This is an Accepted Manuscript of an article published by Taylor & Francis in IISE Transactions on 02 Aug 2016, available online: http://dx.doi.org/10.1080/0740817X.2016.1213469

Which items should be stored together? A basic partition problem to assign storage space in group-based storage systems

Dominik Kress[a,*], Nils Boysen[b], Erwin Pesch[a,c]

[a] *University of Siegen, Management Information Science, Kohlbettstraße 15, 57068 Siegen, Germany*
[b] *Friedrich-Schiller-University Jena, Operations Management, Carl-Zeiß-Straße 3, 07743 Jena, Germany*
[c] *Center of Advanced Studies in Management, HHL Leipzig, Jahnallee 59, 04109 Leipzig, Germany*

Abstract

We consider a basic partition problem that subdivides stock keeping units (SKUs) into disjoint subsets, such that the minimum number of groups has to be accessed when retrieving a given order set under a pick-by-order policy. We formalize this SKU partition problem and show its applicability in a wide range of storage systems that are based on separating their storage space into groups of SKUs stored in separate areas; examples are carousel racks and mobile shelves. We analyze the computational complexity and propose two mathematical models for the problem under consideration. Furthermore, we present an ejection chain heuristic and a branch and bound procedure. We analyze these algorithms and the mathematical models in computational tests.

Keywords: Warehousing, Storage assignment, Partitioning, Ejection chain

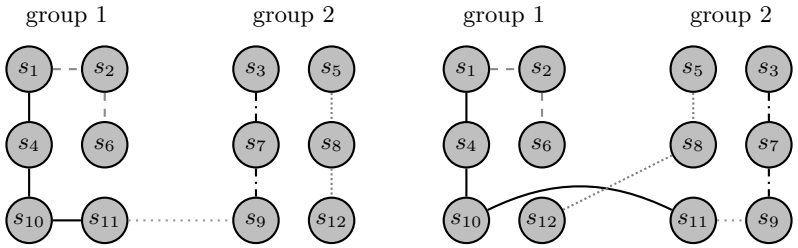
1. Introduction

The storage assignment problem is among the most essential decision problems to be solved in any warehouse. Each stock keeping unit (SKU) is to be assigned a storage position from where it is to be retrieved during order picking. In many warehouses, the detailed slotting, i.e., the decision on the specific shelf each SKU is stored in, is less of an issue, but the problem rather reduces to a partition problem; SKUs are to be jointly stored in one area or group, so that picking-orders can efficiently and conveniently be retrieved without having to access too many groups. In this context, we treat the following basic partition problem, which we denote as the *SKU partition problem*: Consider a given set of SKUs, which are to be partitioned into groups of equal size, and a deterministic set of picking-orders each defining a subset of SKUs demanded by an order's customer. Depending on the partitioning of items, orders require different numbers of groups to be accessed during order picking. We refer to these numbers as

*Corresponding author
Email addresses: dominik.kress@uni-siegen.de (Dominik Kress), nils.boysen@uni-jena.de (Nils Boysen), erwin.pesch@uni-siegen.de (Erwin Pesch)

the orders' *group numbers*. Our objective is to find a partitioning of SKUs that minimizes the weighted sum of group numbers over all picking-orders.

Let us illustrate the SKU partition problem based on an example: Consider $n = 12$ SKUs, $s_1, s_2, \ldots, s_{12}$, to be partitioned into $k = 2$ groups, each of size $C = \frac{n}{k} = 6$, and $m = 5$ picking-orders, $o_1 = \{s_1, s_2, s_6\}$, $o_2 = \{s_1, s_4, s_{10}, s_{11}\}$, $o_3 = \{s_3, s_7, s_9\}$, $o_4 = \{s_5, s_8, s_{12}\}$, and $o_5 = \{s_9, s_{11}\}$, which all have weight $w_\sigma = 1$, $\sigma \in \{1, \ldots, m\}$. Two different solutions for this problem instance are depicted in Figure 1, where orders are represented by edges connecting the relevant SKUs. While the first solution (Figure 1a) contains only a single order, o_5, that requires an access of both groups during retrieval, the second solution (Figure 1b) has two of these orders, i.e., o_2 and o_4. Thus, the objective values of these solutions, i.e., the total number of groups accessed when retrieving all orders, amount to $m+1 = 6$ and $m+2 = 7$, respectively.



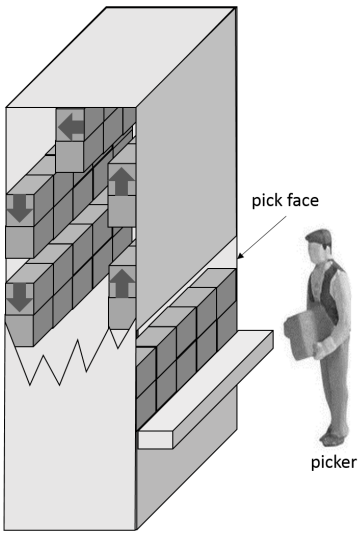
(a) First solution (b) Second solution

Figure 1: Two solutions to an example instance of the SKU partition problem

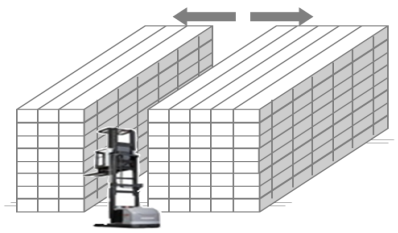
1.1. Applications and literature review

In general, partition problems have plenty potential applications in a wide range of areas. We, however, focus on warehousing and introduce three applications (see Figure 2), where the assignment of storage space to SKUs corresponds to the SKU partition problem.

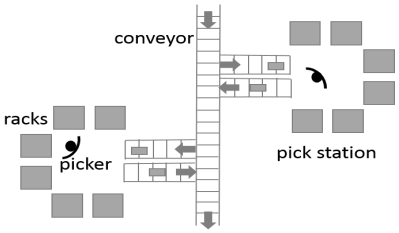
A *carousel* (see Figure 2a) is a special kind of automated storage and retrieval system, in which linked shelves or drawers are turned in an oval closed loop. Required SKUs are turned towards the front and are accessed (typically by a human order picker) via a window-like pick face. These parts-to-picker systems either rotate horizontally or vertically and they are typically applied to store small or medium sized items in a space-efficient, yet easy to access manner. A recent literature survey on carousels is provided by Litvak & Vlasiou (2010). A similar system, to which our SKU partition problem is also applicable, is based on vertical lift modules, where SKUs are stored on trays brought to the pick face by some lift (see, for example, Meller & Klote, 2004). Depending on the current position of the loop, the picker can conveniently access all active shelves that are currently displayed in the pick face. A major source for picker idle time is the time it takes to rotate the carousel in order to change the set of shelves presented in



(a) Vertical carousel



(b) Mobile shelves



(c) Pick-to-belt with stat. pickers

Figure 2: Applications of the SKU partition problem

the pick face. If we think of a subset of SKUs that can simultaneously be accessed as a group, then our basic partition problem can directly be applied to determine groups that minimize the number of carousel moves when having to retrieve a given order set. The order weights can be applied to indicate the frequency with which each order (presumably) occurs. Once a grouping of SKUs is obtained, groups are to be assigned to specific shelf segments. However, this aspect is beyond the scope of this paper.

A vast body of literature on carousels has accumulated over the recent decades. Instead of trying to summarize all these papers, we refer to the excellent survey paper by Litvak & Vlasiou (2010). Most of these papers treat decision problems differing from the one considered in this paper, e.g., order sequencing or throughput estimation. The existing papers on the storage assignment problem in carousels either treat random storage (Hwang & Ha, 1991), a two-class based storage (Ha & Hwang, 1994; Hwang & Ha, 1994), presuppose the so-called organ pipe arrangement (Abdel-Malek & Tang, 1994; Bengü, 1995; Litvak, 2006; Vickson & Fujimoto, 1996), or treat shared storage, where SKUs of the same type are stored in multiple positions (Hassini, 2008; Jacobs et al., 2000; Kim, 2005). All these approaches considerably deviate from our group-based view on the storage location problem in carousels. The main distinction of our view on the problem is a given order set, which is often not yet available when deciding on the storage locations, e.g., when facing highly volatile demands of final customers. However, given order sets are realistic in intermediate warehouses, which have to repeatedly assemble picking orders for parts demanded by predefined, cyclically produced production lots. A more detailed discussion of our assumptions is provided in Section 2.3.

Another form of compact storage relies on *mobile shelves* (see Figure 2b). Here, the parallel racks of a warehouse are close-packed and mounted on rails, so that the neighboring racks need

to be moved aside to open a specific aisle that allows access to a rack. Traditionally, mobile shelves (also denoted as roller racks) are moved manually, e.g., by turning a star handle, and applied for storing rarely accessed books in libraries. However, mobile rack systems can also be automated, which means that some strong engine is applied to move the racks once a button or ripcord is activated. Automated mobile racks are mainly applied in refrigerated warehouses, e.g., for frozen food or pharmaceuticals, where saving energy by compact storage is an important issue (Boysen et al., 2014). Fully loaded racks are very heavy, so that moving the racks and opening an aisle causes considerable waiting time for a picker (with or without a picking vehicle). Warehouse equipment manufacturer SSI Schäfer (2014), for instance, reports on a real-world system where the racks only move with 4 m/min compared to 115 m/min accomplished by the picking vehicle. In such a setting, waiting due to aisle movements becomes the main driver of picking performance, which in turn gives rise to our basic partition problem. Again, a suited partition of SKUs to be accessed via the same aisle reduces the aisle movement when retrieving a given order set.

In spite of their long tradition, just a few papers on mobile shelves have been published. The few papers existing rather treat the order sequence problem (Boysen et al., 2014; Chang et al., 2007; Hu et al., 2009) and assume given storage locations.

Especially in the automotive industry, the sequencing of parts required by the final assembly lines is often carried out in multiple, u-shaped *pick stations* (see Figure 2c), which are interconnected by a conveyor belt system distributing totes, bins or containers associated with the picking-orders (Boysen et al., 2015). Typical picking stations contain man-high racks closely packed in a u-shaped area, so that a picker can retrieve the items located in his/her area without excessive walking. As processing an order in a station requires lead time, e.g., for receiving a container from the conveyor, scanning it, and identifying the demanded parts on the pick list, it seems, thus, advantageous to partition SKUs among stations, such that the number of station visits over all picking-orders is minimized.

Literature on stationary pickers in u-shaped stations is rare. Only Glock & Grosse (2012) treat the storage location and order processing in these stations. They, however, treat a far more complex industry case, which bears no resemblance to our problem.

With regard to its mathematical structure, our SKU partition problem has some similarities to graph partitioning and vertex coloring. However, to the best of the authors' knowledge such a problem has yet not been treated in these areas. Thus, it can be concluded that the SKU partition problem has not been covered in the literature and has a wide range of potential applications.

1.2. Contribution and paper structure

In this paper we will analyze the SKU partition problem, which - as described above - is a basic decision problem reducing the group-based storage assignment it to its very core. We aim to compare two mathematical models and develop efficient solution algorithms. The problem under consideration can then serve as a building block or elementary subproblem in all of the applications described in Section 1.1. In the following, however, we focus on one of the applications, i.e., the storage assignment in carousel racks.

The paper is structured as follows. In Section 2, we provide a formal definition of the SKU partition problem, accompanied by an analysis of its computational complexity (Section 2.1), mathematical models (Section 2.2), and a discussion of simplifying assumptions with respect to the application of storage assignment in carousel racks (Section 2.3). Section 3 introduces solution algorithms. This includes a heuristic "ejection chain" procedure in Section 3.1 as well as a branch and bound approach in Section 3.2. A comprehensive computational study is subject of Section 4. The mathematical models and the algorithms are analyzed in Section 4.1. The effectiveness of the group-based view with respect to picker idle time is investigated in Section 4.2. Finally, Section 5 concludes the paper.

2. Detailed problem definition and analysis

Let $S = \{s_1, \ldots, s_n\}$, $|S| = n$, be the set of SKUs. Furthermore, define the order set O, $|O| = m$, to be the set of picking-orders o_σ, $\sigma = 1, \ldots, m$, i.e., $O = \{o_1, \ldots, o_m\}$. Each picking-order $o_\sigma \in O$ is represented by the set of SKUs demanded within the order, i.e., $o_\sigma \subseteq S$. Furthermore, each picking-order $o_\sigma \in O$ is assigned a nonnegative weight $w_\sigma \in \mathbb{N}$, which corresponds to the frequency with which o_σ is to be picked. We refer to the number of SKUs within order $o_\sigma \in O$ by $|o_\sigma|$. Our task is to partition S into a given number of k groups, i.e., disjoint subsets $S_1, \ldots, S_k$ with $\bigcup_{j\in\{1,\ldots,k\}} S_j = S$. For a given partitioning, we define

$$\chi(o_\sigma) := \sum_{j=1}^{k} H\left(|o_\sigma \cap S_j|\right)$$

as the *group number* of order $o_\sigma \in O$, i.e., the number of groups that have to be accessed when picking it. Here,

$$H(\delta) = \begin{cases} 0 & \text{if } \delta \leq 0 \\ 1 & \text{if } \delta > 0 \end{cases}$$

is the Heaviside step function. Our SKU partition problem aims to determine a partitioning of S, such that all groups have equal size C, i.e., $|S_j| = \frac{n}{k} = C \;\forall\, j = 1, \ldots, k$, and the total

(weighted) number of groups that have to be accessed when retrieving all orders is minimized, i.e., min $\rightarrow F(S_1, \ldots, S_k) = \sum_{o_\sigma \in O} \chi(o_\sigma) \cdot w_\sigma$.

In what follows, we will assume that $C > 1$ because the case $C = 1$ is trivial.

2.1. Computational complexity

Consider an edge-weighted graph $G = (V, E)$ with vertex set V and edge set E. If V is to be partitioned into two non-empty components without any restriction on the size of these subsets and such that the sum of the edge weights of the edges incident to vertices of different components is minimum, then this problem (the 2-Cut problem) can be solved in polynomial time by a repeated application of a Max-Flow algorithm (Karp, 1972). However, when the size of both subsets is to be the same, the 2-Cut problem is strongly NP-hard (Garey et al., 1976). This problem is often called the (weighted) bisection problem; see Karpinski (2002) for approximability results.

We will now show that SKU partition is NP-hard in the strong sense by reduction from the bisection problem.

Consider an instance $G(V, E)$ of the bisection problem with an even number n of vertices in set V and m edges in set E. For the sake of simplicity, consider unit edge weights. Note, however, that the same reasoning applies for positive and integer edge weights that are equal for all edges in E.

Construct an instance of SKU partition with n SKUs and $k = 2$. Assume that there are m orders with two different SKUs each. Each order can be identified with an edge in E and there is a one-to-one correspondence between SKUs and vertices of G. The SKUs of an order are assumed to be identical to the SKUs assigned to both adjacent vertices that correspond to the order-defining edge in G. Let B be a given integer.

Assume there is a bisection of V into two subsets V_1 and V_2 of the same size, where the number of edges that connect vertices from both subsets does not exceed B. Assign all SKUs that correspond to vertices of V_1 to the first group and those SKUs identified with vertices of V_2 to the second group of the SKU partition instance. It is immediately obvious that the number of orders that pick SKUs from both groups does not exceed the value of B if there is a bisection with at most B edges in the cut. Thus, we have:

Theorem 1. *The SKU partition problem is strongly NP-hard, even if the order weights w_σ are equal (and non-zero) for all $\sigma = 1, \ldots, m$, $k = 2$, and each order consists of only two SKUs.*

2.2. Mathematical models for the SKU partition problem

The SKU partition problem can be expressed as a mixed integer programming (MIP) model in multiple ways. A straightforward model uses variables

$$x_{ij} := \begin{cases} 1 & \text{if } s_i \text{ is assigned to } S_j, \\ 0 & \text{else,} \end{cases} \qquad \forall\, i \in \{1,\dots,n\}, j \in \{1,\dots,k\},$$

and

$$y_{\sigma j} := \begin{cases} 1 & \text{if } S_j \text{ is accessed when picking } o_\sigma, \\ 0 & \text{else,} \end{cases} \qquad \forall\, j \in \{1,\dots,k\}, \sigma \in \{1,\dots,m\}.$$

Additionally, we define the following binary parameters:

$$a_{\sigma i} := \begin{cases} 1 & \text{if } s_i \text{ is demanded within } o_\sigma, \\ 0 & \text{else,} \end{cases} \qquad \forall\, i \in \{1,\dots,n\}, \sigma \in \{1,\dots,m\}.$$

Then, a mathematical model is as follows.

$$\min_{\mathbf{x},\mathbf{y}} \quad \sum_{\sigma=1}^{m}\sum_{j=1}^{k} w_\sigma y_{\sigma j} \tag{1}$$

$$\text{s.t.} \quad \sum_{j=1}^{k} x_{ij} = 1 \qquad \forall\, i \in \{1,\dots,n\}, \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} = C \qquad \forall\, j \in \{1,\dots,k\}, \tag{3}$$

$$a_{\sigma i} x_{ij} \le y_{\sigma j} \qquad \forall\, i \in \{1,\dots,n\},\ \sigma \in \{1,\dots,m\}, j \in \{1,\dots,k\}, \tag{4}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i \in \{1,\dots,n\}, j \in \{1,\dots,k\}, \tag{5}$$

$$y_{\sigma j} \ge 0 \qquad \forall\, \sigma \in \{1,\dots,m\}, j \in \{1,\dots,k\}. \tag{6}$$

Objective function (1) minimizes the total weighted number of group access operations over all picking-orders. Constraints (2) and (3) ensure that each SKU is assigned to a group and that the maximum group size is considered, respectively. In (4), care is taken that once an SKU is contained in an order and assigned to a group, then this group is to be accessed when retrieving the order. Finally, the domains of variables are defined in (5) and (6). Note that $y_{\sigma j}$, $\sigma \in \{1,\dots,m\}$, $j \in \{1,...,k\}$, need not be defined as a binary variable, since these values are guaranteed automatically.

Another model is based on mathematical formulations of the well-known clique partitioning

problem (see, for example, Dorndorf & Pesch, 1994; Jaehn & Pesch, 2013, and the references therein). Let

$$z_{ij} := \begin{cases} 1 & \text{if } s_i \text{ and } s_j \text{ are stored in the same group,} \\ 0 & \text{else,} \end{cases} \qquad \forall\, i,j \in \{1,\dots,n\}, j > i, \tag{7}$$

and define continuous nonnegative variables q_{ij}^{σ} for all $o_\sigma \in O$ and $s_i, s_j \in o_\sigma$, $i < j$. q_{ij}^{σ} takes the value 1, if SKUs s_i and s_j, $i < j$, are in the same group and in the same order o_σ and if there exists no SKU s_l, $l < i$, within the same group that is included in o_σ as well. If s_i and s_j are not in the same group, then q_{ij}^{σ} is zero. An example is presented below. Then a mathematical formulation of SKU Partition is as follows.

$$\min_{\mathbf{z},\mathbf{q}} \quad \sum_{\sigma=1}^{m} w_\sigma \Big(|o_\sigma| - \sum_{s_i \in o_\sigma} \sum_{s_j \in o_\sigma, j>i} q_{ij}^{\sigma}\Big) \tag{8}$$

$$\text{s.t.} \quad \sum_{j \in \{1,\dots,i-1\}} z_{ji} + \sum_{j \in \{i+1,\dots,n\}} z_{ij} = C - 1 \qquad \forall\, i \in \{1,\dots,n\}, \tag{9}$$

$$z_{ij} + z_{jl} - z_{il} \le 1 \qquad \forall\, 1 \le i < j < l \le n, \tag{10}$$

$$z_{ij} - z_{jl} + z_{il} \le 1 \qquad \forall\, 1 \le i < j < l \le n, \tag{11}$$

$$-z_{ij} + z_{jl} + z_{il} \le 1 \qquad \forall\, 1 \le i < j < l \le n, \tag{12}$$

$$q_{ij}^{\sigma} \le z_{ij} \qquad \forall\, o_\sigma \in O, s_i, s_j \in o_\sigma, i < j \tag{13}$$

$$q_{jl}^{\sigma} \le 2 - (z_{ij} + z_{il}) \qquad \forall\, o_\sigma \in O, s_i, s_j, s_l \in o_\sigma, i < j < l \tag{14}$$

$$z_{ij} \in \{0,1\} \qquad \forall\, i,j \in \{1,\dots,n\}, i < j, \tag{15}$$

$$q_{ij}^{\sigma} \ge 0 \qquad \forall\, o_\sigma \in O, s_i, s_j \in o_\sigma, i < j. \tag{16}$$

Constraints (9) guarantee that each group is composed of exactly C SKUs. Constraints (10)–(12) are well known from mathematical formulations of the clique partitioning problem (Dorndorf & Pesch, 1994; Jaehn & Pesch, 2013). They guarantee transitivity, i.e., if SKUs s_i and s_j belong to the same group and SKUs s_j and s_l belong to the same group, then s_i and s_l must also belong to the same group. Constraints (13) set the continuous variables q_{ij}^{σ}, $o_\sigma \in O$ and $s_i, s_j \in o_\sigma$, $i < j$, to zero if SKUs s_i and s_j are not in the same group. Restrictions (14) set q_{jl}^{σ}, $o_\sigma \in O$ and $s_i, s_j, s_l \in o_\sigma$, $i < j < l$, to zero, if SKUs s_i, s_j and s_l are in the same group. Observe that the sum over all continuous variables that are related to a specific picking-order $o_\sigma \in O$ is directly related to the number of group access operations needed to pick o_σ. An example is presented in Figure 3, where the SKUs have been partitioned into two groups, $S_1 = \{s_1, s_2, s_3\}$ and $S_2 = \{s_4, s_5, s_6\}$. All edges depicted in the figure (solid and

dashed) represent positive variables z. Let us consider a picking-order $o_1 = \{s_1, s_2, s_3, s_4\}$ (grey vertices). Then the solid edges represent corresponding positive variables q^1_{ij}, $i < j \in \{1, \ldots, 4\}$, in an optimal solution with the groups being fixed. It is easy to see that the number of groups that have to be accessed to pick the order corresponds to the number of elements of the order minus the number of positive variables q^1_{ij}, i.e., $4 - 2 = 2$. Hence, the objective function (8)

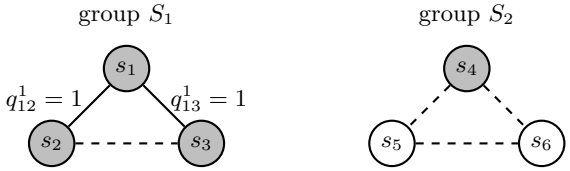


Figure 3: Illustration of the variables of model (8)–(16)

minimizes the (weighted) number of group access operations over all picking-orders. Finally, constraints (15) and (16) define the domains of the variables.

2.3. Simplifying assumptions

In order to derive the rather basic problem described above from the applications described in Section 1.1, some simplifying assumptions are inevitable. The most elementary ones are listed and discussed in the following.

First, we presuppose exactly the same number of SKUs as there are storage positions. If this assumption does not hold in the real world, we can simply introduce "dummy" SKUs that are not contained in any order. Furthermore, we assume that no assignment restrictions exist and any SKU can be assigned to any storage position. This allows for reducing the storage assignment to a partition problem, in which one can abstract from an explicit assignment of SKUs to storage positions. The exact slotting is postponed to a successive planning step, which is a potential direction for future research.

In line with the prior assumption, the sequence of the picking-orders is neglected and we assume that orders are picked according to the pick-by-order policy without batching. Orders are thus retrieved one at a time, so that all relevant groups are accessed until all SKUs of the current order are collected. Hereafter, the next order is started to be picked. If orders are to be retrieved more than once during the planning horizon, this is represented by the order weights. Hence, we presuppose that the picking effort is mainly affected by the total number of group access operations, i.e., when the shelves of a carousel have to be rotated while an order is picked or when the retrieval of a new order is started.

The influence of the specific storage locations within a group is not considered. On the one hand, specific positions may indeed be of minor relevance in the real world. In a carousel, for instance, the pick face is comparatively small, so that all active shelf positions can conveniently

be reached by the picker. On the other hand, specific positions within a group can still be considered in a successive planning step.

Moreover, we presuppose a dedicated storage policy, with all SKUs of the same type being stored in a unique storage position. Because the locations of SKUs do not change, popular items can be stored in especially convenient locations and workers can learn the topology of the warehouse. On the negative side, dedicated storage does not use space efficiently, since storage space has to be dimensioned according to the maximum amount of stored items per SKU (see Bartholdi III & Hackman, 2014). One of our further assumptions is a predefined order set, which is discussed below. Given this assumption, identifying popular items is fairly easy and, therefore, properly dimensioning the storage space is not complicated by uncertain demands, so that applying dedicated storage seems a natural choice in our setting. Shared storage, where more than one location is assigned to each SKU, seems better suited in uncertain environments, so that we leave the integration of shared storage into our problem setting up to future research.

To reduce the SKU partition problem to its very core, we, furthermore, assume that all SKUs require identical storage space. In the real world, this assumption holds, whenever either only standardized bins or containers are applied (e.g. in carousel racks) or unit-loads like ISO-pallets are utilized (e.g. in mobile shelves or pick-to-belt applications). Furthermore, all groups are assumed to have equal capacity, which is typically fulfilled in the applications described in Section 1.1.

Finally, from a practitioner's point of view, the most debatable assumption is certainly the deterministic order set. Typically, storage assignment is a long- to mid-term decision, so that picking-orders are not available and hardly predictable. This is especially true for warehouses that have to serve volatile demands of final customers. Therefore, existing research either only considers how often each SKU is demanded or integrates correlations among product pairs into the storage assignment problem. However, a given order set may be a valid approximation of reality in intermediate warehouses, where, for instance, recurrent orders for parts consumed by cyclically produced production lots in a low-variety make-to-stock supply chain are picked (Boysen & Stephan, 2013). Moreover, even in a volatile environment where anticipating an order set is bound to forecast errors, assuming a deterministic order set may still be better than completely neglecting that some products are frequently ordered together.

3. Algorithms

As we have shown in Section 2.2, the SKU partition problem can be modelled based on model formulations for the clique partitioning problem. Concerning this latter problem, it is well known that ejection chain heuristics, which are based on an idea of Glover (1996) and

Pesch & Glover (1997), perform well (Dorndorf et al., 2008; Dorndorf & Pesch, 1994). This motivated us to implement an ejection chain based heuristic for the SKU partition problem, which we will describe in Section 3.1.

Dorndorf & Pesch (1994) present a branch and bound method for the clique partitioning problem that was later improved by Jaehn & Pesch (2013). We will make use of some of their results to construct a branch and bound method based on model (8)–(16) for the SKU partition problem. This method will be subject of Section 3.2.

3.1. Ejection chain heuristic

Glover (1996) describes the idea of an ejection chain heuristic as follows: "Ejection chain procedures are based on the notion of generating compound sequences of moves, leading from one solution to another, by linked steps in which changes in selected elements cause other elements to be 'ejected from' their current state, position or value assignment." In our case, a *move* alters a given (not necessarily feasible) solution by selecting an SKU $s_l \in S_s$ from a group S_s (*source group*) and inserting it into a group S_t (*target group*), $t \neq s$. Once a move has been performed, the next move is initiated by "ejecting" an SKU $s_{\hat{l}} \in S_t$, $\hat{l} \neq l$, from S_t, i.e., the target group of the preceding move becomes the source group of the next move. This results in a series of successional moves that result in a number of solutions that we refer to as *reference solutions*. Obviously, if a resulting reference solution is infeasible, we can "eject the infeasibility" by moving an SKU from the (unique) group with $C+1$ elements to the (unique) group with $C-1$ elements (*repair move*) to construct a feasible solution, referred to as a *trial solution*. Figure 4 illustrates the basic idea of an ejection chain heuristic for the SKU partition problem.

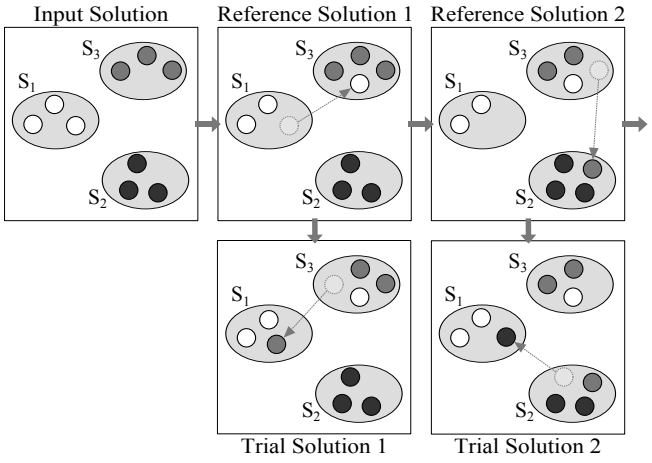


Figure 4: Illustration of ejection chain heuristic

Algorithm 1 describes the framework of the ejection chain heuristic for SKU partition. For a detailed overview, we refer to Appendix A. Given a feasible *input solution* (*inSol*) to an

```
   Data: feasible solution inSol; parameters maxChainLength, tabuLength
   Result: feasible solution bestSol
   // Step 0: initialize
 1 initialize tabuList;
 2 do
       // Step 1: initiate ejection chain from every SKU s_i ∈ S
 3     for all groups S_j, j ∈ {1,...,k}, and all SKUs s_i ∈ S_j of inSol do
 4         clear tabuList;
           // Step 1a: first move of the chain
 5         generate refSol by determining the best move of s_i from S_j to any targetGroup;
 6         update tabuList and set sourceGroup = targetGroup;
           // Step 1b: first repair move of the chain
 7         generate trialSol by determining best repair move; potentially update bestSol;
 8         for chainLength = 2 to maxChainLength do
               // Step 1c: terminate
 9             if all SKUs of sourceGroup of refSol are tabu then break;
10             else
                   // Step 1d: other moves of the chain
11                 alter refSol by determining the best move of any non-tabu SKU s_l of sourceGroup to
                   any targetGroup;
12                 update tabuList and set sourceGroup = targetGroup;
                   // Step 1e: other repair moves of the chain
13                 if refSol is feasible then potentially update bestSol;
14                 else generate trialSol by determining best repair move; potentially update bestSol;
15             end
16         end
17     end
18     if new best solution has been found then inSol = bestSol;
19 while a new best solution has been found;
```

Algorithm 1: Ejection chain heuristic

instance of SKU partition, the heuristic initiates an ejection chain from every SKU s_i (Line 3), $i \in \{1, \ldots, n\}$, i.e. there are n ejection chains of maximum length $maxChainLength$ (maximum number of moves that a chain is composed of) to be explored based on the input solution. If this results in an improved solution, this solution is used to restart the overall process (loop 2–19). A tabu list (Line 1) of length $tabuLength$ is used to prevent the algorithm from cycling when generating a chain. The first move of a chain is performed in Line 5. Here, the change of the objective function value of the input solution due to generating several infeasible solutions by tentatively deleting s_i from its present group and inserting it into every other group is analyzed. The best solution found in this manner is set as the reference solution ($refSol$) and the tabu list is updated. Hereafter, the best possible repair move is performed and the best solution ($bestSol$) known so far is potentially updated (Line 7). The corresponding move is determined by checking every potential move of non-tabu SKUs from the group of size $C+1$ to the group of size $C-1$ of the reference solution. For every other move of the chain, Algorithm 1 proceeds analogously (loop 8–16). Here, altering a reference solution (Line 11) corresponds to checking every potential move of all non-tabu SKUs of the source group ($sourceGroup$) that corresponds to the target group ($targetGroup$) of the last move of the chain (Lines 6 and 12) to every other group. The chain terminates, if $maxChainLength$ moves have been performed

or if all SKUs of *sourceGroup* are tabu (Line 9).

When implementing Algorithm 1, it is of specific importance to quickly compute the increase $\Delta(s_l, S_s, S_t)$, $l \in \{1, \dots, n\}$, $s, t \in \{1, \dots, k\}$, $s \neq t$, of the objective function value of a solution when performing a move of SKU s_l from source group S_s to target group S_t. In this context we define two matrices. First, the $m \times n$ matrix $A = (a_{\sigma i})$ is defined as in Section 2.2. An element $a_{\sigma i}$ of A takes the value 1, if SKU $s_i \in S$ is demanded within order $o_\sigma \in O$. Otherwise, it takes the value 0. Note that, given an instance of SKU partition, A is independent of a specific solution. This does not hold true for the second matrix $B(S_1, \dots, S_k) = (b_{\sigma j}(S_1, \dots, S_k))$ of size $m \times k$, where the elements $b_{\sigma j}$ are defined to be the number of SKUs of order $o_\sigma \in O$ that are included in group S_j, $j \in \{1, \dots, k\}$ (*order-group matrix*). In order to ease notation, however, we will write B instead of $B(S_1, \dots, S_k)$ in the following, as the corresponding solution will always become clear from the context. After initialization of both matrices at the beginning of the ejection chain method, we need to make sure to update matrix B upon altering a solution. Given A, B, s_l, and $s \neq t$, the computation of $\Delta(s_l, S_s, S_t)$ is fairly straightforward (see Appendix A): After initializing $\Delta(s_l, S_s, S_t) = 0$, we iterate over all orders $o_\sigma \in O$. If s_l is included in a specific order o_σ and if $b_{\sigma s} = 1$ and $b_{\sigma t} \geq 1$, then $\Delta(s_l, S_s, S_t)$ decreases by w_σ. If, however, $b_{\sigma s} > 1$ and $b_{\sigma t} = 0$, then $\Delta(s_l, S_s, S_t)$ increases by w_σ.

3.2. Branch and bound

We will now describe a branch and bound method for the SKU partition problem. It is based on model (8)–(16) and adapts some of the ideas of Dorndorf & Pesch (1994) and Jaehn & Pesch (2013). The notation used throughout this section is similar to the one in Jaehn & Pesch (2013). It is presented in Table 1.

Table 1: Branch and bound notation

Symbol	Definition	Description
λ	$\lambda \in \{0, 1, \dots\}$	Node of the search tree
Z	$Z := \{z_{ij} \mid i, j \in \{1, \dots, n\}, i < j\}$	Binary variables (7)
Z^λ	$Z^\lambda \subseteq Z$	z variables fixed to zero or one at λ
$\bar{Z}^\lambda$	$\bar{Z}^\lambda \subseteq Z^\lambda$	z variables fixed to one at λ
$\delta^\lambda(s_i)$	$\delta^\lambda(s_i) := \sum_{z_{ji} \in \bar{Z}^\lambda} z_{ji} + \sum_{z_{ij} \in \bar{Z}^\lambda} z_{ij}$	min. size of group that contains $s_i \in S$ at λ

We enumerate the nodes of the search tree by $\lambda \in \{0, 1, \dots\}$. The root node $\lambda = 0$ corresponds to the SKU partition instance under consideration. Here, we determine a feasible solution by any heuristic that is then improved by calling the ejection chain method described in Section 3.1. The upper bound ub is set to the corresponding objective function value.

The search tree is constructed by branching over z variables as defined in (7). It is system-

atically explored through depth-first or, alternatively, best-first search. When branching over a specific $z_{ij} \in Z$, two child nodes of node λ are constructed by fixing z_{ij} to one in $\lambda + 1$ (z_{ij} is added to $\bar{Z}^{\lambda+1}$ and $Z^{\lambda+1}$) and to zero in $\lambda + 2$ (z_{ij} is added to $Z^{\lambda+2}$). In both child nodes we may fix further variables implicitly through constraint propagation.

For each node of the search tree a lower bound is determined. A node is fathomed if its lower bound is larger than the global upper bound minus the smallest order weight of the SKU partition instance, if all variables are fixed such that they define a feasible solution of the input instance, or if (some) variables are fixed such that they necessarily result in infeasibility. In the second case, we check for a potential improvement of the global upper bound. If a node is not fathomed, it is subject of branching.

In the remainder of this section we will describe details of the lower bounds in Section 3.2.1 as well as the branching procedure and the constraint propagation in Section 3.2.2. We will then summarize the branch and bound method in Section 3.2.3.

3.2.1. Lower bounds

A lower bound of the objective function value of an instance of SKU partition can easily be determined by computing the minimum number of groups needed to store the SKUs included in each order.

Proposition 1 (Lower bound $\bar{lb}^0$)**.** *Consider an instance of SKU partition with optimal objective function value F^*. Then*

$$F^* \geq \sum_{\sigma=1}^{m} w_\sigma \left\lceil \frac{|o_\sigma|}{C} \right\rceil =: \bar{lb}^0.$$

The lower bound presented in Proposition 1 can similarly be applied to the nodes λ of the search tree within the branch and bound method when analyzing the z variables that have been fixed to one in λ.

Proposition 2 (Lower bound $\bar{lb}^\lambda$)**.** *Consider an instance of SKU partition with optimal objective function value F^*. Furthermore, let λ be an arbitrary node of the search tree. Then*

$$F^* \geq \sum_{\sigma=1}^{m} w_\sigma \left\lceil \frac{|o_\sigma| + \bar{l}(o_\sigma, \bar{Z}^\lambda)}{C} \right\rceil =: \bar{lb}^\lambda, \tag{17}$$

where

$$\bar{l}(o_\sigma, \bar{Z}^\lambda) := \sum_{s_j \in S \setminus o_\sigma} H\Big(\sum_{s_l \in o_\sigma, z_{lj} \in \bar{Z}^\lambda} z_{lj} + \sum_{s_l \in o_\sigma, z_{jl} \in \bar{Z}^\lambda} z_{jl} \Big).$$

Proof. Observe that for all orders $o_\sigma \in O$, the value $\bar{l}(o_\sigma, \bar{Z}^\lambda)$ corresponds to the number of SKUs that are not included in the order but must be stored in a group together with at

least one element of the order due to a z variable that has been fixed to one. Hence, the minimum number of groups needed to store the SKUs of o_σ including these $\bar{l}(o_\sigma, \bar{Z}^\lambda)$ SKUs is $\lceil \big(|o_\sigma| + \bar{l}(o_\sigma, \bar{Z}^\lambda)\big)/C \rceil$. This directly leads to the definition of the lower bound (17). □

While the definition of $\bar{lb}^\lambda$ in Proposition 2 is based on the set $\bar{Z}^\lambda$, one can analogously consider the z variables that have been fixed to zero in a node λ of the search tree to define a lower bound $\hat{lb}^\lambda$. To do so, define $\hat{l}(o_\sigma, Z^\lambda \setminus \bar{Z}^\lambda)$ to be the maximum number of SKUs within an order $o_\sigma \in O$ that pairwise cannot be stored within the same group due to the corresponding variables having been fixed to zero in λ. Then the bound $\hat{lb}^\lambda$ is determined as described in Proposition 3. Again, it is based on Proposition 1.

Proposition 3 (Lower bound $\hat{lb}^\lambda$)**.** *Consider an instance of SKU partition with optimal objective function value F^*. Furthermore, let λ be an arbitrary node of the search tree. Then*

$$F^* \geq \sum_{\sigma=1}^{m} w_\sigma \max \left\{ \hat{l}(o_\sigma, Z^\lambda \setminus \bar{Z}^\lambda), \left\lceil \frac{|o_\sigma|}{C} \right\rceil \right\} =: \hat{lb}^\lambda. \tag{18}$$

For each node λ of the branch and bound tree, we define

$$lb^\lambda := \max\{\bar{lb}^\lambda, \hat{lb}^\lambda\}.$$

3.2.2. Branching and constraint propagation

Given a non-fathomed node λ of the search tree, we generate two child nodes $\lambda + 1$ and $\lambda + 2$ by branching over a variable $z_{ij} \in Z \setminus Z^\lambda$. This variable is fixed to one in $\lambda + 1$ and zero in $\lambda + 2$. It is chosen such that the difference of the lower bounds $\bar{lb}^{\lambda+1} - \bar{lb}^\lambda$ is maximized. When determining the branching variable, subsequent constraint propagation is not considered. However, we fix those variables to zero (in nodes $\lambda + 1$ and $\lambda + 2$) that, if they were chosen as branching variables would result in problem $\lambda + 1$ being directly fathomed due to its resulting (large) lower bound. If $\bar{lb}^{\lambda+1} - \bar{lb}^\lambda = 0$ no matter which branching variable is chosen, a random branching variable is selected. Details on the branching procedure are given in Appendix B.1.

After each branching we perform logical tests that are inspired by Jaehn & Pesch (2013), who propose to check whether variables can be fixed due to logical implications based on the transitivity constraints (or "triangular conditions") (10)–(12). For SKU partition, we additionally check whether variables have to be fixed to zero or one in order to not exceed or guarantee the desired group size C (constraints (9)). Details are presented in Appendix B.2. The logical tests based on the triangular conditions after branching over variable z_{ij}, $s_i, s_j \in S$, are illustrated in Figure 5. Consider, for example, case 1 of Figure 5. The branching variable z_{ij} has been fixed to one in node λ and there exists a (previously) fixed variable $z_{\bar{j}i} \in \bar{Z}^\lambda$, $\bar{j} < i$.

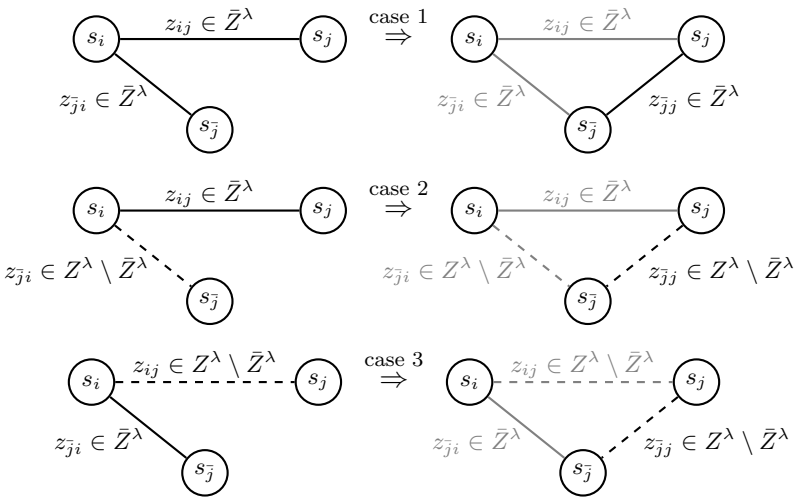


Figure 5: Constraint propagation: triangular conditions

Then $z_{\bar{j}\bar{j}}$ must also be fixed to one. Cases 2 and 3 are similar.

The process of fixing additional variables is propagated until no more propagation is possible (see Appendix B.2). Based on the variables that have been fixed during this procedure in a node λ of the search tree, the lower bound lb^λ needs to be updated. The corresponding procedure is presented in Appendix B.3.

3.2.3. Branch and bound algorithm

The branch and bound algorithm in its depth-first version is summarized in Algorithm 2.

4. Computational study

In order to analyze the mathematical models presented in Section 2.2 and to assess the performance of the algorithms introduced in Section 3, we ran extensive computational tests. The results are presented in Section 4.1. Additional tests were performed to analyze the effectiveness of our group-based view with respect to picker idle time, which is subject of Section 4.2. All tests were performed on an Intel Core i7-4770 CPU at 3.4GHz and 16GB of RAM, running Windows 8 64bit. All algorithms were implemented in C++ (Microsoft Visual Studio 2010). We used IBM ILOG CPLEX in version 12.5 with 64bit.

Throughout this section we will use the following notation. E refers to the ejection chain method. C_X and C_Z refer to calling CPLEX for model (1)–(6) and (8)–(16), respectively. Similarly, B refers to Algorithm 2. The upper bound determined by an exact approach $j \in \{C_X, C_Z, B\}$ for a specific instance I of SKU partition within a given time limit is referred to as $ub_j(I)$. Similarly, the smallest lower bound of all not yet fathomed nodes (the child nodes of which have not yet been constructed) within the search tree of an algorithm $j \in \{C_X, C_Z, B\}$ (global lower bound) upon termination is referred to as $lb_j(I)$. The objective function value of

Data: SKU partition instance I with at least one feasible solution
Result: optimal solution to I
`// Step 0: initialize root node` $\lambda = 0$
determine feasible solution $(S_1, \ldots, S_k)$ by any heuristic;
call ejection chain heuristic (Section 3.1) to potentially improve solution $(S_1, \ldots, S_k)$, and initialize upper bound ub;
determine lower bound $\bar{lb}^0$ (17) and set $lb^0 = \bar{lb}^0 = \hat{lb}^0$;
$Z^0 = \bar{Z}^0 = \emptyset$; $\bar{l}(o_\sigma, \bar{Z}^0) = \hat{l}(o_\sigma, Z^0 \setminus \bar{Z}^0) = 0$ for all $o_\sigma \in O$;
$candidateList = \{0\}$;
while $candidateList \neq \emptyset$ **do**
  pop element λ from $candidateList$;
  if $lb^\lambda \leq ub - \min_{\sigma \in \{1,\ldots,m\}} w_\sigma$ **then**
    `// Step 1: branch`
    determine branching variable z_{ij} (and potentially fix variables to zero in $\lambda + 1$ and $\lambda + 2$);
    add nodes $\lambda + 1$ ($z_{ij} = 1$) and $\lambda + 2$ ($z_{ij} = 0$) to $candidateList$;
    `// Step 2: propagate constraints`
    execute constraint propagation procedure to determine sets of additionally fixed variables in $\lambda + 1$ and $\lambda + 2$;
    update $Z^{\lambda+1}$, $\bar{Z}^{\lambda+1}$, $Z^{\lambda+2}$, $\bar{Z}^{\lambda+2}$;
    `// Step 3: update lower bounds`
    update lower bounds $lb^{\lambda+1}$ and $lb^{\lambda+2}$ based on additionally fixed variables;
    `// Step 4: terminate`
    for $\hat{\lambda} \in \{\lambda + 1, \lambda + 2\}$ **do**
      if *infeasibility is detected* **then**
        remove $\hat{\lambda}$ from $candidateList$;
      end
      if $Z^{\hat{\lambda}} = Z$ *and corresponding solution is feasible* **then**
        potentially update ub and remove $\hat{\lambda}$ from $candidateList$;
      end
    end
  end
end
return solution that corresponds to ub;

Algorithm 2: Branch and bound algorithm

the best solution to instance I determined by the ejection chain method is denoted by $\hat{F}_E(I)$. Finally, we define $\hat{C}(I) = \underset{j \in \{C_X, C_Z\}}{\operatorname{argmin}} \; ub_j(I)$.

The parameters for the ejection chain heuristic were set based on preliminary tests. As a compromise between running time and solution quality, these tests resulted in choosing $maxChainLength = \lfloor 0.9 \cdot k \rfloor$ and $tabuLength = \lfloor 0.6 \cdot maxChainLength \rfloor$. We apply a fairly simple heuristic to determine a first feasible solution within the ejection chain framework. This heuristic first sorts the orders by nonincreasing order sizes. The groups are then filled one after another by running through the SKUs of these orders and, if they have not yet been assigned to a group, inserting them into the first group that does not yet contain C elements. SKUs that are not part of any order are similarly assigned in the last phase of the heuristic.

4.1. Comparison of models and performance of algorithms

We generated random sets of test instances with n ranging from 10 to 200 and different group sizes, $2 < k \leq n/2$, as presented in Table 2. The large instances ($n \geq 100$) are motivated

Table 2: Size of random test instances

n	10	12	14	15	16	18	20	50	100	200
k	5	3,4,6	7	3,5	4,8	3,6,9	4,5,10	5,10,25	4,5,10,20,25,50	4,5,10,20,25,50,100

from the size of real world carousel racks. Consider, for example, the different "pan carousel" models of a well known manufacturer of vertical storage solutions (Vidir, 2015). A shelf ("carrier") of such a pan carousel is around three meters wide and 0.3 meters high. The models feature 9 to 24 carriers. The pick face allows access to exactly one carrier. If we consider typical storage bins being used to store small to medium sized SKUs within the carousel, it is therefore reasonable to assume a capacity of around 10 bins per carrier. This directly relates to the large instance presented in Table 2. With respect to the other parameters, we set m to 50, 100, or 200. For each resulting combination of n, k, and m, three sets of test instances were constructed by randomly generating order sizes $|o_\sigma|$ from a uniform distribution over the interval $[1, \lfloor \frac{n}{5} \rfloor]$ in the first set, $[1, \lfloor \frac{n}{2} \rfloor]$ in the second set, and $[1, \lfloor \frac{2n}{3} \rfloor]$ in the third set, for all $\sigma = 1, \ldots, m$. The corresponding order weights w_σ were drawn from a uniform distribution over the interval $[1, 10]$. The SKUs of each order were randomly selected. Five instances were generated in each set, which results in a total of 1,395 test instances.

We first determined solutions to all test instances I with CPLEX (in depth-first search mode) for both model formulations (1)–(6) and (8)–(16) with a time limit of 60 seconds (for each instance and each model formulation) and with the ejection chain method. In these tests, C_X or C_Z found at least one feasible solution for all considered instances. The same is true for E. Our measure of quality for the latter solution is the ratio

$$Q_E(I) := 100 \cdot \frac{ub_{\hat{C}(I)}(I) - \hat{F}_E(I)}{ub_{\hat{C}(I)}(I)}.$$

Table 3 presents corresponding results on the average quality of the ejection chain heuristic for the small test instances, $n \leq 18$, in the third column. The fourth column is related to the percentage of instances of each set that were solved to optimality (including proving optimality) by CPLEX for at least one of the model formulations within the time limit of 60 seconds. The average running times of CPLEX, $t^{\hat{C}}_{avg}$, and the ejection chain heuristic, t^{E}_{avg}, are given in the last two columns of the table. For the computation of $t^{\hat{C}}_{avg}$, we use the computational times that correspond to $\hat{C}(I)$ for all instances I within the specific set. If $ub_{C_X}(I) = ub_{C_Z}(I)$ for an instance I, the faster model is chosen. We find that most instances were optimally solved by CPLEX. The average quality ratios show the ejection chain heuristic to perform slightly worse than CPLEX in terms of solution quality. However, the solutions are very close to being optimal on average. The average running times of the ejection chain heuristic are in the range

Table 3: Ejection chain performance for small instances

n	k	avg. quality[a]	opt. [%][b]	$t^{\hat{C}}_{avg}$ [s][c]	t^{E}_{avg} [s][c]
10	5	-0.04	100	0.004	0.001
12	3	-0.24	100	0.54	0.001
	4	-0.1	100	3.32	0.001
	6	-0.09	100	8.01	0.001
14	7	-0.07	100	12.01	0.002
15	3	-0.45	100	3.25	0.001
	5	-0.13	76	26.2	0.002
16	4	-0.33	73	23.14	0.002
	8	-0.16	100	16.02	0.003
18	3	-0.8	96	13.02	0.001
	6	0.05	44	41.27	0.003
	9	-0.15	100	10.69	0.004

[a] Average of quality ratios $Q_E(I)$ of all instances I within set.
[b] Percentage of instances of the set that were solved to optimality by CPLEX.
[c] Average comp. time used by CPLEX (index $\hat{C}$) or the ejection chain heuristic (index E).

of only a few milliseconds.

Table 4 presents results on the quality of the ejection chain heuristic for the medium ($20 \leq n \leq 50$) and large ($n \geq 100$) instances. In contrast to the results for small instances,

Table 4: Ejection chain performance for medium and large instances

n	k	avg. quality[a]	opt. [%][b]	$t^{\hat{C}}_{avg}$ [s][c]	t^{E}_{avg} [s][c]
20	4	-0.26	42	43.99	0.003
	5	0.21	33	42.58	0.003
	10	-0.18	100	5.36	0.006
50	5	1.5	0	60	0.033
	10	6.35	0	60	0.071
	25	-0.65	100	0.54	0.089
100	4	-1.93	0	60	0.128
	5	0.3	0	60	0.191
	10	9.09	0	60	0.455
	20	12.7	0	60	0.797
	25	12.86	0	60	0.844
	50	-1.22	98	7.84	0.687
200	4	-4.1	0	60	0.503
	5	-1.8	0	60	0.909
	10	8.39	0	60	3.254
	20	15.84	0	60	5.318
	25	17.39	0	60	6.227
	50	18.95	0	60	7.671
	100	6.87	24	59.68	6.333

[a] Average of quality ratios $Q_E(I)$ of all instances I within set.
[b] Percentage of instances of the set that were solved to optimality by CPLEX.
[c] Average comp. time used by CPLEX (index $\hat{C}$) or ejection chain heuristic (index E).

CPLEX was not able to find optimal solutions within the time limit for most instances. This results in the average quality ratios being positive for most instance sets, i.e., the ejection chain heuristic finding better solutions than CPLEX on average. Only for the cases of very few or plenty groups, the ejection chain heuristic performs worse on average. In terms of computational time, the ejection chain heuristic uses less than eight seconds on average for all instance sets. The maximum computational time we encountered is around 20 seconds for an instance with $n = 200$, $k = 100$, $m = 200$ and a maximum order size of 100.

To compare the performance of CPLEX for models (1)–(6) and (8)–(16) and to test the performance of the branch and bound method presented in Algorithm 2, we ran tests with time limits of 540 seconds for each small instance and 360 seconds for each medium or large instance for each exact method. If an algorithm $j \in \{C_X, C_Z, B\}$ fails to find an optimal solution (and prove its optimality) to an instance I within this time limit, we define the optimality gap by

$$gap_j(I) := 100 \cdot \frac{ub_j(I) - lb_j(I)}{lb_j(I)}.$$

Table 5 presents the corresponding results for the small test instances. We applied depth-first

Table 5: CPLEX and branch and bound for small instances

		CPLEX, model (1)–(6)				CPLEX, model (8)–(16)				Branch and bound, Alg. 2			
n	k	opt.[a] [%]	$rank^{C_X}_{avg}$ [b]	$gap^{C_X}_{avg}$ [c]	$t^{C_X}_{avg}$ [d] [s]	opt.[a] [%]	$rank^{C_Z}_{avg}$ [b]	$gap^{C_Z}_{avg}$ [c]	$t^{C_Z}_{avg}$ [d] [s]	opt.[a] [%]	$rank^{B}_{avg}$ [b]	gap^{B}_{avg} [c]	t^{B}_{avg} [d] [s]
10	5	100	-	-	1.24	100	-	-	0.004	100	-	-	0.05
12	3	100	-	-	0.57	100	-	-	5.46	100	-	-	2.57
	4	100	-	-	4.85	100	-	-	4.18	100	-	-	4.26
	6	100	-	-	23.22	100	-	-	0.01	100	-	-	0.55
14	7	82	1	77.53	47.79	100	-	-	0.01	100	-	-	7.54
15	3	100	-	-	3.3	89	2.6	342.27	139.71	100	-	-	138.71
	5	80	1	90.41	91.18	87	2	13.96	27.81	67	1.13	40.75	134.09
16	4	87	1	59.65	53.91	56	2.35	44.37	108.64	33	1.2	46.77	172.20
	8	62	1.94	113.82	70.18	100	-	-	0.01	98	1	27.97	109.02
18	3	100	-	-	13.64	36	2.9	neg.	32.28	11	1.73	50.04	133.84
	6	38	1.79	115.56	112.95	56	2.15	14.28	66.24	13	1.51	46.24	123.89
	9	47	2.08	135.20	88.79	100	-	-	0.02	56	1.2	29.98	120.50

[a] Percentage of instances of the set that were solved to optimality by alg. $j \in \{C_X, C_Z, B\}$.
[b] Average rank $rank^j(I)$ of the upper bound determined by alg. $j \in \{C_X, C_Z, B\}$ for instances I within the set that were not solved to optimality.
[c] Average gap $gap_j(I)$ resulting from algorithm $j \in \{C_X, C_Z, B\}$ for all instances I within the set that were not solved to optimality by j. Negative average gaps are indicated by "neg."
[d] Average comp. time for instances within the set that were solved to optimality by algorithm $j \in \{C_X, C_Z, B\}$.

search for all algorithms and all instances. At least one feasible solution was found by all algorithms for all considered instances. For each algorithm $j \in \{C_X, C_Z, B\}$, the table includes the percentage of instances of the test sets solved to optimality within the time limit (including the proof of optimality), the corresponding average computational time, t^j_{avg}, as well as the average gap, gap^j_{avg}, of the instances within each set that were not solved to optimality within the time limit. Let I be an instance that was not solved to optimality by a specific algorithm. Then we rank the corresponding upper bound by calling Algorithm 3. Table 5 includes the average rank, $rank^j_{avg}$, of the upper bounds for the instances within each set that were not solved to optimality within the time limit by the corresponding algorithm j. We find that C_X tends to be better than C_Z in terms of the number of instances solved to optimality and the corresponding computational times for large ratios n/k, while the opposite is true for small ratios n/k. For the instances that were not solved to optimality by CPLEX, C_X tends to result in better upper bounds than C_Z. B's computational times for determining optimal solutions

Data: SKU partition instance I; upper bounds $ub_j(I)$, $j \in \{C_X, C_Z, B\}$; algorithm $\bar{j} \in \{C_X, C_Z, B\}$ that did not solve I to optimality
Result: $rank^{\bar{j}}(I)$

initialize list $L(I) = (ub_{C_X}(I), ub_{C_Z}(I), ub_B(I))$;
sort the elements of $L(I)$ in nondecreasing order;
if $ub_{\bar{j}}(I)$ = *value of first element of* $L(I)$ **then** $rank^{\bar{j}}(I) = 1$;
else
    if $ub_{\bar{j}}(I)$ = *value of second element of* $L(I)$ **then** $rank^{\bar{j}}(I) = 2$;
    else if $ub_{\bar{j}}(I)$ = *value of third element of* $L(I)$ **then** $rank^{\bar{j}}(I) = 3$;
end

Algorithm 3: Ranking a solution

are relatively large. However, the upper bounds determined by B are of high quality. When compared to C_X, the optimality gap resulting from B is (on average) smaller for the instances that were not solved to optimality.

Table 6 compares the performance of the exact algorithms for the medium and large instances. We applied depth-first search for the medium instances, while, for the large instances,

Table 6: CPLEX and branch and bound for medium and large instances

		CPLEX, model (1)–(6)				CPLEX, model (8)–(16)				Branch and bound, Alg. 2			
n	k	opt.[a] [%]	$rank^{C_X}_{avg}$ [b]	$gap^{C_X}_{avg}$ [c]	$t^{C_X}_{avg}$ [d] [s]	opt.[a] [%]	$rank^{C_Z}_{avg}$ [b]	$gap^{C_Z}_{avg}$ [c]	$t^{C_Z}_{avg}$ [d] [s]	opt.[a] [%]	$rank^{B}_{avg}$ [b]	gap^{B}_{avg} [c]	t^{B}_{avg} [d] [s]
20	4	56	1.35	88.63	49.77	33	2.93	167.51	18.16	0	1.93	63.81	-
	5	33	1.37	101.26	67.90	33	2.9	54.66	4.39	0	1.67	58.05	-
	10	22	2.37	149.16	48.27	100	-	-	0.03	16	1.18	30.89	97.95
50	5	0	1.56	140.58	-	0	3	neg.	-	0	1.44	107.88	-
	10	0	2.04	266.81	-	0	2.91	neg.	-	0	1.04	93.6	-
	25	0	3	470.82	-	100	-	-	0.54	0	1.98	40.72	-
100	4	0	1.22	72.39	-	0	3	$-\infty$[f]	-	0	1.78	118.67	-
	5	0	1.4	116.70	-	0	3	$-\infty$[f]	-	0	1.6	128.66	-
	10	0	2	325.32	-	0	3	$-\infty$[f]	-	0	1	135.31	-
	20	0	2.33	699.59	-	0	2.67	$-\infty$[f]	-	0	1	107.77	-
	25	0	2.27	802.53	-	0	2.73	$-\infty$[f]	-	0	1	93.57	-
	50	0	3	1285.62	-	98	1	0.01	6.71	0	2	43.44	-
200	4	0	1.07	105.79	-	0	3	$-\infty$[f]	-	0	1.93	148.53	-
	5	0	1.29	163.92	-	0	3	$-\infty$[f]	-	0	1.71	162.64	-
	10	0	2.02	457.18	-	0	2.96	$-\infty$[f]	-	0	1.02	179.38	-
	20	0	2.07	956.96	-	0	2.93	$-\infty$[f]	-	0	1	159.09	-
	25	0	2.02	1213.35	-	0	2.98	$-\infty$[f]	-	0	1	145.83	-
	50	0	2.29	2448.97	-	0	2.71	$-\infty$[f]	-	0	1	97.78	-
	100	0	3	∞[e]	-	87	1	0.01	89.77	0	2	44.78	-

[a] Percentage of instances of the set that were solved to optimality by alg. $j \in \{C_X, C_Z, B\}$.
[b] Average rank $rank^j(I)$ of the upper bound determined by alg. $j \in \{C_X, C_Z, B\}$ for instances I within the set that were not solved to optimality.
[c] Average gap $gap_j(I)$ resulting from algorithm $j \in \{C_X, C_Z, B\}$ for all instances I within the set that were not solved to optimality by j. Negative average gaps are indicated by "neg." If algorithm $j \in \{C_X, C_Z, B\}$ does not find a feasible solution to an instance I, we have $ub_j(I) = \infty$.
[d] Average comp. time for instances within the set that were solved to optimality by algorithm $j \in \{C_X, C_Z, B\}$.
[e] $gap^{C_X}_{avg} = \infty$ due to at least one instance I with $lb_{C_X}(I) = 0$ in set.
[f] $gap^{C_Z}_{avg} = -\infty$ due to at least one instance I with $ub_{C_Z}(I) = \infty$ in set.

we applied best-first search, because preliminary tests on large instances had shown that all algorithms tend to perform better with respect to the global bounds on the objective function value in this mode. Obviously, the strength of C_Z for small ratios n/k carries over to medium and large instances. C_X, however, is not able to find optimal solutions within the time limit

for instances with $n \geq 50$, even in the case of large ratios n/k. In most cases, C_Z is not able to determine feasible solutions for large instances with $k < n/2$, while C_X and B find at least one feasible solution for all considered instances. As a result (and analogously to our findings for the small instances), C_X tends to result in better upper bounds than C_Z in case of the instances that were not solved to optimality by CPLEX. B, on average, results in the best upper bounds with the smallest optimality gaps, while solving the least instances to optimality (including the prove of optimality).

4.2. Effect on picker idle time

As we have stated in Section 1.1, a major source of picker idle time when using carousels is the time it takes to rotate the carousel in order to change the set of shelves presented in the pick face. This fact has given rise to the group-based perspective taken in this paper. In this section we will analyze the effectiveness of this perspective with respect to picker idle time by comparing an organ pipe arrangement and a random assignment of storage positions to SKUs with grouping the SKUs based on the branch and bound algorithm presented in this paper.

As before, we will consider real-world carousel racks with a shelf height of around 0.3 meters and a capacity of 10 bins per shelf. The retrieval speed of such a pan carousel is 8077 mm per minute (Vidir, 2015). Thus, it takes about 2.2 seconds to rotate the carousel by one shelf and thus change the group of SKUs presented in the pick face.

Our test sets were randomly generated as described in Section 4.1 and feature $k \in \{15, 20, 25\}$ shelves, $n = 10k$ SKUs, and $m \in \{100, 200\}$ orders. For each combination of k and m, three sets of test instances were constructed by randomly generating order sizes $|o_\sigma|$ from a uniform distribution over the interval $[1, \lfloor \frac{n}{10} \rfloor]$ in the first set, $[1, \lfloor \frac{n}{5} \rfloor]$ in the second set, and $[1, \lfloor \frac{n}{2} \rfloor]$ in the third set, for all $\sigma = 1, \ldots, m$. The corresponding order weights w_σ were drawn from a uniform distribution over the interval $[1, 10]$. Five instances were generated in each set.

We applied three algorithms to solve a given instance of SKU partition. First, we applied Algorithm 2 (in depth-first search mode) with a time limit of 540 seconds. In line with Section 1.1 and our simplifying assumptions described in Section 2.3, the assignment of groups to specific shelves after a solution to an instance of SKU partition has been determined is left for future research. Hence, for this algorithm, we proceeded by simply numbering the shelves of the carousel from one to k and assigning group $j \in \{1, \ldots, k\}$ to shelf j of the carousel. Second, we randomly assigned storage positions to SKUs. Third, we applied the organ pipe arrangement as referenced in Section 1.1. Basically, the organ pipe arrangement first sorts the SKUs in nonincreasing order of their overall demand. Then, based on this ordering, the SKUs are assigned to the shelves of the carousel in groups of C. The first group (high level of

demand) is assigned to shelf $h = \lceil (k+1)/2 \rceil$. The next groups are assigned to shelves $h-1$, $h+1$, $h-2$ and so forth.

The idle time induced by picking a specific order was approximated by determining the smallest possible time needed for rotating the carousel to all necessary shelves, while assuming that the first shelf can immediately be accessed and that the carousel can rotate in both directions. The sequence of picking-orders was neglected (see Section 2.3). Hence, the total idle time induced by picking all orders was computed by multiplying the idle time for each order with its order weight and summing over all orders. The results are presented in Table 7.

Table 7: Picker idle time

			t_{avg}^{idle} $[min]$[a]			ratio[b]	
n	k	m	Algorithm 2	random	organ pipe	random	organ pipe
150	15	100	181.17	212.88	202.71	1.18	1.12
		200	392.92	436.21	429	1.11	1.09
200	20	100	267.74	305.99	294.34	1.14	1.1
		200	548.42	602.56	589.71	1.1	1.08
250	25	100	338.67	387.05	369.33	1.14	1.09
		200	725.04	788.02	770.26	1.09	1.06

[a] Average picker idle time for corresponding instance set.
[b] Ratio of average picker idle times (compared to Algorithm 2) for corresponding instance set.

We conclude that using Algorithm 2 results in substantially smaller (approximate) average picker idle times when compared to random assignment of SKUs to storage positions or the organ pipe arrangement. The larger computational effort needed to compute the corresponding partitions therefore tends to pay off quickly.

5. Conclusion

In this paper we have introduced the SKU partition problem, which is a basic partition problem that subdivides stock keeping units (SKUs) into disjoint subsets, such that the minimum number of groups has to be accessed when retrieving a given order set under a pick-by-order policy. We have presented applications of SKU partition in a wide range of storage systems that are based on separating their storage space into groups of SKUs stored in separate areas. Furthermore, we have shown that SKU partition is NP-hard in the strong sense and we have presented two MIP models. One of them is based on a well-known model formulation for the clique partitioning problem. We have, then, developed an ejection chain heuristic as well as a branch and bound procedure. In a computational study we have shown that the ejection chain heuristic tends to provide high-quality solutions within a very short time. Furthermore, with respect to applying the branch and bound method and solving SKU partition with CPLEX based on the different model formulations, we have shown that CPLEX is well suited for small to medium sized models. When the number of groups is relatively large, the clique partitioning

based model formulation performs best. For large instances, the branch and bound procedure introduced in this paper results in the best upper bounds on the objective function value and the smallest optimality gaps. Finally, we have shown that the perspective taken in this paper results in substantially smaller (approximate) average picker idle times when compared to an organ pipe arrangement or a random assignment of SKUs to storage positions in carousel racks.

Future research should extend our basic SKU partition problem by integrating further peculiarities relevant for different storage systems. The rotation time of carousels between groups, for instance, heavily depends on their assignment to specific shelf segments. In a mobile rack setting, also the detailed slotting of SKUs within a group influences the picking effort, if frequently demanded items are stored in the front parts of the racks.

References

Abdel-Malek, L., & Tang, C. (1994). A heuristic for cyclic stochastic sequencing of tasks on a drum-like storage system. *Computers & Operations Research*, *21*, 385–396.

Bartholdi III, J. J., & Hackman, S. T. (2014). *Warehouse & distribution science: Release 0.96*. Atlanta: Georgia Institute of Technology.

Bengü, G. (1995). An optimal storage assignment for automated rotating carousels. *IIE Transactions*, *27*, 105–107.

Boysen, N., Briskorn, D., & Emde, S. (2014). Sequencing of picking orders in mobile rack warehouses. Working Paper, Friedrich-Schiller-University Jena.

Boysen, N., Emde, S., Hoeck, M., & Kauderer, M. (2015). Part logistics in the automotive industry: Decision problems, literature review and research agenda. *European Journal of Operational Research*, *242*, 107–120.

Boysen, N., & Stephan, K. (2013). The deterministic product location problem under a pick-by-order policy. *Discrete Applied Mathematics*, *161*, 2862–2875.

Chang, T.-H., Fu, H.-P., & Hu, K.-Y. (2007). A two-sided picking model of M-AS/RS with an aisle-assignment algorithm. *International Journal of Production Research*, *45*, 3971–3990.

Dorndorf, U., Jaehn, F., & Pesch, E. (2008). Modelling robust flight-gate scheduling as a clique partitioning problem. *Transportation Science*, *42*, 292–301.

Dorndorf, U., & Pesch, E. (1994). Fast clustering algorithms. *ORSA Journal on Computing*, *6*, 141–153.

Garey, M. R., Johnson, D. S., & Stockmeyer, L. (1976). Some simplified NP-complete graph problems. *Theoretical Computer Science*, *1*, 237–267.

Glock, C. H., & Grosse, E. H. (2012). Storage policies and order picking strategies in U-shaped order-picking systems with a movable base. *International Journal of Production Research*, *50*, 4344–4357.

Glover, F. (1996). Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics*, *65*, 223–253.

Ha, J.-W., & Hwang, H. (1994). Class-based storage assignment policy in carousel system. *Computers & Industrial Engineering*, *26*, 489–499.

Hassini, E. (2008). Storage space allocation to maximize inter-replenishment times. *Computers & Operations Research*, *35*, 2162–2174.

Hu, K.-Y., Chang, T.-H., Fu, H.-P., & Yeh, H. (2009). Improvement order picking in mobile storage systems with a middle cross aisle. *International Journal of Production Research*, *47*, 1089–1104.

Hwang, H., & Ha, J.-W. (1991). Cycle time models for single/double carousel system. *International Journal of Production Economics*, *25*, 129–140.

Hwang, H., & Ha, J.-W. (1994). An optimal boundary for two class-based storage assignment policy in carousel system. *Computers & Industrial Engineering*, *27*, 87–90.

Jacobs, D. P., Peck, J. C., & Davis, J. S. (2000). A simple heuristic for maximizing service of carousel storage. *Computers & Operations Research*, *27*, 1351–1356.

Jaehn, F., & Pesch, E. (2013). New bounds and constraint propagation techniques for the clique partitioning problem. *Discrete Applied Mathematics*, *161*, 2025–2037.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller, & J. W. Thatcher (Eds.), *Complexity of Computer Computations* (pp. 85–104). New York: Plenum Press.

Karpinski, M. (2002). Approximability of the minimum bisection problem: An algorithmic challenge. In K. Diks, & W. Rytter (Eds.), *Mathematical Foundations of Computer Science 2002 - 27th International Symposium, MFCS 2002, Warsaw, Poland, August 2002, Proceedings* (pp. 59–67). Berlin: Springer.

Kim, B. (2005). Maximizing service of carousel storage. *Computers & Operations Research*, *32*, 767–772.

Litvak, N. (2006). Optimal picking of large orders in carousel systems. *Operations Research Letters*, *34*, 219–227.

Litvak, N., & Vlasiou, M. (2010). A survey on performance analysis of warehouse carousel systems. *Statistica Neerlandica*, *64*, 401–447.

Meller, R. D., & Klote, J. F. (2004). A throughput model for carousel/VLM pods. *IIE Transactions*, *36*, 725–741.

Pesch, E., & Glover, F. (1997). TSP ejection chains. *Discrete Applied Mathematics*, *76*, 165–181.

SSI Schäfer (2014). Vollautomatisierte Verschieberegalanlage (in German). `http://www.ssi-schaefer.de/fileadmin/ssi/documents/navigationsbaum/logistiksysteme/vollautomatische_systeme/verschieberegallager/vollautomatisierte_v_de.pdf`. (last access: August 2015).

Vickson, R. G., & Fujimoto, A. (1996). Optimal storage locations in a carousel storage and retrieval system. *Location Science*, *4*, 237–245.

Vidir (2015). Pan carousel. `http://www.storevertical.com/products/vertical-storage-system/pan-carousel`. (last access: August 2015).

Appendix A. Details on the ejection chain heuristic

In this section we present details on the ejection chain heuristic introduced in Section 3.1. First, we present details on the computation of $\Delta(s_l, S_s, S_t)$ in Algorithm 4.

Data: matrices A and B; groups S_s, S_t, $s \neq t$; SKU s_l
Result: $\Delta(s_l, S_s, S_t)$
$\Delta(s_l, S_s, S_t) = 0$;
for *all orders* $o_\sigma \in O$ **do**
  if $a_{\sigma l} = 1$ **then**
    if $b_{\sigma s} = 1 \wedge b_{\sigma t} \geq 1$ **then** $\Delta(s_l, S_s, S_t) -= w_\sigma$;
    else if $b_{\sigma s} > 1 \wedge b_{\sigma t} = 0$ **then** $\Delta(s_l, S_s, S_t) += w_\sigma$;
  end
end

Algorithm 4: Computation of $\Delta(s_l, S_s, S_t)$

Given A, B, $s \in \{1, \ldots, k\}$, an SKU $s_l \in S_s$, and a value $\Delta^{best} = \infty$, Algorithm 5 determines the best move of $s_{\hat{l}} = s_l$ from S_s to $S_{\hat{t}}$, $\hat{t} \neq s$ and the corresponding $\Delta(s_{\hat{l}}, S_s, S_{\hat{t}})$.

Data: matrices A and B; group S_s; SKU $s_l \in S_s$; value Δ^{best}
Result: $s_{\hat{l}}$; $S_{\hat{t}}$; Δ^{best}
for *all groups* S_t, $t \neq s$ **do**
  compute $\Delta(s_l, S_s, S_t)$ with Alg. 4 (A, B, S_s, S_t, s_l);
  if $\Delta(s_l, S_s, S_t) < \Delta^{best}$ **then**
    $\Delta^{best} = \Delta(s_l, S_s, S_t)$;
    $s_{\hat{l}} = s_l$; $S_{\hat{t}} = S_t$;
  end
end

Algorithm 5: Determining the best move of specific SKU

Similarly, the computation of the change of the objective function value when generating a trial solution is presented in Algorithm 6.

Data: matrices A and B; groups S_s, S_t, $s \neq t$; *tabuList*
Result: Δ^{best}; $s_{\hat{l}}$
$\Delta^{best} = \infty$;
if *all SKUs of* S_S *are tabu* **then** pick a random element $s_{temp} \in S_s$ and temporarily treat it as non-tabu;
for *all non-tabu SKUs* s_l *in* S_s **do**
  compute $\Delta(s_l, S_s, S_t)$ with Alg. 4 (A, B, S_s, S_t, s_l);
  if $\Delta(s_l, S_s, S_t) < \Delta^{best}$ **then**
    $\Delta^{best} = \Delta(s_l, S_s, S_t)$; $s_{\hat{l}} = s_l$;
  end
end
restore the tabu status of s_{temp} if necessary;

Algorithm 6: Computation of objective function change when generating a trial solution

Algorithm 7 illustrates the details of the ejection chain heuristic. Note that we refer to the oder-group matrix of *inSol* with objective function value $F(inSol)$ by B_{in}. Similar notation is used for $refSol$ and $bestSol$.

Data: feasible solution $inSol$; parameters $maxChainLength$, $tabuLength$
Result: feasible solution $bestSol$

initialize $tabuList$;
$\Delta^{best} = \infty$;
compute A, B_{in}, and $F(inSol)$;
$bestSol = inSol$; $B_{best} = B_{in}$; $F(bestSol) = F(inSol)$;
do
  for *all groups* S_j, $j \in \{1, \ldots, k\}$, *of* $inSol$ **do**
    for *all SKUs* $s_i \in S_j$ **do**
      clear $tabuList$;
      $sourceGroup = S_j$; $\Delta^{best} = \infty$;
      determine $targetGroup$ and update Δ^{best} with Alg. 5 (A, B_{in}, $sourceGroup$, s_i, Δ^{best});
      generate $refSol$ from $inSol$ by moving s_i from $sourceGroup$ to $tagetGroup$;
      $F(refSol) = F(inSol) + \Delta^{best}$;
      update $tabuList$ and construct B_{ref} from B_{inSol};
      $sourceGroup = targetGroup$;
      Determine Δ^{best} and $s_{\hat{l}}$ with Alg. 6 (A, B_{ref}, $sourceGroup$, S_j, $tabuList$);
      if $F(refSol) + \Delta^{best} < F(bestSol)$ **then**
        generate $trialSol$ from $refSol$ by moving $s_{\hat{l}}$ from $sourceGroup$ to S_j;
        $bestSol = trialSol$; $F(bestSol) = F(refSol) + \Delta^{best}$; update B_{best};
      end
      for $chainLength = 2$ **to** $maxChainLength$ **do**
        if *all SKUs of* $sourceGroup$ *of* $refSol$ *are tabu* **then** break;
        else
          $\Delta^{best} = \infty$;
          for *all non-tabu SKUs* s_l *of* $sourceGroup$ *of* $refSol$ **do**
            determine/update $s_{\hat{l}}$, $targetGroup$, and Δ^{best} with Alg. 5 (A, B_{ref}, $sourceGroup$, s_l, Δ^{best});
          end
          alter $refSol$ by moving $s_{\hat{l}}$ from $sourceGroup$ to $targetGroup$;
          $F(refSol)$ += Δ^{best};
          update $tabuList$ and B_{ref};
          $sourceGroup = targetGroup$;
          if $refSol$ *is feasible and* $F(refSol) < F(bestSol)$ **then**
            $bestSol = refSol$; $F(bestSol) = F(refSol)$; update B_{best};
          end
          else
            let $sourceGroup_2$ ($targetGroup_2$) be the group of $refSol$ with $C+1$ ($C-1$) elements;
            Determine Δ^{best} and $s_{\hat{l}}$ with Alg. 6 (A, B_{ref}, $sourceGroup_2$, $targetGroup_2$, $tabuList$);
            if $F(refSol) + \Delta^{best} < F(bestSol)$ **then**
              generate $trialSol$ from $refSol$ by moving $s_{\hat{l}}$ from $sourceGroup_2$ to $targetGroup_2$;
              $bestSol = trialSol$; $F(bestSol) = F(refSol) + \Delta^{best}$; update B_{best};
            end
          end
        end
      end
    end
  end
  if $F(bestSol) < F(inSol)$ **then** $inSol = bestSol$; $F(inSol) = F(bestSol)$; $B_{in} = B_{best}$;
  $restart = true$;
  else $restart = false$;
while $restart$;

Algorithm 7: Ejection chain heuristic

Appendix B. Details on the branch and bound algorithm

In this section we present details on the branch and bound algorithm introduced in Section 3.2. In the course of the algorithm we need to store the values $\hat{l}(o_\sigma, Z^\lambda \setminus \bar{Z}^\lambda)$ and $\bar{l}(o_\sigma, \bar{Z}^\lambda)$ for all non-fathomed nodes λ of the search tree (the child nodes of which have not yet been constructed) and all $o_\sigma \in O$. The vales $\lceil |o_\sigma|/C \rceil$ are stored globally (for the whole search tree) for all $o_\sigma \in O$.

Appendix B.1. Branching

Algorithm 8 illustrates how to determine the branching variable. It makes use of matrix A as defined in Section 3.1. For all potential branching variables, the difference $\bar{lb}^{\lambda+1} - \bar{lb}^{\lambda}$

Data: matrix A, parent node λ
Result: branching variable $z_{\hat{i}\hat{j}}$, set $fixed^0$ of variables fixed to zero during branching
1 $\Delta = 0$, $\Delta^{best} = 0$, $fixed^0 = \emptyset$; $\bar{l}(o_\sigma, \bar{Z}^{\lambda+1}) = \bar{l}(o_\sigma, \bar{Z}^\lambda)$ for all $o_\sigma \in O$;
2 **for** *all* $z_{ij} \in Z \setminus Z^\lambda$ *with* $\delta^\lambda(s_i) < C$ *and* $\delta^\lambda(s_j) < C$ **do**
3 $\Delta = 0$;
4 **for** *all orders* $o_\sigma \in O$ *with* $(|o_\sigma| + \bar{l}(o_\sigma, \bar{Z}^\lambda)) \bmod C = 0$ *and with* $(a_{\sigma i} = 1 \wedge a_{\sigma j} = 0)$ **do**
5 **if** $\nexists\, s_r \in o_\sigma$ *with* $(z_{rj} \in \bar{Z}^\lambda \vee z_{jr} \in \bar{Z}^\lambda)$ **then** Δ += w_σ;
6 **end**
7 ... `// repeat Lines 4--6 analogously for` $(a_{\sigma i} = 0 \vee a_{\sigma j} = 1)$
8 **if** $\bar{lb}^\lambda + \Delta > ub - \min_{\sigma \in \{1,\dots,m\}} w_\sigma$ **then** insert z_{ij} into $fixed^0$, $Z^{\lambda+1}$, and $Z^{\lambda+2}$;
9 **else if** $\Delta > \Delta^{best}$ **then** $\Delta^{best} = \Delta$; $z_{\hat{i}\hat{j}} = z_{ij}$;
10 **end**
11 **if** $\Delta^{best} = 0$ **then** choose random element $z_{ij} \in Z \setminus Z^\lambda$ with $\delta^\lambda(s_i) < C$ and $\delta^\lambda(s_j) < C$; $z_{\hat{i}\hat{j}} = z_{ij}$;
12 **for** *all orders* $o_\sigma \in O$ **do**
13 **if** $\left(a_{\sigma\hat{i}} = 1 \wedge a_{\sigma\hat{j}} = 0\right) \vee \left(\nexists\, s_r \in o_\sigma \text{ with } (z_{r\hat{j}} \in \bar{Z}^\lambda \vee z_{\hat{j}r} \in \bar{Z}^\lambda)\right)$ **then** $\bar{l}(o_\sigma, \bar{Z}^{\lambda+1})$ += 1;
14 **if** $\left(a_{\sigma\hat{i}} = 0 \wedge a_{\sigma\hat{j}} = 1\right) \vee \left(\nexists\, s_r \in o_\sigma \text{ with } (z_{r\hat{i}} \in \bar{Z}^\lambda \vee z_{\hat{i}r} \in \bar{Z}^\lambda)\right)$ **then** $\bar{l}(o_\sigma, \bar{Z}^{\lambda+1})$ += 1;
15 **end**
16 $\delta^{\lambda+1}(s_{\hat{i}})$ += 1; $\delta^{\lambda+1}(s_{\hat{j}})$ += 1;
17 $\bar{lb}^{\lambda+1} = \bar{lb}^\lambda + \Delta^{best}$; $\bar{Z}^{\lambda+1} = \bar{Z}^\lambda \cup \{z_{\hat{i}\hat{j}}\}$; $Z^{\lambda+1} = Z^\lambda \cup \{z_{\hat{i}\hat{j}}\}$;

Algorithm 8: Determining the branching variable

is determined in Lines 2–10 of Algorithm 8. Some variables may be fixed to zero during this process (Line 8). If $\bar{lb}^{\lambda+1} - \bar{lb}^{\lambda} = 0$ no matter which branching variable is chosen, a random branching variable is selected (Line 11). Hereafter, the values $\bar{l}(o_\sigma, \bar{Z}^{\lambda+1})$ for all $o_\sigma \in O$ (Lines 12–15) as well as the values $\delta^{\lambda+1}(s_i)$ for all $s_i \in S$ are updated (Line 16).

Appendix B.2. Constraint propagation

The constraint propagation procedure for a child node λ of the search tree is presented in Algorithm 9. The triangular conditions are checked in Lines 18–35 (cases 1 and 2 of Figure 5) and 39–42 (case 3 of Figure 5) of Algorithm 9. The process of fixing additional variables is propagated until no more propagation is possible (loops 7–36, 37–43 and 6–52). For SKU partition, we additionally check whether variables have to be fixed to zero in order to not

Data: matrix A, branching variable $z_{\hat{i}\hat{j}}$, child node λ and set $fixed^0$ constructed via branching
Result: set of variables $fixed^0$ (additionally fixed to zero)

if $z_{\hat{i}\hat{j}} \in \bar{Z}^\lambda$ **then** $queue^1 = \{z_{\hat{i}\hat{j}}\}$; $queue^0 = fixed^0$;
else
  $queue^1 = \emptyset$; $fixed^0 = fixed^0 \cup \{z_{\hat{i}\hat{j}}\}$; $queue^0 = fixed^0$; $Z^\lambda = Z^{\lambda-2} \cup \{z_{\hat{i}\hat{j}}\}$; $\bar{Z}^\lambda = \bar{Z}^{\lambda-2}$;
  $\bar{lb}^\lambda = \bar{lb}^{\lambda-2}$; $\bar{l}(o_\sigma, \bar{Z}^\lambda) = \bar{l}(o_\sigma, \bar{Z}^{\lambda-2})$ for all $o_\sigma \in O$
end
do
  while $queue^1 \neq \emptyset$ **do**
    pop element z_{ij} from $queue^1$;
    if $\delta^\lambda(s_i) = C$ **then**
      for $\bar{j} = 1$ **to** $i-1$ **do**
        if $z_{\bar{j}i} \notin Z^\lambda$ **then** push (insert) $z_{\bar{j}i}$ (in)to $queue^0$ ($fixed^0$ and Z^λ);
      end
      for $\bar{j} = i+1$ **to** n **do**
        if $z_{i\bar{j}} \notin Z^\lambda$ **then** push (insert) $z_{i\bar{j}}$ (in)to $queue^0$ ($fixed^0$ and Z^λ);
      end
    end
    if $\delta^\lambda(s_j) = C$ **then** ... `// fix variables in analogy to Lines 10--15`
    for *all* $s_{\bar{j}}$, $\bar{j} < i$, *with* $(z_{\bar{j}i} \in \bar{Z}^\lambda \wedge z_{\bar{j}j} \notin Z^\lambda)$ **do**
      push (insert) $z_{\bar{j}j}$ (in)to $queue^1$;
      $\delta^\lambda(s_j)$ += 1; $\delta^\lambda(s_{\bar{j}})$ += 1;
      for *all orders* $o_\sigma \in O$ *with* $(a_{\sigma j} = 1 \wedge a_{\sigma\bar{j}} = 0)$ **do**
        if $\nexists\, s_r \in o_\sigma$ *with* $(z_{r\bar{j}} \in \bar{Z}^\lambda \vee z_{\bar{j}r} \in \bar{Z}^\lambda)$ **then**
          if $(|o_\sigma| + \bar{l}(o_\sigma, \bar{Z}^\lambda)) \bmod C = 0$ **then** $\bar{lb}^\lambda$ += w_σ;
          $\bar{l}(o_\sigma, \bar{Z}^\lambda)$ += 1;
        end
      end
      ... `// repeat Lines 21--26 analogously` $\forall\, o_\sigma \in O$ `with` $a_{\sigma j} = 0 \wedge a_{\sigma\bar{j}} = 1$
      insert $z_{\bar{j}j}$ into $\bar{Z}^\lambda$;
    end
    ... `// repeat Lines 18--29 analogously` $\forall\, s_{\bar{j}}$, $\bar{j} < i$, `with` $z_{\bar{j}j} \in \bar{Z}^\lambda \wedge z_{\bar{j}i} \notin Z^\lambda$
    ... `// repeat Lines 18--30 analogously` $\forall\, s_{\bar{j}}$, `with` $i < \bar{j} < j$ `and` $\bar{j} > j$
    for *all* $s_{\bar{j}}$, $\bar{j} < i$, *with* $((z_{\bar{j}i} \in Z^\lambda \setminus \bar{Z}^\lambda) \wedge (z_{\bar{j}j} \notin Z^\lambda)) \vee ((z_{\bar{j}j} \in Z^\lambda \setminus \bar{Z}^\lambda) \wedge (z_{\bar{j}i} \notin Z^\lambda))$ **do**
      push (insert) $z_{\bar{j}j}$ [first case] or $z_{\bar{j}i}$ [second case] (in)to $queue^0$ ($fixed^0$ and Z^λ);
    end
    ... `// repeat Lines 32--34 analogously` $\forall\, s_{\bar{j}}$, `with` $i < \bar{j} < j$ `and` $\bar{j} > j$
  end
  while $queue^0 \neq \emptyset$ **do**
    pop element z_{ij} from $queue^0$;
    for *all* $s_{\bar{j}}$, $\bar{j} < i$, *with* $((z_{\bar{j}i} \in \bar{Z}^\lambda) \wedge (z_{\bar{j}j} \notin Z^\lambda)) \vee ((z_{\bar{j}j} \in \bar{Z}^\lambda) \wedge (z_{\bar{j}i} \notin Z^\lambda))$ **do**
      push (insert) $z_{\bar{j}j}$ [first case] or $z_{\bar{j}i}$ [second case] (in)to $queue^0$ ($fixed^0$ and Z^λ);
    end
    ... `// repeat Lines 39--41 analogously` $\forall\, s_{\bar{j}}$, `with` $i < \bar{j} < j$ `and` $\bar{j} > j$
  end
  for *all* $s_i \in S$ *with* $\delta(s_i) < C$ **do**
    determine $fix = \left|\{z_{ji} | z_{ji} \in \bar{Z}^\lambda, j < i\}\right| + \left|\{z_{ij} | z_{ij} \in \bar{Z}^\lambda, j > i\}\right|$;
    determine $free = \left|\{z_{ji} | z_{ji} \notin Z^\lambda, j < i\}\right| + \left|\{z_{ij} | z_{ij} \notin Z^\lambda, j > i\}\right|$;
    if $(free > 0) \wedge (fix + free = C)$ **then**
      push all elements of $free$ to $queue^1$;
      ... `// update` δ^λ, $\bar{lb}^\lambda$, $\bar{l}(o_\sigma, \bar{Z}^\lambda)$ $\forall\, o_\sigma \in O$, $\bar{Z}^\lambda$ `as in Lines 20--28`
    end
  end
while $queue^1 \neq \emptyset$;

Algorithm 9: Constraint propagation

exceed or guarantee reaching the desired group size C (constraints (9)). This is done in Lines 9–17 and 44–51 of Algorithm 9. Note that the lower bound $\bar{lb}^{\lambda}$ along with the values $\bar{l}(o_\sigma, \bar{Z}^{\lambda})$ for all $o_\sigma \in O$ is implicitly updated when fixing variables to one (for example in Lines 21–26).

Appendix B.3. Updating lower bounds

Based on the variables that have been fixed during constraint propagation in a node λ of the search tree, the lower bound lb^{λ} needs to be updated. The corresponding procedure is presented in Algorithm 10. After initializing the values $\hat{l}(o_\sigma, Z^{\lambda} \setminus \bar{Z}^{\lambda})$ for all $o_\sigma \in O$ with the

Data: matrix A, branching variable $z_{\hat{i}\hat{j}}$, child node λ constructed via branching, variables $fixed^0$ (fixed to zero)
Result: lb^{λ}
1 **for** *all orders* $o_\sigma \in O$ **do**
2 **if** $z_{\hat{i}\hat{j}} \in \bar{Z}^{\lambda}$ **then** $\hat{l}(o_\sigma, Z^{\lambda} \setminus \bar{Z}^{\lambda}) = \hat{l}(o_\sigma, Z^{\lambda-1} \setminus \bar{Z}^{\lambda-1})$;
3 **else** $\hat{l}(o_\sigma, Z^{\lambda} \setminus \bar{Z}^{\lambda}) = \hat{l}(o_\sigma, Z^{\lambda-2} \setminus \bar{Z}^{\lambda-2})$;
4 **end**
5 **for** *all orders* $o_\sigma \in O$ **do**
6 **if** $\exists\, z_{ij} \in fixed^0$ *with* $a_{\sigma i} = a_{\sigma j} = 1$ **then**
7 **for** *all* $s_{\bar{i}} \in o_\sigma$ **do**
8 $candList = \emptyset$;
9 **for** *all* $s_{\bar{j}} \in o_\sigma$, $\bar{j} > \bar{i}$ **do**
10 **if** $z_{\bar{i}\bar{j}} \in Z^{\lambda} \setminus \bar{Z}^{\lambda}$ **then** insert $s_{\bar{j}}$ in $candList$;
11 **end**
 `// We assume` $candList$ `to be ordered by increasing SKU indices.`
12 **if** $|candList| > 1$ **then**
13 let s_r be the first element of $candList$;
14 **do**
15 **for** *all elements* s_v, $v > r$, *of* $candList$ **do**
16 **if** $z_{rv} \notin Z^{\lambda} \setminus \bar{Z}^{\lambda}$ **then** delete s_v from $candList$;
17 **end**
18 **if** s_r *is not the last element of* $candteList$ **then** increment s_r (next element of $candList$);
19 **while** s_r *is not the last element of* $candList$;
20 **end**
21 **if** $(candList \neq \emptyset) \wedge (|candList| + 1 > \hat{l}(o_\sigma, Z^{\lambda} \setminus \bar{Z}^{\lambda}))$ **then**
22 $\hat{l}(o_\sigma, Z^{\lambda} \setminus \bar{Z}^{\lambda}) = |candList| + 1$;
23 **end**
24 **end**
25 **end**
26 **end**
27 determine $\hat{lb}^{\lambda}$ with (18);
28 **if** $\hat{lb}^{\lambda} > \bar{lb}^{\lambda}$ **then** $lb^{\lambda} = \hat{lb}^{\lambda}$;
29 **else** $lb^{\lambda} = \bar{lb}^{\lambda}$;

Algorithm 10: Updating lower bound after constraint propagation

ones from the parent node, the algorithm (if necessary) updates these values by counting the maximum number of SKUs within the orders that pairwise cannot be stored within the same group due to the corresponding variables having been fixed to zero (Lines 5–26). Finally, the lower bound is determined in Lines 27–29.