFS` algorithm on the instance of Figure 1.

6.3.1. Preprocessing and upper bounds

The algorithm starts with an initial lower bound $LB = 9$, which corresponds to the feasible solution $\{1, 3, 5\}$, of total weight 6. We use this solution for illustrative purposes.

Table 1 reports the KP-based lookup-table that is obtained by solving the KP relaxation of the KPCG by dynamic programming (which is used to compute the bound UB_{L_1} defined in (14) of Section 5.1). For each pair (j, s), the table contains the optimal solution value of the KP relaxation restricted to the items in $\{j, \ldots, 7\}$, for all values of $j = 7, 6, \ldots, 1$, and to a capacity of $s = 0, \ldots, c$ (the capacity of the instance is 8). Notice that the items are processed in reverse order, starting from the last item of index $n = 7$.

Table 2 contains the MCKP-based lookup-table that is obtained by solving the MCKP relaxation of the KPCG by dynamic programming (which, in the algorithm, is used for computing the bound UB_{L_2} defined in (18) of Section 5.2.1). Let us consider the clique partition

$$\mathscr{P}(V) = \{C_1 = \{1, 2\}, C_2 = \{3, 4\}, C_3 = \{5, 6\}, C_4 = \{7\}\}.$$

For each pair (j, s), the table contains the optimal solution values of the MCKP relaxation restricted to items in $\{j, \ldots, n\}$, for all values of $j = 7, 6, \ldots, 1$, and to a capacity of $s = 0, \ldots, c = 8$. The rows of this lookup-table are filled sequentially according to Equation (17) and by considering the partition of the vertices in $\check{V}$. In Table 3, we show how the row of item 4 in Table 2 is obtained with the partition $\mathscr{P}(\check{V}) = \{\{4\}, \{5, 6\}, \{7\}\}$ from the row of clique $\{7\}$ of Table 3. In Table 3, we report in boldface the bounds which are strictly stronger than the bounds showed in Table 1. It is worth noticing that the bound $UB_{L_2}(V)$ is equal to 10 and that it corresponds to the optimal $KPCG$ solution value.

During the preprocessing phase, the `CFS` algorithm eliminates item 2 thanks to the first pegging condition (22), as $LB = 9 \geq UB_{L_2}(\overline{N}(2)) + p_2 = 7 + 2 = 9$. Indeed, we have $\overline{N}(2) = \{3, 5, 6, 7\}$ (equal to $\hat{V}$), $c(\hat{V}) = c - w_2 = 8 - 1 = 7$, and the bound can be read from the MCKP-based lookup table in row $j = 3$ (which is the first element in $\hat{V}$) and column $s = 7$. Item 6 is eliminated in a similar fashion. None of the other items can be removed by either of the pegging conditions (22), (23).

After the pre-processing phase, $\hat{V} = V \setminus \{2, 6\}$ and the clique partition becomes $\mathscr{P}(\hat{V}) = \{\{1\}, \{3, 4\}, \{5\}, \{7\}\}$. As far as $UB_P(\hat{V})$, as defined in (21) of Section 5.2.2 is concerned, the value of $\bar{\beta}$ is equal to $\frac{4}{5}$, which corresponds to the profit-over-weight ratio of item 7.

Table 1: KP-based lookup-table for the demonstration instance of Figure 1.

	$s = 8$	$s = 7$	$s = 6$	$s = 5$	$s = 4$	$s = 3$	$s = 2$	$s = 1$	$s = 0$
Item 7	4	4	4	4	0	0	0	0	0
Item 6	5	5	5	4	0	0	0	0	0
Item 5	7	5	5	4	3	3	0	0	0
Item 4	8	7	7	4	4	4	0	0	0
Item 3	10	7	7	7	4	4	3	0	0
Item 2	10	9	9	7	6	5	3	2	0
Item 1	12	12	10	9	8	6	5	3	0

Table 2: MCKP-based lookup-table for the demonstration instance of Figure 1.

	$s = 8$	$s = 7$	$s = 6$	$s = 5$	$s = 4$	$s = 3$	$s = 2$	$s = 1$	$s = 0$
Item 7	4	4	4	4	0	0	0	0	0
Item 6	5	5	5	4	0	0	0	0	0
Item 5	7	5	5	4	3	3	0	0	0
Item 4	8	7	7	4	4	4	0	0	0
Item 3	**8**	7	7	**6**	4	4	3	0	0
Item 2	**9**	9	**8**	**6**	6	5	3	2	0
Item 1	**10**	**10**	**9**	**7**	**7**	6	**3**	3	0

Table 3: Optimal solution values (for each possible value of capacity) of the MCKP relaxation for the clique partition $\mathscr{P}(\check{V}) = \{\{4\}, \{5, 6\}, \{7\}\}$ for item 4 and the demonstration instance of Figure 1 .

	$s = 8$	$s = 7$	$s = 6$	$s = 5$	$s = 4$	$s = 3$	$s = 2$	$s = 1$	$s = 0$
Clique $\{4\}$	4	4	4	4	4	4	0	0	0
Clique $\{5, 6\}$	7	7	7	4	4	4	0	0	0
Clique $\{7\}$	8	7	7	4	4	4	0	0	0

Accordingly, the values of the different π variables are $\bar{\pi}_{C_1} = \frac{11}{5}$, $\bar{\pi}_{C_2} = \frac{8}{5}$, $\bar{\pi}_{C_3} = \frac{3}{5}$, and $\bar{\pi}_{C_4} = 0$. The bound $UB_P(\hat{V})$ is $\frac{54}{5}$.

After the preprocessing phase, the `CFS` algorithm starts exploring the nodes of the branch-and-bound tree, which is reported in Figure 3. The `CFS` algorithm explores in a depth-first fashion 8 nodes in total (including the root node). The incumbent solution is updated only at node 3, where the algorithm finds the optimal solution $\{1, 3, 7\}$. Figure 3 reports, for each node of the tree, the sets $\hat{V}$ and $\hat{I}$. If a node is pruned, we report the value of the bound which allowed for pruning it, whereas, if pruning did not take place, we report the branching and pruned sets B and P. In the following section, we describe in detail the operations performed by the `CFS` algorithm on the different nodes of the branching tree.

6.3.2. Processing the root node and the first node

At the root node, $\hat{I}$ is the empty set and $\hat{V}$ corresponds to the initial vertex set V minus the items 2 and 6, which have been removed by the preprocessing procedure. The attempt to prune the node fails since $UB_{L_2}(\hat{V}) = 10$, $UB_{MT}(\hat{V}) = 11$, $UB_P(\hat{V}) = 10$ and all these upper bounds are greater than $LB = 9$. Therefore, the `CFS` algorithm starts

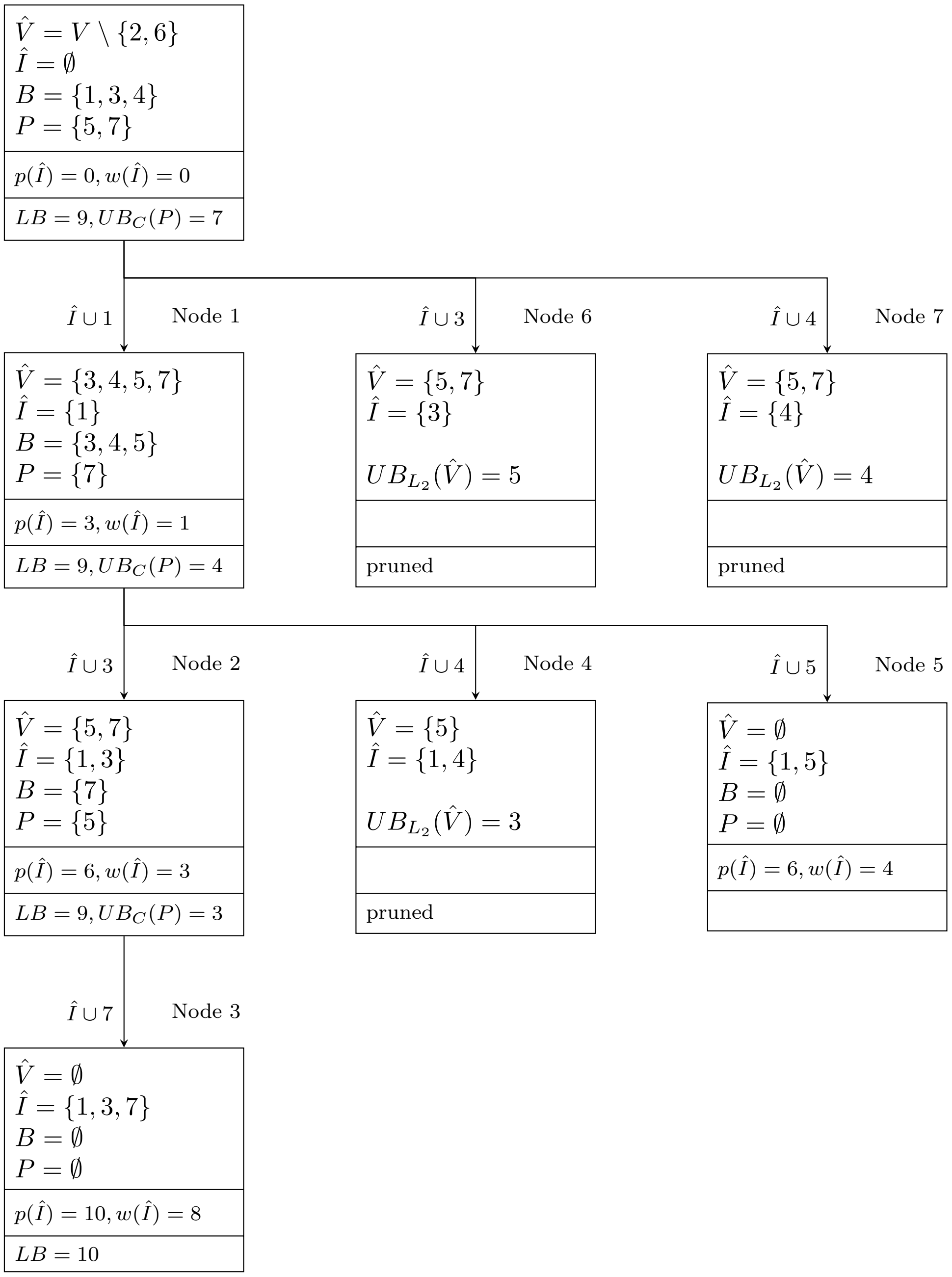


Figure 3: Branch-and-bound tree produced by the CFS algorithm when solving KPCG on the instance reported in Figure 1.

the branch-and-bound phase by creating the branching and pruned sets B and P. The execution of the clique-partitioning algorithm `PARTITION` (see Section 4.1) generates two cliques: $C_1 = \{7\}, C_2 = \{5\}$, which lead to $P := \{5, 7\}$.

The clique-partition bound (as defined in (11)) is $UB_C(P) = \pi^*_{C_1} + \pi^*_{C_2} = 3 + 4 = 7 \leq LB = 9$. It is worth noticing that all the profits p_i corresponding to items not in P are strictly greater than the available `budget`, which is equal to $LB - UB_C(P) = 2$. For this reason, the procedure `PARTITION` cannot further enlarge the set P and it halts. The initial sets P and B are $P = \{5, 7\}$ and $B = \{1, 3, 4\}$. Three nodes are then created by branching on the items contained in B.

First, the algorithm branches on item 1, creating node 1, in which the set $\hat{V}$ is $\{3, 4, 5, 7\}$ and the set $\hat{I}$ is $\{1\}$. The attempt to prune the node fails because $UB_{L_2}(\hat{V}) = 7$, $UB_{MT}(\hat{V}) = 8$, and $UB_p(\hat{V}) = 7$, and they are all greater than $LB - p(\hat{I}) = 9 - 3 = 6$.

The execution of `PARTITION` generates the single clique $C_1 = \{7\}$, which leads to the branching and pruned sets $P = \{7\}$ and $B = \{3, 4, 5\}$ The clique-partition bound is $UB_C(P) = \pi^*_{C_1} = 4 \leq LB = 9$. Since we have `budget` $= LB - UB_C(P) - p(\hat{I}) = 9 - 4 - 3 = 2$ and all the profits of the items in B are strictly greater than `budget`, the procedure halts. At this point, branching is carried out on the items in B, generating 3 additional child nodes.

6.3.3. Processing the remaining nodes

In node 2, the set $\hat{V}$ is $\{5, 7\}$ and the set $\hat{I}$ is $\{1, 3\}$. According to the mapping of the vertices defined in Section 5.1, we have $\alpha_1(\hat{V}) = 5, \alpha_2(\hat{V}) = 7$. Part (a) of Figure 4 reports in green the vertices in $\hat{I}$ and in blue the vertices removed from the instance during the pre-processing phase, whereas part (b) reports in red the vertices of $\hat{V}$. The branching Set B is the singleton $\{7\}$ and the pruned set P is the singleton $\{5\}$.

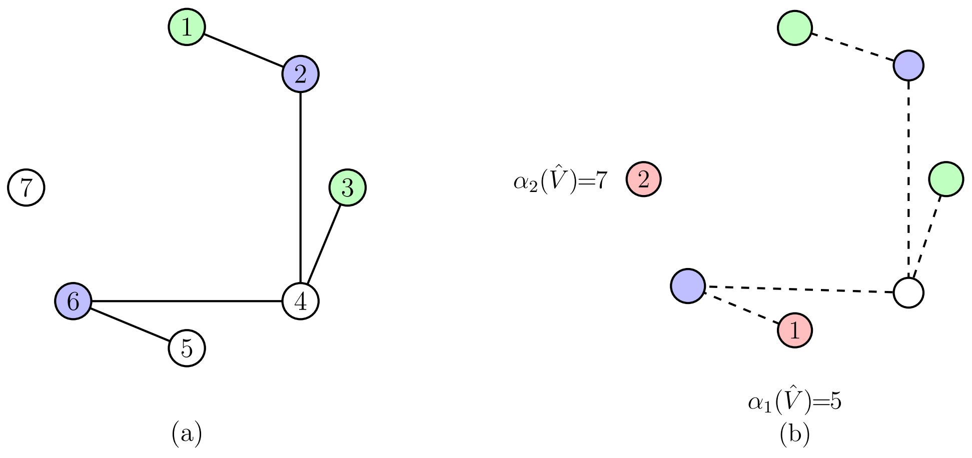


Figure 4: (a) Instance graph with the set $\hat{I} = \{1, 3\}$ of node 2 of the branch-and-bound tree (reported in Figure 3) highlighted in green and the items eliminated during the pre-processing phase highlighted in blue. (b) The graph $G[\hat{V}]$ induced by the vertices in the common anti-neighbourhood of the vertices in $\hat{I}$; the vertices in $\hat{V} = \{5, 7\}$ are highlighted in red and reindexed from 1 to $|\hat{V}| = 2$. The figure also reports the mapping for each $j \in \hat{V}$ to its index $\alpha_j(\hat{V})$ in V, as explained in Section 5.1; dashed lines represent edges (conflicts) involving vertices not in $\hat{V}$.

In node 3, the set $\hat{V}$ is the empty set and the set $\hat{I}$ is $\{1, 3, 7\}$. Therefore, the node corresponds to a leaf of the branch-and-bound tree on which no further branching operations take place. In this node, both the incumbent solution I_{inc} and the corresponding incumbent solution value LB are updated.

In nodes 4, 6, and 7, the MCKP-based lookup-table bound is strong enough to prune the nodes.

In node 5, $\hat{V}$ becomes empty since the residual capacity of the node $c(\hat{V})$ is 4, which is greater than the 5 units of weight of item 7. Accordingly, both the branching and pruned sets become empty.

As no further nodes are left, the `CFS` algorithm terminates.

7. Computational Experience

In this section, we present the results of an extensive computational campaign carried out to assess the performance of our new branch-and-bound algorithm `CFS`. Our main goal is threefold:

(*i*) comparing `CFS` to the ILP formulations presented in Section 3 (Section 7.1.1)

(*ii*) evaluating the strength and the impact of the different bounding procedures used by `CFS` (Sections 7.1.2 and 7.1.3)

(*iii*) comparing the performance of `CFS` to the state-of-the-art algorithm from the literature introduced in [4] which, in the following, we refer to as `BCM`; the implementation of `BCM` has been kindly provided by the authors of [4] (Section 7.1.4).

The experiments are run on a 20-core Intel(R) Xeon(R) CPU E5-2690 v2@3.00GHz, with 128 GB of main memory, running a 64-bit Linux operating system. Both `CFS` and `BCM` are implemented in `C++`, compiled with `gcc 4.8.4` (with the `-o3` optimization setting), and run on a single core. On this machine, the performance of the algorithm `dfmax`, commonly used for comparing the efficiency of machines with different hardware, is of 0.189, 1.155, and 4.369 seconds for the benchmark graphs `r300.5`, `r400.5` and `r500.5`, respectively. Throughout the section, all CPU times are reported in seconds. On each instance and each run, we adopt a time limit of 600 seconds. The ILP formulations are solved using the state-of-the art commercial MILP solver `CPLEX`, version 12.8.0 (called just `CPLEX` in what follows), run in single-threaded mode, leaving all its parameters to their default value.

As main testbed for our experiments, we consider the instances tested in [4], which feature conflict graphs with nine values of density, ranging from 0.1 to 0.9. The same set of instances has been considered in other works, including [11, 13, 40]. In particular, all these instances were obtained starting from the Bin Packing Problem instances proposed in [12]. We refer to this testbed as the *main testbed* (results are reported in Section 7.1).

Along the lines of [4], we also consider a second testbed containing instances on which `BCM`, which is a combinatorial branch-and-bound algorithm, is outperformed by the branch-and-cut algorithm based on mathematical programming techniques implemented in `CPLEX`. Similarly to `BCM`, `CFS` is not as efficient as `CPLEX` on these instances (while still resulting more efficient than `BCM`). We report the experiments on those instances in Section 7.2. We refer to this testbed as the *very sparse testbed.*

7.1. Results on the main testbed

The main testbed features instances belonging to one of eight types and to one of two classes. Each of them is obtained adopting one of four capacity multipliers. The type specifies the number of items and their weight structure, the class the profit structure, and the capacity multiplier the value by which the capacity is scaled. In all instances of type one to four, the items have weights uniformly distributed in $[20, 100]$ and the (unscaled) knapsack capacity c is equal to 150. In them, the number of items n (equal to the number of vertices of the conflict graph $|V|$) is 120, 250, 500, and $1,000$, respectively. Instances of types five to eight feature items with weights uniformly distributed in $[250, 500]$ and c equal to $1,000$. For these instances, the number of items n is 60, 120, 349, and 501, and each triplet of items forms an exact packing (i.e., it perfectly fits the knapsack, saturating its capacity). The two classes specify the way the item profits are determined. The instances of the first class, the *random class*, which we identify by the letter "R", feature items with random profits uniformly distributed in $[1, 100]$. The instances of the second class, the *correlated class*, which we identify by the letter "C", feature items with correlated profits equal to $p_i = w_i + 10$, $i = 1, \dots, n$. As to the capacity multiplier, according to [4] we consider instances with: (i) the original capacity (150 or $1,000$, based on the instance type) (ii) three times the original capacity, and (iii) ten times the original capacity. Since high values of the knapsack capacity c tend to make the instances harder to solve, in this work we also consider a fourth setting leading to more challenging instances: (iv) fifteen times the original capacity. When referring to an instance in the main testbed, the natural number that follows the letter C or R identifies the multiplier adopted for the knapsack capacity. For each choice of type, class, and multiplier, the testbed contains 90 instances (10 instances for each of the 9 density values of the conflict graph), comprising, overall, 5760 instances.

7.1.1. Performance of the ILP formulations

First, we experiment with the three ILP formulations ILP_1, ILP_2, and ILP_3, which we presented in Section 3, and assess the strength of their LP relaxations, which we denote by LP_1, LP_2, and LP_3. We recall that, as mentioned in Section 3, for the construction of the clique collection $\hat{\mathscr{C}}(V)$ for ILP_3 we adopt the procedure proposed in [4]. For this experiment, we consider a subset of instances of the main testbed of class C, capacity multipliers 1, 3, and 10, and one instance for all eight types and nine densities. The subset contains a total of 216 instances.

When comparing the three ILP formulations by solving them with `CPLEX`, we observe that `CPLEX` manages to solve 168 instances with ILP_1 (77%), 175 with ILP_2 (81%), and 157 with ILP_3 (72%). The results are better summarized in the performance profile in Figure 5, which provides a graphical representation of the relative performance of `CPLEX` with the three formulations.[4] The figure confirms that the best performances are obtained with ILP_2.

[4]For each instance, we compute the normalized time τ as the ratio of the computing time of the considered algorithm (which is ∞ if the instance is not solved to optimality) over the minimum computing time taken by all the algorithms we tested. For each value of τ on the horizontal axis, the vertical axis reports the percentage of instances for which the corresponding algorithm spent at most τ times the computing time of the fastest algorithm. At $\tau = 0$, the value of the curves is equal to the percentage of instances in which the corresponding algorithm has proven to be the fastest. At the right end of the chart (for τ approaching ∞),

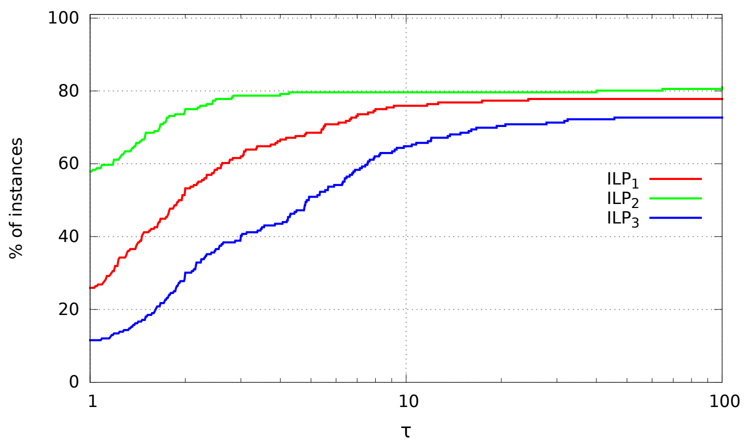


Figure 5: Performance profile comparing the three ILP formulations ILP_1, ILP_2, and ILP_3 on a subset of instances of the main testbed.

To better understand why ILP_2 outperforms the other two formulations, we have assessed the strength of the three corresponding LP relaxations. Table 4 reports the computing time needed to solve each of them, aggregating the instances by class, capacity multiplier, and density. Average results over eight instances are reported for each combination of the three. As the table shows, the optimality gaps obtained with the three LP relaxations become very large for larger values of the capacity multiplier as well as for larger values of the graph density.

It is worth mentioning that these gaps, as well as all the gaps reported in the following sections, are computed, for each instance, as the percentage difference between the value of the upper bound and the optimal solution value. In order to be able to fairly compare the gaps, we use as denominator the optimal solution value of the instance. In case an instance is not solved to optimality, the gap is computed by resorting, rather than to the optimal solution value, to the best solution value found by any of the algorithms presented in this work.

By looking at Table 4, on, for instance, the C instances with density equal to 0.6 and a capacity multiplier ranging from 1 to 10, the average gaps range from 3.26% to 192.52% for LP_1, from 3.84% to 196.00% for LP_2, and from 1.68% to 113.68% for LP_3. Similarly, on the C instances with capacity multiplier equal to 10 and for densities ranging from 0.1 to 0.9, the average gaps range from 5.69% to 417.94% for LP_1, from 7.22% to 424.42% for LP_2, and from 2.09% to 234.97% for LP_3.

When comparing the average strength of the three bounds and the corresponding average gaps over all instances used in this experiment, the table shows that the gaps obtained with LP_1 (63.6% on average) are smaller than those obtained with LP_2 (65.4% on average), although not by a large margin. Moreover, it shows that this small improvement comes at a computing time of, on some instances, about two orders of magnitude larger, with an average of 0.7 seconds with LP_1 and one of 0.01 seconds with LP_2. The gaps obtained with

each curve corresponds to the percentage of instances solved by a specific algorithm. The best performance is achieved by the algorithm whose curves occupy the upper part of the chart.

LP_3 are smaller (37.36% on average) than either of those obtained with LP_1 and LP_2 but, unfortunately, solving LP_3 can be extremely time consuming (up to about three and a half orders of magnitude more), with an average computing time of 3.9 seconds.

For these reasons, we will adopt LP_2 as the baseline for assessing, in the next set of experiments, the strength of the bounds we introduced in Section 5 for our novel algorithm `CFS`.

Table 4: Performance of the upper bounds provided by LP_1, LP_2, LP_3 (subset of instances).

		LP_1		LP_2		LP_3	
class/mult	dens	time	LP gap	time	LP gap	time	LP gap
C1	0.1	0.11	0.73	0.00	0.82	1.04	0.73
	0.3	0.30	1.62	0.01	2.14	3.75	1.55
	0.6	1.01	3.26	0.01	3.84	6.22	1.68
	0.9	1.38	10.31	0.02	10.97	4.75	5.63
C3	0.1	0.11	0.64	0.00	1.54	0.99	0.57
	0.3	0.27	3.75	0.01	5.04	3.63	1.99
	0.6	1.05	14.70	0.01	16.12	6.67	9.75
	0.9	1.36	59.66	0.02	61.48	5.11	45.81
C10	0.1	0.10	5.69	0.00	7.22	0.99	2.09
	0.3	0.28	52.87	0.01	54.81	3.59	29.87
	0.6	0.83	192.52	0.02	196.00	5.98	113.68
	0.9	1.22	417.94	0.03	424.42	4.77	234.97

7.1.2. Assessing the strength of the upper bounds adopted in `CFS`

We now assess the strength of the different bounds we introduced in Section 5, aiming at understanding which ones are better suited for their adoption in our `CFS` algorithm. For this experiment, we take into account a subset of instances of the main testbed of both classes C and R, of types 1, 4, 5, and 8 and with capacity multiplier of value 1 and 15.

Using the LP relaxation LP_2 as baseline, we compare the tightness of two KP-based upper bounds, i.e., the Martello-Toth bound UB_{MT} (15) and the KP-based lookup-table bound UB_{L_1} (14), and of four MCKP-based upper bounds, i.e., the MCKP-based lookup-table bound UB_{L_2} (18), the partition-based bound UB_P (21), the bound obtained by solving to optimality the LP relaxation of the MCKP relaxation of the KPCG (18), which we refer to here as $UB_{\tilde{P}}$, and the bound obtained by solving to optimality the unrestricted (i.e., with integer variables) version of the MCKP relaxation of the KPCG (16), which we refer to here as UB_{P^*}. The latter two are computed via the `MCKNAP.c` code developed by Pissinger, which is available online.[5]

It is important to mention that while $UB_{MT}, UB_{L_1}, UB_{L_2}$, and UB_{L_2} can all be obtained in a negligible amount of computing time, $UB_{\tilde{P}}$ and UB_{P^*} require a higher computational effort. Since in `CFS` these bounds are employed as a pruning mechanism at each node of the branch-and-bound tree, in order to obtain an efficient algorithm it is crucial that the time spent to compute them bounds be extremely small.

[5] `http://hjemmesider.diku.dk/~pisinger/codes.html`

We measure the quality of the upper bound in Tables 5 and 6. The results are reported aggregated by class, capacity multiplier, and type in Table 5, and by class, capacity multiplier, and density in Table 6. For each combination of these, we report average results obtained over nine instances (one per value of density) in Table 5, and over eight instances (one per type) in Table 6.

As to the KP-based bounds, the tables show that UB_{MT} is often close to LP_2 when the capacity multiplier is equal to 1, whereas UB_{MT} becomes substantially worse than LP_2 when the capacity multiplier is equal to 15 (on, in particular, instances of type 1, 4, and 5). UB_{L_1} is, by construction, at least as tight as UB_{MT} and, as the table shows, their difference, while observable for low-capacity instances with a multiplier of 1 and a large density, is not too pronounced. UB_{L_1} turns out to be stronger than LP_2 on instances with a small capacity multiplier and, in particular, a low density. As to the MCKP-based bounds, the tables show that these are always tighter than the KP-based ones (UB_{L_2} is, by construction, at least as tight as UB_{L_1}), and that they are much stronger than LP_2. When comparing them, we observe that the MCKP-based lookup-table bound provided by UB_{L_2} may differ from the optimal solution value of the MCKP relaxation UB_{P^*} by being (see the C15 instances) tighter than it. This is because, in our implementation, UB_{L_2} is enhanced by considering a second clique partition which is constructed by taking the vertices (items) into account greedily according to a non-decreasing profit-over-weight ratio, i.e., in reversed order w.r.t. the order introduced in (1). The MCKP-based lookup-table bound which we report in Tables 5 and 6 (which is used in `CFS`) is set to the best among the values provided by both partitions. It follows that UB_{L_2} is at least as tight as UB_{P^*}, as the latter only considers one of the two partitions.

We conclude this set of experiments with Table 7, which compares the results obtained with `CFS` to those obtained with a modified version of the algorithm which employs as MCKP-based bound either $UB_{\tilde{P}}$ or UB_{P^*}, rather than UB_P. In this table (as well as in the other ones in the remainder of the paper), the number of steps reported for `CFS` refers to the number of calls to the recursive procedure. It is worth noting that the recursion is implemented with a lookahead of one step thanks to which the leaf nodes of the branch-and-bound tree do not count as steps. A similar implementation detail is described in [46].

The table shows that, while the adoption of these two tighter MCKP bounds reduces, in general, the number of steps (for, in particular, low-density instances), this reduction comes at a high computational cost which, ultimately, results in a smaller amount of instances being solved within the time limit. This set of experiments suggests the adoption in `CFS` of UB_{MT}, UB_{L_2} and UB_P, which achieve a good trade-off between strength and computational burden, allowing, as we will better show in the remainder of the section, for an effective algorithm for solving the KPCG.

7.1.3. Assessing the impact of the different components of our branch-and-bound algorithm

Based on the experimental analysis we illustrated before, our implementation of `CFS` relies on three upper bounds: UB_{MT}, UB_{L_2} and UB_P. We now report the results of an experiment carried out to assess the impact on the overall performance of `CFS` of each of these three bounds. We experiment with four variants of `CFS` which employ either only two of the three bounds or none of them, namely:

- `CFS no` UB_{MT}, obtained by removing the bounding procedure based on UB_{MT}

Table 5: Gap of the upper bounds on C and R instances grouped by class (subset of instances).

			KP-based upper bound		MCKP-based upper bounds			
class/mult	type	LP_2	UB_{MT}	UB_{L_1}	UB_{L_2}	UB_{P}	$\text{UB}_{\tilde{P}}$	UB_{P*}
C1	1	9.3	9.5	7.0	2.8	5.3	5.1	2.8
	4	5.8	5.8	3.5	2.9	4.6	4.6	2.9
	5	1.0	1.0	0.2	0.2	1.0	1.0	0.2
	8	0.5	0.5	0.5	0.1	0.4	0.4	0.1
C15	1	508.6	1111.6	1111.6	114.4	161.6	161.6	161.6
	4	1596.3	1647.4	1646.9	549.3	705.5	702.3	702.2
	5	297.4	496.8	496.6	71.8	93.2	93.2	93.2
	8	252.2	252.2	252.1	191.4	215.7	215.5	215.5
R1	1	34.0	34.7	31.2	12.0	15.0	14.9	12.0
	4	27.2	27.3	25.9	14.1	15.4	15.3	14.1
	5	27.3	27.5	7.1	5.3	19.2	19.2	5.4
	8	26.6	26.6	5.3	3.7	24.4	24.4	3.7
R15	1	457.6	937.4	937.4	121.6	121.6	121.6	121.6
	4	2260.4	2592.0	2592.0	579.0	579.0	579.0	579.0
	5	256.4	519.5	518.6	65.3	65.3	65.3	65.3
	8	487.6	495.7	495.4	231.7	231.8	231.8	231.7

Table 6: Gap of the upper bounds on C and R instances grouped by density (subset of instances).

			KP-based upper bounds		MCKP-based upper bounds			
class/mult	dens	LP_2	UB_{MT}	UB_{L_1}	UB_{L_2}	UB_{P}	$\text{UB}_{\tilde{P}}$	UB_{P*}
C1	0.1	1.2	1.3	0.2	0.0	1.1	1.1	0.0
	0.3	1.9	2.0	0.9	0.3	1.4	1.4	0.3
	0.6	4.3	4.3	3.2	1.7	3.0	3.0	1.7
	0.9	10.1	10.2	9.0	4.6	6.5	6.3	4.6
C15	0.1	112.8	174.7	174.7	91.5	103.1	103.1	103.1
	0.3	374.8	510.7	510.6	218.0	268.1	265.2	265.2
	0.6	810.8	1078.0	1077.8	270.7	349.0	348.9	348.9
	0.9	1711.8	2224.6	2224.1	180.9	264.6	264.6	264.6
R1	0.1	13.5	13.6	2.1	0.8	12.2	12.2	0.8
	0.3	16.9	17.0	5.4	2.2	12.9	12.9	2.2
	0.6	29.3	29.5	17.1	8.5	19.1	19.1	8.5
	0.9	64.6	64.8	51.2	25.5	33.3	33.1	25.5
R15	0.1	143.5	231.9	231.8	110.8	110.8	110.8	110.8
	0.3	422.5	614.8	614.6	227.0	227.1	227.1	227.0
	0.6	919.9	1294.8	1294.4	285.7	285.7	285.7	285.7
	0.9	1948.9	2696.2	2695.6	195.7	195.7	195.7	195.7

Table 7: Comparison on a subset of instances of the main testbed between CFS and a version of CFS which, as MCKP-based bound, employs either $UB_{\tilde{P}}$ or UB_{P^*}.

		CFS			CFS with $UB_{\tilde{P}}$			CFS with UB_{P^*}		
class/mult	dens	time	solved	steps	time	solved	steps	time	solved	steps
C10	0.1	41.4	47	1,771,531	67.1	43	762,894	108.0	46	81,694
	0.2	79.0	50	12,464,774	160.1	45	8,795,802	0.4	30	44,285
	0.3	1.6	50	377,895	2.3	50	377,895	3.4	50	377,895
	0.4	11.5	70	1,862,474	13.9	70	1,862,474	17.7	70	1,862,474
	0.5	28.8	80	3,010,873	33.6	80	3,010,873	39.6	80	3,010,873
R10	0.1	11.2	71	103,812	11.9	70	58,989	16.5	70	51,711
	0.2	47.9	59	1,662,823	35.5	52	465,522	18.3	50	294,554
	0.3	42.8	70	2,245,549	107.7	68	1,971,012	70.4	61	1,348,133
	0.4	2.0	70	150,268	3.1	70	149,961	3.4	70	149,465
	0.5	7.2	80	330,597	10.2	80	330,037	11.0	80	329,239

- CFS no UB_{L_2}, obtained by removing the bounding procedure based on UB_{L_2}
- CFS no UB_P, obtained by removing the bounding procedure based on UB_P
- CFS basic, obtained by removing all three bounding procedures.

For this experiment, we consider 240 instances taken from the main testbed, of class C and R, capacity multiplier of value 1, 3, and 10, all 8 types, and density up to 0.05.

By just looking at the number of instances solved within the time limit, we observe that both CFS and CFS no $\mathrm{UB}_{\mathrm{MT}}$ solve 99 instances out of 114 (86.8%), that both CFS no $\mathrm{UB}_{\mathrm{L_2}}$ and CFS no UB_{P} solve 96 (84.2%), and that CFS basic only manages to solve 66 (57.9%). While this shows that at least one of the three bounding procedures is needed for the computational efficiency of CFS, those numbers may seem to indicate that the benefits of adopting all three bounds rather than two of them are not too pronounced. This is not the case, though, as one can see from the results summarized in the performance profile reported in Figure 6, which considers the instances on which CFS basic takes at least 0.01 seconds to find an optimal solution. Besides confirming that the performance of the CFS algorithm completely deteriorates when (in CFS basic) all three bounding procedures are dropped, it shows that even removing any of the three also leads to a substantial deterioration of the performance of the algorithm.

A more detailed comparison between CFS and CFS basic is reported in Table 8, where the instances are aggregated by class, capacity multiplier, and density (up to 0.5). For each combination of the three, the table reports the average over 8 instances. The table shows that the difference in performance between the two variants of the algorithm can be extremely large, up to many orders of magnitude. As an example, we observe that CFS manages to solve the R3 instances with density 0.2 in less than a second performing less than 200 steps, whereas for their solution CFS basic requires, on average, more than a minute and more than 100 million steps. Over the whole subset of instances, the table suggests that the difference between the two variants of the algorithm is particularly large on medium-capacity instances with a capacity multiplier equal to 3 regardless of their class and of their density,

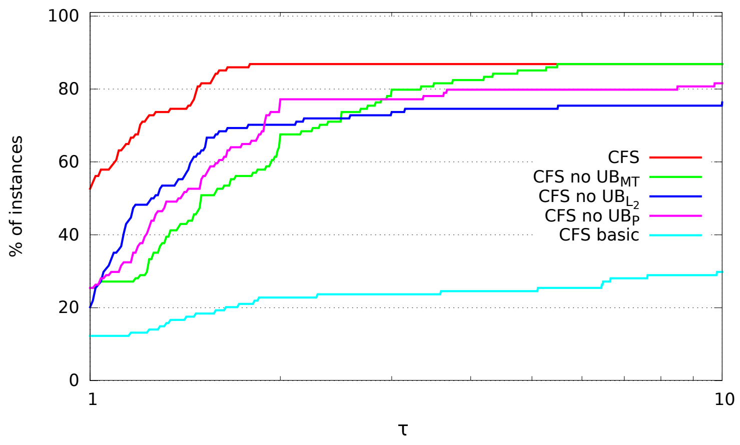


Figure 6: Performance profile showing the impact of the three bounding procedures UB_{MT}, UB_{L_2}, and UB_P on the performance of the CFS algorithm on a subset of instances taken from the main testbed. The profile is truncated at one order of magnitude ($\tau = 10$).

further confirming that the impact of the bounding procedures on the performance of CFS is substantial.

7.1.4. *Comparison with the state-of-the-art algorithm* BCM

In this section, we compare our algorithm CFS to the state-of-the-art algorithm BCM proposed in [4], as well as to the ILP formulation ILP_2 (solved with CPLEX) which, as observed before, is the better-performing one of three ILP formulations on the main testbed.

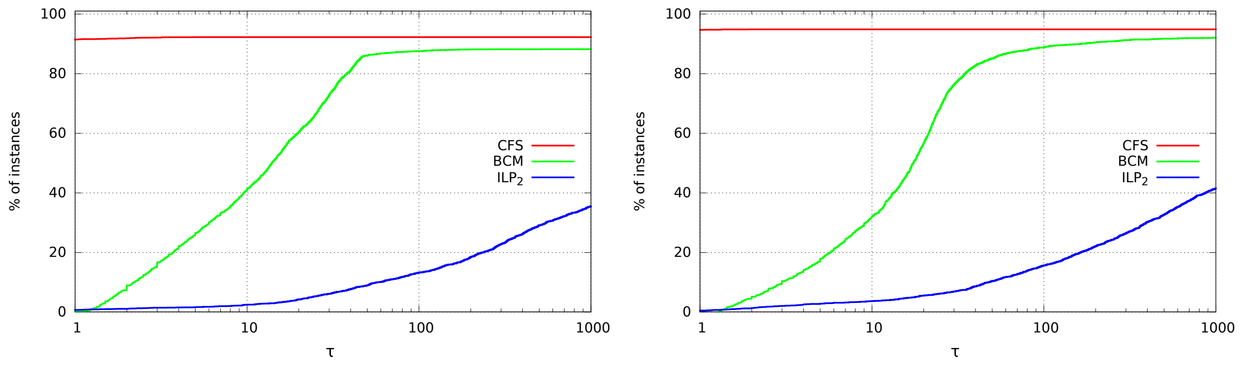


Figure 7: Performance profile comparing CFS to BCM and ILP_2 on the whole main testbed, with instances of class C on the left and instances of class R on the right.

The results are summarized in the performance profiles reported in Figure 7. Besides showing that, on the main testbed, the ILP_2 formulation yields the worst performance among the three solution methods, the two profiles clearly show that the CFS algorithm substantially outperforms BCM. As an example, allowing for one order of magnitude more the time taken by CFS, BCM manages to solve no more than 40% of the instances, whereas ILP_2 only solves less than 10% of them on both classes C and R. Allowing for two more orders of magnitude the time taken by CFS, BCM reaches a plateau, still solving less than 90% of the instances, whereas ILP_2 only solves less than 20% of them. While the performance of BCM does not

Table 8: Performance of `CFS` when compared to `CFS basic` (subset of instances).

		`CFS`			`CFS basic`		
class/mult	dens	solved	time	steps	solved	time	steps
C1	0.1	8	0.0	1	8	0.0	83
	0.2	8	0.0	4	8	0.0	1,477
	0.3	8	0.0	4	8	0.0	1,262
	0.4	8	0.0	6	8	0.0	1,134
	0.5	8	0.0	10	8	0.0	1,132
C3	0.1	8	0.0	234	5	108.9	434,261,687
	0.2	8	0.0	3,731	3	61.1	171,596,377
	0.3	8	0.5	16,682	5	0.5	2,028,623
	0.4	8	0.6	19,372	5	5.2	17,704,019
	0.5	8	0.9	63,543	7	13.2	30,227,171
C10	0.1	5	41.5	2,079,125	2	30.4	61,662,582
	0.2	5	97.2	15,589,771	4	33.5	12,152,384
	0.3	5	1.7	422,801	5	1.1	461,864
	0.4	7	12.7	2,128,162	7	8.7	2,271,613
	0.5	8	29.3	3,079,190	8	19.7	3,338,595
R1	0.1	8	0.0	7	8	0.0	362
	0.2	8	0.0	6	8	0.0	45
	0.3	8	0.0	9	8	0.0	598
	0.4	8	0.0	12	8	0.0	256
	0.5	8	0.0	11	8	0.0	227
R3	0.1	8	0.0	102	4	0.8	2,189,786
	0.2	8	0.0	185	6	63.1	102,960,668
	0.3	8	0.0	657	6	2.7	3,385,209
	0.4	8	0.0	400	7	6.2	2,545,807
	0.5	8	0.0	1,096	8	18.4	3,832,851
R10	0.1	7	3.1	64,813	3	0.6	470,881
	0.2	6	48.3	1,804,824	5	9.7	3,496,101
	0.3	7	42.5	2,226,578	7	83.0	16,191,876
	0.4	7	2.0	155,378	7	2.6	584,678
	0.5	8	8.2	391,182	8	9.5	1,370,445

improve even if more time is allocated to it, the performance of ILP_2 does improve, but very modestly. Indeed, allowing for three orders of magnitude with ILP_2 only about 40% of the instances can be solved.

A further summary of the results is reported in Table 9, indicating the total number of instances solved to optimality by each of the three methods within the time limit for different combinations of class and capacity multiplier. In total, `CFS` manages to solve 5,389 instances out of 5,760, whereas `BCM` successfully solves only 5,201 instances (188 less than `CFS`), and `CPLEX` with ILP_2 only 4,569 (820 less than `CFS`). More detailed results, reporting number of instances solved, computing time, and steps/number of nodes can be found in the appendix in Tables 11 and 12. Focusing on `CFS` and `BCM` (as `CPLEX` with ILP_2 is substantially outperformed by the other two algorithms in terms of number of instances solved and average computing times), we observe that `CFS` substantially outperforms the state-of-the-art algorithm `BCM`. When looking at some groups of instances such as, e.g., the C15 instances of type 1 and 5, `CFS` is, on average, faster than `BCM` by more than 2 orders of magnitude and, for groups of instances in which the two algorithms solve the same number, `CFS` always turn out to be faster, not only on average, but even on each individual instance.

Table 9: Summary of the results obtained on all the instances in the main testbed. The results for the algorithm that solved more instances are reported in boldface.

	`CFS`	`BCM` [4]	ILP_2
Class/Mult	Solved	Solved	Solved
C1	**720**	**720**	**720**
C3	**720**	**720**	584
C10	**617**	552	446
C15	**600**	550	428
Total on C instances (out of 2880 instances)	**2657**	2542	2178
R1	**720**	**720**	**720**
R3	**720**	**720**	680
R10	**670**	630	508
R15	**622**	590	483
Total on R instances (out of 2880 instances)	**2732**	2660	2391
Grand total (out of 5760 instances)	**5389**	5201	4569

7.2. Result on the very sparse testbed

We conclude this section by presenting computational results on the very sparse testbed, which contains instances with very sparse conflict graphs on which `BCM` (which is a purely combinatorial branch-and-bound algorithm) is outperformed by a branch-and-cut algorithms based on mathematical programming, see [4]. This set was generated along the lines of the sparser instances used in [21] and in [53]. They comprise graphs with densities equal to 0.001, 0.002, 0.005, 0.01, 0.02 and 0.05, a capacity of 1,000 and 2,000, and sets of 500 and 1,000 items, for a total of 24 groups of instances. All weights are sampled uniformly at random from the interval [1, 100]. Each group contains 10 random instances (class R) and 10 correlated ones (class C). The profits are randomly drawn in the former and defined as $p_j = w_j + 10$, $i \in N$, in the latter. Overall, this dataset comprises 480 instances.

Thanks to extensive preliminary computational results, when comparing the performance of `CPLEX` with the three ILP formulations ILP_1, ILP_2, and ILP_3, formulation ILP_1 turns out to achieve the best performance. The performance of ILP_3 is, while worse, almost comparable, whereas ILP_2 is completely outperformed by the other two. In light of this, we will now compare, on this testbed, `CFS` and `BCM` to ILP_1. The results are summarized in Table 10, while more details can be found in Tables 13 and 14 in the appendix.

The results show that `CPLEX` with ILP_1 manages to solve the largest amount of instances, 429 (89%), whereas `BCM` only solves 263 instances (54%), which is consistent with the results reported in [4]. Interestingly, while not as efficient as `CPLEX` with ILP_1 on this testbed, `CFS` manages to solve 332 instances (69%), i.e., 69 more instances than `BCM`. Moreover, for instances with density up to 0.005, the performance of `CFS` is comparable to the performance of `CPLEX` with ILP_1 and substantially better than that of `BCM`. When considering all the instances with density up to 0.005 (240 in total), `CPLEX` with ILP_1 manages to solve all of them and `CFS` manages to solve all of them but four, whereas `BCM` fails to solve 29 of them.

One of the main reasons behind the very good performance of `CPLEX` with ILP_1 is that the LP relaxation of ILP_1 provides a very strong upper bound when the conflict graph is very sparse. Indeed, the LP-relaxation gap of ILP_1, ILP_2, and ILP_3 is typically much tighter than the gap we measured on the main testbed, being, on average, equal to, respectively, 2.8%, 10.8%, and 1.6%. The average computing times are 0.05 seconds for ILP_1, negligible for ILP_2, and 0.18 seconds for ILP_3. We remark that our algorithm heavily relies on the conflict graph for reducing the search space after each branching operation and such reduction is often very modest when the graph is extremely sparse. Moreover, the bounds offered by the bounding procedures that we use, i.e., UB_{MT}, UB_{L_2}, and UB_P, are weaker than the three LP-relaxation bounds, being equal, on average, to 15.8%, 5.4%, 5.6%, respectively. It is worth noticing that, when the conflict graph is very sparse, the clique partition adopted in the MCKP-based bounds is very likely to contain very small cliques, thus only capturing a fraction of the conflicts in the graph (only those among items in the same cliques), which results in a weak bound. For these reasons, sparse graphs with densities between 0.01 and 0.05 are not suitable for being solved with our purely combinatorial branch-and-bound algorithm `CFS`. On the contrary, when the conflict graph is extremely sparse, with densities up to 0.005, the conflicts are almost irrelevant and the KP-based bounds are sufficiently strong, which allows `CFS` to achieve a performance comparable to that of `CPLEX` with ILP_1.

Table 10: Summary of the results obtained on all the instances in the very sparse testbed. The results for the algorithm that solved more instances are reported in boldface.

	`CFS`	`BCM` [4]	ILP_1
Class/Capacity	Solved	Solved	Solved
C1000	88	65	**107**
C2000	68	44	**93**
Total on C instances (out of 240 instances)	156	109	**200**
R1000	103	92	**119**
R2000	73	62	**110**
Total on R instances (out of 240 instances)	176	154	**229**
Grand total (out of 480 instances)	332	263	**429**

8. Conclusions

In this paper, we have addressed the Knapsack Problem with Conflict Graph (KPCG), which is an important generalization of the classical Knapsack Problem to the case with conflicts between pairs of items. The KPCG is particularly relevant as it often appears as a subproblem when solving other well-known combinatorial optimization problems with methods based on column generation, such as, e.g., the Bin Packing Problem or the Bin Packing Problem with Conflict Graph. The KPCG is computationally very challenging, and solving it via methods based on integer programming is hard for graphs with densities between 0.1 and 0.9.

In this work, we have proposed a new purely-combinatorial branch-and-bound algorithm, called `CFS`, which relies on the notions of *branching and pruned sets* and on a collection of bounding procedures, based on different relaxations, for fathoming the nodes of the implicit enumeration tree.

We have extensively assessed the performance of `CFS` on a large testbed of instances typically adopted in the literature. The results show that, for graphs with densities equal to 0.1 or more, our algorithm clearly outperforms the state-of-the-art method proposed in [4] as well as different integer programming formulations solved with `CPLEX` by, in some cases, up to several orders of magnitude in terms of computing times.

Future lines of research may include the extension of the ideas proposed in this paper to other generalizations of the Knapsack Problem with an underlying graph, modeling, for example, precedence constraints, such as the Knapsack Problem with Precedences, see, e.g., [5].

References

[1] H. Akeb, M. Hifi, and M. E. Ould Ahmed Mounir. Local branching-based algorithms for the disjunctively constrained knapsack problem. *Computers & Industrial Engineering*, 60(4):811–820, 2011.

[2] E. Amaldi, S. Coniglio, and S. Gualandi. Improving cutting plane generation with 0-1 inequalities by bi-criteria separation. In *International Symposium on Experimental Algorithms*, pages 266–275. Springer, 2010.

[3] E. Amaldi, S. Coniglio, and S. Gualandi. Coordinated cutting plane generation via multi-objective separation. *Mathematical Programming*, 143(1-2):87–110, 2014.

[4] A. Bettinelli, V. Cacchiani, and E. Malaguti. A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, 29(3):457–473, 2017.

[5] N. Boland, A. Bley, C. Fricke, G. Froyland, and R. Sotirov. Clique-based facets for the precedence constrained knapsack problem. *Mathematical programming*, 133(1-2): 481–511, 2012.

[6] J.-M. Bourjolly, G. Laporte, and H. Mercure. A combinatorial column generation algorithm for the maximum stable set problem. *Operations Research Letters*, 20(1):21–29, 1997.

[7] S. Coniglio and M. Tieves. On the generation of cutting planes which maximize the bound improvement. In *International Symposium on Experimental Algorithms*, pages 97–109. Springer, 2015.

[8] R. Corrêa, D. Delle Donne, I. Koch, and J. Marenco. General cut-generating procedures for the stable set polytope. *Discrete Applied Mathematics*, 245:28–41, 2018.

[9] F. Della Croce and R. Tadei. A multi-kp modeling for the maximum-clique problem. *European Journal of Operational Research*, 73(3):555 – 561, 1994.

[10] M. Dyer. An $O(n)$ algorithm for the multiple-choice knapsack linear program. *Mathematical Programming*, 29(1):57–63, 1984.

[11] S. Elhedhli, L. Li, M. Gzara, and J. Naoum-Sawaya. A branch-and-price algorithm for the bin packing problem with conflicts. *INFORMS Journal on Computing*, 23(3): 404–415, 2011.

[12] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *J. Heuristics*, 2(1): 5–30, 1996.

[13] A. Fernandes Muritiba, M. Iori, E. Malaguti, and P. Toth. Algorithms for the bin packing problem with conflicts. *Informs Journal on computing*, 22(3):401–415, 2010.

[14] M. Fischetti and A. Lodi. Local branching. *Mathematical programming*, 98(1-3):23–47, 2003.

[15] M. Giandomenico, F. Rossi, and S. Smriglio. Strong lift-and-project cutting planes for the stable set problem. *Mathematical Programming*, 141(1-2):165–192, 2013.

[16] M. Giandomenico, A. N. Letchford, F. Rossi, and S. Smriglio. Ellipsoidal relaxations of the stable set problem: theory and algorithms. *SIAM Journal on Optimization*, 25(3): 1944–1963, 2015.

[17] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[18] S. Held, W. Cook, and E. C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4(4):363–381, 2012.

[19] M. Hifi. An iterative rounding search-based algorithm for the disjunctively constrained knapsack problem. *Engineering Optimization*, 46(8):1109–1122, 2014.

[20] M. Hifi and M. Michrafy. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society*, 57(6):718–726, 2006.

[21] M. Hifi and M. Michrafy. Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. *Computers & Operations Research*, 34(9):2657–2673, 2007.

[22] M. Hifi and N. Otmani. An algorithm for the disjunctively constrained knapsack problem. *International Journal of Operational Research*, 13(1):22–43, 2012.

[23] S. Hosseinian, D. B. M. M. Fontes, and S. Butenko. A nonconvex quadratic optimization approach to the maximum edge weight clique problem. *Journal of Global Optimization*, 72(2):219–240, 2018.

[24] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[25] H. Jiang, C. Li, and F. Manyà. An exact algorithm for the maximum weight clique problem in large graphs. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 830–838, 2017.

[26] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[27] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer, 2004.

[28] C. Li, H. Jiang, and F. Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017.

[29] C. Li, Z. Fang, H. Jiang, and K. Xu. Incremental upper bound for the maximum clique problem. *INFORMS Journal on Computing*, 30(1):137–153, 2018.

[30] C. Li, Y. Liu, H. Jiang, F. Many, and Y. Li. A new upper bound for the maximum weight clique problem. *European Journal of Operational Research*, 270(1):66–77, 2018.

[31] S. Martello and P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1(3):169–175, 1977.

[32] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, Chichester, New York, 1990.

[33] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

[34] E. Maslov, M. Batsyn, and P. Pardalos. Speeding up branch and bound algorithms for solving the maximum clique problem. *Journal of Global Optimization*, 59(1):1–21, 2014.

[35] G. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society*, 43(5):443–457, 1992.

[36] U. Pferschy and J. Schauer. The knapsack problem with conflict graphs. *Journal of Graph Algorithms and Applications*, 13(2):233–249, 2009.

[37] D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83(2):394–410, 1995.

[38] S. Rebennack, M. Oswald, D. Theis, H. Seitz, G. Reinelt, and P. Pardalos. A branch and cut solver for the maximum stable set problem. *Journal of Combinatorial Optimization*, 21(4):434–457, 2011.

[39] F. Rossi and S. Smriglio. A branch and cut algorithm for the maximum cardinality stable set problem. *Operations Research Letters*, 28(2):63–74, 2001.

[40] R. Sadykov and F. Vanderbeck. Bin packing with conflicts: a generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):244–255, 2013.

[41] P. San Segundo and C. Tapia. Relaxed approximate coloring in exact maximum clique search. *Computers & Operations Research*, 44:185–192, 2014.

[42] P. San Segundo, D. Rodríguez-Losada, and A. Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[43] P. San Segundo, F. Matia, D. Rodriguez-Losada, and M. Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, 2013.

[44] P. San Segundo, A. Nikolaev, and M. Batsyn. Infra-chromatic bound for exact maximum clique search. *Computers & Operations Research*, 64:293–303, 2015.

[45] P. San Segundo, A. Nikolaev, M. Batsyn, and P. M. Pardalos. Improved infra-chromatic bound for exact maximum clique search. *Informatica, Lithuanian Academy of Sciences*, 27(2):463–487, 2016.

[46] P. San Segundo, S. Coniglio, F. Furini, and I. Ljubić. A new branch-and-bound algorithm for the maximum edge-weighted clique problem. *European Journal of Operational Research*, 278(1):76–90, 2019.

[47] P. San Segundo, F. Furini, and J. Artieda. A new branch-and-bound algorithm for the maximum weighted clique problem. *Computers & Operations Research*, 110:18 – 33, 2019.

[48] S. Shimizu, K. Yamaguchi, and S. Masuda. A branch-and-bound based exact algorithm for the maximum edge-weight clique problem. In *Computational Science/Intelligence & Applied Informatics, CSII 2018*, pages 27–47, 2018.

[49] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[50] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global optimization*, 37 (1):95–111, 2007.

[51] L. Wei, Z. Luo, R. Baldacci, and A. Lim. A new branch-and-price-and-cut algorithm for one-dimensional bin-packing problems. *INFORMS Journal on Computing*, page Published online in Articles in Advance 01 Nov 2019, 2019.

[52] W. Wu and J.-K. Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[53] T. Yamada, S. Kataoka, and K. Watanabe. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. *Journal of Information Processing (JIP)*, 43(9):2864–2870, 2002.

[54] E. Zemel. An $O(n)$ algorithm for the linear multiple choice knapsack problem and related problems. *Information Processing Letters*, 18(3):123–128, 1984.

Appendix

Tables 11 and 12 report the detailed results obtained on all the instances in the main testbed obtained when comparing our algorithm `CFS` to the `BCM` algorithm and to the formulation ILP_2 solved with `CPLEX`. Tables 14 and 13 report the detailed results obtained on all the instances in the very sparse testbed obtained when comparing our algorithm `CFS` to the `BCM` algorithm and to the formulation ILP_1 solved with CPLEX.

Table 11: Comparison of `CFS`, `BCM`, and ILP_2 on all the instances o the main testbed (aggregated by type).

		`CFS`			`BCM` [4]			ILP_2		
Class	type	solved	time	steps	solved	time	nodes	solved	time	nodes
C1	1	90	0.0	10	90	0.0	59	90	0.2	257
	2	90	0.0	19	90	0.0	93	90	1.1	687
	3	90	0.0	27	90	0.0	164	90	8.2	1,302
	4	90	0.0	96	90	0.0	368	90	24.2	4,779
	5	90	0.0	10	90	0.0	130	90	0.0	35
	6	90	0.0	11	90	0.0	174	90	0.1	79
	7	90	0.0	5	90	0.0	175	90	0.5	209
	8	90	0.0	5	90	0.0	178	90	3.6	476
C3	1	90	0.0	390	90	0.0	1,931	90	1.5	2,236
	2	90	0.0	1,845	90	0.1	11,076	90	25.8	23,911
	3	90	0.1	12,293	90	1.3	79,608	54	162.8	179,946
	4	90	1.6	117,433	90	27.3	768,841	21	141.9	194,077
	5	90	0.0	170	90	0.0	800	90	0.2	277
	6	90	0.0	442	90	0.0	2,697	90	2.0	3,035
	7	90	0.0	1,751	90	0.1	13,847	90	46.5	45,063
	8	90	0.0	8,139	90	0.6	87,145	59	35.3	106,971
C10	1	90	0.1	25,284	90	1.6	390,071	90	3.5	3,375
	2	90	25.2	2,973,643	73	31.9	5,538,512	68	126.2	88,374
	3	61	15.9	1,278,758	50	18.2	2,868,097	22	166.0	146,267
	4	50	47.2	4,892,372	40	108.8	10,917,062	1	575.1	91,400
	5	90	0.0	342	90	0.0	8,842	90	0.2	132
	6	90	0.5	109,309	90	6.8	1,196,970	90	5.3	7,922
	7	86	35.9	5,047,011	70	24.5	5,015,045	65	143.1	180,369
	8	60	7.3	1,201,721	49	17.4	2,616,920	20	156.4	444,277
C15	1	90	0.3	94,719	90	20.5	4,630,800	90	2.8	2,450
	2	80	27.8	5,710,907	70	26.4	5,389,661	60	112.3	110,218
	3	60	6.9	1,079,724	50	19.2	2,868,097	20	238.1	71,065
	4	50	48.5	4,892,372	40	114.6	10,917,062	-	-	-
	5	90	0.0	342	90	0.0	8,842	90	0.2	132
	6	90	0.4	124,867	90	24.3	5,331,239	90	3.8	4,214
	7	80	23.9	4,870,992	70	24.4	5,015,045	58	134.3	152,075
	8	60	7.4	1,201,721	50	18.4	2,831,336	20	126.5	161,151
R1	1	90	0.0	3	90	0.0	24	90	0.1	25
	2	90	0.0	6	90	0.0	37	90	0.8	68
	3	90	0.0	13	90	0.0	75	90	4.8	231
	4	90	0.0	23	90	0.1	150	90	10.1	688
	5	90	0.0	1	90	0.0	15	90	0.0	2
	6	90	0.0	2	90	0.0	28	90	0.1	2
	7	90	0.0	3	90	0.0	54	90	0.4	5
	8	90	0.0	4	90	0.1	84	90	2.7	11
R3	1	90	0.0	42	90	0.0	277	90	0.4	464
	2	90	0.0	115	90	0.0	863	90	5.0	2,840
	3	90	0.0	465	90	0.2	4,036	90	55.1	24,645
	4	90	0.1	1,942	90	2.3	20,494	50	127.2	73,543
	5	90	0.0	17	90	0.0	107	90	0.1	32
	6	90	0.0	44	90	0.0	302	90	0.5	625
	7	90	0.0	132	90	0.0	1,044	90	5.0	3,933
	8	90	0.0	479	90	0.2	3,907	90	64.7	28,637
R10	1	90	0.0	2,192	90	0.1	7,337	90	1.6	1,251
	2	90	0.8	66,015	90	9.1	311,111	87	107.4	68,381
	3	89	49.9	1,926,944	69	57.0	942,483	33	100.0	223,094
	4	51	23.2	597,536	40	25.0	570,826	8	333.6	228,047
	5	90	0.0	68	90	0.0	390	90	0.1	38
	6	90	0.0	3,919	90	0.2	12,147	90	1.5	1,140
	7	90	1.5	130,533	90	17.7	584,911	80	91.7	78,108
	8	79	19.5	1,031,967	69	43.2	735,325	30	77.0	227,310
R15	1	90	0.1	19,800	90	0.4	35,524	90	1.6	1,264
	2	80	2.4	317,410	80	26.1	1,365,483	82	136.6	61,441
	3	70	38.7	2,743,028	60	28.5	1,313,975	26	256.8	67,955
	4	50	12.9	559,601	40	22.6	785,357	-	-	-
	5	90	0.0	68	90	0.0	396	90	0.1	37
	6	90	0.0	8,913	90	0.3	23,126	90	1.6	1,267
	7	82	11.1	672,291	80	20.9	972,270	79	126.7	80,363
	8	70	21.3	1,283,377	60	20.1	806,071	26	207.3	180,780

Table 12: Comparison of `CFS`, `BCM`, and ILP_2 on all the instances o the main testbed (aggregated by density).

		`CFS`			`BCM` [4]			ILP_2		
Class	dens	solved	time	steps	solved	time	nodes	solved	time	nodes
C1	0.1	80	0.0	1	80	0.0	80	80	0.1	1
	0.2	80	0.0	2	80	0.0	81	80	0.2	12
	0.3	80	0.0	3	80	0.0	110	80	0.4	51
	0.4	80	0.0	5	80	0.0	131	80	0.6	117
	0.5	80	0.0	9	80	0.0	133	80	1.3	382
	0.6	80	0.0	23	80	0.0	177	80	2.7	962
	0.7	80	0.0	32	80	0.0	201	80	5.5	1,351
	0.8	80	0.0	66	80	0.0	296	80	14.4	3,300
	0.9	80	0.0	66	80	0.0	299	80	17.5	2,627
C3	0.1	80	0.2	48,344	80	0.3	4,218	77	11.2	42,906
	0.2	80	0.1	5,216	80	2.4	39,115	79	26.7	17,391
	0.3	80	0.4	15,303	80	6.6	109,139	72	27.7	36,378
	0.4	80	0.5	20,283	80	9.1	198,950	66	41.3	75,282
	0.5	80	0.5	34,620	80	9.6	349,663	52	74.6	163,642
	0.6	80	0.2	25,993	80	3.9	256,177	50	41.9	152,269
	0.7	80	0.1	8,477	80	1.0	96,129	50	11.0	84,391
	0.8	80	0.0	1,892	80	0.2	26,353	66	68.4	41,079
	0.9	80	0.0	145	80	0.0	6,945	72	27.2	11,619
C10	0.1	47	41.4	1,771,531	33	37.3	3,781,119	47	53.3	439,083
	0.2	50	79.0	12,464,774	30	4.4	1,396,644	30	4.5	240,572
	0.3	50	1.6	377,895	50	64.7	13,635,344	30	3.7	111,525
	0.4	70	11.5	1,862,474	50	3.7	866,608	48	169.0	86,596
	0.5	80	28.8	3,010,873	69	23.6	3,627,158	50	106.4	93,064
	0.6	80	1.2	138,326	80	52.5	5,334,419	50	38.2	47,904
	0.7	80	0.1	9,393	80	3.7	441,813	50	12.4	39,167
	0.8	80	0.0	920	80	0.3	32,404	70	75.8	17,093
	0.9	80	0.0	144	80	0.0	7,146	71	28.7	7,375
C15	0.1	30	2.0	609,281	30	129.5	28,335,387	30	1.6	200,530
	0.2	50	81.0	16,550,640	30	4.6	1,413,573	30	3.7	97,639
	0.3	50	1.6	377,895	50	67.1	13,653,789	30	3.6	78,495
	0.4	70	11.7	1,862,474	50	3.8	866,608	48	146.7	62,181
	0.5	80	29.6	3,010,873	70	24.9	3,765,881	50	94.4	59,007
	0.6	80	1.2	138,326	80	55.2	5,334,419	50	32.4	31,024
	0.7	80	0.1	9,393	80	3.9	441,813	50	13.6	23,249
	0.8	80	0.0	920	80	0.3	32,404	70	79.4	10,799
	0.9	80	0.0	144	80	0.1	7,146	70	36.4	3,368
R1	0.1	80	0.0	4	80	0.0	31	80	0.1	
	0.2	80	0.0	5	80	0.0	34	80	0.2	5
	0.3	80	0.0	9	80	0.0	41	80	0.3	20
	0.4	80	0.0	11	80	0.0	54	80	0.6	22
	0.5	80	0.0	9	80	0.0	57	80	0.9	31
	0.6	80	0.0	8	80	0.0	67	80	1.6	64
	0.7	80	0.0	9	80	0.0	75	80	4.8	177
	0.8	80	0.0	6	80	0.0	84	80	5.3	331
	0.9	80	0.0	2	80	0.0	86	80	7.2	509
R3	0.1	80	0.0	90	80	0.1	269	80	0.1	20
	0.2	80	0.0	231	80	0.1	1,225	80	1.0	279
	0.3	80	0.0	335	80	0.4	2,900	80	10.9	3,152
	0.4	80	0.0	595	80	0.7	5,340	78	35.6	15,568
	0.5	80	0.0	783	80	0.8	7,908	70	23.3	28,598
	0.6	80	0.0	845	80	0.6	8,380	70	46.3	40,878
	0.7	80	0.0	521	80	0.3	5,545	70	53.4	35,082
	0.8	80	0.0	194	80	0.1	2,424	74	45.2	20,093
	0.9	80	0.0	48	80	0.0	919	78	31.2	7,890
R10	0.1	71	11.2	103,812	69	59.9	451,932	72	16.0	28,850
	0.2	59	47.9	1,662,823	50	39.5	1,310,130	43	85.1	198,861
	0.3	70	42.8	2,245,549	50	3.9	189,370	44	139.2	221,454
	0.4	70	2.0	150,268	70	38.9	1,125,654	50	53.8	149,285
	0.5	80	7.2	330,597	70	3.9	149,029	50	51.6	153,972
	0.6	80	0.4	23,291	80	11.6	266,952	50	16.2	91,389
	0.7	80	0.1	2,738	80	1.3	44,168	54	38.5	57,907
	0.8	80	0.0	307	80	0.2	7,968	70	40.4	18,381
	0.9	79	0.0	53	79	0.0	1,511	75	44.6	7,949
R15	0.1	32	24.8	1,314,639	30	1.6	142,708	42	127.9	108,082
	0.2	50	6.1	778,386	50	70.7	3,441,505	39	96.7	83,026
	0.3	70	57.9	3,883,815	50	4.0	264,416	50	125.0	63,217
	0.4	70	2.1	155,236	70	37.9	1,623,716	50	39.5	45,185
	0.5	80	7.7	338,821	70	3.6	186,837	50	38.9	64,570
	0.6	80	0.5	23,309	80	10.6	368,299	50	17.9	39,013
	0.7	80	0.1	2,738	80	1.1	53,851	62	107.7	30,982
	0.8	80	0.0	307	80	0.2	9,164	70	61.6	8,017
	0.9	80	0.0	53	80	0.0	1,629	70	34.0	2,256

Table 13: Comparison of `CFS`, `BCM`, and ILP_1 on the *very sparse* testbed of *correlated* instances.

		`CFS`		`BCM` [4]		ILP_1	
Items/Cap	Density	Solved	Time	Solved	Time	Solved	Time
500/1000	0.001	10	0.0	10	0.0	10	0.0
	0.002	10	0.0	10	0.6	10	0.0
	0.005	10	0.2	10	6.7	10	0.0
	0.01	10	0.8	9	103.3	10	0.0
	0.02	10	56.7	1	272.7	10	0.3
	0.05	1	165.8	0	-	10	90.6
500/2000	0.001	10	4.2	10	0.4	10	0.0
	0.002	10	0.1	10	5.2	10	0.0
	0.005	10	7.3	8	199.6	10	0.0
	0.01	7	49.8	0	-	10	0.0
	0.02	0	-	0	-	9	6.1
	0.05	0	-	0	-	0	-
1000/1000	0.001	10	0.1	10	5.4	10	0.0
	0.002	10	0.2	10	11.1	10	0.0
	0.005	10	5.9	5	379.9	10	0.0
	0.01	7	163.9	0	-	10	0.1
	0.02	0	-	0	-	7	2.4
	0.05	0	-	0	-	0	-
1000/2000	0.001	10	3.1	9	84.7	10	0.0
	0.002	10	45.8	7	210.3	10	0.0
	0.005	7	182.0	0	-	10	0.0
	0.01	4	0.0	0	-	9	0.1
	0.02	0	-	0	-	5	193.8
	0.05	0	-	0	-	0	-

Table 14: Comparison of `CFS`, `BCM`, and ILP_1 on the *very sparse* testbed of *random* instances.

		`CFS`		`BCM` [4]		ILP_1	
Items/Cap	Density	Solved	Time	Solved	Time	Solved	Time
500/1000	0.001	10	0.0	10	0.0	10	0.0
	0.002	10	0.0	10	0.1	10	0.0
	0.005	10	0.0	10	0.4	10	0.0
	0.01	10	0.1	10	2.1	10	0.0
	0.02	10	1.2	10	32.8	10	0.0
	0.05	9	132.7	3	133.2	10	1.2
500/2000	0.001	10	0.0	10	0.1	10	0.0
	0.002	10	0.0	10	0.3	10	0.0
	0.005	10	0.1	10	2.4	10	0.0
	0.01	10	10.4	9	190.7	10	0.0
	0.02	3	116.5	1	39.6	10	0.1
	0.05	0	-	0	-	10	81.2
1000/1000	0.001	10	0.0	10	0.4	10	0.0
	0.002	10	0.0	10	1.6	10	0.0
	0.005	10	0.1	10	16.8	10	0.0
	0.01	10	15.0	8	152.6	10	0.1
	0.02	4	125.7	1	468.8	10	0.6
	0.05	0	-	0	-	9	255.0
1000/2000	0.001	10	0.0	10	2.3	10	0.0
	0.002	10	0.0	10	20.3	10	0.0
	0.005	9	69.5	2	189.9	10	0.0
	0.01	1	565.8	0	-	10	0.1
	0.02	0	-	0	-	10	2.1
	0.05	0	-	0	-	0	-