Learning when to use a decomposition

Markus Kruber[1], Marco E. Lübbecke[1], and Axel Parmentier[2]

[1] Chair of Operations Research, RWTH Aachen University, Kackertstrasse 7, 52072 Aachen, Germany, {`kruber, luebbecke`}`@or.rwth-aachen.de`
[2] CERMICS, École des Ponts Paristech, Université Paris Est, 6 et 8 Avenue Blaise Pascal, 77420 Champs sur Marne, France, `axel.parmentier@enpc.fr`

Abstract. Applying a Dantzig-Wolfe decomposition to a mixed-integer program (MIP) aims at exploiting an embedded model structure and can lead to significantly stronger reformulations of the MIP. Recently, automating the process and embedding it in standard MIP solvers have been proposed, with the detection of a decomposable model structure as key element. If the detected structure reflects the (usually unknown) actual structure of the MIP well, the solver may be much faster on the reformulated model than on the original. Otherwise, the solver may completely fail. We propose a supervised learning approach to decide whether or not a reformulation should be applied, and which decomposition to choose when several are possible. Preliminary experiments with a MIP solver equipped with this knowledge show a significant performance improvement on structured instances, with little deterioration on others.

Keywords: mixed-integer programming; branch-and-price; column generation; automatic Dantzig-Wolfe decomposition; supervised learning

1 Setting and Approach

Dantzig-Wolfe (DW) reformulation of a mixed-integer program (MIP) became an indispensable tool in the computational mathematical programming bag of tricks. On the one hand, it may be *the* key to solving specially structured MIPs. On the other hand, successfully applying the technique may require a solid background, experience, and a non-negligible implementation effort.

In order to make DW reformulation more accessible also to non-specialists, general solvers were developed that make use of the method. One such solver is `GCG` [4], an extension to the well-established MIP solver `SCIP` [1]. Several *detectors* first look for possible DW reformulations of the original MIP model. Different types of reformulations are used by practitioners, and even if most MIPs can be forced into each of these types, their relevance highly depends on the model structure. For instance, staircase forms suit well to temporal knapsack problems [3] while bordered block diagonal forms work well on vehicle routing problems. As in general solvers, we do not know *a priori* the structure of the original model, and many *decompositions* of each type are detected. These decompositions are then evaluated: if they are of "good quality," the MIP is reformulated according to a "best suited" decomposition.

If one finds and selects a decomposition that captures a structure underlying the original model, the reformulated model may be solved much faster than the original one. However, from the many different decompositions, we may select one that does not reflect the actual underlying model structure; and in this case the solver may completely fail. We currently do not have any consistently reliable *a priori* measure to distinguish between these cases, except heuristic proxies, see e.g., [2]. What is more, the latter case is not uncommon, and GCG, presented with an arbitrary MIP, successfully detects a decomposition that leads to an improved performance over SCIP on the original model only in a small fraction of cases. This is to be expected, but will not render GCG (in its present form) a competitive general MIP solver. Our work thus aims at providing a mean to decide whether it pays to DW reformulate a given MIP model or not. Figure 1 illustrates how the result may look like.
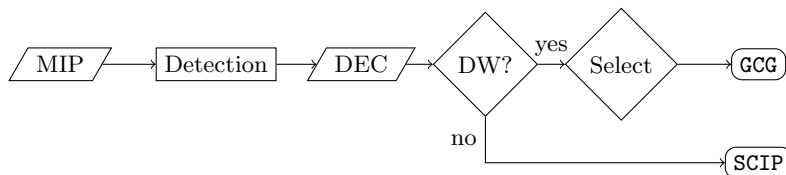
Fig. 1. Multiple detectors DW solver GCG with "SCIP exit strategy"

Literature. Decomposable model structure may be detected directly from the MIP, see e.g., [2, 12] and the references therein. Machine learning techniques have been recently used in computational mathematical optimization, e.g., automated MIP solver configuration [13], load balancing in parallel branch-and-bound [10], or variable selection in branching decisions [6, 9]. We are not aware of works that try to learn MIP model structure to be exploited in decompositions.

Our Supervised Learning Approach. We would like to learn an answer to the question: Given a MIP $\mathcal{P}$, a DW decomposition $\mathcal{D}$, and a time limit τ, will GCG using $\mathcal{D}$ optimally solve $\mathcal{P}$ faster than SCIP (or have a smaller optimality gap at time τ)? We define a mapping ϕ that transforms a tuple $(\mathcal{P}, \mathcal{D}, \tau)$ into a vector of sufficient statistics or *features* $\phi(\mathcal{P}, \mathcal{D}, \tau) \in \mathbb{R}^d$. Thanks to this mapping ϕ, the question above becomes a standard binary classification problem on $\mathbb{R}^d$. We can therefore train a standard classifier $f : \mathbb{R}^d \to \{0, 1\}$ to solve this problem. Given an instance $(\mathcal{P}, \mathcal{D}, \tau)$, the quantity $f \circ \phi(\mathcal{P}, \mathcal{D}, \tau)$ is equal to one iff the predicted answer to the question above is positive. Practically, we have built a database of SCIP and GCG runs for tuples $(\mathcal{P}, \mathcal{D}, \tau)$, a mapping ϕ, and trained classifiers f from the scikit-learn library [11] on the instances $\phi(\mathcal{P}, \mathcal{D}, \tau)$. Answers to the probabilistic versions $g : \mathbb{R}^d \to [0, 1]$ of these classifiers can be interpreted as the probability that GCG using $\mathcal{D}$ outperforms SCIP if the time limit is τ.

`GCG` starts by detecting decompositions $\mathcal{D}_1, \ldots, \mathcal{D}_k$ for $\mathcal{P}$. This detection takes time τ_{det}. We then decide how to make use of the remaining time $\tau - \tau_{\mathrm{det}}$:

$$\text{Continue } \texttt{GCG} \text{ if } \max_{i=1,\ldots,k} g \circ \phi(\mathcal{P}, \mathcal{D}_i, \tau - \tau_{\mathrm{det}}) \geq \alpha. \text{ Otherwise run } \texttt{SCIP}. \quad (1)$$

The threshold α reflects our level of conservatism towards solving $\mathcal{P}$ using a DW reformulation. If we decide to continue the run with `GCG`, we

$$\text{use decomposition } \mathcal{D} \text{ with maximum } g \circ \phi(\mathcal{P}, \mathcal{D}, \tau - \tau_{\mathrm{dec}}), \quad (2)$$

that is, one with largest predicted probability that `GCG` beats `SCIP`.

There are three key elements for such an approach to perform well: First, the features must catch relevant information, see Section 2.2. Second, for training a classifier we need to present it data, i.e., tuples $(\mathcal{P}, \mathcal{D}, \tau)$ in the learning phase that are similar to those we expect to see later when using the classifier. Our learning dataset contains instances from a wide range of families of structured and non-structured models. Third, an appropriate binary classifier must be used. It is our working hypothesis that a decomposition is likely to work if a similar decomposition works on a similar model. We thus tested classifiers whose answer depends only on the distance of the feature vector of the instance considered to those of the instances in the training set: nearest neighbors, support vector machines with an RBF kernel because such a kernel is stationary [5], and random forests because they can be seen as a weighted nearest neighbor scheme [8].

2 Decompositions and features

2.1 Bird's view on Dantzig-Wolfe reformulation

We would like to solve what we call the *original* MIP

$$\min \left\{ c^t x, \ Ax \geq b, \ x \in \mathbb{Z}_+^n \times \mathbb{Q}_+^q \right\}. \quad (3)$$

Today, the classical approach to solve (3) is branch-and-cut, implemented e.g., in the `SCIP` solver [1]. Without going into much detail, the data in (3) can sometimes be re-arranged such that a particular *structure* in the model becomes visible. Among several others, one such structure is the so-called *arrowhead* or double-bordered block diagonal form. It consists of partitioning the variables $x = [x^1, \ldots, x^\kappa]^t$ and right hand sides $b = [b^1, \ldots, b^\kappa, b^\ell]^t$ to obtain

$$
\begin{aligned}
&\min c^t x \\
&\text{s.t.} \begin{bmatrix} D^1 & & & & F^1 \\ & D^2 & & & F^2 \\ & & \ddots & & \vdots \\ & & & D^\kappa & F^\kappa \\ A^1 & A^2 & \cdots & A^\kappa & G \end{bmatrix} \cdot \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x^\kappa \\ x^\ell \end{bmatrix} \geq \begin{bmatrix} b^1 \\ b^2 \\ \vdots \\ b^\kappa \\ b^\ell \end{bmatrix} \quad (4) \\
&x \in \mathbb{Z}_+^n \times \mathbb{Q}_+^q.
\end{aligned}
$$

with sub-matrices of appropriate dimensions. Such a re-arrangement is called a *decomposition* of (3) and finding it is called *detection*. Matrices D^i are called *blocks*, variables x^ℓ are *linking variables* and $(A^1 \cdots A^\kappa\, G)x \geq b^\ell$ are *linking constraints*. Such forms are interesting because a Dantzig-Wolfe reformulation can be applied, leading to a solution of the resulting MIP by branch-and-price. Its characteristic is that the linear relaxations in each node of the search tree are solved by column generation, an alternation between solving *pricing problems*, i.e., integer programs over $D^i x^i \geq b^i$, and a linear program (the so-called *master problem*) involving exponentially many variables and constraints systematically derived from the linking constrains. See [4] for details (which are not necessary for the following). The algorithmic burden is considerable but may pay off if the pricing problems capture a well solvable sub-structure of the model (3). In such a case we call the decomposition *good*. When we know some good decomposition of a MIP model we call the model *structured*, if we do not know a good one the model is "non-structured." Automatic detection of decompositions, DW reformulation, and branch-and-price is implemented in the GCG solver [4].

2.2   Features considered

We now give an idea of the feature map ϕ that turns a MIP, decomposition, and time limit into a vector $\phi(\mathcal{P}, \mathcal{D}, \tau) \in \mathbb{R}^d$ that we give as input to supervised learning classifiers. We define a large number of features (more than 100) to catch as much information as possible, and then use a regularization approach to avoid overfitting. We only sketch the main types, without being exhaustive.

Our first features are *instance statistics*, like the number of variables, constraints, and non-zeros, the proportion of binary, integer, and continuous variables, or of certain types of constraints, such as knapsack or set covering. We collect *decomposition based statistics* like the number κ of blocks, or the proportion of non-zeros or variable/constraint types per block. As the dimension d of ϕ is fixed and κ varies across decompositions, we consider block statistics via their average, variance, and quantiles. A small number of linking constraints and variables is empirically considered good on "non-structured" MIPs [2].

Richer decomposition based features come from *adjacencies*, e.g., from the bipartite graph with a vertex for each row i of A and each column j of A, and an edge (i, j) iff $a_{ij} \neq 0$. Blocks can be seen as clusters of vertices in this graph. We can then build features inspired from graph clustering.

Many features can be obtained from the *detectors themselves*. The simplest is the *detector indicator feature* $\mathbb{1}_t(\mathcal{D})$, which is a binary feature equal to one iff decomposition $\mathcal{D}$ was found by detector t ("$\mathcal{D}$ is of type t"). Some detectors use metrics to evaluate the quality of their decomposition, for instance, whether blocks are "equal." These metrics can occur in ϕ. Some features mentioned above play different roles in different types of decompositions. Type specific behavior can be captured by products of detector indicators $\mathbb{1}_t$ with other features.

Finally, as the detection time varies from one instance to another, an important feature is the time remaining $\tau - \tau_{\text{dec}}$ after detection. Functions of this time feature can be used within products with other features.

3 Preliminary computational results

3.1 Experimental setup

As Dantzig-Wolfe decomposition leverages embedded structure in MIP instances, we want to learn from a wide range of model structures. At the same time, using a non-appropriate decomposition (i.e., assuming a "wrong" structure) almost certainly leads to poor solver performance. It is therefore most important to detect the "absence" of structure (or structure currently not exploitable by GCG). We have therefore built a dataset of 300 structured and 100 "non-structured" instances. We underline the fact that the information whether an instance contains structure or not is *not* part of the input features. We considered the following families of structured instances: vertex coloring (clr), set covering (stcv), capacitated p-median (cpmp), survivable fixed telecommunication network design (sdlb), cutting stock (ctst), generalized assignment (gap), network design (ntlb), lot sizing (ltsz), bin packing (bp), resource allocation (rap), stable set (stbl), and capacitated vehicle routing (cvrp) problems. The instances assumed "non-structured" are randomly chosen from MIPLIB 2010 [7]. GCG detected decompositions for each instance, leading to a total of 1619 decompositions. We launched SCIP and GCG with a time limit of two hours on each. Table 1 provides the number of instances per family and their difficulty, for which we use as proxy the solution status of SCIP after 2 hours. All experiments were performed on a i7-2600 3.4GHz PC, 8MB cache, and 16GB RAM, running OpenSuSE Linux 13.1. We used the binary classifiers of the `python scikit-learn` library [11], SCIP in version 3.2.1 [1], and GCG in version 2.1.1 [4].

	all	clr	stcv	cpmp	sdlb	ctst	gap	ntlb	ltsz	bp	rap	stbl	cvrp	miplib
instances	400	25	25	25	25	25	25	25	25	25	25	25	25	100
opt. sol.	65.5%	19	3	18	10	25	23	25	25	6	12	22	6	68
feas. sol.	31.5%	6	21	7	11	-	2	-	-	19	12	3	19	26
no sol.	3.0%	-	1	-	4	-	-	-	-	-	1	-	-	6

Table 1. Number of instances listed per problem class and solution status of SCIP

3.2 Overall computational summary of our experiment

We have randomly split our dataset into a training set containing 3/5 of the instances and a test set containing the remaining. Table 2 aggregates the overall results of our experiments when using a k-nearest neighbors (KNN) classifier. We report statistic for four solvers: the standard branch-and-cut MIP solver SCIP; GCG that tries to detect a decomposition and perform a DW reformulation accordingly; columns SL correspond to GCG with the learned classifiers (1) and (2) built in (i.e., the proposal of this paper); and finally, columns OPT list the results we would obtain if we always selected the best solver. The rows show, for

the sets of all, structured, and non-structured instances, the number of instances that could not be solved to optimality, the total time needed on the respective entire test set, and the geometric mean of the CPU times per instance.

Instances	All				Structured				Non-structured			
Solver	SCIP	GCG	SL	OPT	SCIP	GCG	SL	OPT	SCIP	GCG	SL	OPT
No opt. sol.	52	66	44	39	39	37	31	26	13	29	14	13
CPU time (h)	111.3	142.6	93.1	85.7	83.5	82.2	65.9	58.5	27.8	56.8	29.2	27.2
Geo. mean (s)	127.1	370.4	78.6	67.8	73.4	146.9	39.2	32.2	672.9	5145.0	766.0	646.5

Table 2. Aggregated statistics on a test set of 131 instances, 99 structured and 32 non-structured. Overall GCG achieves a better performance than SCIP in 34 cases.

The OPT columns show that there is potential in applying a DW reformulation on structured instances. However, we see that always using the current default GCG as a standalone solver is not an option. It needs to be complemented with an option to run SCIP in the case that a DW reformulation does not appear to be promising. This is what we do with our supervised learning approach. The resulting solver SL performs better than SCIP on structured instances with only little performance deterioration on "non-structured" instances. This trend should be confirmed on a larger test set. The performance of SL relies on the quality of the decisions taken by classifiers (1) and (2), that we now explain in detail.

3.3 Deciding whether to continue running GCG or to start SCIP

Table 3 shows the ability of classifier (1) to choose between SCIP and GCG given the decompositions provided by GCG's detectors and a time limit. The first row lists the share of instances where GCG and SCIP are respectively the best options. The 2×2 cells below give the *confusion matrix*, based on the prediction of the respective classifiers on the left side (rows) and on the true classes on the top (columns). The diagonal of each cell corresponds to cases where the classifier has predicted the right answer. The top right corner of each cell corresponds to false negatives (GCG is better, but we predict SCIP) and the bottom left to false positives (SCIP is better but we predict GCG). As GCG may perform very poorly when not a "right" structure is detected we try to keep the false positives low, even if this implies accepting more false negatives and thus not exploiting the full potential of GCG. The confidence threshold α in (1) was set to 0.5 here.

Even if the size of the test set is too small to arrive at final conclusions, we identify some trends. The RBF classifier shows a highly risk-averse behaviour with predicting GCG only in 7.6% of all cases. Only half of its predictions are correct, which is a poor performance. KNN and RF classifiers are willing to take more risk and predict that GCG is the best option in 20.6% to 29.0% of all instances. With a true predictions over all GCG predictions ratio of around 2/3, KNN shows the best precision. This explains that our SL scheme with KNN

		All instances		Structured		Non-structured	
		SCIP	GCG	SCIP	GCG	SCIP	GCG
Classifier	Pred.	74.0%	26.0%	68.7%	31.3%	90.6%	9.4%
RBF	SCIP	73.3%	19.1%	66.7%	23.2%	93.8%	6.3%
Unbal.	GCG	3.8%	3.8%	5.1%	5.1%	0.0%	0.0%
KNN	SCIP	69.5%	9.9%	64.6%	11.1%	84.4%	6.3%
distance.	GCG	6.9%	13.7%	7.1%	17.2%	6.3%	3.1%
RF	SCIP	63.4%	11.5%	55.6%	13.1%	87.5%	6.3%
Unbal.	GCG	10.7%	14.5%	13.1%	18.2%	3.1%	3.1%
RF	SCIP	60.3%	10.7%	50.5%	11.1%	90.6%	9.4%
Bal.	GCG	13.7%	15.3%	18.2%	20.2%	0.0%	0.0%

Table 3. Accuracy of solver selection, i.e., classifier (1)

catches roughly 2/3 of the improvement potential of of OPT with respect to SCIP in Table 2.

3.4 Selecting a best decomposition in GCG

Table 4 shows the percentage of instances on which classifier (2) predicts the right decomposition on the entire test set, on the subset of instances for which GCG is selected by classifier (1), and on the subset of instances such that GCG on the best decomposition actually outperforms SCIP. The classifier predicts the right answer only on about half of the instances. However, in practice, this classifier is called only if GCG has been selected by classifier (1). And on these instances, the right answer is predicted on around 80% of the cases. This difference of performance can be explained by the fact that there is no information on the relative performance of GCG on instances where SCIP performs better than GCG on all decompositions in the training dataset. A direction to include this information and improve the performance is to use a one-versus-one approach, where binary classifiers take two decompositions in input and predict which one is better, instead of using our "one-versus-SCIP" approach.

Classifier	All instances	GCG predicted by (1)	GCG better
RBF	42.7%	80.0%	76.7%
KNN	58.8%	88.9%	77.4%
RF unbalanced	51.1%	72.7%	76.5%
RF balanced	64.9%	71.1%	79.4%

Table 4. Accuracy of best decomposition selection, i.e., classifier (2)

4 Discussion and future work

It is our understanding that DW reformulation will be successful in terms of solver performance only on a small fraction of MIPs. It is therefore crucial to reliably decide whether a reformulation will pay off on a given input, *or not*. Several factors influence the success of the reformulation, among them the strength of the relaxation, an acceptable computational burden of repeatedly solving many subproblems, etc. An experienced modeler may have a sense for such factors, and this work aims equipping the GCG solver with a sort of such a sense. Our preliminary computational experience suggests that a supervised learning approach be promising. There are three immediate next steps.

1. We are currently completely re-designing the detection loop in GCG, leading to *many* more potential decompositions, and thus a much richer training set for our learning algorithms. 2. Detection can be very time consuming. One should predict *before* detection if the decomposition detected will be worth being used. This may save detection time in particular on "non-structured" instances where GCG is currently unable to find a good decomposition. 3. The quality of the features given as input to a machine learning classifier determines its performance. A direction to improve feature quality is to use run time, i.e., *a posteriori* information about a decomposition. One could run column generation on each decomposition for a limited amount of time in order to collect solution time and integrality gap of pricing problems, degeneracy of and dual bound provided by the master problem, etc. Figure 2 illustrates such a "strong detection" solution scheme. Reporting about this ongoing work is postponed to the full version of this short paper and will be implemented in a near future release of GCG.
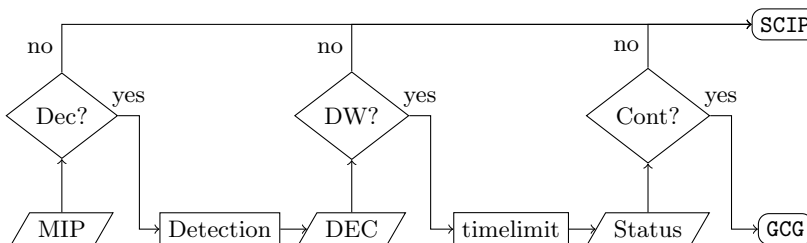

Fig. 2. Three decisions taken along a "strong detection" solution scheme

References

[1] Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation 1(1), 1–41 (2009)
[2] Bergner, M., Caprara, A., Ceselli, A., Furini, F., Lübbecke, M.E., Malaguti, E., Traversi, E.: Automatic Dantzig-Wolfe reformulation of mixed integer programs. Mathematical Programming 149(1-2), 391–424 (2015)

[3] Caprara, A., Furini, F., Malaguti, E.: Uncommon Dantzig-Wolfe reformulation for the temporal knapsack problem. INFORMS Journal on Computing 25(3), 560–571 (2013)

[4] Gamrath, G., Lübbecke, M.E.: Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In: Festa, P. (ed.) Experimental Algorithms. Lecture Notes in Computer Science, vol. 6049, pp. 239–252. Springer (2010)

[5] Genton, M.G.: Classes of kernels for machine learning: a statistics perspective. Journal of Machine Learning Research 2, 299–312 (2001)

[6] Khalil, E., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence (2016)

[7] 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. Mathematical Programming Computation 3(2), 103–163 (2011)

[8] Lin, Y., Jeon, Y.: Random forests and adaptive nearest neighbors. Journal of the American Statistical Association 101(474), 578–590 (2006)

[9] Marcos Alvarez, A., Louveaux, Q., Wehenkel, L.: A machine learning-based approximation of strong branching. INFORMS Journal on Computing

[10] Marcos Alvarez, A., Wehenkel, L., Louveaux, Q.: Machine learning to balance the load in parallel branch-and-bound (2015)

[11] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830 (2011)

[12] 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. Lecture Notes in Computer Science, vol. 7874, pp. 394–402. Springer (2013)

[13] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In: RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI) (Jul 2011)