Further experiments and insights on `Projective Cutting-Planes`

Abstract. This paper is a continuation of the study from [15] that introduced the `Projective Cutting-Planes` method to optimize (a linear function) over polytopes $\mathscr{P}$ with prohibitively-many constraints. The main goal of the new paper is to explore new applications of `Projective Cutting-Planes` and to present more numerical results on them. The current paper first presents an application of `Projective Cutting-Planes` on a robust linear programming problem, in which $\mathscr{P}$ is defined as a primal polytope. Then, we will explore the *Multiple-Length Cutting-Stock* problem, in which $\mathscr{P}$ is defined as a dual polytope in a `Column Generation` model. Finally, we will also provide additional insight for certain developments from [15], to draw more general conclusions from numerical results reported along both papers.

1. Introduction. To (try to) make the current paper relatively self-contained, we first briefly present the main ideas of the `Projective Cutting-Planes` method introduced in [15]. We focus on a Linear Program (LP) of the form:

$$\text{opt}\left\{\mathbf{b}^\top\mathbf{x} : \mathbf{a}^\top\mathbf{x} \leq c_a,\ \forall(\mathbf{a}, c_a) \in \mathcal{A}\right\} = \text{opt}\left\{\mathbf{b}^\top\mathbf{x} : \mathbf{x} \in \mathscr{P}\right\}, \tag{1.1}$$

where $\mathcal{A}$ is the set of unmanageably-many constraints and "opt" stands for either "min" or "max".

Given any $\mathbf{x} \in \mathscr{P}$, the projection sub-problem $\texttt{project}(\mathbf{x} \to \mathbf{d})$ asks to advance from $\mathbf{x}$ along $\mathbf{d}$ up to the pierce point $\mathbf{x} + t^*\mathbf{d}$ where $\mathscr{P}$ is hit, *i.e.*, determine the step length $t^* = \max\{t \geq 0 :\ \mathbf{x} + t\mathbf{d} \in \mathscr{P}\}$. To solve sub-problem, one also has to find a constraint of $\mathcal{A}$ satisfied with equality by $\mathbf{x} + t^*\mathbf{d}$, see the formal definition in [15, Def. 2.1].

The proposed `Projective Cutting-Planes` relies on the above projection sub-problem to iteratively generate a sequence of inner solutions $\mathbf{x}_{\texttt{it}}$ that converge along the iterations `it` to an optimal solution $\texttt{opt}(\mathscr{P})$. Each inner solution $\mathbf{x}_{\texttt{it}}$ is chosen as a point on the segment joining the previous inner solution $\mathbf{x}_{\texttt{it}-1}$ and the pierce point $\mathbf{x}_{\texttt{it}-1} + t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$ returned by the last projection. At the same time, each call to the projection sub-problem returns a (first-hit) constraint (of $\mathcal{A}$) satisfied with equality by the pierce point. By generating such a constraint at each iteration `it`, the `Projective Cutting-Planes` constructs a sequence $\mathscr{P}_1 \supsetneq \mathscr{P}_2 \supsetneq \cdots \supset \mathscr{P}$ of outer approximations of $\mathscr{P}$; like in the standard `Cutting-Planes`, this generates a sequence of outer solutions $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ that converge to $\texttt{opt}(\mathscr{P})$ along the iterations `it`.

At the first iteration $\texttt{it} = 1$, one can choose any starting feasible solution $\mathbf{x}_1$. The first direction $\mathbf{d}_1$ is often $\mathbf{d}_1 = \mathbf{b}$, to make the first projection $\texttt{project}(\mathbf{x}_1 \to \mathbf{d}_1)$ advance along the along the direction with the fastest rate of objective function improvement. As hinted above, the projection $\texttt{project}(\mathbf{x}_{\texttt{it}} \to \mathbf{d}_{\texttt{it}})$ returns a first-hit constraint at each iteration `it`; this constraint is added to the constraints of $\mathscr{P}_{\texttt{it}-1}$ to construct $\mathscr{P}_{\texttt{it}}$. Then, one chooses a point $\mathbf{x}_{\texttt{it}+1}$ on the segment joining $\mathbf{x}_{\texttt{it}}$ and $\mathbf{x}_{\texttt{it}} + t^*_{\texttt{it}}\mathbf{d}_{\texttt{it}}$ and the next direction points towards the new current outer optimal solution $\texttt{opt}(\mathscr{P}_{\texttt{it}})$, *i.e.*, $\mathbf{d}_{\texttt{it}+1} = \texttt{opt}(\mathscr{P}_{\texttt{it}}) - \mathbf{x}_{\texttt{it}+1}$. Then, the process is repeated by solving $\texttt{project}(\mathbf{x}_{\texttt{it}+1} \to \mathbf{d}_{\texttt{it}+1})$; this projection returns a new pierce point and a new constraint that is added to the constraints $\mathscr{P}_{\texttt{it}}$ to construct $\mathscr{P}_{\texttt{it}+1}$.

A key aspect is the design of a projection algorithm that could compete in terms of computational complexity with the separation algorithm. As hinted in the main paper, there are several techniques that can bring us very close to this goal. This new paper will show (on the robust optimization problem) that it is possible to generalize the separation algorithm to a projection algorithm without significantly increasing the complexity. The computational bottleneck of both sub-problems is enumerating a set of nominal constraints that generate prohibitively-many robust cuts. On

Multiple-Length Cutting-Stock problem, we will confirm an idea already hinted in the main paper: in many cases, if the separation sub-problem can be solved by `Dynamic Programming`, so can be the projection sub-problem. The main difference is that the projection sub-problem usually requires minimizing a ratio instead of a difference. Such a change of objective function does not always induce an important slowdown because it does not necessarily generate an explosion of the number of states if the interior points $\mathbf{x}_{\texttt{it}}$ are chosen very carefully.

The remaining is organized as follows. Section 2 presents the application of the proposed method on a on a robust optimization problem and the (dual) `Column Generation` formulation for *(Multiple-Length) Cutting-Stock*. Section 3 reports numerical results on these problems, followed by conclusions in the fourth section. In appendix, we provide additional insight into the implementation of the projection algorithms studied throughout this work and the main paper [15], *e.g.*, a fast data structure to record a Pareto frontier (needed by our `Dynamic Programming` scheme).

2. Adapting `Projective Cutting-Planes` for robust optimization and *Multiple-Length Cutting-Stock*. We first recall [15, (2.2.1)] that for any feasible $\mathbf{x} \in \mathscr{P}$ and $\mathbf{d} \in \mathbb{R}^n$, the projection sub-problem $\texttt{project}(\mathbf{x} \rightarrow \mathbf{d})$ can be solved by minimizing the fractional program (2.1) below; we will thus instantiate (2.1) on a robust linear problem in Section 2.1 and on *Multiple-Length Cutting-Stock* in Section 2.2.

$$t^* = \min_{\mathbf{a}} \left\{ \frac{c_a - \mathbf{a}^\top \mathbf{x}}{\mathbf{a}^\top \mathbf{d}} : \ (\mathbf{a}, c_a) \in \mathcal{A}, \ \mathbf{d}^\top \mathbf{a} > 0 \right\}. \tag{2.1}$$

2.1. A robust optimization problem. The main idea in robust optimization is to seek an optimal solution that remains feasible if certain constraint coefficients deviate (reasonably) from their nominal values. The robust optimization literature is now constantly growing and the acceptable coefficient deviations can be defined in many ways, *e.g.*, using linear or ellipsoid uncertainty sets. However, to avoid unessential complication, we here focus only on the robustness model from [6]; the reader may refer to this paper for more references, motivations and related ideas. There are two main principles behind this robustness model: (i) the deviation of a coefficient is at most $\delta = 1\%$ of the nominal value (ii) there are at most Γ coefficients that are allowed to deviate in each nominal constraint. The underlying assumption is that the nominal coefficients of a given constraint can not change all at the same time, always in an unfavorable manner.

2.1.1. The model with prohibitively-many constraints and the standard `Cutting-Planes`. Let us first consider a set $\mathcal{A}_{\texttt{nom}}$ of nominal constraints that is small enough to be enumerated in practice, *i.e.*, there is no need of `Cutting-Planes` to solve the nominal version of the problem (with no robustness). We then associate to each $(\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}}$ a prohibitively-large set $\texttt{Dev}_\Gamma(\mathbf{a})$ of *deviation vectors* $\widehat{\mathbf{a}}$, *i.e.*, vectors $\widehat{\mathbf{a}} \in \mathbb{R}^n$ that have at maximum Γ non-zero components and that satisfy $\widehat{a}_i \in \{-\delta a_i, \ 0, \ \delta a_i\} \ \forall i \in [1..n]$, using $\delta = 0.01$ in practice. Each such deviation vector $\widehat{\mathbf{a}}$ yields a robust cut $(\mathbf{a} + \widehat{\mathbf{a}})^\top \mathbf{x} \leq c_a$, so that we can state $(\mathbf{a} + \widehat{\mathbf{a}}, c_a) \in \mathcal{A}$. In theory, each $\widehat{a}_i$ $(\forall i \in [1..n])$ might be allowed to take a fractional value in the interval $[-\delta a_i, \delta a_i]$, thus leading to infinitely-many robust cuts (semi-infinite programming); however, the strongest robust cuts are always obtained when each non-zero $\widehat{a}_i$ is either δa_i or $-\delta a_i$. There are at most $\binom{n}{\Gamma} 2^\Gamma$ deviation vectors for each nominal constraint $(\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}}$, because there are $\binom{n}{\Gamma}$ ways to choose the non-zero components of $\widehat{\mathbf{a}}$ and each one of them can be either positive or negative, hence the 2^Γ factor.

The generic LP (1.1) is instantiated as follows:

$$\min \left\{ \mathbf{b}^\top \mathbf{x} : (\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{x} \leq c_a \ \forall (\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}} \ \forall\, \widehat{\mathbf{a}} \in \texttt{Dev}_\Gamma(\mathbf{a});\ a_i \in [\texttt{lb}_i, \texttt{ub}_i] \ \forall i \in [1..n] \right\} \tag{2.1.1}$$

This is a minimization problem unlike the general LP (2.1), but the main steps of the (standard or new) `Cutting-Planes` method described in Section [15, § 2] remain exactly the same. The only difference is that the feasible solutions (pierce points) determined by `Projective Cutting-Planes` represent upper bounds instead of lower bounds. The last condition $a_i \in [\texttt{lb}_i, \texttt{ub}_i]$ of (2.1.1) constitutes the initial constraints $\mathcal{A}_0$, most instances using $\texttt{lb}_i = 0\ \forall i \in [1..n]$, *i.e.*, the variables are most often non-negative.

We consider a canonical `Cutting-Planes` for the above (2.1.1), based on the following separation sub-problem: given any $\mathbf{x} \in \mathbb{R}^n$, minimize $c_a - (\mathbf{a}+\widehat{\mathbf{a}})^\top \mathbf{x}$ over all $(\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}}$ and over all $\widehat{\mathbf{a}} \in \texttt{Dev}_\Gamma(\mathbf{a})$. For a fixed nominal constraint $(\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}}$, the strongest possible deviation $\widehat{\mathbf{a}}_{\mathbf{x}}^\top \mathbf{x}$ of $(\mathbf{a}, c_a)$ with respect to $\mathbf{x}$ is determined by maximizing $\widehat{\mathbf{a}}_{\mathbf{x}} = arg\ \max \left\{ \widehat{\mathbf{a}}^\top \mathbf{x} : \widehat{\mathbf{a}} \in \texttt{Dev}_\Gamma(\mathbf{a}) \right\}$. To find this $\widehat{\mathbf{a}}_{\mathbf{x}}$, one needs to determine the largest Γ absolute values in the terms of the sum $\mathbf{a}^\top \mathbf{x} = \sum_{i=1}^n a_i x_i$; this way, $\widehat{\mathbf{a}}_{\mathbf{x}}^\top \mathbf{x}$ can be written as a sum of Γ terms of the form $\delta |a_i x_i|$. We next describe how these largest Γ values can be determined by a partial-sorting algorithm of linear complexity.

Remark 8. *If Γ is a fixed parameter, the largest Γ entries in a table of n values (e.g., such as $|a_1 x_1|$, $|a_2 x_2|$, $\ldots |a_n x_n|$ above) can be determined in $O(n)$ time. We use a partial-sorting algorithm essentially described as follows: iterate over $i \in [1..n]$ and attempt at each step to insert the i^{th} entry in the list of the highest Γ values; this operation takes constant time using the appropriate list data structure.*[20] *In practice, however, the repeated use of this algorithm takes around 15% of the total running time for $\Gamma \geq 10$.* □

Compared to the above `Cutting-Planes`, the algorithm from [6] is slightly different because it returns multiple robust cuts at each separation call. This idea might be very effective in practice, both for the standard `Cutting-Planes` and the `Projective Cutting-Planes`. However, for now, the goal of this study is to compare the projection and the separation sub-problems in a standard setting, and so, we prefer a canonical approach with a unique (robust) cut per iteration.

2.1.2. Implementing the `Projective Cutting-Planes`. Before presenting the projection algorithm (Section 2.1.3 next), let us first discuss the overall `Projective Cutting-Planes` for the robust optimization problem (2.1.1). In fact, if we consider the projection algorithm as a black-box component, the implementation of `Projective Cutting-Planes` becomes rather straightforward, simply following the steps indicated in [15, § 2].

The only slightly problematic question is how to select the interior point $\mathbf{x}_{\texttt{it}}$ at each iteration $\texttt{it} \geq 1$. In common with most problems studied in this work, experiments suggest that it is not very efficient to define $\mathbf{x}_{\texttt{it}}$ as the best feasible solution

[20] This list of the largest Γ values is recorded in a self-balancing binary tree, as implemented in the `C++ std::multiset` data structure. At each iteration i, the partial-sorting algorithm has to check if the current value v_{new} is larger than the minimum value $v_{\min}$ recorded in the tree. If this is the case, the insertion of v_{new} may make the tree size exceed Γ, and so, $v_{\min}$ has to be removed. Each insertion and each removal takes constant time with regards to n, by considering Γ as a parameter. However, these operations can still lead to a non-negligible multiplicative constant factor (like $\log(\Gamma)$) in the complexity of the partial sorting algorithm; this explains how this partial sorting can take 15% of the total running time of the overall `Cutting-Planes`.

found up to the iteration $\mathtt{it}$ (*i.e.*, the last pierce point $\mathbf{x}_{\mathtt{it}} = \mathbf{x}_{\mathtt{it}-1} + t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$). Although such an aggressive `Projective Cutting-Planes` variant could find better feasible solutions in the beginning, it may eventually need *more iterations in the long run*. For best long-term results, it is certainly better to choose a more interior point $\mathbf{x}_{\mathtt{it}}$, not too close to the boundary of $\mathscr{P}$, enabling the inner solutions $\mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{x}_3, \ldots$ to follow a central path (a similar concept is used in some interior point algorithms). As such, we define $\mathbf{x}_{\mathtt{it}}$ using the formula $\mathbf{x}_{\mathtt{it}} = \mathbf{x}_{\mathtt{it}-1} + \alpha t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$ with $\alpha = 0.1\ \forall \mathtt{it} > 1$.

To construct an initial feasible solution $\mathbf{x}_1$, one could be tempted to try $\mathbf{x}_1 = \mathbf{0}_n$, but this is sometimes impossible because $\mathbf{0}_n$ may be infeasible. However, it is not difficult to generate $\mathbf{x}_1$ by constructing a feasible solution in a relatively simple LP whose feasible area stays (deeply) inside the feasible area of (2.1.1). We construct this (deeply) inner LP as follows: for each $(\mathbf{a}, c_a) \in \mathcal{A}_{\mathtt{nom}}$, we insert a constraint $\mathbf{a}^\top\mathbf{x} + \delta|\mathbf{a}|^\top\mathbf{x} \leq c_a$, where $|\mathbf{a}| = [|a_1|\ |a_2|\ \ldots |a_n|]^\top$. If $\mathbf{x}$ is non-negative (as in most instances), than any solution $\mathbf{x}$ that satisfies $\mathbf{a}^\top\mathbf{x} + \delta|\mathbf{a}|^\top\mathbf{x} \leq c_a\ \forall(\mathbf{a},\ c_a) \in \mathcal{A}_{\mathtt{nom}}$ is feasible with regards to all robust cuts — because a robust cut uses a deviation vector $\widehat{\mathbf{a}}$ that satisfies $\widehat{\mathbf{a}} \leq \delta|\mathbf{a}|$, so that $(\mathbf{a}+\widehat{\mathbf{a}})^\top\mathbf{x} \leq \mathbf{a}^\top\mathbf{x} + \delta|\mathbf{a}|^\top\mathbf{x} \leq c_a$.[21] Finally, the first direction $\mathbf{d}_1$ points to the solution of the nominal problem, *i.e.*, we take $\mathbf{d}_1 = \mathtt{opt}(\mathscr{P}_0) - \mathbf{x}_1$, where $\mathscr{P}_0$ is the polytope of the nominal problem with no robust cut.

2.1.3. Solving the projection sub-problem. Based on (2.1), the projection sub-problem reduces to minimizing $\dfrac{c_a - (\mathbf{a}+\widehat{\mathbf{a}})^\top\mathbf{x}}{(\mathbf{a}+\widehat{\mathbf{a}})^\top\mathbf{d}}$ over all nominal constraints $(\mathbf{a}, c_a) \in \mathcal{A}_{\mathtt{nom}}$ and over all deviation vectors $\widehat{\mathbf{a}} \in \mathtt{Dev}_\Gamma(\mathbf{a})$ such that $(\mathbf{a}+\widehat{\mathbf{a}})^\top\mathbf{d} > 0$. Just as the separation algorithm, the projection algorithm iterates over all nominal constraints $\mathcal{A}_{\mathtt{nom}}$, in an attempt to reduce the above ratio (the step length) at each $(\mathbf{a}, c_a) \in \mathcal{A}_{\mathtt{nom}}$; for each $(\mathbf{a}, c_a) \in \mathcal{A}_{\mathtt{nom}}$ it is possible to find several increasingly stronger $\widehat{\mathbf{a}}$ that gradually decrease the above ratio. Let t^*_i denote the optimal step length obtained after considering the robust cuts associated to the first i constraints from $\mathcal{A}_{\mathtt{nom}}$. It is clear that t^*_i can only decrease as i grows. Starting with $t_0 = 1$, the projection algorithm determines t^*_i from t^*_{i-1} by applying the following five steps:

1. Set $t = t^*_{i-1}$ and let $(\mathbf{a}, c_a)$ denote the i^{th} constraint from $\mathcal{A}_{\mathtt{nom}}$.
2. Determine the strongest deviation vector $\widehat{\mathbf{a}}_t$ with respect to $\mathbf{x}+t\mathbf{d}$ by maximizing:
$$\widehat{\mathbf{a}}_t = arg\ \max\left\{\widehat{\mathbf{a}}^\top(\mathbf{x}+t\mathbf{d}) : \widehat{\mathbf{a}} \in \mathtt{Dev}_\Gamma(\mathbf{a})\right\}. \tag{2.1.2}$$
For this, one has to extract the largest Γ absolute values from the terms of the sum $\mathbf{a}^\top(\mathbf{x}+t\mathbf{d})$; we apply the partial-sorting algorithm used for the separation sub-problem in Remark 8.
3. If $(\mathbf{a}+\widehat{\mathbf{a}}_t)^\top(\mathbf{x}+t\mathbf{d}) \leq c_a$, then $\mathbf{x}+t\mathbf{d}$ is feasible with regards to the first i constraints from $\mathcal{A}_{\mathtt{nom}}$ (and the associated robust cuts), because any deviation vector $\widehat{\mathbf{a}} \in \mathtt{Dev}_\Gamma(\mathbf{a})$ satisfies $\widehat{\mathbf{a}}^\top(\mathbf{x}+t\mathbf{d}) \leq \widehat{\mathbf{a}}_t^\top(\mathbf{x}+t\mathbf{d})$. In this case, the final value $t^*_i = t$ has been obtained and the algorithm terminates for this value of i. Otherwise, the robust cut $(\mathbf{a}+\widehat{\mathbf{a}}_t, c_a)$ leads to a smaller feasible step length:

[21] In fact, even for the instances with some negative variables this procedure could still lead to feasible solutions $\mathbf{x}_1$ in practice. We also noticed this (deeply) inner LP can remain feasible by replacing $\mathbf{a}^\top\mathbf{x} + \delta|\mathbf{a}|^\top\mathbf{x} \leq c_a$ with $\mathbf{a}^\top\mathbf{x} + 2\delta|\mathbf{a}|^\top\mathbf{x} + \Delta \leq c_a$, for some small $\Delta > 0$. The use of this parameter Δ makes the generated solutions $\mathbf{x}_1$ even more deeply interior, pushing them away from the boundary; experiments suggest it is usually better to start from such (well-centered) solutions rather than from a boundary point. This is in line with similar ideas in interior point algorithms for standard LP, *i.e.*, it is better to start out with very interior points associated to high barrier terms and to converge towards the boundary only at the end of the solution process, when the barrier terms converge to zero.

$$t' = \frac{c_a - (\mathbf{a} + \widehat{\mathbf{a}}_t)^\top \mathbf{x}}{(\mathbf{a} + \widehat{\mathbf{a}}_t)^\top \mathbf{d}} < t. \tag{2.1.3}$$

4. If $t' = 0$, then the overall projection algorithm returns $t^* = 0$ without checking the remaining nominal constraints, because it is not possible to return a step length below 0 since $\mathbf{x}$ is feasible. In practice, we used the condition "if $t < 10^{-6}$" because very small step lengths usually represent numerical computation errors.
5. Set $t = t'$ and repeat from Step 2 (without incrementing i). The underlying idea is that the deviation vector $\widehat{\mathbf{a}}_t$ determined via (2.1.2) is not the strongest one with regards to $\mathbf{x}+t'\mathbf{d}$, because $\widehat{\mathbf{a}}_t$ generates the highest deviation in (3.1.2) with regards to a different point (*i.e.*, $\mathbf{x} + t\mathbf{d}$). But there might exist a different robust cut $(\mathbf{a}+\widehat{\mathbf{a}}_{t'}, c_a)$ for the same nominal constraint such that $\widehat{\mathbf{a}}_{t'}^\top (\mathbf{x} + t'\mathbf{d}) > \widehat{\mathbf{a}}_t^\top (\mathbf{x} + t'\mathbf{d})$. This could further reduce the step length below t', proving that $\mathbf{x}+t'\mathbf{d}$ is infeasible.

By sequentially applying the above steps to all constraints $(\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}}$ one by one, the step length returned at the last constraint of $\mathcal{A}_{\texttt{nom}}$ provides the sought t^* value.

2.1.3.1. Comparing the running times of the projection and the separation algorithms. In theory, the above projection algorithm could repeat many times Steps 2-5 for each i, iteratively decreasing t in a long loop. However, experiments suggest that long loops arise only rarely in practice; the value of t is typically decreased via (2.1.3) only a dozen of times at most *for all* (thousands of) nominal constraints, *i.e.*, for *all* i. For many nominal constraints $(\mathbf{a}, c_a) \in \mathcal{A}_{\texttt{nom}}$, the above algorithm only concludes at Step 3 that $\mathbf{x} + t\mathbf{d}$ does respect all robust cuts associated to $(\mathbf{a}, c_a)$ so that the only needed calculations are the partial-sorting algorithm (called once at Step 2) and several simple `for` loops over $[1..n]$.

Furthermore, the overall projection algorithm can even stop earlier without scanning all nominal constraints, by returning $t^* = 0$ at Step 4. An exact separation algorithm could not stop earlier, because $c_a - (\mathbf{a} + \widehat{\mathbf{a}}_{\mathbf{x}})^\top \mathbf{x}$ can certainly decrease up to the last nominal constraint $(\mathbf{a}, c_a)$. As such, the projection algorithm can become even faster than the separation one in certain cases. Indeed, for the last (very large) instance from Table 4 with $\Gamma = 50$, a separation iteration takes around 0.62 seconds (in average), while the projection one takes 0.56 seconds (in average). At the other end of the spectrum, for an instance like `nesm` with $\Gamma = 50$, an intersection iteration can take about 30% more time than a separation one. All things considered, one can say that the running time of the above intersection algorithm is similar to that of the separation algorithm.

Finally, the computational speed of the proposed projection algorithm can not be achieved by simply calling the separation algorithm multiple times. An approach based on repeated separation would make the projection algorithm *at least* twice as slow as the separation one: a first call to the separation algorithm would find a first robust cut satisfied with equality by some $\mathbf{x}+t\mathbf{d}$ and then one needs *at least* a second call to check if $\mathbf{x} + t\mathbf{d}$ can be further separated to decrease t. Experiments suggest that a third or a fourth call is often needed in practice. More generally, one of the goals of this work is to explore techniques that can bring us (very) close to designing a projection algorithm as fast as the separation one.

2.2. Multiple-Length Cutting-Stock.

2.2.1. The model with prohibitively-many constraints and the standard `Cutting-Planes`. *Cutting-Stock* is one of the most celebrated problems usually solved by `Column Generation`, as first proposed in the pioneering work of Gilmore and

Gomory in the 1960s. Given a stock of standard-size pieces (*e.g.*, of wood or paper), this problem asks to cut these standard pieces into smaller pieces (items) to fulfill a given demand. The pattern–oriented formulation of *Cutting-Stock* consists of a primal program with prohibitively-many variables, using one variable for each feasible (cutting) pattern. After applying a linear relaxation on this primal program, one obtains the following dual LP:

$$\left.\begin{array}{rll} & \max \mathbf{b}^\top \mathbf{x} & \\ y_a: & \mathbf{a}^\top \mathbf{x} \leq c_a, & \forall (\mathbf{a}, c_a) \in \mathcal{A} \\ & \mathbf{x} \geq \mathbf{0}_n & \end{array}\right\} \mathscr{P} \tag{2.2.1}$$

The notations from (2.2.1) can be directly interpreted in *(Multiple-Length) Cutting-Stock* terms. Each constraint $(\mathbf{a}, c_a) \in \mathcal{A}$ is associated to a primal column representing a (cutting) pattern $\mathbf{a} \in \mathbb{Z}_+^n$ such that a_i is the number of items i to be cut from a standard-size piece (for any item $i \in [1..n]$). Considering a vector $\mathbf{w} \in \mathbb{Z}_+^n$ of item lengths, all feasible patterns $\mathbf{a} \in \mathbb{Z}_+^n$ have to satisfy $\mathbf{w}^\top \mathbf{a} \leq W$, assuming W is the unique length of the standard-size pieces. The vector $\mathbf{b} \in \mathbb{Z}_+^n$ represents the demands for the n items. Writing the primal LP associated to (2.2.1), one can see how the primal objective function asks to minimize the total cost of the selected patterns.

In pure *Cutting-Stock*, all feasible patterns $(\mathbf{a}, c_a) \in \mathcal{A}$ have a fixed unitary cost $c_a = 1$, but we will focus on the more general *Multiple-Length Cutting-Stock* in which the input standard-size pieces can actually have different lengths and different costs. While all discussed algorithms could address an arbitrary number of lengths, we prefer to avoid unessential complication and to consider two lengths $0.7W$ and W of costs 0.6 and resp. 1. The cost of a pattern $\mathbf{a}$ is thus the cost of the smallest standard-size piece that can accommodate $\mathbf{a}$, *e.g.*, if $\mathbf{w}^\top \mathbf{a} \leq 0.7W$ then $c_a = 0.6$, else $c_a = 1$.

The standard `Column Generation` method is equivalent to a `Cutting-Planes` algorithm that optimizes the above LP (2.2.1) by iteratively solving the separation subproblem $\min_{(\mathbf{a},c_a)\in\mathcal{A}} c_a - \mathbf{a}^\top \mathbf{x}$ on the current optimal outer solution $\mathbf{x} = \texttt{opt}(\mathscr{P}_{\texttt{it}})$ at each iteration `it`. In *(Multiple-Length) Cutting-Stock*, this sub-problem is typically solved by `Dynamic Programming`. In a nutshell, the main idea is to assign a state s_ℓ for each feasible length $\ell \in [1..W]$; all patterns of length ℓ have the same cost c_ℓ and the pattern $\mathbf{a}_\ell \in \mathbb{Z}_+^n$ that minimizes $c_\ell - \mathbf{a}_\ell^\top \mathbf{x}$ gives the objective value of s_ℓ, *i.e.*, $\texttt{obj}(s_\ell) = c_\ell - \mathbf{a}_\ell^\top \mathbf{x}$. The `Dynamic Programming` scheme generates transitions among such states, and, after calculating them all, returns $\min_{\ell\in[1..W]} c_\ell - \mathbf{a}_\ell^\top \mathbf{x}$ in the end.

2.2.2. Adapting `Projective Cutting-Planes` for Multiple-Length Cutting-Stock. The `Projective Cutting-Planes` was designed in [15, § 2] as a rather generic methodology that allows a certain flexibility and different variations. We now need a few customizations to make it reach its full potential on *Multiple-Length Cutting-Stock*. As with other problems explored in this work, a key observation is that defining $\mathbf{x}_{\texttt{it}}$ as the best solution ever found up to iteration `it` is not efficient in the long run, partly because $\mathbf{x}_{\texttt{it}}$ could fluctuate too much from iteration to iteration.

However we did perform experiments for such a $\mathbf{x}_{\texttt{it}}$ choice, using the formula $\mathbf{x}_{\texttt{it}} = \mathbf{x}_{\texttt{it}-1} + t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$. This choice leads to an aggressive `Projective Cutting-Planes` variant that starts very well by strictly increasing the lower bound with each iteration `it`, *i.e.*, check that $\mathbf{b}^\top \mathbf{x}_{\texttt{it}} = \mathbf{b}^\top \left(\mathbf{x}_{\texttt{it}-1} + t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}\right) \geq \mathbf{b}^\top \mathbf{x}_{\texttt{it}-1}$ is surely satisfied because the objective function does not deteriorate by advancing along $\mathbf{x}_{\texttt{it}-1} \rightarrow \mathbf{d}_{\texttt{it}-1}$ (see also Step 2 from [15, § 2]). This `Projective Cutting-Planes` variant has the advantage that the lower bound $\mathbf{b}^\top \mathbf{x}_{\texttt{it}}$ becomes constantly increasing along the iterations `it`, eliminating the infamous "yo-yo" effect

appearing very often (if not always) in `Column Generation`. However, our preliminary experiments (available on-line, see Figure 6, p. 15) suggest that this aggressive choice leads to more iterations in the long run. Furthermore, we will also see in Section 2.2.3.2 that the projection sub-problem $\texttt{project}(\mathbf{x} \to \mathbf{d})$ can be solved more rapidly when $\mathbf{x}$ is a "truncated" solution, *e.g.*, when x_i is a multiple of $\gamma = 0.2$ for each $i \in [1..n]$.

For this reasons, we decided to propose a different `Projective Cutting-Planes` variant, defining $\mathbf{x}_{\texttt{it}}$ using the following approach. Let us first introduce the operator $\lfloor \mathbf{x} \rfloor$ that truncates $\mathbf{x}$ down to multiples of some $\gamma \in \mathbb{R}_+$ (we used $\gamma = 0.2$), *i.e.*, x_i becomes $\gamma \cdot \lfloor \frac{1}{\gamma} x_i \rfloor$ for any $i \in [1..n]$. Let $\mathbf{x}^{\text{bst}}$ denote the best truncated feasible solution generated up to the current iteration; $\mathbf{x}^{\text{bst}}$ can be determined as follows: start with $\mathbf{x}^{\text{bst}} = \mathbf{0}_n$ at iteration $\texttt{it} = 1$, and replace $\mathbf{x}^{\text{bst}}$ with $\lfloor \mathbf{x}_{\texttt{it}} + t^*_{\texttt{it}} \mathbf{d}_{\texttt{it}} \rfloor$ at each iteration $\texttt{it} > 1$ where $\mathbf{b}^\top \lfloor \mathbf{x}_{\texttt{it}} + t^*_{\texttt{it}} \mathbf{d}_{\texttt{it}} \rfloor > \mathbf{b}^\top \mathbf{x}^{\text{bst}}$. We propose to choose the inner solution $\mathbf{x}_{\texttt{it}}$ at each iteration $\texttt{it}$ based on the following rules:

- set $\mathbf{x}_{\texttt{it}} = \mathbf{0}_n$ in half of the cases (half of the iterations);
- set $\mathbf{x}_{\texttt{it}} = \mathbf{x}^{\text{bst}}$ in 25% of the cases;
- set $\mathbf{x}_{\texttt{it}} = \lfloor \frac{1}{2} \mathbf{x}^{\text{bst}} \rfloor$ in 25% of the case.

Regarding the iterations $\texttt{it} = 1$ and $\texttt{it} = 2$, let us choose $\mathbf{x}_1 = \mathbf{0}_n$ and $\mathbf{d}_1 = \frac{1}{W}\mathbf{w}$, and resp. $\mathbf{x}_2 = \mathbf{0}_n$ and $\mathbf{d}_2 = \mathbf{b}$. The choice of projecting along $\mathbf{0}_n \to \frac{1}{W}\mathbf{w}$ at the very first iteration is inspired by research in dual feasible functions for *Cutting-Stock* problems [4], which shows that $\frac{1}{W}\mathbf{w}$ is often a dual-feasible solution (in pure *Cutting-Stock*) of very high quality. The choice at iteration 2 is a rather standard one, since the projection towards $\mathbf{b}$ enables one to advance along the direction with the fastest rate of objective function improvement. By solving these two sub-problems, the `Projective Cutting-Planes` also generates a few initial constraints in (2.2.1). To ensure an unbiased comparison, our standard `Column Generation` algorithm also generates such initial constraints in the beginning, *i.e.*, it solves the separation sub-problem on $\mathbf{b}$ and $\frac{1}{W}\mathbf{w}$ before launching the standard iterations.

2.2.3. Solving the Projection Sub-problem. Numerous `Column Generation` algorithms for cutting and packing problems rely on `Dynamic Programming` (DP) to solve the separation sub-problem. And, in many such cases, if the separation sub-problem can be solved by `Dynamic Programming`, so can be the projection one.

Given a feasible $\mathbf{x} \in \mathscr{P}$ in (2.2.1) and a direction $\mathbf{d} \in \mathbb{R}^n$, recall that the projection subproblem $\texttt{project}(\mathbf{x} \to \mathbf{d})$ asks to minimize (2.1). For *Multiple-Length Cutting-Stock*, (2.1) is instantiated as follows:

$$t^* = \min_{\mathbf{a}} \left\{ \frac{f(\mathbf{w}^\top \mathbf{a}) - \mathbf{a}^\top \mathbf{x}}{\mathbf{d}^\top \mathbf{a}} : \ \mathbf{a} \in \mathbb{Z}^n_+, \ \mathbf{w}^\top \mathbf{a} \leq W, \ \mathbf{d}^\top \mathbf{a} > 0 \right\}, \tag{2.2.2}$$

where the function $f : [0, W] \to \mathbb{R}_+$ maps each $\ell \in [0, W]$ to the cost of the cheapest (shortest) standard–size piece of length at least ℓ available in stock. The DP scheme proposed next can work for any non-decreasing function f, *i.e.*, under the natural assumption that shorter pieces are cheaper than longer pieces. Such functions f can encode many different *Cutting-Stock* variants, like variable–sized bin–packing or elastic cutting stock; see more examples in [13, §4.1.1].

2.2.3.1. The main DP algorithm, the state definition and the state transitions. We consider a set $\mathcal{S}_\ell$ of DP states for every feasible length $\ell \in [0..W]$. Each state $\mathbf{s} \in \mathcal{S}_\ell$ is associated to all patterns $\mathbf{a} \in \mathcal{A}$ of

(a) length $\mathbf{s}_{\texttt{len}} = \mathbf{w}^\top \mathbf{a} = \ell$;
(b) cost $\mathbf{s}_c = f(\mathbf{w}^\top \mathbf{a}) - \mathbf{a}^\top \mathbf{x} = f(\ell) - \mathbf{a}^\top \mathbf{x}$;

(c) profit $\mathtt{s}_p = \mathbf{d}^\top \mathbf{a}$.

All states in $\mathcal{S}_\ell$ have the same length ℓ but their cost and profit can vary. Under this cost/profit interpretation, (2.2.2) reduces to minimizing the cost/profit ratio $\mathtt{obj}(\mathtt{s}) = \dfrac{\mathtt{s}_c}{\mathtt{s}_p}$ over all states $\mathtt{s}$ ever generated, *i.e.*, $\min\left\{\mathtt{obj}(\mathtt{s}) = \frac{\mathtt{s}_c}{\mathtt{s}_p} : s \in \mathcal{S}_\ell,\ \ell \in [0..W]\right\}$. Notice any feasible pattern can be associated to a state, although we will see that certain states are dominated and do not need to be recorded. Finally, the above cost $\mathtt{s}_c = f(\mathbf{x}^\top \mathbf{a}) - \mathbf{a}^\top \mathbf{x}$ is always non-negative because $\mathbf{x} \in \mathscr{P}$.

The proposed DP algorithm starts only with an initial null state of length 0, cost 0 and profit 0. It then performs a DP iteration for each item $i \in [1..n]$; if $b_i > 1$, this iteration is performed b_i times because a pattern can cut up to b_i copies of item i (and there is no use in exceeding the item demand b_i). Each such DP iteration generates transitions from the current states to update other (or produce new) states. A state transition $\mathtt{s} \to \mathtt{s}'$ associated to an item i leads to a state $\mathtt{s}'$ such that:

(a) $\mathtt{s}'_{\mathtt{len}} = \mathtt{s}_{\mathtt{len}} + w_i$, *i.e.*, the length simply increases by adding a new item;

(b) $\mathtt{s}'_p = \mathtt{s}_p + d_i$, *i.e.*, we add the profit of item i;

(c) $\mathtt{s}'_c = \mathtt{s}_c + f(\mathtt{s}'_{\mathtt{len}}) - f(\mathtt{s}_{\mathtt{len}}) - x_i$, *i.e.*, the term $f(\mathtt{s}'_{\mathtt{len}}) - f(\mathtt{s}_{\mathtt{len}})$ updates the cost of the standard-size stock piece from which the pattern is cut, and $-x_i$ comes from the $-\mathbf{a}^\top \mathbf{x}$ term from the above cost definition $f(\ell) - \mathbf{a}^\top \mathbf{x}$.

Algorithm 1 provides the pseudo-code executed for each item $i \in [1..n]$ considered b_i times. The most complex operation arises at Step 5, where one needs to check that the new state $\mathtt{s}'$ is not dominated by an existing state in $\mathcal{S}_{\ell+w_i}$ before inserting it in $\mathcal{S}_{\ell+w_i}$; the efficient implementation of this step is described in Section 2.2.3.2.

Algorithm 1 The `Dynamic Programming` steps executed b_i times for each item i

1. **for** $\ell = W - w_i$ **to** 0:
2. **for each** $\mathtt{s} \in \mathcal{S}_\ell$: ▷ for each state with length ℓ
3. initialize state $\mathtt{s}'$ with $\mathtt{s}'_{\mathtt{len}} = \ell + w_i$, according to above formula (a)
4. calculate $\mathtt{s}'_p$, $\mathtt{s}'_c$ with above formulae (b) and (c)
5. **if** $\mathtt{s}'$ is not dominated by an existing state in $\mathcal{S}_{\ell+w_i}$ (Section 2.2.3.2) **then**
   - $\mathcal{S}_{\ell+w_i} \leftarrow \mathcal{S}_{\ell+w_i} \cup \{\mathtt{s}'\}$
   - record the transition $\mathtt{s} \to \mathtt{s}'$ (to reconstruct an optimal pattern in the end)

This pseudo-code is a generalization of the separation algorithm. To solve the separation sub-problem on some $\mathbf{d} \in \mathbb{R}^n$, one would only need the following simplification: consider only singleton sets $\mathcal{S}_\ell = \{\mathtt{s}\}$ where $\mathtt{s}$ is a state of cost $\mathtt{s}_c = f(\mathbf{w}^\top \mathbf{a}) = f(\ell)$ and maximum profit $\mathtt{s}_p = \mathbf{d}^\top \mathbf{a}$, *i.e.*, for each length $\ell \in [0, W]$, it is enough to record only the maximum profit state, the cost being fixed to $f(\ell)$. In the end, the separation algorithm simply returns $\min\{\mathtt{s}_c - \mathtt{s}_p :\ \mathtt{s} \in \mathcal{S}_\ell,\ \ell \in [0..W]\}$.

The projection sub-problem is more difficult because recording a unique state per length is no longer enough. To illustrate this, notice that a state with a cost/profit ratio of $\frac{5}{4}$ does not necessarily dominate a state with a cost/profit ratio of $\frac{3}{2}$ only because $\frac{5}{4} < \frac{3}{2}$. Indeed, the $\frac{5}{4}$ state can evolve to a sub-optimal state by following a transition that decreases the cost by 1 and increases the profit by 4 because $\frac{5-1}{4+4} = \frac{4}{8} \not\leq \frac{3-1}{2+4} = \frac{2}{6}$. This could never happen in a (knapsack-like) separation sub-problem, *i.e.*, the relative order of two states defined by cost$-$profit differences would never change because all transitions induce linear (additive) changes to such differences.

2.2.3.2. Reducing the number of DP states to accelerate the DP projection algorithm. Several ideas can be applied to reduce the number of recorded states. First, let us show it is enough to record a unique maximum-profit state for each feasible

cost of a state in each $\mathcal{S}_\ell$. For this, consider two states $\mathtt{s}^*, \mathtt{s} \in \mathcal{S}_\ell$ such that $\mathtt{s}^*_c = \mathtt{s}_c$ and $\mathtt{s}^*_p > \mathtt{s}_p$. The state $\mathtt{s}$ is dominated and can be ignored because any transition(s) equally applied on $\mathtt{s}^*$ and $\mathtt{s}$ would lead to the same cost $\mathtt{s}^*_c + \Delta_c = \mathtt{s}_c + \Delta_c > 0$ and to profits $\mathtt{s}^*_p + \Delta_p > \mathtt{s}_p + \Delta_p$; this way, it is easy to check that $\dfrac{\mathtt{s}^*_c + \Delta_c}{\mathtt{s}^*_p + \Delta_p} < \dfrac{\mathtt{s}_c + \Delta_c}{\mathtt{s}_p + \Delta_p}$ holds — recall we are only interested in positive profit states (positive denominators) because of the condition $\mathbf{d}^\top \mathbf{a} > 0$ from (2.2.2).

Let us now compare $\mathtt{s}^*$ to a state $\mathtt{s} \in \mathcal{S}_\ell$ that satisfies $\mathtt{s}_c > \mathtt{s}^*_c$ and $\mathtt{s}_p \leq \mathtt{s}^*_p$. Such state $\mathtt{s}$ is also dominated by $\mathtt{s}^*$ because it can only lead via transitions to $\dfrac{\mathtt{s}^*_c + \Delta_c}{\mathtt{s}^*_p + \Delta_p} < \dfrac{\mathtt{s}_c + \Delta_c}{\mathtt{s}_p + \Delta_p}$. As such, a state $\mathtt{s} \in \mathcal{S}_\ell$ with a higher cost than an existing state $\mathtt{s}^* \in \mathcal{S}_\ell$ (*i.e.*, $\mathtt{s}_c > \mathtt{s}^*_c$) must have a higher profit to be non-dominated, *i.e.*, a non-dominated state $\mathtt{s}$ such that $\mathtt{s}_c > \mathtt{s}^*_c$ satisfies $\mathtt{s}_p > \mathtt{s}^*_p$. This can be seen as a formalization of a very natural principle "pay a higher cost only when you gain a higher profit". However, the cost and the profits of all non-dominated states in $\mathcal{S}_\ell$ can thus be ordered using a relation of the form:

$$c_1 < c_2 < c_3 < \dots \tag{2.2.3a}$$

$$p_1 < p_2 < p_3 < \dots \tag{2.2.3b}$$

Let us now discuss the length of the lists (2.2.3.a)–(2.2.3.b) that have to be recorded for each $\mathcal{S}_\ell$ $\forall \ell \in [0..W]$. If there are fewer potential costs values, these lists have to be shorter, and so, the total number of states is reduced. Accordingly, if all pattern costs $f(\ell)$ ($\forall \ell \in [0, W]$) are multiples of 0.2 and if we only use truncated solutions $\mathbf{x}_{\mathtt{it}}$ such that all components of $\mathbf{x}_{\mathtt{it}}$ are also multiples of 0.2, the maximum number of feasible costs values is 6, because any state cost has the form $f(\ell) - \mathbf{a}^\top \mathbf{x}$ for some $\mathbf{a} \in \mathbb{Z}^n_+$ and thus it has to belong to $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$. This way, the resulting DP algorithm might often need to record only a few states per length, and so, it is not necessarily significantly slower than a separation DP algorithm recording a unique state per length.

Finally, we need a fast data structure to manipulate lists of cost/profit pairs satisfying (2.2.3.a)–(2.2.3.b), because it is important to accelerate the following two operations executed by Algorithm 1:[22]

(i) iterate over all elements of $\mathcal{S}_\ell$ to implement the `for` loop at Line 2;

(ii) insert a new state at Line 5 after checking that it is not dominated.

Remark 9. *A list of cost/profit values satisfying (2.2.3.a)–(2.2.3.b) can be seen as a Pareto frontier with two objectives (minimize the cost and maximize the profit). It is not difficult to scan the elements of such a frontier to implement the above operation (i). The most computationally-expensive task is to insert a new state for the above operation (ii), because this requires checking if the new state is dominated by an existing state. This not be efficiently checked by naively scanning the whole list of cost/profit values. We propose to record this list in a self-balancing binary tree that can perform many look-up operations in logarithmic time. Furthermore, the insertion of a new non-dominated state can lead to the removal of other existing states that*

[22] To further accelerate the DP, experiments suggest it can be useful (in practice) to sort the items $i \in [1..n]$ in descending order of the value $\frac{w_i}{1+\mathbf{x}^{\text{bst}}_i}$. Precisely, Algorithm 1 is executed for each of the items $[1..n]$ considered in this order. In a loose sense, this amounts to considering that it is better to start with longer items that did not contribute too much to the best truncated inner solution $\mathbf{x}^{\text{bst}}$ ever found.

become dominated. Appendix A.1 describes in detail the self-balancing binary tree that we used to (try to) perform such operations as rapidly as possible. □

3. Numerical Experiments. We here report numerical results on the robust optimization problem and then on two *Multiple-Length Cutting-Stock* variants.

3.1. A Robust optimization problem. We consider the instances from [6] using $\Gamma \in \{1, 10, 50\}$; these instances originate in the `Netlib` or the `Miplib` libraries. In fact, we discarded all instances that are infeasible for $\Gamma = 50$, since our methods are not designed to find infeasibilities. We also ignored all instances that require less than 5 cuts in *loc. cit.* (*i.e.*, `seba`, `shell` and `woodw`) because they are too small to produce meaningful comparisons. We thus remain with a test bed of 21 instances with between $n = 1000$ and $n = 15000$ variables.[23] Recall our robust optimization problem has a minimization objective, so that the inner solutions $\mathbf{x}_{\mathtt{it}}$ determined by `Projective Cutting-Planes` along the iterations `it` generate *upper* bounds $\mathbf{b}^\top \mathbf{x}_{\mathtt{it}}$.

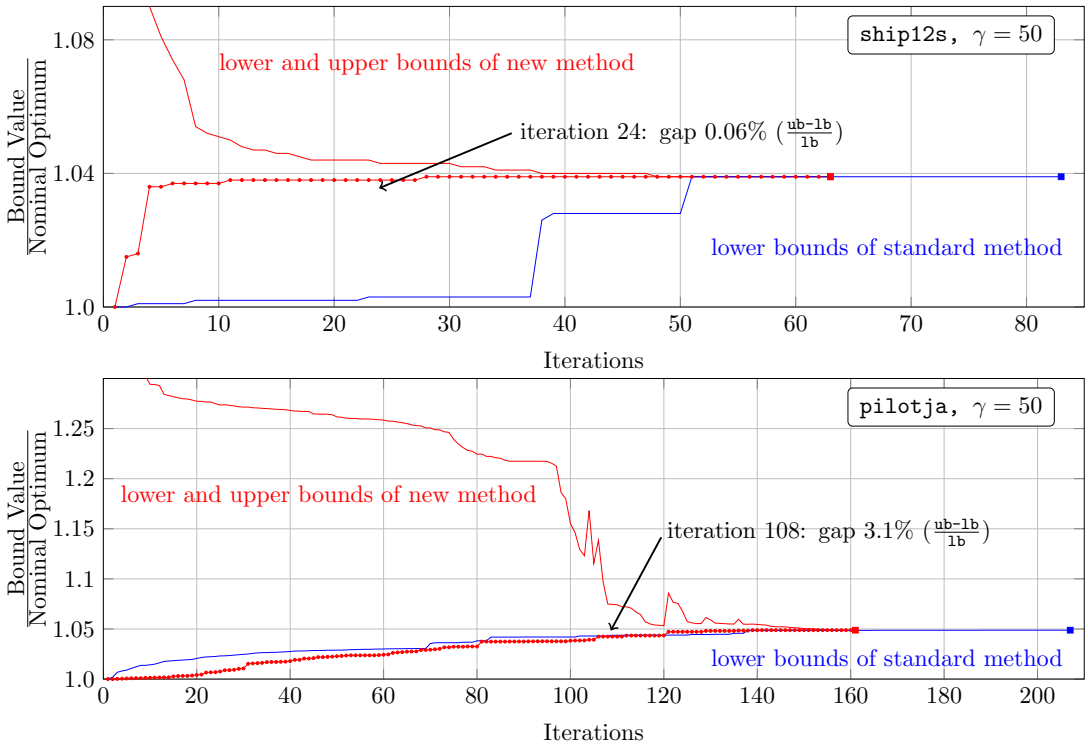

Figure 5: The progress over the iterations of the lower and upper bounds reported by the `Projective Cutting-Planes` (in red), compared to those of the standard `Cutting-Planes` (lower bounds only, in blue).

[23] Most instances have between $n = 1000$ and $n = 5000$ variables and a number of constraints between 500 and 3000. We refer to [6, Table 1] for the nominal objective value of each instance. We mention that `stocfor3` is an exceptionally large instance with $n = 15695$ and more than 15000 constraints. For even greater detail on their characteristics, the instances are publicly available on-line in a human-readable format (the original MPS files are difficult to parse) at cedric.cnam.fr/∼porumbed/projcutplanes/instances-robust.zip.

3.1.1. A general view on the running profile. Figure 5 plots the running profile of the standard `Cutting-Planes` compared to that of the `Projective Cutting-Planes` on two instances. The standard `Cutting-Planes` needed 83 and resp. 207 iterations to fully converge. After only half this number of iterations, the `Projective Cutting-Planes` reported a feasible solution with a proven low gap of 0.06% or resp. 3.1%, as indicated b y the arrows in the figure.

3.1.2. The main tabular results. Table 4 next page compares the total computing effort (iterations and CPU time) needed to fully solve each instance with the new and the standard method. For `Projective Cutting-Planes`, we also provide the computing effort needed to reach a gap of 1% between the lower and the upper bounds; this may require a time of seconds in general, significantly less the standard `Cutting-Planes`. For example, the standard `Cutting-Planes` needed between one and two hours (depending on γ) to determine the optimal solution for the last instance `stocfor3`, while the `Projective Cutting-Planes` reported in less than 3 seconds a feasible solution with a proven gap below 1% (see columns "gap 1%" in bold in the last row). In many practical settings, this could represent a satisfactory feasible solution.[24]

For `sctap2` and `sctap3` with $\Gamma = 50$, the standard `Cutting-Planes` is seriously slowed down by degeneracy issues, *i.e.*, it performs too many Simplex pivots that only change the basis without improving the objective value. It thus needs significantly more iterations than normally expected — see the figures in bold in the rows of `sctap2` and `sctap3`. We suppose that such degeneracy phenomena are also visible for `czprob` with $\Gamma = 50$ in Table 1 of [6], because their algorithm takes 100 more time for $\Gamma = 50$ than for $\Gamma = 10$, which is unusual.

REMARK 10. *Except for the above experiments, the degeneracy issues of the standard* `Cutting-Planes` *are not very visible in other* `Cutting-Planes` *implementations from this work (including in [15]). However, such problems well recognized in the* `Cutting-Planes` *literature, especially in* `Column Generation`*; as [11, §4.2.2] put it, "When the master problem is a set partitioning problem, large instances are difficult to solve due to massive degeneracy [...] Then, the value of the dual variables are no meaningful measure for which column to adjoin to the Reduces Master Problem". In* `Projective Cutting-Planes`*, the inner-outer solutions* $\mathbf{x}_{it}$ *and* $\texttt{opt}(\mathscr{P}_{it-1})$ *represent together a more "meaningful measure" for selecting a new constraint, avoiding iterations that keep the objective value constant. In fact, as hinted at point 2 of [15, § 2], a projection can not keep the objective value constant when* $\mathbf{x}_{it}$ *is strictly interior (which is surely the case when* $\alpha < 1$*). This comes from the fact that the objective value can not deteriorate or remain constant by advancing along* $\mathbf{x}_{it} \rightarrow \mathbf{d}_{it}$*, because* $\mathbf{x}_{it} + \mathbf{d}_{it} = \texttt{opt}(\mathscr{P}_{it-1})$ *and* $\mathbf{x}_{it}$ *belongs to the strict interior of* $\mathscr{P}_{it-1} \supseteq \mathscr{P}$*.*

3.2. Multiple–Length Cutting-Stock. We here consider a *Multiple-Length Cutting-Stock* variant with two types of input standard-size pieces: one of length W and cost 1, and one of length $0.7W$ and cost 0.6. Preliminary experiments confirm that introducing a third type of standard-size piece lead to similar experimental conclusions. We prefer *Multiple-Length Cutting-Stock* over the standard *Cutting-Stock* because: (i) the constraints $(\mathbf{a}, c_a) \in \mathcal{A}$ of the `Column Generation` dual LP (2.2.1) do not satisfy all $c_a = 1$, and (ii) it is not possible to generate lower bounds using the Dual Feasible Functions that proved so effective in standard *Cutting-Stock* [4].

[24] It is also true that the robust optimal solution is always within 103% of the nominal optimum. The value of the starting solution $\mathbf{x}_1$ might be only a few percents higher than the nominal optimum.

	$\gamma = 50$							$\gamma = 10$							$\gamma = 1$						
	OPT	new method				std. method		OPT	new method				std. method		OPT	new method				std. method	
	(+%)	gap 1%		full converg.		full converg.		(+%)	gap 1%		full converg.		full converg.		(+%)	gap 1%		full converg.		full converg.	
Instance		iters	time	iters	time	iters	time		iters	time	iters	time	iters	time		iters	time	iters	time	iters	time
25fv47	2.548	146	1.168	149	1.191	199	1.265	2.541	158	1.188	169	1.273	204	1.455	1.457	135	1.023	147	1.11	149	1.12
bnl2	1.847	486	13.66	491	13.81	1927	48.13	1.84	703	20.92	708	21.08	1295	31.31	0.7903	501	14.47	504	14.56	552	12.66
czprob	0.6401	61	0.485	734	32.3	1293	58.52	0.3749	32	0.181	125	0.899	170	1.302	0.1223	14	0.0935	25	0.196	25	0.124
ganges	0.4736	1	<0.001	25	0.059	25	0.059	0.4302	1	<0.001	31	0.085	33	0.074	0.0531	1	<0.001	25	0.063	25	0.049
gfrd-pnc	0.0649	64	0.1404	64	0.141	64	0.092	0.0649	64	0.1086	64	0.109	64	0.090	0.0592	67	0.1415	67	0.142	67	0.100
maros	12.12	272	2.402	278	2.458	379	3.518	12.11	281	2.508	300	2.683	395	3.486	5.76	200	1.812	219	1.983	227	1.714
nesm	0.8752	56	0.579	80	0.798	80	0.659	0.8752	56	0.604	80	0.824	80	0.658	0.4515	58	0.647	82	0.880	82	0.639
pilotja	4.877	121	2.418	161	3.042	207	3.701	4.815	125	2.316	172	3.074	179	3.418	2.344	110	1.522	135	1.908	143	1.693
pilotnov	8.51	96	4.227	120	4.615	119	3.714	8.51	103	6.12	139	6.912	141	4.915	4.402	94	3.355	120	3.805	119	1.776
pilotwe	6.109	102	0.97	118	1.102	143	1.204	6.108	102	1.045	119	1.19	144	1.308	3.193	98	0.853	115	1.005	124	1.066
scfxm2	2.114	93	0.387	139	0.584	146	0.537	2.113	101	0.401	152	0.603	150	0.498	0.9889	88	0.357	131	0.536	142	0.486
scfxm3	2.142	139	0.957	196	1.353	215	1.27	2.141	142	0.955	227	1.575	222	1.309	0.977	91	0.605	197	1.352	213	1.216
sctap2	2.844	185	1.946	242	2.567	**6545**	147.3	2.814	332	3.685	696	8.4	954	10.62	1.533	191	2.035	353	3.88	302	2.644
sctap3	3.04	145	2.649	239	4.55	**9463**	366.1	2.995	180	3.45	773	15.49	1168	20.22	1.602	213	3.785	406	7.394	347	4.799
ship08l	0.1244	1	0.002	20	0.111	29	0.171	0.1157	1	0.002	19	0.128	23	0.134	0.0300	1	0.002	19	0.127	24	0.147
ship08s	0.1396	2	0.006	32	0.122	42	0.139	0.129	2	0.006	34	0.134	35	0.123	0.0317	1	0.001	32	0.123	38	0.127
ship12l	0.3528	1	<0.001	48	0.442	65	0.576	0.3462	1	<0.001	48	0.418	65	0.555	0.0600	1	0.004	45	0.451	56	0.483
ship12s	0.3898	4	0.015	63	0.377	83	0.376	0.3857	5	0.019	64	0.387	86	0.398	0.0617	4	0.015	58	0.305	63	0.281
sierra	0.0239	1	0.001	54	0.414	61	0.567	0.0239	1	0.001	54	0.412	61	0.569	0.0223	1	0.004	51	0.538	51	0.483
stocfor2	1.522	6	0.022	437	5.047	484	6.67	1.522	7	0.025	438	5.387	486	6.562	0.7588	3	0.054	438	7.573	712	10.3
stocfor3	1.482	29	**2.192**	3777	2125	4329	2701	1.482	32	**1.862**	3781	2029	4330	2851	0.7327	1	**0.99**	3720	3023	6069	3482

Table 4: Results of `Projective Cutting-Planes` and standard `Cutting-Planes` on robust optimization instances. The columns OPT indicate the increase in percentage of the robust objective value with respect to the nominal one (with no robustness). Columns "gap 1%" indicate the computing effort needed to reach the iteration `it` when the gap between the upper bound $\mathbf{b}^\top \mathbf{x}_{\mathtt{it}}$ and the lower bound $\mathtt{optVal}(\mathscr{P}_{\mathtt{it}})$ is below 1%, *i.e.*, either $0 < \mathtt{optVal}(\mathscr{P}_{\mathtt{it}}) \leq \mathbf{b}^\top \mathbf{x}_{\mathtt{it}} \leq 1.01\mathtt{optVal}(\mathscr{P}_{\mathtt{it}})$ or $\mathtt{optVal}(\mathscr{P}_{\mathtt{it}}) \leq \mathbf{b}^\top \mathbf{x}_{\mathtt{it}} \leq 0.99\mathtt{optVal}(\mathscr{P}_{\mathtt{it}}) < 0$.

Table 5 compares the `Projective Cutting-Planes` (from Section 2.2.2) to the standard `Column Generation` on thirty instances from the literature.[25] Column 1 represents the instance, Column 2 indicates the optimal value of (2.2.1), Columns 3–6 report the results of the new method, and Columns 7–10 provide the results of the standard `Column Generation`. For both methods, Table 5 first indicates the computing effort (iterations and CPU time) needed to reach a gap of 20% (*i.e.*, so that `ub`$\leq 1.2\cdot$`lb`) and then the total computing effort needed to fully converge.

Instance	OPT	`Projective Cutting-Planes`				Standard `Column Generation`			
		gap 20%		full convergence		gap 20%		full convergence	
		iters	time[s]	iters	time[s]	iters	time[s]	iters	time[s]
`m01-1`	49.3	90	0.02	166	0.05	187	0.07	194	0.08
`m01-2`	53	82	0.02	140	0.04	171	0.06	202	0.07
`m01-3`	48.2	70	0.02	134	0.04	180	0.07	212	0.08
`m20-1`	56.6	79	0.02	101	0.03	101	0.03	148	0.04
`m20-2`	58.7	73	0.02	**103**	0.02	123	0.04	**175**	0.05
`m20-3`	64.8	61	0.01	116	0.02	118	0.03	136	0.03
`m35-1`	73.9	61	0.01	61	0.01	64	0.01	64	0.01
`m35-2`	71.5	125	0.02	125	0.02	143	0.02	143	0.02
`m35-3`	73.7	67	0.01	67	0.01	82	0.01	82	0.01
`vb50c1-1`	866.3	46	0.8	82	2.2	83	5.5	113	8.3
`vb50c1-2`	842.5	39	1.6	86	2.5	91	7.6	121	9.6
`vb50c1-3`	860.2	37	1.5	85	3.1	87	6.9	115	9.5
`vb50c2-1`	672.3	55	2.2	114	9.8	82	13.1	127	20.2
`vb50c2-2`	593.1	40	1.9	**80**	5.1	88	11	**139**	21.1
`vb50c2-3`	480.048	36	3.5	181	47.2	75	20.6	216	76.3
`vb50c3-1`	282	37	11.7	122	57.6	67	36.1	179	105
`vb50c3-2`	239.398	37	16.8	115	64.6	60	30.6	145	85.1
`vb50c3-3`	271.398	36	12.9	132	65.3	68	38.2	173	109
`vb50c4-1`	579.548	40	3.5	115	17.5	73	12.5	158	35.5
`vb50c4-2`	551.01	36	3	123	21.9	73	18.5	166	46.6
`vb50c4-3`	700.039	40	2.3	111	9.9	81	11.9	147	24.8
`vb50c5-1`	337.8	40	8.7	**133**	51.9	61	24.8	**228**	109
`vb50c5-2`	349.799	30	4.8	**130**	44.1	64	21	**207**	81.4
`vb50c5-3`	295.775	36	11	115	53.6	71	28.4	177	83.9
`wäscher-1`	24.0648	71	0.2	319	4.2	294	2.3	483	4.7
`wäscher-2`	22.0003	69	0.2	501	8.6	158	1	481	6.7
`wäscher-3`	12.1219	31	0.03	110	0.3	110	0.3	170	0.5
`hard-sch-1`	51.4254	112	14.7	**345**	69.2	345	48.1	**712**	115
`hard-sch-2`	51.4426	116	15.1	**339**	67	365	50.9	**685**	110
`hard-sch-3`	50.5957	110	15.1	**295**	58.6	357	52.8	**630**	107

Table 5: The new method compared to the standard method on *Multiple-Length Cutting-Stock*. The `Column Generation` needs at least 60% more iterations on roughly a quarter of instances (see bold figures in Columns 5 and 9). All reported CPU times are smaller than those reported in the companion paper (Section 2, p. 6) of [16], for both the new method and the standard `Column Generation`. This can not only be explained by the hardware evolution, but also by a better implementation.

Table 5 demonstrates that `Projective Cutting-Planes` reaches the 20% gap

[25] We use ten instance sets, taking the first 3 instances from each set. For each set, the number (ID) of each individual instance is indicated by a suffix, *e.g.*, we write `m01-1`, `m01-2`, `m01-3` to refer to the first, second and third instance respectively from the set `m01`. The characteristics of the instances (*i.e.*, the values of n, W, $\mathbf{b}$, etc) and their origins are described in Table 1 from [16].

three or four times more rapidly than the standard `Column Generation` (compare Columns 3–4 to Columns 7–8). This is mostly due to the fact that `Projective Cutting-Planes` can generate high-quality lower bounds from the very first iterations, as it will also be shown by the running profiles from Figure 6, Section 3.2.1 next.

Regarding the complete convergence, the standard `Column Generation` requires in average 93% more CPU time and 44% more iterations than the `Projective Cutting-Planes`. For the last three (most difficult) instances, the `Projective Cutting-Planes` reduced the number of iterations by half. By applying stabilization techniques on the classical `Column Generation`, the reduction of the number of iterations would generally be between 10% and 20%, for all instances except the (very easy) `m20` and `m35` [16, Table 2].

3.2.1. An aggressive `Projective Cutting-Planes`. Let us now consider an aggressive `Projective Cutting-Planes`, defining $\mathbf{x}_{\mathtt{it}} = \mathbf{x}_{\mathtt{it}-1} + t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$, *i.e.*, $\mathbf{x}_{\mathtt{it}}$ is the best feasible solution discovered up to the current iteration (the last pierce point).

Figure 6 presents the evolution along the iterations of the lower bounds of the aggressive and standard `Projective Cutting-Planes` compared to the Lagrangean bounds of the standard Column Generation. This figure demonstrates that the aggressive `Projective Cutting-Planes` starts very well by strictly increasing the lower bound at each iteration (no "yo-yo" effect); however, the full convergence of this aggressive algorithm needs significantly more CPU time than the standard `Projective Cutting-Planes`. Since it does not use truncated interior points $\mathbf{x}_{\mathtt{it}}$, the iterations of the aggressive algorithm are slower. As such, even for the first instance `m01-1` where the aggressive version needs less iterations, the total convergence time is approximately three times larger than that of the standard `Projective Cutting-Planes`. For `vb50c1-1`, the aggressive algorithm needs 9 times more (CPU) time.

3.2.2. Results over Multiple Runs. Although we have only presented individual results on individual instances until now, we can demonstrate that similar trends show up across multiple runs. Table 6 presents results over 10 runs, reporting the average, the standard deviation and the minimum/maximum number of iterations required for full convergence by both the new and the standard method. To randomize the two methods, we determine each optimal solution $\mathtt{opt}(\mathscr{P}_{\mathtt{it}})$ by randomly breaking ties in case of equality (at each iteration `it`). The maximum number of iterations needed by the `Projective Cutting-Planes` is usually lower than the minimum number of iterations of the standard `Column Generation`, and so, there is no need for statistical tests to confirm this difference is statistically significant. In addition, all standard deviations are usually rather limited for both methods, generally representing less than 5% of the average value. Other preliminary experiments confirm that similar trends show up across all instances from each instance set.

3.3. The oscillations of the inner solutions and the "bang-bang" effects. The goal of this section is to (try to) gain more insight into why an "aggressive" definition of $\mathbf{x}_{\mathtt{it}}$ like $\mathbf{x}_{\mathtt{it}} = \mathbf{x}_{\mathtt{it}-1} + t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$ leads to poor results in the long run on certain problems and to reasonable results on others. A possible explanation is related to the oscillations of the inner solutions $\mathbf{x}_{\mathtt{it}}$ along the iterations `it`. The above aggressive $\mathbf{x}_{\mathtt{it}}$ definition generates stronger oscillations ("bang-bang" effects) for the robust optimization (Section 3.1) and the Benders decomposition ([15, § 4.1]) problems than for graph coloring ([15, §. 4.2]) or *Multiple-Length Cutting-Stock* (Section 3.2).

We provide below the values of the first 15 components of $\mathbf{x}_{\mathtt{it}+1} = \mathbf{x}_{\mathtt{it}} + t^*_{\mathtt{it}}\mathbf{d}_{\mathtt{it}}$ for

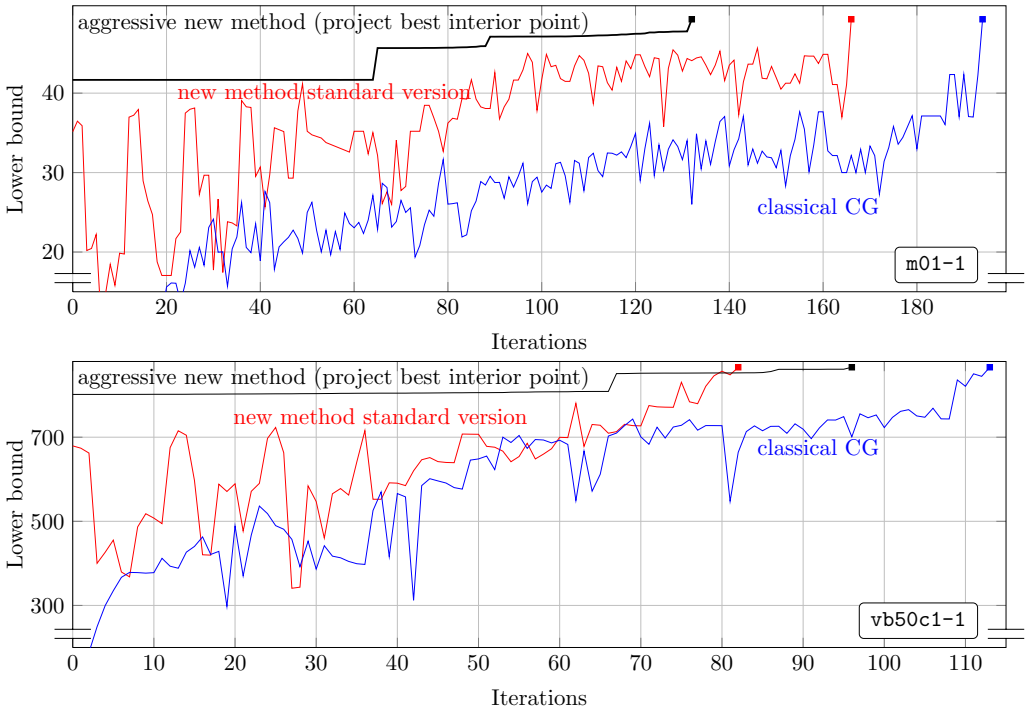

Figure 6: Two representative examples of running profiles, comparing the aggressive and the standard `Projective Cutting-Planes` against `Column Generation`. While the aggressive `Projective Cutting-Planes` starts very well (the black curves show no "yo-yo" effect), it converges rather slowly in terms of CPU time.

Instance	OPT	`Projective Cutting-Planes`		standard `Column Generation`	
		avg (std. dev)	min/max	avg (std. dev)	min/max
`m01-1`	49.3	159 (4.7)	152/168	191 (8.2)	173/202
`m20-1`	56.6	98.8 (3.2)	91/102	162 (6.9)	152/175
`m35-1`	73.9	63.3 (3.2)	61/69	66.3 (2)	64/69
`vb50c1-1`	866.3	82 (0)	82/82	113 (0)	113/113
`vb50c2-1`	672.3	114 (0)	114/114	127 (0)	127/127
`vb50c3-1`	281.949	173 (18.2)	119/180	196 (7.2)	174/198
`vb50c4-1`	579.548	115 (0)	115/115	158 (0)	158/158
`vb50c5-1`	337.675	201 (22.8)	133/209	238 (3.3)	228/239
`wäscher-1`	24.0648	308 (13)	287/328	485 (7.5)	466/494
`hard-sch-1`	51.4253	356 (7.7)	346/370	707 (11.4)	691/726

Table 6: Statistical comparison (over ten runs) of the number of iterations needed by the two methods to fully converge on the first instance from each instance set.

$\texttt{it} \in \{1, 11, 21, 31, 41\}$, *i.e.*, as generated by `Projective Cutting-Planes` using the above aggressive $\mathbf{x}_{\texttt{it}}$ definition. For each problem, we selected the very first instance from the main table of results, *i.e.*, from Table 4, from the second group of rows of [15, Table 1], from [15, Table 3], and then from Table 5. It is clear that these values exhibit stronger oscillations for the first two problems than for the last two. This explains why setting $\mathbf{x}_{\texttt{it}} = \mathbf{x}_{\texttt{it}-1} + t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$ is appropriate for graph coloring, while the best settings for the first two problems take a form $\mathbf{x}_{\texttt{it}} = \mathbf{x}_{\texttt{it}-1} + \alpha t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$

with $\alpha < 0.5$. Regarding *Multiple-Length Cutting-Stock*, although we could not obtain the best results using $\mathbf{x}_{\mathtt{it}} = \mathbf{x}_{\mathtt{it}-1} + t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$ in Section 3.2, this choice would still leads to reasonable results in Table 6 (where the number of iterations is reasonably small, even if the CPU time is too large).

```
The robust optimization problem:
0     37.36 0     59.62 0     69.77 0     97    199.2 0     0     417   4403  0     65.66
20.76 22.81 0     49.76 0     45.46 0     65.86 236.4 0     136.3 254.6 3500  0     64.43
27.38 18.04 0     46.28 0     37.49 0     55.68 248.7 0     180.8 201.3 3205  0     64.03
33.26 13.8  0     43.21 0     30.41 0     46.66 259.6 0     220.7 154.1 2942  0     63.67
36.22 11.68 0     41.63 0     26.86 0     42.14 265.1 0     240.7 130.3 2811  0     63.49
The benders reformulation (IP version):
2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76  2.76
0.37  0.37  0.37  0.37  0.37  0.37  4.08  0.37  0.37  0.37  0.37  0.37  0.37  0.37  0.37
0.112 0.112 0.112 0.112 0.112 0.112 1.93  0.112 0.112 1.64  0.112 0.112 0.112 0.112 0.112
0.026 0.026 0.026 0.026 0.026 0.026 1.62  0.026 0.026 1.62  0.026 0.026 0.026 0.026 0.026
0.018 0.024 0.018 0.018 0.018 0.029 1.67  0.03  0.018 1.12  0.018 0.079 0.018 0.029 0.018
Standard graph coloring:
0.025 0.021 0.036 0.021 0.033 0.021 0.021 0.021 0.029 0.029 0.021 0.025 0.029 0.025 0.033
0.04  0.064 0.033 0.028 0.084 0.028 0.035 0.019 0.04  0.059 0.019 0.073 0.054 0.025 0.05
0.04  0.063 0.033 0.029 0.085 0.028 0.044 0.019 0.04  0.058 0.019 0.072 0.056 0.025 0.051
0.038 0.062 0.032 0.03  0.089 0.027 0.045 0.018 0.039 0.056 0.018 0.07  0.054 0.025 0.051
0.037 0.06  0.032 0.031 0.088 0.026 0.044 0.018 0.038 0.055 0.018 0.068 0.053 0.025 0.051
Multiple length cutting stock:
0.28  0.43  0.72  0.79  0.23  0.7   0.55  0.39  0.69  0.01  0.41  0.4   0.05  0.25  0.95
0.27  0.43  0.72  0.79  0.24  0.7   0.55  0.39  0.69  0.01  0.41  0.4   0.05  0.24  0.95
0.28  0.43  0.72  0.79  0.23  0.7   0.55  0.39  0.69  0.01  0.41  0.4   0.05  0.24  0.95
0.28  0.44  0.72  0.79  0.23  0.7   0.55  0.4   0.69  0.01  0.41  0.41  0.05  0.24  0.95
0.28  0.43  0.72  0.79  0.23  0.7   0.55  0.39  0.69  0.01  0.42  0.39  0.05  0.24  0.95
```

4. Conclusions. This paper has expanded the research work on `Projective Cutting-Planes` first started in [15]. We presented two new problems on which we illustrated new techniques for solving the projection sub-problem (almost) as rapidly as the separation sub-problem. The new numerical experiments confirm all conclusion from [15]. We recall that the main added value is that `Projective Cutting-Planes` has a built-in mechanism to generate feasible (inner) solutions along the iterations, *i.e.*, we obtain a sequence of inner solutions that converge towards $\mathtt{opt}(\mathscr{P})$, somehow similarly to the solutions of the central path in interior point algorithms. There is no such built-in mechanism in `Cutting-Planes`: even if some ad-hoc methods could be sometimes used in `Cutting-Planes` to generated feasible solutions along the iterations, these inner solutions represent a by-product of the algorithm with no influence on the `Cutting-Planes` evolution.

`Projective Cutting-Planes` can offer certain advantages *beyond* the reduction of the computing effort needed to fully converge. For instance, the robust linear programming experiments from Table 4 (Columns "gap 1%") demonstrate that `Projective Cutting-Planes` can produce a feasible solution with a provable optimality gap below 1% using in certain cases less than 5% of the total convergence time. Furthermore, `Projective Cutting-Planes` can easily avoid the degeneracy problems of standard `Cutting-Planes`. The separation sub-problem of the standard `Cutting-Planes` determines each new constraint only guided by the current optimal (outer) solution. The `Projective Cutting-Planes` generates new constraints by taking into account *a pair* of inner–outer solutions. This, coupled with the fact that all considered pro-

jection sub-problems $\texttt{project}(\mathbf{x}_{\texttt{it}} \to \mathbf{d}_{\texttt{it}})$ satisfy $\mathbf{b}^\top \mathbf{d}_{\texttt{it}} > 0$, enables the `Projective Cutting-Planes` to more easily avoid many degeneracy issues (*i.e.*, iterations that keep the objective value constant) that can arise in `Cutting-Planes`. Although in our experiments these issues of the standard `Cutting-Planes` are only visible in Section 3.1 (Remark 10, p. 11), it is well-known that they do arise quite frequently in `Column Generation` as well.[26]

The numerical tests on *Multiple-Length Cutting-Stock* confirm that an aggressive `Projective Cutting-Planes` that chooses $\mathbf{x}_{\texttt{it}}$ as the best solution ever discovered (*i.e.*, $\mathbf{x}_{\texttt{it}} = \mathbf{x}_{\texttt{it}-1} + t^*_{\texttt{it}-1}\mathbf{d}_{\texttt{it}-1}$) eliminates the infamous "yo-yo" effect that appears very often (if not always) in `Column Generation`. However, this `Projective Cutting-Planes` variant is not particularly effective in the long run (see Figure 6, p. 15). We prefer a more conservative variant that chooses $\mathbf{x}_{\texttt{it}}$ as indicated in Section 2.2.2. Compared to this `Projective Cutting-Planes` variant, the standard `Column Generation` requires in average 93% more CPU time and 44% more iterations in Table 5. By applying stabilization techniques on the classical `Column Generation`, the reduction of the number of iterations would generally be between below 20%, for all instances except the (very easy) sets `m20` and `m35` [16, Table 2].

We hope this paper, which continues the initial research from [15], sheds useful light on solving other LPs with prohibitively-many constraints.

REFERENCES

[1] H. Ben Amor and J. M. V. de Carvalho, *Cutting stock problems*, in Column Generation, G. Desaulniers, J. Desrosiers, and M. M. Solomon, eds., vol. 5, Springer, 2005, pp. 131–161.

[2] J. F. Benders, *Partitioning procedures for solving mixed-variables programming problems*, Numerische mathematik, 4 (1962), pp. 238–252.

[3] A. Charnes and W. W. Cooper, *Programming with linear fractional functionals*, Naval research logistics quarterly, 9 (1962), pp. 181–186.

[4] F. Clautiaux, C. Alves, and J. M. V. de Carvalho, *A survey of dual-feasible and superadditive functions*, Annals of Operations Research, 179 (2009), pp. 317–342.

[5] A. M. Costa, *A survey on benders decomposition applied to fixed-charge network design problems*, Computers & operations research, 32 (2005), pp. 1429–1450.

[6] M. Fischetti and M. Monaci, *Cutting plane versus compact formulations for uncertain (integer) linear programs*, Mathematical Programming Computation, 4 (2012), pp. 239–273.

[7] J. Gondzio, *Interior point methods 25 years later*, European Journal of Operational Research, 218 (2012), pp. 587–601.

[8] J. Gondzio, P. González-Brevis, and P. Munari, *Large-scale optimization with the primal-dual column generation method*, Math. Prog. Comp., 8 (2016), pp. 47–82.

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

[10] A. N. Letchford, F. Rossi, and S. Smriglio, *The stable set problem: Clique and nodal inequalities revisited*, European Journal of Operational Research, in major revision (2018).

[11] M. E. Lübbecke and J. Desrosiers, *Selected topics in column generation*, Operations Research, 53 (2005), pp. 1007–1023.

[12] E. Malaguti, M. Monaci, and P. Toth, *An exact approach for the vertex coloring problem*, Discrete Optimization, 8 (2011), pp. 174–190.

[13] D. Porumbel, *Ray projection for optimizing polytopes with prohibitively many constraints in set-covering column generation*, *Mathematical Programming*, 155 (2016), pp. 147–197.

[14] D. Porumbel, *From the separation to the intersection subproblem for optimizing polytopes with prohibitively many constraints in a Benders decomposition context*, Discrete Optimization, 29 (2018), pp. 148–173.

[26] As [1, §4] put it, "Column generation processes are known to have a slow convergence and degeneracy problems". Section 4.2.2 of [11] explains that "large instances are difficult to solve due to massive degeneracy" — see also the references from *loc. cit* for longer explanations of the mechanisms that lead to degeneracy issues.

[15] D. Porumbel, `Projective Cutting-Planes`, unpublished notes, (2019).
[16] D. Porumbel and F. Clautiaux, *Constraint aggregation in column generation models for resource-constrained covering problems*, INFORMS JoC, 29 (2017), pp. 170–184.
[17] F. Vanderbeck, *Computational study of a column generation algorithm for bin packing and cutting stock problems*, Mathematical Programming, 86 (1999), pp. 565–594.

APPENDICES

Appendix A. Numerical aspects on three projection algorithms. For the sake of completeness, we will now describe all implementation details of the projection algorithms studied throughout this paper and the initial study [15].

A.1. A fast data structure to manipulate a Pareto frontier. We here present the data structure used by the projection algorithm for *Multiple-Length Cutting-Stock* to record `Dynamic Programming` (DP) states. This data structure is not essentially linked to *Cutting-Stock*: it can manipulate any Pareto frontier with two objectives. In Section 2.2.3.2 (Remark 9, p. 9) we described how the DP projection algorithm needs to handle a list of states I whose cost and profits c_i/p_i $\forall i \in I$ satisfy the Pareto dominance relation (2.2.3.a)–(2.2.3.b), recalled below for the reader's convenience.

$$c_1 < c_2 < c_3 \cdots < c_{|I|}$$

$$p_1 < p_2 < p_3 \cdots < p_{|I|}$$

As described in Remark 9, one computationally expensive task (of Algorithm 1, p. 8) is the insertion of a new pair c^+/p^+ at Step 5; it can be very inefficient to scan the whole list I only to check if c^+/p^+ is dominated or not. As such, we propose to record I using a *self-balancing binary tree*, which is a data structure designed to manipulate ordered lists, *e.g.*, it performs a lookup, an insertion and a removal in logarithmic time with respect to $|I|$. The order of states in the tree is given by the simple comparison of costs, *i.e.*, if $c_i < c_j$, then c_i/p_i is ordered before c_j/p_j.

Any self-balancing binary tree implementation has to be able to compare c^+ to the pair c^*/p^* with the highest cost no larger than c^+, *i.e.*, $c^* = \max\{c_i :\ c_i \leq c^+,\ i \in I\}$. Without comparing to c^*/p^*, it is certainly impossible to decide whether c^+/p^+ should be inserted before or after c^*/p^* in the binary tree. Thus, any implementation of the self-balancing tree has to provide a means to determine c^*/p^*. In the worst case, it is certainly possible to (temporary) insert c^+/p^+ in the tree and return the element before c^+/p^+; this operation relies on the insertion operator and other constant-time manipulations, and so, it takes logarithmic time.

Once c^*/p^* is determined, we apply an insertion routine as follows. First, if $p^* \geq p^+$, then the new pair c^+/p^+ is directly rejected because it is dominated by definition. Otherwise, if $p^* < p^+$, then c^+/p^+ has to be inserted in the tree, and so, other recorded pairs may become dominated and need to be removed. For instance, if $c^* = c^+$ and $p^* < p^+$, then c^*/p^* is immediately removed from the tree. Furthermore, our insertion routine enumerates one by one all next recorded pairs $c^\#/p^\#$ ordered after c^*/p^* (and after c^+/p^+) that satisfy $p^\# \leq p^+$ and removes them all. Indeed, such pairs $c^\#/p^\#$ are certainly dominated by c^+/p^+, given that $p^\# \leq p^+$ and $c^\# > c^+$; the latter inequality follows from the fact that $c^\#/p^\#$ is ordered after c^*/p^* in the tree.

A.2. Numerical difficulties when solving the Benders integer model. For both the separation and the projection sub-problem, the `Cutting-Planes` algorithm for the Benders reformulation (3.2.8a)–(3.2.8c) from the main paper [15] can

encounter a number of numerical issues that are worthwhile investigating. The main one in Appendix A.2.1 regards the optimization of the relaxed master programs, to determine $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ at each iteration $\texttt{it}$. The second one in Appendix A.2.2 concerns the projection algorithm.

A.2.1. Numerical difficulties when solving the integer master problem. The ILP solver for determining $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ at each iteration $\texttt{it}$ can be particularly prone to numerical or precision problems, especially if $\mathscr{P}_{\texttt{it}}$ contains too many constraints (3.2.8b) with exceedingly large coefficients – as determined by the sub-problem algorithm. For the (`cplex`) LP solver, many values (of variables or slacks) can be zero in theory and slightly larger than zero in practice; multiplying such "ϵ-sized" values with extremely large coefficients can generate noising terms and numerical precision problems.

Recall that the separation sub-problem performs a normalization of these coefficients by imposing $\mathbf{1}^\top \mathbf{u} = 1$ in (3.2.4). Regarding the projection sub-problem, we mentioned at point (ii) from Section 3.2.3 that the returned $\mathbf{u}$ does not need to be normalized. This is perfectly fine in theory, but if the optimal solution of the LP (3.2.11a)–(3.2.11e) used by the sub-problem has some exorbitant coefficients (see reasons in Appendix A.2.2 below), the projection algorithm can return a constraint (3.2.8b) with exceedingly large coefficients. To avoid such drawbacks, we apply the following principles when solving the projection sub-problem:

- If (3.2.11a)–(3.2.11e) has multiple optimal solutions, it is better to take one with reasonable coefficients; as such, when solving (3.2.11a)–(3.2.11e), the projection algorithm breaks ties by minimizing $\mathbf{1}^\top \bar{\mathbf{u}}$.
- Before inserting a constraint (3.2.8b) into the master problem, it is better to normalize it; for this, we multiply $\mathbf{u}$ by a scalar such that the largest coefficient u_{ij} of a term $u_{ij}x_{ij}$ in (3.2.8b) becomes equal to 10.

Despite above efforts, the master ILP solver (for both the standard or the new `Cutting-Planes`) might take too long to optimize certain relaxed master ILPs associated to (3.2.8a)–(3.2.8c), *i.e.*, it can be too difficult to determine $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ at certain (rare) iterations $\texttt{it}$. Although such problems are not frequent, they could completely block the overall algorithm for a prohibitively long time; accordingly, the *integer* Benders model can become very difficult to solve in such cases, almost impossible.

The key for overcoming this drawback comes from the fact that it is not really essential to determine the optimal solution $\texttt{opt}(\mathscr{P}_{\texttt{it}})$ at each iteration $\texttt{it}$ (as described soon). Accordingly, we enforce a limit of $\frac{n}{120} + 1$ seconds on the running time of the ILP solver; as soon as this limit is exceeded, the `Cutting-Planes` continues with the best sub-optimal solution of $\mathscr{P}_{\texttt{it}}$ found by the ILP solver in the given time, which is different from $\texttt{opt}(\mathscr{P}_{\texttt{it}})$. This does not change the correctness of the overall algorithm, because this sub-optimal solution could be separated anyway at the next call to the sub-problem algorithm. If this is not the case, we allow 500 more time to the ILP solver and we let it try again to determine $\texttt{opt}(\mathscr{P}_{\texttt{it}})$. If this second try fails, we consider the instance can not be solved. This technique is described in greater detail in Appendix C.2 of [14].

A.2.2. Numerical aspects when solving the projection sub-problem. The projection algorithm could be prone to numerical problems when α is close to 1. More exactly, a small precision error in computing the step length t^* can lead to an infeasible pierce point $\mathbf{x} + t^*\mathbf{d}$, and, if α is very close to 1, this could lead to choosing an infeasible inner solution $\mathbf{x} + \alpha \cdot t^*\mathbf{d}$ at the next iteration. For the standard separation, a small precision error can be less problematic if the returned constraint

does separate the current optimal outer solution.

In particular, the LP (3.2.11a)–(3.2.11e) used to solve the projection subproblem in the main paper [15] can be particularly prone to numerical problems, because the decision variables $\overline{\mathbf{u}}$ can be unbounded. In fact, the only constraint that can limit the magnitude of $\overline{\mathbf{u}}$ is $-\sum_{\{i,j\}\in E} b_{\text{wd}} d_{ij} \overline{u}_{ij} = 1$ from (3.2.11d); but since the terms d_{ij} can be positive, negative or zero, it is possible to satisfy this constraint by assigning some extremely high values to certain $\overline{u}_{ij}$ variables. Furthermore, certain factors $b_{\text{wd}} x_{ij}$ in the sum $\sum_{\{i,j\}\in E} b_{\text{wd}} x_{ij} \overline{u}_{ij}$ from the objective function (3.2.11a) can be zero in theory and slightly different from zero in practice (at least when using `Cplex`); multiplying such non-zero factors $b_{\text{wd}} x_{ij}$ with an extremely large $\overline{u}_{ij}$ can lead to non-zero artificial (noising) terms in the objective function.

To reduce such phenomena, we impose a limit of 100 on the maximum value the variables $\overline{\mathbf{u}}$ can take in (3.2.11a)–(3.2.11e). In fact, any practical algorithm for (3.2.11a)–(3.2.11e) has to impose such a limit in practice because the memory is finite and the variables $\overline{\mathbf{u}}$ can not be really unbounded. This leads to restricting the feasible set of (3.2.11a)–(3.2.11e); in theory, the resulting restricted LP might not minimize t^* to the full, and so, it might return an overestimated t^* so that $\mathbf{x} + t^*\mathbf{d}$ could be potentially infeasible. However, one can certify that the projection sub-problem is correctly solved by checking that the optimal solution satisfies $\overline{u}_{ij} < 100 \; \forall \{i,j\} \in E$. This is very often confirmed and the optimal $\overline{\mathbf{u}}$ hardly ever contains values above 0.5 in practice. When there is however some $\overline{u}_{ij} = 100$, this is most certainly due to the numerical issues described above; for such rare cases, we could check the feasibility of $\mathbf{x} + t^*\mathbf{d}$ using the separation sub-problem. In practice, `Projective Cutting-Planes` never reported a final solution that proved to be infeasible; the fact that some of the intermediate solutions $\mathbf{x} + t^*\mathbf{d}$ might actually be (slightly) infeasible does not change the correctness of the final solution reported in the end.

A.3. Projection with Reinforced Relaxed Stables for Graph Coloring.

A.3.1. Detailed Definition of the reinforced relaxed (RR-) stables. We recall from the initial work [15, Section 3.3.4] that, besides standard graph coloring, we also considered a second model in which each constraint $\mathbf{a}^\top \mathbf{x} \leq 1$ of the dual `Column Generation` model is defined by an RR–stable $\mathbf{a} \in \mathcal{P}$, *i.e.*, by an (extreme) solution $\mathbf{a}$ of the auxiliary polytope $\mathcal{P}$ from [15, Definition 3.1]. This auxiliary polytope $\mathcal{P}$ is an outer approximation of the stable set polytope. It is constructed by reinforcing with cuts the description of the *relaxed stables* (*i.e.*, vectors respecting the edge inequalities), hence the name reinforced relaxed stables (RR–stables).

Recalling [15, (3.3.4)], $\mathcal{P}$ is defined by six classes of (reinforcing) cuts of the form $\mathbf{e}^\top \mathbf{a} \leq 1 \; \forall (\mathbf{e}, 1) \in \mathcal{R}$ or $\mathbf{f}^\top \mathbf{a} \leq 0 \; \forall (\mathbf{f}, 0) \in \mathcal{R}$. We now present these six classes of cuts, *without* applying the Charnes–Cooper transformation on them. In fact, the cuts (a)–(d) are statically added when calling the first intersection sub-problem (and they are re-used for all the next sub-problems), while the cuts (e)–(f) are dynamically added one by one using the `cut generation` algorithm from Section 3.3.4.

(a) The first cut class simply comes from the edge inequalities defining the standard 0–1 stables, *i.e.*, at this stage, we only impose $a_u + a_v \leq 1 \; \forall \{u,v\} \in E$, obtaining the description of the relaxed stables, *i.e.*, the fractional stable polytope.

(b) Generalizing the above idea, we use cuts (b) to generate a number of clique inequalities of the form $\mathbf{a}(\mathscr{C}) = \sum_{v\in\mathscr{C}} a_v \leq 1$, using only cliques $\mathscr{C}$ of maximum size $\min(5, k)$. These cliques are enumerated one by one by backtracking; the role of the parameter 5 is to keep the number of such cliques within reasonable limits. To avoid combinatorial explosions, the backtracking algorithm uses the

rule that any vertex has to be ignored after appearing in 20 inequalities, *i.e.*, after 20 apparitions, the vertex is discarded from generating future cliques. We use the most standard backtracking algorithm to enumerate all cliques of maximum size $\min(5, k)$, except that we discard any vertex after it appears in 20 cliques.

(c) To generate cuts (c), we actually construct a collection of clique inequalities that "cover" V. They are generated by iterating over the vertices $V = [1..n]$, using a method that is reminiscent of Algorithm 1 from [10] or of [12, § 2.2.2]. At the first iteration $i = 1$, this method simply selects a clique $\mathscr{C}$ of a given maximum size k' (see below) that contains the vertex $i = 1$ and imposes the clique inequality $\mathbf{a}(\mathscr{C}) \leq 1$. We now introduce a set $V' = V \setminus \mathscr{C}$ that will evolve along the iterations; all subsequent cliques will be determined by maximizing the number of elements from V'. At the second iteration, we move to the next element $i \in V'$ to determine a new clique $\mathscr{C} \ni \{i\}$ of maximum size k' and impose $\mathbf{a}(\mathscr{C}) \leq 1$. After performing $V' \leftarrow V' \setminus \mathscr{C}$, we move to the next iteration and repeat. At each iteration, we search for a clique $\mathscr{C}$ of bounded size k' with a maximum of elements from V', *i.e.*, we apply the `Branch & Bound with Bounded Size` (BBBS) from Appendix A.3.3 with very large weights for all $v' \in V'$ and with small weights to all $v \in V \setminus V'$. The value of k' is given by the minimum clique size for which this BBBS algorithm can solve the standard maximum clique of bounded size (with weights $\mathbf{1}_n$) on G within *at most 0.01 seconds*. Experiments suggest that such cuts (c) can even accelerate the `cut generation` by a factor of ten on the Leighton graphs (`le450_25c`, `le450_25c`, etc.), especially when k' is much larger than the value of k used at point (f).

(d) A cut of this class is associated to any $u, v, w \in V$ such that $\{u, v\} \in E$, $\{u, w\} \notin E$ and $\{v, w\} \notin E$. Using notation $N_v = \{v' \in V : \{v, v'\} \in E\}$, a maximum standard 0–1 stable $\mathbf{a}^{\text{std}}$ satisfies the following:

$$a^{\text{std}}_u + a^{\text{std}}_v \leq a^{\text{std}}_w + \mathbf{a}^{\text{std}}(N_w - N_u \cap N_v),$$

because if the maximum stable $\mathbf{a}^{\text{std}}$ contains u or v (exclusively), then it also has to contain either w *or* a neighbor of w. This neighbor of w can only belong to $N_w - N_u \cap N_v$ because it can not be connected to both u and v (since one of u or v belongs to the stable). This idea has also been generalized to the case of triangles $\{\mu, u, v\} \subset V$ not connected to a vertex $w \in V$. We obtain the following cuts:

$$a_u + a_v \leq a_w + \mathbf{a}(N_w - N_u \cap N_v) \qquad \forall \{u, v\} \in E, \{u, w\} \notin E, \{v, w\} \notin E$$

$$a_\mu + a_u + a_v \leq a_w + \mathbf{a}(N_w - N_\mu \cap N_u \cap N_v) \quad \forall \{\mu, u\}, \{\mu, v\}, \{u, v\} \in E, \{\mu, w\} \notin E, \{u, w\} \notin E, \{v, w\} \notin E$$

However, we insert such cuts only when they have less than 10 non-zero coefficients, because we noticed in practice that they are the most effective when they have 3 or 4 non-zero coefficients. For instance, when $N_w - N_u \cap N_v = \emptyset$, the first cut simplifies to $a_u + a_v \leq a_w$. Such a cut would eliminate $[1/\omega \;\; 1/\omega \;\; 1/\omega \ldots 1/\omega]$ from $\mathcal{P}$, where ω is the maximum clique size of G. This enabled us in [15, Remark 5] to show that the optimum of the proposed `Column Generation` model with RR stables can be larger than ω.

(e) These cuts are classical odd-cycle (or odd-hole) inequalities that can be separated in polynomial time. First, notice than a (simple) odd cycle H yields a cut $\sum_{v \in H} a_h \leq \frac{|H|-1}{2}$ because a stable with $\frac{|H|+1}{2}$ vertices of H would select two consecutive vertices of the cycle. To separate such a cut, it is enough to re-write it in the form $1 \leq \sum_{v \in H} (1 - 2a_v)$, equivalent to $1 \leq \sum_{\{u,v\} \in EC(H)} (1 - a_u - a_v)$,

where $EC(H)$ represents the $|H|$ edges of the cycle inside H. The separation sub-problem can be solved by finding the shortest *odd cycle* in G considering edge weights $1 - a_u - a_v \; \forall \{u, v\} \in E$ — these weights are always non-negative because of above cuts (a). This shortest odd cycle can be found by applying Dijkstra's algorithm on an augmented graph with: (i) a source linked to all vertices V, (ii) all vertices V without any edges between them, (iii) a set V' of copies of V linked to V via edges $\{u, v'\} \in V \times V'$ of weight $1 - a_u - a_v$ for any $\{u, v\} \in E$ (*i.e.*, v' is a copy of v), and (iv) a target vertex linked to all vertices V'.

(f) The last cut class consists of k-clique inequalities $\mathbf{a}(\mathscr{C}) \leq 1$ associated to cliques $\mathscr{C}$ with at maximum k elements, where k is a parameter that defines the model — it always has to be indicated in the numerical results as in (Column 9 of) [15, Table 3]. Separating these cuts reduces to solving a maximum weight clique problem with bounded size k; the weights $\mathbf{a}$ are given by the optimal solution at the current `cut generation` iteration. For large values of k, (the iterative call to) this problem can become the main computational bottleneck of the overall `Cutting-Planes`. This is why we present in Appendix A.3.3 a specific `Branch & Bound with Bounded Size` (BBBS) algorithm dedicated to this clique problem with bounded size.

A.3.2. A `cut generation` algorithm with Reinforced Relaxed Stables. For the reader's convenience, let us recall the Charness-Cooper LP (re-)formulation (3.3.6a)–(3.3.6d) solved by the projection algorithm.

$$t^* = \min \; \overline{\alpha} - \mathbf{x}^\top \overline{\mathbf{a}} \tag{A.3.1a}$$

$$\mathbf{e}^\top \overline{\mathbf{a}} \leq \overline{\alpha}, \; \mathbf{f}^\top \overline{\mathbf{a}} \leq 0 \quad \forall (\mathbf{e}, 1) \in \mathcal{R}, \; \forall (\mathbf{f}, 0) \in \mathcal{R} \tag{A.3.1b}$$

$$\mathbf{d}^\top \overline{\mathbf{a}} = 1 \tag{A.3.1c}$$

$$\overline{\mathbf{a}} \geq \mathbf{0}_n, \; \overline{\alpha} \geq 0 \tag{A.3.1d}$$

The above LP (A.3.1a)–(A.3.1d) is solved by `cut generation` because enumerating all reinforcing cuts $\mathcal{R}$ is computationally very exhausting, if not impossible. Notice that these reinforcing cuts $\mathcal{R}$ are slightly modified when they are inserted in the above LP (re-)formulation, *i.e.*, we use $\mathbf{e}^\top \overline{\mathbf{a}} \leq \overline{\alpha}$ in (A.3.1b) instead of $\mathbf{e}^\top \mathbf{a} \leq 1$ as described in Appendix A.3.1. However, the difficulty of the separation sub-problem for (A.3.1b) does not depend on the right-hand side $\overline{\alpha}$, but on the structure of the cuts $\mathcal{R}$. To make the overall `Projective Cutting-Planes` reach its full potential, it is important to have a fast separation algorithm.

A positive distinguishing characteristic of the above LP reformulation (A.3.1a)–(A.3.1d) is that the prohibitively-many reinforcing cuts $\mathcal{R}$ from (A.3.1b) do not depend on $\mathbf{x}$ or $\mathbf{d}$; these constraints remain the same along all iterations of the overall `Projective Cutting-Planes`. As such, after solving the projection sub-problem $\texttt{project}(\mathbf{x}_{\texttt{it}} \rightarrow \mathbf{d}_{\texttt{it}})$ at some iteration `it` of the overall `Projective Cutting-Planes`, one can keep all generated cuts (A.3.1b) and only update (A.3.1c) to move to the next iteration $\texttt{it} + 1$. The first four cut classes (a)–(d) from $\mathcal{R}$ are actually static and they are inserted in (A.3.1a)–(A.3.1d) at the very first iteration of the overall `Cutting-Planes`. Only the cuts (e)–(f) are dynamically generated one by one, by repeatedly solving a separation sub-problem.

This sub-problem asks to find maximum between $\max\limits_{(\mathbf{e},1) \in \mathcal{R}} \mathbf{e}^\top \overline{\mathbf{a}} - \overline{\alpha}$ and $\max\limits_{(\mathbf{f},0) \in \mathcal{R}} \mathbf{f}^\top \overline{\mathbf{a}}$, for the current optimal solution $(\overline{\mathbf{a}}, \overline{\alpha})$ at the current `cut generation` iteration. Since the cuts (e) can be separated in polynomial time by applying Dijkstra's algorithm on

a bipartite graph with $2n+2$ vertices (Appendix A.3.1), the most computationally-critical step is the separation of the k-clique inequalities (f).

A.3.2.1. Controlling a trade-off between speed and efficiency using k-clique inequalities. The k-clique cuts $\sum_{v\in\mathscr{C}} a_v \leq 1$ take the form $\sum_{v\in\mathscr{C}} \overline{a}_v \leq \overline{\alpha}$ in (A.3.1b), where $\mathscr{C}$ is a clique with k vertices. The separation of these cuts requires solving a maximum weight clique problem with bounded size k, which is NP–Hard when k is not a constant parameter. We propose in Appendix A.3.3 a dedicated `Branch & Bound with Bounded Size` (BBBS) algorithm to solve this problem. The repeated call to this algorithm becomes the most important computational bottleneck for the overall `Cutting-Planes`, especially when k is not very small. We also tried to generate the cuts (f) by solving the maximum weight clique problem with no size restriction ($k = \infty$). This is the well-known maximum clique problem for which there exist elaborately-tuned off-the-shelf software (*e.g.*, we used the well-known `Cliquer`, due to S. Niskanen and P. Östergård, see users.aalto.fi/∼pat/cliquer.html), but our BBBS algorithm is faster when k is not too large.

The value of k can be used to control a trade-off between speed and efficiency, between the total computation time of the `Projective Cutting-Planes` and the reported optimal value (of the new (3.3.1) model with RR stables). Experiments confirm that the above maximum weight clique problem with bounded size k can be solved more rapidly (*i.e.*, BBBS becomes faster) when k is lower.[27] On the other hand, by lowering k, the outer approximation $\mathcal{P} \supsetneq \texttt{conv}(\mathcal{P}_{0-1})$ becomes coarser, in the sense that $\mathcal{P}$ contains more artificial RR stables that are not standard stables. This leads to more artificial constraints in the new (3.3.1) model, so that the lower bound reported in the end becomes smaller.

On sparser graphs, the resulting `Projective Cutting-Planes` with RR–stables is naturally faster than the classical `Column Generation` with standard stables. Sparser graphs have smaller cliques and larger stables, so that the maximum weight clique problem with bounded size (for the `Projective Cutting-Planes`) becomes easier and the maximum weight stable problem (for the standard `Column Generation`) becomes harder.

A.3.2.2. Accelerating the Cut Generation using Stabilization. As stated above, most of the computing effort is spent on repeatedly separating the constraints (f) by solving a maximum weight clique problem. We also propose a few stabilisation ideas to accelerate the `cut generation` for (A.3.1a)–(A.3.1d):

1. We use a simple-but-effective solution smoothing technique: instead of calling the separation algorithm on the current optimal solution, we call it on the midpoint between the current optimal solution and the previous optimal solution. If the current optimal solution can not be separated this way, we have to call the separation algorithm again, this second time on the current optimal solution.
2. We propose a (meta-)heuristic algorithm for the sub-problem before solving it exactly. This heuristic executes $5 \cdot n$ iterations of a Tabu Search algorithm.[28]

[27] This was actually observed both for the BBBS algorithm developed in this work and for the `Cplex` ILP solver applied on the same problem. In theory, a very small k can even be seen as a parameter, so that the maximum weight clique problem with bounded size k is no longer NP-Hard (it becomes polynomial by enumerating all such cliques in $O(n^k)$ time).

[28] This Tabu Search algorithm encodes each candidate solution as a bit string of length n with exactly k ones. The objective function is the sum of the weights of the edges induced by the vertices selected by the bit string. Each two vertices $u, v \in V$ are associated to an edge weight, either $\frac{1}{2}(a_u + a_v)$ if $\{u, v\} \in E$ or a prohibitively-small negative weight when $\{u, v\} \notin E$. A Tabu Search iteration selects the best non-Tabu vertex swap, the one maximizing the objective function. We also use incremental data structures to perform a fast streamlined calculation of the

We always start the `cut generation` in a heuristic mode, trying to solve all maximum weight cliques with this heuristic. But once the heuristic fails, the `cut generation` algorithm switches to an exact mode (running the BBBS) for 15 iterations. After each 15 iterations, it tries again to solve the problem heuristically. Unless this repeated heuristic call is successful, the `cut generation` remains in the exact mode for another 15 iterations.

A.3.3. A Branch–and–Bound with Bounded Size (BBBS) for the Maximum Weight Clique Problem. The maximum weight clique with bounded size is a rather general graph–theoretic problem that could be modeled and solved with many different methods. Against our expectations, we did not find any dedicated off-the-shelf software to solve it as rapidly as necessarily. We thus have to introduce a new `Branch & Bound with Bounded Size` (BBBS) algorithm devoted to this problem. This BBBS was mainly used to separate the cuts (f) from Appendix A.3.1, as needed by the `cut generation` algorithm from Appendix A.3.2 above. At the same time, BBBS can directly solve the complementary problem, *i.e.*, the (bounded-size) maximum weight stable. We thus also used BBBS to solve the separation sub-problem of the classical `Column Generation` with standard stables in the results reported in [15, Section 4.2.2].

The main algorithmic engine

The BBBS algorithm relies on a fairly straightforward Branch & Bound (B&B) routine that successively adds vertices to existing cliques, to construct increasingly larger cliques (B&B nodes). A branching tree with such nodes is constructed in a deep–first manner. More exactly, all cliques are constructed (to generate B&B nodes) by adding vertices to existing cliques following an initial order $v_1,\ v_2, \dots v_n$ such that $w_1 \geq w_2 \geq w_3 \geq \cdots \geq w_n$, *i.e.*, the vertices are initially sorted by decreasing weight. As such, the very first B&B node is simply the clique $\{v_1\}$. The second generated B&B node is $\{v_1, v_i\}$ where $i = \min\{i :\ \{v_i, v_1\} \in E\}$ and the third node is $\{v_1,\ v_i,\ v_j\}$ where $j = \min\{j :\ \{v_j, v_1\} \in E,\ \{v_j, v_i\} \in E\}$, assuming $k \geq 3$.

The total number of generated B&B nodes (and the total running time) depends substantially on the quality of the lower and upper bounds used for branch pruning. The lower bound is simply given by the best clique ever generated, *i.e.*, there is a unique global lower bound for the whole branching tree at each moment. In addition, recall (point 2 from Appendix A.3.2.2) that one can first try to solve the bounded–size maximum weight clique problem using a (meta-)heuristic prior to launching BBBS; this can provide a second lower bound. Preliminary experiments suggest that trying other better or faster (meta–)heuristics do not usually lead to an impressive acceleration of the BBBS, and so, we hereafter focus on the upper bound.

The upper bound of each node

The upper bound is determined at each B&B node and it is more critical for reducing the running time. Let us first present the most basic upper bound to be generalized next. Consider the current B&B node corresponding to a constructed clique $\mathscr{C}$ of k' elements with $k' < k$ (otherwise the node is a leaf). The remaining as-yet-unconsidered vertices constitute a list $(u_1,\ u_2,\ u_3, \dots)$ sorted by decreasing weight—this is how

objective function variation associated to each vertex swap. After deselecting a vertex, it becomes Tabu for 10+`random(5)` iterations, where `random(5)` returns a uniformly random integer value in $\{0, 1, 2, 3, 4, 5\}$. For $k = \infty$, we used the multi-neighborhood Tabu search (www.info.univ-angers.fr/~hao/clique.html) due to Q. Wu, J.K. Hao and F. Glover.

the vertices were sorted in the beginning. After eliminating from $(u_1,\ u_2,\ u_3, \dots)$ all vertices that are not connected to all $v \in \mathscr{C}$, one obtains a reduced list $L_{\mathscr{C}}$ of vertices linked to all vertices from $\mathscr{C}$. The simplest upper bound is then given by the sum of the weights of the first $k - k'$ vertices in $L_{\mathscr{C}}$ (plus the weight of $\mathscr{C}$).

We now present a higher–quality upper bound that can greatly accelerate BBBS in practice (by a factor of up to seven). The main idea is to go beyond simply summing up the weights of the first $k - k'$ vertices in $L_{\mathscr{C}}$: it is better to investigate in greater detail which of the vertices in $L_{\mathscr{C}}$ should really contribute to the upper bound value. A vertex u of $L_{\mathscr{C}}$ can *not* contribute to the upper bound value if a preceding $v \in L_{\mathscr{C}}$ (of higher weight) satisfying $\{v, u\} \notin E$ has already contributed to the bound value. We say that v shadows u.

The pseudo-code below implements the above idea to calculate the lower bound: notice how the first **continue** leads to ignoring the current vertex u when some v shadows u. However, this idea can not be applied twice: if there is a second vertex u' such that $\{v, u'\} \notin E$, v can not shadow both u and u' because selecting u and u' can be better than selecting v (assuming $\{u, u'\} \in E$). This explains why the pseudo-code below first inserts u into a list L of vertices that can shadow other vertices (Line 15), but then removes any $v \in L$ at Line 6 if v shadows u, *i.e.*, v can shadow only one vertex at most. However, v could still shadow some $u' \in V$, but only if $\{v, u, u'\}$ is a stable and such cases are detected using a second list L'.

1: $\texttt{ub} \leftarrow \sum_{v \in \mathscr{C}} \texttt{weight}(v)$, $\texttt{addedVtx} = 0$
2: $L \leftarrow \emptyset$ ▷ vertices v that shadow other vertices
3: $L' \leftarrow \emptyset$ ▷ non-edges $\{v, u\} \notin E$ such that v shadows u
4: **for all** $u \in L_{\mathscr{C}}$ **do** ▷ scan $L_{\mathscr{C}}$ by descending order of weight
5: **if** $\exists v \in L$ such that $\{v, u\} \notin E$ **then** ▷ v shadows u
6: $L \leftarrow L \setminus \{v\}$ ▷ v can not shadow more vertices but it can
7: $L' \leftarrow L' \cup \{(v, u)\}$ ▷ shadow some u' if $\{v, u, u'\}$ is a stable at Line 10
8: **continue**
9: **end if**
10: **if** $\exists (v, u') \in L'$ such that $\{v, u, u'\}$ is a stable **then** ▷ v shadows both u and u'
11: $L' \leftarrow L' \setminus \{(v, u')\}$ ▷ v can shadow at maximum two vertices
12: **continue** ▷ u is shadowed by v and u'
13: **end if**
14: $\texttt{ub} \leftarrow \texttt{ub} + \texttt{weight}(u)$, $\texttt{addedVtx} \leftarrow \texttt{addedVtx} + 1$
15: $L \leftarrow L \cup \{u\}$ ▷ u can shadow subsequent vertices in $L_{\mathscr{C}}$
16: **if** $(\texttt{addedVtx} == k - |\mathscr{C}|)$ **then**
17: **break**
18: **end if**
19: **end for**
20: **return** ub

Finally, to make the BBBS reach its full potential, one could still apply a number of further engineering and implementation optimizations (as in many applied algorithms). For instance, experiments suggest that the BBBS can be faster if we limit the size of the lists L and L' to $\min(10, \frac{2}{3}k)$. In addition, we decided not to use the above improved upper bound when k is exceptionally large (greater than half the average degree of G).

Appendix B. The detailed `Column Generation` model and its Lagrangian bounds. The `Column Generation` model optimized throughout this work is

$$y_a: \quad \left.\begin{array}{ll} \max \mathbf{b}^\top \mathbf{x} & \\ \mathbf{a}^\top \mathbf{x} \leq c_a, & \forall (\mathbf{a}, c_a) \in \mathcal{A} \\ \mathbf{x} \geq \mathbf{0}_n & \end{array}\right\} \mathscr{P} \tag{B.1}$$

All proposed algorithms related to `Column Generation` were presented from the standpoint of this LP, both for graph coloring in [15, (3.3.1)] and for *(Multiple-Length) Cutting-Stock* in (2.2.1). This is actually the dual of the master LP below, obtained by relaxing $y_a \in \mathbb{Z}_+$ into $y_a \geq 0$.

$$\mathbf{x}: \quad \begin{array}{lll} \min \sum\limits_{(\mathbf{a}, c_a) \in \mathcal{A}} c_a y_a & & \\ \sum\limits_{(\mathbf{a}, c_a) \in \mathcal{A}} a_i y_a & \geq b_i & \forall i \in [1..n] \\ y_a \geq 0 & & \forall (\mathbf{a}, c_a) \in \mathcal{A} \end{array} \tag{B.2}$$

The (enormous) set $\mathcal{A}$ encode constraints in the above dual LP (B.1) or equivalently columns in the above primal (B.2). These columns represent stables in graph coloring, cutting patterns in *(Multiple-Length) Cutting-Stock*, or, more generally routes in vehicle routing problems, assignments of courses to timeslots in timetabling, or any specific subsets in the most general set-covering problem. For each column $(\mathbf{a}, c_a) \in \mathcal{A}$, $\mathbf{a} \in \mathbb{Z}_+^n$ is generally an incidence vector such that a_i indicates how many times an element $i \in [1..n]$ is covered by $\mathbf{a}$. We use a primal decision variable y_a to encode the number of selections of each column $(\mathbf{a}, c_a) \in \mathcal{A}$. The objective of (B.2) asks to minimize the total cost of the selected columns, under the (set-covering) constraint that each element $i \in [1..n]$ has to be covered at least b_i times.

On several occasions, we referred to the Lagrangian lower bounds of the standard `Column Generation`. When all columns have equal unitary costs (*i.e.*, $c_a = 1\ \forall (\mathbf{a}, c_a) \in \mathcal{A}$ as in graph coloring), we simply used the Farley lower bound

$$\mathscr{L}(\mathbf{x}) = \frac{\mathbf{b}^\top \mathbf{x}}{1 - m_{rdc}(\mathbf{x})}, \tag{B.3}$$

where $m_{rdc}(\mathbf{x})$ is the minimum reduced cost with regards to the optimal (dual) values $\mathbf{x} = \mathtt{opt}(\mathscr{P}_{\mathtt{it}})$ at the current iteration `it`, *i.e.*, $m_{rdc}(\mathbf{x}) = \min\limits_{(\mathbf{a}, c_a) \in \mathcal{A}} c_a - \mathbf{a}^\top \mathbf{x}$.

In *Multiple-Length Cutting-Stock*, the column costs are no longer unitary, but we can still apply the Farley bound (B.3) after normalizing all columns in $\mathcal{A}$. More exactly, we replace $(\mathbf{a}, c_a)$ with $(\frac{\mathbf{a}}{c_a}, 1)$ for each $(\mathbf{a}, c_a) \in \mathcal{A}$ and we obtain a normalized model (B.2) that has the same objective value as the original model because the variables $\mathbf{y}$ are continuous. Let $c_{\min} = \min\{c_a : (\mathbf{a}, c_a) \in \mathcal{A}\}$. The normalized minimum reduced cost $m_{rdc}^{\text{norm}}(\mathbf{x})$ satisfies $m_{rdc}^{\text{norm}}(\mathbf{x}) \geq \frac{1}{c_{\min}} m_{rdc}(\mathbf{x})$ when $m_{rdc}(\mathbf{x}) \leq 0$, because any $(\mathbf{a}, c_a) \in \mathcal{A}$ that achieves $m_{rdc}(\mathbf{x}) = c_a - \mathbf{a}^\top \mathbf{x} \leq 0$ satisfies $\frac{1}{c_a}\left(c_a - \mathbf{a}^\top \mathbf{x}\right) \geq \frac{1}{c_{\min}}\left(c_a - \mathbf{a}^\top \mathbf{x}\right)$. The Farley bound evolves to $\mathscr{L}(\mathbf{x})$ below.

$$\frac{\mathbf{b}^\top \mathbf{x}}{1 - m_{rdc}^{\text{norm}}(\mathbf{x})} \geq \mathscr{L}(\mathbf{x}) = \frac{\mathbf{b}^\top \mathbf{x}}{1 - \frac{1}{c_{\min}} m_{rdc}(\mathbf{x})} \tag{B.4}$$

The above $\mathscr{L}(\mathbf{x})$ is a valid lower bound when $m_{rdc}(\mathbf{x}) \leq 0$, but not necessarily when $m_{rdc}(\mathbf{x}) > 0$, because we used $m_{rdc}(\mathbf{x}) \leq 0$ in the proof. An example can simply confirm this. Consider an instance with two standard-size pieces in stock: a piece of length 0.7 and cost 0.6 and a piece of length 1 and cost 1. The demand consists of

two small items of lengths $w_1 = 0.7$ and $w_2 = 0.3$. Taking $x_1 = 0.5$ and $x_2 = 0.4$, one obtains $m_{rdc}(\mathbf{x}) = 0.6 - 0.5 = 1 - 0.5 - 0.4 = 0.1$ and we $\mathscr{L}(x) = \frac{0.9}{1-\frac{1}{0.6}0.1} = 1.08$ which is *not* a valid lower bound, since the optimum for this instance is 1 (cut both items from a standard-size piece of length 1).

Recall (last paragraph of Section 2.2.2) that the first two iterations of `Projective Cutting-Planes` for *Multiple-Length Cutting-Stock* solve the projection sub-problems `project`$(\mathbf{0}_n \rightarrow \frac{1}{W}\mathbf{w})$ and `project`$(\mathbf{0}_n \rightarrow \mathbf{b})$, obtaining two initial lower bounds. As described above, $\mathscr{L}(\frac{1}{W}\mathbf{w})$ is not a valid lower bound in standard `Column Generation`, because $m_{rdc}(\frac{1}{W}\mathbf{w}) \geq 0$. As such, even if we also (warm-)start the `Column Generation` by solving the separation sub-problem on $\frac{1}{W}\mathbf{w}$ and $\mathbf{b}$, the `Column Generation` generates only one initial lower bound $\mathscr{L}(\mathbf{b})$ for these two initial iterations.