Convergent Dual Bounds Using an Aggregation of Set-Covering Constraints for Capacitated Problems

Daniel Cosmin Porumbel
Université Lille Nord de France, F-59000 Lille, France
Université d’Artois, LG2A, F-62400, Béthune, France daniel.porumbel@univ-artois.fr

François Clautiaux
Université Bordeaux 1, Institut de Mathématiques de Bordeaux, INRIA Bordeaux Sud-Ouest,
francois.clautiaux@math.u-bordeaux1.fr

Extended formulations are now widely used to solve hard combinatorial optimization problems. Such formulations have prohibitively-many variables and are generally solved via Column Generation (CG). CG algorithms are known to have frequent convergence issues, and, up to a sometimes large number of iterations, classical Lagrangian dual bounds may be weak. This paper is devoted to set-covering problems in which all elements to cover require a given resource consumption and all feasible configurations have to verify a resource constraint.

We propose an iterative aggregation method for determining convergent dual bounds using the extended formulation of such problems. The set of dual variables is partitioned into \( k \) groups and all variables in each group are artificially linked using the following group-wise restriction: the dual values in a group have to follow a linear function of the resource consumption of the corresponding elements. This leads to a restricted model of smaller dimension, with only \( 2k \) dual variables. The method starts with one group \((k = 1)\) and iteratively splits the groups. Our algorithm has three advantages: (i) it produces good dual bounds even for low \( k \) values, (ii) it reduces the number of dual variables, and (iii) it may reduce the time needed to solve sub-problems, in particular when dynamic programming is used.

We experimentally tested our approach on two cutting-stock variants: in many cases, our method produces nearly optimal dual bounds after a small number of iterations. Moreover the average computational effort to reach the optimum is reduced compared to classical column generation.

1. Introduction

Column Generation (CG) is a well known technique for solving Linear Programs (LPs) with prohibitively-many decision variables. At each iteration, a CG method considers a Reduced Master Problem (RMP) with a limited number of variables. This RMP is solved
to optimality so as to generate: (i) a primal bound for the CG optimum, and (ii) dual values that are provided as input for a pricing sub-problem that computes a new variable of attractive reduced cost. By iteratively solving the sub-problem, columns are added to the RMP and the primal bounds (RMP optima) converge toward the original CG optimum. A dual bound can be determined at each iteration by computing the Lagrangian bound associated to the current dual multipliers.

A frequent issue of column generation methods concern their convergence and (lack of) stabilization. In numerous cases, a large number of iterations might be needed to obtain a useful dual bound. The last decades have seen a surge of interest in stabilization methods, aiming at decreasing the number of iterations in CG [20, 9, 17, 6, 22, 4, 8].

Aggregation and disaggregation techniques have a long history in optimization [25]. Generally speaking, the goal is to transform programs with high degree of detail into more coarser programs of smaller size. For instance, one can aggregate close time instants [19], nearby locations, related scheduling tasks—see [16] for more examples and numerous references. In the remaining, we focus however on the specific context of column generation. Constraint aggregation is used in column generation for stabilization reasons, to reduce the number of dual variables, or to produce smaller RMPs with less degeneracy [10, 11, 3].

We propose a new aggregation technique sharing certain similarities with the Dynamic Constraint Aggregation (DCS) method [10, 11], and, implicitly, with the Stabilized DCS (SDCS) method [3] which combines DCS with a penalty-based stabilization technique [22]. To construct new columns, DCS first disaggregates the dual values to obtain a dual solution expressed in the original dual space (see also Section 2.2). Then, the initial subproblem is solved and the resulting column is either added to the current aggregated RMP, or put aside if it is not compatible with an associated partition. In our method, we decide a priori how the dual variables will be disaggregated by adding cuts that link the dual variables in each group. We add to the RMP all columns that are computed by the aggregated subproblem and some of the resulting columns are only feasible in the aggregated model. This may lead to non-valid dual cuts with respect to the original dual polytope, i.e., excessively strong constraints. In other words, while DCS aims at reducing the number of iterations needed to converge while staying primal feasible, our method aims at producing useful iterative dual bounds by computing dual feasible solutions.
The above non-valid dual cuts generate, for each value of $k$, an inner approximation $P_k$ of the dual polytope $P$. This polytope $P_k$ has only two variables per group, and leads to an easier optimization problem when $k$ is small. Once an optimal solution for $P_k$ is found, a valid dual bound is obtained for the original problem. The method converges toward the optimum by iteratively breaking groups until the optimum of the aggregated model corresponds to the optimum in the original dimension of $P$.

Fig. 1 presents an intuitive illustration of this convergent method. It is very fast in the first steps (coarse aggregation for small $k$) and performs incremental iterative calculations to progressively increase $k$. Compared to other aggregation methods, our approach has the advantage of providing valid dual bounds before fully converging, i.e., each $P_k$ optimum represents a valid dual bound for the original problem.

In this paper, we focus on set-covering master problems and pricing sub-problems modeled as Integer Linear Programs (ILPs) with a resource constraint. In these models, each element to cover requires a resource consumption. The proposed dual groupwise-linearity restrictions work as follows: in each group, the dual values have to follow a linear function of the resource consumption of the corresponding elements. This allows us to reformulate the problem with a new model whose variables are the slope and y-intercept of the linear function over each group. The motivation for using such aggregation comes from the fact that the dual values are often correlated with the resource consumption at optimality.

The remainder is organized as follows. Section 2 presents the classical column generation method and a review of the literature. Section 3 is devoted to our convergent method: the first part (§3.1-§3.3) describes the aggregation modelling for a fixed $k$, while the second one
presents the incremental calculations of the $P_1, P_2, \ldots$ sequence (§3.4–§3.6). In Section 4, we discuss the theoretical benefits of using groupwise-linearity constraints instead of simpler equality constraints. Section 5 presents all details of an application to the cutting-stock problem. Section 6 is devoted to computational experiments, followed by conclusions in the last section.

2. Context, Motivation and Literature Review

2.1. Column Generation for Set-Covering Formulations with Resource Constraints in the Subproblem

Let us first introduce the set-covering model considered throughout this paper. Such models are often used to solve, among many others, cutting and packing [13, 14], vehicle-routing [21], and employee scheduling [11, 10] problems.

The primal ILP is defined on a ground set of elements $I = \{1, \ldots, n\}$. A column, also called configuration hereafter, of the master problem is an integer vector $a = [a_1 \ldots a_n]^\top \in \mathbb{Z}_+^n$. The set $\mathcal{A}$ of all configurations is defined by all the solutions of a given sub-problem. The set-covering problem requires finding the minimum number of configurations that need to be selected so as to cover each element $i \in [1..n]$ at least $b_i$ times ($b_i \in \mathbb{Z}_+$). To each configuration $a \in \mathcal{A}$, we associate a variable $\lambda_a$ that indicates the number of times configuration $a$ is selected. This leads to the following ILP:

$$\begin{align*}
\min & \sum_{a \in \mathcal{A}} \lambda_a \\
\sum_{a \in \mathcal{A}} a_i \lambda_a & \geq b_i, \quad \forall i \in 1, \ldots, n \\
\lambda_a & \in \mathbb{N}, \quad \forall a \in \mathcal{A}
\end{align*}$$

We consider the linear relaxation of this model, where the last constraint is replaced by $\lambda_a \geq 0, \forall a \in \mathcal{A}$. The dual LP is written using a vector $y = [y_1 \ y_2 \ \ldots \ y_n]^\top \in \mathbb{R}_+^n$ of dual variables and a possibly exponential number of dual constraints.

$$\begin{align*}
\max & \, b^\top y \\
\begin{aligned}
a^\top y & \leq 1, \quad \forall a \in \mathcal{A} \\
y_i & \geq 0, \quad i \in 1, \ldots, n
\end{aligned}
\end{align*}$$

In the remainder, we will refer to this problem as the Dual Set-Covering Problem (DSCP) over polytope $\mathcal{P}$: $\text{DSCP}(\mathcal{P}) = \max \{b^\top y : y \in \mathcal{P}\}$. All methods from this paper aim at finding the optimum $\text{OPT}(\text{DSCP}(\mathcal{P}))$ of this program. It is also termed the CG optimum.
OPT\textsubscript{CG}, or the optimum of objective function \( b \) over \( \mathcal{P} \) (we sometimes omit mentioning \( b \), as no other objective functions are used).

Given the prohibitive large size of \( \mathcal{A} \), this constraint set is generated iteratively. From a dual point of view, the classical column generation can be seen as a cutting-plane algorithm: (i) reach the best dual solution in the current dual polytope defined by the limited set of constraints that have been generated so far, (ii) find a valid dual constraint that is violated by the current solution (report optimum if there is no such violated constraint), and (iii) refine the current formulation by adding a violated dual constraint and repeat from (i). Therefore, the column generation method constructs at each step an outer approximation of the dual polytope.

The pricing sub-problem in CG seeks a configuration of most negative reduced cost:

\[
\min_{a \in \mathcal{A}} (1 - y^\top a) = 1 - \max_{a \in \mathcal{A}} y^\top a \tag{2.3}
\]

In the following, we will forget the constant and seek a configuration \( a^* \) that maximizes \( y^\top a^* \). In this sub-problem, \( y \) represents the input data; at each iteration, \( y \) is the dual optimum solution of the RMP.

A key point in column generation is the speed of the sub-problem solution method. Dynamic Programming (DP) algorithms provide an efficient tool for generating columns for many different families of problems (see among many others [21, 13]). We now formalize a general DP framework for (sub-)problems with a single resource constraint. We consider a resource vector \( \mathbf{w} = [w_1 \ w_2 \ldots \ w_n]^\top \in \mathbb{R}_+^n \) and a lower and upper limit of total resource consumption \( (C^-, C^+ \in \mathbb{N}) \). For a given dual solution \( y \in \mathbb{R}^n \), the sub-problem requires selecting \( a_i \) times each element \( i \in [1..n] \) (i.e., finding a configuration) so as to maximize the total profit \( y^\top a \) under the resource constraint: \( C^- \leq \mathbf{w}^\top a \leq C^+ \).

The DP recursion computes a profit function \( P_{\text{max}} \) that maps any state \((c, i) \in [0, C^+] \times [1..n]\) to the maximum profit \( P_{\text{max}}(c, i) \) that can be obtained with a resource amount of \( c \), only selecting elements of \([1..i]\). Generally speaking, this function can be determined using a recursion of the form below.

\[
P_{\text{max}}(c, i) = \max \left\{ P_{\text{max}}(c - r \cdot w_i, i - 1) + r \cdot y_i : r \in [0..b_i], \ \ r \cdot w_i \leq c \right\} \tag{2.4}
\]

The initial state is \( P_{\text{max}}(0, 0) = 0 \) and the final minimum reduced cost is attained in a state \((c^*, n)\) such that \( P_{\text{max}}(c^*, n) = \max_{c \in [C^-, C^+]} P_{\text{max}}(c, n) \). If the given capacitated problem
imposes additional conditions on feasible patterns, this formula is generalized by imposing additional constraints on the feasible subsets of elements. For example, in Bin Packing with conflicts [12], some pairs of elements cannot be selected together.

The above recursion (2.4) can be computed in $O(n^b C^+)$ time, where $n_b = \sum_{i=1}^{n} b_i$ is the number of individualized elements. We use $n_b$ instead of $n$: the elements with demand multiplicities $b_i > 1$ are not considered only once in (2.4), but $b_i$ times (observe $r \in [0..b_i]$). For the Bin-Packing problem, $n_b = n$ and the complexity becomes $O(nC^+)$. In other resource-constrained problems, such as the vector-packing problem [5], additional constraints can restrict the set of feasible patterns and/or increase the dimension of the state space.

Although we describe here the case with one resource, it can also be applied when several resources are considered. One only needs multiple dimensions to index the states.

2.2. Aggregation in Column Generation and Extended Formulations

A way to cope with prohibitively large Mixed-Integer Programs (MIPs) arising in extended formulations is to approximate them. This can be done by restricting the MIP to a subset of variables and/or constraints. This leads to a primal approximation that can be refined iteratively (typically by column-and-row generation, see [23] for a generic view of those methods).

Another way of obtaining a tractable model is to apply aggregation to the constraints or to the variables. This is done statically by [28], who define a smaller model whose size depends on a given parameter. An interesting conclusion of [28] is that small values of this parameter are often sufficient to obtain excellent bounds. However this method is static and does not converge toward the optimum of the initial model.

In the context of column generation methods, similar aggregations can be obtained by adding equality constraints on dual variables of a same group. This has the advantage to reduce the size of the pricing sub-problem (since all dual variables of the group can be exchanged), and also to stabilize the column generation process. This strategy is useful if many dual variables are “equivalent” and thus many of them can have similar values at optimality. Dual optimal and deep dual optimal inequalities [1] are examples of dual constraints that do not separate all optimal dual solutions.

Dynamic constraint aggregation has been introduced in the context of column generation in [11, 10, 1]. In the aggregated model, each constraint represents a group of original partitioning constraints. From a dual space perspective, a dual variable of the aggregated
model represents a group of original dual variables, and therefore its value is equal to the sum of the actual dual values in the group. When the pricing subproblem is called, the dual variables are disaggregated and the initial subproblem is solved. The column produced by the subproblem is added to the RMP if it is compatible with the current groups, or put aside otherwise.

In the above approach, a compatible column corresponds to a configuration that either covers all elements of a given group or none of them. The elements of a given group are not distinguished in the aggregated model, i.e., they are in the same equivalence class and they are all associated to a unique representative partitioning constraint. For some specific problems, such Cutting-Stock, we are given items with different weights (resource consumptions); if the weights of two items are different, they might have significantly different dual values at optimality, and so, they can hardly be considered equivalent. In such situations, equivalence-based approaches might require too many (small) groups to obtain good approximations. An extreme case is obtained when each group must contain exactly one element, even if the structure of the solution is simple (for example the dual values can be obtained by applying a linear function to the item sizes).

In certain cases, the dual values may often follow a piecewise-linear function of the associated resource consumptions. This type of solution can be obtained by a new type of aggregation, where dual variables in the same group have values that follow a linear function of this resource. This is what we intend to do in the remainder of this paper. More generally, elements with higher resource consumptions arise in less valid configurations, and so, they can be associated to larger dual values; for cutting-stock, the dual solution vector is always non-decreasing (see for example [6]).

3. Iterative Inner Dual Approximation (2IDA)

This section describes the iterative aggregation method, starting from a general overview and gradually addressing all technical details. From now on, we will focus on the the dual polytope $P$, i.e. we consider CG as a cutting-plane algorithm for optimizing (2.2).

3.1. Full Description of the Aggregated Dual Polytope

Let $G_k = \{I^1, I^2, \ldots, I^k\}$ be a partition of $I = \{1, 2 \ldots n\}$ into $k$ groups, each group having $n_j = |I^j|$ elements. Given elements $I^j$ of a group $j \in [1..k]$, let $y^j$, $w^j$, $b^j$ and $a^j$ denote the $n_j$-dimensional column vectors related to dual variables, resource consumptions, demands
and, respectively, coefficients of some configuration $a \in A$. Variables in the dual vector are ordered in such a way that $y$ is the concatenation of $k$ group components: $y^\top = [y_1 \ y_2 \ldots \ y_n] = [(y^1)^\top \ldots (y^k)^\top]$.

We artificially impose the following linearity restriction: given any group $j \in [1..k]$, the values of the dual variables $y^j_1, y^j_2, \ldots y^j_n$ have to follow a linear function of the resource consumption of the corresponding element. This function depends on a slope $\alpha^j$ and a $y$-intercept $\beta^j$. More precisely, the $i^{th}$ component of $y^j$ can be written as $y^j_i = w^j_i \alpha^j + \beta^j$.

By simply adding these constraints to (2.2), we obtain the inner dual polytope in $\mathbb{R}_+^n \times \mathbb{R}^{2k}$:

$$\max b^\top y$$

$$a^\top y \leq 1, \quad \forall a \in A$$

$$y^j_i = w^j_i \alpha^j + \beta^j, \quad \forall j \in [1..k], i \in I^j$$

$$y^j_i \geq 0, \quad \forall j \in [1..k], i \in I^j$$

$$\alpha^j, \beta^j \in \mathbb{R}, \quad \forall j \in [1..k]$$

**Remark 3.1.** By extracting the vector $y$ from any valid solution of above (3.1), we obtain a valid solution of $P$ in (2.2). The opposite is not true in general: only vectors $y$ of $P$ with the suitable groupwise linear structure can be lifted to valid solutions of (3.1).

Let us re-write (3.1) only using decision variables $\alpha^j$ and $\beta^j$. We first re-write the objective function. For each group $j$, $y^j$ can be written as a linear combination of $w^j$ and $1_j$ (vector $[1 \ 1 \ldots \ 1]^\top$ with $n_j$ elements): $y^j = w^j \alpha^j + 1_j \beta^j$. Observe that

$$b^\top y = \sum_{j=1}^k (b^j)^\top y^j = \sum_{j=1}^k (b^j)^\top (w^j \alpha^j + 1_j \beta^j) = \sum_{j=1}^k \left((b^j)^\top w^j\right) \alpha^j + \left((b^j)^\top 1_j\right) \beta^j$$

The first set of constraints from (3.1) can be written $\sum_{j=1}^k (a^j)^\top y^j \leq 1, a \in A$. For each $j$, we have:

$$(a^j)^\top y^j = (a^j)^\top (w^j \alpha^j + 1_j \beta^j) = \left((a^j)^\top w^j\right) \alpha^j + \left((a^j)^\top 1_j\right) \beta^j$$

We are now ready to express model (3.1) with variables $\alpha^j$ and $\beta^j$ only. To simplify the notation, we introduce the following definitions.

**Definition 3.1.** Given a configuration $a \in A$ and a group $j$, we define:

- $c^j_a = (a^j)^\top w^j$: total resource consumption of the elements of $I^j$ selected in $a$. This is the coefficient of variable $\alpha^j$ in (3.3);
– \( N^j_a = (a^j)^\top 1_j \): total number of elements of \( I^j \) selected in \( a \). This is the coefficient of variable \( \beta^j \) in (3.3);

– \( w^j_{\text{min}} \) and \( w^j_{\text{max}} \): the lowest and respectively highest resource consumption in \( I^j \).

By simply substituting (3.2)-(3.3) in model (3.1), after handling the non-negativity constraints, we obtain an equivalent model in the space \( \mathbb{R}^{2k} \):

\[
\begin{align*}
\max \sum_{j=1}^{k} \left( (b^j)^\top w^j \right) \alpha^j + \left( (b^j)^\top 1_j \right) \beta^j \\
\sum_{j=1}^{k} e^j_a \alpha^j + N^j_a \beta^j \leq 1, \forall a \in A \\
w^j_{\text{min}} \alpha^j + \beta^j \geq 0, \forall j \in [1..k] \\
w^j_{\text{max}} \alpha^j + \beta^j \geq 0, \forall j \in [1..k] \\
\alpha^j, \beta^j \in \mathbb{R}, \forall j \in [1..k]
\end{align*}
\]  

(3.4)

**Proposition 3.1.** There is a bijection between the set of feasible solutions 
\([y_1, y_2, \ldots, y_n, \alpha^1, \ldots, \alpha^k, \beta^1, \ldots, \beta^k]\) of model (3.1) in \( \mathbb{R}^n_+ \times \mathbb{R}^{2k} \) and the feasible solutions 
\([\alpha^1, \ldots, \alpha^k, \beta^1, \ldots, \beta^k]^\top \) of model (3.4) in \( \mathbb{R}^{2k} \).

**Proof:** Recall that (3.1) has by construction the constraint \( y^j_i = w_i \alpha^j + \beta^j \). The sought bijection simply maps any \([y^1, \ldots, y^n, \alpha^1, \ldots, \alpha^k, \beta^1, \ldots, \beta^k]^\top \in \mathbb{R}^n_+ \times \mathbb{R}^{2k} \) that is valid in (3.1) to a solution \([\alpha^1, \ldots, \alpha^k, \beta^1, \ldots, \beta^k]^\top \in \mathbb{R}^{2k} \) that is valid in (3.4). The non-negativity constraints of (3.1) are simplified in (3.4): \( y^j_i = w^j_i \alpha^j + \beta^j \geq 0, \forall i \in I^j \iff w^j_{\text{min}} \alpha^j + \beta^j, w^j_{\text{max}} \alpha^j + \beta^j \geq 0 \), as \( w^j_i \alpha^j \in [w^j_{\text{min}} \alpha^j, w^j_{\text{max}} \alpha^j] \forall i \in I^j \), for any \( j \in [1..k] \). \( \square \)

The new formulation (3.4) has \( 2k \) real variables that can be either positive or negative. In this basic version, only the number of variables is decreased, which can be very useful for stabilization reasons in CG: by using less dual variables, the oscillation potential is reduced. Also, less dual variables corresponds to less constraints in the corresponding Reduced Master Problems (RMPs), which can accelerate the LP solver for RMPs.

However, a drawback of (3.4) consists of the fact that it has the same (prohibitively large) number of constraints as (2.2). We address this point in the next section, introducing a new reduced model (3.5) that removes many constraints that are redundant in (3.4).

### 3.2. Reduced Model and Aggregated Column Generation

We first show below in Section 3.2.1 that the new linearity restrictions render many configurations of \( A \) redundant in (3.4). This leads to a new model (3.5) that reduces the size of the initial set-covering LP in terms of both columns and rows. Our method actually optimizes the new model (3.5); the computational aspects are described in Section 3.2.2.
3.2.1. Removing Redundant Constraints in Model (3.4) The main “≤” constraints of (3.4) only make use of coefficients $N_a^j$ and $c_a^j$ (see also Definition 3.1), but they does not explicitly use all $n$ positions of configurations $a \in \mathcal{A}$. To reduce the model, we no longer express configurations $a \in \mathcal{A}$ as vectors in $\mathbb{Z}^n_+$, but as aggregated $\mathbb{Z}_2^k$ vectors of the form $\bar{a} = [c_a^1 N_a^1, c_a^2 N_a^2, \ldots, c_a^k N_a^k]$. We say that such aggregated configuration $\bar{a}$ can be lifted to a configuration $a \in \mathcal{A}$ if $c_a^j = (a_j^1) \top w_j$ and $N_a^j = (a_j^1) \top 1_j$, for all $j \in [1..k]$.

There might exist more than one disaggregated configuration $a \in \mathcal{A}$ that corresponds to an aggregated configuration $\bar{a}$, i.e., more configurations $a$ might generate the same values $c_a^j, N_a^j, \forall j \in [1..k]$. Therefore, many constraints of $\mathcal{P}_k$ in (3.4) are redundant.

**DEFINITION 3.2.** Given group $j \in [1..j]$, the set $R^j$ of feasible resource consumptions is:

$$R^j = \{ c^j \in [0..C^+] : \exists a \in \mathcal{A} \text{ such that } c_a^j = c^j \}$$

**DEFINITION 3.3.** Given a feasible consumption $c^j \in R^j$, we define the minimum and maximum cardinality coefficients for group $j$ and consumption $c^j$ (see also Definition 3.1):

- $N^+(j, c^j)$: the maximum number of $I^j$ elements in a valid configuration $a \in \mathcal{A}$ in which the total resource consumption of the $I^j$ elements is $c^j$;
- $N^-(j, c^j)$: the minimum number of $I^j$ elements in a valid configuration in which the total resource consumption of the $I^j$ elements is $c^j$.

**DEFINITION 3.4.** Let $\mathcal{A}_k$ be the set of dominant (non-redundant) configurations, defined by:

$$\mathcal{A}_k = \{ a \in \mathcal{A} , \text{ such that } \forall j \in [1..j], N_a^j \in \{ N^+(j, c_a^j), N^-(j, c_a^j) \} \}$$

By replacing $\mathcal{A}$ with $\mathcal{A}_k$ in model (3.4), we obtain a new model:

$$\begin{aligned}
\max & \sum_{j=1}^k \left( (b_j^1) \top w_j^1 \right) \alpha_j + \left( (b_j^1) \top 1_j \right) \beta_j \\
\sum_{j=1}^k c_a^j \alpha_j + N_a^j \beta_j & \leq 1, \forall a \in \mathcal{A}_k \\
w_{\min}^j \alpha_j + \beta_j & \geq 0 \quad \forall j \in [1..k] \\
w_{\max}^j \alpha_j + \beta_j & \geq 0 \quad \forall j \in [1..k] \\
\alpha_j, \beta_j & \in \mathbb{R} \quad \forall j \in [1..k]
\end{aligned}$$

(3.5)

Obviously, if two configurations $a, a' \in \mathcal{A}$ lead to the same values of $N_a^j$ and $c_a^j$, only one constraint has to be explicitly considered. The new optimization problem will be hereafter referred to as the Dual Set-Covering Problem (DSCP) over polytope $\mathcal{P}_k$: $\text{DSCP}(\mathcal{P}_k) =$
\[
\max \{ \sum_{j=1}^{k} ((b_j^i)^\top w_j) \alpha_j + ((b_j^i)^\top 1_j) \beta_j : [\alpha^1 \ldots \alpha^k, \beta^1 \ldots \beta^k]^\top \in P_k \}.
\]

We will show below that (3.5) determines the same polytope \( P_k \) as (3.4), and so, any solution of (3.5) can be lifted to a solution of the initial LP (2.2).

**Proposition 3.2.** Any solution of (3.5) can be lifted to a solution that is feasible for the initial set-covering model (2.2).

**Proof:** Remark 3.1 (p. 8) states that any valid solution of (3.1) can be projected into a valid solution of the initial dual polytope \( P \) in (2.2). Proposition 3.1 (p. 9) shows there is a bijection between the valid solutions of (3.1) and (3.4). It is enough to show that (3.4) is equivalent to (3.5). Using the notations from Definitions 3.1 and 3.3, we observe that, given any valid \( a \in A_k \), for each \( j \in [1..k] \), the values of variables \( \alpha_j \) and \( \beta_j \) have to respect one of the two inequalities below:

1. \( c_d^j \alpha_j + N_d^j \beta_j \leq c_d^j \alpha_j + N^+(j, c_d^j) \beta_j, \) if \( \beta_j \geq 0 \);
2. \( c_d^j \alpha_j + N_d^j \beta_j \leq c_d^j \alpha_j + N^-(j, c_d^j) \beta_j, \) if \( \beta_j < 0 \).

Any \( \leq \) constraint of (3.4) obtained from a configuration outside \( A_k \) is dominated by a \( \leq \) constraint obtained from some \( a \in A_k \), i.e., with \( N_d^j = N^+(j, c_d^j) \) or \( N_d^j = N^-(j, c_d^j) \). All configurations outside \( A_k \) are dominated, and so, (3.4) is equivalent to (3.5). \( \square \)

To solve DSCP(\( P_k \)), one need all cardinality coefficients \( N^+(j, c^j), N^-(j, c^j), \forall c^j \in R^j, j \in [1..k] \). This is performed in a coefficient calculation preprocessing stage which is executed only once for each group \( j \in [1..k] \). The goal is to find the maximum and minimum number of elements that can be selected from \( I^j \) so as to consume a total resource amount of \( c^j, \forall c^j \in R^j \). This task is similar to the pricing sub-problem (2.3) for the non-aggregated problem (Section 2.1). If the Dynamic Programming (DP) scheme for (2.4) is fast, so is the DP for this coefficient calculation. Indeed, it is enough to replace \( y_i \) with 1 in (2.4) and to apply an analogous DP twice, i.e., once with a minimization and once with a maximization objective. Such a DP scheme generates by default a state for each feasible \( c^j \) value, i.e., the set \( R^j \) is constructed at the same time. Considering all groups \( j \in [1..k] \) together, this coefficient calculation stage requires the same complexity as one pricing sub-problem for the initial model (2.2).

**Remark 3.2.** The number of constraints in (3.5) is exponentially large in \( k \), but not in \( n \). Computing all constraints of \( P_k \) would thus require an asymptotic running time that is exponentially large in \( k \) and polynomial in \( C^+ \) (to determine all \( N^+(j, c^j) \) and
If we consider that $k$ and $C$ are bounded constants, it is possible to achieve the full construction of $\mathcal{P}_k$ in polynomial time. This would lead to a \textit{pseudo-polynomial dual bound} for the initial combinatorial optimization problem. Although such direct $\mathcal{P}_k$ constructions are used in this paper only for $k = 1$, the main idea of a direct construction represents a promising subject for further work.

### 3.2.2. An aggregated Pricing Subproblem

The model (3.5) is optimized by column generation for all values of $k > 1$. A computational difficulty is related to the pricing algorithm. In a straightforward approach, one could disaggregate the variables to obtain the dual solution $y$ in the original space in $\mathbb{R}^n_+$; then, the pricing algorithm for the non-aggregated model (Section 2.1) can be used. This has the disadvantage of considering a solution space with many symmetries and redundancies, since all $y^j_i$ with $i \in [1..n_j]$ in a group $j \in [1..k]$ correspond to a single pair of dual variables $\alpha^j$ and $\beta^j$.

We now show how the pricing can be solved without disaggregation. Iteratively, for each current solution $[\alpha^1 \ldots \alpha^k, \beta^1 \ldots \beta^k]^\top \in \mathcal{P}_k$, the column generation algorithm needs to solve an aggregated version of the sub-problem (2.3): find the aggregated configuration $\bm{a} = [c^1 \ N^1, c^2 \ N^2, \ldots, c^k \ N^k]$ that maximizes the profit value $\sum_{j=1}^k \alpha^j c^j + \beta^j N(j, c^j)$. As for the non-aggregated pricing (2.3), a column of negative reduced cost is identified whenever this profit is larger than 1. Formally, the aggregated pricing problem is the following:

\[
\begin{align*}
\max & \quad \sum_{j=1}^k \alpha^j c^j + \beta^j N(j, c^j) \\
\text{s.t.} & \quad C^- \leq \sum_{j=1}^k c^j \leq C^+ \\
& \quad N(j, c^j) = \begin{cases} 
N^+(j, c^j) & \text{if } \beta^j \geq 0 \\
N^-(j, c^j) & \text{if } \beta^j < 0
\end{cases} \quad \forall j \in [1..k], \\
& \quad c^j \in R^j, \forall j \in [1..k]
\end{align*}
\]

where $R^j$ is the set of feasible resource consumptions (Definition 3.2). Recall that all sets $R^j$ (with $j \in [1..n]$) can be computed by the coefficient calculation preprocessing stage (Section 3.2.1), using the same complexity as one pricing call for the original problem (2.2).

The only decision variables in (3.6) are $c^j \in R^j$, $\forall j \in [1..j]$. They are sufficient to represent a solution for this pricing sub-problem, since any $N(j, c^j)$ term can be deduced from $c^j$ and $\beta^j$. All $\beta^j$ represent input data: we choose from the beginning to use either $N(j, c^j) = N^+(j, c^j)$ or $N(j, c^j) = N^-(j, c^j)$, depending on the sign of $\beta^j$. Let us denote $p(j, c) =$
\[ \alpha_j + \beta_j N(j,c) \] the potential profit that can be obtained with a resource amount of \( c \) for group \( j \). The pricing problem (3.6) is reformulated:

\[
\begin{align*}
\max & \quad \sum_{j=1}^{k} \sum_{c \in R_j} p(j,c) x_{jc} \\
\text{s.t.} & \quad C^{-} \leq \sum_{j=1}^{k} \sum_{c \in R_j} c^j x_{jc} \leq C^{+}, \\
& \quad \sum_{c \in R_j} x_{jc} = 1, \forall j \in [1..k] \\
& \quad x_{jc} \in \{0,1\}, \forall j \in [1..k], c \in R^j
\end{align*}
\] (3.7)

where decision variable \( x_{jc} \) is 1 only when group \( j \) uses a total resource amount of \( c \).

The resulting aggregated pricing is a multiple-choice variant of the non-aggregated pricing from Section 2.1. For instance, if the non-aggregated problem is the knapsack problem, the aggregated one is the multiple-choice knapsack. Generally speaking, the non-aggregated Dynamic Program (DP) approach from Section 2.1 can be extended to an aggregated DP as follows. We associate a DP state to each total consumption value in \([1..C^+]\). For each level \( j \in [1..k] \), one can scan all feasible choices (total consumptions values \( c^j \)) to generate (or update) other states. We will see that in certain cases, the aggregated DP only needs to perform these computations on two levels associated to new sub-groups \( j_1, j_2 \in [1..k] \) (see Section 3.6), which is very fast in practice. However, in this aggregated pricing, the number \( n_b = \sum_{i=1}^{n} b_i \) of individualized elements is no longer a factor in the asymptotic or practical running time, once the initial preprocessing (Section 3.2.1) is performed.

### 3.3. The Iterative Algorithm and the Incremental \( \mathcal{P}_k \) Optimization

Section 3.2 described an aggregation-based column generation method that optimizes (3.5) for a given fixed partition \( G_k = \{I^1, I^2, \ldots, I^k\} \). The resulting value, hereafter noted \( \text{lb}_{G_k} \) (or simply \( \text{lb}_k \) when the exact structure of \( G_k \) is not essential) represents a lower bound for the sought \( \text{OPT}_{CG} \). Algorithm 1 below presents the general steps of our Iterative Inner Dual Approximation (2IDA) method. The main idea is to iteratively break the groups into smaller subgroups and incrementally refine the description of \( \mathcal{P}_k \).

The partition \( G_k \) of the dual variables is initialized with one group equal to \( I \). Right after that (Line 2), 2IDA applies the preprocessing stage from Section 3.2.1 to determine the cardinality coefficients (e.g., \( N^-(j_1, c_1) \), \( N^+(j_1, c_1) \), etc.) required to describe all aggregated polytopes. Routine \( \text{aggregCGoptim}(\mathcal{P}_k) \) uses partition \( G_k \) to restrict and project the dual polytope \( \mathcal{P} \) to dimension \( \mathbb{R}^{2k} \), obtaining (3.5) as described in Section 3.2.1. For
Algorithm 1: Iterative Inner Dual Approximation

1. $k \leftarrow 1$, $G_1 \leftarrow \{I\}$
2. $P_k \leftarrow \text{det}P_k\text{Coefs}(G_k)$  \hspace{1em} // preprocessing coefficient calculation (Sec. 3.2.1)
3. repeat
4. \hspace{1em} $lb_{G_k} \leftarrow \text{aggregCGoptim}(P_k)$  \hspace{1em} // Sec. 3.2.2 (or full $P_k$ construction for $k=1$)
5. \hspace{1em} $ub_{G_k} \leftarrow \text{upBound}(P_k, lb_{G_k})$  \hspace{1em} // optional upper bound (Section 3.5) of OPT\_CG
6. \hspace{1em} if $\lfloor lb_{G_k} \rfloor < \lceil ub_{G_k} \rceil$ then
7. \hspace{2em} $G_{k+1} \leftarrow \text{grpSplit}(G_k)$  \hspace{1em} // split a group and refine partition (Sec. 3.4)
8. \hspace{2em} $P_{k+1} \leftarrow \text{det}P_k\text{Coefs}(G_{k+1} - G_k)$  \hspace{1em} // determine coefficients for the sub-groups
9. \hspace{2em} $P_{k+1} \leftarrow \text{inheritCnstr}(P_k, G_{k+1})$  \hspace{1em} // inherit $P_k$ constraints into $P_{k+1}$ (Sec. 3.6)
10. \hspace{1em} $k \leftarrow k + 1$
11. until ($\lfloor lb_{G_k} \rfloor = \lceil ub_{G_k} \rceil$) or a stopping condition is reached

$k > 1$, \text{aggregCGoptim}(P_k) \text{ relays on a column generation scheme that iteratively solves the aggregated pricing sub-problem from Section 3.2.2.}$

After determining $lb_{G_k}$, Algorithm 1 can continue by calculating an optional upper bound $ub_{G_k}$ (routine $\text{upBound}(P_k, lb_{G_k})$, Line 5). For this, we propose (Section 3.5) a method that performs disaggregated CG steps, using a polytope $P_u$ that contains much fewer constraints than $P$. While upper bounds are not mandatory, they can be useful for “tail cutting” (i.e., to stop 2IDA if $\lfloor lb_{G_k} \rfloor = \lceil ub_{G_k} \rceil$). Furthermore, the specific upper bound described in Section 3.5 is very appropriate for guiding group split decisions (see below).

Section 3.4 presents several methods for taking group split decisions (call $\text{grpSplit}(G_k)$ in Line 7). The goal is to determine a group $j^* \in [1..k]$ to be split into sub-groups $j_1$ and $j_2$, so as to advance from $k$ to $k+1$. This decision can be taken using information related to the upper bound (if available), problem-specific data, or general ad-hoc indicators (group sizes, resource consumption spreads, etc.). After calling $\text{grpSplit}(G_k)$, 2IDA applies (in Line 8) the coefficient calculation preprocessing stage (Section 3.2.1), but only restricted to the new sub-groups $j_1$ and $j_2$.

The configurations (constraints) generated for a given $k$ are not completely removed when 2IDA passes to $k+1$. For practical purposes, we need to perform an incremental description of the polytope sequence $P_1$, $P_2$, $P_3$, etc. Instead of constructing each $P_{k+1}$
from scratch, 2IDA generates some initial $P_{k+1}$ constraints by lifting (inheriting) certain constraints generated when optimizing over $P_k$. Indeed, $\text{inheritCnstr}(P_k, G_{k+1})$ makes use of some $P_k$ constraints to warm-start the column generation optimization of $\text{DSCP}(P_{k+1})$, see Section 3.6 for a deeper description of this “inheritance” process.

3.4. Split Decisions

2IDA converges towards the optimum of the initial problem for any group split decision. However, from a practical perspective, the split decision is a crucial part for the efficiency of the method. In many optimization settings, a fast convergence is obtained by taking each time a decision that leads to the best local improvement. However, the split decision that leads to the best improvement for a given step $k$ can only be known after optimizing over $P_{k+1}$. Trying all possible split decisions and optimizing all possible $\text{DSCP}(P_{k+1})$ would definitely require prohibitive calculation time. We present below the main principles used throughout the paper to take fast and effective splitting decisions.

3.4.1. Splitting Groups Based on a Binary Decomposition

The most general splitting approach only works with regular interval lengths. To determine the group to be split, a reasonable choice consists of selecting a group $j^*$ with maximum resource consumption spread, i.e., such that $w_{j^*}^{\max} - w_{j^*}^{\min} = \max(w_j^{\max} - w_j^{\min})$. This group $j^*$ is split into subgroups $j_1$ and $j_2$ using a split point $c^* \in [w_{j^*}^{\min}..w_{j^*}^{\max}]$ that separates the elements of $I_{j^*}$ as follows: $I_{j_1} = \{i \in I_{j^*} : w_i \leq c^*\}$ with $n_{j_1} = |I_{j_1}|$ and $I_{j_2} = \{i \in I_{j^*} : w_i > c^*\}$ with $n_{j_2} = |I_{j_2}|$. To maintain the regularity of the intervals sizes, this split point is simply determined via $c^* = \frac{t}{2z}C^+$, where $z$ is the minimum integer that can lead to $n_{j_1}, n_{j_2} \geq 1$ for some $t \in \mathbb{Z}_+$; after determining $z$, we choose a $t$ value so as to minimize $|n_{j_1} - n_{j_2}|$. This basic method is general, but does use any information from the current step of the optimization.

3.4.2. Group-Splitting Following an Outside Reference Solution

The split method from this section aims at removing $P_k$ constraints that block the optimum of $\text{DSCP}(P_k)$ in (3.5) from advancing towards an outside reference solution of better objective value.

**Definition 3.5.** A solution $y^u \in \mathbb{R}_n^+$ is an outside reference solution for $\mathcal{P}$ if $b^\top y^u \geq \text{OPT}_{CG}$, i.e., $y^u$ is associated to an upper bound of $\text{OPT}_{CG}$. This solution is either the optimal solution of (2.2), or belongs to the exterior of $\mathcal{P}$.

Let $y_k^*$ be the optimal solution of $\text{DSCP}(P_k)$ lifted in the original space $\mathbb{R}_n^+$. We focus on the differences between $y_k^*$ and a given outside reference solution $y^u$. It could be very
useful to remove the $P_k$ constraints that separate $y^*_k$ from $y^u$, and, in the first place, those verified with equality by $y^*_k$. This could allow the solution $y^*_{k+1}$ of the next iteration to be closer to $y^u$ than $y^*_k$. We introduce the notion of open direction.

**Definition 3.6.** Given an outside reference solution $y^u$ and a solution $y \in P_k$, we say that $y \rightarrow y^u$ is an open direction if there exists $\epsilon > 0$ such that $y^*_k = y + \epsilon(y^u - y) \in P$.

In concrete terms, there is a feasible solution $y_\epsilon$ different from $y$ that can be obtained by convex combination of $y$ and $y^u$. We always use outside reference solutions $y^u$ generated from the current $y^*_k$ by the upper bounding method from Section 3.5. We will later prove (Proposition 3.3) that such reference solutions $y^u$ are associated to open directions $y^*_k \rightarrow y^u$ (unless $b^\top y^*_k = \text{OPT}_{\text{CG}}$, in which case $b^\top y^u = \text{OPT}_{\text{CG}}$ and 2IDA terminates).

Let us focus on the constraints of $P_k$ that separate $y^*_k$ from $y^u = y^*_k + \epsilon(y^u - y)$ for a small enough $\epsilon$. Since $y_\epsilon$ belongs to $P$, $y^*_k$ and $y_\epsilon$ are separated by a set of $P_k$ inequalities that do not belong to $P$. Such inequalities can only arise by combining valid $P$ constraints with group-wise linearity restrictions associated to $P_k$. We will have more to say about this process in Section 4, but for the now it is enough to insist on the following. Dropping these non-valid restrictions could drive 2IDA towards $y^u$, improving the objective value by at least some $\epsilon b^\top (y^u - y)$.

Therefore, a split decision should select a group $j^*$ that is associated to group-wise linear restrictions separating $y^*_k$ from $y^u$. Since non-valid $P_k$ constraints spring from enforcing the dual values in each group to follow a linear function, we consider groups whose dual values in $y^u$ do not follow a linear function. Based on such information, we determine $j^*$ based on evaluating how far $y^u$ is from a linear function over each $j \in [1..k]$. A specific formula example is provided in Section 5.3, *i.e.*, function $h_3$ in the cutting stock application.

### 3.5. Upper Bounding and Open Directions based on the $P_k$ Description

The method proposed in this section takes $y^*_k$ (optimal solution of DSCP($P_k$) expressed in dimension $\mathbb{R}^n_+$) as input and returns an upper bound $b^\top y^u$ associated to an outside reference solution $y^u$ such that:

1. if $b^\top y^*_k < \text{OPT}_{\text{CG}}$, the direction $y^*_k \rightarrow y^u$ is open
2. if $b^\top y^*_k = \text{OPT}_{\text{CG}}$, then $b^\top y^u = \text{OPT}_{\text{CG}}$.

The first property is useful for guiding the group splitting heuristic from Section 3.4.2 above. It means that $P$ contains a linear combination of $y^*_k$ and $y^u$ that is better than $y^*_k$. The second property allows to stop the 2IDA iterative process as soon as $b^\top y^*_k = \text{OPT}_{\text{CG}}$. 
3.5.1. Theoretical Modelling: the Polytope $P^u$ of $y_k^*$-Tight Constraints

Given the optimal solution $y_k^* \in \mathbb{R}_+^d$ of OPT(DSCP($P_k$)) at current step $k$, we determine $y^u$ by optimizing the dual set-covering problem associated to polytope $P^u \supset P$ defined by:

$$P^u = \{ y \in \mathbb{R}_+^d : a^T y \leq 1, \forall a \in A \text{ such that } a^T y_k^* = 1 \} \quad (3.8)$$

We say that $P^u$ is the polytope of the $y_k^*$-tight constraints of $P$. It is simply obtained from $P$ by only keeping the constraints that are satisfied to equality by $y_k^*$.

**Proposition 3.3.** A solution $y^u$ such that $b^T y^u = \max\{ b^T y : y \in P^u \} = OPT(DSCP(P^u))$, where $P^u$ is defined by (3.8), satisfies the following properties:

1. **Upper bounding:** $OPT_{CG} \leq b^T y^u$;
2. **Open direction:** if $y_k^*$ is not an optimal solution of DSCP($P$), then $b^T y_k^* < b^T y^u$ and $y_k^* \rightarrow y^u$ is an open direction;
3. **Optimality proving:** if $b^T y_k^* = OPT_{CG}$, then $y^u$ also satisfies $b^T y^u = OPT_{CG}$.

**Proof:** Property (1) actually follows from the $P^u$ definition (3.8). Since $P^u$ is constructed from a subset of the constraints of $P$, we directly have $P^u \supset P$. The optimum (of objective function $b^T y$) over $P^u$ dominates the optimum over $P$, and so, $b^T y^u \geq OPT_{CG}$.

We now prove (2). First, the fact that $y_k^*$ is not a optimal shows directly that $b^T y_k^* < OPT_{CG} \leq b^T y^u$. Then, suppose $y_k^* \rightarrow y^u$ is not open, i.e., $y_k^* + \epsilon(y^u - y_k^*) \notin P, \forall \epsilon > 0$. This would imply the existence of some $a \in A$ such that $a^T y_k^* \leq 1$ and $a^T y_k^* + \epsilon a^T (y^u - y_k^*) > 1, \forall \epsilon > 0$. But this could only happen if $a^T y_k^*$ would be exactly 1, i.e., this would lead to a $y_k^*$-saturated constraint violated by $y^u$, which is impossible.

The last property (3) comes from the construction of $P^u$. If $y_k^*$ is optimal in $P$, its basic constraints need to be $y_k^*$-tight, and so, they belong to $P^u$. These constraints are sufficient to ensure the optimality of $y^u$ in $P$.

3.5.2. Practical Aspects: Constructing $P^u$ We solve DSCP($P^u$) by column generation. DSCP($P^u$) has exactly the same structure as DSCP($P$) in (2.2), but the set of possible constraints is restricted to those that are $y_k^*$-tight. The pricing sub-problem requires finding a configuration $a \in A$ with $a^T y_k^* = 1$ ($y_k^*$-tightness) that maximizes $y^T a$ for the current dual solution $y$. This pricing sub-problem can be solved exactly as the initial pricing sub-problem (Section 2.1), by replacing $\max_{a \in A} y^T a$ in (2.3) with a hierarchical objective function $\max_{a \in A} (My_k^* + y)^T a$, where $M$ is a sufficiently large value.
In practice, the same behavior is obtained by lexicographic maximization: first maximize \( \mathbf{a}^\top \mathbf{y}_k^* \), and, subject to this, maximize \( \mathbf{a}^\top \mathbf{y} \). The original dynamic programming scheme (Section 2.1) is not modified and the number of states is the same. Instead of computing a unique profit function \( P_{\text{max}} \) in the recursion (2.4), the lexicographic dynamic program computes an ordered pair of profit functions \( (P_{\text{max}}^*, P_{\text{max}}^u) \). For each state \((c, i) \in [0, C^+] \times [1..n]\), \( P_{\text{max}}^*(c, i) \) and \( P_{\text{max}}^u(c, i) \) represent the maximum of \( (\mathbf{y}_{\text{max}}^*)^\top \mathbf{a} \) and, respectively, of \( \mathbf{y}^\top \mathbf{a} \) over all (partial) configurations \( \mathbf{a} \) in state \((c, i) \), i.e., consuming a resource amount of \( c \) only using elements \([1..i]\). The recursion (2.4) evolves to a lexicographic maximization formula.

**Proposition 3.4.** Given a dual solution \( \mathbf{y} \) and the DSCP(\( P_k \)) optimum \( \mathbf{y}_k^* \) in dimension \( \mathbb{R}^n \), by lexicographically optimizing profits \( (\mathbf{y}_{\text{max}}^*)^\top \mathbf{a}, \mathbf{y}^\top \mathbf{a}) \), the above dynamic program determines a configuration \( \mathbf{a} \in \mathcal{A} \) that maximizes \( \mathbf{y}^\top \mathbf{a} \) subject to \((\mathbf{y}_k^*)^\top \mathbf{a} = 1 \).

**Proof:** The evolution of the lexicographic dynamic program can be interpreted as follows: at each recursion step, the lexicographic maximization formula actually maximizes \( (\mathbf{y}_k^*)^\top \mathbf{a} \), breaking ties using the value of \( \mathbf{y}^\top \mathbf{a} \). As such, the \( \mathbf{y} \) objective is only used for tie breaking, and so, the lexicographic dynamic program returns a configuration \( \mathbf{a}^* \) maximizing the \( \mathbf{y}_k^* \) objective. As such, \((\mathbf{y}_k^*)^\top \mathbf{a}^* \) reaches a maximum of 1, because at least some \( \leq 1 \) constraints in (3.5) need to be saturated by \( \mathbf{y}_k^* \) (otherwise, no other (3.5) constraint can render infeasible a higher value solution such as \( \mathbf{y}_k^* + \epsilon \mathbf{1}_n \), with a small-enough \( \epsilon \)). \( \square \)

Consequently, the complexity of the pricing for DSCP(\( P^u \)) is the same as the complexity of the dynamic program pricing for DSCP(\( P \)). However, since \( P^u \) is defined by a far smaller number of constraints than \( P \), the proposed upper bounding method is generally faster.

**3.6. Streamlined \( P_{k+1} \) Column Generation: Using \( P_k \) for Warm-Starting**

After solving DSCP(\( P_k \)) to optimality at iteration \( k \), Alg. 1 splits a group \( j^* \) and generates two new (sub-)groups \( j_1 \) and \( j_2 \). A new program (3.5), associated with a new polytope \( P_{k+1} \) has to be optimized. First, the preprocessing stage from Section 3.2.1 is called on \( j_1 \) and \( j_2 \) (Line 8) to calculate the new cardinality coefficients (e.g., \( N^-(j_1, c^{j_1}), N^+(j_1, c^{j_1}) \)). Then, one could optimize over \( P_{k+1} \) by applying from scratch the CG from Section 3.2.2, but this may require a too large computational cost. Therefore, we use the columns already generated at iteration \( k \) to warm-start the CG phase at the next iteration \( k + 1 \) (Line 9). The main goal is to perform an incremental iterative description of the polytopes \( P_1, P_2, \ldots \).
Indeed, the first \( P_{k+1} \) constraints are generated by lifting existing \( P_k \) constraints from dimension \( \mathbb{R}^{2k} \) to dimension \( \mathbb{R}^{2k+2} \) (see below). The optimal solution of \( DSCP(P_k) \) is also lifted to construct an initial feasible solution for \( P_{k+1} \), i.e., we apply \( \alpha^{j_1} = \alpha^{j_2} = \alpha^* \) and \( \beta^{j_1} = \beta^{j_2} = \beta^* \) for \( j_1 \) and \( j_2 \), and keep unchanged all other \( \alpha \) and \( \beta \) positions. Starting from this \( P_{k+1} \) solution, 2IDA first applies CG to optimize over an intermediate polytope \( P'_{k+1} \supset P_{k+1} \) (see below) containing inherited constraints only. If the lifted optimal solution of \( DSCP(P_k) \) is optimal in \( DSCP(P'_{k+1}) \), there is no need to continue optimizing over \( P_{k+1} \).

More formally, we note \( A_{k}^{\text{base}} \) the set of configurations associated to constraints of \( P_k \) that are either related to basic primal variables, or non-basic primal variables of reduced cost 0 (with regards to the \( P_k \) optimal solution). We consider only the constraints that have been explicitly generated when optimizing \( P_k \) in (3.5). Our approach is equivalent to first constructing a dual polytope \( P'_{k+1} \) only containing inherited constraints associated to: 

\[
A_{k+1}' = \{ a' \in A_{k+1} : \exists a \in A_{k}^{\text{base}}, \text{ such that } c_a^j = c_{a'}^j \text{ and } N_j a = N_j a' \forall j \in [1..k] \setminus \{j^*\} \},
\]

where \( A_{k+1} \) is the set of non-dominated \( P_{k+1} \) constraints (see Definition 3.4). Since \( A_{k+1}' \subset A_{k+1} \), we have \( P'_{k+1} \supset P_{k+1} \); observe \( \text{OPT}(DSCP(P'_{k+1}))) \geq \text{OPT}(DSCP(P_{k+1}))) \geq \text{OPT}(DSCP(P_k))) \)

Except for the split group \( j^* \), all coefficients of existing \( A_{k}^{\text{base}} \) configurations can be transmitted to recover more rapidly some \( A_{k+1}' \) configurations. Each time the pricing subproblem is called, we seek and add negative reduced cost columns in two phases: (1) try to find all useful columns associated to configurations in \( A_{k+1}' \) and (2) search columns using the original method from Section 3.2.2. For each configuration \( A_{k}^{\text{base}} \), phase (1) seeks a configuration of negative reduced cost having the same coefficients \( (c_a^j, N_j a) \) as \( a \) for any group \( j \neq j^* \) and new coefficients for \( j_1 \) and \( j_2 \). This is solved using the multi-choice dynamic program from Section 3.2.2, but with only two decision levels \( j_1 \) and \( j_2 \).

This latter dynamic program is significantly faster than the original one from Section 3.2.2. Besides only using two decision levels, the maximum resource amount to consider is also reduced from \( C^* \) to \( C^* - \sum_{j \in [1..k] \setminus \{j^*\}} c_a^j \) (we remove the amount consumed by inherited values from \( a \in A_{k}^{\text{base}} \)). In certain cases, \( \text{OPT}(DSCP(P'_{k+1}))) = \text{OPT}(DSCP(P_{k+1}))) \) and this is enough to conclude \( \text{lb}_{k+1} = \text{lb}_k \). In practical terms, this fully justifies using the aggregated pricing approach from Section 3.2.2: based on this inheritance with two levels, one can sometimes incrementally compute \( \text{lb}_{k+1} \) from \( \text{lb}_k \) in almost no-time. The full algorithmic template is presented in Algorithm 2, Appendix C.
4. Comparing Linear and Equality Aggregations

We motivate on the use of linear aggregation with regard to simpler equality aggregations (e.g., \(y_i = y_j\), for aggregated \(i, j \in [1..n]\)). Equality aggregations are mostly useful when one can assume that similar elements will be associated to the same dual value at optimality. This is sometimes true, for instance, in certain instances of the cutting-stock problem, if the weights of the aggregated items are similar. More generally, it is reasonable to apply equality aggregations on rather similar entities in the model, such as, related scheduling tasks [10, 11, 3], close time instants [19], nearby locations, etc. However, in capacitated problems, elements with significantly different resource consumptions can hardly be considered equivalent, as the capacity constraints can substantially change the column sets covering such elements.

2IDA could simply perform equality aggregations by imposing \(\alpha^j = 0, \forall j \in [1..k]\). However, for both linearity and equality restrictions, the aggregated pricing for \(P_k\) generates only constraints that are valid for \(P\), yet \(P\) contains some solutions that cannot be projected into feasible \(P_k\) solutions. Non-valid (overly-strong) \(P\) constraints arise implicitly in \(P_k\) by combining valid columns and exchange vectors [27] that are not valid for \(P\).

We illustrate this exchange process on a valid \(P\) constraint involving some terms \(a_i y_i + a_j y_j\), i.e., the corresponding column has the form \([\ldots a_i \ldots a_j \ldots]^\top\), where the dotted elements indicates other column values. An equality aggregation \(y_i = y_j\) would make \(a_i\) and \(a_j\) interchangeable in the above column. This generates exchange vectors with two non-zero elements (at positions \(i\) and \(j\), see below) that can be combined with valid columns to produce implicit columns; this exchange process can be schematized as follows:

\[
\left[\ldots a_i \ldots a_j \ldots\right]^\top + \psi \cdot \left[\ldots 1 \ldots -1 \ldots\right]^\top \rightarrow \left[\ldots a_i + \psi \ldots a_j - \psi \ldots\right]^\top, \tag{4.1}
\]

where \(\psi \in [-a_i, a_j]\). The resulting constraint (column) may be infeasible. By setting \(\psi\) to \(-a_i\) or \(a_j\), one respectively replaces \(a_i\) by \(a_j\), or \(a_j\) by \(a_i\). Since we deal with equality restrictions, such replacements are always bidirectional (negative or positive). This exchange process might relax the maximum (resp. minimum) resource capacity constraint if it replaces an element by an element of larger (resp. smaller) resource consumption.

We will now see that the dual linearity restrictions produce exchange vectors that are fractional. The constraints of (3.1) (see p. 8) can be written using the \(y\) variables only. Using Definition 3.1 (p. 8), we can write \(y_{j_{\text{min}}}^j = \alpha w_{j_{\text{min}}}^j + \beta\), and \(y_{j_{\text{max}}}^j = \alpha w_{j_{\text{max}}}^j + \beta\), where \(y_{j_{\text{max}}}^j\)
and $y^j_{\min}$ correspond to the elements of $I^j$ of largest (respectively smallest) consumption. This gives us $\alpha = \frac{y^j_{\max} - y^j_{\min}}{w^j_{\max} - w^j_{\min}}$ and $\beta = \frac{w^j_{\min} y^j_{\max} - w^j_{\max} y^j_{\min}}{w^j_{\min} - w^j_{\max}}$. Then, for each element $i$ of group $j$ different from these two extremes, the linearity constraint is equivalent to adding a constraint $y_i = \frac{y^j_{\max} - y^j_{\min}}{w^j_{\max} - w^j_{\min}} w^j_{\max} - \frac{w^j_{\min} y^j_{\max} - w^j_{\max} y^j_{\min}}{w^j_{\min} - w^j_{\max}} w^j_{i} + \frac{w^j_{\min} y^j_{\max} - w^j_{\max} y^j_{\min}}{w^j_{\min} - w^j_{\max}} w^j_{\min}$.

Without restricting generality, we consider that the first element of $I^j$ corresponds to $y^j_{\min}$ and the last one to $y^j_{\max}$. Therefore, the produced exchange vectors restricted to group $j$ have the following form:

$$\begin{bmatrix} a^j_{\min} \\ \vdots \\ a^j_i \\ \vdots \\ a^j_{\max} \end{bmatrix} + \psi \begin{bmatrix} \frac{w^j_{\max} - w^j_i}{w^j_{\max} - w^j_{\min}} \\ \vdots \\ 1 \\ \vdots \\ \frac{w^j_{\min} - w^j_{\max}}{w^j_{\max} - w^j_{\min}} \end{bmatrix} \rightarrow \begin{bmatrix} a^j_{\min} + \psi \frac{w^j_{\max} - w^j_i}{w^j_{\max} - w^j_{\min}} \\ \vdots \\ a^j_i - \psi \\ \vdots \\ a^j_{\max} + \psi \frac{w^j_{\min} - w^j_{\max}}{w^j_{\max} - w^j_{\min}} \end{bmatrix},$$

(4.2)

where $\psi$ can be positive or negative, which may respectively decrease or increase the coefficient of $a_i$.

An implicit configuration $\hat{a}$ generated this way can have fractional coefficients. Therefore, it may lead to violated patterns because it relaxes the integrity of the variables, but it cannot violate the resource constraints—i.e., $C^- \leq \hat{a}^\top w \leq C^+$ still holds. We now show this result formally.

**Proposition 4.1.** Non-valid configurations $\hat{a}$ produced by linearity aggregation from a feasible configuration $a$ have the same resource consumption as $a$. Such non-valid configurations cannot violate resource constraints. This is generally false for configurations produced by equality aggregations.

**Proof:** We first show that the equality aggregation may lead to non-valid configurations $\hat{a}$ that violate the resource constraint. It is sufficient to give an example. Take $C^- = 8$, $C^+ = 10$ and two elements of resource consumption $w_1 = 8$ and $w_2 = 3$ in the same group. Configuration $[0, 3]^\top$ is valid. An exchange vector $[1, -1]^\top$ with $\psi = 3$ leads via (4.1) to $[3, 0]^\top$, which corresponds to a total resource consumption of 24. For the minimum resource consumption, an example can be produced by taking the valid configuration $[1, 0]^\top$ and using $\psi = -1$ in (4.1).

We now show that non-valid configurations $\hat{a}$ generated by linearity restrictions do not violate the resource constraint because the total resource consumption cannot be modified by linearity-based exchange vectors. Recall that the configurations $a$ generated by the pricing
sub-problem are valid configurations, i.e. \( C^- \leq a^\top w \leq C^+ \). An implicit configuration \( \hat{a} \) is obtained by iteratively applying exchange vectors to a valid configuration \( a \).

Let \( a = [a^i_{\min} \ldots a_i \ldots a^i_{\max}]^\top \) be a feasible configuration and \( e \) the exchange vector

\[
\left[ \frac{w^j_{\max} - w_j \psi}{w^j_{\max} - w^j_{\min}}, \ldots, -1, \ldots, \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}} \right]^\top.
\]

We observe in (4.2) that combining \( a \) and \( e \) can lead to new artificial configurations the form \( a + \psi e \).

Initially the resource consumption of configuration \( a \) is \( a^\top w \). Note that in \( a + \psi e \), the only coefficients to be modified are related to \( a^i_{\min}, a^i_{\max} \) and \( a_i \). In the original configuration, the total resource consumption of these three elements is \( a^i_{\min} w^j_{\min} + a_i w_i + a^i_{\max} w^j_{\max} \). In \( a + \psi e \), this resource consumption becomes:

\[
(a^i_{\min} + \frac{w^j_{\max} - w_j}{w^j_{\max} - w^j_{\min}} \psi) w^j_{\min} + (a_i - \psi) w_i + (a^i_{\max} + \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}} \psi) w^j_{\max},
\]

which is equivalent to:

\[
a^i_{\min} w^j_{\min} + a_i w_i + a^i_{\max} w^j_{\max} + \frac{w^j_{\max} - w_j}{w^j_{\max} - w^j_{\min}} \psi w^j_{\min} - \frac{\psi w_i (w^j_{\max} - w^j_{\min})}{w^j_{\max} - w^j_{\min}} + \frac{w_i - w^j_{\min}}{w^j_{\max} - w^j_{\min}} \psi w^j_{\max}
\]

This simplifies to \( a^i_{\min} w^j_{\min} + a_i w_i + a^i_{\max} w^j_{\max} \), which means that the resource consumption is the same in \( a \) and \( a + \psi e \). By iteratively adding exchange vectors to the initial valid configuration \( a \), a configuration \( \hat{a} \) such that \( C^- \leq \hat{a}^\top w = a^\top w \leq C^+ \) is obtained. Therefore, any such configuration \( \hat{a} \) does not violate the resource constraint.

\[ \square \]

5. Cutting-Stock Application and Implementation Aspects

In order to validate our methodology, we applied 2IDA to the well-known Cutting-Stock Problem (CSP). We first discuss theoretical interest in linearity restrictions compared to equality restrictions in this special case. Then, we describe in greater detail some group-splitting strategies improved by additional problem knowledge.

We recall the CSP. We are given a bin size \( C \in \mathbb{N} \) and a list \( 1, \ldots, n \) of items. Each item \( i \) has a weight \( w_i \) and a demand \( b_i \). The objective is to minimize the number of bins to use to accommodate all items. This problem can be solved [13] by an extended formulation whose master problem is a set-covering model, and whose pricing sub-problem is a knapsack problem. Therefore, it lies in the scope of our study, fitting the initial model (2.2) very well. We interpret item weights as resource consumptions; the total capacity consumption of a configuration (pattern) must be between \( C^- = 0 \) and \( C^+ = C \).
5.1. Linear Restrictions vs. Equality Restrictions

Since the equality aggregation is a special case of linear aggregation, the quality of the bound obtained by the latter will be better. For $k = 1$, the linear aggregation even has a worst-case performance that cannot be reached by the equality aggregation.

**PROPOSITION 5.1.** The asymptotic worst case performance of 2IDA with $k = 1$ for the standard Cutting-Stock Problem is $1/2$, and this ratio is tight.

**Proof:** Polytope $P_1$ contains the solution $y_i = \frac{w_i}{C}$. This dual solution is equivalent to the optimal solution of the model of [15], which is known to have an asymptotic worst-case performance of $1/2$.

To prove the tightness, consider an instance with $n = 3$, $w = [\epsilon, C + \epsilon, C + 2\epsilon]$ and $b = [1, M, 1]$, for a small enough $\epsilon > 0$ and a sufficiently large $M \in \mathbb{Z}_+$. The optimum of this instance is $M + 1$. We observe that the set of $R^1$ of feasible resource consumptions is $\{\epsilon, \frac{C}{2} + \epsilon, \frac{C}{2} + 2\epsilon, C - \epsilon, C\}$. Since $w_2 = \frac{C}{2} + \epsilon$ can be as close to $\frac{w_1 + w_3}{2}$ as desired (using a small enough $\epsilon$), $y_2$ can also be arbitrarily close to $\frac{w_1 + w_3}{2}$ – as long as we have only one group, i.e., $k = 1$. We now observe that $y_1 + y_3 \leq 1$, based on the pattern with total consumption $C$. As such, we can conclude that $y_2$ can be arbitrarily close to $\frac{1}{2}$; as such, the objective value $OPT(DSCP(P_1))$ is $b^T y = b_1 y_1 + b_2 y_2 + b_3 y_3 = y_1 + M y_2 + y_3$, which has an asymptotic value (when $\epsilon \to 0$) of $\frac{M}{2} + 1$, while the instance has an optimum of $M + 1$. \[\square\]

This result does not hold for the equality aggregation. For $k = 1$, all dual values are equal to $\frac{1}{[C/w_{\text{min}}]}$, otherwise the valid dual-constraint that uses $[C/w_{\text{min}}]$ items of size $w_{\text{min}}$ would be excluded. Therefore, the ratio between the value found by the model and the optimal solution is not bounded by any constant.

5.2. Split Selection Based on Dual-Feasible Functions

In Section 3.4, we have mentioned general principles for making splitting decisions in 2IDA. For cutting stock, we can also use ad-hoc methods based on the knowledge of good dual solutions (obtained by applying so-called Dual-Feasible Functions [18, 7]).

**Definition 5.1.** A function $f : [0, C] \to [0, 1]$ is a dual-feasible function (DFF), if the following holds $\sum_{i \in I} b_i w_i \leq C \implies \sum_{i \in I} b_i f(w_i) \leq 1$ for any index set $I$ such that $b_i \in \mathbb{Z}_+$ and $w_i > 0$.

To obtain a valid CSP lower bound, it is sufficient to apply a dual-feasible function on the weights of the items and to sum up the obtained values. This way, a dual feasible
function generate a valid dual solution $y$: it takes the weights as an input, and associate to each weight $w_i$ the corresponding dual value $y_i = f(w_i)$.

Given a high-quality DFF $f$, a good method for splitting groups consists of partitioning $[0, C]$ in such a way that $f$ is linear over each group. Therefore, the split method from this section tries to reproduce the structure of the best dual-feasible function available. The motivation comes from the fact that most of the classical DFFs have a piece-wise linear form over $[0, C]$. The different pieces define a natural partition of the element set $I$. The resulting DFF-based split method computes a priori $k$ intervals before starting constructing $\mathcal{P}_k$. If the function has $k$ pieces, splitting the dual values into $k$ groups leads 2IDA at least to the same feasible solutions as those returned by the DFFs. However, the best function is not always useful. For example, $f(x) = \frac{x}{C}$ only defines $k = 1$ interval. Therefore, such specific functions are not used in practice.

5.3. Split Selection Based on Outside Reference Solutions

We now introduce a split method that implements the principles from Section 3.4.2 and tunes the evaluation for Cutting-Stock. While many details below are especially tailored for Cutting-Stock, similar formulas can be used for other applications.

Considering the elements within each group sorted by increasing weight, we can note a split decision by a pair $(j, n_{j_1})$: $j$ is the group to be split and $n_{j_1}$ the index of the largest weight in the first subgroup $j_1$. Our splitting method assigns scores $h(j, n_{j_1})$ to all potential splits. Three pieces of information are used to determine a total score $h(j, n_{j_1}) = h_1(j, n_{j_1}) \cdot h_2(j, n_{j_1}) \cdot h_3(j, n_{j_1})$. In the end, the potential split of maximal score is selected.

The first two scores $h_1$ and $h_2$ are based on the simple ad-hoc considerations related to Cutting-Stock. Score $h_1(j, n_{j_1})$ is based on weight spread information, i.e., $h_1(j, n_{j_1}) = w_{\max}^j - w_{\min}^j$ if $n_{j_1}, n_j - n_{j_1} > 1$, or 1 otherwise. The special cases of splitting a group at positions 1 or $n_j - 1$ should be avoided, since they produce a subgroup of size 1. However, in general, the probability of choosing a group $j$ is proportional to its weight spread, as it is reasonable to split larger segments first.

Score $h_2$ evaluates the impact of a group based on its relative position in $[1..k]$. The “extremal” groups $j = 1$ and $j = k$ are always associated to the lightest and respectively the heaviest elements and therefore are critical in the case of cutting-stock. These two groups are given a bonus by scoring function $h_2$. This part of the procedure is clearly hand-tailored for the cutting-stock and is not likely to generalize to other problems.
The most important part of the final score is given by $h_3$. It is based on a comparison of the current optimal solution $y^*_k$ of $\mathcal{P}_k$ to an outside reference solution such that $y^*_k \rightarrow y^u$ is an open direction. As hinted in Section 3.4.2, we assign higher scores to groups that are more likely to lead (using the process from Section 4) to implicit non-valid constraints that separate $y^*_k$ from $y^u$. For this, we compare $y^*_k$ with $y^u$. While $y^*_k$ has a group-wise linear structure, this is not the case for $y^u$; we actually try to find groups on which the difference $y^u - y^*_k$ is far from indicating a linear transformation.

The difference between $y^*_k$ and $y^u$ for a given group $j$ is evaluated using the $\Delta$ operator, defined by:

$$\Delta^j_{i_1,i_2}(y^*_k,y^u) = \sum_{i=i_1}^{i_2} b^j_i \cdot (y^*_k - y^u)^j_i, \quad (5.1)$$

where $(y^*_k - y^u)^j_i$ is the $i^{th}$ position of the vector $y^*_k - y^u$ restricted to the elements of group $j$. To simplify the notations, we will hereafter ignore the argument $(y^*_k,y^u)$; no confusion can arise because all comparisons below concern $y^*_k$ and $y^u$.

Operator $\Delta$ is useful for detecting differences between $y^*_k$ and $y^u$ and proposing useful splits. Let us analyze the case where the global $\Delta$ evaluation over group $j$ is positive—i.e., consider $\Delta^j_{1,n_j} > 0$ (the opposite case $\Delta^j_{1,n_j} \leq 0$ is symmetric). This indicates a global positive “trend” on group $j$: the objective value of $y^*_k$ could be improved by (non-linearly) increasing most dual values in group $j$; such operations can be performed while maintaining feasibility in $\mathcal{P}$. Secondly, if $\Delta^j_{i,j}$ is negative for some $i \in [1..n_j]$, a group split at $(j,i)$ would allow 2IDA to decrease the first $i$ dual values and increase the others. In this manner, 2IDA would follow the open direction $y^*_k \rightarrow y^u$. A more frequent situation is associated to a segment $[i_a,i_b] \in [1..n_j]$ such that $\Delta^j_{i_a,i_b} < 0$; to isolate dual values of $[i_a..i_b]$ from the rest of the group, one should consider split decisions $(j,i_a - 1)$ or $(j,i_b)$.

For this, we divide $[1..n_j]$ in three segments $[1..i_a)$, $[i_a..i_b)$, $(i_b..n_j]$, determining $i_a$ and $i_b$ as follows: $i_a = \min\{i \in [1..n_j] : \Delta^j_{i,i-1} \geq 0, \Delta^j_{i,j} < 0\}$ and $i_b = \max\{i \in [i_a..n_j] : \Delta^j_{i_a,j} < 0\}$. The middle segment $[i_a,i_b]$ has an “inverse trend” compared to $[1..n_j]$, i.e., $y^*_k$ could feasibly improve in $\mathcal{P}$, by decreasing its values over $[i_a,i_b]$, while increasing its values over the rest of $[1..n_j]$. Roughly speaking, we are looking for a sequence of indices $[i_a..i_b]$ on which $y^*_k$ could decrease by dropping constraints that link elements of $[i_a..i_b]$ and other elements from the group. The most fortunate case is related to $i_a = 1$ (or, analogously, $i_b = n_j$), because this would make $[1..i_a)$ empty; as such, the split decision $(j,i_b)$ would surely allow the first $i_b$
values to decrease while increasing the others. The scoring function $h^3$ is determined by:

$$h^3(j, i_a) = \Delta_{i_a-1}^j, h^3(j, i_b) = \Delta_{n_j+1}^j, h^3(j, i) = 0, \forall i \in \{1..n_j\} \setminus \{i_a, i_b\}.$$ 

6. Numerical Evaluation

We first evaluate 2IDA on the Cutting Stock Problem (CSP) in Sections 6.2-6.4. Section 6.5 continues with experiments on the CSP with Maximum Waste (CSP-MW).

We first focus on investigating the internal 2IDA dynamics (Section 6.2), i.e., we report the running time and the number of iterations (sub-problems) for the lower and the upper bounding calculations. In Section 6.3, we compare the intermediate 2IDA lower bounds with the Lagrangian bounds of a classical Column Generation (CG) algorithm. Section 6.4 validates the fact that, in many cases, 2IDA reaches the final CG optimum ($\text{OPT}_{\text{CG}}$) more rapidly than the classical CG.

6.1. Experimental Conditions and CSP Instances

We implemented both 2IDA and a classical non-stabilized CG algorithm based on Dynamic Programming (DP) for pricing the columns. A similar implementation approach (using lists of states) has been used for all DP routines of 2IDA, namely: the coefficient calculation preprocessing stage (Section 3.2.1), the multiple-choice aggregated pricing for optimizing DSCP($P_k$) in Section 3.2.2 (or in the “constraint inheritance” process from Section 3.6), the upper bounding pricing for optimizing over $P^u$ (Section 3.5.2).

The first split decisions are made based on the intervals returned by some classical dual-feasible functions (DFF, Section 5.2). We only used DFFs with relatively few intervals. As soon as 2IDA requires more intervals, the upper bound and the corresponding outside reference solution (when available) are used to further guide split decisions (see Sections 3.4.2 or 5.3). When no upper bound is available (pure 2IDA), this split decisions are simply taken by the method from Section 3.4.1. We turn to classical CG if $\lceil \text{lb}_k \rceil + 1 = \lceil \text{ub}_k \rceil$ or if $k = 10$. More exactly, when 10 iterations are not sufficient to reach optimality, the current set of columns used for upper bounding is used to continue with a final CG phase that is run up to optimality.

We used a large set of instances from a dozen of benchmark sets (3127 individual instances). Their characteristics are described in great detail in Appendix A. All reported times are obtained using the same computing setting. The technical characteristics (CPU model, LP solver, source code, etc.) are also indicated in Appendix A.
6.2. 2IDA Profile and Usefulness of Upper Bounding

Computing an upper bound for each \( k \) value (Section 3.5) is not a mandatory part in 2IDA. Since it requires running non-aggregated column generation, it is interesting to evaluate the impact of the upper bound in terms of the total computing effort.

Table 1 presents the computing effort (CPU time and sub-problems calls) for two 2IDA versions: one using the intermediate upper bounds from Section 3.5 ("yes" in Column 2) and a "pure" 2IDA version that does not use such intermediate upper bounds ("no" in Column 2). However, recall (Section 6.1) that we always turn to the classical CG algorithm if the optimum is not reached at \( k = 10 \). As such, the indicated upper bounding computing effort includes both the intermediate upper bounds and the upper bounds computed during the final CG process.
The first conclusion is that 2IDA is generally more efficient when intermediate upper bounds are used. These bounds are useful to guide the split decision or to stop the process more rapidly. The number of $y^*_k$-tight constraints in the polytope $P^u$ (see Section 3.5) is significantly smaller than the total number of constraints in $P$. Therefore, the optimization of $P^u$ does not introduce prohibitively large slowdowns. The efficiency is greatly improved when the optimality can be proved only using intermediate upper bounds for some $k < 10$.

For full convergence, 2IDA can spend more computing time on upper bounding than on lower bounding. The fact that the method switches to classical CG after a number of iterations increases the difference between the UB and LB computing efforts. This difference can be significant, most notably for vb50-3 and vb50-5, where the lower bounding stops at $k = 1$ and classical CG is run immediately because $\lceil lb_k \rceil + 1 = \lceil ub_k \rceil$. The running time dedicated to upper bounding is almost ten times larger than the lower bounding time. The construction of $P_1$ is not carried out by column generation, since the model has only 2 variables and it is faster to directly generate all constraints. This explains why the number of calls to multiple-choice knapsack routines is sometimes 0 (columns “LB (2IDA) calls”). In such cases, most of the total 2IDA computing effort is actually spent on proving that this lower bound is optimal.

The 2IDA version with no intermediate upper bounds may be efficient when small instances are considered. The smallest instances vb10 with $n = 10$ could be solved by using no upper bounding at all. In this case, 2IDA reaches rapidly a state in which all groups have at maximum two elements and the 2IDA lower bound is optimal. This also happens for the smaller vbInd instances. Clearly, the results on the most difficult instances are more useful. Therefore, in the remaining we only focus on the version with intermediate upper bounds and we ignore the small instances vb10.

### 6.3. Quality of the Intermediate Dual Bounds

We compare the dual bounds iteratively produced by 2IDA with the Lagrangian dual bounds produced during CG each time the sub-problem is solved. We used the Farley bound, which is a well-known specialization of the Lagrangian bound dedicated to this specific model—see Appendix B for the exact formula and references.

Table 2 reports a comparison constructed by the following protocol. We first run the classical CG on each instance. This produces for each instance a reference computing time $T_{CG}$ and an optimal value $OPT_{CG}$. Then, we run for the same instance both CG and 2IDA
Table 2  Comparison of intermediate dual bounds for CG an 2IDA. We report the average ratio \(L_p / \text{OPT}_{\text{CG}}\) for each set of instances for \(p = 5\%\) to \(p = 50\%\) of the time \(T_{\text{CG}}\). For example, row \(\text{vb20}\) of column "5\%" shows that if both methods are run on instances of data set \(\text{vb20}\), and stopped after \(5\% \cdot T_{\text{CG}}\), CG will output a dual bound equal to 0.41 \(\text{OPT}_{\text{CG}}\), while 2IDA will output a dual bound equal to 0.64 \(\text{OPT}_{\text{CG}}\).%

<table>
<thead>
<tr>
<th>Instance set</th>
<th>Percentage of CG time (T_{\text{CG}}) needed for full convergence</th>
<th>5%</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CG</td>
<td>2IDA</td>
<td>CG</td>
<td>2IDA</td>
<td>CG</td>
<td>2IDA</td>
</tr>
<tr>
<td>(\text{vb20})</td>
<td>0.41</td>
<td>0.64</td>
<td>0.55</td>
<td>0.95</td>
<td>0.70</td>
<td>0.99</td>
</tr>
<tr>
<td>(\text{vb50-1})</td>
<td>0.29</td>
<td>0.88</td>
<td>0.42</td>
<td>0.97</td>
<td>0.65</td>
<td>0.98</td>
</tr>
<tr>
<td>(\text{vb50-2})</td>
<td>0.49</td>
<td>1.00</td>
<td>0.70</td>
<td>1.00</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>(\text{vb50-3})</td>
<td>0.44</td>
<td>1.00</td>
<td>0.57</td>
<td>1.00</td>
<td>0.65</td>
<td>1.00</td>
</tr>
<tr>
<td>(\text{vb50-4})</td>
<td>0.52</td>
<td>1.00</td>
<td>0.66</td>
<td>1.00</td>
<td>0.76</td>
<td>1.00</td>
</tr>
<tr>
<td>(\text{vb50-5})</td>
<td>0.52</td>
<td>1.00</td>
<td>0.60</td>
<td>1.00</td>
<td>0.65</td>
<td>1.00</td>
</tr>
<tr>
<td>(\text{vbInd})</td>
<td>0.41</td>
<td>0.94</td>
<td>0.51</td>
<td>0.95</td>
<td>0.65</td>
<td>0.97</td>
</tr>
<tr>
<td>(\text{m01})</td>
<td>0.49</td>
<td>0.74</td>
<td>0.55</td>
<td>0.92</td>
<td>0.66</td>
<td>0.95</td>
</tr>
<tr>
<td>(\text{m20})</td>
<td>0.64</td>
<td>0.47</td>
<td>0.64</td>
<td>0.80</td>
<td>0.69</td>
<td>0.92</td>
</tr>
<tr>
<td>(\text{m35})</td>
<td>0.89</td>
<td>1.00</td>
<td>0.89</td>
<td>1.00</td>
<td>0.89</td>
<td>1.00</td>
</tr>
</tbody>
</table>

with a time limit of \(p \cdot T_{\text{CG}}\), where \(p \in \{5\%, 10\%, 20\%, 30\%, 40\%\}\). For each such \(p\), we note \(L_p\) the best lower bound obtained after \(p \cdot T_{\text{CG}}\) time. If 2IDA is stopped during step \(k\), \(L_p\) is either the optimum of \(\text{DSCP}(P_{k-1})\) or the best Lagrangian dual bound obtained during step \(k\). Table 2 reports the evolution of \(L_p / \text{OPT}_{\text{CG}}\) when \(p\) goes from 5\% to 40\%.

Within 5\%\(T_{\text{CG}}\) 2IDA finds the CG optimum for all instances \(\text{bin3}\) and \(\text{vb50-\{2,3,4,5\}}\). For the same computing time, the Lagrangian CG lower bounds are equal to respectively 0.89 \(\text{OPT}_{\text{CG}}\), 0.49 \(\text{OPT}_{\text{CG}}\), 0.44 \(\text{OPT}_{\text{CG}}\), 0.52 \(\text{OPT}_{\text{CG}}\), and 0.52 \(\text{OPT}_{\text{CG}}\) on average.

Within 10\%\(T_{\text{CG}}\) time, the lower bounds produced by 2IDA are better than 0.92 \(\text{OPT}_{\text{CG}}\) for all instance sets (except \(\text{m35}\), see below). Within the same time limit 10\%\(T_{\text{CG}}\), the Lagrangian GC bounds are at most 0.7 \(\text{OPT}_{\text{CG}}\) except for instance set \(\text{bin3}\).

2IDA slowly closes the gap that is already small after 20\%\(T_{\text{CG}}\); this is obvious when looking at columns 20\% to 40\%. When the number of groups becomes larger than 10, 2IDA switches to classical column generation. The same convergence issues occur and even if the Lagrangian bound is almost equal to \(\text{OPT}_{\text{CG}}\), the method may need time to converge.

The only case in which the classical Lagrangian bound is better is \(\text{m35}\), only for \(p = 5\%\). These instances have all weights larger than \(\frac{C}{3}\), and so, all columns have two non-zero coefficients. The first (almost linear) approximations of the dual values are likely to be weak. Note however that after 10\% of the time needed for CG, 2IDA becomes more effective in computing dual bounds (0.80 \(\text{OPT}_{\text{CG}}\) on average vs. 0.64 \(\text{OPT}_{\text{CG}}\) for CG).
Table 3 The computing effort required for final convergence with 2IDA and CG. Columns “Avg. CPU” report the time (in ms.). Columns “2IDA vs CG (CPU)” report the number of times 2IDA is (1) faster than CG (column ≺), (2) roughly equally fast (difference < 5%, column ≃), and (3) slower (column ≻). The last two columns compares the total number of pricing calls. For 2IDA, the total number of pricing calls considers both Multiple-Choice Knapsack problems (columns of $P_k$) and classical Knapsack problems (columns used for upper bounding, including the inherited columns from Section 3.6).  

<table>
<thead>
<tr>
<th>Instance set</th>
<th>Avg. CPU [ms]</th>
<th>2IDA vs CG (CPU)</th>
<th>nb sub-problems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2IDA</td>
<td>CG</td>
<td>≺</td>
</tr>
<tr>
<td>vb20</td>
<td>9161</td>
<td>11653</td>
<td>10</td>
</tr>
<tr>
<td>vbIndstr</td>
<td>7580</td>
<td>14252</td>
<td>10</td>
</tr>
<tr>
<td>vb50-1</td>
<td>39827</td>
<td>50164</td>
<td>14</td>
</tr>
<tr>
<td>vb50-2</td>
<td>89432</td>
<td>125399</td>
<td>17</td>
</tr>
<tr>
<td>vb50-3</td>
<td>44068</td>
<td>106356</td>
<td>20</td>
</tr>
<tr>
<td>vb50-4</td>
<td>66498</td>
<td>87604</td>
<td>19</td>
</tr>
<tr>
<td>vb50-5</td>
<td>28490</td>
<td>85680</td>
<td>20</td>
</tr>
<tr>
<td>m01</td>
<td>818</td>
<td>849</td>
<td>549</td>
</tr>
<tr>
<td>m20</td>
<td>423</td>
<td>620</td>
<td>793</td>
</tr>
<tr>
<td>m35</td>
<td>207</td>
<td>379</td>
<td>920</td>
</tr>
<tr>
<td>hard</td>
<td>117908</td>
<td>376112</td>
<td>10</td>
</tr>
</tbody>
</table>

6.4 Computational Effort Required to Reach the Optimum

The main purpose of 2IDA is to produce high-quality dual bounds in a faster way than the classical CG. However, the fact that almost optimal dual bounds are produced in the first iterations hints that the method can help proving optimality in a faster way.

Table 3 compares the CPU time measures and the number of sub-problems calls needed to converge by either method. 2IDA is significantly faster for 2402 instances out of 3127 (76%), and at least as good in 2544 cases (81%). On average, our method is faster than CG for all data sets. This can be explained by the fact that less computationally demanding (aggregated) pricing procedures are run using 2IDA. The first iterations are more effective for producing a good dual bound (as shown above). Furthermore, by only generating $y_k^*$-tight constraints during the 2IDA upper bounding process (Section 3.5), we introduce a form of implicit stabilization around those good initial solutions.

Note that the most difficult instances are hard. For these instances, our approach is always faster (a factor 3 on average) and produces less sub-problems.

To conclude, let us insist on two essential keys for the practical effectiveness. First, dual-feasible functions are useful to make good split decisions. This means that knowing a priori the structure of a good dual solution can be useful in 2IDA. Secondly, the incremental construction of polytope $P_{k+1}$ from $P_k$ (Section 3.6) reduces the computing time by a wide range compared to constructing $P_{k+1}$ from scratch at each iteration. This part should not be overlooked in a good implementation of 2IDA: in certain cases, this inheritance process
allows 2IDA to compute $\text{lb}_{k+1}$ from $\text{lb}_k$ in almost no time (see also detailed instance-by-instance results at www.lgi2a.univ-artois.fr/~porumbel/cs/details2ida.zip).

### 6.5. Cutting Stock Problem with Maximum Waste

Since 2IDA is not dedicated to the cutting-stock problem and its structure, we now consider the Cutting Stock Problem with Maximum Waste (CSP-MW). It is interesting to consider this problem, because there is no equivalent of the DFF bounds for this variant, and thus we have no information about the structure of good dual solutions. CSP-MW is defined as follows: all valid patterns are required to have a total length (weight) between $C^-$ and $C^+$. In practice, one can consider that $C^+ = C$ is the roll length and $C^+ - C^-$ is the maximum allowed waste. This constraint arises in industry when large pieces of waste cannot be recycled. The new maximum waste constraints might make certain items $i$ be cut more

<table>
<thead>
<tr>
<th>Instance</th>
<th>k=1</th>
<th>k=2</th>
<th>k=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>m01-3</td>
<td>940 [960]</td>
<td>41 [100]</td>
<td>54 [40]</td>
</tr>
<tr>
<td>vb50-1p02</td>
<td>980 [1200]</td>
<td>319 [1250]</td>
<td>940 [2240]</td>
</tr>
</tbody>
</table>

**Table 4** CSP-MW Results. Column 1 is the instance name. Columns 2, 4 and 6 respectively provide the bound obtained by 2IDA for $k$ from 1 to 3. Columns 3, 4, and 5 respectively report the Lagrangian bound obtained using a similar CPU time as 2IDA. More exactly, for each $k$, we report the best Lagrangian bound found during CG at the first moment when the CG running time is larger than the one needed by 2IDA. The last two columns indicate the CG optimum for CSP-MW and CSP

* Remark that these 2IDA bounds are better than the CG optimum $\text{OPT}_{CG}$ for basic Cutting Stock (last column).
than \( b \) times (such additional items are not considered as waste).

We consider the same CSP instances. For each benchmark set, we report individual results for the first three instances. The maximum waste is set as follows:

- 0 for the \( m01 \) instances (i.e., \( C^- = C^+ = C = 100 \)). The instances \( m20 \) and 35 were not used because they contain only large items (at least \( \frac{20}{100}C \) or respectively \( \frac{35}{100}C \), see Appendix A), and so, they are most often infeasible under the current no-waste constraint;
- 2 for all \( vb50 \) instances (i.e., \( C^- = 99998 \);
- 0.5\%C for \( vbInd \) instances; for this set, we select the only three instances with \( n \geq 30 \);
- 4, for the hard instances (i.e., \( C^- = 196 \)).

Table 4 compares the 2IDA bounds with the Lagrangian CG bounds. We used a pure 2IDA version with no upper bounding and with splitting decisions taken using weight spread indicators (as described in Section 3.4.1). The 2IDA bounds clearly dominate the Lagrangian bounds. More exactly, even the first 2IDA bounds for \( k = 1 \) (Column 2) are usually larger than the Lagrangian bounds reported after 2-3 times more time (Column 7). The only exception arises for the hard instances, where CG converges more rapidly.

7. Conclusions and Perspectives

We described an aggregation method for computing dual bounds in column generation in a way that is faster than classical Lagrangian bounds. The new method proceeds by constructing an iterative inner approximation of the dual polytope and converges towards the optimum of the original model. Several computational techniques are used to incrementally construct a sequence of inner polytopes. Computational results show that, besides generating fast dual bounds, the method can reach the optimum of the original model more rapidly than classical column generation in many cases.

Future research work will concentrate on the case with several resources, and also more general cases where the pricing sub-problems are solved by an ILP solver.

References


Appendix A: Instance Sets and Computing Setting

All reported CPU times are obtained using the following computing environment: a HP ProBook 4720 laptop clocked at 2.27GHz (Intel Core i3), with the gnu g++ C++ compiler on Linux Ubuntu, kernel version 2.6. The master problems are solved using Cplex 12.3 and Ilog Concert libraries. The source code and the instances are publicly available on-line at www.lgi2a.univ-artois.fr/~porumbel/cs/. Detailed results (instance by instance) are also publicly available at this address.

We now describe in details the instances we used in this paper. They form a set of 3127 individual instances. The number of items ranges from 10 to 200 and the capacities is between 100 and 100000.

**VB-a** instances vb10, vb20 (random instances [26]) and vbIndst (industrial instances [26]). We selected only the industrial instance with one bin type;

**VB-b** instances vb50-1, vb50-2, ... , vb50-5, random instances;

**BP** three randomly generated bin-packing instance sets [7]: m01, m20, m35;

**hard** 10 bin-packing instances [24] long-acknowledged for their difficulty in the bin-packing literature

Appendix B: The Lagrangian Bound for CSP and CSP-MW

We now recall the specialized Lagrangian lower bound used for the standard Cutting-Stock Problem and that can be used for the Cutting-Stock Problem with Maximum Waste. We come back to the main Set-Covering column generation model (2.1)-(2.2), p. 4. Similarly to what is done in [2, § 2.2], [26, §. 3.2], [17, §. 2.1] or [4, § 1.2], Lagrangian bounds are constructed as follows. Consider a given iteration of the CG process with dual values y. We use these values y as multipliers of the Lagrangian relaxation of (2.1). The relation between the CG optimum OPT$_{CG}$ and the Lagrangian bound $L_y$ can be written:

$$\text{OPT}_{CG} \geq L_y = \min_{\lambda \geq 0} \left( \sum_{a \in A} (1 - a^\top y)\lambda_a \right) + b^\top y \geq \min_{\lambda \geq 0} \left( M_{RC} \sum_{a \in A} \lambda_a \right) + b^\top y,$$

where: $\lambda$ is the primal variable vector of the column generation, and $M_{RC} = \min_{a \in A} 1 - a^\top y \leq 0$ is the non-positive minimum reduced cost at the current iteration, The key for calculating the Farley bound resides in observing that the inequality below can be artificially added to the initial LP.

$$\sum_{a \in A} \lambda_a \leq \text{OPT}_{CG}, \quad (B.2)$$

The 2IDA Lagrangian bounds from Section 6.3 are calculated in the same manner, i.e., the whole reasoning still holds by only using $A_k$ instead of $A$. Combining (B.2) with $M_{RC} \leq 0$, we can reduce (B.1) to $\text{OPT}_{CG} \geq M_{RC} \cdot \text{OPT}_{CG} + b^\top y$. The Farley lower bound $L^F_y$ follows immediately; $L^F_y = \frac{b^\top y}{1 - M_{RC}} \leq \text{OPT}_{CG}$.

Appendix C: Constraint Inheritance Pseudocode

We here provide the algorithmic template of the constraint inheritance and optimization process from Section 3.6. Phase (1) roughly corresponds to Line 9 in 2IDA (in Alg. 1,p 14) and phase (2) to Line 4 (with incremented $k$).
Table 5: General characteristics of the instances. Columns 4 and 5 indicate the (approximate) interval of the demand value, and, respectively, item weights.

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>m01</th>
<th>m20</th>
<th>m35</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>vb10</td>
<td>CSTR10b50c[1-5]*</td>
<td>10 10000</td>
<td>[10, 100]</td>
<td>[1, 2] C</td>
<td>200 100000</td>
</tr>
<tr>
<td>vb20</td>
<td>CSTR20b50c[1-5]*</td>
<td>20 10000</td>
<td>[10, 100]</td>
<td>[1, 2] C</td>
<td>000 100</td>
</tr>
<tr>
<td>vb50-c1</td>
<td>CSTR50b50c1*</td>
<td>50 10000</td>
<td>[50, 100]</td>
<td>[1, 3] C</td>
<td>000 100</td>
</tr>
<tr>
<td>vb50-c2</td>
<td>CSTR50b50c2*</td>
<td>50 10000</td>
<td>[50, 100]</td>
<td>[1, 2] C</td>
<td>000 100</td>
</tr>
<tr>
<td>vb50-c3</td>
<td>CSTR50b50c3*</td>
<td>50 10000</td>
<td>[50, 100]</td>
<td>[1, 4] C</td>
<td>000 100</td>
</tr>
<tr>
<td>vb50-c4</td>
<td>CSTR50b50c4*</td>
<td>50 10000</td>
<td>[50, 100]</td>
<td>[1, 2] C</td>
<td>000 100</td>
</tr>
<tr>
<td>vb50-c5</td>
<td>CSTR50b50c5*</td>
<td>50 10000</td>
<td>[50, 100]</td>
<td>[1, 4] C</td>
<td>000 100</td>
</tr>
<tr>
<td>vb50-b100</td>
<td>CSTR50b100c4*</td>
<td>50 10000</td>
<td>[10, 210]</td>
<td>[1, 2] C</td>
<td>000 100</td>
</tr>
</tbody>
</table>

a In the archive http://www.math.u-bordeaux1.fr/~fvanderb/data/randomCSPinstances.tar.Z
Algorithm 2: Two-Phase CG for optimizing DSCP($P_{k+1}$): construct $P'_{k+1}$ then $P_{k+1}$

**Data:** Optimal DSCP($P_k$) solution $[\alpha^1 \ldots \alpha^k \beta^1 \ldots \beta^k]^{\top}$, existing constraints $A^\text{base}_k$

**Result:** $lb_{k+1} = \text{OPT}(\text{DSCP}(P_{k+1}))$

**Phase 1:** Lift current solution and describe $P'_{k+1}$

Express current $[\alpha^1 \ldots \alpha^k \beta^1 \ldots \beta^k]^{\top}$ as a feasible DSCP($P_{k+1}$) solution:
- Keep the values $\alpha^j$ and $\beta^j$ for all $j \neq j^*$ // no change outside split group $j^*$
- $\alpha^{j_1} \leftarrow \alpha^j$, $\alpha^{j_2} \leftarrow \beta^j$
- $\beta^{j_1} \leftarrow \beta^j$, $\beta^{j_2} \leftarrow \beta^j$

repeat
  for $a \in A^\text{base}_k$ do
    solve the aggregated multiple-choice pricing variant (Section 3.2.2) with 2 elements ($j_1$ and $j_2$) and capacity limits $C^- - \sum_{j \in [1..k] - \{j^*\}} c^j_a$ and $C^+ - \sum_{j \in [1..k] - \{j^*\}} c^j_a$ to obtain a configuration $a'$
    if $a'$ has a negative reduced cost then
      $A'_{k+1} \leftarrow A'_{k+1} \cup \{a'\}$
    end
  end

  solve the Reduced Master Problem using only configurations $A'_{k+1}$
  update $\text{OPT}(\text{DSCP}(P'_{k+1}))$ and the current dual solution $[\alpha, \beta]$;

until no configuration $a'$ of negative reduced can be found

**Phase 2:** Describe $P_{k+1}$

if $\text{OPT}(\text{DSCP}(P'_{k+1})) = \text{OPT}(\text{DSCP}(P_k))$ then
  return $\text{OPT}(\text{DSCP}(P_k))$ // $\text{OPT}(\text{DSCP}(P_k)) = \text{OPT}(\text{DSCP}(P_{k+1}))$
end

repeat
  solve the aggregated multiple-choice pricing variant (Section 3.2.2) on $k+1$ levels to obtain a new configuration $a$
  if $a$ has a negative reduced cost then
    $A_{k+1} \leftarrow A_{k+1} \cup \{a\}$
    solve the RMP related to configurations $A_{k+1}$ and update the current dual values
  end
until no configuration $a$ of negative reduced can be found

return $\text{OPT}(\text{DSCP}(P_{k+1}))$