Automatic Decomposition and Branch-and-Price—A Status Report

Marco E. Lübbecke

RWTH Aachen University, Operations Research, Kackertstraße 7,
D-52072 Aachen, Germany
marco.luebbecke@rwth-aachen.de

Abstract. We provide an overview of our recent efforts to automatize Dantzig-Wolfe reformulation and column generation/branch-and-price for structured, large-scale integer programs. We present the need for and the benefits from a generic implementation which does not need any user input or expert knowledge. A focus is on detecting structures in integer programs which are amenable to a Dantzig-Wolfe reformulation. We give computational results and discuss future research topics.

1 Modeling with Integer Programs

Integer programming offers undeniably a powerful and versatile, yet industrially relevant approach to model and solve discrete optimization problems from virtually all areas of scientific and practical applications. To get an impression on modeling, consider a simple combinatorial optimization problem, the *bin packing problem.* We are given n items of size a_i, $i = 1, \ldots, n$, which have to be packed into a minimum number of bins of capacity b each. A standard integer program for this problem is built on binary variables $x_{ij} \in \{0, 1\}$ to decide whether item i is packed in bin j or not. It is common that a *single* variable imposes relatively little structure on the overall solution. The model is as follows.

$$\min \sum_{j=1}^{n} y_j \tag{1a}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n \tag{1b}$$

$$\sum_{i=1}^{n} a_i x_{ij} \leq b \qquad j = 1, \ldots, n \tag{1c}$$

$$x_{ij} \leq y_j \qquad i, j = 1, \ldots, n \tag{1d}$$

$$x_{ij}, y_j \in \{0, 1\} \qquad i, j = 1, \ldots, n \tag{1e}$$

We call this the *original formulation.* Every item has to be packed because of the *set partitioning constraint* (1b); whenever a bin is used it has to be opened via the logical implication (1d); and no bin is overpacked because of the *knapsack constraint* (1c). The objective function (1a) reflects the goal of minimizing the number of opened bins. Note that n bins always suffice. A few remarks are in order. We observe that there is a symmetry w.r.t. the bins, that is, for a given solution $(\bar{x}, \bar{y})$ and a permutation σ of the bin indices, we get essentially "the same" solution with the same objective function value by replacing $\bar{x}_{ij}$ by $\bar{x}_{i\sigma(j)}$ and $\bar{y}_j$ by $\bar{y}_{\sigma(j)}$. Also note that there are rather "local" constraints (1c), namely those concerning the packing of a *single* bin; and there are "global" constraints (1b), namely those which ensure that *for every item* we open *some* bin. In particular, there is a knapsack problem to solve for each bin (which is NP-hard, but a computationally very easy combinatorial optimization problem), and the "individual" solutions to the knapsack problems are linked by a "coordinating" constraint which enforces a global structure in the overall solution. This is typical for many practical situations in which decisions are taken in a distributed way, but which in fact need synchronization in order to achieve a global goal (this is a feature which brings optimal solutions to such decision problems way out of reach of human planners). Examples are vehicle routing, crew and machine scheduling, location problems, and many more.

A standard solver does not "see" this concept of constructing a complex solution out of easier building blocks either, as a branch-and-bound algorithm works "everywhere" on the overall solution simultaneously by construction. One way to make these partial solutions "visible" to the solver is by formulating a different model which is based on "more meaningful" variables. For bin packing, we could base a model on binary variables $\lambda_{pj} \in \{0, 1\}$ which represent whether or not we pack an entire configuration or *pattern* p in bin j. A pattern is a collection of items that respects the knapsack capacity and therefore "knows" about the local constraints. All patterns for bin j are collected in a set P_j, and with the shorthand notation $i \in p$ to state that pattern p contains item i, the new model reads as follows.

$$\min \sum_{j=1}^{n} \sum_{p \in P_j} \lambda_{pj} \tag{2a}$$

$$\sum_{j=1}^{n} \sum_{p \in P_j : i \in p} \lambda_{pj} = 1 \qquad i = 1, \ldots, n \tag{2b}$$

$$\sum_{p \in P_j} \lambda_{pj} \leq 1 \qquad j = 1, \ldots, n \tag{2c}$$

$$\lambda_{pj} \in \{0, 1\} \qquad j = 1, \ldots, n,\ p \in P_j \tag{2d}$$

Constraint (2b) has the same role as constraint (1b) before: we must be sure that among all patterns for all bins, every item is contained in exactly one of those selected. The *convexity constraint* (2c) is new and ensures that at most one pattern is chosen per bin (a bin may also be "empty," that is, closed). No knapsack constraint is needed any more, at the expense of the fact that, for each bin, we essentially enumerated all feasible solutions to the knapsack constraint. After all, this is what we wanted. From a solution to this model we can uniquely reconstruct a solution to the original model (1) via $x_{ij} = \sum_{p\in P_j : i\in p} \lambda_{pj}$, that is, summing over all patterns for bin j that contain an item i. The new model (2) is still symmetric in the bins as all sets of patterns are identical for all bins. This symmetry could be eliminated by noting that we only need to ensure that for each item *some* pattern is selected; exactly which bin is used is not of importance. This leads us to an *aggregated* version of model (2).

$$\min \sum_{p\in P} \nu_p \tag{3a}$$

$$\sum_{p\in P : i\in p} \nu_p = 1 \qquad i = 1,\dots,n \tag{3b}$$

$$\nu_p \in \{0,1\} \qquad p \in P \tag{3c}$$

A binary variable ν_p represents whether we select a pattern p or not. Set P contains all feasible patterns, but no longer any information on bin indices. This is also true for the aggregated variables $\nu_p = \sum_{j=1}^{n} \lambda_{pj}$. As a consequence of the symmetry breaking, there is no unique reconstruction of an original solution to model (1) from a solution to model (3). Of course, one could have formulated a *set partitioning model* like (3) for the bin packing problem without going through the above reformulations, and this is what often happens.

2 Dantzig-Wolfe Reformulation

There is a major reason for favoring models (2) or (3) over model (1): the former is usually *stronger* in the sense that the linear relaxation, i.e., relaxing variable domains from $\{0,1\}$ to $[0,1]$, gives a tighter bound on the integer optimum. Intuitively, this is because the "more meaningful" variables in (3) impose more structure on the overall solution because a part of all original constraints (the "local" knapsack constraints) is already fulfilled with integrality. The theoretical reason is that model (2) is derived from (1) via a Dantzig-Wolfe reformulation.

A sketch of this reformulation is as follows (see e.g., [3] for details). Consider an *original* integer program of the form

$$\min\{c^t x : Ax \geq b,\ Dx \geq d,\ x \in \mathbb{Z}_+^n\}\ . \tag{4}$$

The polyhedron $X := \operatorname{conv}\{x \in \mathbb{Z}_+^n : Dx \geq d\}$ gives an inspiration for the "more meaningful" variables. We assume that X is bounded, but this is no restriction. We express $x \in X$ as a convex combination of the (finitely many) extreme points P of X, which leads to an equivalent *extended* formulation

$$\min\{c^t x : Ax \geq b,\ x = \sum_{p\in P} \lambda_p p,\ \sum_{p\in P} \lambda_p = 1,\ \lambda_p \geq 0,\ x \in \mathbb{Z}_+^n\}\ . \tag{5}$$

The reformulation (5) contains the so-called *master constraints* $Ax \geq b$, the convexity constraint, and the constraint linking the *original* x variables to the *extended* λ variables. In general, model (5) has an exponential number (in n) of λ variables, so its LP relaxation needs to be solved by column generation [3]. That is, one starts with a small subset of λ variables and iteratively adds more variables of negative reduced cost until no such variables can be identified. The pricing subproblem to check whether there exist variables with negative reduced cost is a minimization problem of a linear objective function over X, so it can be solved again as an integer program. The column generation process needs to be invoked in every node of the branch-and-bound tree, yielding a branch-and-price algorithm. Special care must be taken when deciding on how to branch on fractional variables [4,16,17].

In the classical setting, k disjoint sets of constraints are reformulated, namely when the matrix D has a *block-diagonal* form

$$D = \begin{pmatrix} D^1 & & & \\ & D^2 & & \\ & & \ddots & \\ & & & D^k \end{pmatrix} , \tag{6}$$

where $D^i \in \mathbb{Q}^{m_i \times n_i}$, $i = 1, \ldots, k$. In other words, $Dx \geq d$ partitions in $D^i x^i \geq d^i$, $i = 1, \ldots, k$, where $x = (x^1, x^2, \ldots, x^k)$, with an n_i-vector x^i, $i = 1, \ldots, k$. Every $D^i x^i \geq d^i$ is individually Dantzig-Wolfe reformulated. We call k the *number of blocks* of the reformulation. A matrix of the form

$$\begin{pmatrix} D^1 & & & \\ & D^2 & & \\ & & \ddots & \\ & & & D^k \\ A^1 & A^2 & \cdots & A^k \end{pmatrix} \tag{7}$$

with $A^i \in \mathbb{Q}^{m_\ell \times n_i}$, $i = 1, \ldots, k$ is called *bordered block-diagonal*, see Fig. 1(b). It is this form we would like to see in the coefficient matrix of an integer program in order to apply a Dantzig-Wolfe reformulation.

Depending on your background, the following may or may not apply to you: you regularly formulate models like (1) (but cannot solve even moderately sized instances); you have already noticed that going from model (1) to model (2) is by application of a Dantzig-Wolfe reformulation (but you don't know of what use this knowledge may be to you, practically speaking); you would like to implement your own column generation and branch-and-price code and use it to optimally solve models like (2) or (3) (but you don't know whether it is worth the time and considerable effort); you already have your branch-and-price code running (but don't want to change and adapt it every time you consider a new problem). You are any of this kind? Read on...

3 In Need for an Automatic Decomposition

In 2004, when collecting material for "the primer" [3], we learned about François Vanderbeck's efforts to write a generic branch-and-price code BaPCod [12], and we were fascinated by the idea ever since. Vanderbeck developed important contributions [11,13,14,15], so that not only a Dantzig-Wolfe reformulation could be performed according to a user specification but also branching was done in a generic way. Further generic decomposition codes became available like DIP [10] and within the G12 project [9]. However, none of these codes could be used by someone not knowledgeable in decomposition techniques as the user needs to propose how the input integer program is to be decomposed. It is still up to the modeler which constraints $Dx \geq d$ she considers as "local," that is, to be reformulated. Sometimes the choice may be rather obvious as in our introductory example, but sometimes there is more freedom, and thus more freedom to make mistakes, and one needs to know what one is doing.

It simply felt wrong that there are the well-understood and in many special cases successfully applied concepts of Dantzig-Wolfe decomposition, column generation, and branch-and-price, but they are not "out-of-the-box" usable e.g., to "everyday" OR practitioners, despite the availability of generic implementations. And yet those who ran their own codes often needed to start all over with every new application.

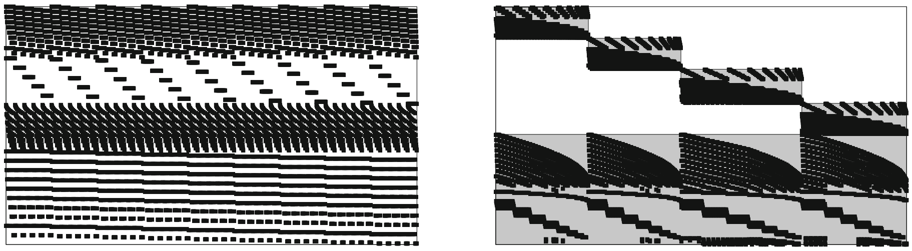
In order to close the last—but maybe most crucial—gap, we made first experiments in [2] with detecting matrix structures suited for reformulation (see Section 4). At that time, this detection and our generic branch-and-price code GCG (see Section 5) were not merged into one project, but results were encouraging. Testing on general mixed integer programs was very nice and gave a successful proof-of-concept, but diverted our attention from the true target instances of this research: those which bear structure. In this talk, we report on experiments with automatic detection of decomposable matrix structures, and generic branch-and-price on a suitable test set of "structured" instances.

4 Recognizing Matrix Structures for Decomposition

The typical matrix structure for which Dantzig-Wolfe decomposition was proposed is a bordered block-diagonal form (7). With a little experience with the technique and an automatic recognition of types of constraints (like set partitioning constraints, knapsack constraints, and the like) one can come up quite easily with a suggestion for a decomposition. This works well for standard problems, but may fail for "unknown" problems. Then, a possibility is to exploit a folklore connection between matrices and graphs [5].

Given a matrix A, construct a hypergraph $H = (V, R \cup C)$ as follows. With every $a_{ij} \neq 0$ associate a vertex $v_{ij} \in V$. For every row i introduce a hyperedge $r_i \in R$ which contains exactly all vertices $v_{r_i j} \in V$ that correspond to non-zero entries of the row; analogously introduce a hyperedge $c_j \in C$ for every column j. When H partitions into several connected components, the matrix A is a block-diagonal matrix, with a bijection between blocks in A and connected components in H.

Hypergraph H can also be used to detect a bordered block-diagonal form. Without the rows in the "border" the remaining matrix is block-diagonal. Thus, a removal of (a minimum number of) hyperedges from R such that the remaining graph partitions into connected components reveals a bordered block-diagonal form in A (with a minimum number of rows in the border). The problem is NP-hard and we experimented with heuristics to solve it. Figure 1 shows a matrix as given in the original model, and a structure detected with this *minimum hypergraph partitioning approach*.



(a) original *10teams* (b) detected structure

Fig. 1. (a) Matrix structure directly from the LP file (*10teams*) and (b) with a bordered block-diagonal structure detected by our algorithm

The algorithm needs as input the number k of connected components we look for in H. Thus, in practice, we check for different small numbers of k. As the results sometimes look artificial, we suggest that a more tailored hypergraph partitioning algorithm should be sought, exploiting the fact that H is extremely sparse and of degree 2. In a different line of research we replace hypergraph partitioning by hypergraph clustering, which eliminates the need to specify the number of blocks (connected components, clusters) beforehand. Experimentation with these alternatives is still under way, but preliminary results look plausible.

5 Towards a Standalone Solver

Based on the discussion above, we developed `GCG` [6] ("generic column generation") which is based on the `SCIP` framework [1] which is free for academic purposes (`scip.zib.de`). Together with the LP file describing the integer program, `GCG` takes as input a second file ("the decomposition") describing which constraints belong to the master problem and which to the blocks, respectively. Alternatively, several of the matrix structure detection algorithms only sketched above are applied to the instance. `GCG` then performs a Dantzig-Wolfe reformulation according to a "best guess" (in addition one can specify whether the so-called convexification or discretization approach should be applied); identifies identical blocks, and aggregates them (the same as going from model (2) to model (3)); performs column generation on the given decomposition; maintains both formulations (the *original* in the x-variables, and the *extended* in the λ-variables) which allows branching and cutting plane separation on the original variables; it has specialized branching rules like Ryan/Foster branching and generic primal heuristics [7,8]; and overall `GCG` uses `SCIP`'s rich functionality of being a state-of-the-art MIP solver (like availability of pseudo-costs, pre-solving, propagation techniques, etc.). This turns the branch-price-and-cut *framework* `SCIP` into a branch-price-and-cut *solver* [6]. A first stable version is about to be released as this abstract goes to press.

6 Discussion

What are the goals of our project? Certainly, expecting a decomposition code to beat a state-of-the-art branch-and-cut code on the average instance is not realistic. On the other hand, anything but outperforming the general-purpose solver on instances that contain a decomposable problem structure would be a failure. Thus, the art remains to tell the instances that are amenable to a Dantzig-Wolfe reformulation from those which are not. And so we are back at the most important and most interesting algorithmic challenge: to efficiently and reliably detect "structure" in an instance or conclude that "none" is contained.

This is an area which may not only produce new and improved algorithms, e.g., for partitioning/clustering the graph underlying a coefficient matrix. We also need a much better theoretical understanding of what makes a good "structure" to look for, and we believe that this will give us insights into how to set up a good integer programming model in the first place. It is this *algorithm engineering* feature which makes this project so interesting to us: to improve the design of a long-known algorithm, letting computational experiments guide our way. We hope that our work contributes to closing the gap between the algorithm on paper and its usefulness to a non-expert in practice.

Acknowledgments. I enjoyed discussions with Michael Bastubbe, Martin Bergner, Alberto Ceselli, Jacques Desrosiers, Fabio Furini, Gerald Gamrath, Marc Pfetsch, and Emiliano Traversi on automated decompositions. This project was supported by the German Science Foundation's (DFG) priority program 1307 "algorithm engineering," under contracts LU 770/4-1 and LU 770/4-2, within the project "generic decomposition algorithms for integer programs."

References

1. Achterberg, T.: SCIP: Solving constraint integer programs. Math. Programming Computation 1(1), 1–41 (2009)
2. Bergner, M., Caprara, A., Furini, F., Lübbecke, M.E., Malaguti, E., Traversi, E.: Partial Convexification of General MIPs by Dantzig-Wolfe Reformulation. In: Günlük, O., Woeginger, G.J. (eds.) IPCO 2011. LNCS, vol. 6655, pp. 39–51. Springer, Heidelberg (2011)

3. Desrosiers, J., Lübbecke, M.: A primer in column generation. In: Desaulniers, G., Desrosiers, J., Solomon, M. (eds.) Column Generation, pp. 1–32. Springer, Heidelberg (2005)
4. Desrosiers, J., Lübbecke, M.: Branch-price-and-cut algorithms. In: Cochran, J. (ed.) Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Chichester (2011)
5. Ferris, M., Horn, J.: Partitioning mathematical programs for parallel solution. Math. Programming 80, 35–61 (1998)
6. Gamrath, G., Lübbecke, M.E.: Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 239–252. Springer, Heidelberg (2010)
7. Joncour, C., Michel, S., Sadykov, R., Sverdlov, D., Vanderbeck, F.: Column generation based primal heuristics. In: International Conference on Combinatorial Optimization (ISCO). Electronic Notes in Discrete Mathematics, vol. 36, pp. 695–702. Elsevier (2012)
8. Lübbecke, M., Puchert, C.: Primal heuristics for branch-and-price algorithms. In: Operations Research Proceedings 2011. Springer (to appear, 2012)
9. Puchinger, J., Stuckey, P., Wallace, M., Brand, S.: Dantzig-Wolfe decomposition and branch-and-price solving in G12. Constraints 16(1), 77–99 (2011)
10. Ralphs, T., Galati, M.: DIP – decomposition for integer programming (2009), `https://projects.coin-or.org/Dip`
11. Vanderbeck, F.: On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. Oper. Res. 48(1), 111–128 (2000)
12. Vanderbeck, F.: BaPCod – a generic branch-and-price code (2005), `https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod`
13. Vanderbeck, F.: Implementing mixed integer column generation. In: Desaulniers, G., Desrosiers, J., Solomon, M. (eds.) Column Generation, pp. 331–358. Springer (2005)
14. Vanderbeck, F.: A generic view of Dantzig-Wolfe decomposition in mixed integer programming. Oper. Res. Lett. 34(3), 296–306 (2006)
15. Vanderbeck, F.: Branching in branch-and-price: A generic scheme. Math. Programming 130(2), 249–294 (2011)
16. Vanderbeck, F., Wolsey, L.: Reformulation and decomposition of integer programs. In: Jünger, M., Liebling, T., Naddef, D., Nemhauser, G., Pulleyblank, W., Reinelt, G., Rinaldi, G., Wolsey, L. (eds.) 50 Years of Integer Programming 1958–2008. Springer, Berlin (2010)
17. Villeneuve, D., Desrosiers, J., Lübbecke, M., Soumis, F.: On compact formulations for integer programs solved by column generation. Ann. Oper. Res. 139(1), 375–388 (2005)