Mathematical Programming Computation (2023) 15:153–194
https://doi.org/10.1007/s12532-022-00230-4

FULL LENGTH PAPER

A data driven Dantzig–Wolfe decomposition framework

Saverio Basso[1] · **Alberto Ceselli**[2]

Received: 3 May 2021 / Accepted: 27 September 2022 / Published online: 2 November 2022

Abstract
We face the issue of finding alternative paradigms for the resolution of generic Mixed Integer Programs (MIP), by considering the perspective option of general purpose solvers which switch to decomposition methods when pertinent. Currently, the main blocking factor in their design is the problem of automatic decomposition of MIPs, that is to produce good MIP decompositions algorithmically, looking only at the algebraic structure of the MIP instance. We propose to employ Dantzig–Wolfe reformulation and machine learning methods to obtain a fully data driven automatic decomposition framework. We also design strategies and introduce algorithmic techniques in order to make such a framework computationally effective. An extensive experimental analysis shows our framework to grant substantial improvements, in terms of both solutions quality and computing time, with respect to state-of-the-art automatic decomposition techniques. It also allows us to gain insights into the relative impact of different techniques. As a side product of our research, we provide a dataset of more than 31 thousand random decompositions of MIPLIB instances, with 121 features, including computations of their root node relaxation.

1 Introduction

Decision support systems have been widely adopted in industry, government and also academia. During the last decades, their evolution has been greatly impacted by the explosive rise of Big Data, as the availability of more data triggered the need of

S. Basso: The work has been partly carried out while the first author was at Dipartimento di Informatica, Università degli Studi di Milano.

✉ Alberto Ceselli
alberto.ceselli@unimi.it

Saverio Basso
saverio.basso@supsi.ch

1 Dalle Molle Institute for Artificial Intelligence, USI/SUPSI, Via la Santa 1, 6962 Viganello, Switzerland

2 Dipartimento di Informatica, Università degli Studi di Milano, Via Celoria 18, 20133 Milan, Italy

quantitative support for more complex decisions. Organizations that once relied only on ad-hoc systems and custom algorithms, now favor continuous delivery, integration and modularity principles.

Standing on huge theoretical, algorithmic and software engineering research efforts from our community, general purpose optimization solvers perfectly suit these needs. They offer full modeling flexibility and grant computational effectiveness, which is improving exponentially over the years [1]. State-of-the-art is built by commercial packages [2–4] and academic ones [5] alike with a common feature: to exploit continuous relaxations, cutting planes and branch-and-bound as an overall framework.

However, on a minority but well noticeable share of problems, such a paradigm appears inappropriate. It is the case for instance of vehicle routing, crew scheduling, cutting stock, facility location, generalized assignment, and many others, where *decomposition techniques* [6] yield often computational methods whose behaviour strongly outperforms that of generic branch-and-cut.

Additionally, there is a steady trend for computation offloading. Cloud storage and computing resources become increasingly affordable, and more and more computing services are being offered directly as applications on cloud platforms [7]. One driving factor is flexibility: fine resource provisioning can be performed by need, exploiting several smaller virtual units rather than few powerful ones. Data size and scalability, when distributed storage and concurrent computing units are available, are known to be an issue for generic branch-and-cut methods, while decomposition approaches naturally fit [8].

These reasons have motivated scholars in the search for paradigms combining the flexibility of general purpose solvers, which can be used and integrated as black boxes once data and models are given as input, with the computational effectiveness of decomposition techniques, conceiving decomposition-based general purpose solvers. While currently branch-and-cut based solvers on single workstations are orders of magnitude faster than decomposition based ones, except on a minority of problems, it is tempting to expect that the latter could overtake the former even on instances with no specific structure, when running on massively concurrent platforms.

We remark that the normal working condition of general purpose solvers is to get a generic Mixed Integer programming Problem (MIP) as input and optimize it, without additional knowledge of the combinatorial structure of the problem it encodes. This is clashing with the common practice for decomposition methods, where good decomposition patterns must be given to the algorithm by a human mathematical programming expert.

Therefore, a first research line considers the option to enrich input with a further element: a description of the decomposition to apply. Successful attempts include [9, 10], and more recently [11, 12]. The state-of-the-art is currently GCG [13], distributed with the SCIP framework [5]. It relies on Dantzig–Wolfe decomposition and fully generic column generation.

A second line of research attempts to relief the general purpose solver user from the explicit choice of a decomposition scheme, looking for a suitable one either on the basis of model metadata [14, 15], like annotations on the type of constraints which are encoded, or by the explicit search for specific structures in it (e.g. partitioning constraints, network flows, temporal layers) via *detectors* [13].

The most ambitious task is however that of *automatic decomposition of MIPs*. The working condition of general purpose solvers is kept: only MIP models and data are required to the user. Decomposition schemes are found by pure *algorithmic* detectors. They solve side combinatorial problems on so-called *static* properties of the MIP, that is without actually optimizing it. Such a research question is known in the literature as *automatic decomposition for MIPs*. Either heuristics [16, 17] or exact approaches [18] have been investigated. The overall GCG package [13] includes also state-of-the-art algorithmic detectors, to be activated as an option.

In particular, the authors of [17] provide an unexpected but fundamental result: algorithmic detectors and generic decomposition based solvers can experimentally outperform commercial ones (in their case [2]) not only on selected problems having a clear decomposable structure, but also on a subset of generic MIPs from the MIPLIB [19]. On average, however, computational results are mixed. As shown in [20], issues preventing further improvement seem to be deep in the choice of decomposition features that algorithmic detectors consider in their search process.
Preliminary investigations. In two preparatory workshop papers [21, 22] we have experimented the potential of machine learning tools in this context. We have mainly focused on two very specific tasks. First, from a given set of generic MIP instances and a set of decompositions produced for them by a randomized greedy algorithm, whose relaxation at the root node is actually computed by column generation, we trained regression and classification models. We found them to be able to predict bound and computing time for *other* random decompositions over the same set of MIP instances, or limited perturbations of them [21]. Second, we used these models for choosing how to *improve* decompositions by simple local changes [22]. We found them effective.

Indeed, other researchers have successfully tried machine learning in this context, producing complementary results on the question whether it is possible to filter which MIP instances might benefit from a decomposition approach in place of branch-and-cut, and which algorithmic detector is best suited [23].

Even if our preliminary investigations showed data driven methods to be promising in specific tasks, the results of [21, 22] are not directly applicable as a full computational tool. In detail, [21] shows how to rank decompositions in terms of distance from the Pareto front on the space of bound and computing time, but the problem of actually generating and selecting a specific decomposition of good rank in an overall optimization framework is only sketched. Neither the results of [22] are sufficient to be directly integrated in an overall framework, the main drawback being the computing time: the algorithms of [22] might take as long as the optimization phase itself. Further integration issues are not discussed, such as which combinations of data driven models, training techniques, decomposition generation and improvement methods work well together. Furthermore, only indirect computational evidence is provided: experiments are limited to the root nodes of a branch-and-price tree, testing is mostly limited to perturbations of training MIP instances, and comparisons with existing methods is limited to a specific use of GCG.

In this paper we fill the gap between the concepts of [21, 22] and their actual use in a full decomposition-based MIP resolution framework, proving its effectiveness in realistic contexts. We also describe and share a library of more than 31 thousands decompositions of MIPLIB instances, generated by a randomized greedy algorithm and scored by their root node computation.

In detail, we first formalize the theoretical framework we employ and we elaborate on results from the literature to provide the overall background needed to our framework (Sect. 2). We also detail which results come from our preliminary findings. Then, we propose an overall architecture, we discuss the design of its main components and the algorithmic engineering steps needed to obtain an effective implementation (Sect. 3). Finally, we carry on an empirical statistical analysis: we describe our experimental setup and the datasets we consider, we report on both preliminary results and parameter fine tuning and full MIP optimization experiments (Sect. 4). We conclude by discussing some perspectives (Sect. 5).

2 Background

From a methodological point of view, our framework relies on two pillars: mathematical programming and machine learning. Indeed their integration is a recent and lively research field [24–26].

We formalize the problem and our approach by means of mathematical programming (Sect. 2.1), and link it to the machine learning models we employ (Sect. 2.2).

2.1 Mathematical programming models and notation

Let us consider the following formulation for a *fully generic* Integer Linear Program (ILP):

$$\begin{aligned}
\text{(P) min} \quad & \sum_{j\in J} c^j x^j + dy & & \\
\text{s.t.} \quad & \sum_{j\in J} A^j x^j + Dy \geq a & & (1)\\
& B^j x^j + E^j y \geq b^j & \forall j \in J \quad & (2)\\
& x^j \in \mathbb{Z}_+^{n_j} & \forall j \in J \quad & (3)\\
& y \in \mathbb{Z}_+^m & & (4)
\end{aligned}$$

where c^j, A^j, a, B^j, b^j, d, D, E^j are rational data, x^j are vectors of n_j integer variables each, and y is a vector of m integer variables. We remark that formulation (P) readily extends to Mixed integer Linear Programs (MIP) by dropping a subset of integrality constraints, although we omit this case to avoid notation overload.

A Linear Programming (LP) relaxation is obtained by replacing (3) and (4) with

$$x^j \in \mathbb{R}_+^{n_j} \;\; \forall j \in J, \quad y \in \mathbb{R}_+^m$$

That however often yields weak dual bounds. The fundamental intuition allowing to successfully apply Dantzig Wolfe Decomposition (DWD) in this setting is in fact to consider the sets, for each $j \in J$

$$\theta^j = \left\{(x, y) | B^j x + E^j y \geq b^j, x \in \mathbb{Z}_+^{n_j}, y \in \mathbb{Z}_+^m\right\}$$

arising from (2), (3) and (4). From a geometric point of view, since (2) are linear, each θ^j induces a polyhedron, whose boundary is defined by a set $\mathcal{S}^j$ of extreme points, and a set $\mathcal{R}^j$ of extreme rays. The latter indicate directions in which the polyhedron induced by θ^j is unbounded, if any. Each θ^j can therefore be replaced in (P) by the convex hull defined by $\mathcal{S}^j$ and $\mathcal{R}^j$, which we denote as conv$\{\theta^j\}$. We can finally obtain a relaxation for (P) by replacing (3) with $x^j \in \mathbb{R}_+^{n_j}$ for each $j \in J$ and (4) with $y \in \mathbb{R}_+^m$:

$$\begin{aligned}
\min \quad & \sum_{j \in J} c^j x^j + dy & \\
\text{s.t.} \quad & \sum_{j \in J} A^j x^j + Dy \geq a & \qquad (5)\\
& (x^j, y^j) \in \text{conv}\{\theta^j\} & \forall j \in J \\
& y^j = y & \forall j \in J \\
& x^j \in \mathbb{R}_+^{n_j} & \forall j \in J \\
& y \in \mathbb{R}_+^m & \\
& y, y^j \in \mathbb{R}_+^m & \forall j \in J \qquad (6)
\end{aligned}$$

By an inner representation principle, conditions (6) can be enforced by imposing x^j and y^j variables to be obtained as a linear combination of elements $(x_k^j, y_k^j) \in \mathcal{S}^j$ and $(x_k^j, y_k^j) \in \mathcal{R}^j$

$$\left(x^j, y^j\right) = \sum_{k \in \Omega^j} \left(x_k^j, y_k^j\right) z_k^j + \sum_{k \in \chi^j} \left(x_k^j, y_k^j\right) w_k^j$$

where each Ω^j is the set of indices of elements in $\mathcal{S}^j$, each χ^j is the set of indices of elements in $\mathcal{R}^j$, $\sum_{k \in \Omega^j} z_k^j = 1$ for each $j \in J$, each $z_k^j \in \mathbb{R}_+$ and each $w_k^j \in \mathbb{R}_+$. That is, our relaxation of (P) is remapped into the following so called explicit *extended form*:

$$\begin{aligned}
\text{(Q)} \min \quad & \sum_{j \in J} c^j x^j + dy & \\
\text{s.t.} \quad & \sum_{j \in J} A^j x^j + Dy \geq a & \\
& (x^j, y^j) = \sum_{k \in \Omega^j} (x_k^j, y_k^j) z_k^j + \sum_{k \in \chi^j} (x_k^j, y_k^j) w_k^j & \forall j \in J \\
& \sum_{k \in \Omega^j} z_k^j = 1 & \forall j \in J \\
& y^j = y & \forall j \in J \\
& x^j \in \mathbb{R}_+^{n_j} & \forall j \in J
\end{aligned}$$

$$\begin{aligned}
&y \in \mathbb{R}_+^m, y^j \in \mathbb{R}_+^m && \forall j \in J \\
&0 \le z_k^j \le 1 && \forall j \in J \ \forall k \in \Omega^j \\
&0 \le w_k^j && \forall j \in J \ \forall k \in \chi^j \qquad (7)
\end{aligned}$$

Vectors x^j and y^j can then be removed by projection, exploiting constraints (7).

When applied wisely, such a relaxation can be much stronger than a plain continuous relaxation, thanks to the so called *convexification* of the set of constraints (2). It is actually as strong as a cutting plane approach, in which all the facets of the region described by constraints (2) (3) and (4) are generated.

From a computational point of view the extended form (Q) has one additional column for each extreme point in $\mathcal{S}^j$ and each extreme ray in $\mathcal{R}^j$, that are combinatorial in number. It is therefore common to pair DWD with column generation techniques [27] to implicitly handle the set of z_k^j and w_k^j variables. Column generation is currently a very well understood technique, which is also known to scale very well as the amount of computing resources increases [8]. In fact DWD with column generation proves successful as bounding procedure in many branch-and-bound algorithms [6]. Effective computing frameworks have also been developed, which take MIPs formulated as (P) and fully optimize them by column generation and branch-and-bound [13].

A fundamental observation is therefore the following: *any* MIP can be written as (P), whenever an *arbitrary* partitioning of its constraints is made, defining which constraints belong to (1) and which belong to (2) for each $j \in J$. In fact, such a choice induces a corresponding partitioning of the MIP variables in the vectors x^j, those appearing only in constraints of family (2) whose index block is j. Those variables y appearing in more than a single block $j \in J$ build an additional *vertical border* of linking variables. Those constraints in which variables from multiple x^j vectors appear build an additional *horizontal border* of linking constraints.

Intuitively, after proper rearrangement, the partitioning is defining a block diagonal structure in the constraint matrix: once constraints (1) and variables y are removed, the MIP disaggregates into an independent subproblem for each $j \in J$. We therefore refer to such a partitioning, and with an abuse of terminology also to the block structure of constraints (2) induced by such a choice, as a *decomposition pattern*. We finally formalize the following.

Def. Automatic Decomposition of MIP. Given an arbitrary algebraic representation of a MIP, rewrite it as (P) by finding a suitable decomposition pattern, that is a partitioning of its constraints into a horizontal border and a set of blocks identified by J, which is also inducing a partitioning of its variables into vectors y and x^j.

Among the combinatorially many possible decompositions of a MIP, we are interested in those maximizing some quality measure, the ultimate one being the computational effort it takes to solve the MIP by exploiting it.

Def. A static feature of a MIP [23] is any numeric value which can be computed from its algebraic formulation only, without actually optimizing the MIP.

Def. A static feature of a decomposition is any numeric value which can be computed from the algebraic formulation of the MIP after the definition of the set J and the corresponding partitioning (that is, from data c^j, A^j, a, B^j, b^j, d, D, E^j), without actually optimizing the MIP.

That is, by *dynamic features* we indicate all the remaining ones, and in particular those requiring an optimization process to be evaluated. We remark that currently, according to these definitions the ultimate measure of decomposition quality cannot be encoded as a static label.

2.2 Computational methods

The issue of automatic decomposition of MIP is currently the main obstacle in making decomposition-based solvers like [13] as effective as generic branch-and-cut based solvers like [2–4]. While the latter require no input besides the MIP itself, decomposition-based solvers require a decomposition pattern to be given along (or to be generated by pre-processing plugins). If finding an arbitrary one is trivial, identifying one of high quality asks for specific mathematical programming skills.

The main issue is evaluating the quality of a decomposition *before* actually using it in an optimization run.

Theoretical investigations trying to highlight structural results on the quality of decompositions are hard to carry out, unless specific combinatorial problems are considered. For instance, in [28] the authors investigate the strength of Dantzig–Wolfe reformulations for the Stable Set problem, and provide interesting insights. In particular, they were able to give a characterization of the strongest and weakest decomposition in terms of dual bound.

Currently, the most successful approaches to general MIPs are fully empirical. That, is some *proxies* for decomposition quality are considered, which readily come from single specific static features, namely number of blocks ($|J|$) and size of the borders (number of constraints (5) and number of variables y). Algorithms optimizing these features are termed *static algorithmic detectors* in the literature. Unfortunately, these measures are not always consistent [20]. On the other hand, it is known that decompositions quality is not random. For instance, in [29] the authors performed a statistical investigation by generating and solving all the possible Dantzig–Wolfe decompositions of a collection of instances. Their study highlights that only a small number of different bounds occur, suggesting that a hierarchy of Dantzig–Wolfe decompositions exists and that more often than not, random reformulations produce weak bounds. Therefore, decompositions quality is expected to be predictable to some extent.

In fact, statistical investigations are more fruitful. Most rely on a common assumption: the effort it takes to fully optimize the corresponding MIP can be *estimated* by a data driven approach, considering the optimization process of instances which are similar in terms of static features. We report, in particular, the results of [23]: the authors collect a dataset of optimization run logs, matching static MIP features with the computing time required by decompositions produced by various static algorithmic detectors of [13]. Then they train machine learning models which are able, given a new MIP instance, to predict which is the most successful detector to use, and if a decomposition approach is promising or not for that specific MIP. Their positive results open the road to a realistic option in practice: to preprocess the MIP instance and run decomposition-based solvers *only if* it is predicted to be amenable for a decomposition approach, running standard branch-and-cut otherwise. In principle, machine learning models could be employed to further enhance the performance of that approach, tuning the sets of detector parameters as well [26, 30]. However, even that would bring little improvement if the predictive features are not those considered by the algorithmic detectors.

Indeed, independently and concurrently, we have carried out preliminary investigations which also use a data-driven approach, but aim at an orthogonal intent: that of clarifying which static features are worth considering. In particular, we investigate if *combinations of static features* exist, which are predictive of decompositions quality, and how to exploit them.

We focus on unstructured problems, that is, those problems for which no semantic information is enriching the MIP. As explained in the Introduction, this is a typical working condition of a general purpose solver. Additionally, structured instances from the literature typically present only few (usually one) good decompositions. Therefore we expect problems with special structures to be easily detected and handled by techniques from the literature.

We tackle the automatic decomposition problem with a two step detection process, that is decomposition *generation* and decomposition *scoring*. Our models mainly focus on scoring. We keep as proxies for a decomposition quality two parameters: the duality gap produced by the corresponding relaxation (Q), and the time it takes to compute it by means of column generation, normalizing them to scores, which are real values in a range [0, 1], the higher the better.

In [21] we propose supervised learning models, mapping static decomposition features to bound and time scores, exploiting a dataset including about 1000 decompositions for each of 36 base MIP problems from the MIPLIB. These decompositions were sampled by a randomized greedy algorithm that iteratively picks constraints with a probability directly proportional to their sparsity, builds well-formed blocks and possibly aborts and restarts the process if the structure of the tentative decomposition is not satisfying certain criteria (in our implementation, at least three distinct blocks must be present). With an abuse of terminology, through the paper we refer to a decomposition generated by such a procedure as *random decomposition*, although it is in fact a sampling from a set which is much smaller than that of all possible decompositions, and is not performed with uniform probabilities.

We show that data driven regressors are successful in predicting time score, when applied to both new decompositions for one of the known 36 base MIP instances, and to new decompositions for *new* MIP instances which are obtained by perturbing data of the base ones. Instead, bound score prediction is reliable mostly when facing new decompositions of known MIP instances. In [21] we also sketch a proposal for a fully *Data Driven Detector*, that is a software component that receives in input a MIP instance and produces as output a decomposition pattern using only data driven models. In that proposal, given a MIP instance, our detector generates a set of random decompositions and then, in its best configuration, scores them with data driven bound and time regressors. The set is then ranked with a dominance function in which for each decomposition i we compute the percentage of decompositions that i dominates in the set, in terms of bound and time predicted scores. More in detail, given decompositions i and j with time score T_i (resp. T_j) and bound score B_i (resp. B_j), decomposition i dominates j if and only if:

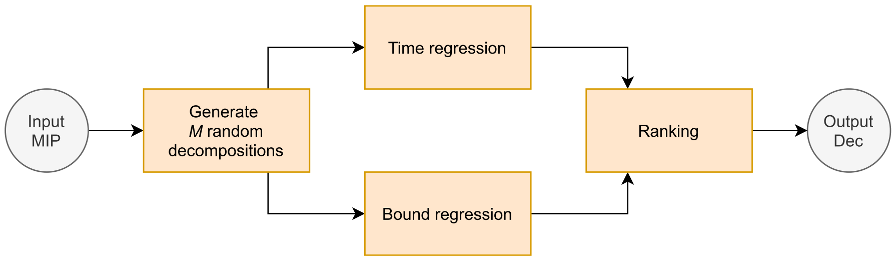


Fig. 1 Sketch for the proposal of a data driven detector

$$(T_i > T_j \wedge B_i \geq B_j - \epsilon) \vee (T_i \geq T_j - \epsilon \wedge B_i > B_j).$$

The decomposition with the higher dominance score is then returned as output. Additional configurations include a preprocessing filter based on a classifier of good and bad decompositions, in the sense of Pareto-Optimal in the space of time and score bounds, and the possibility of returning more than one output. For completeness, in Fig. 1 we report the outline of our proposal.

Besides seeking for good computational behavior, such an architecture is also designed to evaluate the potential of a statistical approach to the generation of good decompositions. In fact, surprising properties of random sampling solutions have already been discussed in the literature [31], although in different contexts. Overall, experimental analysis on synthetic, previously unseen MIP instances reveals that our proposal is promising. In particular, the top 8 decompositions ranked by our prototype, for every instance, showed an average dominance score equal to 84%, that is, the decompositions were very close to the Pareto frontier in the space of time and bound scores of those that were generated by our algorithm.

In [22] we tackle the problem of further refining decompositions to improve their computational performance features. To this end, we develop a *local search algorithm* that, given a decomposition for a MIP instance, exploits our data driven models to reformulate it, providing as output a new decomposition pattern. A *solution* is therefore an assignment of each constraint to either one of the blocks or to the border. A *move* consists in removing one constraint from the border, and inserting it in one of the blocks. Hence, the *neighborhood* we consider is the set of all decompositions whose border is identical to that of the current solution, except for a single constraint which is missing (and appears instead in one of the blocks). More in detail, we generate each *neighbor* decomposition by *moving* a constraint from the border to one of the blocks, and we repeat this for every constraint in the border and every block. Therefore, although polynomial in size, from a computational point of view, the neighborhood can contain a very wide selection of decompositions. Furthermore, this move guarantees that the bound cannot worsen, since the operation has the effect of including one more constraint in the definition of the set θ^j on which convexificaton is performed (and therefore starts having effect on the single extreme points and rays in $\mathcal{S}^j$ and $\mathcal{R}^j$ instead of their linear combination). To *explore* the neighborhood we evaluate all its elements with our data driven time predictor only, and select a candidate with a *best improvement* search strategy. More in detail, a decomposition is chosen that is

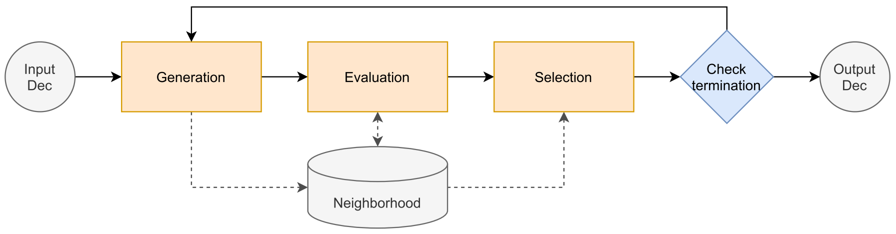


Fig. 2 Local search algorithm outline

reported to be faster. We repeat these operations until termination. The outline of the algorithm is reported in Fig. 2.

We performed preliminary tests at the root node of the branching tree with a prototype implementation, on a small set of 5 *previously unseen* problems from MIPLIB2017 [32]. In this preliminary setting, local search is capable of consistently improve decompositions provided by either static detectors or the data driven detector from [21].

3 Framework architecture

In the following we propose our fully data driven framework for detection and enhancement of Dantzig–Wolfe decompositions. In terms of architecture, the framework is composed of the following major components: Decomposition-trainer (D-trainer), Decomposition-preprocessor (D-preprocessor), Decomposition-optimizer (D-optimizer).

The overall structure of the framework is detailed in Fig. 3. Its typical use flow is designed to be the following. The user activates, in an offline phase, the D-trainer: it setups and trains the data driven models that will be used by the other components, exploiting a database of existing optimization run logs, and possibly enriches it by

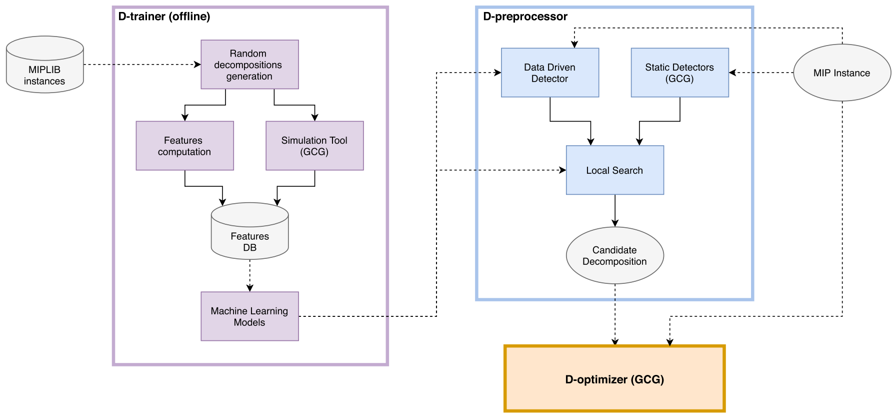


Fig. 3 Framework outline

further simulations. As an example, the D-trainer database can be initially populated by runs of random decompositions of MIPLIB instances (as we did in our experiments).

At runtime, instead, the user submits a new MIP instance to our framework. Such a MIP instance is handled by the D-preprocessor, which detects if it is amenable for a decomposition approach, generates a decomposition and potentially enhances it.

When this operation is complete, the MIP instance together with its decomposition pattern are given as input to the D-optimizer, which carries out the optimization process.

3.1 Component design

The most challenging component from a methodological point of view is clearly the D-preprocessor, as the D-trainer allows the embedding of effective existing machine learning techniques, and the D-optimizer may consist in running generic implementation of a branch-and-price algorithm, which have already been proposed in the literature.

D-trainer The Decomposition-trainer is an offline component that setups our machine learning models to be later used by the D-preprocessor. In particular, we assume to work with a dataset of decompositions, described by a set of features, that can be either provided beforehand or can be generated by the tool from a collection of MIP instances. When facing the second scenario, we first generate a well diversified set of random decomposition patterns for every MIP instance. We recall that by random decompositions we actually mean those produced by the randomized greedy algorithm that we proposed in [20]. Furthermore, the features we propose to use are those listed in [20]. They can all be computed by algebraic computations on the elements of either the MIP instance or the decomposition pattern, except for a total unimodularity score, discussed in Appendix 1. At every iteration of the algorithm, one constraint is sampled and assigned to a new block, while taking care of merging blocks that share common variables. When these operations are finished, preprocessing features are computed along with computational ones, obtained through simulations that run until either a timelimit is reached or the root node of the branching tree has been fully processed. Finally, the dataset is cleaned and normalized, with a setup identical to [20].

The dataset is then finally used to train independent regressors, for time and bound. We found decision forests trained by means of gradient boosting to be a pertinent choice in this step.
D-optimizer The Decomposition-optimizer works at run-time and makes use of generic column generation frameworks to solve the candidate decomposition, obtained from D-preprocessor, to optimality. We remark that this is a completely generic approach and therefore simulation performance is strongly bounded by the available tools.

3.1.1 D-preprocessor

When a user submits to our framework a MIP instance for optimization, we use our machine learning models to detect and improve a decomposition. A detection phase is employed to provide a starting decomposition either by using static algorithmic detectors like those of [13] or using a data driven detector by filtering, with a setup like that of [21]. In detail, we generate M decomposition as follows. We run the greedy randomized algorithm of [20] searching for decompositions with at least 3 blocks; if the algorithm fails 10 times, we perform 5 more attempts, requiring at least 2 blocks; if none can be found, the dummy decomposition with a single block is taken. We repeat this process M times, we compute their static features, apply the models provided by the D-trainer and rank them through a custom dominance function based on the scores provided by the time and bound regressors. Then we consider the decompositions produced by the static algorithmic detectors and score them. The best decomposition is finally chosen.

When detection is complete we enhance the starting decomposition with local search, with settings similar to the ones proposed in [22] and reviewed in Sect. 2.2. That is, we iteratively explore a neighborhood of decompositions and choose the reformulation that is predicted to be of highest time score by our models, with the aim of progressively strengthen the dual bound. We present the D-preprocessor pseudo-code in Algorithm 1. We report that two main issues complicate local search: (a) the neighborhood is large, and each solution needs to be evaluated by an external regression model and (b) the effect of each move can be measured only indirectly, by considering a *prediction* on the effect of convexifying one constraint at a time. Traditional local search methods did not prove successful.

In our preliminary attempts, the algorithm terminates either when a certain percentage of constraints is present in blocks (Cvx) or when the next candidate decomposition predicted quality has a considerable drop from the best predicted score (T_{best}) found among all iterations.

Although promising, performances were not always consistent. Therefore, we propose to further improve the local search methods in three directions. First, we consider the impact of different selection strategies that involve multiple decompositions. Second, we consider how to improve computing times by means of sampling. Third, we design an efficient implementation that allows to tackle even large scale instances.

Large neighbourhoods exploration by constructive selection strategies We explore the possibility of simultaneously selecting two moves, that convexify constraints in different blocks. To this end, after the neighbourhood has been generated and a score has been assigned to each of its solutions, we scan the neighborhood for a pair of moves to generate a new solution.

We choose the moves that yield decompositions with minimum predicted time that have been generated through the selection of constraints that form an orthogonal pair, that is, two constraints that share no variables. This has the effect of reducing block merging. We take care of selecting ones whose score is within a certain threshold from a decomposition of maximum time score. If no orthogonal pair can be found having these features, only a single decomposition of highest predicted time score is chosen.

Instead, when an orthogonal pair is detected we generate a new decomposition by moving first the most promising constraint to its assigned block. Since this move can change the overall structure of the candidate decomposition, we check that the newly modified block and the second constraint are still orthogonal. If they are, we move the

```
Data: MIP instance $P$, detection strategy $DD\_detection$, number of starting decompositions $n$
Result: decomposition $D'$
if $DD\_detection$ then
    $\Delta = \emptyset$;
    for $i \in seq(0,n)$ do
        $D_{random}$ = generate random decomposition(P);
        $\Delta = \Delta \cup D_{random}$;
    end
    T = predict times($\Delta$);
    B = predict bounds($\Delta$);
    R = compute ranking($\Delta$, T, B);
    D = return first ranked($\Delta$, R);
else
    D = find static decomposition(P);
end
$D_{cand}$ = D;
enhance = true;
while $enhance$ do
    $\Gamma = \emptyset$ ;
    for $i \in border(D_{cand}),\ j \in blocks(D_{cand})$ do
        $D_{new} = D_{cand}$;
        move(i, j) =
            remove i from border($D_{new}$);
            add i to block($D_{new}$, j);
            merge blocks($D_{new}$);
        $\Gamma = \Gamma \cup D_{new}$;
    end
    T = predict times($\Gamma$);
    sort($\Gamma$, T);
    D' = return highest time score decomposition($\Gamma$);
    if $check\ termination()$ then
        enhance = false;
    else
        $D_{cand}$ = D';
    end
end
```

Algorithm 1: D-preprocessor pseudo-code

constraint in its respective block. Otherwise, we scan for another orthogonal constraint and if one is found, we insert it. We then proceed to the next steps of the algorithm.

An outline of the orthogonal selection is proposed in Fig. 4.

Improving generation efficiency through sampling Repeating the exploration of the full neighborhood space at every iteration is expected to be very taxing in terms of computing time, and possibly memory, when facing large instances or when taking into account more complex neighborhood exploration strategies like orthogonal selection. For this reason, we propose an alternative generation procedure that makes use of sampling to improve efficiency while guaranteeing that a representative portion of the neighborhood is considered. It relies on a standard argument from statistics, but we are not aware of similar adaptations in a local search context as ours. In detail, we look for an estimate of the neighborhood with probabilistic guarantees. We remark that by drawing a random subset of decompositions, and computing their average score (resp.

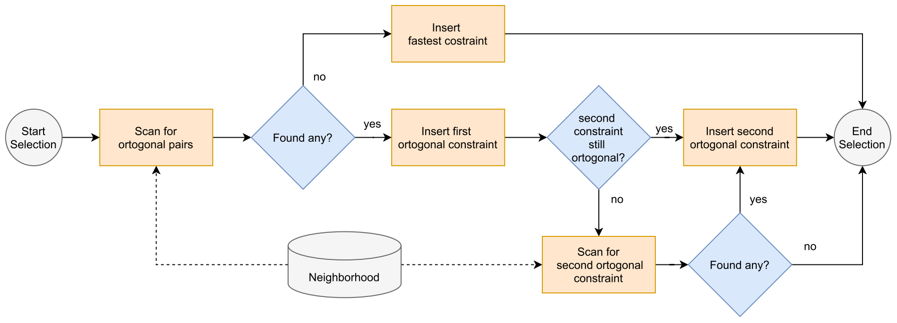


Fig. 4 Local search algorithm (orthogonal selection outline)

the standard deviation of this sample), the sample mean (resp. sample variance) is an unbiased reliable estimator of the mean score (resp. its standard deviation) [33].

Let N be the subset of decompositions in our sample, with $k = |N|$, and let s_i be the score of each decomposition $i \in N$. First, we can compute their sample mean $\bar{s}$

$$\bar{s} = \sum_{i \in N} \frac{s_i}{k}$$

and their sample variance

$$\sigma^2 = \frac{\sum_{i \in N} (s_i - \bar{s})^2}{k - 1}$$

With these two values we compute a $(1-\alpha)$ confidence interval estimate of the average score $\tilde{s}$ in the neighborhood:

$$\tilde{s} \in \left(\bar{s} - z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{k}}, \bar{s} + z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{k}} \right)$$

with probability $(1 - \alpha)$, where $z_{\frac{\alpha}{2}}$ is is the quantile value $\frac{\alpha}{2}$ of the standard normal random distribution. That is, our modified generation samples from a uniform distribution k decompositions in the neighbourhood, evaluates them with our data driven time regressor and then computes the sample mean and the sample variance to build a confidence interval estimate of the average value in the neighborhood. Whenever the interval falls within a certain threshold, that we set as a parameter, the generation step is stopped, and the decomposition of highest score is returned. Otherwise, sampling is repeated. In such a way we have a guarantee, although probabilistic, that the chosen decomposition has a score which is better than the average. Furthermore, since the squared difference of its score with respect to $\tilde{s}$ is the maximum over the sample, and therefore trivially not lower than σ^2, we also expect it to be better than a large fraction of the decompositions in the neighbourhood. Empirically, it is in fact the case.

The principle of such a procedure is in fact similar to a best improving move in local search. However (a) the scores are estimated by regressors, which give no special

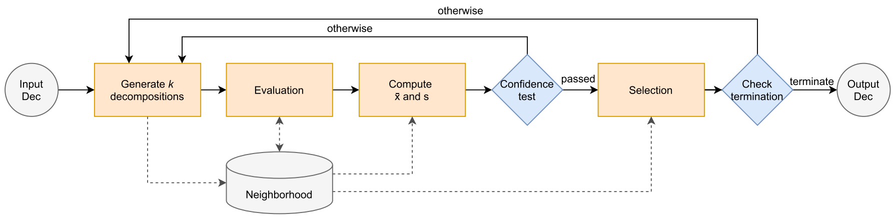


Fig. 5 Local search algorithm (sampling outline)

structure to drive exploration algorithms (b) scores are statistical estimates, which are more reliable when many of them are measured and compared.

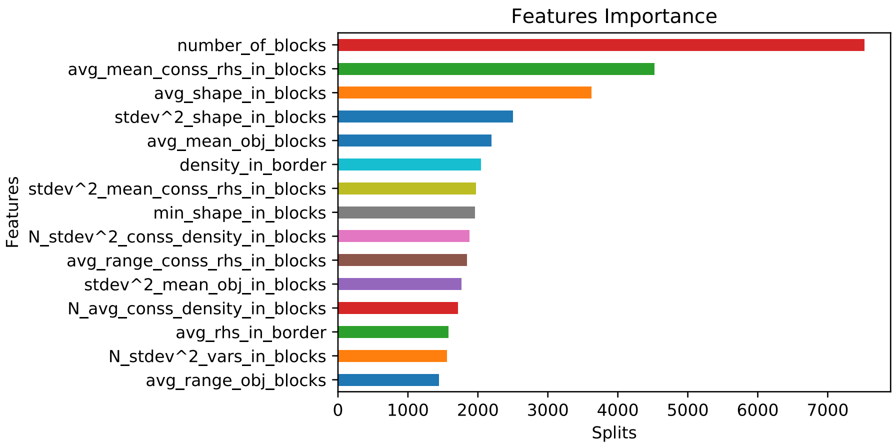
After generation has been completed the algorithm proceeds to selection and the subsequent steps. A graphical outline of the algorithm with sampling can be found in Fig. 5.

3.2 Implementation

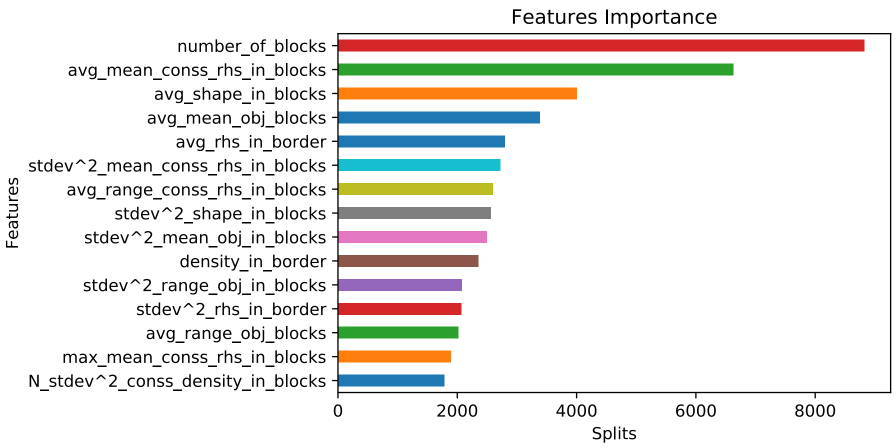
Finally, we detail our implementation choices.
D-trainer When needed, we handled the generation, cleaning and feature normalization of the dataset with simple R scripts. Simulations were computed with the Generic Column Generation (GCG) 2.1.1 tool.

We used xgboost library 1.0.2 of Python 3.6 to train our data driven models, with the following custom parameters: maximum depth of a tree $(max\_depth) = 13$, learning rate $(eta) = 0.1$, maximum number of iterations $(nrounds) = 100$.

In Fig. 6 we report an estimate of the most important features of our regressor models for both time and bound. Abbreviations in the figures are as follows: average (avg), standard deviation (stdev), constraints (conss), right hand sides (rhs), objective coefficient values (obj), number of variables/number of constraints (shape), number of nonzeros/(number of constraints · number of variables) (density), max - min (range). Leading N means normalized. In detail, xgboost trains a forest of decision trees [34]. Each of them recursively splits the dataset by performing tests on single feature values (split points), until a sufficiently homogeneous partition is obtained. As a measure of feature importance we take the number of nodes in which the feature was chosen for the split test. On the y-axis we present the name of the feature whilst on the x-axis we report the number of xgboost feature splits. We notice that 80% of the top features are shared between the two models. In particular, the number of blocks results the most important one. This is expected and consistent with the findings of [17], that show that the number of blocks can be a proxy for the solving time of a given MIP with a given decomposition. Indeed, such a phenomenon is likely to appear not only for Dantzig–Wolfe, but for any decomposition method. However, this results not being the only good indicator: our models combine it with features mainly describing the shape of the blocks and the right hand side of the constraints. Further research is likely to be necessary to understand the specific impact of each feature, both independently and in combination with the others. Nevertheless, we experimentally observe such a phenomenon not to be negligible, as the splits using one of those two features are more than those using the number of blocks.



(a) Time



(b) Bound

Fig. 6 Time and bound feature importance

D-optimizer We chose the generic branch-and-price implementation provided by GCG 3.0.1 with standard settings, giving as input both the MIP instance and the decomposition scheme produced by the D-preprocessor. In our implementation, GCG makes use of CPLEX 12.6.3 and SCIP 7.0.0.

3.2.1 D-preprocessor

The core of our framework was developed by integrating minimal Python 3.6 scripts with a C++11 ad-hoc library. The former is used to handle the whole architecture by managing input and output, logs, parameters, loading and usage of machine learning models, and calling additional bash scripts and the static algorithmic detectors of GCG 3.0.1. In particular, it coordinates detection and employs our local search algorithm. Despite the skeleton of the framework, that includes quality evaluations and termination of the algorithms, is managed through Python, we designed our code to be as efficient as possible and deployed all computational intensive operations in our C++11 library. Boost 1.72 was used to manage its integration. Our C++11 library performs 3 major operations: generation of random decompositions through a sampling algorithm, efficient features computation, neighborhood generation for local search.

As reported, we use the same sampling algorithm that was designed in [20]. However, we further improved it by exploiting thread parallelization through the OpenMP library, with the aim of making data driven detection faster.

The last two operations are instead pivotal to employ local search algorithms on large scale instances. We achieved efficiency mainly by designing smart data structures. In particular, we used the Boost library and vectors of dynamic bitsets to encode the constraint matrix of the input instance and masks to show the variables set of each block of a decomposition. This setup provides faster operations and has the advantage of being memory efficient, since each element of the bitset occupies only one bit, allowing to store large scale instances without issues.

However, when exploring full neighborhoods during local search, memory management is still important. In our solution, when we generate a new neighborhood we start from a candidate decomposition. We create a new temporary one each time we make a move, merging blocks that share common variables. Since these operations have to be repeated each time we create a new decomposition, we avoid resizing any data structure and we simply use vectors of boolean variables to keep track of which blocks are active and which are now merged in other blocks. Features are computed when a candidate decomposition is generated and are stored in suitable data structures. Since temporary decompositions modify one block, and disable the ones that have been merged, we recompute only the features for that specific block. Aggregators are then used to compute means and other common statistics. Once features have been computed, we store them and we discard the decomposition to create the next temporary one. When generation has been completed and the neighborhood has been scored, the new candidate decomposition is chosen and has to be recomputed. On one hand, discarding decompositions has the advantage of saving memory. On the other hand, it does not allow for easy computation of multiple decompositions in parallel. A prototype of the latter scenario was implemented in the initial stages of development. However, this version could not handle large input instances when exploring the full neighborhood. It could however be viable when using sampling but, in general, requires tuning depending on the workstation and the input instance.

We remark that further improvements could be possibly achieved by storing sparse matrices in suitable data structures through compression. This should further help memory management and access speeds, in particular, when facing large scale instances.

4 Experimental analysis

Datasets In the following, we detail the two datasets that were used for training and testing our framework.

Dataset A, was taken from [20] and consists of 31,507 decompositions for 36 problems from MIPLIB2003 [35] and MIPLIB2010 [19]. For each decomposition, we collected 117 static features and run simulations with GCG 2.1.1 to measure computing time and dual bound, adding them as scores to the dataset. We passed all of Dataset A to the D-trainer component to train our data driven models.

Dataset B consists of 30 previously unseen problems from MIPLIB2017 [32], split equally among easy, hard and open instances. When possible, we chose instances that were listed in MIPLIB as "similar", from a feature analysis, to those that are part of Dataset A, that is used for training. A full report of the problems can be found in Table 5, in the Appendix. We used Dataset B for benchmarks. We remark that none of the instances of this Dataset is part of the training Dataset A: it consists of all previously unseen problems.

We make Dataset A openly available to the research community[36].

Benchmark algorithms and parameter configuration In the following, we benchmark state of the art static detectors and different configurations of our framework. In particular, we compare against multiple configuration of GCG 3.0.1: GCG with standard settings (GCG), GCG with only hmetis based detectors (GCG_{Hmetis}) and GCG with all detectors manually enabled (GCG_{Full}). We chose GCG and GCG_{Full} to represent two different scenarios in which the user either uses stock GCG to detect well known structures or activates all the detectors for a more general approach. Since we are focusing on unstructured instances, we also considered exploiting graph based detectors only (GCG_{Hmetis}). We note, however, that they are also included in GCG_{Full}. Additional standalone, fine tuned detectors were considered during preliminary tests, however, performance was similar with the other configurations, improving only specific instances. Furthermore, with some settings, we faced frequent undocumented solver errors. Finally, we compared against the community based algorithms of [37], whose best implementation is featured in the DECOMP module of the commercial solver SAS (SAS_{comm}). Unfortunately we had no access to the full stand-alone version of SAS, but we could run experiments on the SAS-OnDemand cloud service. According to the aim of our test, only the community algorithm was enabled.

As outlined, when a new instance is given as input to our system, we detect a starting decomposition by either using GCG static algorithmic detectors (standard settings) or by filtering and selecting among randomly generated decompositions with data driven ranking models. Then, we improve this decompostion with local search methods. We label the configuration that makes use of our data driven techniques for detection DDW. In this setting, we also consider two additional scenarios, detailed in section 3.1.1, that include sampling for improving local search computing time (DDW_{sample}) and sampling along with an orthogonal selection move (DDW_{ortho}). When we use GCG for the initial detection instead, we label this version DDW_{GCG}. Since both GCG and DDW_{GCG} share the same starting decomposition, this scenario can be seen a straight up comparison of performance for local search. Otherwise, the framework employs our data driven filtering and ranking for initial detection.

For our data driven detectors, after preliminary experiments, we chose $M = 1200$ when facing small instances that presented a constraint matrix made of, at most, 1 million entries. Otherwise, $M = 120$ was used. For our local search algorithm, $Cvx = 85\%$ and a quality threshold set to 15% from T_{best} were found suitable for termination. Finally, under the DDW_{sample} and DDW_{ortho} settings, we chose a sample size of 1000 decompositions and a target interval estimate of the neighborhood mean of ± 0.01 with a 95% confidence.

Experiments were performed on a workstation with Ubuntu 16.04 operating system, equipped with a quad-core Intel(R) Core(TM) i7-6700K 4.00 GHz CPU and 32 GB RAM. Results concerning SAS_{comm} include only bounds quality, as no detail about the actual hardware dedicated to the cloud service is given. We also remark that not only the detection algorithm changes, but the full computing framework, that could in principle apply different pre and post processing techniques than GCG.

Computational issues We preface that in some conditions we were not able to either use our local search methods or conclude some experiments.

First, we note that GCG_{Full} would crash with all the available detectors enabled. We were only able to conclude experiments only by disabling the following algorithms: $dbscan$, $constype$ and $compgreedly$.

Then, we report that with some optimization problems, the decomposition detected by GCG had no horizontal border. This had the side effect that we could not apply our local search moves, that create new decompositions by moving one constraint from the border to one of the blocks. When using data driven detection instead we faced issues during local search when the size of the neighborhood was too large (above 10 million decompositions) causing problems with data structures and computational overhead. Furthermore, some decompositions could not be simulated with GCG, as the tool would crash for undocumented internal reasons after few minutes. In the following experiments, problematic instances are not reported or are presented with empty records in tables.

4.1 Parameters tuning and preliminary experiments

We performed preliminary experiments over selected instances of Dataset B, at the root node, with a 2 h timelimit for optimization. In particular, we focused on improving computational time of heuristic features and on tuning selection strategies for our local search algorithm. In the following, we report a summary of our preliminary results and observations: full details can be found in the Appendix, in Sect. 1.

Heuristic features tuning First, we analyzed the impact of generating features at run-time, during our local search algorithms. In particular, preliminary experiments showed that computing our total unimodularity feature (TU), a multi-round heuristic score that describes similarities between each block of a given decomposition and a total unimodularity matrix, could be computationally extensive. We therefore considered different parameters, and found that a one-shot approach had little impact on time, while providing strong bound improvements for particular instances.

Selection strategies profiling Then, we studied the impact of alternative selection strategies for our local search algorithm. At every iteration, we considered choosing the best candidate decomposition by exploiting the prediction of our data driven time regressors, the number of blocks or hybrid approaches. In particular, we found that hybrid solutions performed better. When the given decomposition was given by a static detector, a more classical approach, that took into account the number of blocks first and the predicted time score second worked marginally better. When the given decomposition was obtained from our data driven approach instead, considering the time regressor first, and then the number of blocks provided the best results: about 10% better relative bounds with respect to the other configurations. We therefore used this selection in all the subsequent experiments.

4.2 Overall framework analysis

A more extensive experimental analysis was performed on the full Dataset B. We first discuss detection and local search performance, then we present results at the root node and with no node limit.

Detection and local search performance We first discuss performance and behavior of detectors. In every experiment, we imposed a 2 h timelimit for local search for DDW_{GCG} and DDW. In case of timeout at least one iteration was always completed. This timelimit was instead set to 10 mins for DDW_{sample} and DDW_{ortho}. In Table 1 we present average results for every configuration (Conf.) and Easy (E), Hard (H) and Open (O) instance categories (S.), dividing them in the following categories: preprocessing times, local search (LS) neighbourhood size and local search termination information. We report for time profiling: detection time (T. Det) and local search time (T. LS) in seconds. Neighbourhood exploration is summarized with the size of the neighborhood at the first iteration (Size) and the overall number of decomposition sampled (Sampled), whilst information about termination shows the percentage of constraints in blocks before (S.Cvx) and after (E.Cvx) preprocessing, the overall number of times local search terminated due to reaching the quality threshold (Sl) and the number of instances (Ipr) in which we were able to perform at least one iteration of local search (S.Cvx lower than 85%). Additionally, we also report the number of iterations of the algorithm (It).

Experimental observation 1 *Static detection is very fast on all instances. Data driven detectors are competitive in all scenarios but two.*

Generally, static detectors are really fast and can find a decomposition in few seconds (T. Det). This can be observed with the DDW_{GCG} configuration. In the other settings, that employ data driven techniques, on average, more than 10 mins are necessary to complete detection. We report, however, that this due to overhead in the random sampling algorithm when facing a couple of very large problems with more than 22,000 variables and 24,000 constraints. When we disregard these two results, average time drops to 77 s. Therefore, detection performance is competitive but additional optimization of our random sampling algorithm is necessary to face massive instances.

Experimental observation 2 *A full run of local search requires on average 50 min. When sampling is enabled, local search is completed in less than 3 min.*

On a negative note, Table 1 (T. LS) shows that while completing local search for DDW_{GCG} takes, on average, about 7 mins, DDW requires 50 min, in particular when facing medium or large size problems. This is due to the dimension of the starting

Table 1 Summary of pre-processing framework performance for each configuration

Conf.	S.	Times (s)		LS Neighborhood size		LS Termination				It
		T. Det	T. LS	Size (1st It)	Sampled	S.Cvx (%)	E.Cvx (%)	Sl	Ipr	
DDW GCG	E	1.55	724.41	26,679.20	0.00	72.94	78.67	5	6	25
	H	7.36	167.49	5645.43	0.00	85.22	90.67	0	3	95
	O	3.20	284.16	4752.00	0.00	88.82	93.00	0	1	63
Overall		3.80	454.91	15491.33	0.00	80.06	85.40	5	10	61
DDW	E	108.66	237.27	1,175,235.10	0.00	70.58	79.79	3	7	38
	H	1395.23	3422.59	2,125,484.10	0.00	50.68	58.03	2	7	43
	O	943.40	5100.19	6,813,521.78	0.00	55.05	55.55	0	6	30
Overall		811.36	2836.16	3,252,720.28	0.00	58.90	64.80	5	20	37
DDW sample	E	108.66	67.15	800.00	92,798.60	70.58	77.01	3	8	48
	H	1395.23	150.14	800.00	94,372.10	50.68	54.49	4	8	102
	O	943.40	270.55	888.89	102,768.44	55.05	57.47	4	7	98
Overall		811.36	158.89	827.59	96,435.28	58.90	63.18	11	23	82
DDW ortho	E	108.66	77.34	800.00	46,255.40	70.58	76.90	4	8	36
	H	1395.23	96.92	800.00	51,572.10	50.68	54.21	6	8	59
	O	943.40	279.97	888.89	49,462.44	55.05	56.50	4	7	46
Overall		811.36	146.98	827.59	49,084.03	58.90	62.75	14	23	47

Table 2 Time comparison between GCG configurations and three different runs of DDW_{sample}, when solving instances at the root node

	GCG	GCG_{Hmetis}	GCG_{Full}	DDW_{sample}		
				Run 1	Run 2	Run 3
Time (s)	905.17	416.92	847.79	344.08	347.41	420.92
Solver errors	1	1	0	6	5	5
Timeouts	14	13	14	6	5	7

neighborhood (Size), that on average is over 3 millions decompositions and, in the most extreme cases, over 10 millions. Therefore, although this setting is important for testing and validation, at this time, configurations that employ sampling are the ones that would be used in a real world scenario. Indeed, the overall number of decompositions sampled over all iterations (Sampled) by DDW_{sample} make about 2% of the size of the neighborhood generated during the first iteration of DDW. Therefore, neighborhood generation is much faster: the average time for local search is lower than 3 min. Furthermore, the number of iterations (It) is more than doubled even when using a 10 min timelimit.

We also report that, in general, local search could not be applied to every instance. Static detectors, in fact, are capable of finding decompositions much more convexified (S.Cvx) than data driven detectors. In this case, local search could be employed (Ipr) only for a third of the decompositions, the ones with a $S.Cvx$ lower than 85%. The other decompositions were simply moved to the optimization step.

Finally, we remark that, as an additional experiment, we studied the performance of our framework when facing instances that, from a statistical analysis, are "similar" to ones that are present in our training dataset. We reported this experiment in Table 10, in the Appendix. Although familiarity seems to have a positive effect on results, further investigations and tests on larger datasets are necessary to fully understand its impact.
Root node profiling In this part, we investigate performance when solving the root node, with a 2 h timelimit for optimization, over all the instances of Dataset B. In the following, we compare DDW_{sample} against GCG, GCG_{Hmetis} and GCG_{Full}. Since sampling might have an impact on the quality of the decomposition, we repeated DDW_{sample} experiments three times (run 1, run 2, run 3), to evaluate consistency and performance. For each algorithm, we report in Table 2 the number of solver errors, the number of timeouts and the average time (Time), in seconds, required for optimization of instances that did not run in timeout. Lower values are better. Full results can be found in Table 11 in the Appendix.

Experimental observation 3 *DDW_{sample} is on average faster than GCG configurations when solving the root node, while hitting about half the number of timeouts.*

We observe that DDW_{sample} is on average more than twice as fast than GCG and GCG_{Full} and performs slightly better than GCG_{Hmetis}, at the root node. Furthermore, GCG, GCG_{Hmetis} and GCG_{Full} reach respectively timeout in 47%, 43% and 47% of the experiments: only about 20% of the instances timed out when using our framework. However, as reported at the beginning of this section, we remark that we

Table 3 Comparison between three DDW_{sample} runs against SAS_{comm} and GCG configurations, at the root node. We report the number of the times each configurations obtains the best gap and the number of draws

Algo.	SAS_{comm}		GCG		GCG_{Hmetis}		GCG_{Full}	
$DWD\ sample$ run 1	**DDW Best**	**10**	DDW Best	6	DDW Best	6	DDW Best	6
	Draw	4	**Draw**	**15**	**Draw**	**14**	**Draw**	**14**
	SAS Best	9	GCG Best	3	GCG Best	4	GCG Best	4
$DWD\ sample$ run 2	**DDW Best**	**10**	DDW Best	7	DDW Best	7	DDW Best	7
	Draw	5	**Draw**	**15**	**Draw**	**14**	**Draw**	**14**
	SAS Best	9	GCG Best	3	GCG Best	4	GCG Best	4
$DWD\ sample$ run 3	**DDW Best**	**10**	DDW Best	8	DDW Best	7	DDW Best	8
	Draw	5	**Draw**	**14**	**Draw**	**13**	**Draw**	**13**
	SAS Best	9	GCG Best	3	GCG Best	5	GCG Best	4

faced undocumented solver errors in another 18% of the instances. As an approximate reference for SAS_{comm} optimization times, we report that the solver performed on average slightly faster than our detectors, while hitting 6 timeouts.

A comparison in terms of best bound found is harder to carry on: different solvers are more effective in different instances, in a comparable fashion. Full results for bound profiling are detailed in Table 12 in the Appendix. In an effort for producing a meaningful comparison, in Table 3 we report for the three runs of DDW_{sample}, GCG, GCG_{Hmetis}, GCG_{Full}. and SAS_{comm} the number of times our framework obtains the smaller gap from the best known solution (DDW Best), the number of solutions with the same gap (Draw) and the number of times static detectors obtain better gaps (SAS/GCG Best). In the following, results take into account only instances for which there were no computational issues for the algorithms considered in each comparison.

Experimental observation 4 *At the root node, our framework obtains competitive bounds with GCG configurations, performing equally or better in most cases.*

When compared to GCG, our framework finds competitive bounds at the root node. Indeed, in almost 50% of the instances the detectors obtain the same results. However, we note that data driven detectors found better bounds in about 23% of the remaining instances, while GCG was better in another 10%. Computation for the remaining experiments could not be completed. Performance against GCG_{Hmetis} and GCG_{Full} is similar, albeit these detector improve around 14% of the experiments. The remaining considerations still apply. Summarizing, we found no scenario in which data driven detectors obtained less decomposition that provide better results than static detectors, while providing faster optimization as well.

Experimental observation 5 *At the root node, our framework and SAS_{comm} improve two different sets of decompositions.*

When compared to SAS_{comm}, our framework obtained better results in about 33% of the instances, while SAS_{comm} was better in another 30%. Differently from GCG,

Table 4 Comparison between data driven framework configurations and GCG configurations. We report the number of the times each configurations obtains the best gap and the number of draws

Algorithm	GCG		GCG_{Full}	
DDW_{GCG}	DDW Best	4	DDW Best	2
	Draw	**14**	**Draw**	**12**
	GCG Best	2	GCG Best	5
DDW	**DDW Best**	**11**	**DDW Best**	**7**
	Draw	7	**Draw**	**7**
	GCG Best	3	GCG Best	5
DDW_{sample}	**DDW Best**	**10**	DDW Best	6
	Draw	7	**Draw**	**8**
	GCG Best	4	GCG Best	6
DDW_{ortho}	**DDW Best**	**9**	DDW Best	6
	Draw	7	**Draw**	**9**
	GCG Best	5	GCG Best	5

the number of ties is much lower, suggesting that indeed the two methods provide a very different set of decompositions. That is, data driven techniques could allow for better results than the community algorithm for a substantial set of instances.

Experimental observation 6 *Sampling shows consistent times and bounds over all the runs*

We also remark that, even when using sampling in multiple experiments, our data driven framework shows close results in terms of average time, number of timeouts and bound quality over all the runs. This can be observed in Tables 2 and 3: the variance among results from run 1, run 2 and run 3 is very small. That is, given an instance, all the runs obtain similar performance. This suggests that some instances may be more suited to a decomposition approach than others, that our method is reliable and that sampling is representative of the neighbourhood. Indeed, our techniques were designed to provide statistical guarantees through interval estimates. However, we remind that no assumptions were made over the "goodness" of the neighbourhood. We provide further insights and an interpretation of sampling behavior in the Appendix, in Section A.3.

Branch and bound profiling Finally, for this experiment, we chose a 5 h timelimit for optimization, without limiting the number of nodes of the branching tree.

In Table 4 we compare the gap from the best known solution obtained by the configurations of our framework (DDW_{GCG}, DDW, DDW_{sample}, DDW_{ortho}) against GCG and GCG_{Full}. Since GCG_{Hmetis} and GCG_{Full} performed similarly in previous experiments, only results for GCG_{Full} were reported for this test. For each comparison, we present the number of times our framework obtains the smaller gap (DDW Best), the number of solutions with the same gap (Draw) and the number of times static detectors obtain better results (GCG Best). More detailed results can be found in the Appendix, in Table 13.

All configurations hit the 5 h simulation timelimit on every instance, except for `gen.mps` that was solved to optimality.

Experimental observation 7 *Performance of DDW_{GCG} is similar to GCG.*

More in detail, we report that when DDW_{GCG} is used, improvements are present but strongly limited. Indeed, when starting from a static decomposition, our framework can only improve 13% of GCG decompositions, while most of the experiments have the same performance. Only 7% of the cases perform worse. When compared to GCG_{Full}, most of the instances are tied, but overall GCG_{Full} provides better bounds in more cases. We remark that better results could be potentially achieved by using the decomposition provided by GCG_{Full} as input for our local search methods. We assume however that the improvements would be limited with this setup as well.

In fact, we suspect that the limited gains are likely due to static detectors. As discussed, GCG provides decompositions with a well defined structure that, on average, is strongly convexified. Even when our local search can be used, we expect its impact to be strongly limited. This is also consistent with preliminary investigations [22].

Experimental observation 8 *DDW performs better than GCG configurations.*

Analyzing configurations that start from a data driven decompositions, we found that DDW obtains consistently better results when compared to GCG, getting better bounds in about 37% of the experiments. GCG can only obtain better performance in 10% of the tests. When compared with GCG_{Full}, results are close but our framework can still provide noticeable improvements and reaches the minimum gap more often than static detectors. However, we remark that, until further optimization, this configuration requires a time expensive pre-processing operation.

Experimental observation 9 *DDW_{sample} and DDW_{ortho} perform better than GCG. They are competitive with GCG_{Full}. The two methods do not dominate one another.*

Overall, when sampling is enabled, bound improvement is slightly reduced. However, the difference between using DDW and DDW_{sample} is rather small and acceptable: reducing the neighborhood size allows to employ our local search algorithm to large size instances without incurring into noticeable penalties. In fact, DDW_{sample} still performs better than GCG in 33% on the experiments, while static detectors obtain the better bound in 13% of the instances. However, in this case, performance is equal with GCG_{Full}. We remark that in this scenario the two methods do not dominate one another and using DDW_{sample} might indeed provide improvements for different sets of instances than GCG_{Full}. DDW_{ortho} obtains larger scores in some tests but overall, performance is similar. Summarizing, sampling retains most of the capabilities of DDW while making the algorithm much faster.

We also observe that DDW_{sample} obtained better bound improvements at the root node when compared to static detector configurations. We suspect that the timelimit and the generic nature of the solver might have an impact on branching performance. If this is the case, longer experiments might be required to fully investigate if the uplift found at the root node can translate in better solving times.

5 Discussion and prospectives

This paper is motivated by the huge interest around general purpose solvers. It aims at providing methods that overcome some issues, arising in prospective trends, like using them as a distributed service [7]. It is in fact not easy to make branch-and-cut scale in similar scenarios. In this perspective, we tackled the problem of understanding how to make use of alternative paradigms in a generic setting, since decomposition based ones promise strong scalability [38]. We focused on achieving automatic Dantzig–Wolfe decomposition methods. We proposed the first full data driven framework. That is, we designed and implemented an architecture that, ahead of time, trains machine learning models from optimization run logs. At optimization time, it receives as input a MIP instance and creates a decomposition pattern by employing algorithmic and data driven detectors that exploit the machine learning models. This candidate decomposition is further refined by local search before optimization. We presented an extensive experimental campaign, checking its behaviour on MIPLIB2017 instances which were not part of training. We compared it to state of the art algorithmic detectors. At the root node, we found our framework to reach the minimum gap more often than every other considered detector, along with faster average optimization times and less timeouts. When full optimization was considered, our DWD configuration was better than both GCG and GCG_{Full}, whilst using sampling provided better results than GCG and was comparable to GCG_{Full}. Even in these scenarios, our detectors can provide the best decomposition for instances that are not suited for specific static algorithms. This is also particularly evident in SAS_{comm} comparisons.

Our experimental campaign is not only instrumental in proving the computational effectiveness of our approach, but is also designed to provide some insights into the whole process of decomposition generation and improvement. For instance, we found out that computing a “degree of total unimodularity” of blocks, even by a rough estimate, has a strong impact in models. We also found that features like right hand side values and ratio between variables and constraints in each block are chosen as strongly predictive, along with the number of blocks. We finally mention that particular subsets of instances look to be more fit for our approach than for both GCG and SAS_{comm}: further investigating their features might allow some understanding on what might be still “missing” in current algorithmic detectors.

Prospectives. Overall, from a pure average computing time point of view, the performance of our framework alone is still far from state of the art solvers such as CPLEX. Indeed our system, and more in general alternative paradigms, are currently more promising when integrated side by side along with current solvers to optimize special classes of problems that are suitable to decomposition approaches. Nevertheless, state of the art solvers, have received decades of fine tuning and engineering. We feel that a technological update is required in our setting as well to guarantee the same stability and proper speed-ups.

Certainly, extending Integer Programming techniques for branching, generic heuristics and cuts to a decomposition setting would be strongly beneficial.

As introduced above, we expect a real breakthrough when branch-and-cut solvers will be compared to decomposition based ones on computing architectures which use many smaller computing units, that might become available dynamically, instead of a single powerful dedicated workstation. In fact, concurrent and distributed [38] techniques would fit perfectly, and might provide an additional (and orthogonal) speed-up in generic decomposition-based solvers.

Nonetheless, our efforts in obtaining automatic decompositions confirmed that a data driven approach is viable and performs in most settings better than state of the art algorithmic detectors such as those of GCG.

That leads us to the following key observation. In their very basic nature, the decompositions produced by data-driven methods can be seen as *heuristic solutions*, which could potentially be produced also by *algorithmic detectors*. Therefore, the improvement we get with respect to state-of-the-art detectors is more likely to be due to the use of different models, which are implicitly identified through our data-driven approach, than to the use of better optimization algorithms. Indeed, this is matching with our previous statistical evaluation [20]: the number of blocks and the size of the border, which are often the main indicators used in algorithmic detectors, are certainly important but not enough to fully capture decompositions quality.

In particular, according to our research, the next step in data driven detectors appears to be bound prediction. We feel that this might be a key to fully unfold automatic decomposition potential, along with the ability of using our models as *white box* to guide the search in new algorithmic detectors.

Given the effort it requires, we feel that the creation of our dataset of several thousands random decompositions of MIPLIB instances, together with their scoring, might be itself a valuable side-product of our research. We make it openly available to the research community [36], to ease other researchers in contributing along these directions.

In fact, we hope that our research might open new perspectives in the field of automatic decomposition, possibly enlarging the research focus from decomposition algorithms to decomposition *quality models*.

Acknowledgements The authors wish to thank three anonymous reviewers, a technical reviewer and the associate editor, whose insightful comments and suggestions helped to improve the paper. The work has been partially funded by University of Milan, Piano Sostegno alla Ricerca.

Funding Open access funding provided by Università degli Studi di Milano within the CRUI-CARE Agreement. the work has been partially funded by Università degli Studi di Milano, Piano Sostegno alla Ricerca, and partially undertook when S. Basso was at Dipartimento di Informatica, Università degli Studi di Milano.

Data availability statement All data analyzed during this study are publicly available. URLs are included in this published article. This manuscript has associated data in a data repository

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Code availability The full code was made available for review. We remark that a set of packages were used in this study, that were either open source or available for academic use. Specific references are included in this published article.

Appendix A

A.1 Framework tests: base instances of the Dataset

For each instance of our testing Dataset B, we present in Table 5 its name (Instance), a difficulty label (Status), a familiarity score with respect to our training dataset A (Fam.), the overall number of variables (Var.), the number of integer (Int.) and binary (Bin.) variables, the overall number of constraints (Constr.) and the percentage of non-zeroes in the constraint matrix (Nzs). We also report the best value of the objective (z*) found to date (17/08/2020).

Overall, we reported direct familiarity (1) for any instance that was labeled similar, from a statistical analysis, to one of the decompositions present in our training dataset. Otherwise, we reported indirect familiarity (2) or no familiarity (3). An instance that is indirectly familiar has no apparent correlation to our training dataset but shows similarities to other instances that, in our testing dataset, are reported as directly familiar.

Preliminary experiments

Preliminary experiments were performed on selected instances of Dataset B. We chose 15 instances divided in 10 easy problems and 5 hard problems. In the following, we present preliminary results and parameter tuning profiling. Experiments were always stopped at the root node with a simulation timeout of 2 h.
Heuristic features tuning Our datasets detail a total unimodularity feature (TU), a score based on how similar each block of a decomposition is with a total unimodular matrix. We compute this score with an heuristic algorithm, as detailed in [20] and we use it to train our models and to score new decompositions. Preliminary results suggested that its computation with standard settings ($k = 10$ runs of the heuristic) would be time expensive when exploring large neighborhoods. We therefore investigated its impact on time and dual bound by testing two configurations: either the heuristic is used (TU on) with a conservative one-shot approach ($k = 1$) or it is not used (TU off). For the former, we used our framework with standard settings. For the latter, we tested the instances with another framework that had identical settings but had been trained without taking into account the TU feature. Both cases were tested with DDW_{GCG}.

In Table 6 we report for every instance and every configuration the absolute local search time in seconds. We also present the relative performance (Rel. Perf.) as the ratio between TU off and TU on. Scores below 1 mean that a slowdown occurred,

Table 5 Instance statistics for every base optimization problem of Dataset B

Instance	Status	Fam.	Var	Int	Bin	Constr.	Nzs (%)	z*
beasleyC2	Easy	1	2500	0	1250	1750	0.11	144.00
berlin	Easy	2	5304	0	2652	2704	0.07	1044.00
bienst1	Easy	1	505	0	28	576	0.75	46.75
csched007	Easy	1	1758	0	1457	351	1.03	351.00
gen	Easy	3	870	6	144	780	0.38	112,313.36
mkc1	Easy	1	5325	0	3087	3411	0.09	– 607.21
newdano	Easy	1	505	0	56	576	0.75	65.67
piperout-d20	Easy	1	11,961	149	11,788	15,562	0.10	29,948.00
ran14x18-disj-8	Easy	1	504	0	252	447	4.56	3712.00
timtab1CUTS	Easy	1	397	94	77	371	1.18	764,772.00
bg512142	Hard	1	792	0	240	1307	0.38	184,202.75
dg012142	Hard	1	2080	0	640	6310	0.11	2,300,867.00
neos-2294525-abba	Hard	1	10,842	0	10,086	11,122	0.07	321.15
neos-4338804-snowy	Hard	2	1344	42	1260	1701	0.28	1471.00
ns1430538	Hard	2	33616	0	1680	34,960	0.02	88.00
queens-30	Hard	3	900	0	900	960	10.81	– 40.00
rococoC11-010100	Hard	1	12,321	166	12,155	4010	0.10	20,889.00
set3-20	Hard	3	4019	0	1424	3747	0.09	159,462.57
tbfp-bigm	Hard	1	2406	0	2404	35999	0.09	24.16
tw-myciel4	Hard	3	760	1	759	8146	0.45	10.00
cdc7-4-3-2	Open	3	11811	0	11811	14478	0.15	– 289.00
n370b	Open	1	10,000	0	5000	5150	0.04	1,236,963.00
nag	Open	3	2884	35	1350	5840	0.16	945.00
neos-1420790	Open	3	4926	0	540	2310	0.11	3121.42
neos-3009394-lami	Open	1	2757	52	2704	2028	0.12	5.50
ns1631475	Open	3	22,696	211	22,470	24,496	0.02	11,100.00
rococoC12-010001	Open	3	16,741	187	16,554	4636	0.08	34270.00
set3-09	Open	1	4019	0	1424	3747	0.09	176,497.15
siena1	Open	3	13,741	0	11,775	2220	0.85	10,359,207.14
van	Open	3	12,481	0	192	27,331	0.14	4.57

otherwise using the heuristic provides faster computing. Aggregate data shows the total completion time for each configuration and category and their relative performance.

Appendix: Experimental observation 1 *Computing the total unimodularity feature with a one-shot approach has negligible impact on local search time*

Table 6 Local search time comparison between TU off and on when using DDW_{GCG} (Root node)

Instance	Status	Time (s)		
		TU off	TU on	Rel. Perf.
beasleyC2	Easy	9.10	9.71	0.94
berlin	Easy	170.76	196.04	0.87
bienst1	Easy	3.49	9.13	0.38
csched007	Easy	47.07	55.37	0.85
gen	Easy	0.77	0.79	0.98
mkc1	Easy	805.88	853.26	0.94
newdano	Easy	2.67	4.44	0.60
ran14x18-disj-8	Easy	136.29	102.11	1.33
timetab1CUTS.mps	Easy	1.99	15.07	0.13
Overall		1178.03	1245.93	0.95
neos-4338804-snowy	Hard	13.68	14.34	0.95
ns1430538	Hard	5134.06	5193.11	0.99
queens-30	Hard	18.53	34.61	0.54
set3-20	Hard	1063.38	821.92	1.29
Overall		6229.64	6063.98	1.03
Overall		7407.67	7309.91	1.01

Using the heuristic with a one shot approach causes minor degradation on local search time for almost every instance. However, the overall impact on the total completion time is limited and very well manageable as we find slightly slower results on the easy category and faster ones on the hard category. This is within computational tolerance of two different runs.

In Table 7 instead, we report for every instance and for every configuration the absolute time in seconds required to solve the decomposition generated by our framework. We also present the relative bound improvement as the ratio between TU on and off, that is, scores above one mean that using TU is beneficial, whilst scores lower than 1 mean the opposite.

Appendix: Experimental observation 2 *Our total unimodularity feature is effective in improving the strength of the dual bound*

Results confirm that computing TU can improve bounds in specific instances. In particular, among easy problems, the dual bound of `bienst1` and `newdano` is doubled when using the heuristic. Otherwise, no relevant negative performance impact is measured. Solving time is generally comparable to when TU is not used except for `timetab1CUTS` that shows a considerable slowdown in exchange of a slightly better bound. Hard problems instead were not sensitive to any improvement or worsening of either bound or time. We suspect that these instances might simply not share any characteristic of totally unimodular ones and therefore our heuristic does not have any impact on them.

Table 7 Optimization comparison between TU off and on when using DDW_{GCG} (Root node)

Instance	Status	Time (s)		Relative bound improv.
		TU off	TU on	
beasleyC2	Easy	8.09	8.41	1.0000
berlin	Easy	75.05	71.00	1.0000
bienst1	Easy	10.59	5.59	2.4770
csched007	Easy	72.08	154.76	0.9890
gen	Easy	11.33	14.02	1.0000
mkc1	Easy	623.53	611.64	1.0000
newdano	Easy	2.58	1.99	2.1370
ran14x18-disj-8	Easy	5.11	5.67	1.0000
timetab1CUTS	Easy	2.36	496.58	1.0900
Overall		810.72	1369.66	1.2990
neos-4338804-snowy	Hard	7330.06	7329.39	1.0000
ns1430538	Hard	2128.53	2044.34	1.0000
queens-30	Hard	7324.97	7309.91	1.0000
set3-20	Hard	6259.35	6505.64	1.0000
Overall		23,042.91	231,89.28	1.0000
Overall		23,853.63	24,558.94	1.2140

We also notice that the instances that perform better in our full experimental analysis are the ones that show bigger bound improvements in this test. This may be correlated with the total unimodularity feature. However, further investigations, with a bigger dataset, are required to confirm this behavior.

Following the results of these tests, we decided to include our total unimodularity feature ($k = 1$) for the training of our models and for detection.

Selection strategies profiling In the following experiments we investigate the impact of different selection strategies for our local search algorithms on the candidate decomposition.

In [22], decomposition selection was based on the best score provided by our data driven time regressor but we know, from literature [17], that the number of blocks of a decomposition is also a suitable indicator of the time that is required to solve it. That is, solving decompositions with a small number of blocks is on average slower than solving ones with many. Therefore, in order to evaluate the relative effect of data driven choices with respect to facts from the literature, we tested our framework by integrating both configurations, along with others that combine the two methods, in the selection step of local search.

We remind that this operation is a critical part of the algorithm as choosing a suboptimal move may have a deep impact on the structure of the decomposition and its computing time, likely propagating its effects on all the subsequent iterations of local search.

In particular, we consider the following configurations, in which decompositions are chosen by:

T best time regressor score only.
B larger number of blocks only.
TB best time regressor score first, larger number of blocks second (in case of a tie).
BT larger number of blocks first, best time regressor score second (in case of a tie).

We tested these settings on both DDW_{GCG} and DDW. As a baseline for performance, we remark that a positive comparison against randomly chosen decomposition was presented for configuration T in [22].

In Table 8 we report the results for DDW_{GCG}. For every instance and selection strategy, we present the solving time in seconds. Since changing selection strategy has no noticeable overhead, local search time was disregarded for this experiment. We also report the bound improvement as the ratio between any of T, B, BT selections and TB. Scores below 1 mean that TB performs better, otherwise it performs worse.

Appendix: Experimental observation 3 *When starting from a static decomposition, the* BT *strategy works marginally better*

Results show that all configurations work quite well, but overall, combining selection strategies seems to be more effective. In particular, BT allows to obtain the best average bounds in the shortest overall time. However, we remark that the TB configuration is quite close both in terms of time and bound.

In Table 9 we present instead results for DDW. For every instance and configuration, we report the solving time in seconds and the bound improvement with respect to TB.

Appendix: Experimental observation 4 *When starting from a data driven decomposition, the* TB *setting performs better, also providing about a 10% improvement in bounds*

Similarly to the results of experiments with DDW_{GCG}, all the configurations yield decompositions quite close in terms of solving time. This is expected, as the number of blocks is a suitable proxy for solving time and our regressors heavily take into account this feature, as reported in Fig. 6. However, the TB strategy allows to obtain, on average, a 10% boost in dual bound. Even if our local search algorithm does not make any assumption on the quality of the bound of the candidate decomposition, we suspect that using the number of blocks as the only criterion for selection might have a negative effect and produce decompositions with suboptimal structures, in particular in our local search settings.

Overall, the TB configuration was the most consistent in all settings and was chosen for the full experimental analysis.

A.2 Familiarity profiling

A brief overview of the impact of familiarity is presented in Table 10 for DDW_{GCG}, DDW, DDW_{sample} and DDW_{ortho}. For each configuration, we report the average bound improvement as the ratio between the bound obtained by solving the decomposition provided by our framework and the one proposed by GCG. We confront direct familiarity (1) against the other scenarios. We chose a 5 h timelimit for optimization. We also report the results for DDW_{sample} at the root node with a 2 h timelimit.

Table 8 Selection strategy profiling (DDW_{GCG}, Root node)

Instance	Time (s)				Relative bound improv.		
	TB	BT	T	B	BT	T	B
beasleyC2	8.41	8.07	13.68	7.54	1.0000	1.0000	1.0000
berlin	71.00	70.53	69.74	70.46	1.0000	1.0000	1.0000
bienst1	5.59	1.74	2.87	6.98	1.0000	0.8514	0.5391
csched007	154.76	5.55	102.95	7211.12	0.9926	0.9978	0.9926
gen	14.02	14.12	10.64	15.26	1.0000	1.0000	1.0000
mkc1	611.64	612.82	607.98	617.31	1.0000	1.0000	1.0000
neos-4338804-snowy	7329.39	7329.39	7344.01	7329.35	1.0000	1.0000	1.0000
newdano	1.99	2.29	2.78	38.02	1.0946	0.8243	0.8919
queens-30	7309.91	7309.77	7345.46	7310.07	1.0000	1.0000	1.0000
ran14x18-disj-8	5.67	5.13	5.16	1344.07	1.0000	1.0000	1.0141
set3-20	6505.64	4881.87	7343.93	81.39	1.0000	1.0000	1.0000
timetab1CUTS	496.58	195.72	1058.83	429.63	0.9585	1.0017	1.0210
Overall	22,514.60	20,437.00	23,908.03	24,461.20	1.0038	0.9729	0.9549

Table 9 Selection strategy profiling (DDW, Root node)

Instance	Time (s)			Relative bound improv.		
	TB	BT	T	B	BT	T	B
beasleyC2	85.70	104.27	78.92	104.81	0.9519	0.9519	0.9519
bienst1	92.86	50.22	462.67	120.47	0.6185	0.4804	0.5766
csched007	177.99	133.42	161.24	7344.00	1.0000	1.0000	0.9873
dg012142	10.14	16.44	14.86	8.22	1.0000	1.0000	1.0000
gen	180.62	42.54	24.32	12.80	0.9999	1.0000	0.9999
neos-4338804-snowy	7200.00	7200.00	7200.00	1211.57	1.0000	1.0000	1.0083
newdano	248.35	141.82	710.93	148.54	0.7005	0.5022	0.9259
ran14x18-disj-8	2614.47	3918.05	1911.84	1193.73	0.9955	1.0049	0.9928
timtab1CUTS	1249.60	333.29	1265.31	997.93	0.9751	0.9871	1.0122
Overall	11,859.73	11,940.05	11,830.09	11,142.07	0.9157	0.8807	0.9394

Table 10 Relative bound improvement over direct familiarity (1) and otherwise, with respect to GCG

Fam.	DDW_{GCG}	DDW	DDW_{sample} (Root)	DDW_{sample}	DDW_{ortho}
1	1.1034	1.1307	1.2954	1.0843	1.1141
Otherwise	1.0000	1.0290	0.9966	1.0371	1.0264

A.3 Sampling analysis

In the following we consider sampling, providing an interpretation about its behavior and the consistency of its results. In particular, we run DDW_{sample} 3 times for each test and measured the bound of each simulation at the root node, with a timelimit of 2 h.

We report in Fig. 7, the difference (gap) between the best predicted score in the sample and the sample mean for each iteration of local search, for each instance. We present these values for each run, with a different color.

We can observe that in the majority of the instances the gap is greater than 0 and not constant, as it tends to grow along with the number of iterations. That is, even if the gap starts small, when the algorithm progresses and the decompositions gets more complex, the average neighborhood quality decreases and it is easier to find good candidates among many bad ones. In some instances, this behavior however seems to stop after a certain point, when the gap collapses. We expect that when many different constraints have been convexified, decompositions might get more and more homogeneous, as more blocks with shared variables tend to merge. After this happens, the gap tends to slowly build up again as diversification within the neighborhood increases with the number of iterations.

We remind that in general, our termination policy allows to avoid potential bad moves: when the next candidate decomposition has a predicted score that is much lower from the best one found during the current run, our algorithm terminates before making that particular bad move.

Overall, some insights about consistency are also visible: sampling often shows repeating patterns in predicted scores among all the different runs. It also suggests, however, that the starting decomposition has a very strong impact on the overall quality of the neighborhood.

A.4 Root Node results

We report in Table 11 the time, in seconds, required to solve all the instances at the root node, with a 2 h timelimit. We present results for three different runs of DDW_{sample} (run 1, run 2 and run 3) and the following static detectors configurations: GCG, GCG_{Hmetis} and GCG_{Full}.

Bounds obtained from this experiment are reported in Table 12 instead: for each instance, we present the gap, in percentage, from the best known solution. We show results for the three different runs of DDW_{sample}, SAS_{comm}, GCG, GCG_{Hmetis} and GCG_{Full}.

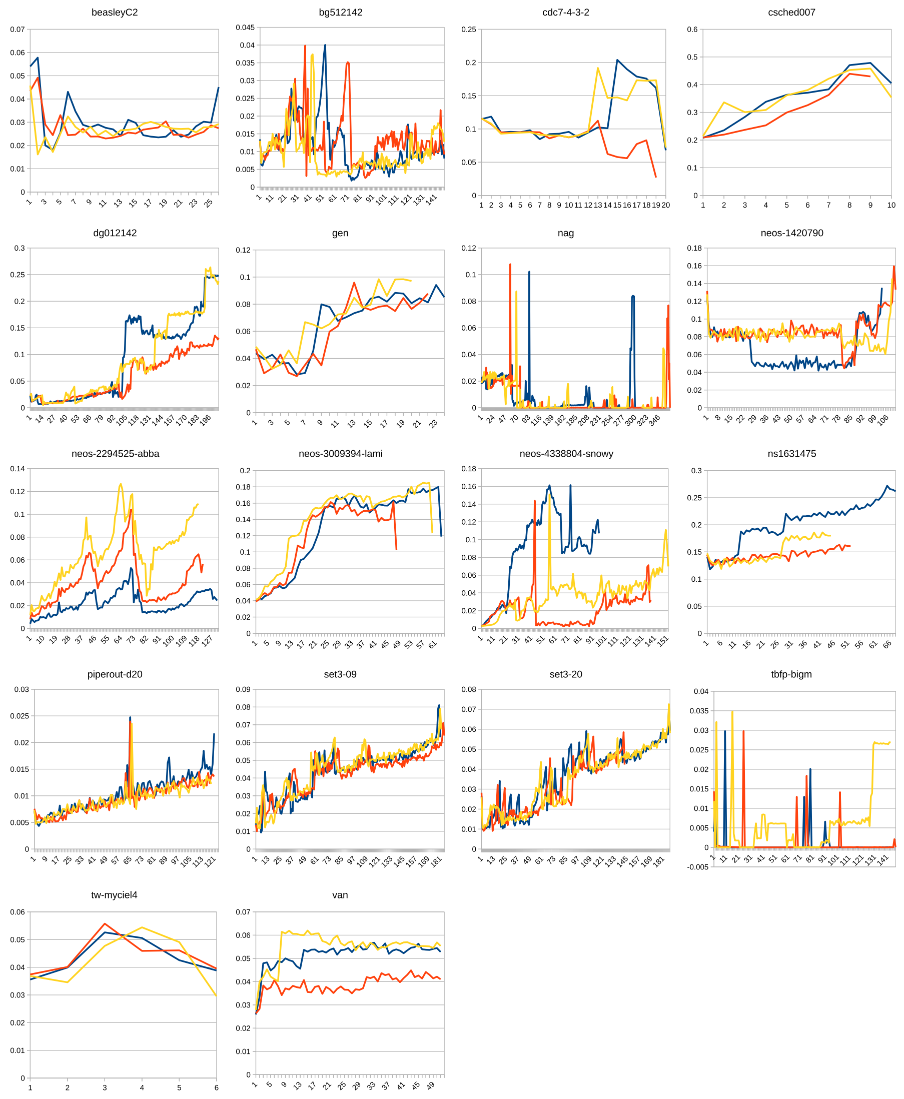


Fig. 7 Sampling behavior profiling

A.5 Full results

Our final comparison is reported in Table 13, in which we show the gap from the best known solution with no node limit. Timelimit for the optimization process is 5 h. We present results for each instance and the following algorithms: GCG, GCG_{Full}, DDW_{GCG}, DDW, DDW_{sample} and DDW_{ortho}.

Table 11 Time, in seconds, of GCG configurations and DDW_{sample} for solving instances at the root node. We consider three different runs for DDW_{sample}

Instances	GCG	GCG_{Hmetis}	GCG_{Full}	DDW_{sample}		
				Run 1	Run 2	Run 3
beasleyC2	8.41	13.74	8.52	85.88	70.29	82.70
berlin	34.01	34.10	33.35	84.88	70.07	70.68
bg512142	0.71	10.36	10.28	16.71	9.58	5.04
bienst1	3.70	2.00	1.93	316.08	81.43	29.13
cdc7-4-3-2	7238.48	7207.18	7256.03	7240.81	7200.91	7211.10
csched007	7.49	7344.06	7344.05	21.81	113.63	190.58
dg012142	7343.99	7343.99	7343.99	35.11	720.88	12.36
gen	5.90	1.47	1.34	29.95	19.78	14.46
mkc1	629.67	9.36	9.13	815.60	961.88	1062.04
n370b	16.68	101.30	16.66	116.88	101.57	103.93
nag	7343.98	5.97	6.05			
neos-1420790	7344.20	7344.07	7344.13	524.85	7344.80	7344.02
neos-2294525-abba	7343.97	7343.97	7343.97	7343.95	44.23	789.00
neos-3009394-lami	7344.22	7344.00	7344.08		31.56	33.02
neos-4338804-snowy	6887.48		6596.36			
newdano	3.81	2.36	2.34	236.90	707.09	427.80
ns1430538	6740.62	5238.69	6416.02	7342.03	7341.56	7341.30
ns1631475	77.12	1520.00	76.12	168.19	142.17	7344.07
piperout-d20	7339.23	7343.99	7343.99			
queens-30	7343.86	7354.05	7343.89	7333.50	7343.84	7343.84
ran14x18-disj-8	2.31	7.47	7.55	1543.49	2477.24	1939.97
rococoC11-010100	7343.99	7343.99	7343.98	132.08	88.73	75.75
rococoC12-010001	7344.01	7344.01	7344.01	29.32	29.87	29.45
set3-09	7343.87	40.99	40.18	511.35	322.04	7341.57
set3-20	62.37	153.76	153.03	7341.86	279.61	260.32
siena1	7343.80	135.94	7343.85			
tbfp-bigm		7344.01	7344.01			
timtab1CUTS	2.45	227.12	185.78	1510.60	662.82	2431.88
tw-myciel4	7344.00	7344.53	7344.17	13.71	13.70	18.48
van	7343.64	7343.65	7343.64	7303.75	7302.53	8026.67

Table 12 Gap from the best known solution, in percentage, of SAS, GCG configurations and DDW_{sample} when solving instances at the root node. We consider three different runs for DDW_{sample}

Instances	SAS	GCG	GCG	GCG	DDW_{sample}		
	$comm$		$Hmetis$	$Full$	Run 1	Run 2	Run 3
beasleyC2	43.58	58.40	31.10	58.40	61.30	61.49	61.11
berlin	87.54	87.54	87.54	87.54	87.54	87.54	87.54
bg512142	19.19	21.63	20.34	20.34	21.63	21.63	21.63
bienst1	74.92	74.92	57.34	57.34	47.51	13.37	32.62
cdc7-4-3-2	2353.98	3986.85	3986.85	3986.85	3986.85	3986.85	3986.85
csched007	22.87	23.29	23.29	23.29	23.29	22.12	22.47
dg012142	91.86	67.06	67.06	67.06	67.06	67.06	67.06
gen	0.00	0.08	0.08	0.08	0.04	0.04	0.02
mkc1	0.00	0.76	0.76	0.76	0.76	0.76	0.76
n370b	17.48	17.48	19.79	17.48	20.10	20.10	20.10
nag		50.79	50.79	50.79			
neos-1420790	7.65	5.72	5.72	5.72	5.72	5.72	5.72
neos-2294525-abba	51.37	51.37	51.37	51.37	51.37	51.37	51.37
neos-3009394-lami	58.44	88.31	88.31	88.31		84.42	87.01
neos-4338804-snowy	1.63	1.63		1.63			
newdano	82.15	82.15	69.63	69.63	38.32	51.27	43.65

Table 12 continued

Instances	SAS	GCG	GCG	GCG	DDW_{sample}		
	$comm$		$Hmetis$	$Full$	Run 1	Run 2	Run 3
ns1430538	13.03	13.02	13.02	13.02	16.34	16.34	16.34
ns1631475	100.00	92.63	92.63	92.63	92.63	92.63	92.63
piperout-d20		14.57	14.57	14.57			
queens-30	100.00	77.28	77.28	77.28	77.28	77.28	77.28
ran14x18-disj-8	6.90	7.21	6.87	6.87	4.53	3.70	4.31
rococoC11-010100	58.00	58.00	58.00	58.00	57.94	58.00	58.00
rococoC12-010001		21.83	21.83	21.83	21.83	21.83	21.68
set3-09	99.28	99.28	99.28	99.28	99.28	99.28	99.28
set3-20	93.51	93.51	93.51	93.51	93.51	93.51	93.51
siena1	1.76	1.89	1.89	1.89			
tbfp-bigm			837.29	837.29			
timtab1CUTS	18.16	19.27	12.05	12.58	11.75	11.87	12.36
tw-myciel4	60.37	60.00	60.00	60.00	60.00	60.00	60.00
van	27.96	62.32	62.32	62.32	62.32	62.32	62.32

Table 13 Bound profiling. We report for each configuration and each instance the percentage gap from the best known solution after 5 h of computation

Instances	GCG	GCG_{Full}	DDW_{GCG}	DDW	DDW_{sample}	DDW_{ortho}
beasleyC2	43.75	43.75	43.75	53.40	55.48	53.01
berlin	87.25	87.25	87.25	86.97	86.97	86.97
bg512142	21.60	19.54	20.72		21.63	21.63
bienst1	13.37	0.00				27.27
cdc7-4-3-2	3986.85	3986.85		3986.85	3986.85	3986.85
csched007	18.04	23.29	22.19	21.81	21.80	22.20
dg012142	67.06	67.06		67.06	67.06	67.06
gen	0.01	0.00	0.01	0.00	0.00	0.00
mkc1	0.76	0.76	0.76	0.76	0.76	0.76
n370b	17.20	17.20	17.23			
nag	50.79			50.79		
neos-1420790	5.72	5.72	5.72	5.72	5.72	5.72
neos-2294525-abba	51.37	51.37	51.37	51.22		
neos-3009394-lami	88.31	88.31				
neos-4338804-snowy	1.63	1.63	1.63			
newdano	73.13	17.71	43.65	52.15	52.15	38.32
ns1430538	13.02	13.02	13.02	16.34	16.34	16.34
ns1631475	92.63	92.63	92.63	92.63	92.63	92.63
piperout-d20	14.57	14.57	14.57			
queens-30	77.28	77.28	77.28	77.28	77.28	77.28
ran14x18-disj-8	5.97	4.73	5.84	4.95	4.95	4.73
rococoC11-010100	58.00	58.00		47.79	47.79	47.79
rococoC12-010001	21.83	21.83		20.13	20.13	20.13
set3-09	99.28	99.28	99.28	98.98	99.28	99.28
set3-20	93.51		93.51	93.10	93.25	93.29
siena1	1.89	1.89				
tbfp-bigm	837.29	837.29				
timtab1CUTS	18.36	12.58	11.97	13.20	13.20	11.83
tw-myciel4	60.00	60.00	60.00	50.00	50.00	50.00
van	62.32	62.32			58.12	

References

1. Achterberg, T., Wunderling, R.: Mixed integer programming: analyzing 12 years of progress. In: Facets of Combinatorial Optimization, pp. 449–481. Springer (2013)
2. IBM Cplex webpage: https://www.ibm.com/analytics/cplex-optimizer. Accessed November 2020
3. GUROBI webpage: http://www.gurobi.com. Accessed November 2020
4. FICO xpress webpage: http://www.fico.com/en/products/fico-xpress-optimization-suite. Accessed November 2020
5. Achterberg, T.: SCIP: solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009)

6. Vanderbeck, F., Wolsey, L.: Reformulation and decomposition of integer programs. In: Jünger, M., Liebling, Th.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958–2008. Springer, Berlin (2010)
7. SAS Viya webpage: https://www.sas.com/en_us/software/viya.html. Accessed March 2022
8. Basso, S., Ceselli, A.: Asynchronous column generation. In: Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 197–206. SIAM (2017)
9. Puchinger, J., Stuckey, P.J., Wallace, M.G., Brand, S.: Dantzig–Wolfe decomposition and branch-and-price solving in G12. Constraints **16**(1), 77–99 (2011)
10. Ralphs, T.K., Galati, M.V.: DIP—decomposition for integer programming. https://projects.coin-or.org/Dip. Accessed March 2017
11. Coluna framework: https://github.com/atoptima/Coluna.jl. Accessed February 2021
12. Frangioni, A., Lobato, R.D.: SMS++: a Structured Modelling System with Applications to Energy Optimization. In: PGMO DAYS 2018. https://smspp.gitlab.io/. Accessed November 2020
13. Gamrath, G., Lübbecke, M.E.; Experiments with a generic Dantzig–Wolfe decomposition for integer programs. In: Lecture Notes in Computer Science, vol. 6049, pp. 239–252 (2010)
14. Vanderbeck, F.: BaPCod—a generic branch-and-price code. https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod. Accessed March 2017
15. Frangioni, A., Sanchez, L.P.: Transforming mathematical models using declarative reformulation rules. In: Lecture Notes in Computer Science 6683, 5th Learning and Intelligent Optimization Conference, pp. 407–422 (2011)
16. Wang, J., Ralphs, T.: Computational experience with hypergraph-based methods for automatic decomposition in discrete optimization. In: Gomes, C., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. LNCS vol. 7874, pp. 394–402 (2013)
17. Bergner, M., Caprara, A., Ceselli, A., Furini, F., Lübbecke, M., Malaguti, E., Traversi, E.: Automatic Dantzig–Wolfe reformulation of mixed integer programs. Math. Program. A **149**(1–2), 391–424 (2015)
18. Bastubbe, M., Lübbecke, M.E.: A branch-and-price algorithm for capacitated hypergraph vertex separation. Math. Program. Comput. **12**(1), 39–68 (2020)
19. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010. Math. Program. Comput. **3**(2), 103–163 (2011)
20. Basso, S., Ceselli, A., Tettamanzi, A.: Random sampling and machine learning to understand good decompositions. Ann. Oper. Res. **284**, 501–526 (2018)
21. Basso, S., Ceselli, A.: Computational evaluation of ranking models in an automatic decomposition framework. In: Proceedings of EURO/ALIO 2018, Electronic Notes in Discrete Mathematics, Volume 69, pp. 245–252 (2018)
22. Basso, S., Ceselli, A.: Computational evaluation of data driven local search for MIP decompositions. In: Proceedings of ODS 2019, Advances in Optimization and Decision Science for Society, Services and Enterprises. AIRO Springer Series, vol. 3, pp. 207–217 (2019)
23. Kruber, M., Lübbecke, M.E., Parmentier, A.: Learning when to use a decomposition. In: Integration of AI and OR Techniques in Constraint Programming, Lecture Notes in Computer Science, vol. 10335, pp. 202–210. Springer (2017)
24. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'Horizon. Eur. J. Oper. Res. **290**(2), 405–421 (2021)
25. Prouvost, A., Dumouchelle, J., Scavuzzo, L., Gasse, M., Chételat, D., Lodi, A.: Ecole: a gym-like library for machine learning in combinatorial optimization solvers. arXiv:2011.06069 (2020)
26. Iommazzo, G., D'Ambrosio, C., Frangioni, A., Liberti, L.: A learning-based mathematical programming formulation for the automatic configuration of optimization solvers. In: Lecture Notes in Computer Science, 6th International Conference on Machine Learning, Optimization and Data science—LOD 2020 (2020)
27. Desaulniers, G., Desrosiers, J., Solomon, M.M. (eds.): Column Generation. Springer, Berlin (2005)
28. Lübbecke, M.E., Witt, J.T.: The strength of Dantzig-Wolfe reformulations for the stable set and related problems. Discrete Optim. **30**, 168–187 (2018)
29. Bastubbe, M., Lübbecke, M.E., Witt, J.T.: A computational investigation on the strength of Dantzig–Wolfe reformulations. In: 17th International Symposium on Experimental Algorithms (SEA 2018), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 103, pp. 11:1–11:12. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2018)

30. Iommazzo, G., D’Ambrosio, C., Frangioni, A., Liberti, L.: ’Learning to configure mathematical programming solvers by mathematical programming. In: Lecture Notes in Computer Science 12096. Learning and Intelligent Optimization—LION, vol. **2020**, pp. 377–389 (2020)
31. Fischetti, M., Lodi, A., Monaci, M., Salvagnin, D., Tramontani, A.: Improving branch-and-cut performance by random sampling. Math. Program. Comput. **8**, 113–132 (2016)
32. MIPLIB 2017: http://miplib.zib.de Accessed April 2019
33. Ross, S.: Simulation, 5th edn. Academic Press (2014)
34. Larose, D.T., Larose, C.D.: Data Mining and Predictive Analytics. Wiley (2015)
35. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett. **34**(4), 361–372 (2006)
36. Basso, S., Ceselli, A.: “MIPLib random decompositions dataset”. https://urldefense.com/v3/ https://doi.org/10.13130/RD_UNIMI/T99WYI UNIMI Dataverse V1 (2022)
37. Khaniyev, T., Elhedhli, S., Erenay, F.S.: Structure detection in mixed-integer programs. INFORMS J. Comput. **30**(3), 570–587 (2018)
38. Basso, S., Ceselli, A.: Distributed asynchronous column generation. Comput. Oper. Res. **146**, 105894 (2022)