Sequential Vector Packing *

Mark Cieliebak[2], Alexander Hall[1], Riko Jacob[1], and Marc Nunkesser[1]

[1] Department of Computer Science, ETH Zurich, Switzerland,
{alex.hall|riko.jacob|marc.nunkesser}@inf.ethz.ch
[2] sd&m Schweiz AG, 8050 Zurich, Switzerland,
mark.cieliebak@sdm.com

Abstract. We introduce a novel variant of the well known d-dimensional bin (or vector) packing problem. Given a sequence of non-negative d-dimensional vectors, the goal is to pack these into as few bins as possible. In the classical problem the bin size vector is given and the sequence can be partitioned arbitrarily. We study a variation where the vectors have to be packed in the order in which they arrive. The bin size vector can be chosen once in the beginning, under the constraint that the coordinate-wise bounds sum up to at most a given total bin size. This setting arises from a special resource constrained scheduling problem. We prove that the problem is $\mathcal{NP}$-hard and we propose LP based bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$- and $(1, 2)$-approximation algorithms. Furthermore, we investigate properties of natural greedy algorithms, and present an easy to implement heuristic which is fast and performs well in practice. Experiments with the heuristic and an ILP formulation yield very promising results on real world data.

Key words. Multi-dimensional bin packing, vector bin packing, resource constrained scheduling, approximation algorithms, $\mathcal{NP}$-hardness.

1 Introduction

Needless to say, many variations of bin packing have attracted a huge amount of scientific interest over the past decades, partly due to their relevance in diverse scheduling applications. The variation which we investigate arises from a specific *resource constrained scheduling* problem: A sequence of jobs is given and must be processed in this order. Each job needs certain amounts of various types of resources (for instance 1 unit of resource A, 2 units of resource B, 0 units of all others). Several jobs can be processed in one batch, if the resources they consume altogether are bounded. Specifically, for each individual type we have a reservoir containing some amount of the resource and none of these amounts may be exceeded during one batch.

Within a given bound on the total amount of available resources one has the freedom to choose how these individual amounts are distributed once and for all (e.g., 10 units in the reservoir of resource A, 23 units in the reservoir of resource B, ...). The aim is to tune these amounts in such a way that the jobs are processed in as few batches as possible.

* Work partially supported by European Commission - Fet Open project DELIS IST-001907 Dynamically Evolving Large Scale Information Systems, for which funding in Switzerland is provided by SBF grant 03.0378-1.

Our industry partner, who has provided us with the problem setting and large real world data sets, wishes to remain unnamed. We therefore cannot go into the details of the original application that motivated the definition of sequential vector packing. To nevertheless get a flavor of the type of industry problems that fall into this setting, we give here a potential alternative application:

The task is to optimize an assembly line that consists of a conveyor belt on which different work pieces arrive in the work area and a set of robot arms that can process these work pieces. The robot arms can perform different tasks depending on the resources (or tools) they load in a setup phase before each step from a toolbox. There are d different such resources. The work pieces require different processing, i.e., robot arms equipped with a specific amount of each resource (or number of each tool). The sequence $\mathbf{S}$ in which the work pieces arrive on the assembly line is fixed and cannot be altered, it can be thought of as a stack. The toolbox has a total size of B and contains an amount b_j of each resource j, $j \in \{1, \ldots, d\}$, such that $\sum_{j=1}^{d} b_j \leq B$. One production cycle consists of a first setup phase in which the robots load the necessary resources, so that in a second phase as many work pieces as possible can be popped from the stack and processed. For a set of work pieces to be processed all the necessary resources have to have been loaded in the setup phase. The optimization task is to choose the values $b_j, j \in \{1, \ldots, d\}$ such that the total number of cycles needed to process the whole sequence is minimized. A similar application is described, e.g., by Müller-Hannemann and Weihe [10].

We now give a formal definition of the problem. The jobs or work pieces correspond to a sequence of vectors and the available resources or tools to a bin vector, respectively. To the best of our knowledge this setting is novel. Due to its basic character we believe that it may be of interest also in contexts other than resource constrained scheduling.

Definition 1 (Sequential vector packing).
Given: *a sequence* $\mathbf{S} = \mathbf{s}_1 \cdots \mathbf{s}_n$ *of* demand vectors $\mathbf{s}_i = (s_{i1}, \ldots, s_{id}) \in \mathbb{Q}_+^d$, $d \in \mathbb{N}$, *and a* total bin size $B \in \mathbb{Q}_+$.
Goal: *a* bin vector *(or short:* bin*)* $\mathbf{b} = (b_1, \ldots, b_d) \in \mathbb{Q}_+^d$ *with* $\sum_{j=1}^{d} b_j = B$ *such that* $\mathbf{s}_1 \cdots \mathbf{s}_n$ *can be packed in this order into a minimum number of such bins* $\mathbf{b}$*. More precisely, the sequence can be packed into* k *bins, if* breakpoints $0 = \pi_0 < \pi_1 < \cdots < \pi_k = n$ *exist, such that (inequalities over vectors are to be read component-wise)*

$$\sum_{i=\pi_l+1}^{\pi_{l+1}} \mathbf{s}_i \leq \mathbf{b} \qquad \text{for } l \in \{0, \ldots, k-1\}.$$

We denote the j*th component,* $j \in \{1, \ldots, d\}$*, of the demand vectors and the bin vector as* resource j*, i.e.,* s_{ij} *is the demand for resource* j *of the* i*th demand vector. We also refer to* $\mathbf{s}_i$ *as* position i*.*

The *sequential unit vector packing* problem considers the restricted variant where each vector $\mathbf{s}_i$, $i \in \{1, \ldots, n\}$, contains exactly one entry equal to 1, all others are zero—i.e., each work piece needs only one tool. Note that any solution for this version can be transformed in such a way that the bin vector is integral, i.e., $\mathbf{b} \in \mathbb{N}^d$, by potentially rounding down resource amounts to the closest integer (therefore one may also

restrict the total bin size to $B \in \mathbb{N}$). The same holds, if all vectors in the sequence are integral, i.e., $\mathbf{s}_i \in \mathbb{N}^d$, $i \in \{1, \ldots, n\}$. In the following we will call an algorithm A a *bicriteria* (α, β)*-approximation algorithm* for the sequential vector packing problem if it finds for each instance $(\mathbf{S}, \beta \cdot B)$ a solution which needs no more than α times the number of bins of an optimal solution for $(\mathbf{S}, B)$. That is, the approximation algorithm may not only approximate the value of the objective function within a factor of α, but it may also relax the total bin size by a factor of β.

Related Work. There is an enormous wealth of publications both on the classical bin-packing problem and on variants of it. The two surveys by Coffman, Garey and Johnson [1, 8] give many pointers to the relevant literature until 1997. In [2] Coppersmith and Raghavan introduce the multidimensional (on-line) bin packing problem. There are also some variants that take into consideration precedence relations on the items [17, 16] that remotely resemble our setting. Still, we are unaware of any publication that deals with the sequential vector packing problem. In the context of scheduling algorithms, allowing a certain relaxation in bicriteria approximations (here increasing the bin size) is also called resource augmentation, cf. [9, 13].

New Contributions and Outline. In Section 2 we present approximation algorithms for the sequential vector packing problem. These are motivated by the strong $\mathcal{NP}$-hardness results that we give in Section 3. The approximation algorithms are based on an LP relaxation and two different rounding schemes, yielding a bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation and a $(1, 2)$-approximation. Recall that the former algorithm, e.g., for $\varepsilon = \frac{1}{3}$, yields solutions with at most 3 times the optimal number of bins while using at most 1.5 times the given total bin size B, the latter may use at most the optimal number of bins and at most twice the given total bin size B. In Section 4 we present two simple greedy strategies and argue why they perform badly in the worst case. Furthermore, we give an easy to implement heuristic and present two optimizations concerning subroutines of it. In particular, we show how one can "evaluate" a given bin vector—i.e., compute the number k of bins needed with this bin vector—in time $O(k \cdot d)$ after a preprocessing phase which takes $O(n)$ time. Finally, in Section 5 we give the results of experiments with the heuristics and an ILP formulation.

2 Approximation Algorithms

We present an ILP formulation which we subsequently relax to an LP. We continue by first describing a simple rounding scheme which yields a bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation and then show how to to obtain a $(1, 2)$-approximation.

2.1 ILP Formulation

For a sequential vector packing instance $(\mathbf{S}, B)$, let $\mathbf{w}_{u,v} := \sum_{i=u+1}^{v} \mathbf{s}_i$, for $u, v \in \{0, \ldots, n\}$ and $u < v$, denote the *total demand* (or *total demand vector*) of the subsequence $\mathbf{S}_{u,v} := \mathbf{s}_{u+1} \cdots \mathbf{s}_v$. If $\mathbf{w}_{u,v} \leq \mathbf{b}$ holds, we can pack the subsequence $\mathbf{S}_{u,v}$ into bin $\mathbf{b}$. The following integer linear programming (ILP) formulation solves the sequential vector packing problem. Let $X := \{x_i | i \in \{0, \ldots, n\}\}$ and $Y := \{y_{u,v} | u, v \in \{0, \ldots, n\}, u < v\}$ be two sets of 0-1 variables, let $\mathbf{b} \in \mathbb{Q}_+^d$.

$$
\begin{aligned}
\textbf{ILP: minimize} \quad & \sum_{i=1}^{n} x_i & & \\
\text{s.t.} \quad & x_0 & = 1 & (1)\\
& \sum_{u=0}^{i-1} y_{u,i} = \sum_{v=i+1}^{n} y_{i,v} & = x_i \text{ for } i \in \{0,\dots,n\} & (2)\\
& \sum_{\substack{u,v:\\ u<i\le v}} \mathbf{w}_{u,v} \cdot y_{u,v} & \le \mathbf{b} \text{ for } i \in \{1,\dots,n\} & (3)\\
& \sum_{j=1}^{d} b_j & = B & (4)\\
& \mathbf{b} \in \mathbb{Q}_+^d, x_i, y_{u,v} \in \{0,1\} & \text{for } x_i \in X, y_{u,v} \in Y &
\end{aligned}
$$

The 0-1 variable x_i indicates whether there is a breakpoint at position $i \geq 1$. Hence the objective: to minimize the sum over all x_i. The 0-1 variable $y_{u,v}$ can be seen as a flow which is routed on an (imagined) edge from position $u \in \{0,\dots,n-1\}$ to position $v \in \{1,\dots,n\}$, with $u < v$. The Constraints (2) ensure that flow conservation holds for the flow represented by the $y_{u,v}$ variables and that x_i is equal to the inflow (outflow) which enters (leaves) position i. Constraint (1) enforces that only one unit of flow is sent via the Y variables. The path which is taken by this unit of flow directly corresponds to a series of breakpoints. If we for instance have consecutive breakpoints at positions u and v, there will be a flow of 1 from u to v, i.e., $x_u = y_{u,v} = x_v = 1$.

In Constraints (3) the bin vector $\mathbf{b}$ comes into play: for any two consecutive breakpoints (e.g., $x_u = x_v = 1$) the constraint ensures that the bin vector is large enough for the total demand between the breakpoints (e.g., the total demand of the subsequence $\mathbf{S}_{u,v}$). Note that Constraints (3) sum over all edges that span over a position i (in a sense the cut defined by position i), enforcing that the total resource usage is bounded by $\mathbf{b}$. For the two consecutive breakpoints x_u and x_v this amounts to $\mathbf{w}_{u,v} \cdot y_{u,v} \leq \mathbf{b}$. Finally, Constraint (4) ensures the correct total size of the bin vector.

2.2 An Easy $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-Approximation

As a first step we relax the ILP formulation to an LP: here this means to have $x_i, y_{u,v} \in [0,1]$ for $x_i \in X, y_{u,v} \in Y$. We claim that Algorithm EPS ROUNDING computes a $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation:

1. Solve the LP optimally. Let $(X^\star, Y^\star, \mathbf{b}^\star)$ be the obtained fractional solution.
2. Set $(X, Y, \mathbf{b}) = (X^\star, Y^\star, \frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star)$ and $i_s = 0$. Stepwise round $(X, Y, \mathbf{b})$:
3. Let $i_e > i_s$ be the first position for which $\sum_{i=i_s+1}^{i_e} x_i \geq \varepsilon$.
4. Set $x_i = 0$ for $i \in \{i_s + 1, \dots, i_e - 1\}$, set $x_{i_e} = 1$. Reroute the flow accordingly (see also Figure 1): (a) Set $y_{i_s,i_e} = 1$. (b) Increase $y_{i_e,i}$ by $\sum_{i'=i_s}^{i_e-1} y_{i',i}$, for $i > i_e$. (c) Set $y_{i_s,i'} = 0$ and $y_{i',i} = 0$, for $i' \in \{i_s + 1, \dots, i_e - 1\}$, $i > i'$.
5. Set the new i_s to i_e and continue in Line 3, until $i_s = n$.

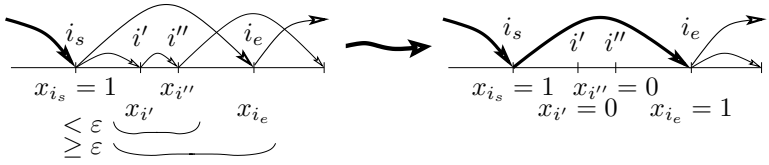


Fig. 1. An example of the rerouting of flow in lines 4 (a)-(c) of the algorithm.

Theorem 1. *The algorithm* EPS ROUNDING *is a* $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$*-approximation algorithm for the sequential vector packing problem.*

Proof. (Sketch, for full proof see appendix.) It is easy to see that the algorithm produces an integer solution and that by rerouting the flow in Line (4) the objective function is only increased by a factor of at most $\frac{1}{\varepsilon}$, due to the condition in Line (3). To see that the increase in the bin size is bounded, note that in the fractional solution before the rerouting at least $1 - \varepsilon$ flow is sent directly from i_s to i_e or beyond. After rerouting a flow of 1 is sent from i_s to i_e, thus the Constraint (3) is violated by at most $\frac{1}{1-\varepsilon}$. □

Note that one of course would not actually implement the algorithm EPS ROUNDING. Instead it suffices to compute the bin vector $\mathbf{b}$ with the LP and then multiply it by $\frac{1}{1-\varepsilon}$ and evaluate the obtained bin vector, e.g., with the algorithm given in Section 4.

2.3 A $(1, 2)$-Approximation

We start by proving some properties of the LP relaxation and then describe how they can be applied to obtain the rounding scheme yielding the desired bicriteria ratio.

Properties of the Relaxation Let $(X, Y, \mathbf{b})$ be a fractional LP solution; recall that the Y variables represent a flow. Let $e_1 = (u, v)$ and $e_2 = (u', v')$ denote two flow carrying edges, i.e., $y_{u,v} > 0$ and $y_{u',v'} > 0$. We say that e_1 *is contained in* e_2 if $u' < u$ and $v' > v$, we also call (e_1, e_2) an *embracing pair*. We say an embracing pair (e_1, e_2) is *smaller* than an embracing pair $(\hat{e}_1, \hat{e}_2)$, if the length of e_1 (for $e_1 = (u, v)$, its length is $v - u$) is less than the length of $\hat{e}_1$. Further, let $y_{\min} \in \mathbb{Q}_+$ denote the minimum amount of flow on any flow carrying edge. We show that the following structural property holds:

Lemma 1 (no embracing pairs). *Any optimal fractional LP solution* $(X^\star, Y^\star, \mathbf{b}^\star)$ *can be modified in such a way that it contains no embracing pairs, without increasing the objective function and without modifying the bin vector.*

Proof. We set $Y = Y^\star$ and show how to stepwise treat embracing pairs contained in Y, proving after each step that $(X^\star, Y, \mathbf{b}^\star)$ is still a feasible LP solution. We furthermore show that this procedure terminates and in the end no embracing pairs are left in Y.

Let us begin by describing one iteration step, assuming $(X^\star, Y, \mathbf{b}^\star)$ to be a feasible LP solution which still contains embracing pairs. Let (e_1, e_2), with $e_1 = (u, v)$ and $e_2 = (u', v')$, be the smallest embracing pair—as defined above. If there are several smallest embracing pairs, choose one of these arbitrarily. We now modify the flow Y

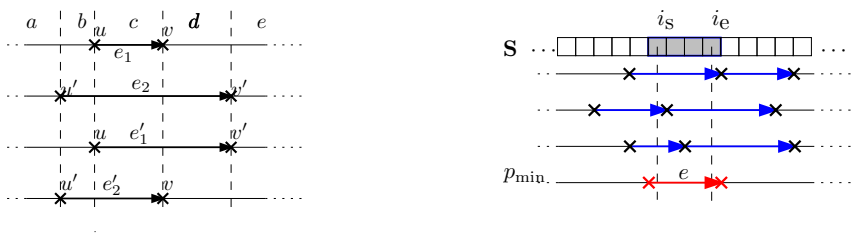


Fig. 2. Left: Replacement of λ units of flow on e_1 and e_2 by λ units of flow on e'_1 and e'_2 in Lemma 1. Right: Extracting the integral solution. Edge e together with other potential edges in $Y^\star$ in Theorem 2.

to obtain a new flow Y' by rerouting $\lambda = \min\{y_{u,v}, y_{u',v'}\}$ units of flow from e_1, e_2 onto the edges $e'_1 = (u, v')$ and $e'_2 = (u', v)$: $y'_{u,v} = y_{u,v} - \lambda$, $y'_{u',v'} = y_{u',v'} - \lambda$ and $y'_{u',v} = y_{u',v} + \lambda$, $y'_{u,v'} = y_{u,v'} + \lambda$; see also left picture in Figure 2. The remaining flow values in Y' are taken directly from Y. It is easy to see that the flow conservation constraints (2) still hold for the values $X^\star, Y'$ (consider a circular flow of λ units sent in the residual network of Y on the cycle u', v, u, v', u'). Since $X^\star$ is unchanged this also implies that the objective function value did not change, as desired. It remains to prove that the Constraints (3) still hold for the values $Y', \mathbf{b}^\star$ and that the iteration terminates.

Feasibility of the Modified Solution. Constraints (3) are parameterized over $i \in \{1, \ldots, n\}$. We argue that they are not violated separately for $i \in \{u' + 1, \ldots, u\}$, $i \in \{u + 1, \ldots, v\}$, and $i \in \{v + 1, \ldots, v'\}$, i.e., the regions b, c, and d in Figure 2 (left). For regions a and e (the rest) it is easy to check that the values of the affected variables do not change when replacing Y by Y'. So let us consider the three regions:

Region b (d) The only variables in (3) which change when replacing Y by Y' for this region are: $y'_{u',v'} = y_{u',v'} - \lambda$ and $y'_{u',v} = y_{u',v} + \lambda$. This means that flow is moved to a shorter edge, which can only increase the slack of the Constraints: With $\mathbf{w}_{u',v} < \mathbf{w}_{u',v'}$ it is easy to see that (3) still holds in region b. Region d is analogous to b.

Region c Here the only variables which change in (3) are: $y'_{u,v} = y_{u,v} - \lambda$, $y'_{u',v'} = y_{u',v'} - \lambda$, $y'_{u',v} = y_{u',v} + \lambda$, and $y'_{u,v'} = y_{u,v'} + \lambda$. In other words λ units of flow were moved from e_1 to e'_1 and from e_2 to e'_2. Let us consider the fraction of demand which is contributed to (3) by these units of flow before and after the modification. Before (on e_1 and e_2) this was $\lambda \cdot (\mathbf{w}_{u,v} + \mathbf{w}_{u',v'})$ and afterwards (on e'_1 and e'_2) it is $\lambda \cdot (\mathbf{w}_{u',v} + \mathbf{w}_{u,v'})$. Since both quantities are equal, the left hand side of (3) remains unchanged in region c.

Termination of the Iteration. First we show that the modification did not introduce an embracing pair which is strictly smaller than (e_1, e_2). We assume the contrary and say that w.l.o.g. the flow added to edge e'_1 created a new embracing pair (e, e'_1) which is smaller than the (removed) embracing pair (e_1, e_2). Clearly e is also contained in e_2. Therefore, before the modification (e, e_2) would have been an embracing pair as well. Since (e, e_2) is smaller than (e_1, e_2) it would have been chosen instead, which gives our contradiction. It follows that in the course of the iteration the length of the e_1-type edges is non-decreasing.

Now, observe that in each iteration the flow on an e_1-type edge is reduced by $\lambda \geq y_{\min} \geq y^\star_{\min}$ units (notice that the modification does not decrease the minimum amount of flow on flow carrying edges). At the same time only flow amounts on edges that are strictly longer than the e_1-type edge are increased. Since $y^\star_{\min} \in \mathbb{Q}_+$ this clearly cannot be repeated for an infinite number of steps: after a finite number of steps the length of the considered e_1-type edges must increase, at the latest when, after some finite number of steps, the flow on all edges of this length is reduced to zero. Such an increase in length can obviously only happen at most n times. □

Path Flow Algorithm We will use the structural insights of the last section to prove that bin vector $2 \cdot \mathbf{b}^\star$ yields a $(1, 2)$-approximation to the optimal solution.

For an optimal fractional LP solution $(X^\star, Y^\star, \mathbf{b}^\star)$ without embracing pairs—due to Lemma 1 such a solution exists—let $p_{\min}$ denote the shortest flow carrying path, where shortest is meant with respect to the number of breakpoints. Clearly, the length of $p_{\min}$ is at most the objective function value $\sum_{i=1}^{n} x^\star_i$. Below we will show that the integral solution corresponding to $p_{\min}$ is feasible for the bin vector $2 \cdot \mathbf{b}^\star$ and thus $p_{\min}$ and $2 \cdot \mathbf{b}^\star$ are our $(1, 2)$-approximation. Observe that the approximation algorithm does not actually need to transform an optimal LP solution given, e.g., by an LP solver into a solution without embracing pairs. The existence of path $p_{\min}$ in such a transformed solution is merely taken as a proof that the bin vector $2 \cdot \mathbf{b}^\star$ yields less than $\sum_{i=1}^{n} x^\star_i$ breakpoints. To obtain such a path, we simply evaluate $2 \cdot \mathbf{b}^\star$ with the algorithm presented in Section 4 ($\mathbf{b}^\star$ given by the LP solver).

Theorem 2. *Given an optimal fractional LP solution $(X^\star, Y^\star, \mathbf{b}^\star)$ without embracing pairs, let $p_{\min}$ denote the shortest flow carrying path. The integral solution corresponding to $p_{\min}$ is feasible for $2 \cdot \mathbf{b}^\star$.*

Proof. We only have to argue for the feasibility of the solution w.r.t. the doubled bin vector. Again we will consider Constraints (3). Figure 2 (right) depicts an edge e on path $p_{\min}$ and other flow carrying edges. We look at the start and end position i_s and i_e in the subsequence defined by e. Denote by $E_{i_s} = \{(u, v) | 0 \leq u < i_s \leq v \leq n\}$ (and E_{i_e}, respectively) the set of all flow carrying edges that cross i_s (i_e) and by $i_{\min}, (i_{\max})$ the earliest tail (latest head) of an arc in $E_{i_s}, (E_{i_e})$. Furthermore, let $E' = E_{i_s} \cup E_{i_e}$. Summing up the two Constraints (3) for i_s and i_e gives $2\mathbf{b}^\star \geq \sum_{(u,v)\in E_{i_s}} y^\star_{u,v} \cdot \mathbf{w}_{u,v} + \sum_{(u,v)\in E_{i_e}} y^\star_{u,v} \cdot \mathbf{w}_{u,v} = A$ and thus

$$2\mathbf{b}^\star \geq A \geq \sum_{i_{\min} < i \leq i_{\max}} \sum_{\substack{(u,v)\in E': \\ u<i\leq v}} y^\star_{u,v} \cdot \mathbf{s}_i \tag{5}$$

$$\geq \sum_{i_s < i \leq i_e} \sum_{\substack{(u,v)\in E': \\ u<i\leq v}} y^\star_{u,v} \cdot \mathbf{s}_i = \sum_{i_s < i \leq i_e} \mathbf{s}_i = \mathbf{w}_{i_s,i_e} \ . \tag{6}$$

The second inequality in (5) is in general an inequality because the sets E_{i_s} and E_{i_e} need not be disjoint. For the first equality in (6) we rely on the fact that there are no embracing pairs. For this reason each position between i_s and i_e is covered by an edge that covers either i_s or i_e. We have shown that the demand between any two breakpoints on $p_{\min}$ can be satisfied by the bin vector $2 \cdot \mathbf{b}^\star$. □

Observe that for integral resources the above proof implies that even $\lfloor 2\mathbf{b}^\star \rfloor$ has no more breakpoints than the optimal solution. Note also that it is easy to adapt both approximation algorithms so that they can handle pre-specified breakpoints. The corresponding x_i values can simply be set to one in the ILP and LP formulations.

3 Complexity Considerations

In this section, we study the computational complexity of the sequential vector packing problem. First, we show that finding an optimal solution is $\mathcal{NP}$-hard, and then we consider special cases of the problem that allow a polynomial time algorithm or that are fixed parameter tractable (FPT). Our $\mathcal{NP}$-hardness proofs also identify parameters that cannot lead to an FPT-algorithm.

Hardness of Minimizing the Number of Breakpoints (Bins) For all considered problem variants it is easy to determine the objective value once a bin vector is chosen. Hence, for all variants of the sequential vector packing problem considered in this article, the corresponding decision problem is in $\mathcal{NP}$.

To simplify the exposition, we first consider a variant of the sequential unit vector packing problem, where the sequence of vectors has prespecified breakpoints, always after w positions. Then the sequence effectively decomposes into a set of *windows* of length w, and for each position in such a window i it is sufficient to specify the resource that is used at position $j \in \{1, \ldots, w\}$, denoted as $s_j^i \in \{1, \ldots, d\}$. This situation can be understood as a set of sequential unit vector packing problems that have to be solved with the same bin vector. The objective is to minimize the total number of (additional) breakpoints, i.e., the sum of the objective functions of the individual problems. Then, we also show strong $\mathcal{NP}$-hardness for the original problem. The proofs are deferred to the appendix due to space constraints.

Lemma 2. *Finding the optimal solution for sequential unit vector packing with windows of length 4 (dimension d and bin size B as part of the input) is $\mathcal{NP}$-hard.*

Theorem 3. *The sequential unit vector packing problem is strongly $\mathcal{NP}$-hard.*

Polynomially Solvable Cases and FPT Here, we consider the influence of parameters on the complexity of the problem. We say that an algorithm is fixed parameter tractable (FPT) with respect to parameter k if its running time is bounded by $f(k) \cdot n^{O(1)}$ for an arbitrary function f (usually something like 2^k or 2^{2^k}). See for example [11] or [4] for a thorough introduction to the concept.

If the windows in the vector packing problem are limited to length three, the problem can be solved in polynomial time: There is no interaction between the resources, thus, it is impossible that avoiding a breakpoint induced by one resource depends upon the multiple availability of another resource. Hence, a natural greedy algorithm that always takes the resource that currently causes most breakpoints is optimal. Additionally, Lemma 2 shows that the problem is $\mathcal{NP}$-hard even if all windows have length 4. Hence, the parameter window size does not allow an FPT-algorithm if $\mathcal{P} \neq \mathcal{NP}$.

On the other hand, the parameter B allows (partly because $d \leq B$) to enumerate and evaluate (Sections A.2 and A.3) the number of breakpoints in time $f(B) \cdot n^{O(1)}$, i.e., this is an FPT-algorithm. A constant upper limit on the number of breakpoints allows to enumerate all positions of breakpoints and to determine the necessary bin vector in polynomial time. Note that this is not an FPT algorithm.

4 Practical Algorithms

Greedy Algorithms We describe two natural greedy heuristics for sequential unit vector packing. Given an input $(\mathbf{S}, B)$ we denote by $k(\mathbf{b})$ the minimal number of breakpoints needed for a fixed bin vector $\mathbf{b}$. Observe that it is relatively easy to calculate $k(\mathbf{b})$ in linear time (see end of this section and Section A.3). The two greedy algorithms we discuss here are: `Greedy-Grow` and `Greedy-Shrink`. `Greedy-Grow` grows the bin vector starting with the all one vector. In each step it increases some resource by an amount of 1 until the total bin size B is reached, greedily choosing the resource whose increment improves $k(\mathbf{b})$ the most. `Greedy-Shrink` shrinks the bin vector starting with a bin vector $\mathbf{b} = \sum_{i=1}^{n} \mathbf{s}_i$, which yields $k(\mathbf{b}) = 0$ but ignores the bin size B. In each step it then decreases some resource by an amount of 1 until the total bin size B is reached, greedily choosing the resource whose decrement worsens $k(\mathbf{b})$ the least.

Also in the light of the following observations (for proofs see the appendix) it is important to specify the tie-breaking rule for the case that there is no improvement at all after the addition of a resource. `Greedy-Grow` can be forced to produce a solution only by this tie breaking rule, which is an indicator for its bad performance. Also `Greedy-Shrink` can produce bad solutions depending on the tie breaking scheme as the following observation shows.

Observation 1 *Given any instance $(\mathbf{S}, B)$, this instance can be modified to an instance $(\mathbf{S}', B')$, with $n' = n, d' = 2d, B' = 2B$ such that* `Greedy-Grow` *needs to apply its tie breaking rule in every second step.*

Observation 2 *There are instances with d resources on which the solution produced by* `Greedy-Shrink` *is a factor of $\lfloor d/2 \rfloor$ worse than the optimal solution, if the tie breaking is done deterministically.*

For the experiments we use a *round-robin* tie breaking rule that cycles through the resources , i.e., every time a tie occurs it chooses the (cyclic) successor of the resource that was increased in the last tie.

Enumeration Heuristic We present an enumeration heuristic for integral demand vectors $\mathbf{s}_i \in \mathbb{N}^d$, $i \in \{1, \ldots, n\}$, that is inspired by a variant of Schöning's 3-SAT algorithm [15] that searches the complete hamming balls of radius $\lfloor n/4 \rfloor$ around randomly chosen assignments, see [3]. The following algorithm uses a similar combination of randomized guessing and complete enumeration of parts of the solution space that are exponentially smaller than the whole solution space. The idea is to guess uniformly at random (u.a.r.) subsequences $\mathbf{S}_{i_1,i_2}$ of the sequence that do not incur a breakpoint in

a fixed optimal solution $\mathbf{b}_{\text{opt}}$. For such a subsequence we know that $\mathbf{b}_{\text{opt}} \geq \mathbf{w}_{i_1,i_2}$. In particular, if we know a whole set W of such total demand vectors that all come from subsequences without breakpoints for $\mathbf{b}_{\text{opt}}$, we know that $\mathbf{b}_{\text{opt}} \geq \max_{\mathbf{w}\in W} \mathbf{w}$ must hold for a component-wise maximum. This idea leads to the randomized heuristic enumeration (RHE) algorithm:

Phase 1: Start with a "lower bound vector" $\mathbf{t} = 0$. For a given subsequence length ssl and a number p of repetitions, in each of p rounds choose $\underline{\sigma}_i =$u.a.r $\{0, \ldots, n - \text{ssl}\}$, set $\overline{\sigma}_i = \underline{\sigma}_i + \text{ssl}$, and then set $\mathbf{t} = \max\{\mathbf{t}, \mathbf{w}_{\underline{\sigma}_i,\overline{\sigma}_i}\}$.

Phase 2: Find a bin vector $\mathbf{b}$ of total size B with $\mathbf{b} \geq \mathbf{t}$ that minimizes $k(\mathbf{b})$. Do this by enumerating all $\mathbf{b} \geq \mathbf{t}$ of total size B.

As easy as the enumeration in Phase 2 seems, this should be done efficiently. See Section A.2 in the appendix for an algorithm. It is straight-forward to analyze the success probability of this algorithm if we relate the subsequence length to an estimate k' of the minimum number of breakpoints k, see Lemma 4 in the appendix.

Observe that the first subsequence that is guessed increases the lower bound vector by its full length. Subsequent guesses can, but need not, improve the lower bounds. The growth of the lower bound depends on the distribution of demand vectors in the fixed input sequence and is therefore difficult to analyze for arbitrary such sequences. On the other hand analyzing the growth of the lower bound seems possible for random input sequences, but we doubt that this would give any meaningful insights. For this reason we only give experimental evidence that this algorithm performs well, see Section 5.

Evaluation For demand vectors $\mathbf{s}_i \in \mathbb{Q}_+^d$, $i \in \{1, \ldots, n\}$, the evaluation of a given bin vector $\mathbf{b}$, i.e., computing $k(\mathbf{b})$, can be done in the obvious way in $O(n \cdot d)$ time. With a preprocessing phase and some algorithmic engineering we can show the following theorem (see Section A.3 for a complete discussion).

Theorem 4. *Given a sequence* $\mathbf{S}$ *we can construct a data structure in* $O(n)$ *preprocessing time with* $O(n \cdot d \cdot B)$ *resp.* $O(n)$ *space such that an evaluation query for sequential vector packing resp. sequential unit vector packing for a bin vector* $\mathbf{b}$ *takes* $O(kd)$ *time, where* k *denotes the number of breakpoints for* $\mathbf{b}$*.*

5 Experiments

In this section we report on some experiments on real world data. This data is electronically available at `www.inf.ethz.ch/personal/mnunkess/SVP/`. We implemented the greedy algorithms, the enumeration heuristic and the integer linear program. For the latter we use CPLEX 9.0. All programs were run on a 3GHz P4 workstation with 3GB RAM, running Linux 2.4.22. All figures can be found in the appendix.

Mixed Integer Program Ideally, we want to describe the quality of the heuristic optimization algorithms by comparison to the optimal value. One attempt to compute such optimal values is to use the ILP-formulation of Section 2.1 and solve it with an ILP-solver. In our setting, this worked only for the small real world instances, and not for

the mediums sized ones. Still, those could be solved with an non-straightforward reformulation of the ILP as a mixed integer linear program as described in Appendix A.4.

Setup and Computational Results We mainly compare solution qualities, because the running times of the different approaches are orders of magnitude apart. On many of our instances a calculation for a fixed B takes some seconds for the greedy algorithms and some hours for the mixed integer linear program. We let the enumeration heuristic run for 10 minutes which seems like a realistic maximum time that an "online" user would be willing to wait for a result. We then fix the number of repetitions of the guessing phase of `RHE` to be as many as it takes to let $\|\mathbf{t}\|_1$, the sum of the guessed lower bounds, exceed some fraction of B. This fraction is initially chosen as 99% and adaptively decreased after each run as long as the targeted time of 10 minutes is not exceeded. The subsequence length is initially fixed with respect to the estimated number of breakpoints that we get by running both greedy approaches. We set it to one half times the average distance between two breakpoints and increase it adaptively if the lower bound does not grow any more after a fixed number of repetitions in the initialization phase. By the adaptive choice of both the subsequence length and the fraction the algorithm is robust with respect to changing values of B, d and the time that it is run.

In Figure 4 we show the relative performances on our biggest real world instance (Inst1), cf. Table 1 in the appendix that summarizes information on our instances. The different data points correspond to the algorithms `greedy-grow`, `greedy-shrink`, `RHE` and the linear relaxations of the two different ILP formulations. The values represent the ratio of the solution found by the algorithm to the optimal integral solution that we calculated using the mixed integer programming formulation. The figure shows that for small values of B `greedy-grow` produces results that are close to optimal, whereas for bigger values the quality gets worse. An explanation for this behavior is that `greedy-grow` builds a solution iteratively. As the results of Section 4 show, the greedy algorithms can be forced to take decisions based only on the tie-breaking rule. On this instance tie-breaking is used at several values of B, which leads to an accumulation of errors in addition to the inherent heuristic nature of the method. Note that by definition `greedy-shrink` is optimal for $B = \|\sum_{i=1}^{n} \mathbf{s}_i\|_1$, which is 196 on this instance. In order to have a meaningful scale we let the x-axis stop before that value.

In Figure 5(a) we present the quality of the solutions delivered by `RHE` relative to the optimal solution on four further instances. Note that for different instances different values of B make sense. Instance Inst1-doubled is obtained from Inst1, by the doubling transformation used in Observation 1 (in the appendix). In Figure 5(b) we compare the best of the two greedy results to the result of `RHE` (Note that even if we use the greedy algorithms to determine the parameter settings of RHE, these results are not visible to RHE). Instance rand-doubled is an instance where first the demand unit vectors are drawn uniformly at random and then a doubling transformation is performed to make it potentially more complicated. It could not be solved to optimality with our MIP approach and does therefore not occur in Figure 5(a). Compared to the other instances the greedy algorithms do not perform too badly. One reason for this is that we chose a uniform distribution for the resources. Therefore the tie-breaking rules do the right choices "on average". On the other hand on the doubled real-world instance Inst1-doubled `RHE`

gives superior results and in particular for higher values of B the greedy algorithms perform comparatively poorly.

6 Conclusion

In this paper, we have introduced the sequential vector packing problem, presented $\mathcal{NP}$-hardness proofs for different variations, two approximation algorithms, several heuristics, and an experimental evaluation. From our point of view the most interesting open challenges would be to search for approximation algorithms which do not relax the bin size on the one hand and for inapproximability results on the other hand.

References

[1] E. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Algorithm Design for Computer System Design*, chapter Approximation Algorithms for Bin Packing: An updated Survey, pages 49–106. Springer, 1984.

[2] D. Coppersmith and P. Raghavan. Multidimensional on-line bin packing: Algorithms and worst-case analysis. *Operations Research Letters*, 4:48–57, 1989.

[3] E. Dantsin, A. Goerdt, E. A. Hirsch, and U. Schöning. Deterministic algorithms for k-sat based on covering codes and local search. In *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 236–247. LNCS, 2000.

[4] R. G. Downey and M. R. Fellows. *Parametrized Complexity*. Monographs in Computer Science. Springer, 1999.

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM Press.

[6] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[8] D. S. Hochbaum, editor. *Approximation Algorithms*, chapter Approximation Algorithms For Bin Packing: A Survey, pages 46–93. PWS Publishing Company, 1997.

[9] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:617–643, 2000.

[10] M. Müller-Hannemann and K. Weihe. Moving policies in cyclic assembly–line scheduling. *Theoretical Computer Science*, to appear, 2005.

[11] R. Niedermeier. Invitation to fixed-parameter algorithms. Universität Tübingen, 2002. Habilitation Thesis.

[12] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms*. Academic Press, 2nd edition, 1978.

[13] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 140–149, 1997.

[14] F. Ruskey. Combinatorial generation. 2005.

[15] U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32:615–623, 2002.

[16] T. S. Wee and M. J. Magazine. Assembly line balancing as generalized bin-packing. *Operations Research Letters*, 1:56–58, 1982.

[17] J. Yang and J. Y.-T. Leung. The ordered open-end bin-packing problem. *Operations Research*, 51(5):759–770, 2003.

A Appendix

A.1 Omitted Proofs

Proof of Theorem 1. We show the desired result in three separate steps.

Integer Rounding. The following invariant is easy to see by considering Figure 1: at the beginning of each iteration step (i.e., at Line 3) the current, partially rounded solution $(X, Y, \mathbf{b})$ corresponds to a valid flow, which is integral until position i_s. From this invariant it follows that $(X, Y, \mathbf{b})$ in the end corresponds to a valid integer flow.

At Most $\frac{1}{\varepsilon}$-times the Number of Breakpoints. In line 4, x_i values which sum up to at least ε (see Line 3) are replaced by $x_{i_e} = 1$. Therefore, our rounding increases the total value of the objective function by at most a factor of $\frac{1}{\varepsilon}$.

At Most $\frac{1}{1-\varepsilon}$-times the Total Bin Size. Again consider one step of the iteration. We need to check that by rerouting the flow to go directly from i_s to i_e we do not exceed the LP bin capacity by more than $\frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star$. First let us consider the increase of $y_{i_e,i}$, for $i > i_e$, in Line 4 (b). Since the total increase is given directly by some $y_{i',i}$ which are set to 0 (Line 4 (c)), the Constraint (3) still holds after the change. In other words, flow on edges is rerouted here onto shorter (completely contained) edges; this does not change the feasibility of Constraint (3).

Now we consider the increase of y_{i_s,i_e} to 1. We will show that the total demand w_{i_s,i_e} between the new breakpoints i_s and i_e is bounded by $\frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star$. With $x_{i_s} = 1$ and since $\sum_{i=i_s+1}^{i_e-1} x_i < \varepsilon$ (see Line 3), we know that $\sum_{i=i_e}^{n} y_{i_s,i} = x_{i_s} - \sum_{i=i_s+1}^{i_e-1} y_{i_s,i} \geq 1 - \sum_{i=i_s+1}^{i_e-1} x_i > 1 - \varepsilon$; note that the first equality holds by the flow conservation constraint (2). By Constraint (3) we obtain $w_{i_s,i_e} \cdot \sum_{i=i_e}^{n} y_{i_s,i} \leq \sum_{i=i_e}^{n} w_{i_s,i} \cdot y_{i_s,i} \leq \sum_{u,v:u<i_e\leq v} w_{u,v} \cdot y_{u,v} \leq \mathbf{b}$. Thus plugging these two inequalities together, we know for the total demand $w_{i_s,i_e} < \frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star = \mathbf{b}$. Since this holds for all iteration steps and thus for all consecutive breakpoints of the final solution, it is clear that multiplying the bin vector of the LP solution by a factor $\frac{1}{1-\varepsilon}$ yields a valid solution for the ILP. □

Proof of Lemma 2. By reduction from the $\mathcal{NP}$-complete problem Clique [7] or more generally from k-densest subgraph [6]. Let $G = (V, E)$ be an instance of k-densest subgraph, i.e., an undirected graph without isolated vertices in which we search for a subset of vertices of cardinality k that induces a subgraph with the maximal number of edges.

We construct a sequential unit vector packing instance $(\mathbf{S}, B)$ with windows of length 4 and with $d = |V|$ resources. Assume as a naming convention $V = \{1, .., d\}$. There is precisely one window per edge $e = (u, v) \in E$, the sequence of this window is $s^e = uvuv$. The total bin size is set to $B = d+k$. This transformation can be carried out in polynomial time and achieves, as shown in the following, that $(\mathbf{S}, B)$ can be solved with at most $|E| - \ell$ (additional) breakpoints if and only if G has a subgraph with k vertices containing at least ℓ edges.

Because every window contains at most two vectors of the same resource, having more than two units of one resource does not influence the number of breakpoints. Every resource has to be assigned at least one unit because there are no isolated vertices in G.

Hence, a solution to $(\mathbf{S}, B)$ is characterized by the subset R of resources to which two units are assigned (instead of one). By the choice of the total bin size we have $|R| = k$. A window does not induce a breakpoint if and only if both its resources are in R, otherwise it induces one breakpoint.

If G has a node induced subgraph G' of size k containing ℓ edges, we chose R to contain the vertices of G'. Then, every window corresponding to an edge of G' has no breakpoint, whereas all other windows have one. Hence, the number of (additional) breakpoints is $|E| - \ell$.

If $(\mathbf{S}, B)$ can be scheduled with at most $|E| - \ell$ breakpoints, define R as the resources for which there is more than one unit in the bin vector. Now $|R| \leq k$, and we can assume $|R| = k$ since the number of breakpoints only decreases if we change some resource from one to two, or decrease the number of resources to two. Now, R defines a subgraph G' with k vertices of G. The number of edges is at least ℓ because only windows with both resources in R do not have a breakpoint. □

Now, it remains to consider the original problem without pre-specified breakpoints.

Lemma 3. *Let $(\mathbf{S}, B)$ be an instance of sequential (unit) vector packing of length n with k pre-specified breakpoints and d resources ($d \leq B$) where every resource is used at least once. Then one can construct in polynomial time an instance $(\mathbf{S}', B')$ of the (unit) vector packing problem with bin size $B' = 3B + 2$ and $d' = d + 2B + 2$ resources that can be solved with at most $\ell + k$ breakpoints if and only if $(\mathbf{S}, B)$ can be solved with at most ℓ breakpoints.*

Proof. The general idea is to use for every prespecified breakpoint some "stopping" sequence F_i with the additional resources in a way that the bound B' guarantees that there is precisely one breakpoint in F_i. This sequence F_i needs to enforce exactly one breakpoint, no matter whether or not there was a breakpoint within the previous window (i.e., between F_{i-1} and F_i). If we would use the same sequence for F_{i-1} and F_i, a breakpoint within the window would yield a "fresh" bin vector for F_i. Therefore, the number of breakpoints in F_i could vary depending on the demands in the window (and whether or not they incur a breakpoint).

To avoid this, we introduce two different stopping sequences F and G which we use alternatingly. This way we are sure that between two occurrences of F there is at least one breakpoint.

The resources $1, \ldots, d$ of $(\mathbf{S}', B')$ are one-to-one the resources of $(\mathbf{S}, B)$. The $2B+2$ additional resources are divided into two groups $f_1, \ldots, f_{B+1}$ for F and $g_1, \ldots, g_{B+1}$ for G. The first pre-specified breakpoint in $\mathbf{S}$, the third and every other odd breakpoint is replaced by the sequence $F := f_1 f_2 \cdots f_{B+1} f_1 f_2 \cdots f_{B+1}$, the second and all even breakpoints by the sequence $G := g_1 g_2 \cdots g_{B+1} g_1 g_2 \cdots g_{B+1}$.

To see the backward direction of the statement in the lemma, a bin $\mathbf{b}$ for $(\mathbf{S}, B)$ resulting in ℓ breakpoints can be augmented to a bin vector $\mathbf{b}'$ for $(\mathbf{S}', B')$ by adding one unit for each of the new resources. This does not exceed the bound B'. Now, in $(\mathbf{S}', B')$ there will be the original breakpoints and a breakpoint in the middle of each inserted sequence. This shows that $\mathbf{b}'$ results in $\ell + k$ breakpoints for $(\mathbf{S}', B')$, as claimed.

To consider the forward direction, let $\mathbf{b}'$ be a solution to $(\mathbf{S}', B')$. Because every resource must be available at least once, and $B' - d' = 3B + 2 - (d + 2B + 2) = B - d$,

at most $B - d < B$ entries of $\mathbf{b}'$ can be more than one. Therefore, at least one of the resources f_i is available only once, and at least one of the resources g_j is available only once. Hence, there must be at least one breakpoint within each of the k inserted stopping sequences. Let $k + \ell$ be the number of breakpoints induced by $\mathbf{b}'$ and $\mathbf{b}$ the projection of $\mathbf{b}'$ to the original resources. Since all resources must have at least one unit and by choice of B' and d' we know that $\mathbf{b}$ sums to less than B.

Now, if a subsequence of $\mathbf{S}$ not containing any f or g resources can be packed with the resources $\mathbf{b}'$, this subsequence can also be packed with $\mathbf{b}$. Hence, $\mathbf{b}$ does not induce more than ℓ breakpoints in the instance $(\mathbf{S}, B)$ with pre-specified breakpoints. □

Proof of Theorem 3. By Lemma 2 and Lemma 3, with the additional observation that all used numbers are polynomial in the size of the original graph. □

Proof of Observation 1. The idea is to split each resource r into two resources r_1, r_2 and to replace each occurrence of r in a demand vector $\mathbf{s}$ by a demand for r_1 and r_2. We call this transformation *doubling* and will come back to it in the experimental section. Then, considering `Greedy-Grow`'s approach to reduce the number of breakpoints, increasing r_1 or r_2 alone is not enough. Only if r_1 and r_2 are both increased, the number of breakpoints may decrease. That is, for all resources the number of saved breakpoints in the beginning is zero and greedy is forced to take an arbitrary resource in Step 1 and then the partner of this resource in Step 2. Then `Greedy-Grow` again chooses an arbitrary resource in Step 3 and its partner in Step 4, and so on. With this scheme it is obvious that `Greedy-Grow` can be fooled to produce arbitrary solutions. □

Proof of Observation 2. Let $k = \lfloor d/2 \rfloor$, consider the following unit vector instance with $2k$ resources and $B = 3k$: "$1 \cdots k\ 1 \cdots k(k+1)(k+1)(k+2)(k+2) \cdots (2k)(2k)$". At the beginning of the algorithm $\mathbf{b}$ is set to $(2, \ldots, 2)$. In the first step the removal of each of the resources incurs one breakpoint. Therefore, `Greedy-Shrink` deletes an arbitrary resource depending on the tie-breaking scheme. As tie breaking is deterministic, we let this resource be one of the last k ones. After this deletion the situation remains unchanged except for the fact that the chosen resource must not be decreased any more. It follows that in k steps `Greedy-Shrink` sets the last k resources to one, which incurs a total cost of k, whereas the optimal solution sets the first k resources to one, which incurs a cost of 1. Thus, the ratio of greedy versus optimal solution is $k = \lfloor d/2 \rfloor$. □

Lemma 4. *Let $\mathbf{b}_{opt}$ be an optimal bin vector for an integral instance $(\mathbf{S}, B)$, choose ssl as $\lfloor \frac{n}{4k'} \rfloor$ and assume $k' \geq k/2$, where k is the minimal number of breakpoints. Then for each of the demand vectors $\mathbf{w}_{\underline{\sigma}_i, \overline{\sigma}_i}, i \in \{1, \ldots, p\}$ it holds that $\Pr[\mathbf{w}_{\underline{\sigma}_i, \overline{\sigma}_i} \leq \mathbf{b}_{opt}] \geq \frac{1}{2}$.*

Proof. A sufficient (but not necessary) condition for $\mathbf{w}_{\underline{\sigma}_i, \overline{\sigma}_i} \leq \mathbf{b}_{\text{opt}}$ is that the optimal solution $\mathbf{b}_{\text{opt}}$ has no breakpoint in the subsequence $\mathbf{S}_{\underline{\sigma}_i, \overline{\sigma}_i}$. As there are at most $2k'$ such breakpoints the probability that we hit one with a random interval of length $\lfloor \frac{n}{4k'} \rfloor$ is bounded by $\frac{1}{2}$. □

A.2 Enumeration

As easy as the enumeration in the second phase of our RHE-algorithm looks, this should be done efficiently. So let us have a look at the problem at hand: We want to enumerate all possible $b_1, \ldots, b_d$ with sum B and individual lower and upper bounds $\ell(i), u(i) \in \{0, \ldots, B\}$ on the summands $\ell(i) \leq b_i \leq u(i)$, $i \in \{1, \ldots, d\}$. For short, we also denote these bounds as vectors $\mathbf{l}$ and $\mathbf{u}$. In the literature on combinatorial generation algorithms such summations with upper bounds only are known as *d-compositions with restricted parts*, see ([14] or [12]). There is a bijection to combinations of a multiset. All d-compositions with restricted parts can be enumerated by a constant amortized time (CAT) algorithm, which can be easily extended to the case with lower bounds without changing the CAT behavior. We give the modified algorithm `summations(B,d,U)` that enumerates the d-compositions with restricted parts in colexicographical order for convenience and refer to [14] for its unchanged analysis. The total number of d-compositions with restricted parts for given $\mathbf{l}$ and $\mathbf{u}$ is the Whitney number of the second kind of the finite chain product $(\overline{(u(1) - \ell(1)) + 1}) \times \cdots \times (\overline{u(r) - \ell(r)) + 1})$, where $\overline{x}$ denotes a chain of x elements, see again [14] for details.

Procedure `summations` (*position p, resource r, bound n*)

input : dimension d, lower bound vector $\mathbf{l}$, upper bound vector $\mathbf{u}$, sum B
output : `summations` (B, d, U), $U = \sum_{r=1}^{d} u(r)$ evaluates all d-compositions with restricted parts $\mathbf{b}$ with the above parameters.

```
if p = 0 then
  | evaluate b
else
  | for c ∈ {max(0, p − n + u(r) − ℓ(r)) ... min(u(r) − ℓ(r), p)} do
  |   | b_r ⟵ c + ℓ(r)
  |   | summations (p − c, r − 1, n − u(r) + ℓ(r))
  |   | b_r ⟵ ℓ(r)
  | end
end
```

The initial call is `summations(B,d,U)` for $U = \sum_{r=1}^{d} u(r) - \ell(r)$. This algorithm has CAT behavior for $B \leq U/2$. For $B > U/2$ there is a similar algorithm that can be adapted from algorithm `gen2` in [14]. We sum up the results of this section in the following theorem.

Theorem 5. *The d-compositions with restricted parts and lower bounds needed in algorithm RHE can be enumerated in constant amortized time.*

A.3 Evaluation

For the general problem with demand vectors $\mathbf{s}_i \in \mathbb{Q}_+^d$, $i \in \{1, \ldots, n\}$, the evaluation of a given bin vector $\mathbf{b}$ can be done in the obvious way in $O(n \cdot d)$ time: Scan through

the sequence starting at the last breakpoint π_ℓ (initially at $\pi_0 = 0$) updating the total demand vector $\mathbf{w}_{\pi_\ell,i}$ of the current bin until the addition of the next vector $\mathbf{s}_{i+1}$ in the sequence would make the demand vector exceed $\mathbf{b}$. Then add breakpoint $\pi_{\ell+1} = i$ and continue the scan with the next bin starting from there.

In the special case of sequential unit vector packing the runtime can be improved to $O(n)$, since for each demand vector only one of the d resources needs to be updated and checked.

For a single evaluation of a bin vector this algorithm is basically the best one can hope for. On the other hand, in the setting of our heuristic enumeration algorithm where many bin vectors are evaluated, the question arises, whether we can speed up the evaluations if we allow for preprocessing. We describe here an approach that we developed for our application, that is, an approach for the sequential unit vector packing problem with large values of n compared to k the number of breakpoints. It is possible to extend parts of the approach to the general problem with a loss in space efficiency.

A first simple approach builds an $(n \times d \times B)$ table T_1 as sketched in Figure 3. In this table we store in entry $T_1(p, r, \delta)$ the position of the next breakpoint in the sequence starting from position p for a bin vector $\mathbf{b}$ with demand δ for resource r, i.e., $b_r = \delta$, and $b_k = \infty$ for $k \neq r$. To evaluate a given bin vector $\mathbf{b}$ we start at position 1 and

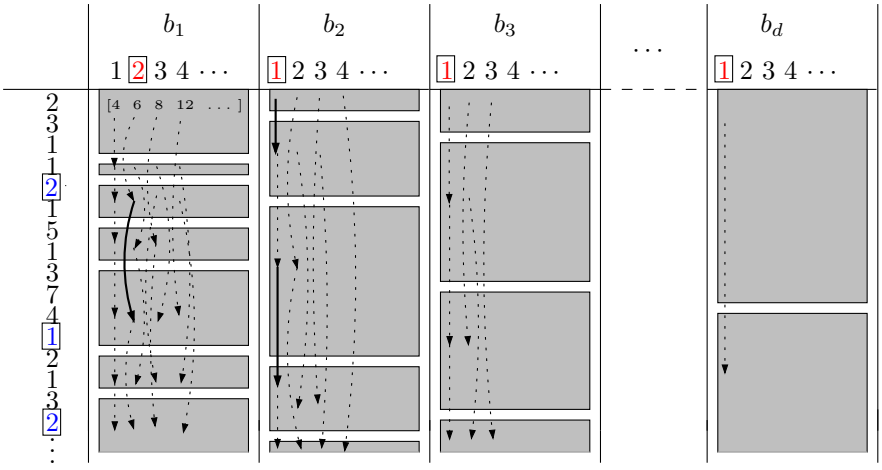


Fig. 3. Simple data structure that accelerates the evaluation of a bin vector. The rows correspond to positions in $\mathbf{S}$. The solid arcs show the minima of Equation 7 for the example bin vector $(2, 1, 1, \ldots, 1)$. Breakpoints are highlighted in the sequence (leftmost column). Arcs that point to blocks below the end of the figure are not shown.

inspect positions $(1, r, b_r)$ for $1 \leq r \leq d$. The next breakpoint must be at the minimum of these values. Thus we have

$$\pi_{i+1} = \min_{1 \leq r \leq d} T_1(\pi_i, r, b_r) \ . \tag{7}$$

Equation 7 directly gives an $O(kd)$ algorithm for the evaluation of a bin vector. Here k denotes as usual the number of breakpoints. On instances with $kd \ll n$ this is a speedup. The space complexity of this approach seems to be $\Theta(n \cdot d \cdot B)$ at first glance. But notice that between two occurrences of a resource r in the sequence the value of $T_1(\cdot, r, \cdot)$ remains the same. More precisely, if for all p' with $p_1 \leq p' < p_2$ it holds that $s_{p'} \neq r$, then we have $T_1(p_1, r, \delta) = T_1(p_2, r, \delta)$ for all δ. Let us call such an interval with equal entries for a given resource r a *block*. An example can be found in Figure 3, where the blocks are depicted as grey rectangles. The total number of blocks is bounded by $n + d = O(n)$ because at each position exactly one block ends in the setting of unit vector packing. Also the answer vectors in the blocks need not be stored explicitly: In the ith block of resource r the table entry for b_r is simply the end position of block $i + b_r$ as indicated by the arrows in the figure. Therefore, in our approach we store the block structure in an array of size $O(n)$ to get a constant lookup time for a given table entry $T_1(p, r, \delta)$. More precisely, we store d arrays $\{A_1, \ldots, A_d\}$ of total size $O(n)$, such that $\{A_r(i)\}$ gives the end position of block i of resource r or equivalently the position of the ith occurrence of r in $\mathbf{S}$. It is easy to see that the block structure can be (pre-)computed in linear time.

However, with this approach a different problem arises: After the computation of breakpoint π_{i+1}, we need to know at which positions we should access each of the arrays next. To answer this question we introduce a second table. Let T_2 be an $(n \times 2)$-table that stores in $T_2(p, 1)$ the index of the (unique) new block[3] that starts at position p and in $T_2(p, 2)$ the index of the current block of resource $(p \bmod d) + 1$ in array $A_{(p \bmod d)+1}$. In order to recompute the indices for breakpoint π_{i+1} we read the d rows $\{T_2(\pi_{i+1} - d + 1, \cdot), \ldots, T_2(\pi_{i+1}, \cdot)\}$. Each resource r occurs once in the second column of the read rows and might occur several times in the first column. As index for resource r take the value of the last occurrence of r in the read rows, regardless of the column, i.e., the occurrence with the highest row index. This approach correctly computes all new indices in the arrays $\{A_1, \ldots, A_d\}$ in $O(d)$ time, which is also the time that a single step takes without this index computation. Obviously, table T_2 needs $O(n)$ space. Alternatively, this table T_2 can be understood as a persistent version of the list containing for every resource its next occurrence, that is updated during a scan along the sequence. In this situation a general method for partially persistent data structures like [5] can be applied and yields the same time and space bounds. Altogether, we have shown the following theorem:

Theorem 4. *Given a sequence* $\mathbf{S}$ *we can construct a data structure in* $O(n)$ *preprocessing time with* $O(n \cdot d \cdot B)(O(n))$ *space such that an evaluation query for sequential (unit) vector packing for a bin vector* $\mathbf{b}$ *takes* $O(kd)$ *time, where* k *denotes the number of breakpoints for* $\mathbf{b}$.

Note that for RHE if we have already found a bin vector with k' many breakpoints we can stop all subsequent evaluations already after $k' \cdot d$ many steps.

[3] Strictly speaking the first column in table T_2 is not necessary as it simply reproduces the sequence. Here it clarifies the presentation and the connection to the technique in [5].

A.4 Mixed Integer Linear Program

Even if the ILP-formulation of Section 2.1 is a good starting point for our theoretical results, it turns out that in practice only medium sized instances can be solved with it. One problem is the potentially quadratic number of arc flow variables that makes the formulation prohibitive already for small instances. To reduce the number of variables, it is helpful to have an upper bound on the length of the longest arc. One such bound is of course B, but ideally there are smaller ones. As our real-world instances are windowed instances the window size is a trivial upper bound that helps to keep the number of variables low. A further problem is that, even if the bin vector is already determined, the MIP-solver needs to branch on the x and y variables to arrive at the final solution. This can be avoided by representing the bin vector components b_r as a sum of 0-1-variables z_i^r, such that $b_r = \sum_i z_i^r$ and $z_i^r \geq z_{i+1}^r$. If an arc a uses i units of resource r, i.e., the r-th entry of w_a is i, we include a constraint of the form $y_a \leq z_i^r$. This allows to use the arc only if there are at least i resources available. With these additional constraints, the arc variables y and x need no longer be binary. For integral values of z_i^r, only arc-variables that do not exceed the bin vector can have a value different from zero, so that in this case every fractional path of the resulting flow is feasible with respect to the bin vector, and thus all paths have the same (optimal) number of breakpoints (otherwise a shorter path could be selected resulting in a smaller objective function value). With this mixed integer linear program the number of branching nodes is drastically reduced, but the time spent at every such node is also significantly increased. Still, the overall performance of this program is a lot better than the original integer program, small instances (dimension 8) can now be solved to optimality within a few minutes. A bigger instance of dimension 22 and with a total bin size of up to 130 can be solved to optimality with this program within less than 3 hours, for different values of the total bin size.

We observed that on this mixed integer programm feasible solutions are found after a small fraction of the overall running time. Hence, we consider this approach also reasonable suited as a heuristic to come up with good solutions.

A.5 Figures

Table 1. summary information on the instances

name	demand unit vectors	dimension	window size	note
inst1	4464	24	28	
inst1-doubled	8928	48	56	inst1 with “doubled” resources
inst2	9568	8	28	
inst3	7564	7	28	
inst4	4464	22	28	
rand-doubled	2500	26	2500	random “doubled” instance

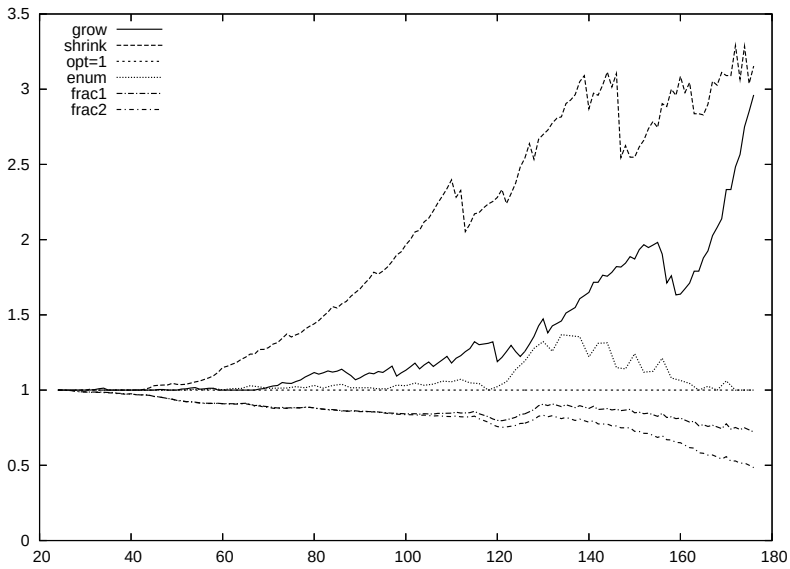
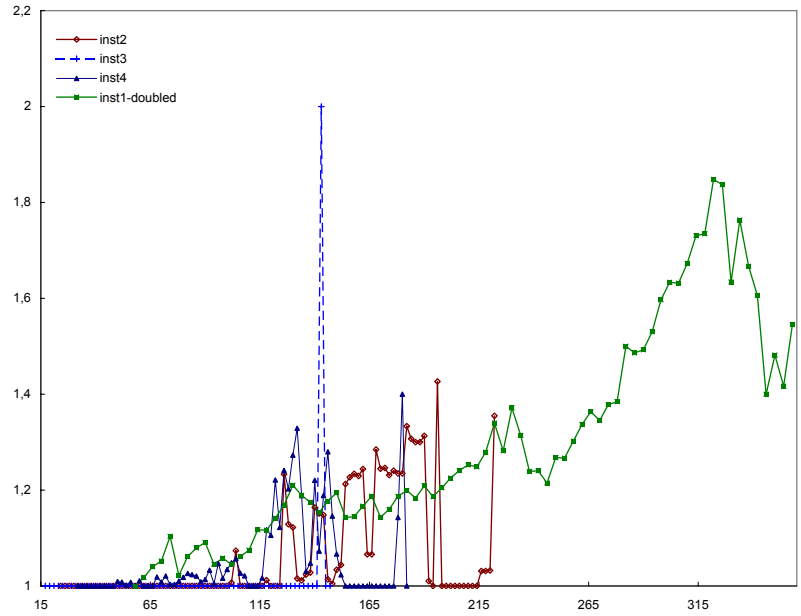
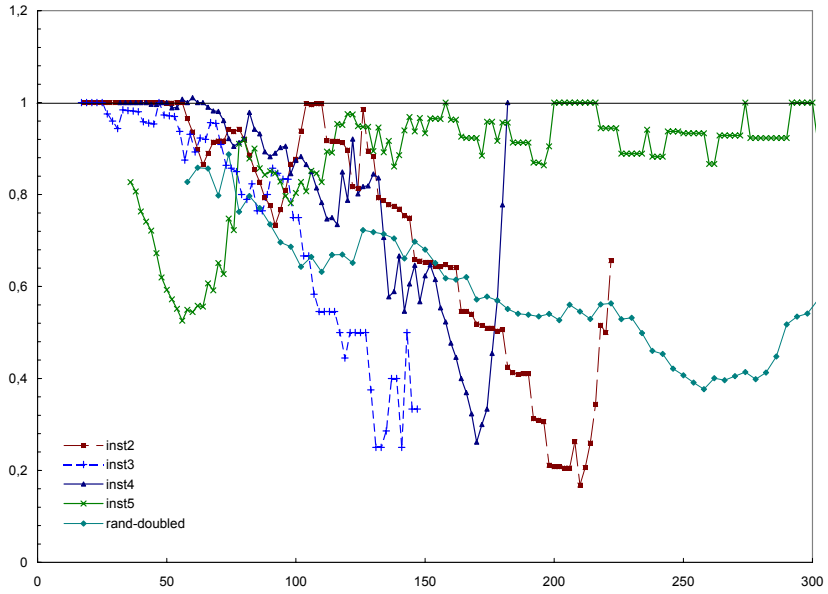


Fig. 4. Computational results for the different approaches. Plot of ratio of solution value to optimal solution value versus total bin size B



(a) Ratio of enumeration heuristic to optimal solution on five instances



(b) Ratio of enumeration heuristic to best of greedy algorithms on five instances

Fig. 5. Results on the instances of Table 1