Halbig, Katrin; Göß, Adrian; Weninger, Dieter

Exploiting user-supplied Decompositions inside Heuristics

Journal of Heuristics

Provided in Cooperation with:
Springer Nature

Suggested Citation: Halbig, Katrin; Göß, Adrian; Weninger, Dieter (2025) : Exploiting user-supplied Decompositions inside Heuristics, Journal of Heuristics, ISSN 1572-9397, Springer US, New York, NY, Vol. 31, Iss. 4,
https://doi.org/10.1007/s10732-025-09572-3

Journal of Heuristics (2025) 31:36
https://doi.org/10.1007/s10732-025-09572-3

Exploiting user-supplied Decompositions inside Heuristics

Katrin Halbig[1] · **Adrian Göß**[2] · **Dieter Weninger**[1,3]

Received: 24 May 2024 / Revised: 5 July 2025 / Accepted: 4 October 2025

Abstract
Numerous industrial fields, like supply chain management, face mixed-integer optimization problems on a regular basis. Such problems typically show a sparse structure and vary in size, as well as complexity. However, in order to satisfy customer demands, it is crucial to find good solutions to all such problems quickly. Current research often focuses on the development of a tailored approach for one specific problem class with a common structure. Information supplied by everyday users is usually overlooked, but may result in a decomposition with weakly connected blocks due to the structural sparsity. Hence, we present three heuristics to exploit decomposition information and analyze their value based on the type of decomposition. In particular, we newly introduce the heuristic Dynamic Partition Search, enhance the Penalty Alternating Direction Method published earlier in a basic form, and extend a framework from literature to Decomposition Kernel Search. All heuristics are implemented in the non-commercial solver SCIP. We examine their performance in a comprehensive computational study across three different test sets. The computational results indicate that knowledge about relevant decomposition information can boost the solution process of mixed-integer optimization problems.

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

✉ Katrin Halbig
katrin.halbig@fau.de

Adrian Göß
adrian.goess@utn.de

Dieter Weninger
dieter.weninger@fau.de

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

2 Analytics & Optimization Lab, University of Technology Nuremberg (UTN), Ulmenstr. 52i, 90443 Nuremberg, Germany

3 Department of Mathematics, Friedrich-Alexander-Universität Erlangen-Nürnberg, Cauerstr. 11, 91058 Erlangen, Germany

Published online: 22 October 2025

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

1 Introduction

Decomposition methods have been used for solving linear and mixed-integer problems for over half a century. The publications of Dantzig and Wolfe (1960) and of Benders (1962/63) can be regarded as important milestones in the entire field of research. In addition, Lagrangian relaxation (Geoffrion 1974) and its special case Lagrangian decomposition (Guignard and Kim 1987) have played an important role for many years. In Soumis (1997) a survey on different decomposition methods is given. Detailed descriptions of the most important decomposition algorithms with applications for mixed-integer problems are given in Ralphs and Galati (2005); Vanderbeck and Wolsey (2010); Wolsey (2020).

When modeling practical applications as mixed-integer problems, the constraint matrices are typically very sparse, that is, only few entries are nonzero. In particular, in MIPLIB 2017 (Gleixner et al. 2021), which comprises 1065 instances of mixed-integer problems arising from more than 400 distinct applications, the fraction of nonzero entries, the so-called density, ranges from 10^{-7} to 1 (100 %), but shows an average of less than 3 % (see Zuse Institute Berlin (2023) for more details). Such sparsity is particularly evident in mixed-integer supply chain models (Schewe et al. 2020). Sparse problems can usually be decomposed into loosely coupled blocks (Borndörfer et al. 1998), which opens the possibility for decomposition methods.

Besides taking advantage of sparsity for decomposition, it can be beneficial to leverage higher-level problem structures. There is a static and a dynamic way on how to proceed in this regard. In a static approach, certain variables and constraints of a model are permanently assigned to elements of an algorithm. A dynamic approach allows variables and constraints to be assigned to elements of an algorithm based on additional information. On the example of Benders decomposition (Benders 1962/63), the division in master and subproblem can be performed statically, which is well suited for specific problems such as capacitated facility location problems (Fischetti et al. 2016; Weninger and Wolsey 2023). The utilization of the problem structure is achieved here by fixing the decision variables in the master problem so that the remaining subproblem is a convenient flow problem to solve. In contrast, a dynamic implementation allows for a flexible allocation of variables and constraints to the master or subproblem on the basis of additional decomposition information. This approach is applied in software packages such as CPLEX (2022) and SCIP (Bestuzheva et al. 2021), which provide interfaces to communicate decomposition information, e.g., as for a Benders decomposition. We too will pursue the dynamic approach in this article.

One important component of efficient algorithms for solving mixed-integer linear problems is finding feasible solutions quickly or improving already existing solutions, for which heuristics play a central role (Berthold 2014). Accordingly, 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. We present three heuristics, whereby two of the heuristics are

construction heuristics and one heuristic can be used both as a construction heuristic and as an improvement heuristic.

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* can be used profitably in the solution process. In practice, a decomposition may be provided by the user. Such an approach has two advantages: First, the time-consuming computation of decomposition information is omitted and, second, the user typically has a very precise idea of the underlying structure of the problem and can communicate it appropriately on the basis of a decomposition to a particular heuristic. We focus on providing decompositions for heuristics in SCIP (Bestuzheva et al. 2021, 2023) to find solutions faster or to improve known solutions. In addition to the heuristics that are the focus of this work, the authors are only aware of one heuristic named *graph induced neighborhood search* (GINS), which was extended in Gamrath et al. (2020) to work with a decomposition.

Our contributions in this publication consist of the development of three heuristics that explicitly 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.

In the next section, we begin with an introduction of the definitions and notation used in the remainder of the article. Three sections follow, each with a heuristic exploiting decomposition information. In Section 3 and Section 4 we describe two construction heuristics, whereby the first is presented for the first time and the second is closely related to Lagrangian decomposition in the way it works. After that we describe in Section 5 an extension of the kernel search framework, where the aim is to find a subset of variables, the so-called kernel, which can be leveraged to find a proper solution. This approach can be used both as a construction heuristic and as an 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 article.

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 $x \in \mathbb{R}^n$, lower and upper bounds $\ell, u \in (\mathbb{R} \cup \{\pm\infty\})^n$, constraint matrix $A = (a_{ij})_{i,j} \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$, and objective direction $c \in \mathbb{R}^n$. Further, there exist integrality restrictions for all variables x_j with $j \in \mathcal{I} \subseteq [n], \mathcal{I} \neq \emptyset$, where $[n]{:=}\{1, \ldots, n\}$ for $n \in \mathbb{N}$.

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 $q \in [k]$ 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 $(D_q^{\text{row}}, D_q^{\text{col}})$. That is, nonzero entries are only allowed in a block or with row/column index in $L^{\text{row}}/L^{\text{col}}$. The special rows L^{row} and columns L^{col}, which may be empty, are called *linking rows* and *linking columns* or *linking constraints* and *linking variables*, respectively.

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 $I \subseteq [m]$ and $J \subseteq [n]$. Then, we can define the submatrices $A_q:=A_{[D_q^{\text{row}}, D_q^{\text{col}}]}$, $A_q^{\text{row}}:=A_{[L^{\text{row}}, D_q^{\text{col}}]}$, $A_q^{\text{col}}:=A_{[D_q^{\text{row}}, L^{\text{col}}]}$, and $A^{\text{row,col}}:=A_{[L^{\text{row}}, L^{\text{col}}]}$. Respective restrictions to the vectors x, c, b, ℓ, and u are defined analogously.

With this notation, 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 as

$$\begin{pmatrix} A_1 & 0 & \cdots & 0 & A_1^{\text{col}} \\ 0 & A_2 & 0 & 0 & A_2^{\text{col}} \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & A_k & A_k^{\text{col}} \\ A_1^{\text{row}} & A_2^{\text{row}} & \cdots & A_k^{\text{row}} & A^{\text{row,col}} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ x^{\text{col}} \end{pmatrix} \geq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \\ b^{\text{row}} \end{pmatrix}.$$

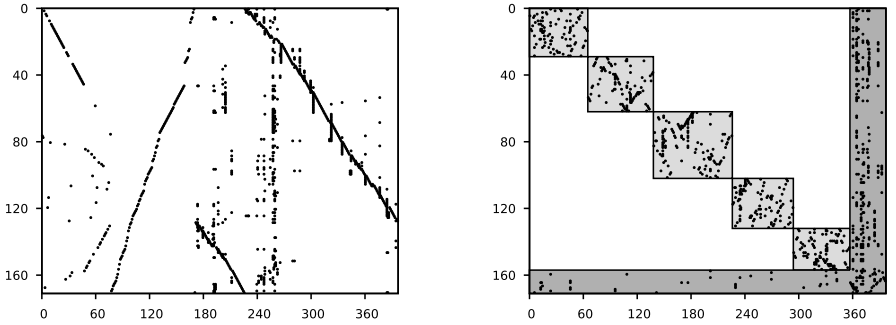
This representation of a matrix is also called *bordered block diagonal form* (Borndörfer et al. 1998).

An example for a constraint matrix and its rearranged representation is shown in Figure 1. Nonzero entries are visualized as black dots and are scattered wildly in the original arrangement, whereas the rearranged representation based on a decomposition shows a block diagonal with linking rows and linking columns.

We note that every matrix admits a trivial decomposition by setting $L^{\text{row}} = [m]$ and $L^{\text{col}} = [n]$, in which case $k = 0$. The other extreme occurs if $L^{\text{col}} = L^{\text{row}} = \emptyset$, in which case problem (1) breaks down to k decoupled subproblems. On the one hand, infeasibility of one such subproblem directly implies the infeasibility for (1), whereas on the other hand, corresponding subsolutions can be merged to one for (1).

3 Dynamic partition search

In the following we introduce a construction heuristic called *Dynamic Partition Search* (DPS). Without loss of generality, the absence of linking variables is assumed in this section. This is possible, because one could consider all linking variables as a separate block and afterwards recompute the partition of the constraints. Then, according to a decomposition, DPS splits a problem like (1) into independent subproblems, one



(a) Original arrangement of nonzeros.

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

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

for each block. Simultaneously, the linking constraints are also split according to the variables of each block. In an iterative scheme, DPS fixes the minimal share of these linking constraints in each subproblem to the overall right-hand side. In real-world applications, the right-hand side can represent, for example, the demands of different products that need to be supplied by several production plants (i.e., blocks) in total. After solving the resulting subproblems, either a solution can be returned or the partition of shares is dynamically updated, which led to the naming. This heuristic is inspired by an approach proposed by Yıldız et al. (2022) which searches for a partition that belongs to an optimal solution.

3.1 Definitions and reformulation

We start with definitions specific for DPS and necessary reformulations. Consider the mixed-integer problem (1) and a corresponding decomposition $\mathcal{D}$. To simplify the notation, for each block $q \in [k]$ we summarize all constraints, bounds and integrality conditions in

$$S_q := \left\{ x_q \;:\; A_q x_q \geq b_q,\ \ell_q \leq x_q \leq u_q,\ x_j \in \mathbb{Z} \text{ for } j \in \mathcal{I} \cap D_q^{\text{col}} \right\}.$$

Thus, we can rewrite problem (1) based on the decomposition $\mathcal{D}$ as

$$\min \quad \sum_{q \in [k]} c_q^\top x_q \tag{2a}$$

$$\text{s.t.} \quad x_q \in S_q, \qquad \forall q \in [k], \tag{2b}$$

$$\sum_{q \in [k]} A_q^{\text{row}} x_q \geq b^{\text{row}}. \tag{2c}$$

Note that problem (2) splits into k independent subproblems if we omit the linking constraints (2c).

In order to divide this problem into independent subproblems, a share of the right-hand side is required. Thus, we define a matrix $P = (p_1, \ldots, p_k) \in \mathbb{R}^{|L^{\text{row}}| \times k}$, called *partition*, with a share of $p_q \in \mathbb{R}^{|L^{\text{row}}|}$ for each block $q \in [k]$ fulfilling $\sum_{q \in [k]} p_q = b^{\text{row}}$. Hence, we can reformulate (2) with respect to a partition P and receive

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

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

$$A_q^{\text{row}} x_q \geq p_q, \qquad \forall q \in [k], \tag{3c}$$

$$\sum_{q \in [k]} p_q = b^{\text{row}}. \tag{3d}$$

Note that the p_q describe the partitioning of the right-hand side b^{row} between the single blocks. That is, the right-hand side is split between the blocks in order to fulfill constraint (3d).

We also point out that in practical scenarios typically not every linking constraint contains variables from every block. In order to keep the notation simple, we do not differentiate here, as one would eliminate the empty rows in (3c) and fix the corresponding entry in the partition to zero.

Assuming we know a partition $P^{\text{feas}} = (p_1^{\text{feas}}, \ldots, p_k^{\text{feas}})$ such that problem (3) is feasible, we can determine a corresponding solution, which is as well a solution of the original problem (1), by solving the k independent subproblems

$$\min \left\{ c_q^\top x_q \,:\, x_q \in S_q,\ A_q^{\text{row}} x_q \geq p_q^{\text{feas}} \right\}. \tag{4}$$

As we will explain in the next subsection, DPS aims to find such a partition by dynamically updating an initial guess.

3.2 Algorithm

Let $t = 0$ denote the iteration index to start with and P^t a chosen initial partition. There are no hard restrictions on the partition, but the maximal activity of each block q in each linking row i, that is, $p_{iq} \leq \sup\{(A_q^{\text{row}})_{i\cdot} x_q \,:\, \ell_q \leq x_q \leq u_q\}$ should be respected, in order to ensure possible feasibility. Nevertheless, one should keep in mind that the choice of the initial partition has a huge influence on the later solution. An initial partition can be chosen, for example, by splitting the right-hand side uniformly or, if available, based on an LP solution.

Then DPS checks whether P^t will lead to a feasible solution by solving k independent subproblems (4) until for each either a feasible solution is found or infeasibility is detected. If all of the k subproblems are feasible, a solution for problem (1) is constructed by the concatenation of the k subsolutions. Otherwise, P^t will not lead to a feasible solution for (1) and is therefore updated.

In order to receive a notion on how to perform such an update, modified subproblems are solved. In particular, slack variables $z_q \in \mathbb{R}_{\geq 0}^{|L^{\text{row}}|}$ are introduced for each block

$q \in [k]$, which are summarized in matrix $Z = (z_1, \ldots, z_k)$. These variables ensure feasibility of the constraints $A_q^{\text{row}} x_q \geq p_q^t$. The original objective function is replaced by a sum of the slack variables z_q weighted by a penalty parameter $\lambda^t \in \mathbb{R}_{>0}^{|L^{\text{row}}|}$. This leads for every block q to the subproblem

$$\begin{aligned} \min \quad & (\lambda^t)^\top z_q && \text{(5a)} \\ \text{s.t.} \quad & x_q \in S_q, && \text{(5b)} \\ & A_q^{\text{row}} x_q + z_q \geq p_q^t, && \text{(5c)} \\ & z_q \in \mathbb{R}_{\geq 0}^{|L^{\text{row}}|}. && \text{(5d)} \end{aligned}$$

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.

Given a block q, we note that subproblem (4) is feasible if and only if subproblem (5) has an optimal solution with optimal objective value zero. Hence, if (5) has a positive objective value for one block q, the partition and the penalty parameter are updated. For each single linking constraint i of (3d), we inspect the value of the slack variable z_{iq} in every block $q \in [k]$ 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 $\bar{z}_{i\cdot}^t$ of the slack variables is subtracted from the subpartition $p_{i\cdot}^t$ and—to keep constraint (3d) satisfied—the same amount is added to all p_{iq}^t with $\bar{z}_{iq}^t = 0$, i.e., for $q \in [k]$,

$$p_{iq}^{t+1} = \begin{cases} p_{iq}^t - \bar{z}_{iq}^t, & \text{if } \bar{z}_{iq}^t \neq 0, \\ p_{iq}^t + \frac{1}{v} \sum_{\beta \in [k]} \bar{z}_{i\beta}^t, & \text{if } \bar{z}_{iq}^t = 0, \end{cases} \tag{6}$$

   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, see Section 3.3 for details.
2. If all slack variable values $\bar{z}_{i\cdot}^t$ are greater than zero, only the penalty parameter λ_i^t is increased.
3. If all slack variable values $\bar{z}_{i\cdot}^t$ are zero, no update for constraint i is necessary.

If all coefficients of one row $(A_q^{\text{row}})_{i\cdot}$ and variables 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 (3d) is still satisfied.

After updating, the iteration index t is incremented and the subproblems (5) are solved again. This process is repeated until problem (3) has a feasible solution for the current partition, and thus a feasible solution for (1) is found, or until a maximum number T of iterations is reached. DPS is formally described in Algorithm 1.

We note that in the special case of two blocks and only one linking constraint, at most one update is necessary. This results from the simple form of the occurring objective in subproblem (5). In this special case the original problem can be

Algorithm 1: Dynamic Partition Search

Input: Initial partition P^0 fulfilling (3d), penalty parameters $\lambda^0 \in \mathbb{R}_{>0}^{|L^{\text{row}}|}$, and $T \in \mathbb{N}$.
Output: A feasible solution for problem (1) if one has been found.

```
1 for t = 0, 1, ..., T do
2     For each q ∈ [k], solve subproblem (5).
3     if all subproblems have an optimal objective value of zero then
4         Feasible solution for problem (1) found. Stop.
5     else
6         Update partition P^t and penalty parameters λ^t.
7     end
8 end
```

directly declared as infeasible if both slack variables are positive, but note that in the general case the problem is not necessarily infeasible if all slack variables are positive.

When setting up the subproblems (5), the original objective function is completely replaced by the weighted sum of the slack variables. This is necessary to push the slack variables to zero and thus have a proof for the feasibility of the original subproblems (4). However, the omission of the original objective function has the effect that often inferior solutions of (1) are found. To compensate this drawback we propose the following: If Algorithm 1 returned a feasible solution in step 4, we fix the partition in each subproblem (5) 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. As this may be time consuming and even suboptimal solutions suffice for an update of the partition, we propose to stop the solution process of each subproblem as soon as we know that its solution value will be strictly positive. Then, 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 have assumed that all linking constraints are of type '$\geq$'. In practice, constraints are often given as ranged ones, i.e., there is a left and a right hand side. For this case, we have implemented two partitions P_ℓ and P_r and calculate one common update vector for both sides to not lose any information.

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 cannot 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.

A basic version of DPS was published by the authors of the present article with SCIP version 8 (Bestuzheva et al. 2021). Since then the heuristic has been extended and improved according to this work.

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. This heuristic is not designed to handle linking constraints. However, the presence of linking constraints can usually be avoided by reassignment and relabeling. A detailed description of penalty alternating direction methods can be found in Geißler et al. (2017). For practical applications with a block-separable structure, such as supply chain management problems, this approach has proven to be successful (Schewe et al. 2020).

Classical alternating direction methods (ADMs) are extensions of Lagrangian type approaches (Boyd et al. 2011). In Geißler et al. (2017) 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. We use iteration index t for the penalty-loop and θ for the ADM-loop. Then, the subproblem of block $q \in [k]$ with penalty parameters $\mu > 0$ can be written as

$$\min \quad \sum_{j \in L^{\text{col}}} \sum_{\beta \in [k] \setminus \{q\}} \mu_j^{q,\beta,t,+} s_j^{\beta,+} + \mu_j^{q,\beta,t,-} s_j^{\beta,-} \tag{7a}$$

$$\text{s.t.} \quad A_q x_q + A_q^{\text{col}} x^{\text{col}} \geq b_q, \tag{7b}$$

$$x_j + s_j^{\beta,+} - s_j^{\beta,-} = \xi_j^{\beta,\theta}, \qquad \forall j \in L^{\text{col}}, \beta \in [k] \setminus \{q\}, \tag{7c}$$

$$\ell_j \leq x_j \leq u_j, \qquad \forall j \in L^{\text{col}} \cup D_q^{\text{col}}, \tag{7d}$$

$$x_j \in \mathbb{Z}, \qquad \forall j \in \mathcal{I} \cap (L^{\text{col}} \cup D_q^{\text{col}}), \tag{7e}$$

$$s_j^{\beta,+}, s_j^{\beta,-} \in \mathbb{R}_{\geq 0}, \qquad \forall j \in L^{\text{col}}, \beta \in [k] \setminus \{q\}, \tag{7f}$$

Algorithm 2: Penalty Alternating Direction Method

Input: Initial values for $\xi^{q,\theta}$ and penalty parameters $\mu^t > 0$, see Section 4.3.
Output: A feasible solution for problem (1) if one has been found.

```
Set t = 1, θ = 1.
while no feasible solution was determined do
    while no partial minimum was attained do
        for q = 1, ..., k do
            Solve subproblem (7) for block q.
            Update ξ_j^{q,θ} for all j ∈ L^col in all subproblems of blocks β ∈ [k] \ {q}.
        end
        Set θ ← θ + 1.
    end
    Choose new penalty parameters μ^{t+1} ≥ μ^t, see Section 4.3.
    Set t ← t + 1.
end
```

where the slack variables $s_j^{\beta,+}$ and $s_j^{\beta,-}$ represent the difference between two copies of linking variable x_j. Variable $s_j^{\beta,+}$ closes the gap upwards, and $s_j^{\beta,-}$ closes the gap downwards. The right-hand side $\xi_j^{\beta,\theta}$ represents the solution value of the linking variable in the other block β, which contains a copy of x_j, and needs to be initialized beforehand.

Analogous to DPS, we assume that every linking variable occurs in every block, thereby simplifying notation, although this does not occur commonly in practice. Actually, one would not insert a copy of a linking variable if the variable does not already occur in this block.

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. If the subproblem belonging to block q has been solved, the values $\xi_j^{q,\theta}$ for all $j \in L^{\text{col}}$, $\beta \in [k] \setminus \{q\}$ are updated with the determined solution values of the linking variables x_j from block q. 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 (7) the original objective function $c_q^\top x_q + (c^{\text{col}})^\top x^{\text{col}}$ is replaced by a penalty term. This approach has the advantage that PADM usually converges faster to a feasible solution than if the original objective function is taken into account in addition to the penalty term. As with DPS, one can add a reoptimization step (see end

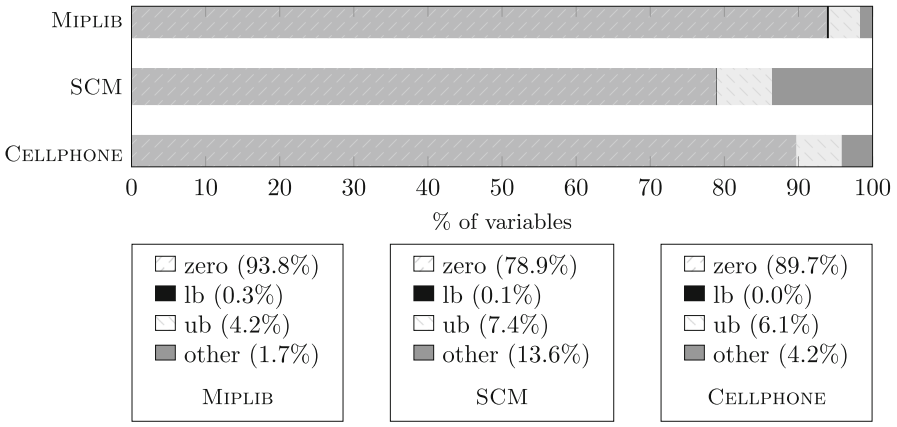


Fig. 2 The majority of variables are zero in a best or optimal solution

of Section 3.2) in which the linking variables are fixed in order to subsequently take the objective function into consideration.

4.3 Implementation details

Before executing PADM, there is to decide how to initialize the values $\xi_j^{q,\theta}$. 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 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. In addition, bounds for the support of integer and mixed integer optimal solutions were shown in Aliev et al. (2018), which indicate that optimal solutions are sparse. Thus it is reasonable to initialize $\xi_j^{q,\theta}$ with zero.

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 by setting $\mu^t = 1$ for $t = 1$ and increase them after each ADM-loop by a factor of ten if the corresponding slack variable 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. (2020). 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.

Note further that subproblems (7) can be warmstarted by using the solution from a previous ADM iteration. The values for variables x are adopted directly and the values for the slack variables are adjusted accordingly.

A basic version of PADM was published with SCIP version 7 (Gamrath et al. 2020) and the extended version as presented in this article was published with SCIP version

8 (Bestuzheva et al. 2021), both by the authors of the present article. The code of the latter is also identical to the code used for the computational results in Section 6.2.

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 (Angelelli et al. 2012) and (multi-dimensional) knapsack problems (Angelelli et al. 2010). Later on, it was refined as in Guastaroba et al. (2017), 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 Guastaroba et al. (2017) 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.

For ease of explanation, we assume the upper bound of variables to be nonnegative. This is without loss of generality, since the variable domain can be shifted while receiving a sole reformulation of the original MIP.

5.1 Kernel search framework

In Guastaroba et al. (2017) 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 current kernel is united with one of the buckets and the original problem is restricted to this union. That is, every integer variable not in this union is fixed to zero or its lower bound and then the resulting problem is solved. Depending on the resulting solution, the kernel is updated before combining it with the next bucket. The idea of 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. For the remainder of the section, we refer to a solution value of a variable which is neither zero nor at the lower bound as *non-trivial*; otherwise *trivial*.

We give 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 $N_b \in \mathbb{N}$ 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

$$\begin{aligned} \text{MIP}(J) := \min\{c^\top x \ : Ax \geq b,\ l \leq x \leq u, \\ x_j = \max\{0, l_j\} \text{ for } j \in \mathcal{I} \setminus J,\ x_j \in \mathbb{Z} \text{ for } j \in J\}. \end{aligned} \tag{8}$$

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

Algorithm 3: Kernel Search

Input: Problem (1) with $z^{\text{inc}} \in \mathbb{R} \cup \{+\infty\}$.
Output: Solution x^{new} with $z^{\text{new}} \leq z^{\text{inc}}$ for problem (1) if one has been found.
1 Set $z^{\text{new}} \leftarrow z^{\text{inc}}$.
2 Determine a kernel $K \subseteq \mathcal{I}$.
3 Divide $\mathcal{I} \setminus K$ into buckets $(\mathcal{B}_i)_{i=1,\ldots,N_{\text{b}}}$ with $\mathcal{B}_1 = \emptyset$ and $\mathcal{B}_i \neq \emptyset$ for $i > 1$.
4 **for** $i = 1, \ldots, N_{\text{b}}$ **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($K \cup \mathcal{B}_i$) *is infeasible* **then**
7 Continue with next iteration.
8 **else if** $c^\top \tilde{x} \leq z^{\text{new}}$ **then**
9 Set $x^{\text{new}} \leftarrow \tilde{x}$ and $z^{\text{new}} \leftarrow c^\top \tilde{x}$.
10 Add elements from $\mathcal{B}_i$ with non-trivial value in $\tilde{x}$ to K.
11 Delete elements from K with trivial value in $\tilde{x}$.
12 **end**
13 **end**

5.2 Extension to decomposition kernel search

For the determination of the initial kernel K in step 2 of Algorithm 3, the authors of Angelelli et al. (2010, 2012) 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. As 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, we include it as a preferred reference to determine activity.

The basic KS framework presented in Section 5.1 does not incorporate decomposition information. Hence, in order to make use of it, we propose several extensions, starting with the construction of the initial kernel and the respective buckets. Our observed solutions typically show active variables in (almost) all blocks and (almost) all variable types. Hence, in order to respect this fact and keep the solution process of

the restricted problems fast, we construct kernel/buckets for each block as well as variable type and unite afterwards. Note that an additional split regarding the continuous variables is in contrast to the approaches in Angelelli et al. (2010, 2012); Guastaroba et al. (2017), where the kernel is considered to consist of binary/integer variables only.

In particular, consider the qth block, and variable type τ abbreviated with *bin*, *int*, *con* for binary, pure integer, and continuous variables, respectively. Then, for a solution x^* to (a relaxation of) problem (1), we define the binary/integer/continuous *sub-kernel* (of block q) as the index set $K_q^\tau := \{j \in D_q^{\text{col}} \mid x_j = x_j^\tau,\ x_j \text{ active w.r.t. } x^*\}$, respectively. The resulting initial sub-kernel of block q and the overall initial kernel are constructed as simple unions of these sets. Note that, in contrast to the basic KS framework, it is $K \subseteq [n]$. Hence, in (8) we have to replace $\mathcal{I}$ by $[n]$.

Although seeming exaggeratedly complicated on first sight, we construct the buckets analogously for each variable-type-block combination and unite afterwards. In particular, this results in buckets $\mathcal{B}_i$, $i = 1, \ldots, N_{\text{b}}$, as unions 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*. This constitutes a crucial difference to the bucket definition in KS. Identifying the relevant subset per block first and uniting second forces every block to contribute to a bucket. This may not be the case for applying basic KS.

Now, we address the solution to the restricted problems (see step 5 of Algorithm 3). Similar as in Angelelli et al. (2010, 2012), we add a constraint to enforce the activation of at least one binary or pure integer variable with index in the current bucket by

$$\sum_{j \in \mathcal{B}_i^{\text{bin}} \cup \mathcal{B}_i^{\text{int}}} x_j \geq \left(\sum_{j \in \mathcal{B}_i^{\text{int}}} l_j \right) + 1. \tag{9}$$

Note that we excluded continuous variables here due to scaling effects, compare (Chvátal 1983). This enables us to formulate the entire *Decomposition Kernel Search (DKS)* framework; see Algorithm 4.

5.3 Implementation details

It remains to clarify the definition of the multi-level buckets resulting from sub-kernels. Following (Angelelli et al. 2012), we first find a number of buckets and use this to split the variables that are not in the kernel into equally sized buckets (possibly except for the last one). 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 of number of kernel to non-kernel variables averaged over block and variable type. It is noteworthy that we also excluded the continuous variables in this calculation, as their typically diminishing ratio of active to inactive variables by constructing the kernels leads to very small buckets.

Each bucket induces a subproblem in DKS, which is solved. We have limited the heuristic to use a maximum of ten percent of the total time allocated to SCIP for solving such a problem. Note that this adds a non-deterministic component to the

Algorithm 4: Decomposition Kernel Search

Input: Problem (1) with decomposition $\mathcal{D}$ and $z^{\text{inc}} \in \mathbb{R} \cup \{+\infty\}$.
Output: Solution x^{new} with $z^{\text{new}} \leq z^{\text{inc}}$ 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 (9) 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) is infeasible then
        Continue with next iteration.
    end
    Set x^new ← x̃ and z^new ← c^⊤ x̃.
    Add elements from B_i with non-trivial value in x̃ to K.
    Delete elements from K with trivial value in x̃.
end
```

implementation of DKS. We consider this justified to avoid the heuristic consuming the entire runtime if solving the first/root node of a subproblem is already problematic. At the same time, we assume that this limit is sufficiently large to reflect any potential negative impact of DKS on the overall solution process.

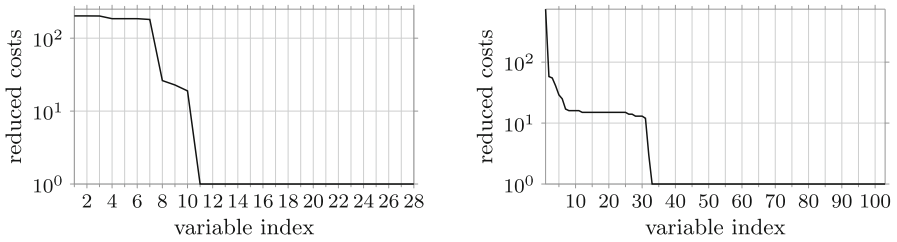
Finally, we would like to point out two extensions of the procedure presented. The first extension is based on an observation on the values of the reduced costs and the second extension tries to realize a balance between solution time and solution quality by an adaptive *MIP-gap control*.

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 Figure 3 we visualize the situation for the pure integer and continuous variables of an instance from the SCM test set. 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 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., Angelelli et al. (2010). In our case, this can be applied analogously with low reduced costs due to the minimization problem. It may imply that variables of one group serve similar purposes and, thus, should be investigated simultaneously in order to choose the "best" one.

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 $r^{\max}_{q,\tau}$. The ith sub-bucket $\mathcal{B}^{\tau}_{i,q}$ then contains the indices of variables with reduced costs in the interval $((r^{\max}_{q,\tau})^{(i-1)/N_{\text{b}}}, (r^{\max}_{q,\tau})^{i/N_{\text{b}}}]$. The final multi-level buckets $\mathcal{B}_i$ again result from uniting. 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.



(a) Reduced costs for pure integer variables. (b) Reduced costs for continuous variables.

Fig. 3 Reduced costs for the variables of an instance from SCM test set

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. 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 $\Delta_{\max} \geq 0$, and a factor of $\delta = 1$. If a problem reaches the time limit, we increase Δ for the consecutive problems by $\delta\Delta_{\max}/(N_\mathrm{b}-1)$. Therefore, if all problems hit the time limit, the last bucket is solved for the maximal gap $\Delta_{\max}$. 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.

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 (Halbig 2023).

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 Erlangen National High Performance Computing Center (NHR@FAU) (2023) for more details. The optimization problems are solved by using SCIP 8.0.0 (Bestuzheva et al. 2021) linked with SoPlex 6.0.0 (Bestuzheva et al. 2021) as LP solver.

We used a time limit of 20 minutes. To avoid interactions between the presented algorithms, in runs corresponding to one heuristic the other two heuristics were deactivated. Furthermore, we have deactivated the only other decomposition heuristic in

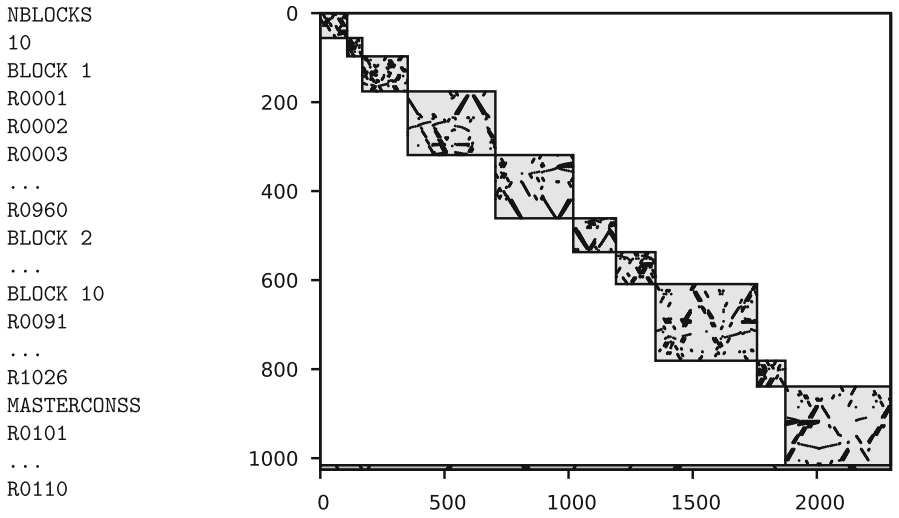


Fig. 4 Example of `.dec`-format with rearranged constraint matrix for the MIPLIB 2017 instance `binkar10_1` with decomposition from Zuse Institute Berlin (2023)

SCIP, namely GINS (Gamrath et al. 2020), which, however, is not within the scope of this article. 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

Decompositions for problem instances can be created inside a solver itself. Widespread and efficient methods for this are, for example, specialized graph (Battista and Tamassia 1989; Borndörfer et al. 1998; Gutwenger and Mutzel 2001; Harary and Prins 1966; Hopcroft and Tarjan 1973; Tutte 1966) or hypergraph partitioning methods (Karypis and Kumar 1998a, b). In practice, however, the user of optimization software is typically aware of details about the model and the underlying problem structure which can generate an appropriate decomposition. Naturally, such a decomposition may be much more useful in terms of saved solving time than a solver-created one. Even further, one saves computation time when it comes to the creation of the decomposition.

SCIP 7.0 (Gamrath et al. 2020) 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, for example, as a separate `.dec`-file. An example of the `.dec`-format with associated rearranged constraint matrix A is shown in Figure 4. Such a file starts with a section about the number of blocks and continues with one section for each block listing the associated row names. Finally, the special section `MASTERCONSS` contains all rows belonging to the linking constraints L^{row}. 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.

During the solution process the original problem (1) may be changed, for example, by deleting rows or fixing columns in presolve (Achterberg et al. 2020; Gamrath et al. 2015; Gemander et al. 2020). 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 Gamrath et al. (2020).

Test sets

We consider three sets of instances. The first set of instances is based on the MIPLIB 2017 (Gleixner et al. 2021; Zuse Institute Berlin 2023) 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 Zuse Institute Berlin (2023) (denoted by "miplib2017"). The other three types are generated by GCG (Gamrath and Lübbecke 2010) version 3.0.2 using different parameters and are accessible at Halbig (2023). 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 only trivial decompositions 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 SE (2023). The most important components are stock keeping, capacity restrictions, transport, production and demand fulfillment. A more detailed description can be found in Gamrath et al. (2019). Since these instances contain many independent components (see Gamrath et al. (2015)), we selected only non-trivial integer and mixed-integer components. In our case, a component is classified as 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 four different economic aspects. These are time buckets (B), organizational units of the supply network (locations, L), products (P) which can be raw materials as well as end products, and 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 "max"). 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, given by $\{B, L, P, S\} \times \{0, 2, 4\}$. After removing obvious duplicates and decompositions with only one block we get a test set of 322 instance-decomposition combinations.

The third set contains artificially 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

Table 1 Primal-dual gap. If instances are solved to optimality, SGM's of solving time with a shift of 1 are given in brackets

Test set	#instances with primal-dual gap			
	$> 10\,\%$	$\leq 10\,\%$	$= 0.0\,\%$	
MIPLIB (#694)	272	177	245	(74.2 s)
SCM (#322)	163	130	29	(148.0 s)
CELLPHONE (#504)	441	63	0	

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 SE (2023) and is publicly available at SAP SE or an SAP affiliate company (2023). 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.

Despite the absence of *user-supplied* decompositions for MIPLIB, which partly contradicts the title of this article, we consider testing on it to be valuable for two reasons. First, it provides an insight into the performance of our heuristics beyond the field of supply chain management. Second, the instances and the corresponding decompositions are publicly available—unlike the SCM test set—such that others can draw comparisons with respect to our results.

Table 1 provides measurements for the difficulty of solving the instances grouped by test set. We consider the comparison run without presented heuristics and examine the primal-dual gap at the end of the solving process (maximal 20 minutes). While none of the instances in CELLPHONE can be solved to optimality within the time limit, test set MIPLIB in particular also contains instances that can be solved, namely 245 instances with an average solution time of 74.2 seconds.

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 (2013). The best known solution for an instance as base value to determine a primal integral is given by publicly available solutions (see Zuse Institute Berlin (2023)) and/or by results of previous and for this article used runs.

For comparing performance on sets, we compute the shifted geometric mean (SGM) of these values with a shift value of 1. In contrast to the arithmetic mean, the shifted geometric mean has the advantage that extreme values do not have such a strong influence on the average. For further discussion on evaluating computational results with SGM see for example (Achterberg 2007). For a broader analysis, the aggregation operators max ("worst"), min ("best"), and SGM are applied on instance level, i.e., across all available decompositions, before summarizing those again with the SGM.

For every run featuring a heuristic and setting, the SGM of the primal integrals is divided by the one corresponding to the comparison run. This reduces the comparison to a single ratio, where a value of less than 1 indicates an improvement in performance. Further, it allows to quantify such an improvement in percent.

To get more meaningful results and not dilute them, we clean our data first. For each heuristic separately, we remove an instance-decomposition combination for all settings used for that heuristic if any of the following occurs: (i) SCIP has encountered problems that are beyond the control of the heuristic for one setting; (ii) the heuristic was not called at least for one setting; (iii) the primal integral is less than 10^{-4} for all settings. On instance-decomposition combinations with property (ii) we 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.

6.1 Dynamic partition search

In the following we analyze the performance of DPS. We investigate two different times at which to call the heuristic—right after the presolving process, i.e., pre-root (`default`) and right after root node computation (`lp`). In the first case, the initial partition is computed by a uniform split of the right-hand side, in the second case, the LP solution is available and thus the initial partition bases on it. We run each case with (`reopt` or `lp_reopt`) and without reoptimization. Thus we consider four different settings.

First, we examine the number of times DPS was called on the three different test sets, and the number of times DPS successfully found a feasible solution. These results are given in Table 2. An instance-decomposition combination is counted in column "Called" if the heuristic was called at least once on it. Note that DPS can get called several times on one instance-decomposition combination, as SCIP performs restarts, which abort the solution process and start it completely from scratch, reusing information from the previous pass. In column "Found" all instance-decomposition combinations are counted on which DPS successfully found at least one feasible solution.

Having a closer look on Table 2, running pre-root DPS gets called more often than after root. For example, on MIPLIB running pre-root DPS could get called on 338 out of 694 instance-decomposition combinations and after root on 76 out of 694. Due to general restrictions it is not possible to call DPS in every case. For instance, the original problem is transformed during the solving process of SCIP and new constraints may be added that are not representable as a single linear constraint (such as `orbitope` constraints). If such constraints are linking ones, DPS cannot execute appropriately. Another restriction is a high estimated memory usage that could exceed the memory limit due to a high number of blocks. Here, 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. Comparing pre- to after-root runs, the solving process for the latter might simply not have reached the calling point.

Table 2 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

In 76 out of these 338 cases on MIPLIB DPS successfully found a feasible solution. Activating the reoptimization step there are an additional 22 instance-decomposition combinations. As for the figures in the "Found" column, we note that the reoptimization step can only find a solution if DPS itself has already found one before. However, the first solution may occasionally not be accepted by SCIP because its objective is considered infinite or the same solution has already been found by another heuristic.

In the following, we evaluate the performance of the heuristic. Besides cleaning the data as described in the introduction to this section, we focus on subsets of instances for which all decomposition types or all numbers of blocks are available. This allows for an in-depth analysis regarding the suitable choice of a decomposition when using DPS.

On test set MIPLIB, after cleanup 46 instances that have all four decompositions are left. Unfortunately, in general no performance improvement can be measured, even if the best decomposition is selected. An improvement can be observed only on some particular instances. For the sake of completeness, the specific values can be found in Table 10 in Appendix A.1. There, a similar pattern shows in Table 11.

Turning to SCM, we note that after cleanup, 182 out of 322 instance-decomposition combinations are left, belonging to 21 individual instances. In Table 3, we display the corresponding results by decomposition type, comparing between block counts. In general, all settings seem to have a negative influence on the solution process when considering decomposition types B or S, compare Table 3a and Table 3d. The situation changes when focusing on setting `default` with decomposition type L (Table 3b) or on `lp` with decomposition type P (Table 3c). Here, improvements of up to 12% can be observed, which increases to up to 18% when implying the best decomposition as known. We also highlight the favor towards fewer blocks. The minor worsening caused by the reoptimization step is potentially dedicated to the additional time required for it. The high-level analysis given in Tables 12 and 13 in Appendix A.1 cannot confirm this effect, as it lacks the distinction by block count.

Table 3 Results for DPS on SCM, grouped by decomposition type. Including only instances with decompositions yielding all three block counts: 2, 4, and max

	2	4	max	best
(a) decomposition type B, #19 instances				
`default`	1.29	1.61	1.67	1.24
`reopt`	1.31	1.65	1.67	1.26
`lp`	1.02	1.08	1.19	1.01
`lp_reopt`	1.03	1.08	1.20	1.01
(b) decomposition type L, #10 instances				
`default`	**0.88**	**0.96**	**0.98**	**0.82**
`reopt`	**0.91**	**0.99**	1.00	**0.84**
`lp`	1.00	1.02	1.28	1.00
`lp_reopt`	1.02	1.03	1.28	1.02
(c) decomposition type P, #13 instances				
`default`	1.54	1.48	1.52	1.47
`reopt`	1.56	1.44	1.44	1.34
`lp`	**0.95**	**0.93**	1.24	**0.93**
`lp_reopt`	**0.96**	**0.96**	1.24	**0.95**
(d) decomposition type S, #11 instances				
`default`	2.07	1.70	1.86	1.62
`reopt`	1.85	1.70	1.87	1.63
`lp`	1.29	1.19	1.33	1.19
`lp_reopt`	1.30	1.20	1.33	1.19

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 4 Results for DPS on CELLPHONE, grouped by decomposition type. Including only instances with decompositions yielding all three block counts: 2, 4, and max

	2	4	max	best
(a) decomposition type B, #56 instances				
`default`	1.93	1.99	1.89	1.85
`reopt`	2.00	2.05	**0.99**	**0.97**
`lp`	1.06	1.05	1.03	1.03
`lp_reopt`	1.07	1.05	1.03	1.03
(b) decomposition type L, #56 instances				
`default`	1.95	2.01	1.86	1.84
`reopt`	1.36	1.40	**0.82**	**0.80**
`lp`	1.08	1.07	1.02	1.00
`lp_reopt`	1.08	1.05	1.02	**0.99**
(c) decomposition type P, #56 instances				
`default`	2.02	2.13	1.91	1.72
`reopt`	2.01	1.81	**0.82**	**0.75**
`lp`	1.39	1.64	1.03	1.01
`lp_reopt`	1.41	1.64	1.03	1.01

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 5 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

Finally, we investigate the performance on CELLPHONE. After cleanup, still all 504 instance-decomposition combinations are left, belonging to 56 individual instances. The results in Table 4 indicate that running DPS after-root (`lp` and `lp_reopt`) is not beneficial. This is even intensified when running pre-root (`default`). Interestingly, however, the reoptimization step in this case enforces a large improvement of up to 18% when combined with the decomposition in the maximal number of blocks. In combination, we deduce that finding solutions to max-block decompositions pre-root with DPS is hindering the solution process of SCIP unless a reoptimization step is invoked. This is confirmed in Tables 14 and 15 in Appendix A.1.

In summary for all test sets, the performance of DPS depends strongly on the structure and type of the used decomposition. Whereas DPS is ineffective on MIPLIB, it can show significant performance improvement on the supply chain management problems in SCM and CELLPHONE when combined with the right block count. Especially, running DPS pre-root with the reoptimization step combined with decompositions showing a naturally maximal number of blocks is expected to support the solution process.

6.2 Penalty alternating direction method

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

In Table 5 we observe how often PADM was called and found a solution. This table is structured in the same way as Table 2 for the results of DPS. For MIPLIB and

Table 6 Results for PADM on SCM, grouped by decomposition type. Including only instances with decompositions yielding all three block counts: 2, 4, and max

	2	4	max	best
(a) decomposition type B, #7 instances				
`default`	**0.89**	**0.90**	5.01	**0.88**
`reopt`	**0.95**	**0.93**	5.04	**0.91**
`lp`	1.00	1.00	2.26	1.00
`lp_reopt`	1.03	1.01	2.25	1.00
(b) decomposition type L, #9 instances				
`default`	**0.87**	**0.88**	**0.89**	**0.86**
`reopt`	**0.89**	**0.90**	**0.90**	**0.87**
`lp`	1.14	1.06	**0.93**	**0.93**
`lp_reopt`	1.16	1.08	**0.94**	**0.94**
(c) decomposition type P, #9 instances				
`default`	**0.99**	1.00	**0.99**	**0.98**
`reopt`	1.00	1.01	1.00	**0.98**
`lp`	1.01	1.01	1.22	1.00
`lp_reopt`	1.01	**0.98**	1.23	**0.97**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

SCM our heuristic is able to find a feasible, accepted solution in 50 to 70 percent of all instances 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 ("max"), and for all decompositions running after-root, the heuristic could not get called at all. Here, the reassignment of linking constraints to one block failed. In principle, this issue can be overcome by tailoring PADM to these instances. But, as our implementation is supposed to be applicable to MIPs in general, this is out of scope for the present work.

Now, we evaluate the performance of PADM. The results are structured in the same way as for the evaluation of DPS in Section 6.1 and the data is also cleaned up beforehand.

On MIPLIB 124 instance-decomposition combinations belonging to 52 individual instances are left after cleanup. Unfortunately, no significant performance improvement can be measured, even if the best decomposition is selected. With setting `lp` or `lp_reopt` the ratios are nearly neutral, but this is also due to the fact that PADM could get called less often. See Tables 16, 17 in A.2 for the concrete figures.

The results for test set SCM regarding instances with all three block counts and grouped by decomposition type are given in Table 6. Note that there exist no such instances with respect to decomposition type S.

Starting with Table 6c, PADM seems to have a neutral to slightly positive impact when called with decompositions of type P. For PADM running pre-root, we observe a performance boost of up to 13% independent of the number of blocks in a decomposition of type L, see Table 6b. However, running it after the root, only the natural maximal block decomposition (max) seems to allow for improvement. This is in con-

trast to the figures in Table 6a, where across all settings this kind of decomposition has a huge negative effect when combined with PADM. To be fair, one must take the low number of 7 and 9 instances into account, but the results for decomposition types B and L are quite inverse in this regard. The overall tendency of PADM to perform better when called pre-root is also confirmed by the broader analysis, see Tables 18 and 19 in Appendix A.2.

Finally, we take a look at the results for CELLPHONE. PADM could not get called at all after the root node computation and for decompositions with more than four blocks, but when it was called, it found a feasible solution in all cases. However, we cannot observe a significant improvement in performance for one of the considered (sub-)sets. The concrete values and a possible explanation for this behavior can be found in Tables 20 to 22 in Appendix A.2.

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 want to compare our extensions made to the standard KS framework as presented in Algorithm 3. That is, we define the setting `dks` which differs from `ks` by using decomposition information, sorting the bucket variables logarithmically by reduced costs, and adding the constraint (9), as well as an objective cutoff.

As DKS was implemented inside the SCIP framework, we fixed two decisive SCIP-specific parameters for both settings 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 if found instead of the LP-relaxation. Second, we forced SCIP to call the heuristic 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 also tried to leverage the character of KS and DKS as a construction heuristic, i.e., for finding a feasible solution at the start which the remaining solving process can work with.

In similar fashion to Table 2 and Table 5, we compare calls of and feasible solutions found by DKS. As we see in Table 7, both settings were 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 skipping the heuristic. In terms of found solutions, `dks` shows a significantly lower number. We attribute this to the existence of the objective cutoff constraint. This does not imply that `dks` fails to find solutions, instead it only focuses on strictly improving ones. An extreme of this

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

Test set	Setting	#Called	#Found
MIPLIB (#694)	dks	376	178
	ks	379	262
SCM (#322)	dks	121	27
	ks	121	75
CELLPHONE (#504)	dks	359	0
	ks	360	270

phenomenon can be observed regarding the CELLPHONE test set. As explained later on, dks seems to not find improving solutions due to the problem type of the instances in the test set.

We follow the organization from previous subsections and, hence, consider the performance of DKS. Further, all instances were cleaned according to the explanation in the beginning of the current section. However, the figures received across all test sets when comparing by decomposition type are ambiguous (compare Tables 23 and 30 in Appendix A.3). Now, as KS was initially proposed for binary problems only, we present a distinction by problem—not decomposition—type.

Hence, 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.

Looking at Table 8 for MIPLIB, both settings have a slight or significant negative influence on the solution process, when run on MBPs or PIPs, respectively. For the last problem class, however, the statement is bound to the small number of instances and thus not representative. Turning to Table 8c, dks worsens the performance a little more than ks. We account this effect to the construction of the subproblems, which is probably more complicated to solve when respecting three types of variables. On BPs, the original problem class to tackle with ks, the latter achieves an improvement of 2%, even if the worst possible decompositions are selected. Setting dks seems to have a neutral influence, see Table 8a. Lastly, coming to IPs in Table 8b, the extensions in DKS seem to take effect. While ks has a minor negative impact, dks shows improvements by up to 4%, when considering the best decomposition, in comparison to default SCIP. Here as well, even the worst possible decomposition leads to a 2% improvement. We emphasize that the reason for this result lies in the application of logarithmically reduced cost sorting, as well as the multi-level structured buckets. Both try to handle binary and integer variables according to their use in the problem.

On the SCM test set the performance of DKS over all decompositions, as displayed in Table 9, is comparable to not using such a heuristic on MBPs with a slight improvement when knowing the best decomposition. With general MIPs, in contrast, they are treated nearly neutral by dks and significantly worse by ks. It seems that dks cancels

Table 8 Results for DKS on MIPLIB, grouped by problem type. Including all instances with at least one decomposition

setting	worst	sgm	best
(a) problem type BP, #34 instances			
dks	1.01	1.00	1.00
ks	**0.98**	**0.98**	**0.98**
(b) problem type IP, #33 instances			
dks	**0.98**	**0.96**	**0.94**
ks	1.02	1.02	1.01
(c) problem type MBP, #110 instances			
dks	1.06	1.04	1.02
ks	1.02	1.01	1.01
(d) problem type MIP, #32 instances			
dks	1.43	1.33	1.22
ks	1.07	1.07	1.07
(e) problem type PIP, #6 instances			
dks	1.36	1.31	1.31
ks	1.28	1.31	1.30

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 9 Results for DKS on SCM, grouped by problem type. Including all instances with at least one decomposition

setting	worst	sgm	best
(a) problem type MBP, #11 instances			
dks	1.02	1.00	**0.99**
ks	1.00	1.00	**0.99**
(b) problem type MIP, #22 instances			
dks	1.01	1.01	1.00
ks	1.32	1.31	1.31

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

the solution process early enough due to the constraint (9) such that its performance is comparable.

The CELLPHONE test set constitutes of general MIP instances only. This makes a presentation of the results split by problem-type unnecessary. For the other two test sets, we could already observe that general MIPs are hard to tackle for kernel search frameworks which shows again here (compare Tables 29 and 30 in Appendix A.3).

In conclusion, the performance of DKS depends more on the problem type than on the type of decomposition. We observe promising results without the use of decomposition information on binary problems, for which the base framework KS was originally introduced (Angelelli et al. 2012). In contrast, the heuristics have difficulties on mixed-/pure-integer problems. However, for problems consisting only of binary and pure integer variables, the extensions introduced in DKS enable a significant performance

improvement compared to KS and default SCIP even considering the worst possible decomposition.

7 Conclusion

In this article, we propose three heuristics exploiting decompositions which are considered to be provided by the user. We assume that this gives additional information about the underlying problem structure and time-consuming automatic computation is omitted. 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 tackle a specific instance.

All methods presented are included in the non-commercial solver SCIP for testing and the complete source code is made accessible. 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 is only reasonable 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 right-hand side in a good or optimal solution. If the equal distribution is not present or unknown, using the LP relaxation to determine an initial partition appears beneficial. In contrast, PADM makes use of decomposition where it is expected that especially the linking variables are zero in a good solution. Again, initialization based on the LP solution may be preferable if such information is unknown or if relatively many variables are expected to be nonzero. For DKS, we observed no clear tendency towards a specific decomposition property, but to the underlying problem type. If it shows binary and integer variables, but no continuous ones, the leverage of any available decomposition with DKS appears beneficial.

Finally, two research tasks should be addressed which can be looked at based on the methods presented here. First, the extent to which parallel computing of individual blocks of a decomposition is exploitable 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 already designed in such a way that the subproblems can be solved in parallel. PADM can be parallelized by splitting the inner for-loop in two for-loops. However, the success of KS, and thus DKS, seems to depend on passing information to consecutive iterations. Hence, a parallel solving requires a restructuring of the algorithm which may affect its functionality.

Second, as decompositions can be provided by simple but structured text files, the question arises whether machine learning approaches are able to 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.

Table 10 Results for DPS on MIPLIB, grouped by decomposition type. Including only instances (#46) with all four decomposition types

	deactivate_nonzeros	hrcgpartition	miplib2017	plain_miplib	best
`default`	1.36	1.25	1.32	1.41	1.10
`reopt`	1.38	1.20	1.32	1.42	1.07
`lp`	1.08	1.08	1.04	1.08	1.03
`lp_reopt`	1.08	1.08	1.03	1.08	1.02

Table 11 Results for DPS on MIPLIB. Including all instances (#214) with at least one decomposition type

setting	worst	sgm	best
`default`	1.47	1.27	1.12
`reopt`	1.49	1.28	1.12
`lp`	1.10	1.05	1.02
`lp_reopt`	1.10	1.05	1.02

A Further computational results

In addition to the analysis of the computational results in Section 6, the remaining tables are provided in the following subsections to enable a complete comparison of all settings and subsets of the three test sets.

A.1 Computational results on DPS

Table 10 contains the results for DPS on MIPLIB, when respecting only instances that have all four types of decomposition. Aggregating on instance level first and then computing SGM's gives a summary in Table 11.

On the SCM test set, the analysis grouped by decomposition type when considering all instances is given in Table 12 with a summary in Table 13.

Tables 14 and 15 provide a similar analysis for DPS on CELLPHONE.

In addition to the evaluation in Section 6.1, we would like to take a closer look at the behavior of DPS on test set SCM, as we could make an interesting observation. As its instances originate from supply chain management, they show a special structure based 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 right-hand 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 the root node the initial partition is derived from the LP solution. If there is a

Table 12 Results for DPS on SCM, grouped by decomposition type. Including all instances with at least one decomposition of the given type

setting	worst	sgm	best
(a) decomposition type B, #33 instances			
`default`	1.44	1.29	1.13
`reopt`	1.42	1.28	1.14
`lp`	1.14	1.07	1.01
`lp_reopt`	1.14	1.07	1.01
(b) decomposition type L, #25 instances			
`default`	1.31	1.25	1.18
`reopt`	1.13	1.08	1.02
`lp`	1.26	1.18	1.14
`lp_reopt`	1.23	1.16	1.12
(c) decomposition type P, #28 instances			
`default`	1.20	1.18	1.16
`reopt`	1.22	1.17	1.11
`lp`	1.11	1.01	**0.96**
`lp_reopt`	1.12	1.02	**0.97**
(d) decomposition type S, #28 instances			
`default`	1.40	1.27	1.18
`reopt`	1.35	1.25	1.19
`lp`	1.16	1.10	1.06
`lp_reopt`	1.17	1.10	1.07

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 13 Results for DPS on SCM. Including all instances (#33) with at least one decomposition type

setting	worst	sgm	best
`default`	1.51	1.22	1.04
`reopt`	1.53	1.21	**0.94**
`lp`	1.25	1.08	**0.97**
`lp_reopt`	1.25	1.08	**0.97**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

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.

A.2 Computational results on PADM

In Table 16 the performance of PADM is analyzed across the instances of MIPLIB that have all four decomposition types. A summary which aggregates by instance

Table 14 Results for DPS on CELLPHONE, grouped by decomposition type. Including all instances with at least one decomposition of the given type

setting	worst	sgm	best
(a) decomposition type B, #56 instances			
`default`	2.04	1.94	1.85
`reopt`	2.13	1.59	**0.98**
`lp`	1.06	1.05	1.03
`lp_reopt`	1.07	1.05	1.03
(b) decomposition type L, #56 instances			
`default`	2.07	1.94	1.84
`reopt`	1.64	1.16	**0.80**
`lp`	1.11	1.05	1.00
`lp_reopt`	1.11	1.05	**0.99**
(c) decomposition type P, #56 instances			
`default`	2.34	2.02	1.72
`reopt`	2.38	1.44	**0.75**
`lp`	1.64	1.33	1.01
`lp_reopt`	1.64	1.33	1.01

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 15 Results for DPS on CELLPHONE. Including all instances (#56) with at least one decomposition type

setting	worst	sgm	best
`default`	2.38	1.96	1.68
`reopt`	2.51	1.39	**0.65**
`lp`	1.64	1.14	**0.98**
`lp_reopt`	1.65	1.14	**0.97**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 16 Results for PADM on MIPLIB, grouped by decomposition type. Including only instances (#13) with all four decomposition types

	deactivate_nonzeros	hrcgpartition	miplib2017	plain_miplib	best
`default`	1.43	1.22	1.01	1.29	1.00
`reopt`	1.45	1.23	1.03	1.31	1.01
`lp`	1.00	**0.99**	1.00	1.00	**0.99**
`lp_reopt`	1.00	1.00	1.00	1.00	**0.99**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

first is presented in Table 17. On SCM, distinguishing by decomposition type and aggregating different block counts on instance level gives the figures in Table 18. This is also computed across all decomposition types, giving Table 19.

Omitting the results after root, as PADM has not been called there, the performance split by block count on CELLPHONE is given in Table 20. Here, we cannot observe

Table 17 Results for PADM on MIPLIB. Including all instances (#214) with at least one decomposition type

setting	worst	sgm	best
`default`	1.11	1.04	**0.99**
`reopt`	1.11	1.04	**0.98**
`lp`	1.02	1.01	1.01
`lp_reopt`	1.02	1.01	1.00

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 18 Results for PADM on SCM, grouped by decomposition type. Including all instances with at least one decomposition of the given type

setting	worst	sgm	best
(a) decomposition type B, #33 instances			
`default`	1.95	1.33	**0.97**
`reopt`	1.96	1.32	**0.97**
`lp`	1.23	1.08	1.00
`lp_reopt`	1.24	1.08	1.00
(b) decomposition type L, #25 instances			
`default`	1.24	1.23	1.21
`reopt`	1.06	1.04	1.02
`lp`	1.20	1.16	1.11
`lp_reopt`	1.19	1.15	1.10
(c) decomposition type P, #28 instances			
`default`	1.23	1.11	**0.93**
`reopt`	1.25	1.13	**0.94**
`lp`	1.16	1.07	1.00
`lp_reopt`	1.17	1.08	1.00
(d) decomposition type S, #28 instances			
`default`	1.00	1.00	**0.99**
`reopt`	1.00	1.00	1.00
`lp`	1.00	1.00	**0.99**
`lp_reopt`	1.01	1.00	**0.99**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 19 Results for PADM on SCM. Including all instances (#33) with at least one decomposition type

setting	worst	sgm	best
`default`	1.96	1.17	**0.93**
`reopt`	1.96	1.16	**0.92**
`lp`	1.39	1.06	**0.97**
`lp_reopt`	1.39	1.07	**0.96**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 20 Results for PADM on CELLPHONE, grouped by decomposition type. Including only instances with decompositions yielding the two block counts: 2 and 4

	2	4	best
(a) decomposition type B, #40 instances			
`default`	2.34	2.40	2.31
`reopt`	2.34	2.36	2.28
(b) decomposition type L, #49 instances			
`default`	2.25	2.27	2.23
`reopt`	2.26	2.28	2.24
(c) decomposition type P, #18 instances			
`default`	2.20	2.21	2.19
`reopt`	2.21	2.21	2.20

Table 21 Results for PADM on CELLPHONE, grouped by decomposition type. Including all instances with at least one decomposition of the given type

setting	worst	sgm	best
decomposition type B, #56 instances			
`default`	2.03	1.53	**0.97**
`reopt`	2.03	1.54	**0.99**
decomposition type L, #56 instances			
`default`	2.13	1.61	**0.97**
`reopt`	2.14	1.62	**0.97**
decomposition type P, #56 instances			
`default`	1.83	1.33	**0.99**
`reopt`	1.83	1.33	**0.99**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

Table 22 Results for PADM on CELLPHONE. Including all instances (#56) with at least one decomposition type.

setting	worst	sgm	best
`default`	2.18	1.48	**0.97**
`reopt`	2.19	1.49	**0.97**

Bold figures indicate a perfromance increase, i.e., a value less than 1, by the investigated method.

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 (Berthold and Hendel 2015) 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 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.

Again, Table 21 and Table 22 provide a concise version of the results across decomposition type and overall.

A.3 Computational results on DKS split by decomposition

Following the default order of MIPLIB, SCM, then CELLPHONE, the results for all three test sets grouped by decomposition type and analyzed for instances showing all such are given in Tables 23, 25 and 28. When considering all instances on both supply chain test sets, the figures are given in Tables 26 and 29. Summaries are provided in Tables 24, 27 and 30, respectively.

Table 23 Results for DKS on MIPLIB, grouped by decomposition type. Including only instances (#60) with all four decomposition types

	deactivate_nonzeros	hrcgpartition	miplib2017	plain_miplib	best
dks	1.17	1.17	1.14	1.17	1.12
ks	1.04	1.04	1.04	1.04	1.03

Table 24 Results for DKS on MIPLIB. Including all instances (#215) with at least one decomposition type.

setting	worst	sgm	best
dks	1.10	1.07	1.04
ks	1.03	1.02	1.02

Table 25 Results for DKS on SCM, grouped by decomposition type. Including only instances with decompositions yielding all three block counts: 2, 4, and max

	2	4	max	best
(a) decomposition type B, #14 instances				
dks	1.01	1.00	1.01	1.00
ks	1.54	1.54	1.54	1.54
(b) decomposition type L, #5 instances				
dks	1.02	1.03	1.02	1.02
ks	1.06	1.07	1.07	1.06
(c) decomposition type P, #9 instances				
dks	1.02	1.01	1.02	1.00
ks	1.02	1.02	1.02	1.02
(d) decomposition type S, #9 instances				
dks	1.01	1.01	1.02	1.00
ks	1.03	1.02	1.02	1.02

Table 26 Results for DKS on SCM grouped by decomposition type. Respecting all instances having at least 2, 4, or max blocks

setting	worst	sgm	best
(a) decomposition type B, #33 instances			
dks	1.01	1.00	1.00
ks	1.20	1.20	1.20
(b) decomposition type L, #25 instances			
dks	1.01	1.00	1.00
ks	1.27	1.27	1.26
(c) decomposition type P, #28 instances			
dks	1.02	1.01	1.00
ks	1.01	1.01	1.01
(d) decomposition type S, #28 instances			
dks	1.01	1.00	1.00
ks	1.01	1.01	1.01

Table 27 Results for DKS on SCM. Including all instances (#33) with at least one decomposition type

setting	worst	sgm	best
dks	1.02	1.00	1.00
ks	1.20	1.20	1.19

Table 28 Results for DKS on CELLPHONE, grouped by decomposition type. Including only instances with decompositions yielding all three block counts: 2, 4, and max

	2	4	max	best
(a) decomposition type B, #40 instances				
dks	1.01	1.01	1.01	1.01
ks	1.20	1.20	1.20	1.20
(b) decomposition type L, #40 instances				
dks	1.01	1.01	1.01	1.01
ks	1.20	1.20	1.20	1.20
(c) decomposition type P, #39 instances				
dks	1.01	1.01	1.01	1.01
ks	1.21	1.21	1.21	1.21

Table 29 Results for DKS on CELLPHONE grouped by decomposition type. Respecting all instances having at least 2, 4, or max blocks

setting	worst	sgm	best
(a) decomposition type B, #56 instances			
dks	1.01	1.01	1.00
ks	1.14	1.14	1.14
(b) decomposition type L, #56 instances			
dks	1.01	1.01	1.01
ks	1.14	1.14	1.14
(c) decomposition type P, #56 instances			
dks	1.01	1.01	1.00
ks	1.14	1.14	1.14

Table 30 Results for DKS on CELLPHONE. Including all instances (#56) with at least one decomposition type

setting	worst	sgm	best
dks	1.01	1.01	1.00
ks	1.14	1.14	1.14

Acknowledgements The authors would like to thank SAP for its long-term 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.

Funding Open Access funding enabled and organized by Projekt DEAL. No funding was received to assist with the preparation of this article.

Declarations

Competing interests The authors have no competing interests to declare that are relevant to the content of this article.

References

Achterberg, T.: Constraint Integer Programming. PhD thesis. Technische Universität, Berlin (2007)

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

Aliev, I., Loera, J.A.D., Eisenbrand, F., Oertel, T., Weismantel, R.: The support of integer optimal solutions. SIAM J. Optim. **28**(3), 2152–2157 (2018). https://doi.org/10.1137/17M1162792

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

Angelelli, E., Mansini, R., Speranza, M.G.: Kernel search: A new heuristic framework for portfolio selection. Comput. Optim. Appl. **51**(1), 345–361 (2012). https://doi.org/10.1007/s10589-010-9326-6

Battista, G.D., Tamassia, R.: 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). https://doi.org/10.1109/SFCS.1989.63515

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

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

Berthold, T.: Heuristic algorithms in global MINLP solvers. PhD thesis, (2014). http://www.zib.de/berthold/Berthold2014.pdf

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

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., van der Hulst, R., Koch, T., Lübbecke, M., Maher, S.J., Matter, F., Mühmer, E., Müller, B., Pfetsch, M.E., Rehfeldt, D., Schlein, S., Schlösser, F., Serrano, F., Shinano, Y., Sofranac, B., Turner, M., Vigerske, S., Wegscheider, F., Wellner, P., Weninger, D., Witzig. J.: The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. http://www.optimization-online.org/DB_HTML/2021/12/8728.html

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

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

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

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

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

Dantzig, G.B., Wolfe, P.: Decomposition Principle for Linear Programs. Oper. Res. **8**(1), 101–111 (1960). https://doi.org/10.1287/opre.8.1.101. (ISSN 0030-364X)

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

Fischetti, M., Ljubić, I., Sinnl, M.: 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. https://doi.org/10.1016/j.ejor.2016.03.002. http://www.sciencedirect.com/science/article/pii/S0377221716301126

Gamrath, G., Lübbecke, M.E.: 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

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

Gamrath, G., Gleixner, A., Koch, T., Miltenberger, M., Kniasew, D., Schlögel, D., Martin, A., Weninger, D.: Tackling industrial-scale supply chain problems by mixed-integer programming. J. Comput. Math. **37**, 866–888 (2019). https://doi.org/10.4208/jcm.1905-m2019-0055

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

Geißler, B., Morsi, A., Schewe, L., Schmidt, M.: Penalty alternating direction methods for mixed-integer optimization: A new view on feasibility pumps. SIAM J. Optim. **27**(3), 1611–1636 (2017). https://doi.org/10.1137/16M1069687. (ISSN 2192-4414)

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

Geoffrion, A.M.: 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. https://doi.org/10.1007/BFb0120690

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

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

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

Gutwenger, C., Mutzel, P.: 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. https://doi.org/10.1007/3-540-44541-2_8

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

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

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

Karypis, G., Kumar, V.: Multilevel k-way Hypergraph Partitioning. In In Proceedings of the Design and Automation Conference, pages 343–348, (1998a)

Karypis, G., Kumar, V.: hMetis: A Hypergraph Partitioning Package, Version 1.5.3, (1998b)

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

SAP SE. SAP Software Solutions | Business Applications and Technology, last accessed on 2023-06-20. https://www.sap.com

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

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

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

Tutte, W.T.: Connectivity in graphs. Mathematical Expositions, 15, (1966)

Vanderbeck, F., Wolsey, L.A.: Reformulation and Decomposition of Integer Programs. Springer, Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-540-68279-0_13 . (ISBN 978-3-540-68279-0)

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

Wolsey, L.A.: Integer Programming. John Wiley and Sons, Ltd, (2020). ISBN 9781119606475. https://doi.org/10.1002/9781119606475. https://onlinelibrary.wiley.com/doi/book/10.1002/9781119606475

Yıldız, B., Boland, N., Savelsbergh, M.: Decomposition branching for mixed integer programming. Oper. Res. **70**(3), 1854–1872 (2022). https://doi.org/10.1287/opre.2021.2210

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