Exploiting user-supplied Decompositions inside Heuristics

Katrin Halbig *, Adrian Göß , Dieter Weninger

Friedrich-Alexander Universität Erlangen-Nürnberg, Department of Data Science, Cauerstr. 11, 91058 Erlangen, Germany

June, 2023

Abstract Many mixed-integer models are sparse and can, therefore, usually be decomposed into weakly connected blocks. Such decompositions could be determined algorithmically or be specified by the user. We limit ourselves to the later, as the user usually has a very precise idea of which decomposition makes sense for structural reasons. In the present work, we address the exploitation of user-supplied decompositions within the non-commercial solver SCIP to control heuristics. In order to demonstrate the potential, three different heuristics leveraging such decomposition information are considered. Our results show that such an approach has a positive influence on the overall solution behavior of SCIP, provided that the decomposition information supplied describes the basic structural properties of the model appropriately for the particular heuristic.

Keywords Mixed-integer programming · Heuristic · Decomposition · Optimization solver · Supply chain management

Mathematics Subject Classification 90C11 · 90C59 · 90C90 · 90B06

1 Introduction

Decomposition methods have been used for solving linear and mixed-integer linear problems for over half a century. The publications of Dantzig and Wolfe [18] and of Benders [6] can be identified as starting points of the entire research area. In addition, Lagrangian relaxation [27] and its special case Lagrangian decomposition [30] have played an important role for many years. In [44] a survey on different decomposition methods is given. Detailed descriptions of the most important decomposition algorithms with applications for mixed-integer programming are given in [39, 46, 48].

In most practical applications encoded as a mixed-integer problem, the constraint matrix is typically very sparse, i.e., most of its entries are 0. In MIPLIB 2017 [28], which comprises 1065 instances arising from more than 400 distinct applications of mixed-integer problems, the fraction of nonzero entries, the so-called density, ranges from 10^{-7} to 1 (100 %), at an average of less than 3 % (see [50] for more details). Additionally, mixed-integer supply chain models are typically very sparse [43]. Sparse problems can usually be decomposed into loosely coupled blocks [13], which opens the possibility for decomposition methods.

E-mail addresses: {katrin.halbig,adrian.goess,dieter.weninger}@fau.de
*Corresponding author

In addition to the exploitation of sparsity for decomposition, it is beneficial to leverage other problem structures. In [20] the authors describe an efficient method for Benders' decomposition that exploits the structure of the capacitated facility location problem. Based on this, Benders' decomposition approaches allowing integer variables in the subproblem were presented in [47] for the special case of single-source capacitated facility location. For the solution of the symmetric traveling salesman problem, approaches combining Lagrangian relaxation with the structural utilization of 1-trees have been established [34]. If the problem has a diagonal block structure, this can sometimes be exploited in a Dantzig-Wolfe decomposition, as it is, for example, the case for the generalized assignment problem [16].

Meanwhile some software packages offer the possibility to quickly set up decomposition algorithms. Commercial solvers such as CPLEX [17] and non-commercial solvers such as SCIP [11] provide interfaces to communicate decomposition information, mainly used for Benders' decomposition. The solver GUROBI [31] can perform a solution improvement heuristic using user-provided partition information on the variables. All approaches above have in common that the users have to define the structure themselves. GCG [7, 21] performs a Dantzig-Wolfe decomposition of the problem, where the decomposition is based on a structure either provided by the users or automatically detected. SAS [42] also calculates a decomposition itself and solves the resulting problem using a Dantzig-Wolfe approach.

One important part of successful algorithms for the solution of mixed-integer linear problems is finding feasible solutions quickly. To this end, usually heuristics are employed to support the branch-and-cut procedure [9]. Heuristics are subdivided into two categories: *construction heuristics* which attempt to determine a feasible solution from scratch, and *improvement heuristics* which try to improve a given feasible solution.

In the present work we want to deal with the extent to which the provision of a so called *decomposition information* or for short *decomposition* by the user can be used profitably in the solution process. Such an approach has two advantages: First, the time-consuming computation of a decomposition information is omitted and, second, the user typically has a very precise idea of the underlying structure behind the problem and can communicate it on the basis of a decomposition appropriately to a particular heuristic. We focus on exploiting decompositions for the control of heuristics in SCIP [11, 12].

Our contributions in this publication consist of the presentation of three heuristics, which exploit decomposition information to solve mixed-integer linear problems, the provision of high-performance open-source C implementations, the traceability of part of the computational results through the availability of test instances and decompositions, and results from a tight integration with the non-commercial solver SCIP.

We begin, in the next section, with an introduction to the definitions and notation used in the remainder of the paper. Three sections follow, each with a heuristic exploiting decomposition information, which constitute the main part of this paper. Initially we describe two construction heuristics, DPS and PADM, in Section 3 and Section 4 that solve subproblems, which are directly derived from a given decomposition, on an alternating basis. As opposed to DPS, which works with linking constraints only, PADM requires a decomposition with linking variables only. Finally, we describe in Section 5 an extension of the kernel search framework that can be used both as construction and improvement heuristic. Numerical results on all three heuristics are presented and discussed in Section 6. A brief summary of the results and a short discussion of open research questions for future work in Section 7 conclude the paper.

2 Definitions and Notation of Decompositions

Mixed-Integer Programs (MIP) are commonly formulated as optimization problem

$$\min \left\{ c^\top x \,:\, Ax \geq b,\, \ell \leq x \leq u,\, x_j \in \mathbb{Z} \text{ for } j \in \mathcal{I} \right\} \tag{1}$$

with variables $x \in \mathbb{R}^n$ whose values are further restricted by lower and upper bounds $\ell, u \in (\mathbb{R} \cup \{\pm\infty\})^n$, linear constraints formulated as system of inequalities using a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$, and integrality restrictions for all variables x_j with $j \in \mathcal{I} \subseteq \{1, \ldots, n\}$, $\mathcal{I} \neq \emptyset$. The elements of A are denoted by a_{ij} with $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$.

For a number $k \geq 0$ we call a partition $\mathcal{D} := (D^{\text{row}}, D^{\text{col}})$ of the rows and columns of A into $k+1$ pieces each,

$$D^{\text{row}} := (D_1^{\text{row}}, \ldots, D_k^{\text{row}}, L^{\text{row}}), \quad D^{\text{col}} := (D_1^{\text{col}}, \ldots, D_k^{\text{col}}, L^{\text{col}}),$$

a *decomposition* of A if $D_q^{\text{row}} \neq \emptyset$, $D_q^{\text{col}} \neq \emptyset$ for $q \in [k] := \{1, \ldots, k\}$ and if it holds for all $i \in D_{q_1}^{\text{row}}$, $j \in D_{q_2}^{\text{col}}$ that $a_{ij} \neq 0$ implies $q_1 = q_2$. We call k the number of *blocks* and each block $q \in [k]$ is specified by $(D_q^{\text{row}}, D_q^{\text{col}})$. The special rows L^{row} and columns L^{col}, which may be empty, are called *linking rows* and *linking columns*, respectively. Equivalent terms are *linking constraints* and *linking variables*, which can also be used in the more general context of nonlinear problems.

In other words, the inequality system $Ax \geq b$ can be rewritten with respect to a decomposition $\mathcal{D}$ by a suitable permutation of the rows and columns of A as equivalent system

$$\begin{pmatrix} A_{[D_1^{\text{row}}, D_1^{\text{col}}]} & 0 & \cdots & 0 & A_{[D_1^{\text{row}}, L^{\text{col}}]} \\ 0 & A_{[D_2^{\text{row}}, D_2^{\text{col}}]} & 0 & 0 & A_{[D_2^{\text{row}}, L^{\text{col}}]} \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & A_{[D_k^{\text{row}}, D_k^{\text{col}}]} & A_{[D_k^{\text{row}}, L^{\text{col}}]} \\ A_{[L^{\text{row}}, D_1^{\text{col}}]} & A_{[L^{\text{row}}, D_2^{\text{col}}]} & \cdots & A_{[L^{\text{row}}, D_k^{\text{col}}]} & A_{[L^{\text{row}}, L^{\text{col}}]} \end{pmatrix} \begin{pmatrix} x_{[D_1^{\text{col}}]} \\ x_{[D_2^{\text{col}}]} \\ \vdots \\ x_{[D_k^{\text{col}}]} \\ x_{[L^{\text{col}}]} \end{pmatrix} \geq \begin{pmatrix} b_{[D_1^{\text{row}}]} \\ b_{[D_2^{\text{row}}]} \\ \vdots \\ b_{[D_k^{\text{row}}]} \\ b_{[L^{\text{row}}]} \end{pmatrix},$$

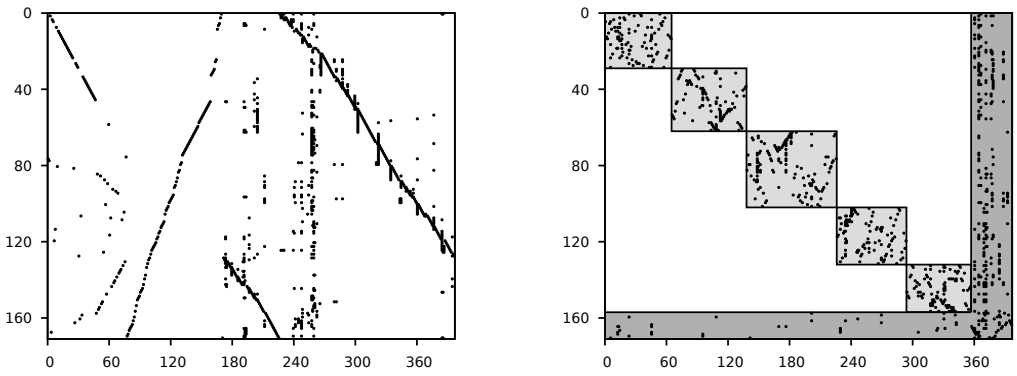
where we use the shortcut $A_{[I,J]}$ to denote the $|I|$-by-$|J|$ submatrix that arises from the deletion of all entries from A except for rows I and columns J, for nonempty row and column subsets $I \subseteq \{1, \ldots, m\}$ and $J \subseteq \{1, \ldots, n\}$. This representation of the matrix A is also called *bordered block diagonal form* [13].

Note first that every matrix admits the trivial decomposition by setting $L^{\text{row}} = \{1, \ldots, m\}$ and $L^{\text{col}} = \{1, \ldots, n\}$, in which case $k = 0$. The other extreme situation occurs for $L^{\text{col}} = L^{\text{row}} = \emptyset$, in which case problem (1) can be solved by first solving each of its k independent diagonal blocks and then combining the k obtained solutions to a solution for (1). If any of the subproblems is infeasible, this infeasibility also holds for (1).

An example for a rearrangement of a matrix A is shown in Figure 1. Nonzero entries $a_{ij} \neq 0$ are visualized as black dots and are scattered wildly in the original arrangement, whereas after a rearrangement based on a decomposition they are only inside blocks or inside the border.

3 Dynamic Partition Search

In the following we propose a construction heuristic called *Dynamic Partition Search* (DPS). According to a decomposition, DPS splits a mixed-integer linear program into independent subproblems. Here, the linking constraints are also split. That is in our case, that each block receives only that part of the linking constraints that contains



(a) Original arrangement of nonzeros.

(b) Arrangement of nonzeros according to decomposition hrcgpartition (see Section 6).

Figure 1: Original and rearranged matrix for MIPLIB 2017 instance `timtab1`.

variables of the block and a limitation of the resources which may be used by this block. DPS searches for a *partition* of the resources or rather of the right-hand side of the linking constraints on the blocks such that we obtain a feasible solution. It achieves this by dynamically updating an initially guessed partition, which led to the naming of the heuristic. This heuristic is inspired by an approach proposed by Yıldız et al. [49] which searches for a partition that belongs to an optimal solution.

3.1 Definitions and Reformulation

First we look at the problem definition and related reformulation. We have given a problem of the form

$$\min \left\{ c^\top x \,:\, Ax \leq U,\, \ell \leq x \leq u,\, x_j \in \mathbb{Z} \text{ for } j \in \mathcal{I} \right\} \tag{2}$$

and a corresponding decomposition $\mathcal{D}$. This heuristic requires that there are only linking constraints and no linking variables, so $L^{\text{col}} = \emptyset$. Otherwise, the decomposition can be adapted, for example, by adding the linking variables to the last block and recompute the constraint labels. Note that the problem is formulated in this section with a right-hand side U in contrast to (1). This is based on the idea that resources need to be distributed among different parties (i.e. blocks).

To simplify the notation, for each block $q \in [k]$ we define $x_q \coloneqq x_{[D_q^{\text{col}}]}$ and $c_q \coloneqq c_{[D_q^{\text{col}}]}$ and summarize all constraints, bounds and integrality conditions in

$$P_q \coloneqq \left\{ x_q \,:\, A_{[D_q^{\text{row}}, D_q^{\text{col}}]} x_q \leq U_{[D_q^{\text{row}}]},\, \ell_{[D_q^{\text{col}}]} \leq x_q \leq u_{[D_q^{\text{col}}]},\, x_j \in \mathbb{Z} \text{ for } j \in \mathcal{I} \cap D_q^{\text{col}} \right\}.$$

In our specific case P_q is the feasible set of a mixed-integer linear problem, but the heuristic is also applicable if P_q is a feasible set of an arbitrary mixed-integer nonlinear problem (MINLP) and, thus, can also be used for MINLPs with linear linking constraints. Moreover, we define $\tilde{A}_q^0 \coloneqq A_{[L^{\text{row}}, D_q^{\text{col}}]}$ for each block and $\tilde{U} \coloneqq U_{[L^{\text{row}}]}$. Thus, we can rewrite problem (2) based on the decomposition $\mathcal{D}$ as

$$\min \quad \sum_{q \in [k]} c_q^\top x_q \tag{3a}$$

$$\text{s.t.} \quad x_q \in P_q, \qquad \forall q \in [k], \tag{3b}$$

$$\sum_{q \in [k]} \tilde{A}_q^0 x_q \leq \tilde{U}. \tag{3c}$$

Note that problem (3) splits into k independent subproblems if we omit the linking constraints (3c).

Since typically not every linking constraint contains variables from every block, we denote by $B(i) \subseteq [k]$ the set of blocks linked by constraint $i \in L^{\text{row}}$, and we denote by $L^{\text{row}}_q \subseteq L^{\text{row}}$ the set of linking constraints containing variables of block q. Moreover, we define $\tilde{A}_q \coloneqq A_{[L^{\text{row}}_q, D^{\text{col}}_q]}$, that means, $\tilde{A}_q$ contains all nonempty rows of $\tilde{A}^0_q$.

The linking constraints can be further divided into the parts of the single blocks by means of additional variables $\pi_q \in \mathbb{R}^{|L^{\text{row}}_q|}$ for each block $q \in [k]$. Thus problem (3) can be reformulated to the equivalent problem

$$\begin{aligned}
\min \quad & \sum_{q\in[k]} c_q^\top x_q && && \text{(4a)}\\
\text{s.t.} \quad & x_q \in P_q, && \forall q \in [k], && \text{(4b)}\\
& \tilde{A}_q x_q \le \pi_q, && \forall q \in [k], && \text{(4c)}\\
& \sum_{q\in B(i)} \pi_{iq} = \tilde{U}_i, && \forall i \in L^{\text{row}}. && \text{(4d)}
\end{aligned}$$

The *partition variables* π_q describe the partitioning of the right-hand side $\tilde{U}$ between the single blocks. To be more precise, a *partition* p of vector $\tilde{U} \in \mathbb{R}^{|L^{\text{row}}|}$ in k blocks is defined as a matrix $p \in \mathbb{R}^{|L^{\text{row}}|\times k}$ such that $\sum_{q\in[k]} p_{iq} = \tilde{U}_i$ for each row i. Thereby an entry p_{iq} in partition p is fixed to zero if $q \notin B(i)$. That is, the right-hand side is split between the blocks in order to fulfill constraint (4d) if π_q is fixed to $p_{\cdot q}$. Every row i of this matrix corresponds to one linking constraint and is denoted by $p_{i\cdot}$, every column q corresponds to one block and is denoted by $p_{\cdot q}$, whereby entries fixed to zero are ignored.

If we know values $p^{\text{feas}}_{\cdot q}$ of the partition variables for one feasible solution, which specify a partition p^{feas}, it is straightforward to calculate a feasible solution for the overall problem. The variables π_q are just fixed to $p^{\text{feas}}_{\cdot q}$ and the k independent and smaller subproblems

$$P_q(p^{\text{feas}}_{\cdot q}) \coloneqq \min\left\{ c_q^\top x_q \,:\, x_q \in P_q,\, \tilde{A}_q x_q \le p^{\text{feas}}_{\cdot q} \right\} \tag{5}$$

are solved until a feasible solution is found. The heuristic searches for a partition related to a feasible solution by dynamically updating an initial guess.

3.2 Algorithm

To get started, for iteration index $t = 0$ we choose an initial partition p^t. There are no further hard restrictions on the partition, but the minimal activity of each row of $\tilde{A}_q x_q$ should be respected, because otherwise the partition can not lead to a feasible solution. That is, we recommend choosing $p^0_{iq} \ge \min\{(\tilde{A}_q)_i x_q \,:\, \ell_q \le x_q \le u_q\}$. Nevertheless, one should keep in mind that the choice of the initial partition has a huge influence on the later solution. DPS does not require an LP solution, but if it is available, the initial partition can be chosen based on it. Otherwise, for example, it is possible to split the right-hand side uniformly.

Then it is checked whether partition p^t will lead to a feasible solution. This can be done by solving k independent subproblems (5) until a feasible solution is found. If all of them are feasible, a solution for problem (2) is found and given by the concatenation of the k subsolutions. If at least one of the subproblems is infeasible, the partition p^t will not lead to a feasible solution and has to be updated.

In case partition p^t will not lead to a feasible solution, we need information about the infeasibility for the update of p^t. To obtain this information, we solve slightly modified

subproblems. For each block we introduce new variables $z_q \in \mathbb{R}_{\geq 0}^{|L_q^{\text{row}}|}$ to ensure feasibility of the constraints $\tilde{A}_q x_q \leq p_{\cdot q}^t$. Moreover, the original objective function is replaced by a weighted sum of the slack variables z_q. The weights are defined by a penalty parameter $\lambda^t \in \mathbb{R}_{>0}^{|L^{\text{row}}|}$. This leads for every block q to the subproblem

$$\begin{aligned} \min \quad & (\lambda^t_{[L_q^{\text{row}}]})^\top z_q && \text{(6a)} \\ \text{s.t.} \quad & x_q \in P_q, && \text{(6b)} \\ & \tilde{A}_q x_q - z_q \leq p_{\cdot q}^t, && \text{(6c)} \\ & z_q \in \mathbb{R}_{\geq 0}^{|L_q^{\text{row}}|}. && \text{(6d)} \end{aligned}$$

Note that the penalty parameter λ is independent of q. When we later increase it, this will effect an information exchange between the blocks. If in some blocks a linking constraint is violated, i.e., $z_{iq} > 0$, all blocks will avoid the violation of this constraint in the next iteration.

Subproblem (5) is feasible if and only if subproblem (6) has an optimal solution with an optimal objective value of zero. If there exists a subproblem with an optimal objective value strict greater than zero, the partition and the penalty parameter are updated. For each single linking constraint i of (4d), we inspect in every block $q \in B(i)$ the value of the slack variable z_{iq} corresponding to linking constraint i and distinguish the following cases:

1. If at least one slack variable is positive and at least one is zero, we update in the following way: The value vector $\bar{z}_i^t$ of the slack variables is added to the subpartition $p_{i\cdot}^t$ and—to keep constraint (4d) satisfied—the same amount is subtracted from all p_{iq}^t with $\bar{z}_{iq}^t = 0$, i.e., for $q \in B(i)$

$$p_{iq}^{t+1} = \begin{cases} p_{iq}^t + \bar{z}_{iq}^t, & \text{if } \bar{z}_{iq}^t \neq 0, \\ p_{iq}^t - \frac{1}{v} \sum_{\phi \in B(i)} \bar{z}_{i\phi}^t, & \text{if } \bar{z}_{iq}^t = 0, \end{cases} \tag{7}$$

   where v is the number of blocks with $\bar{z}_{iq}^t = 0$. Furthermore, the corresponding penalty parameter λ_i^t is increased to avoid the repetitive violation of linking constraint i.

2. If all slack variable values $\bar{z}_i^t$ are greater than zero, only the penalty parameter λ_i^t is increased.

3. If all slack variable values $\bar{z}_i^t$ are zero, no update for constraint i is necessary.

If all coefficients and variables of one row $\tilde{A}_{iq} x_q$ are integral, the corresponding partition value p_{iq}^{t+1} can be rounded to an integral value. However, it must be ensured that constraint (4d) is still satisfied.

After updating, the iteration index t is incremented and the subproblems (6) are solved again. This process is repeated until a feasible solution for problem (4) is found or until a maximum number T of iterations is reached. DPS is formally described in Algorithm 1.

In the special case of two blocks and only one linking constraint, at most one update is necessary. In both blocks, the objective function consists of only one partition variable, which is minimized. If block 1 (or 2) has a positive slack variable after solving with the initial partition, adding this value to the partition of block 1 (or 2) and subtracting it from block 2 (or 1) must necessarily result in a feasible solution. Otherwise, and especially if both slack variables are positive, the original problem is infeasible. Note that in the general case the problem is not necessarily infeasible if all slack variables are positive.

When setting up the subproblems (6), the original objective function is replaced by the weighted sum of the slack variables. This is necessary to push the slack variables to

Algorithm 1: Dynamic Partition Search

Input: Initial partition p^0 fulfilling (4d), penalty parameters $\lambda^0 \in \mathbb{R}_{>0}^{|L^{\text{row}}|}$, and $T \in \mathbb{N}$.

Output: A feasible solution for problem (2) if one has been found.

```
for t = 0, 1, ..., T do
    For each q ∈ [k], solve subproblem (6).
    if all subproblems have an optimal objective value of zero then
        Feasible solution for problem (2) found. Stop.
    else
        Update partition p^t and penalty parameters λ^t.
    end
end
```

zero and thus have a proof for the feasibility of the original subproblems (5). However, this can lead to arbitrary bad solutions of (2). To compensate this drawback we propose the following: If Algorithm 1 returned a feasible solution, we fix in each subproblem (6) the partition to the current value and the slack variables to zero. Afterwards, we reoptimize the subproblems with the original objective function $c_q^\top x_q$.

3.3 Implementation Details

In this section we discuss a few key details of the implementation of Algorithm 1.

DPS as presented above solves the subproblems to optimality. Since this step is very expensive in general, we would like to terminate the solution process early. Obviously, even with a suboptimal solution in some or all subproblems the partition can be updated. But only if all subproblems have an optimal value of zero, we have a verified feasible solution of problem (4). So we stop the solution process of each subproblem as soon as we know that the solution value will be greater than zero in the current iteration and thus at least one additional update is needed. This feature is implemented as a so called event handler in SCIP, which tracks the dual bound.

In order to simplify the presentation, we assumed that all linking constraints are of type '$\leq$'. Whereas '$\geq$'-constraints can be treated analogously, the implementation of DPS is more complex for equations or even ranged constraints, where both sides are finite. In that case, we have to set up and maintain two partitions, p_ℓ and p_r. Internally SCIP treats all constraints as ranged constraints, thus the linking constraints (3c) are of the form $\tilde{b} \leq \sum_{q\in[k]} \tilde{A}_q x_q \leq \tilde{U}$ where $\tilde{b}, \tilde{U} \in (\mathbb{R} \cup \{\pm\infty\})^{|L^{\text{row}}|}$. Although these constraints can be reformulated to '$\leq$'-constraints, we keep them in order to not lose information. So for each linking constraint $i \in L^{\text{row}}$ with finite right-hand and left-hand side the problem reformulation as well as the partition update is slightly different and is applied as follows: For each block $q \in [k]$ we introduce two partition values $p_{iq,\ell}$ and $p_{iq,r}$, and two slack variables $z_{iq,\ell}$ and $z_{iq,r}$, thus (6c) becomes $p_{iq,\ell} \leq \tilde{A}_{iq} x_q - z_{iq,r} + z_{iq,\ell} \leq p_{iq,r}$. When updating the partition it is important that the upper ranged constraint stays feasible, that is, to ensure that $p_{iq,\ell} \leq p_{iq,r}$. So we calculate one common update vector for both sides in which the values of the slack variables of the right-hand side count in with positive sign and the values of the slack variables of the left-hand side count in with negative sign. This update vector is added to both partitions.

Besides updating the partitions, we also have to update the penalty parameter λ_i^t for each linking constraint $i \in L^{\text{row}}$. They are initialized with value 1 and updated if two times in succession at least one of the corresponding slack variables is strictly positive. In this case, λ_i^t is increased by the number of violated blocks multiplied with 100. This

update rule can not lead to numerical problems, since in our implementation the number of iterations is limited to 50 and the number of blocks is usually small enough.

When a feasible solution was found, it can get reoptimized with the original objective function. To keep this step efficient, several solving limits are set. We solve for every partially fixed subproblem only the root node and stop after one improving solution. Apart from that, we stop for each subproblem when the elapsed time exceeds the elapsed time until DPS found the non-reoptimized solution.

A basic version of DPS was published with SCIP version 8 [11]. Since then the heuristic has been extended and improved according to the present paper.

4 Penalty Alternating Direction Method

Here we describe a construction heuristic called *Penalty Alternating Direction Method* (PADM). It splits a MIP into several subproblems according to a decomposition, whereby the linking variables get copied and their difference between the copies is penalized appropriately during the iterations in order to determine a feasible solution. The heuristic shown here is not designed to handle linking constraints. If the decomposition contains linking constraints, one can attempt to assign them to one of the blocks, for example by assigning each constraint to that block to which most of its variables belong and then recomputing the variable labels. A detailed description of penalty alternating direction methods can be found in [25]. For practical applications with a block-separable structure, such as supply chain management problems, this approach has proven to be successful [43].

Classical alternating direction methods (ADMs) are extensions of Lagrangian type approaches [14]. In [25] it is shown that ADMs converge under reasonable assumptions to so called *partial minima*. In our context partial minima are characterized by equal solution values of the linking variables between two successive ADM iterations. However, a partial minimum generally does not correspond to a feasible solution of the original problem, which motivates to embed an ADM inside a penalty framework for solving mixed-integer problems. This can be achieved by using two nested loops. We call the outer loop *penalty-loop* and the inner loop *ADM-loop*.

4.1 Definitions and Reformulation

Consider problem (1) with a corresponding decomposition $\mathcal{D}$ in k blocks without linking constraints. We denote by $L_q^{\text{col}} \subseteq L^{\text{col}}$ the set of linking columns occurring in block $q \in [k]$, that is, for each $j \in L_q^{\text{col}}$ there exists a row $i \in D_q^{\text{row}}$ with $a_{ij} \neq 0$. Furthermore, we denote by $B(j) \subseteq [k]$ those blocks containing linking column $j \in L^{\text{col}}$, which means for each $b \in B(j)$ there exists $i \in D_b^{\text{row}}$ such that $a_{ij} \neq 0$. Moreover, we use iteration index t for the penalty-loop and τ for the ADM-loop. Based on this notation the subproblem of block $q \in [k]$ with penalty parameters $\mu > 0$ can be written as

$$\min \quad \sum_{j \in L_q^{\text{col}}} \sum_{b \in B(j) \setminus \{q\}} \mu_j^{q,b,t,+} s_j^{b,+} + \mu_j^{q,b,t,-} s_j^{b,-} \tag{8a}$$

$$\text{s.t.} \quad A_{[D_q^{\text{row}}, D_q^{\text{col}}]} x_{[D_q^{\text{col}}]} + A_{[D_q^{\text{row}}, L_q^{\text{col}}]} x_{[L_q^{\text{col}}]} \geq b_{[D_q^{\text{row}}]}, \tag{8b}$$

$$x_j + s_j^{b,+} - s_j^{b,-} = \xi_j^{b,\tau}, \qquad \forall j \in L_q^{\text{col}}, b \in B(j) \setminus \{q\}, \tag{8c}$$

$$\ell_j \leq x_j \leq u_j, \qquad \forall j \in L_q^{\text{col}} \cup D_q^{\text{col}}, \tag{8d}$$

$$x_j \in \mathbb{Z}, \qquad \forall j \in \mathcal{I} \cap (L_q^{\text{col}} \cup D_q^{\text{col}}), \tag{8e}$$

$$s_j^{b,+}, s_j^{b,-} \in \mathbb{R}_{\geq 0}, \qquad \forall j \in L_q^{\text{col}}, b \in B(j) \setminus \{q\}, \tag{8f}$$

Algorithm 2: Penalty Alternating Direction Method

Input: Initial values for $\xi^{q,\tau}$ and penalty parameters $\mu^t > 0$.
Output: A feasible solution for problem (1), if one has been found.

```
Set t = 1, τ = 1.
while no feasible solution was determined do
    while no partial minimum was attained do
        for q = 1, ..., k do
            Solve subproblem (8) for block q.
            Update ξ_j^{q,τ} for all j ∈ L_q^col in all subproblems b ∈ B(j) \ {q}.
        end
        Set τ ← τ + 1.
    end
    Choose new penalty parameters μ^{t+1} ≥ μ^t.
    Set t ← t + 1.
end
```

where the slack variables $s_j^{b,+}$ and $s_j^{b,-}$ represent the difference between two copies of linking variable x_j. The right-hand side $\xi_j^{b,\tau}$ represents the solution value of the linking variable in the other block b, which contains a copy of x_j, and need to be initialized beforehand.

4.2 Algorithm

Algorithm 2 shows the basic procedure of PADM. The outer penalty-loop is aborted when a feasible solution has been found, or when a solving limit has been reached, for example. The inner ADM-loop is terminated when a partial minimum is reached, this means the values of the linking variables remain constant. The for-loop inside the ADM-loop solves the individual subproblems and then updates $\xi_j^{q,\tau}$. If a partial minimum does not lead to a feasible solution, the penalty parameters must be increased appropriately to force convergence to a partial minimum that is also feasible.

Since it can be very time consuming to execute the ADM-loop until a partial minimum is reached, it is also possible to terminate earlier. For example, we can abort the inner loop after a fixed number of iterations.

In subproblem (8) the original objective function

$$c_{[D_q^{\text{col}}]}^\top x_{[D_q^{\text{col}}]} + c_{[L_q^{\text{col}}]}^\top x_{[L_q^{\text{col}}]}$$

is completely replaced by a penalty term. This approach has the advantage that PADM usually converges faster to a feasible solution. However, the omission of the original objective function has the effect that often no good solutions are found. To solve this problem the following procedure can be used. First we try to find a feasible solution as described in Algorithm 2. Then the linking variables are fixed to the values of the solution and each subproblem is solved with the original objective function. In some cases this can improve the originally found feasible solution.

4.3 Implementation Details

Before executing PADM, there is to decide how to initialize the values $\xi_j^{q,\tau}$. In statistical studies the values of the variables in an optimal or best known solution were examined. The results for our three test sets described in Section 6 are shown in Figure 2. In the

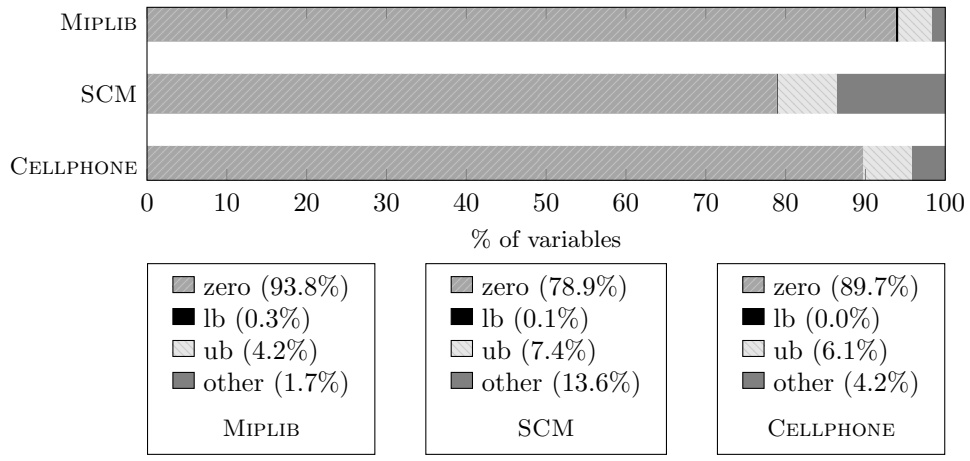


Figure 2: The majority of variables are zero in a best or optimal solution.

test set MIPLIB 93.8 % of the variables take the value zero, 0.3 % are on their nonzero lower bound, 4.2 % are on their nonzero upper bound (but not on their lower bound), and only 1.7 % of the variables have a nonzero value between their bounds. Similar but not such extreme results are observed for both test sets SCM and CELLPHONE. Thus the majority of variables are zero and it is obvious to initialize $\xi_j^{q,\tau}$ with zero.

This heuristic consists of an outer penalty-loop and an inner ADM-loop. In our implementation the ADM-loop stops after a maximum of 4 iterations. Subsequently the penalty parameters μ are updated. We initialize the penalty parameters with one and increase $\mu_j^{q,b,t,+}$ (or $\mu_j^{q,b,t,-}$) after each ADM-loop by factor ten if the slack variable $s_j^{b,+}$ (or $s_j^{b,-}$) is strictly positive in the subproblem of block q. Since this can lead to very large values and thus to numerical problems, we apply a sigmoid rescaling similar to that described by Schewe et al. [43]. If the largest penalty parameter $\|\mu\|_\infty$ exceeds the threshold of $\|c\|_\infty$, each entry μ_i of μ is rescaled by the sigmoid function

$$S(\mu_i) := 5 \cdot \left(\frac{\mu_i - \frac{1}{2}\|\mu\|_\infty}{\frac{1}{10}\|\mu\|_\infty + |\mu_i - \frac{1}{2}\|\mu\|_\infty|} \right) + 5 + 0.1.$$

This step keeps the order of the penalty parameters but maps them into the more controllable interval $[0.1, 10.1]$. If no feasible solution was found after a maximum of 100 penalty-loops, the algorithm stops.

The subproblems (8) can be warmstarted by using the solution from a previous ADM iteration. Let $(\bar{x}, \bar{s})$ be the solution of iteration $\tau - 1$ of block q and $\xi^{b,\tau}$ be the current assignments of the linking variables. Then $(\bar{x}, \hat{s})$ with

$$\begin{aligned} \hat{s}_j^{b,+} &:= \max\{\xi_j^{b,\tau} - \bar{x}_j, 0\} \\ \text{and} \quad \hat{s}_j^{b,-} &:= \max\{\bar{x}_j - \xi_j^{b,\tau}, 0\} \quad \forall j \in L_q^{\text{col}},\, b \in B(j) \setminus \{q\} \end{aligned}$$

is a feasible solution of block q in iteration τ and can be used as start solution.

If Algorithm 2 found a feasible solution, this solution can get reoptimized with the original objective function. In our code the whole problem (1) is copied and the linking variables are fixed at the value in the found solution. This partially fixed problem splits up into independent components, which is handled by SCIP during presolve [22, 26]. To keep the reoptimization step efficient, several solving limits are set. The partially fixed problem is warmstarted with the already found solution and we stop after one improving solution. Moreover, we solve only the root node and we stop if the elapsed time exceeds the elapsed time until the non-reoptimized solution was found by PADM.

A basic version of PADM was published with SCIP version 7 [24] and the extended version as presented in this paper was published with SCIP version 8 [11]. The code of the latter is also identical to the code used for the computational results in Section 6.2.

4.4 Potential of Parallelization

Algorithm 2 in its basic version is not well suited for parallelization due to the condition of the inner for-loop, since one would actually like to compute all subproblems completely in parallel and then determine $\xi_j^{q,\tau+1}$. This can be realized by splitting the inner for-loop in two for-loops. In the first loop all subproblems are solved, and in the second loop the $\xi_j^{q,\tau}$ are updated. We have now different options to update $\xi_j^{q,\tau}$. One option is to keep the update as it is. Another option motivated by [14] is the following: Let $\bar{x}^b$ be part of the solution of iteration τ of block b. Then, for each block q, we can calculate the updates by

$$\xi_j^{q,\tau+1} = \frac{\sum_{b \in B(j)} \bar{x}_j^b}{|B(j)|} \qquad \forall j \in L_q^{\text{col}}.$$

If the associated linking variable is an integer variable, we can afterwards round $\xi_j^{q,\tau+1}$ appropriately. Since these values are independent of q, we can define $\xi_j^\tau := \xi_j^{q,\tau}$ and simplify subproblem (8) by replacing (8c) and (8f) by

$$x_j + s_j^+ - s_j^- = \xi_j^\tau \quad \text{and} \quad s_j^+, s_j^- \in \mathbb{R}_{\geq 0} \qquad \forall j \in L_q^{\text{col}}$$

and by replacing the objective function (8a) by

$$\min \sum_{j \in L_q^{\text{col}}} \mu_j^{t,+} s_j^+ + \mu_j^{t,-} s_j^- .$$

Note that the size of one subproblem is now independent of the number of blocks and that the total number of constraints (8c) is now linear in the number of blocks and no longer quadratic.

We will not continue to pursue the parallel variants. Our code used for the computational experiments in Section 6 can not run in parallel, therefore a computational study on this would not be meaningful. However, we think this is an interesting topic for further research.

5 Decomposition Kernel Search

Problems with (exclusively) binary variables usually have a special structure which can be leveraged to develop tailored heuristics, working efficiently on these problems. *Kernel Search* (KS) lies in the intersection of construction and improvement heuristics, as it tries to find a feasible solution quickly and, then, iteratively attempts to improve it. KS was initially introduced in fields of the binary problems of portfolio optimization [4] and (multi-dimensional) knapsack problems [3]. Later on, it was refined as in [29], respecting the "hardness" of resulting subproblems and integrating binary and pure integer variables. We too make use of the original KS framework and extend it by incorporating decomposition information. We call the resulting framework *Decomposition Kernel Search* (DKS).

In order to keep this chapter self-contained, we formulate the basic KS framework as stated in [29] in a suitable way for our presentation. After an explanation of the characteristics in basic KS, we highlight our adjustments made in order to incorporate the decomposition information. Further, we describe implementational details and highlight enhancements which, to the best of our knowledge, were introduced here for the first time.

5.1 Kernel Search Framework

In [29] the kernel is initially defined as a subset of the occurring integer variables. It is supposed to contain the "promising" variables, i.e., variables likely to play a central role in good solutions. The remaining integer variables are partitioned into *buckets*. In an iterative manner, the original problem is restricted to the union of the current kernel with one bucket, fixing every integer variable not in this union to zero or its lower bound, and then solved. Depending on the solution, the kernel is updated before combining it with the next bucket. The idea behind KS constitutes of solving the restricted problems quickly due to their (hopefully) small amount of unfixed integer variables and improving a known solution thereby. As can be seen from Figure 2, the majority of the variables in a good or optimal solution are zero. Hence, by fixing them to their lower bound beforehand, which often is zero, we potentially decrease the solution time needed.

We start by giving some notation in order to formulate the basic algorithm precisely. The subset of integer variables $K \subseteq \mathcal{I}$ denotes the current kernel. Further, we define $N_\mathrm{b} \in \mathbb{N}$ as the number of buckets which the remaining integer variables $\mathcal{I} \setminus K$ are divided into. We speak of a restriction of the original MIP (1) to a set of integer variables $J \subseteq \mathcal{I}$, whenever all integer variables with indices in $\mathcal{I} \setminus J$ are fixed to their lower bounds (or zero for negative bounds) and all integer variables with indices in J are unfixed. We note that in the basic KS framework continuous variables are either not present or unfixed. In particular, the restricted problem is defined as

$$\mathrm{MIP}(J) \coloneqq \min\{c^\top x \,:\, Ax \geq b,\, l \leq x \leq u,\, x_j = \max\{0, l_j\} \text{ for } j \in \mathcal{I}\setminus J,\, x_j \in \mathbb{Z} \text{ for } j \in J\}. \tag{9}$$

Lastly, we denote an upper bound to the solution value of (1) with $\overline{z}$ which can be $+\infty$. The notation enables to formulate the basic KS algorithm; see Algorithm 3. Note that setting $\mathcal{B}_1 = \emptyset$ forces the first restricted MIP to be solved on the initial kernel only.

Algorithm 3: Kernel Search

Input: Problem (1) with $\overline{z} \in \mathbb{R} \cup \{+\infty\}$.
Output: Solution x^new with $z^\mathrm{new} \leq \overline{z}$ for problem (1) if one has been found.

1 Set $z^\mathrm{new} \leftarrow \overline{z}$.
2 Determine a kernel $K \subseteq \mathcal{I}$.
3 Divide $\mathcal{I} \setminus K$ into buckets $(\mathcal{B}_i)_{i=1,\dots,N_\mathrm{b}}$ with $\mathcal{B}_1 = \emptyset$ and $\mathcal{B}_i \neq \emptyset$ for $i > 1$.
4 **for** $i = 1, \dots, N_\mathrm{b}$ **do**
5 Solve MIP$(K \cup \mathcal{B}_i)$ to obtain $\tilde{x}$ or message "MIP$(K \cup \mathcal{B}_i)$ is infeasible".
6 **if** MIP$(K \cup \mathcal{B}_i)$ *infeasible* **then**
7 Continue with next iteration.
8 **else if** $c^\top \tilde{x} \leq z^\mathrm{new}$ **then**
9 Set $x^\mathrm{new} \leftarrow \tilde{x}$ and $z^\mathrm{new} \leftarrow c^\top \tilde{x}$.
10 Adjust the kernel K w.r.t. $\mathcal{B}_i$ and $\tilde{x}$.
11 **end**
12 **end**

5.2 Extension to Decomposition Kernel Search

For the determination of the initial kernel K in step 2 of Algorithm 3, [3, 4] use the LP relaxation. If a variable's value in the relaxation exceeds its lower bound, its index is added to K. In the remainder of this section, we will call such a variable *active* with respect to the relaxation. Note that the index of a free variable, i.e., a variable without any bound, will always be added to K.

	$\mathcal{B}_0$	$\mathcal{B}_1$	$\mathcal{B}_2$
bin	0	1	4
int	1	0	4
con	1	4	9

(a) Number of active variables for instance A which contains 28 binary, 31 pure integer, and 52 continuous variables.

	$\mathcal{B}_0$	$\mathcal{B}_1$	$\mathcal{B}_2$
bin	0	3	2
int	3	0	2
con	1	5	13

(b) Number of active variables for instance B which contains 28 binary, 28 pure integer and 103 continuous variables.

Table 1: Exemplary number of active variables in two instances.

Now, as we want to incorporate decomposition information in the KS framework, we require several extensions. The major one tackles the construction of the initial kernel and the respective buckets. As exemplified in Table 1 for two different instances from the SCM test set used in Section 6, our observed solutions typically show active variables in (almost) all blocks and (almost) all variable types. This motivates a block-wise construction of the kernel and the buckets, where we consider the block with linking variables as the zeroth block. This construction can lead to kernels and buckets with a large size in comparison. In order to keep the solution process of the restricted problems fast and as there exist binary, pure integer, and continuous variables in the general MIP setting, we, thus, propose a similar separation for the variable types. Note that an additional split regarding the continuous variables is performed in contrast to the approaches in [3, 4, 29], where the kernel is considered to consist only of binary/integer variables.

In particular, we consider the qth block, $q \in \{0\} \cup [k]$ and define $D_0^{\text{col}} \coloneqq L^{\text{col}}$ as the indices corresponding to the linking variables. Then, for every variable x_j with $j \in D_q^{\text{col}}$ and $\tau \leftarrow \text{type}(x_j)$, we check for its activity. If the variable is active with respect to a given solution x^* of (a relaxation of) problem (1), its index j is added to the set K_q^τ which we call the binary/integer/continuous *sub-kernel* (of block q), for the respective $\tau \in \{\text{bin}, \text{int}, \text{con}\}$. Here and from now on, we use *bin*, *int*, *con* as abbreviations for binary, pure integer, and continuous, respectively. The resulting initial sub-kernel of block q and the overall initial kernel are constructed as simple unions of the sets above, i.e.,

$$K_q \coloneqq K_q^{\text{bin}} \cup K_q^{\text{int}} \cup K_q^{\text{con}}, \qquad q \in \{0\} \cup [k],$$

and

$$K \coloneqq \bigcup_{q \in \{0\} \cup [k]} K_q,$$

respectively. Note that, in contrast to the basic KS framework, it is $K \subseteq [n]$. Hence, in (9) we have to replace the first occurrence of $\mathcal{I}$ by $[n]$.

Note that we include a solution to (1) as a possible indication to determine activity of variables. Since DKS is integrated in SCIP, feasible solutions may have been found before DKS is called. Assuming the current best solution to be close to the optimal one, the former might give a better starting point for the kernel identification than the LP relaxation.

Although seeming exaggeratedly complicated on first sight, we construct the buckets analogously for each variable-type-block combination and unite afterwards. In particular, for every variable type τ and for every block q, we define so-called *sub-buckets* $\mathcal{B}_{i,q}^\tau$, $i = 1, \ldots, N_{\text{b}}$. Then, for each bucket index $i = 1, \ldots, N_{\text{b}}$, we set bucket $\mathcal{B}_i$ as the union of sub-buckets $\mathcal{B}_{i,q}^\tau$ over every variable type τ and block q. According to the inclusion of different variable types per bucket, we call the final construction *multi-level-buckets*.

After refining the composition of kernel and buckets, the solution to the restricted problems (see step 5 of Algorithm 3) will now be addressed. Similar as in [3, 4], we add

a constraint to enforce the activation of at least one binary or pure integer variable with index in the current bucket by

$$\sum_{j \in \mathcal{B}_i^{\text{bin}} \cup \mathcal{B}_i^{\text{int}}} x_j \geq \left(\sum_{j \in \mathcal{B}_i^{\text{int}}} l_j \right) + 1. \tag{10}$$

We excluded continuous variables from the definition of inequality (10) for two reasons: (i) Continuous variables can exceed their lower bound by a small $\varepsilon > 0$ and be considered active without a major influence on the objective value. (ii) An enforcement onto a single continuous variable to exceed its lower bound by a constant, e.g., 1 (as above), can even lead to infeasibility, as continuous variables can be influenced by scaling, see, e.g. [15].

Now, we are able to formulate the entire *Dekomposition Kernel Search (DKS)* framework; see Algorithm 4.

Algorithm 4: Dekomposition Kernel Search

Input: Problem (1) with decomposition $\mathcal{D}$ and $\overline{z} \in \mathbb{R} \cup \{+\infty\}$.
Output: Solution x^{new} with $z^{\text{new}} \leq \overline{z}$ for problem (1) if one has been found.

```
Take the best solution x* to (a relaxation of) problem (1).
Use x* to determine sub-kernels K_q^τ and unite them to a kernel K.
Construct the multi-level buckets B_i for i = 1, ..., N_b.
for i = 1, ..., N_b do
    Add the objective cutoff and (10) w.r.t. B_i to MIP(K ∪ B_i).
    Solve MIP(K ∪ B_i) to obtain x̃ or message "MIP(K ∪ B_i) is infeasible".
    if MIP(K ∪ B_i) infeasible then
        Continue with next iteration.
    end
    Set x^new ← x̃ and z^new ← c^⊤ x̃.
    Adjust the kernel K w.r.t. B_i and x̃.
end
```

5.3 Implementation Details

It remains to clarify how to define the buckets resulting from sub-kernels. Following [4], we first find a number of buckets. This is used to split the variables that are not in the kernel into equally sized buckets (possibly except for the last one). For clear notation, we define the complement of a binary sub-kernel of block $q \in \{0\} \cup [k]$ as $\bar{K}_q^{\text{bin}} := \{j \in D_q^{\text{col}} : \text{type}(x_j) = \text{bin}\} \setminus K_q^{\text{bin}}$. The analogous notation is used for $\bar{K}_q^{\text{int}}$ and $\bar{K}_q^{\text{con}}$. As the different blocks and variable types may vary in absolute numbers of variables, we compute the number of buckets N_{b} as a ratio averaged over block and variable type. In particular, we define the set of block-variable combinations to take into account with

$$\mathcal{C} := \{(q, \tau) \in (\{0\} \cup [k]) \times \{\text{bin}, \text{int}\} : (|K_q^\tau| > 0) \wedge (|\bar{K}_q^\tau| > 0)\},$$

and set

$$N_{\text{b}} := \begin{cases} \dfrac{1}{|\mathcal{C}|} \displaystyle\sum_{(q,\tau) \in \mathcal{C}} |K_q^\tau| / |\bar{K}_q^\tau|, & \text{if } |\mathcal{C}| > 0, \\ 0, & \text{else.} \end{cases}$$

It is noteworthy that we excluded the continuous variables in this calculation. This originates from the observation that the number of active continuous variables is low

in comparison to the overall number of continuous variables. Therefore, including the continuous variables in the averaged ratio, the latter tends to zero and leads to many but too small buckets. Accordingly, in each considered restricted problem, only a small amount of additional unfixed variables is investigated which is not suited to obtain a reasonable better solution.

Now, we need to explicitly define the sub-buckets. Hence, we consider a block index $q \in \{0\} \cup [k]$, a variable type τ, and a bucket index $i \in \{1, \dots, N_\mathrm{b}\}$. For clear notation, we interpret the complement $\bar{K}_q^\tau$ of sub-kernel K_q^τ as an ordered tuple and denote $\bar{K}_q^\tau[n_0 : n_1]$ as the subset of $\bar{K}_q^\tau$ containing its n_0th to n_1th element. Then, we set

$$n_0 \leftarrow \left\lceil \frac{|\bar{K}_q^\tau|}{N_\mathrm{b}} \cdot (i-1) \right\rceil + 1 \qquad \text{and} \qquad n_1 \leftarrow \left\lceil \frac{|\bar{K}_q^\tau|}{N_\mathrm{b}} \cdot i \right\rceil,$$

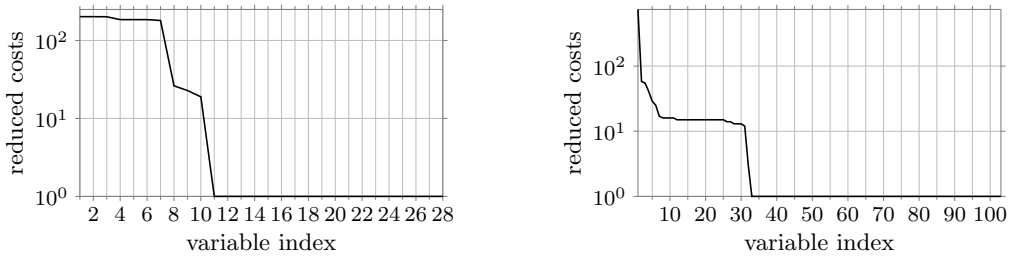
and define $\mathcal{B}_{i,\,q}^\tau \coloneqq \bar{K}_q^\tau[n_0 : n_1]$. As mentioned above, the final buckets $\mathcal{B}_i$, $i \in \{1, \dots, N_\mathrm{b}\}$, result from uniting. We note that the bucket sizes may differ by one per variable type and block involved due to the rounding scheme used.

Finally, we would like to point out two extensions of the procedure presented. The first extension is based on an observation on the values of the reduced costs and the second extension tries to realize a balance between solution time and solution quality by an adaptive *MIP-gap control*.

Logarithmic reduced costs grouping An analysis of several test instances revealed an interesting pattern in the reduced costs. For each variable type, the variables could be divided into groups with reduced costs in different orders of magnitude. In particular, in Figure 3 we visualize the situation for the pure integer and continuous variables of instance B from Table 1. We note that almost all binary variables of this instance belonged to the initial kernel, whereas the rest showed reduced costs of zero. To simplify the presentation on the logarithmic scale, values between 0 and 1 are presented as 1. For example, one can observe that we could divide the 28 pure integer variables in Figure 3a into a group of seven variables with reduced costs of about 200, three variables with about 20, and the remaining 18 variables with values of 1. For a maximization problem, it is considered that the higher the reduced cost of a variable the greater a notion of importance in the optimal solution, see, e.g., [3]. In our case, this can be applied analogously with low reduced costs due to the minimization problem, which gives rise to the idea that variables of one group serve similar purposes and, thus, should be investigated simultaneously in order to choose the "best" one. Hence, such a mechanism investigates groups with low reduced costs before larger ones, trying to identify the structure of the problem.

In detail, we algorithmically implemented the observation on the reduced costs as follows. For each variable type τ and block q, we compute the maximal reduced costs $r_{q,\tau}^{\max}$. Assuming $N_\mathrm{b} > 1$, one computes the base value $b_{q,\tau} = (r_{q,\tau}^{\max})^{1/N_\mathrm{b}}$. For bucket index $i = 1, \dots, N_\mathrm{b}$, the indices of variables with reduced costs in the interval $(b_{q,\,\tau}^{i-1}, b_{q,\,\tau}^{i}]$ are assigned to bucket $\mathcal{B}_{i,\,q}^\tau$. For sub-buckets $\mathcal{B}_{1,\,q}^\tau$, we define the lower bound to be $-\infty$ in order to respect the edge case, where variables have a reduce cost of below 1. The final buckets $\mathcal{B}_i$ are then received by uniting over all blocks and variable types. Note that such a procedure replaces the bucket construction described above, but probably leads to the fact that the sizes of the buckets differ greatly. Though, it is noteworthy that the amount of buckets N_b computed before stays the same, as throughout all blocks and variable types the averaged ratio approximately equals the number of reasonable classes for dividing logarithmically.

MIP-gap control We tried to figure out how to keep the solution time of the restricted problems in step 6 of DKS (Algorithm 4) low, while maintaining an improving result.



(a) Reduced costs for pure integer variables.

(b) Reduced costs for continuous variables.

Figure 3: Reduced costs for pure integer/continuous variables on instance B.

An overall time limit for DKS (specified below) is distributed uniformly on the restricted problems to investigate. If one problem requires less time, the saved time is distributed among the subsequent problems. In the event that the time limit for a restricted problem is reached, an adaptive adjustment of the MIP-gap is made.

Technically, we have implemented the adaptive MIP-gap control as follows. As introduced in Section 5.1, consider the number of buckets N_b. We start by defining a current gap $\Delta = 0$, a maximal gap $\Delta_\mathrm{max} \geq 0$, and a factor of $\delta = 1$. If a problem reaches the time limit, we increase Δ for the consecutive problems by $\delta\Delta_\mathrm{max}/(N_\mathrm{b} - 1)$. Therefore, if all problems hit the time limit, the last bucket is solved for the maximal gap Δ_max. But, if a problem hits the gap limit and, thus, finishes earlier than the time limit, we want to give more time in order to find a better solution. Here, the factor δ comes into play. We divide δ by two every time the hit limit changes from gap to time and vice versa. In other terms, we try to find a gap which balances the allocated time and a desired high solution quality via binary search.

Settings As DKS was implemented inside the SCIP framework, we fixed three decisive SCIP-specific parameters in order to use DKS in all its extensions. First, we set the parameter `HEUR_PRIORITY` $= -1102500$, which can be considered as medium priority in comparison to other heuristics. This enables DKS to use a current best solution, which has hopefully been found by other heuristics before its run, instead of the solution to the LP-relaxation when identifying the initial kernel. Second, we forced SCIP to call DKS only at the root node after having already solved the LP-relaxation. On the one hand, this gives other heuristics the chance to find a feasible solution which can be used in DKS. On the other hand, we tried to leverage the character of DKS as a construction heuristic, i.e., finding a feasible solution at the start which the remaining solving process can work with. We note that DKS may consume a lot of time, as itself solves constraint subproblems of the original one, which motivates its seldom use. This does not prohibit a use of DKS on deeper levels of the branch-and-bound tree throughout further experiments, but is simply out of scope for the current study. Third, we limited DKS to use a maximum of ten percent of the overall time assigned to SCIP for the solution of a problem. Clearly, such a value avoids the heuristic to consume all running time, though we consider ten percent to be quite a lot.

Additionally, we set varying values for four internal, DKS-specific parameters, resulting in three different settings to be investigated. A summary is given in Table 2. In the following, we describe the parameter's effect and the respective choice of values.

As described in Section 5.2, the decomposition information is used to identify a kernel and the respective buckets. This can be avoided by setting a parameter called `use_decomp` to false. In such a case, the heuristic does not respect different blocks and distinguishes by variable type only. The described (logarithmic) sorting in reduced costs can be conducted in either case. Its (de)activation can be controlled by setting `sort_redcost` respectively.

	use_decomp	sort_redcost	add_usecon	add_objcut
dks_default	True	True	True	True
dks_no_cut	True	True	True	False
ks_default	False	False	False	False

Table 2: Parameter combinations defining DKS settings.

The main part of the computing time required by DKS is devoted to solving the resulting subproblems. Since trying to accomplish an exact solution of the subproblems may lead to unreasonable long running times of the heuristic, we introduce the parameters `add_usecon` and `add_objcut` to overcome that issue. The parameter `add_usecon` enables the inclusion of constraint (10) und the parameter `add_objcut` inserts a objective function value cutoff constraint to avoid worse solutions.

For our computational study, we defined three different settings: `dks_default`, `dks_no_cut`, `ks_default`. The first one enables every extension included and, thus, serves as a default setting for DKS. In order to compare the specific use of the objective cutoff, we defined `dks_no_cut` which only differs to `dks_default` in the use of the mentioned constraint. Lastly, we wanted to compare these two settings to the basic KS framework as presented in Algorithm 3. Hence, we forced the setting `ks_default` to not use the decomposition, as well as no additional constraints or sorting.

6 Computational Results

In this section we present computational results for the decomposition heuristics introduced in Section 3, Section 4, and Section 5. All heuristics were implemented in C and integrated as heuristic plugin in SCIP. The code is available at https://github.com/khalbig/decomposition-heuristics [36].

Computational setup All presented computational results were generated on a compute cluster using compute nodes with Xeon E3-1240 v6 processors with 3.7 GHz and 32 GB RAM; see [19] for more details. The optimization problems are solved by using SCIP 8.0.0 [11] linked with SoPlex 6.0.0 [11] as LP solver.

We used a time limit of 20 minutes. To avoid interactions between the presented algorithms, in runs for one heuristic the other two were deactivated. Furthermore, we have deactivated the only other decomposition heuristic in SCIP, namely GINS [24], which, however, is not within the scope of this paper. Apart from that, all parameters of SCIP that do not belong to one of the presented heuristics have been left at their default settings.

Providing Decompositions To have a decomposition available, of course, the solver itself could create such information. Widespread and efficient methods to create a decomposition are, for example, specialized graph partitioning algorithms [5, 13, 32, 33, 35, 45] or hypergraph partitioning [37, 38]. But in practice the user is aware of details of the model and the underlying problem structure to generate an appropriate decomposition. Usually such a decomposition is much more useful than a solver-created decomposition and one saves the time of creation.

SCIP 7.0 [24] has been extended by the necessary capabilities to read decompositions into a central storage, so heuristics and other plugins can query them. Classical formats to store MIPs such as `.mps`, `.lp`, or SCIP's own `.cip` format do not convey decomposition information to the solver. A decomposition can be entered to SCIP as a separate `.dec`-file, for example, which contains the row labels D^{row} of a decomposition

```
NBLOCKS
10
BLOCK 1
R0001
R0002
R0003
...
R0960
BLOCK 2
...
BLOCK 10
R0091
...
R1026
MASTERCONSS
R0101
...
R0110
```

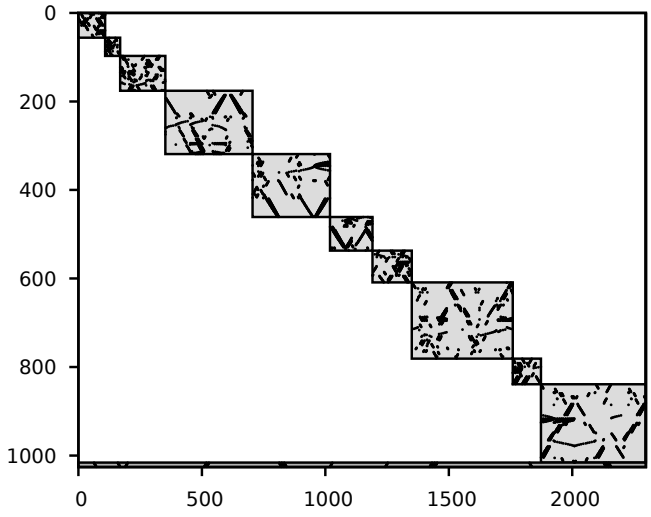


Figure 4: Example of `.dec`-format with rearranged constraint matrix for the MIPLIB 2017 instance `binkar10_1` with decomposition from [50].

assigned to the respective block or to the linking rows. After scanning through the row labels, the column labels D^{col} are automatically determined by means of the row labels to complete the decomposition information.

An example of the `.dec`-format with associated rearranged constraint matrix A is shown in Figure 4. A `.dec`-file starts with a section that contains the number of blocks and then follows for each block a section that lists the names of the rows that are part of this block. Finally, the special section `MASTERCONSS` contains all rows belonging to L^{row}.

During the solution process the original problem (1) may be changed, for example, by deleting rows or fixing columns in presolve [2, 22, 26]. Such model changes then also lead to an appropriate adjustment of the decomposition information.

More details about the management of decompositions in SCIP can be found in [24].

Test sets We consider three sets of instances. The first set of instances is based on the MIPLIB 2017 [28, 50] benchmark set. This set originally contains 240 instances, but we removed the 8 infeasible instances since our heuristics are not designed to detect infeasibility. Each instance has up to four decompositions. The first type of decomposition is given by the publicly available decompositions at [50] (denoted by "miplib2017"). The other three types are generated by GCG [21] version 3.0.2 using different parameters and are accessible at [36]. For type "plain_miplib" default parameters for the MIPLIB 2017 were used, for type "deactivate_nonzero" the nonzero classifier was deactivated in addition, and for type "hrcgpartition" the hypergraph method was activated in addition. Trivial decompositions with only one block and decompositions that were immediately recognized as duplicates were removed, as well as instances for which no decomposition could be detected. The final test set contains 216 unique instances with up to four decompositions each, resulting in a test set with 694 instance-decomposition combinations.

The second set is based on 41 real-world supply chain management (SCM) instances supplied by our industry partner SAP [40]. The most important components are stock keeping, capacity restrictions, transport, production and demand fulfillment. A more detailed description can be found in [23]. Since these instances contain many independent components (see [22]), we selected only non-trivial integer and mixed-integer components. In our case, a component is called non-trivial if it has at least 100 variables,

including at least one integer variable, and if it cannot be solved to optimality within 10 seconds with SCIP 8.0.0. To avoid that some original instances are overrepresented we removed some of the components, which belong to the same original instances and have a similar size. The remaining components form our test set of 33 SCM instances.

The SCM instances were decomposed according to economic aspects. Each instance covers a period of time, which is subdivided into so-called time buckets. Thus one possibility is to decompose according to time buckets (B). Another option is to decompose according to organizational units of the supply network, called locations (L). A third option is a decomposition according to products (P). In our instances products can be raw materials as well as intermediate products and end products. The fourth and last way is to decompose by production process models (S), which transform one or more products into one or more other products.

The number of blocks is equivalent to the number of different time buckets, locations and so on (denoted by "0"). Since this number can be very large, which is not always an advantage, we have also remerged the blocks into 2 and 4 blocks respectively. So each instance can have up to 12 different decompositions. After removing obvious duplicates and decompositions with only one block we get a test set of 322 instance-decomposition combinations.

The third set contains randomly generated supply chain management instances based on real-world supply chain management models. They represent a fictive company procuring components, producing cellphones of different types, transporting to distribution centers, and satisfying costumers demand. This set contains 56 instances with different number of time buckets and customers. Analogously to the SCM test set it is decomposed in time buckets (B), locations (L) and products (P). Also the number of blocks is determined in the same way. This test set is provided by SAP [40] and is publicly available at [41]. We have selected all 56 instances with discrete time buckets only and with lot size factor 3, and all corresponding nine types of decompositions. This results in a test set of 504 instance-decomposition combinations and we refer to it hereinafter as CELLPHONE.

Measurement In order to evaluate algorithmic performance of the presented heuristics, we compare *shifted geometric means* of *primal integrals*, both for the test set as a whole and also for insightful subgroups. The primal integral is an absolute measure for the performance of heuristics. It takes the evolution of the incumbent solution over time into account, thus favoring algorithms that find good solutions early. For a detailed description of the primal integral see Berthold [8]. The best known solution for an instance as base value is given by publicly available solutions (see [50]) and/or by results of previous and for this paper used runs.

After computing the primal integral for each instance-decomposition combination, we calculate the shifted geometric mean (SGM) of these values. For values $w_1, \ldots, w_N \geq 0$ we determine the SGM by

$$\operatorname{sgm}(w_1, \ldots, w_N, s) \coloneqq \left(\prod_{i=1}^{N} (w_i + s) \right)^{\frac{1}{N}} - s, \tag{11}$$

where we use a shift value of $s = 1$. Compared to the arithmetic mean, outliers with large absolute values are not overrepresented, and owing to the shift, very small values are not overrepresented. For further discussion on evaluating computational results with SGM see for example [1].

The SGM for one setting for a set of instance-decomposition combinations is compared to the SGM for the same set of instance-decomposition combinations running without all presented three heuristics. A ratio less than 1 indicates an improvement in performance.

Table 3: Number of instances on which DPS was called or found an accepted feasible solution

Test set	Setting	#Called	#Found
MIPLIB (#694)	`default`	338	76
	`reopt`	338	98
	`lp`	76	24
	`lp_reopt`	76	32
SCM (#322)	`default`	182	93
	`reopt`	182	104
	`lp`	143	71
	`lp_reopt`	143	73
CELLPHONE (#504)	`default`	504	482
	`reopt`	504	482
	`lp`	430	60
	`lp_reopt`	430	60

6.1 Dynamic Partition Search

In the following we analyze the performance of DPS. The construction heuristic was called at two different times—as first heuristic after the presolving process, i.e., pre-root (`default`) and as first heuristic after root node computation (`lp`). In the first case, we use for each linking constraint $i \in L^{\text{row}}$ as initial partition the evenly divided right-hand side, in the second case, the LP solution is available and thus the initial partition bases on it. Either way, we respect the minimal and maximal activity of each row i of $\tilde{A}_q x_q$. Furthermore, we run each with (`reopt` or `lp_reopt`) and without reoptimization. Thus we consider four different settings.

First, we examine how many times DPS was called on the three different test sets with the four different settings, and how many times DPS successfully found a feasible solution. These results are given in Table 3. An instance-decomposition combination is counted in column "Called" if the heuristic was called at least once on it. Although DPS is called only at the root node, it can get called several times on one instance-decomposition combination, because of restarts in SCIP. In column "Found" all instance-decomposition combinations are counted on which DPS successfully found at least one feasible solution.

Now we look exemplarily at a few numbers of Table 3. On MIPLIB running pre-root DPS could get called on 338 out of 694 instance-decomposition combinations. Due to general restrictions it is not possible to call DPS in every case. This restrictions are, for example, nonlinear linking constraints (such as `orbitope` constraints) due to upgrading of constraints during presolving, or a high estimated memory usage that could exceed the memory limit due to a high number of blocks. In addition, running after-root the solving process might not have reached the calling point. In 76 out of these 338 cases DPS successfully found a feasible solution. If reoptimization is activated, DPS found pre-root on 22 instance-decomposition combinations in addition a feasible solution, whereby the reoptimization step run successfully in 74 cases. Obviously, the reoptimization step can only find a solution if DPS has already found one before. However, the first solution is sometimes not accepted by SCIP because the objective value is considered infinite or because exactly this solution has already been fed in by another heuristic. It is noteworthy that on CELLPHONE running pre-root DPS could get called in every case and also in almost all cases an accepted solution was found.

In the following, we evaluate the performance of the heuristic using the primal in-

Table 4: Ratio of SGM's of primal integral grouped by decomposition for DPS for test set MIPLIB

setting	deactivate_nonzeros #93	hrcgpartition #94	miplib2017 #89	plain_miplib #62	best #124	all #338
`default`	1.69	1.60	1.50	1.53	1.48	1.58
`reopt`	1.72	1.58	1.50	1.54	1.47	1.59
`lp`	1.14	1.11	1.05	1.11	1.07	1.10
`lp_reopt`	1.13	1.10	1.03	1.11	1.06	1.09

tegral. To get more meaningful results and not dilute them, we need to clean our data first. We remove an instance-decomposition combination for all settings if any of the following occurs: (i) SCIP did not terminate in a clean way for one setting; (ii) our heuristic was not called at least for one setting; (iii) the primal integral is less than 10^{-4} for all settings. Property (ii) would not measure the performance of the heuristic but rather the performance variability of the used compute nodes. Instance-decomposition combinations with Property (iii) would lead to extreme ratios, which do not give a reliable statement of the performance.

In Table 4, Table 5, and Table 6 the ratios of the primal integrals for the three test sets are given. Column "all" first gives the number of instance-decomposition combinations after cleaning the data and then the ratio of the shifted geometric means of the primal integral for all four settings. For column "best" we select for every unique instance the corresponding decomposition that yields the best ratio of the primal integrals. So, assuming we know which of the given decompositions is best for DPS, this is the achievable performance improvement. The other columns subdivide all instance-decomposition combinations of column "all" according to the type of decomposition.

In Table 4 for test set MIPLIB, one can see that after clean up 338 out of 694 instance-decomposition combinations are left corresponding to 124 individual instances. Unfortunately, no performance improvement can be measured, even if the best decomposition is selected. Decompositions of type miplib2017 still seem to be the most suitable for DPS.

However, we can spot some instances where DPS achieves a strong performance improvement. For example, running pre-root on instance `proteindesign121hz512p9` DPS is the only root heuristic that can find a feasible solution for all four types of decompositions, whereby the initial partition leads directly to a feasible solution and no update is necessary. The solution of DPS is the only one until the time limit is reached and SCIP stops with a MIP-gap of 5%. In contrast, SCIP without DPS restarts and the first feasible solution is found after 45 branching nodes by the LP relaxation. The run stops with a MIP-gap of 8%. Comparing the settings, DPS drastically reduced the primal integral by about 78% on this instance. As a second example, running after-root on instance `lotsize` with decomposition deactivate_nonzeros or plain_miplib the solution fed by DPS reduces the MIP-gap from 244% to 67% and with reoptimization to 37%. This leads to a reduction of the primal integral of 27% without reoptimization and 38% with reoptimization.

The results for test set SCM are given in Table 5. After cleanup, 182 out of 322 instance-decomposition combinations are left, belonging to 21 individual instances. Whereas no performance improvement can be seen for setting `default` on the whole test set, even if the best decomposition is selected, the situation changes with reoptimization. Assuming we know which is the most useful decomposition, we get an 8% improvement. Running pre-root the performance varies between the different types of decomposition, with only decompositions by location (L) with more than two blocks leading to an average improvement up to 4%. Running after-root we get an improvement of 2% and 4%, respectively, assuming we know the best decomposition. Looking at the different

Table 5: Ratio of SGM's of primal integral grouped by decomposition for DPS for test set SCM

	B			L			P			S			best	all
	2	4	0	2	4	0	2	4	0	2	4	0		
setting	#21	#21	#19	#18	#10	#10	#16	#16	#13	#11	#12	#15	#21	#182
`default`	1.29	1.60	1.67	1.31	**0.96**	**0.98**	1.35	1.31	1.52	2.07	1.58	1.57	1.09	1.42
`reopt`	1.29	1.58	1.67	1.08	**0.99**	1.00	1.39	1.27	1.44	1.85	1.60	1.58	**0.92**	1.38
`lp`	1.04	1.10	1.19	1.20	1.02	1.28	**0.96**	**0.94**	1.24	1.29	1.19	1.24	**0.98**	1.13
`lp_reopt`	1.06	1.10	1.20	1.18	1.03	1.28	**0.98**	**0.97**	1.24	1.30	1.21	1.24	**0.96**	1.13

Table 6: Ratio of SGM's of primal integral grouped by decomposition for DPS for test set CELLPHONE

	B			L			P			best	all
	2	4	0	2	4	0	2	4	0		
setting	#56	#56	#56	#56	#56	#56	#56	#56	#56	#56	#504
`default`	1.93	1.99	1.89	1.95	2.01	1.86	2.02	2.13	1.91	1.68	1.96
`reopt`	2.00	2.05	**0.99**	1.36	1.40	**0.82**	2.01	1.81	**0.82**	**0.65**	1.39
`lp`	1.06	1.05	1.03	1.08	1.07	1.02	1.39	1.64	1.03	**0.98**	1.14
`lp_reopt`	1.07	1.05	1.03	1.08	1.05	1.02	1.41	1.64	1.03	**0.97**	1.14

types of decomposition, an improvement can be observed on decompositions by product (P) with maximal four blocks. Note that the figures with best decomposition can be even worse than over one type of decompositions or over all possible decompositions. Although seeming counterintuitive at first sight, this results from taking a different amount of figures into account, when calculating the shifted geometric means. In particular, the number of instance-decomposition combinations varies between various types of decompositions. This phenomenon also occurs in further tables of this paper.

Considering these results, one can derive the following appropriate control for using DPS on SCM: If a decomposition by product (P) with maximal 4 blocks is available, DPS is called on SCM after-root. Given a decomposition by location (L) with more than two blocks DPS is called pre-root. The reoptimization step should be deactivated, although it leads to a big performance improvement if one knows the best decomposition. In all other cases DPS is switched off.

On this test set we could also observe an interesting behavior. These instances are supply chain management problems and therefore have a special structure based mainly on a network with a multi-commodity flow. They contain many flow conservation constraints at the network nodes, thus these constraints can also be linking ones. Since these are equations with side equal to zero, the initial partition value running pre-root is zero for every block. Consequently, if the incoming flow corresponds to one block and the outgoing flow to another block, there can be no commodity flow at this network node for the initial partition. Nevertheless, DPS can find a solution directly or after some iterations, because, for example, unfulfilled demand is penalized in the original objective function. When iterating, the partition is updated only to the extent that a feasible solution is found. Reoptimization can subsequently improve the objective value but parts of the commodity flow are still fixed to zero. Running after node the initial partition is derived from the LP solution. If there is a positive flow in the LP solution, also the partition value is not zero for every block and, thus, the commodity flow is not fixed to zero. Thus reoptimization has better possibilities to improve the objective value. Calling pre-root DPS can nonetheless lead to an improvement in performance since at this early stage in the solving process only few other heuristics are able to find feasible solutions.

Finally, we take a look at the results for test set CELLPHONE in Table 6. After

cleanup, still all 504 instance-decomposition combinations are left, belonging to 56 individual instances. Again no performance improvement can be seen for setting `default`, even if the best decomposition is selected, but the situation changes with reoptimization. Assuming we know which is the best decomposition, we get a remarkable improvement of 35%. The performance varies widely between the different types of decomposition. Whereas using decompositions with two or four blocks leads to a clear performance degradation, using decompositions with a higher block number depending on the problem structure leads to a clear performance improvement.

Running after-root the results are not such extreme. Knowing the best decomposition we could get an improvement of 2% and 3%, respectively, but there is no specific type of decomposition which leads to that improvement.

The types of decomposition L_0 and P_0 with reoptimization and calling DPS pre-root lead by far to the best results on Cellphone. An appropriate control for a call of DPS on Cellphone could be that DPS is called pre-root with reoptimization if and only if a decomposition by location (L) or product (P) with a block number determined by the number of locations or products is available. Otherwise DPS is switched off.

In summary for all test sets, the performance of DPS depends strongly on the structure of the instance and on the used decomposition. Whereas Miplib is very inhomogeneous and the decompositions are mostly automatically generated, SCM and Cellphone are known to be supply chain management problems and the decompositions have been created according to economic aspects. In particular, we know the structure of the Cellphone instances, which are very similar and only scaled in size. We observed that the more we know about the instances and decompositions of a test set, the better the performance, as the heuristic can be adjusted accordingly.

6.2 Penalty Alternating Direction Method

For the following performance analysis PADM was called at two different times—as first heuristic after the presolving process, i.e., pre-root (`default`), and as first heuristic after root node computation (`lp`). In the first case, we initialize for each linking variable $j \in L^{\text{col}}$ right-hand side $\xi_j^{b,\tau}$ with zero; see Figure 2. In the second case, the LP solution is available and thus $\xi_j^{b,\tau}$ is initialized with the LP solution value of the corresponding linking variable. Furthermore, we run each with (`reopt` or `lp_reopt`) and without reoptimization. Thus we investigate four different settings.

In Table 7 we see how often PADM was called and found a solution. This table is structured in the same way as Table 3 for the results of DPS. For Miplib and SCM our heuristic is able to find a feasible, accepted solution in 50 to 70 percent if it was called. For the Cellphone test set it is noticeable that for decompositions with a block number given by the number of time buckets, locations or products in the model ("0"), and for all decompositions running after-root, the heuristic could not get called at all. The reason is that the linking constraints could not get assigned appropriately to one block. This issue could in principle be overcome, but we would like to keep the implementation of PADM applicable to MIPs in general, and we also do not want to completely restructure the decomposition within PADM, so we did not invest any further work.

Now, we evaluate the performance of PADM using the primal integral. The results are given in Table 8, 9 and 10, which are structured in the same way as the corresponding tables for the evaluation of DPS in Section 6.1 and the data is also cleaned up beforehand.

On Miplib, see Table 8, 124 instance-decomposition combinations belonging to 52 individual instances are left after cleanup. Unfortunately, no performance improvement can be measured, even if the best decomposition is selected. With setting `lp` or `lp_reopt` the ratios are partially neutral, but this is also due to the fact that PADM could get called less often. Note again that the ratio for the best decomposition can be worse than the

Table 7: Number of instances on which PADM was called or found an accepted feasible solution

Test set	Setting	#Called	#Found
Miplib (#694)	`default`	126	63
	`reopt`	126	68
	`lp`	30	15
	`lp_reopt`	30	16
SCM (#322)	`default`	136	95
	`reopt`	136	95
	`lp`	100	56
	`lp_reopt`	100	56
Cellphone (#504)	`default`	247	247
	`reopt`	247	247
	`lp`	0	0
	`lp_reopt`	0	0

Table 8: Ratio of SGM's of primal integral grouped by decomposition for PADM for test set Miplib

setting	deactivate_nonzeros #29	hrcgpartition #32	miplib2017 #35	plain_miplib #28	best #52	all #124
`default`	1.32	1.19	1.29	1.42	1.29	1.30
`reopt`	1.31	1.15	1.30	1.45	1.27	1.29
`lp`	1.00	1.06	1.05	1.00	1.06	1.03
`lp_reopt`	**0.99**	1.06	1.05	1.00	1.05	1.03

Table 9: Ratio of SGM's of primal integral grouped by decomposition for PADM for test set SCM

	B			L			P			S			best	all
	2	4	0	2	4	0	2	4	0	2	4	0		
setting	#21	#17	#7	#18	#10	#9	#18	#17	#9	#5	#4	#1	#26	#136
`default`	1.47	1.96	5.01	1.30	**0.90**	**0.89**	1.27	1.37	**0.99**	1.01	1.01	1.02	1.17	1.35
`reopt`	1.41	1.99	5.04	1.04	**0.92**	**0.90**	1.30	1.40	1.00	1.02	1.04	1.00	**0.97**	1.32
`lp`	1.11	1.09	2.26	1.28	1.06	**0.93**	1.14	1.12	1.22	1.03	1.04	**0.98**	**0.99**	1.15
`lp_reopt`	1.12	1.10	2.25	1.27	1.07	**0.94**	1.15	1.12	1.23	1.07	1.00	1.00	**0.98**	1.15

Table 10: Ratio of SGM's of primal integral grouped by decomposition for PADM for test set Cellphone

	B			L			P			best	all
	2	4	0	2	4	0	2	4	0		
setting	#49	#40	#0	#51	#49	#0	#30	#28	#0	#52	#247
`default`	2.10	2.40	-	2.19	2.27	-	2.34	2.20	-	2.01	2.24
`reopt`	2.15	2.36	-	2.20	2.28	-	2.34	2.20	-	2.00	2.25

ratio for all instance-decomposition combinations because not all types of decompositions exist for each instance.

However, we can identify some cases where PADM provides a strong performance improvement. Having a look at instance `net12`, we observe that PADM could get called pre-root if one of the types hrcgpartition, miplib2017 or plain_miplib of decompositions was used. In all three cases, PADM successfully fed in a feasible solution, which could not get improved by the reoptimization step. Running without PADM SCIP finds the first feasible solution after 46 branching nodes. The primal integral was reduced by our heuristic to about 42%. Also on `50v-10` with decomposition deactivate_nonzeros running after-root PADM is successful. Without reoptimization the found solution is not an improving solution and the ratio of the primal integral is neutral, but with reoptimization the found solution improves the current primal bound and the primal integral is reduced by approx. 29%.

The results for test set SCM are given in Table 9. On the complete test set there is no performance improvement but looking at the best decomposition or single types of decompositions one can observe a performance improvement up to 11%. Especially the type L_0 stands out positively across all settings. The degradation in performance for decompositions by location into two blocks comes mainly from the 9 additional instances that inherently have only two locations. Therefore, the supply chain management problems with multiple locations in this test set appear to have the ideal structure to be solved by PADM using the corresponding decomposition. To get an overall performance boost one should call PADM only if a decomposition of type L_0 is available and otherwise disable PADM.

Finally, we take a look at the results for Cellphone in Table 10. A '-' marks that PADM could not get called for this setting and type of decomposition. The rows for `lp` and `lp_reopt` are not displayed since PADM could not get called at all. If PADM was called, it found a feasible solution in all cases. However, we can not observe an improvement in performance for one of the considered (sub-)sets and it is also noteworthy that the ratios all fall within the same range. Investigating the results in more detail, we were able to find a possible explanation for this behavior: The pre-root heuristic Shift-and-Propagate [10] seems to be a powerful tool for this kind of instances, since in the comparison run without our decomposition heuristics the first solution was mostly found by it. Comparing the solution quality, one can see that Shift-and-Propagate

Table 11: Number of instances on which DKS was called or found an accepted feasible solution

Test set	Setting	#Called	#Found
MIPLIB (#694)	`dks_default`	376	178
	`dks_no_cut`	376	256
	`ks_default`	379	262
SCM (#322)	`dks_default`	121	27
	`dks_no_cut`	121	75
	`ks_default`	121	75
CELLPHONE (#504)	`dks_default`	359	0
	`dks_no_cut`	359	321
	`ks_default`	360	270

outscored PADM in all cases. Even enabling the reoptimization step does not improve the situation, as it was successful in only two cases and otherwise the solution limits have been reached. It might appear that this is just wasting computation time for a useless solution from PADM, but the fact is that Shift-and-Propagate is no longer called if a feasible solution already exists, such as fed in by PADM. Thus, the performance degradation comes mainly from disabling Shift-and-Propagate, which also explains the consistent degradation across the types of decompositions, since Shift-and-Propagate is independent of decomposition information.

In conclusion, we could observe promising results for PADM on certain subsets. Performance might be improved, especially on CELLPHONE, by adapting the algorithm specifically to the test set under consideration and changing the default parameters of SCIP. We explicitly decided against this in order to be able to apply PADM to general MIPs and also do not want to interfere with the behavior of the software SCIP per se. In spite of this, PADM can provide a strong performance improvement in particular cases and especially the results for the real-world supply chain management instances (SCM) are quite pleasing.

6.3 Decomposition Kernel Search

For DKS, we introduced three different settings to investigate which are summarized in Table 2. In similar fashion to Table 3 and Table 7, we compare calls of and feasible solutions found by DKS. As we see in Table 11, every setting was called for almost the same amount of times. The slight difference can be explained by ill-posed decompositions for the requirements of DKS. In particular, the absence of kernel variables or the (estimated) usage of too much memory can lead to skip the heuristic. In terms of found solutions, `dks_default` shows a significant less number. Since the objective cutoff constraint is the only difference between `dks_default` and `dks_no_cut`, we attribute the smaller number to this property. We note that it does not imply that `dks_default` fails to find solutions, but only focuses on strictly improving ones. An extreme of this phenomenon can be observed regarding the CELLPHONE test set. As explained later on, `dks_default` seems to not find improving solutions due to the problem type of the instances in the test set.

We follow the presentation from previous subsections and, hence, consider the performance of DKS regarding the primal integral. Further, all instances were cleaned according to Section 6.1. We start by showing the performance of all three settings on the MIPLIB test set in Table 12. One observes that the standard `ks_default` achieves

Table 12: Ratio of SGM's of primal integral grouped by decomposition for DKS for test set MIPLIB

setting	deactivate_nonzeros #108	hrcgpartition #105	miplib2017 #100	plain_miplib #66	best #124	all #379
`dks_default`	1.13	1.09	1.13	1.16	1.03	1.18
`dks_no_cut`	1.12	1.07	1.11	1.15	1.01	1.16
`ks_default`	1.03	1.01	1.04	1.02	1.02	1.06

Table 13: Ratio of SGM's of primal integral grouped by problem type for DKS for test set MIPLIB

setting	BP #49	IP #48	MBP #193	MIP #77	PIP #12	all #379
`dks_default`	1.03	**0.82**	1.10	1.54	1.31	1.13
`dks_no_cut`	**0.98**	**0.80**	1.12	1.46	1.30	1.12
`ks_default`	**0.81**	1.02	1.03	1.16	1.30	1.03

(a) Ratio of SGM's of primal integral grouped by problem type for DKS for test set MIPLIB over all decompositions

setting	BP #16	IP #16	MBP #64	MIP #24	PIP #4	all #124
`dks_default`	**0.95**	**0.72**	1.02	1.31	1.49	1.03
`dks_no_cut`	**0.82**	**0.72**	1.06	1.23	1.48	1.01
`ks_default`	**0.81**	1.03	1.03	1.09	1.48	1.02

(b) Ratio of SGM's of primal integral grouped by problem type for DKS for test set MIPLIB with best decompositions

a slightly better result on all decomposition types compared to the DKS-settings, but also fails to improve the runs without any Kernel Search framework.

Due to the ambiguous figures and as DKS was initially proposed for binary problems only, we present in addition a distinction by problem type. Here and in the remaining subsection, we differ between

- binary problems (BP), containing binary variables only,
- integer problems (IP), containing pure integer and binary variables,
- pure integer problems (PIP), containing pure integer variables only,
- mixed-binary problems (MBP), containing continuous and binary variables, and
- mixed-integer problems (MIP), containing all three types of variables.

Having a look in the split by problem type in Table 13, setting `ks_default` attains an improvement of nearly 19% on average on BPs over all decompositions and only has slightly negative performance on the biggest problem type class MBP. All three settings fail to tackle mixed- or pure-integer problems. Particularly noteworthy, however, is the performance of `dks_default` and `dks_no_cut` on IPs which show binary and pure integer variables, but no continuous ones. Here, we achieve an improvement of 18% and 20% on average and even 28% with the best decomposition. We emphasize that its reason lies in the application of logarithmically reduced cost sorting, as well as the multi-level structured buckets. Hence, one could try to call DKS inside the SCIP framework only when trying to tackle IPs with decomposition information.

On the SCM test set, we investigate the split over decomposition types first. Here, we observe that `dks_default` performs on average slightly worse than without the heuristic,

Table 14: Ratio of SGM's of primal integral grouped by decomposition for DKS for test set SCM

	B			L			P			S			best	all
	2	4	0	2	4	0	2	4	0	2	4	0		
setting	#14	#14	#14	#11	#5	#5	#10	#10	#9	#10	#10	#9	#14	#121
dks_default	1.01	1.00	1.01	1.01	1.03	1.02	1.02	1.01	1.02	1.01	1.01	1.02	1.00	1.01
dks_no_cut	1.56	1.58	1.66	1.74	1.09	1.09	1.05	1.05	1.02	1.04	1.04	1.02	1.53	1.27
ks_default	1.54	1.54	1.55	1.72	1.07	1.07	1.03	1.03	1.02	1.04	1.03	1.02	1.54	1.24

Table 15: Ratio of SGM's of primal integral grouped by problem type for DKS for test set SCM

setting	MBP #27	MIP #94	all #121
dks_default	1.01	1.02	1.01
dks_no_cut	**0.99**	1.35	1.27
ks_default	1.00	1.32	1.24

(a) Ratio of SGM's of primal integral grouped by problem type for DKS for test set SCM over all decompositions

setting	MBP #3	MIP #11	all #14
dks_default	**0.98**	1.01	1.00
dks_no_cut	**0.97**	1.71	1.53
ks_default	1.00	1.72	1.54

(b) Ratio of SGM's of primal integral grouped by problem type for DKS for test set SCM with best decompositions

even when considering the best decomposition per instance. For the other two settings, the results are slightly to majorly worse, also showing no clear pattern. Hence, we also investigate the problem-type-wise presentation.

Here, the performance of DKS on SCM test set over all decompositions, as displayed in Table 15a, shows comparable to not using DKS on MBPs, whereas general MIPs are treated slightly worse by `dks_default` and quite worse by `dks_no_cut` and `ks_default`. Though, considering to have the best decomposition at hand, see Table 15b, both DKS-settings achieve an improvement of 2-3% on the remaining three MBPs. On MIPs, `dks_default` seems to cancel the solution process early enough such that its performance is comparable, but the other two settings seem to fail heavily on such problem types.

On the CELLPHONE test set, we start by giving a similar decomposition-wise evaluation in Table 16. Here, an analogous picture shows. When we compare to Table 11, `dks_default` did not find a new (better) solution. Hence, it used computation time to execute, but did not result in any improvement which ends up with the slight underperformance. The other two settings `dks_no_cut` and `ks_default` worsen the performance by a lot, presumably not terminating in time due to the absence of an objective cutoff. We assume that the overall negative performance results from the problem structure in the CELLPHONE test set. All instances are general MIPs which are hard to tackle for kernel search frameworks, as we could already observe on SCM and MIPLIB.

In conclusion, KS and DKS frameworks show promising results on (mixed-)binary

Table 16: Ratio of SGM's of primal integral grouped by decomposition for DKS for test set CELLPHONE

	B			L			P			best	all
	2	4	0	2	4	0	2	4	0		
setting	#40	#40	#40	#40	#40	#40	#40	#40	#40	#40	#360
dks_default	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.08	1.01	1.02
dks_no_cut	1.21	1.18	1.13	1.21	1.20	1.17	1.21	1.18	1.23	1.10	1.19
ks_default	1.20	1.20	1.20	1.20	1.20	1.20	1.20	1.20	1.20	1.20	1.20

problems, for which the original algorithm was introduced in [4]. Having a good decomposition at hand, it appears beneficial on these problem types to run the heuristic. In contrast, the heuristics struggle on mixed-/pure-integer problems in general which does not recommend their use on them. Nevertheless, the extensions included in DKS enable to achieve a great performance on problems which only consist of binary and pure integer problems at the same time, making it advisable to call DKS on such types.

7 Conclusion

In this paper, we propose three heuristics exploiting decompositions for solving mixed-integer problems. We restrict ourselves to decompositions specified by the user. Since the user usually has a concrete idea of the underlying problem structure, it is guaranteed that this information is made available to a particular heuristic and thus for the entire solution process. This also saves the usually time-consuming automatic generation of a decomposition. Finally, the simple transmission of a decomposition by means of a text file makes it easy for the user to try out many different decompositions and to select one which is convenient to solving an instance.

All heuristics presented in this publication were included for testing in the non-commercial solver SCIP and the complete source code is accessible to everyone. The results of the tests are comprehensible, as two of the three test sets used (Miplib, SCM, Cellphone), including decompositions, are publicly available.

The numerical results clearly show that using one of the proposed heuristics only makes sense if the provided instance and decomposition are constructed such that the respective heuristic can work on it efficiently. For DPS, it would be ideal if the linking constraints are designed in such a way that the blocks involved request a similar share of the resources (i.e. right-hand sides) in a good or optimal solution. The proposed equal distribution of the right-hand side for initialization of the partition then opens up good possibilities for DPS to find a beneficial solution within few iterations. If the equal distribution is not present or unknown, we suggest to use the LP relaxation to determine an initial partition. In contrast, PADM deals with decompositions with linking variables only. Since good solutions are typically sparse (see Figure 2) and we therefore initialize the values ξ with zero, the user ideally chooses a decomposition where it is expected that especially the linking variables are zero in a good solution. Initialization based on the LP solution may be preferable if this is not known or if relatively many variables are expected to be nonzero. For DKS, we observed no clear tendency towards a specific decomposition property leading to constantly good results. However, when the underlying problem shows binary and integer variables, but no continuous ones, the leverage of any available decomposition with DKS appears beneficial. Since a user typically deals with the same problem class over and over again (e.g. daily production schedule with varying demand), the user can easily find out which of the presented heuristics are appropriate and also choose a suitable decomposition.

Finally, two research tasks should be addressed which can be looked at based on the methods presented here. The extent to which parallel computing of individual blocks of a decomposition is beneficial is not examined, but it would be interesting to see how the heuristics may profit from using multiprocessor architectures or parallel computing clusters. DPS is directly parallelizable and for PADM a variant is presented in which the subproblems can be processed in parallel. However, the success of KS, and thus DKS, seems to depend on passing information about the (adapted) kernel of the current iteration to the solution of consecutive buckets. Solving unions of the kernel with each bucket in parallel therefore requires a restructuring of the algorithm which may affect its functionality.

As decompositions can be provided by simple but structured text files, the question arises whether machine learning approaches can distinguish between (dis)advantageous

decompositions. Even further, such approaches may be applicable to construct beneficial decompositions given some fixed problem type. In such context, the terms "(dis)advantageous" and "beneficial" surely depend on the specific setting which, for example, can be based on one of the decomposition heuristics described here.

Acknowledgments

The authors would like to thank SAP for its long-term financial as well as personnel support and for providing test instances. They also thank Gregor Hendel for implementing the management of decompositions in SCIP and generating decompositions for MIPLIB instances. Finally, the authors thank Alexander Martin and Gregor Hendel for reviewing a first draft.

References

[1] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

[2] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020. doi: 10.1287/ijoc.2018.0857. URL https://doi.org/10.1287/ijoc.2018.0857.

[3] E. Angelelli, R. Mansini, and M. G. Speranza. Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research*, 37(11):2017–2026, 2010. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2010.02.002. URL https://www.sciencedirect.com/science/article/pii/S0305054810000328.

[4] E. Angelelli, R. Mansini, and M. G. Speranza. Kernel search: A new heuristic framework for portfolio selection. *Computational Optimization and Applications*, 51(1):345–361, 2012. doi: 10.1007/s10589-010-9326-6. URL https://doi.org/10.1007/s10589-010-9326-6.

[5] G. D. Battista and R. Tamassia. Incremental Planarity Testing (Extended Abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 436–441, 1989. doi: 10.1109/SFCS.1989.63515. URL http://dx.doi.org/10.1109/SFCS.1989.63515.

[6] J. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962/63. URL http://eudml.org/doc/131533.

[7] M. Bergner, A. Caprara, A. Ceselli, F. Furini, M. E. Lübbecke, E. Malaguti, and E. Traversi. Automatic Dantzig–Wolfe reformulation of mixed integer programs. *Mathematical Programming*, 149(1):391–424, 2014. ISSN 1436-4646. doi: 10.1007/s10107-014-0761-5. URL http://dx.doi.org/10.1007/s10107-014-0761-5.

[8] T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013. ISSN 0167-6377. doi: https://doi.org/10.1016/j.orl.2013.08.007. URL https://www.sciencedirect.com/science/article/pii/S0167637713001181.

[9] T. Berthold. *Heuristic algorithms in global MINLP solvers.* PhD thesis, 2014. URL http://www.zib.de/berthold/Berthold2014.pdf.

[10] T. Berthold and G. Hendel. Shift-and-propagate. *Journal of Heuristics*, 21(1):73 – 106, 2015. doi: 10.1007/s10732-014-9271-0.

[11] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano,

Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. URL http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

[12] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. Enabling research through the scip optimization suite 8.0. *ACM Trans. Math. Softw.*, 49(2), jun 2023. ISSN 0098-3500. doi: 10.1145/3585516. URL https://doi.org/10.1145/3585516.

[13] R. Borndörfer, C. E. Ferreira, and A. Martin. Decomposing Matrices into Blocks. *SIAM J. Optim.*, 9(1):236–269, 1998.

[14] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, Jan. 2011. ISSN 1935-8237. doi: 10.1561/2200000016. URL http://dx.doi.org/10.1561/2200000016.

[15] V. Chvátal. *Linear programming*. A Series of books in the mathematical sciences. Freeman, New York (N. Y.), 1983. ISBN 0-7167-1195-8. URL http://opac.inria.fr/record=b1104676. Réimpressions : 1999, 2000, 2002.

[16] M. Conforti, G. Cornuejols, and G. Zambelli. *Integer Programming*. Springer Publishing Company, Incorporated, 2014. ISBN 3319110071.

[17] CPLEX. IBM ILOG CPLEX Optimization Studio CPLEX User's Manual, 2022. URL https://www.ibm.com/docs/en/icos/22.1.1?topic=optimizers-users-manual-cplex.

[18] G. B. Dantzig and P. Wolfe. Decomposition Principle for Linear Programs. *Oper. Res.*, 8(1):101–111, Feb. 1960. ISSN 0030-364X. doi: 10.1287/opre.8.1.101. URL http://dx.doi.org/10.1287/opre.8.1.101.

[19] Erlangen National High Performance Computing Center (NHR@FAU) . Woody throughput cluster (Tier3), last accessed on 2023-06-20. URL https://hpc.fau.de/systems-services/documentation-instructions/clusters/woody-cluster/.

[20] M. Fischetti, I. Ljubić, and M. Sinnl. Benders decomposition without separability: A computational study for capacitated facility location problems. *European Journal of Operational Research*, 253(3):557 – 569, 2016. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2016.03.002. URL http://www.sciencedirect.com/science/article/pii/S0377221716301126.

[21] G. Gamrath and M. E. Lübbecke. Experiments with a generic dantzig-wolfe decomposition for integer programs. In P. Festa, editor, *Experimental Algorithms*, pages 239–252, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7(4):367–398, 2015. ISSN 1867-2957. doi: 10.1007/s12532-015-0083-5. URL http://dx.doi.org/10.1007/s12532-015-0083-5.

[23] G. Gamrath, A. Gleixner, T. Koch, M. Miltenberger, D. Kniasew, D. Schlögel, A. Martin, and D. Weninger. Tackling industrial-scale supply chain problems by mixed-integer programming. *Journal of Computational Mathematics*, 37:866 – 888, 2019. doi: 10.4208/jcm.1905-m2019-0055.

[24] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020. URL http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

[25] B. Geißler, A. Morsi, L. Schewe, and M. Schmidt. Penalty alternating direction methods for mixed-integer optimization: A new view on feasibility pumps. *SIAM Journal on Optimization*, 27(3):1611–1636, 2017. doi: 10.1137/16M1069687.

[26] P. Gemander, W.-K. Chen, D. Weninger, L. Gottwald, A. Gleixner, and A. Martin. Two-row and two-column mixed-integer presolve using hashing-based pairing methods. *EURO Journal on Computational Optimization*, 8(3):205–240, Oct 2020. ISSN 2192-4414. doi: 10.1007/s13675-020-00129-6. URL https://doi.org/10.1007/s13675-020-00129-6.

[27] A. M. Geoffrion. *Approaches to Integer Programming*, chapter Lagrangean relaxation for integer programming, pages 82–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974. ISBN 978-3-642-00740-8. doi: 10.1007/BFb0120690. URL http://dx.doi.org/10.1007/BFb0120690.

[28] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 2021. doi: 10.1007/s12532-020-00194-3. URL https://doi.org/10.1007/s12532-020-00194-3.

[29] G. Guastaroba, M. Savelsbergh, and M. G. Speranza. Adaptive kernel search: A heuristic for solving mixed integer linear programs. *European Journal of Operational Research*, 263 (3):789–804, 2017. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2017.06.005. URL https://www.sciencedirect.com/science/article/pii/S0377221717305234.

[30] M. Guignard and S. Kim. Lagrangean decomposition: A model yielding stronger lagrangean bounds. *Mathematical Programming*, 39(2):215–228, Jun 1987. ISSN 1436-4646. doi: 10.1007/BF02592954. URL https://doi.org/10.1007/BF02592954.

[31] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

[32] C. Gutwenger and P. Mutzel. *Graph Drawing: 8th International Symposium, GD 2000 Colonial Williamsburg, VA, USA, September 20–23, 2000 Proceedings*, chapter A Linear Time Implementation of SPQR-Trees, pages 77–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44541-8. doi: 10.1007/3-540-44541-2_8. URL http://dx.doi.org/10.1007/3-540-44541-2_8.

[33] F. Harary and G. Prins. The block-cutpoint-tree of a graph. *Publicationes Mathematicae Debrecen*, 13:103–107, 1966.

[34] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Math. Program.*, 1(1):6–25, Dec. 1971. ISSN 0025-5610. doi: 10.1007/BF01584070. URL https://doi.org/10.1007/BF01584070.

[35] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973. doi: 10.1137/0202012. URL https://doi.org/10.1137/0202012.

[36] K. Halbig. Decomposition Heuristics, June 2023. URL https://github.com/khalbig/decomposition-heuristics. [dataset, software].

[37] G. Karypis and V. Kumar. Multilevel k-way Hypergraph Partitioning. In *In Proceedings of the Design and Automation Conference*, pages 343–348, 1998.

[38] G. Karypis and V. Kumar. hMetis: A Hypergraph Partitioning Package, Version 1.5.3, 1998.

[39] T. K. Ralphs and M. V. Galati. Decomposition in integer linear programming. In J. Karlof, editor, *Integer Programming: Theory and Practice*, pages 57–110. CRC Press, 2005. URL http://coral.ie.lehigh.edu/~ted/files/papers/DECOMP04.pdf.

[40] SAP SE. SAP Software Solutions — Business Applications and Technology, last accessed on 2023-06-20. URL https://www.sap.com.

[41] SAP SE or an SAP affiliate company. MILP Benchmarks CellphoneCo., Jan. 2023. URL https://github.com/SAP-samples/ibp-sop-benchmarks-milp-cellphoneco. [dataset].

[42] SAS/OR. 15.1 user's guide: Mathematical programming, 2018. URL https://support.sas.com/documentation/onlinedoc/or/151/decomp.pdf.

[43] L. Schewe, M. Schmidt, and D. Weninger. A Decomposition Heuristic for Mixed-Integer Supply Chain Problems. *Operations Research Letters*, 48(3):225–232, 2020. ISSN 0167-6377. doi: 10.1016/j.orl.2020.02.006. URL https://www.sciencedirect.com/science/article/pii/S0167637720300249.

[44] F. Soumis. *Decomposition and Column Generation. In Annotated Bibliographies in Combinatorial Optimization. Chapter 8*. A Wiley-interscience publication. Wiley, 1997. ISBN 9780471965749. URL https://books.google.de/books?id=jz4ZAQAAIAAJ.

[45] W. T. Tutte. Connectivity in graphs. *Mathematical Expositions*, 15, 1966.

[46] F. Vanderbeck and L. A. Wolsey. *Reformulation and Decomposition of Integer Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-540-68279-0. doi: 10.1007/978-3-540-68279-0_13. URL http://dx.doi.org/10.1007/978-3-540-68279-0_13.

[47] D. Weninger and L. A. Wolsey. Benders-type branch-and-cut algorithms for capacitated facility location with single-sourcing. *European Journal of Operational Research*, 2023. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2023.02.042. URL https://www.sciencedirect.com/science/article/pii/S0377221723001935.

[48] L. A. Wolsey. *Integer Programming: 2nd Edition*, pages 1–34. John Wiley and Sons, Ltd, 2020. ISBN 9781119606475. doi: https://doi.org/10.1002/9781119606475.oth1. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119606475.oth1.

[49] B. Yıldız, N. Boland, and M. Savelsbergh. Decomposition branching for mixed integer programming. *Operations Research*, 70(3):1854–1872, 2022. doi: 10.1287/opre.2021.2210. URL https://doi.org/10.1287/opre.2021.2210.

[50] Zuse Institute Berlin. MIPLIB 2017 – The Mixed Integer Programming Library, last accessed on 2023-06-20. URL https://miplib.zib.de/. [dataset].