The final publication is available at http://dx.doi.org/10.1016/j.cor.2015.11.009

Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression

Filipe Brandão
INESC TEC and Faculdade de Ciências, Universidade do Porto, Portugal
fdabrandao@dcc.fc.up.pt

João Pedro Pedroso
INESC TEC and Faculdade de Ciências, Universidade do Porto, Portugal
jpp@fc.up.pt

Abstract

We present an exact method, based on an arc-flow formulation with side constraints, for solving bin packing and cutting stock problems — including multi-constraint variants — by simply representing all the patterns in a very compact graph. Our method includes a graph compression algorithm that usually reduces the size of the underlying graph substantially without weakening the model.

Our formulation is equivalent to Gilmore and Gomory's, thus providing a very strong linear relaxation. However, instead of using column-generation in an iterative process, the method constructs a graph, where paths from the source to the target node represent every valid packing pattern.

The same method, without any problem-specific parameterization, was used to solve a large variety of instances from several different cutting and packing problems. In this paper, we deal with vector packing, bin packing, cutting stock, cardinality constrained bin packing, cutting stock with cutting knife limitation, bin packing with conflicts, and other problems. We report computational results obtained with many benchmark test datasets, some of them showing a large advantage of this formulation with respect to the traditional ones.

Keywords: Bin Packing, Cutting Stock, Vector Packing, Arc-flow Formulation

1. Introduction

The bin packing problem (BPP) is a combinatorial NP-hard problem (see, e.g., Garey and Johnson 1979) in which objects of different volumes must be packed into a finite number of bins, each with capacity W, in a way that minimizes the number of bins used.

The BPP can be seen as a special case of the cutting stock problem (CSP). In the CSP, items of equal size (which are usually ordered in large quantities) are grouped into orders with a required level of demand, while in the BPP the demand for a given size is usually close to one. According to Wäscher et al. (2007), cutting stock problems are characterized by a weakly heterogeneous assortment of small items, in contrast with bin packing problems, which are characterized by a strongly heterogeneous assortment of small items.

The p-dimensional vector bin packing problem (pD-VBP), also called general assignment problem by some authors, is a generalization of bin packing with multiple constraints. In this problem, we are required to pack n items of m different types, represented by p-dimensional vectors, into as few bins as possible. In practice, this problem models, for example, static resource allocation problems where the minimum number of servers with known capacities is used to satisfy a set of services with known demands.

The method presented in this paper allows solving several cutting and packing problems through reductions to vector packing. The reductions are made by defining a matrix of weights, a vector of capacities and a vector of demands. Our method builds very strong integer programming models that can usually be easily solved using any state-of-the-art mixed integer programming (MIP) solver. Computational results obtained with many benchmark test datasets show a large advantage of our method with respect to traditional ones.

In this paper, instances for all the problems will be presented as follows: p - number of dimensions; m - number of different item types; b_i - demand for items of type i; and, for

each dimension d, w_i^d is the weight of item i and W^d is the bin capacity. For the sake of simplicity the dimension may be omitted in the one-dimensional case.

The remainder of this paper is organized as follows. Section 2 presents Valério de Carvalho's arc-flow formulation for bin packing and cutting stock, which provides the basis for our method. The general arc-flow formulation, which is a generalization of Valério de Carvalho's model, is presented in Section 3. Section 4 introduces the multidimensional graphs that can be used to model p-dimensional vector bin packing problems. Section 5 presents the graph compression algorithm, which is crucial for solving large-scale instances. Computational results are provided in Section 6, and Section 7 presents the conclusions.

2. Previous work on exact methods

Valério de Carvalho (2002) provides an excellent survey on integer programming models for bin packing and cutting stock problems. In this section, we will introduce Valério de Carvalho (1999)'s arc-flow formulation, which is equivalent to Gilmore and Gomory (1961)'s formulation in terms of value of the linear relaxation. Gilmore and Gomory's model provides a very strong linear relaxation, but it is potentially exponential in the number of variables with respect to the input size; Valério de Carvalho's model is usually much smaller, being pseudo-polynomial in terms of decision variables and constraints.

2.1. Valério de Carvalho's arc-flow formulation

Among methods for solving BPP and CSP exactly, one of the most important is the arc-flow formulation with side constraints used by Valério de Carvalho (1999). This model has a set of flow conservation constraints and a set of demand constraints to ensure that the demand of every item is satisfied. The corresponding path-flow formulation is equivalent to the classical Gilmore and Gomory (1961)'s formulation.

Wolsey (1977) proposed for the first time an arc-flow formulation for cutting and packing problems. Despite not presenting computational results, some properties that suggested computational advantages of such formulation were presented. For solving cutting and packing

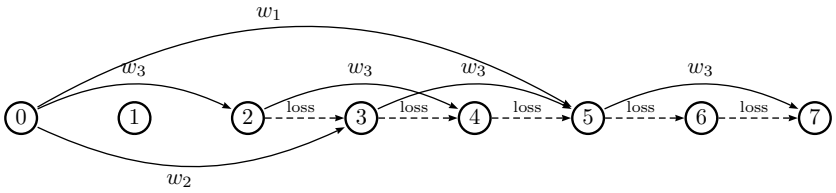
problems, computational experiments with arc-flow formulations were only performed much later by Valério de Carvalho (1999), where an arc-flow formulation was used as a basis to produce a branch-and-price algorithm.

In the one-dimensional case, the problem of determining a valid solution to a single bin can be modeled as the problem of finding a path in a directed acyclic graph $G = (V, A)$ with $V = \{0, 1, 2, \ldots, W\}$ and $A = \{(i, j) \mid j - i = w_t, \text{ for } t = 1..m \text{ and } 0 \leq i < j \leq W\}$, meaning that there exists an arc between two vertices i and $j > i$ if there are items of size $w_t = j - i$. The number of vertices and arcs are bounded by $\mathcal{O}(W)$ and $\mathcal{O}(mW)$, respectively. Additional arcs $(k, k+1)$, for $k = w_{min}, \ldots, W-1$, with w_{min} being the minimum item width, are included for representing unoccupied portions of the bin.

In order to reduce the symmetry of the solution space and the size of the model, Valério de Carvalho introduced some rules. The idea is to consider only a subset of arcs from A. If we search for a solution in which the items are ordered by decreasing values of weight, only paths in which items appear according to this order must be considered.

Example 1. *Figure 1 shows the graph associated with an instance with capacity $W = 7$ and items of sizes 5, 3, 2 with demands 3, 1, 2, respectively.*

Figure 1: Graph corresponding to Example 1.



The BPP and the CSP are thus equivalently formulated as that of determining the minimum flow between vertex 0 and vertex W, with additional constraints enforcing the sum of the flows in the arcs for each item type to be greater than or equal to the corresponding demand. Consider decision variables x_{ij} (associated with arcs (i, j) defined above) corresponding to the number of items of size $j - i$ placed in any bin at a distance of i units from the beginning of the bin. A variable z, representing the number of bins required, aggregates the flow in the graph, and can be seen as a feedback arc from vertex W to vertex 0. The model is as

follows:

$$\text{minimize} \qquad z \tag{1}$$

$$\text{subject to} \qquad \sum_{(i,j)\in A:j=k} x_{ij} - \sum_{(i,j)\in A:i=k} x_{ij} = \begin{cases} -z & \text{if } k = 0, \\ z & \text{if } k = W, \\ 0 & \text{for } k = 1, \ldots, W-1, \end{cases} \tag{2}$$

$$\sum_{(i,j)\in A:j=i+w_k} x_{ij} \geq b_k, \qquad i = k, \ldots, m, \tag{3}$$

$$x_{ij} \geq 0, \text{ integer}, \qquad \forall (i,j) \in A. \tag{4}$$

3. General arc-flow formulation

In this section, we propose a generalization of Valério de Carvalho's arc-flow formulation. The formulation is the following:

$$\text{minimize} \qquad z \tag{5}$$

$$\text{subject to} \qquad \sum_{(u,v,i)\in A:v=k} f_{uvi} - \sum_{(v,r,i)\in A:v=k} f_{vri} = \begin{cases} -z & \text{if } k = \text{S}, \\ z & \text{if } k = \text{T}, \\ 0 & \text{for } k \in V \setminus \{\text{S}, \text{T}\}, \end{cases} \tag{6}$$

$$\sum_{(u,v,j)\in A:j=i} f_{uvj} \geq b_i, \qquad i \in \{1, \ldots, m\} \setminus J, \tag{7}$$

$$\sum_{(u,v,j)\in A:j=i} f_{uvj} = b_i, \qquad i \in J, \tag{8}$$

$$f_{uvi} \leq b_i, \qquad \forall (u,v,i) \in A, \text{ if } i \neq 0, \tag{9}$$

$$f_{uvi} \geq 0, \text{ integer}, \qquad \forall (u,v,i) \in A, \tag{10}$$

where V is the set of vertices, S is the source vertex and T is the target; A is the set of arcs, where each arc has three components (u, v, i) corresponding to an arc between nodes u and v that contributes to the demand for items of type i; arcs (u, v, i) with $i = 0$ are the loss arcs;

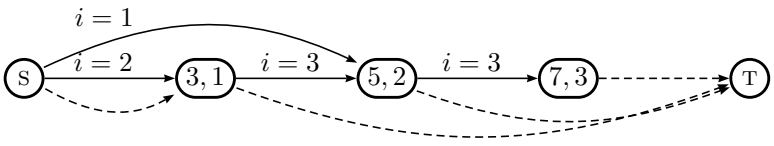
f_{uvi} is the amount of flow along the arc (u, v, i); and $J \subseteq \{1, \ldots, m\}$ is a subset of items for which we decide that demands are required to be satisfied exactly, for efficiency purposes. For having tighter constraints, one may set $J = \{i = 1, \ldots, m \mid b_i = 1\}$ (we have done this in our experiments); but the optimum for $J = \emptyset$ is the same.

In Valério de Carvalho's model, a variable x_{ij} contributes to an item with weight $j - i$. In our model, a variable f_{uvi} contributes to items of type i; the label of u and v may have no direct relation to the item's weight. This new model is more general; Valério de Carvalho's model is a sub-case, where an arc between nodes u and v can only contribute to the demand of an item of weight $v - u$. As in Valério de Carvalho's model, each arc can only contribute to an item, but the new model has several differences with respect to that formulation:

- nodes are more general (e.g., they can encompass multiple dimensions);
- there may be more than one arc between two vertices (multigraph);
- demands in general may be satisfied with excess but for some items they are required to be satisfied exactly (this allows, for example, the MIP solver to take advantage of special ordered sets of type 1 when requiring the demands of items with demand one to be satisfied exactly);
- arcs have upper bounds equal to the total demand of the associated item (which allows reducing the search space by excluding many feasible solutions that would exceed the demand);
- arc lengths are not tied to the corresponding item weight (i.e., $(u, v, i) \in A$ even if $v - u \neq w_i$).

Using this model it is possible to use more general graphs, such as the graph presented in Figure 2, but we always need to ensure that it is a directed acyclic graph whose paths from S to T correspond to every valid packing pattern to the original problem.

Figure 2: An alternative to the graph built in Figure 3.



i	w_i^1	w_i^2	b_i
1	5	1	3
2	3	1	1
3	2	1	2
$W^1 = 7$; $W^2 = 3$			

This graph results from applying the graph construction and compression algorithms presented in the following sections to a two-dimensional vector packing instance with bins of capacity (7,3), and items of sizes (5,1), (3,1), (2,1) with demands 3, 1, 2, respectively.

One of the properties of this model is the following.

Property 1 (equivalence to the classical Gilmore-Gomory model). *For a graph with all valid packing patterns represented as paths from* S *to* T*, model (5)-(10) is equivalent to the classical Gilmore-Gomory model with the same patterns as those obtained from paths in the graph.*

The proof is given in the online supplement of this paper.

After having the solution of the arc-flow integer optimization model, we use a flow decomposition algorithm to obtain the corresponding packing solution. Flow decomposition properties (see, e.g., Ahuja et al. 1993) ensure that non-negative flows can be represented by paths and cycles. Since we require an acyclic graph, any valid flow can be decomposed into directed paths connecting the only excess node (node S) to the only deficit node (node T).

By requiring the demand to be satisfied exactly for items with demand one, we take advantage of the fact that when an arc associated to one of those items is set to one, every other alternative arc is set to zero, and hence the search space is reduced substantially. However, by requiring the demands of some items to be satisfied exactly and by introducing upper bounds on variable values, the model may sometimes become harder to solve. For instance, if the demand of a very small item is required to be satisfied exactly, many optimal solutions will probably be excluded (unless the optimal solution has waste smaller than the item size). In some instances, the solution can be obtained more quickly by choosing carefully the variable's upper bounds and the set of items whose demand must be satisfied exactly. Overall, our

choices regarding these two aspects proved to work very well, as shown in Section 6. Our objective here is to help the MIP solvers to obtain the solution more quickly.

4. Graphs for *p*-dimensional vector packing

Valério de Carvalho's graph can be seen as the dynamic programming search space of the underlying one-dimensional knapsack problem. The vertices of the graph can be seen as states and, in order to model multi-constraint knapsack problems, we just need to add more information to them. In the one-dimensional case, arcs associated with items of weight w_i^1 lie between vertices (a) and $(a + w_i^1)$. In the multi-dimensional case, arcs associated with items of weight $(w_i^1, w_i^2, \ldots, w_i^p)$ lie between vertices $(a^1, a^2, \ldots, a^p)$ and $(a^1 + w_i^1, a^2 + w_i^2, \ldots, a^p + w_i^p)$.

4.1. Graph construction algorithm

Definition 1 (Order). *Items are sorted in decreasing order by the sum of normalized weights ($\alpha_i = \sum_{d=1}^{p} w_i^d / W^d$), using decreasing lexicographical order in case of a tie.*

The source vertex (s) is labeled with 0 in every dimension. Arcs associated with items of weight $(w_i^1, w_i^2, \ldots, w_i^p)$ are created between vertices $(a^1, a^2, \ldots, a^p)$ and $(a^1 + w_i^1, a^2 + w_i^2, \ldots, a^p + w_i^p)$. Since we just need to consider paths that respect a fixed order, we can have an arc with tail in a node only if it is either the source node or the head of an arc associated with a previous item type (according to the order of Definition 1). Our algorithm to construct the graph relies on this rule. Initially, there is only the source node. For each item type, we insert in the graph arcs associated with the item starting from all previously existing nodes. After processing an item, we add to the graph the set of new nodes that appeared as heads of new arcs. This process is repeated for every item and in the end we just need to connect every node, except the source, to the target, with the so-called loss arcs.. Using this algorithm, the graph can be constructed in pseudo-polynomial time $\mathcal{O}(|V|m)$, where $|V|$ is the number of vertices in the graph. Figure 3 shows a small example of the

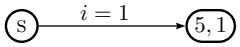
construction of a graph using this method. The pseudo-code of this algorithm is given in the online supplement of this paper.

Figure 3: Graph construction example.

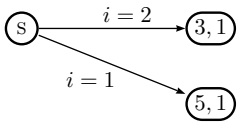
Consider a two-dimensional instance with bins of capacity (7,3), and items of sizes (5,1), (3,1), (2,1) with demands 3, 1, 2, respectively. This instance corresponds to adding cardinality limit 3 to Example 1.

a) We start with a graph with only the source node:

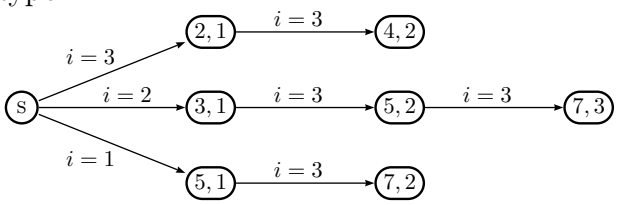
b) At the first iteration, we add arcs associated with the first item type, which has weight (5,1). There is space only for one item of this type.



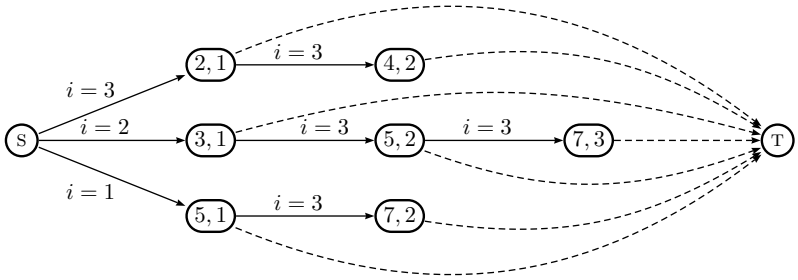
c) At the second iteration, we add arcs for the second item type, which has weight (3,1). The demand does not allow us to form paths with more than 1 consecutive arc for this item type.



d) At the third iteration, we add the arcs associated with the third item type, which has weight (2,1). Since the demand of this item is 2, we can add paths (starting from previously existing nodes) with at most 2 items of this type.



e) Finally, we add the loss arcs connecting each node, except the source, to the target. The resulting graph corresponds to the model to be solved by a general-purpose mixed-integer optimization solver.



In the one-dimensional case, the number of vertices and arcs in the arc-flow formulation is bounded by $\mathcal{O}(W)$ and $\mathcal{O}(mW)$, respectively, and thus graphs are usually reasonably small.

However, in the multi-dimensional case, the number of vertices and arcs are bounded by $\mathcal{O}(\prod_{d=1}^{p}(W^d+1))$ and $\mathcal{O}(m\prod_{d=1}^{p}(W^d+1))$, respectively. Despite the possible intractability indicated by these bounds, the graph compression method that we present in Section 5 usually leads to reasonably small graphs, even for very hard instances with hundreds of dimensions.

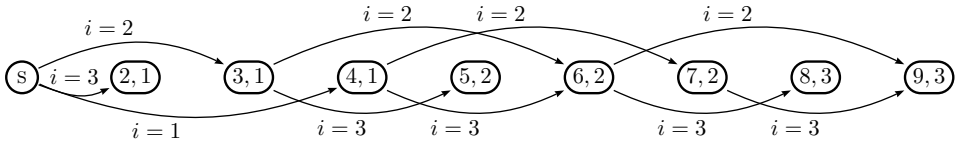
5. Graph compression

In Section 4, we presented an algorithm to build arc-flow graphs, which we will refer as Step-1 graphs, for vector packing problems. In this section, we will present a three-step graph compression method whose first step, which builds Step-2 graphs, consists of breaking the symmetry and is presented in Section 5.1. The two remaining compression steps, which build Step-3 and Step-4 graphs, are presented in Section 5.2. Finally, an alternative construction and compression algorithm based on the same concepts is presented in Section 5.3.

5.1. Symmetry breaking algorithm

Let us consider an instance with bins of capacity $W = (9, 3)$, and items of sizes (4,1), (3,1), (2,1) with demands 1, 3, 1, respectively. Figure 4 shows the Step-1 graph produced by the graph construction algorithm of Section 4.1 without the final loss arcs. This graph contains symmetry. For instance, the paths (s,(4,1),i=1) ((4,1),(7,2),i=2) ((7,2),(9,3),i=3) and (s,(4,1),i=1) ((4,1),(6,2),i=3) ((6,2),(9,3),i=2) correspond to the same pattern with one item of each type, but the second one does not respect the order of Definition 1.

Figure 4: Initial graph/Step-1 graph (with symmetry).



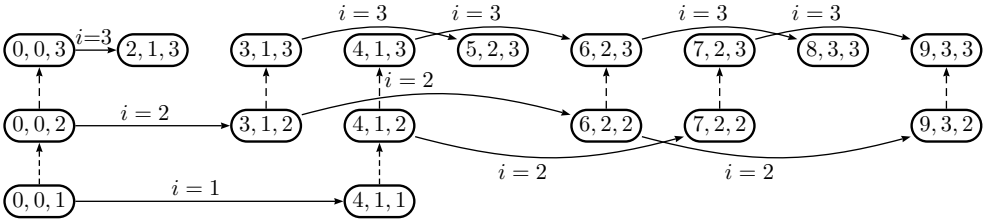
Graph corresponding to an instance with bins of capacity $W = (9, 3)$, and items of sizes (4,1), (3,1), (2,1) with demands 1, 3, 1, respectively.

An easy way to break symmetry is to divide the graph into levels, one level for each item type. We introduce in each node a new entry that indicates the level where it belongs. For exam-

ple, for the two-dimensional case, nodes (a', b') are transformed into sets of nodes $\{(a', b', i'), (a', b', i''), \ldots\}$. Each set has at most one node per level; nodes in consecutive levels are connected by loss arcs. Arcs $((a', b'), (a'', b''), i)$ are transformed into arcs $((a', b', i), (a'', b'', i), i)$. In level i, we have only arcs associated with items of type i. If we connect a node (a', b', i') to a node (a', b', i'') only in case $i' < i''$, we ensure that every path will respect the order (of Definition 1) and thus there is no symmetry. Recall that the initial graph must contain every valid packing pattern (respecting the order) represented as a path from S to T. The pseudo-code of the symmetry breaking algorithm is given in the online supplement of this paper.

Figure 5 shows the graph with levels (Step-2 graph) that results from applying this symmetry breaking method to the graph in Figure 4. Although there is no symmetry, there are still patterns that use some items more than their demand. To avoid this, other alternatives to break symmetry could be used; however this method is appropriate for the sake of simplicity and speed.

Figure 5: Graph with levels/Step-2 graph (without symmetry).



All the patterns respect the order since there are no arcs from higher levels to lower levels. Moreover, it is also easy to check that no valid pattern has been removed. In this graph, the source node is S $= (0, 0, 1)$ since it is the only node without arcs incident to it.

Property 2. *No valid pattern (respecting the order) is removed by breaking symmetry with levels as long as the original graph contains every valid packing pattern (respecting the order) represented as a path from* S *to* T.

The proof is given in the online supplement of this paper.

5.2. Graph compression algorithms

Breaking symmetry completely usually leads to much larger graphs; besides some symmetry may be helpful as long as it leads to substantial reductions in the graph size. In this section we show how to reduce the graph size by taking advantage of common sub-patterns that can be represented by a single sub-graph. This method may introduce some symmetry, but it usually helps by dramatically reducing the graph size. This graph compression method is composed of three steps, the first of which was presented in Section 5.1.

In the graphs we have seen in Section 4, a node label $(a^1, a^2, \ldots, a^p)$ means that, for every dimension d, every sub-pattern from the source to the node uses a^d space in that dimension. This means that a^d corresponds to the length of the longest path from the source to the node in dimension d. Similarly, the length of the longest path from a node to the target in each dimension can also be used as an entry in the node label and nodes with the same label can be combined into one single node. In the main compression step, given a graph $G = (V, A)$, a new graph $G' = (V', A')$ is constructed along these lines by creating a set of vertices $V' = \{\phi(v) \mid v \in V\}$ and a set of arcs $A' = \{(\phi(u), \phi(v), i) \mid (u, v, i) \in A, \phi(u) \neq \phi(v)\}$, where ϕ is the map between the original and new labels. This usually allows large reductions in the graph size. This reduction can be improved by breaking symmetry first (as described in the previous section), which allows us to consider only paths to the target respecting a specific order.

The main compression step is applied to the Step-2 graph. In the Step-3 graph, the longest paths to the target in each dimension are used to relabel the internal nodes ($V \setminus \{\text{S}, \text{T}\}$), dropping the level dimension label of each node. Let $(\varphi^1(u), \varphi^2(u), \ldots, \varphi^p(u))$ be the new label of node u in the Step-3 graph, where

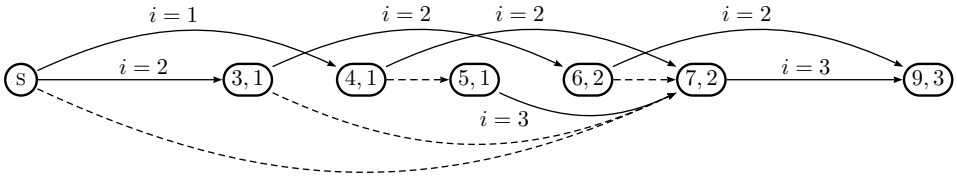
$$\varphi^d(u) = \begin{cases} W^d & \text{if } u = \text{T (base case)}, \\ \min_{(u',v,i)\in A:u'=u}\{\varphi^d(v) - w_i^d\} & \text{otherwise.} \end{cases} \tag{11}$$

For the sake of simplicity, we define w_0^d for loss arcs as zero in every dimension, $w_0^0 = w_0^1 = \ldots = w_0^p = 0$. In the paths from S to T in the Step-2 graph usually there is slack in some

dimension. In this process, we are moving this slack as much as possible to the beginning of the paths. The label in each dimension of every node u (except S) corresponds to the highest position (i.e., closest to T) where the sub-patterns from u to T can start in each dimension, respecting capacity constraints. By using these labels, we are allowing arcs to be longer than the items to which they are associated. We use dynamic programming to compute these labels in linear time in the graph size. The pseudo-code of this algorithm is given in the online supplement of this paper.

Figure 6 shows the Step-3 graph that results from applying the main compression step to the graph of Figure 5. Even in this small instance, a few nodes and arcs were removed, comparing with the initial graph of Figure 4.

Figure 6: Step-3 graph (after the main compression step).



The Step-3 graph has 8 nodes and 17 arcs (considering also the final loss arcs connecting internal nodes to T).

Finally, in the last compression step, a new graph is constructed once more. In order to try to reduce the graph size even more, we relabel the internal nodes once more using the longest paths from the source in each dimension. Let $(\psi^1(v), \psi^2(v), \ldots, \psi^p(v))$ be the label of node v in the Step-4 graph, where

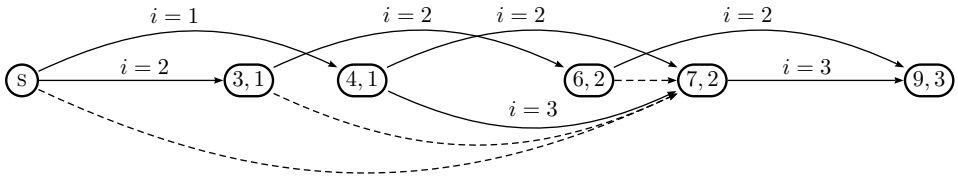
$$\psi^d(v) = \begin{cases} 0 & \text{if } v = \text{S (base case)}, \\ \max_{(u,v',i)\in A: v'=v}\{\psi^d(u) + w_i^d\} & \text{otherwise.} \end{cases} \tag{12}$$

The pseudo-code of this algorithm is given in the online supplement of this paper.

Figure 7 shows the Step-4 graph without the final loss arcs. This last compression step is not as important as the main compression step, but it nevertheless usually removes many nodes and arcs and is easy to compute.

Note that, in this case, the initial Step-1 graph had some symmetry and the final Step-4

Figure 7: Step-4 graph (after the last compression step).



The Step-4 graph has 7 nodes and 15 arcs (considering also the final loss arcs connecting internal nodes to T). In this case, the only difference from the Step-3 graph is the node $(5, 1)$ that collapsed with the node $(4, 1)$. The initial Step-1 graph had 10 nodes and 18 arcs.

graph does not contain any symmetry. Graph compression may increase symmetry in some situations, but in practice this is not a problem, because it usually leads to large reductions in the graph size. Since we are dealing with a very small instance, the improvement obtained by compression is not as substantial as in large instances. For example, in the standard BPP instance HARD4 from Scholl et al. (1997) with 200 items of 198 different sizes and bins of capacity $100,000$, we obtained reductions of 97% in the number of vertices and 95% in the number of the arcs. The solution of the resulting model was found in a few seconds by a MIP solver. Without graph compression it would be much harder to solve this kind of instances using the arc-flow formulation.

Property 3 (Graph compression 1). *Non-redundant patterns of the initial graph are not removed by graph compression.*

Property 4 (Graph compression 2). *Graph compression will not introduce any invalid pattern.*

The proofs of these properties, which assure the correctness of the graph compression algorithm, are given in the online supplement of this paper.

5.3. Building Step-3 graphs directly

As we said in Section 4, in the p-dimensional case, the number of vertices and arcs is limited by $\mathcal{O}(\prod_{d=1}^{p}(W^d + 1))$ and $\mathcal{O}(m \prod_{d=1}^{p}(W^d + 1))$, respectively. On the other hand, our graph compression method usually leads to very high compression ratios and in some cases it may lead to final graphs hundreds of times smaller than the initial ones. Hence, the size of the

initial graph can be the limiting factor and its construction should be avoided.

In practice, we build the Step-3 graph directly in order to avoid the construction of huge Step-1 and Step-2 graphs that may have many millions of vertices and arcs. Algorithm 1 uses dynamic programming to build the Step-3 graph recursively over the structure of the Step-2 graph (without building it). The basic idea for this algorithm comes from the fact that, in the main compression step, the label of any internal node ($\varphi^d(u) = \min_{(u',v,i)\in A:u'=u}\{\varphi^d(v) - w_i^d\}$) only depends on the labels of the two nodes to which it is connected; a node in its level (line 17) and another in the level above (line 14). After directly building the Step-3 graph from the instance's data using this algorithm, we just need to apply the last compression step to obtain the final graph. In practice, this method allows obtaining arc-flow models even for large benchmark instances quickly.

The dynamic programming states are identified by the space used in each dimension (x^d, for $d = 1, \ldots, p$), the current item (i) and the number of times it was already used (c). In order to reduce the number of states, we lift (line 10) each state by solving (using again dynamic programming though this is not explicit in the algorithm) knapsack/longest-path problems in each dimension considering the remaining items (line 4); we try to increase the space used in each dimension to its highest value considering the valid packing patterns for the remaining items. By solving multi-constraint knapsack problems in each dimension, we would obtain directly the corresponding label in the Step-3 graph; however, it is very expensive to solve this problem many times. By lifting states as a result of solving one-dimensional knapsack problems, we obtain a good approximation in a reasonable amount of time, and it usually allows us to take benefit of a substantial reduction in the number of states.

Algorithm 1: Direct Step-3 Graph Construction Algorithm

input : m - number of item types; w - item weights; b - demand; W - capacity vector
output: V - set of vertices; A - set of arcs; S - source; T - target

function `buildGraph`(m, w, b, W):
  dp$[x, i, c] \leftarrow$ NIL, **for all** x, i, c; *// dynamic programming table*
  function `lift`(x, i, c): *// auxiliary function: lift* dp *states solving knapsack/longest-path problems in each dimension*
    input : x - used capacity; i - current item type; c - number of times item i has been used
    function `highestPosition`(d, x, i, c):
      return $\min \; W^d - \sum_{j=i}^{m} w_j^d y_j$ s.t. $\sum_{j=i}^{m} w_j^d y_j \leq W^d - x^d$, $y_i \leq b_i - c$, $y_j \leq b_j, \quad j = i+1, ..., m$, $y_j \geq 0$, integer, $j = i, ..., m$;
    return (`highestPosition`$(1, x, i, c), \ldots,$ `highestPosition`(p, x, i, c));
  $V \leftarrow \{\}$; $A \leftarrow \{\}$;
  function `build`(x, i, c):
    input : x - used capacity; i - current item type; c - number of times item i has been used
    $x \leftarrow$ `lift`(x, i, c); *// lift x in order to reduce the number of* dp *states*
    if dp$[x, i, c] \neq$ NIL **then** *// avoid repeating work*
      return dp$[x, i, c]$;
    $u \leftarrow (W^1, ..., W^p)$; *// base case of $\varphi(x)$ if there are no arcs leaving the node*
    if $i < m$ **then** *// option 1: do not use the current item (go to the level above)*
      $up_x \leftarrow$ `build`$(x, i+1, 0)$;
      $u \leftarrow up_x$; *// value of $\varphi(x)$ if no more items of the current type are introduced*
    if $c < b_i$ **and** $x^d + w_i^d \leq W^d$ **for all** $1 \leq d \leq p$ **then** *// option 2: use the current item*
      $v \leftarrow$ `build`$((x^1 + w_i^1, \ldots, x^p + w_i^p), i, c+1)$;
      $u \leftarrow (\min(u^1, v^1 - w_i^1), \ldots, \min(u^p, v^p - w_i^p))$; *// updates the value of $\varphi(x)$*
      $A \leftarrow A \cup \{(u, v, i)\}$; $V \leftarrow V \cup \{u, v\}$; *// connects u to the node resulting from option 2*
    if $i < m$ **and** $u \neq up_x$ **then**
      $A \leftarrow A \cup \{(u, up_x, 0)\}$; $V \leftarrow V \cup \{up_x\}$; *// connects u to the node resulting from option 1*
    dp$[x, i, c] \leftarrow u$;
    return u; *// returns $u = \varphi(x)$*
  S $\leftarrow$ `build`$(x = (0, \ldots, 0), i = 1, c = 0)$; *// build the graph*
  $V \leftarrow V \cup \{$T$\}$;
  $A \leftarrow A \cup \{(u,$ T$, 0) \mid u \in V \setminus \{S, T\}\}$; *// connect internal nodes to the target*
  return $(G = (V, A),$ S, T$)$;

6. Applications and results

In this section, we present the results obtained using the arc-flow formulation in several cutting and packing problems. Results were obtained using a computer with two Quad-Core Intel Xeon at 2.66GHz, running Mac OS X 10.8.0, with 16 GBytes of memory (though only a few of the hardest instances required more than 4 GBytes of memory). The graph construction algorithm was implemented in `C++` (the source code is available online[1]), and the resulting MIP was solved using `Gurobi` 5.0.0, a state-of-the-art mixed integer programming solver. The parameters used in `Gurobi` were Threads = 1 (single thread), Presolve = 1 (conservative), Method = 2 (interior-point methods), MIPFocus = 1 (feasible solutions), Heuristics = 1, MIPGap = 0, MIPGapAbs = $1 - 10^{-5}$ and the remaining parameters were `Gurobi`'s default values. The use of interior-point methods at the root node considerably improves the time for solving the linear relaxation, compared to using the simplex algorithm. The branch-and-cut solver used in `Gurobi` uses a series of cuts; in our models, the most frequently used were Gomory, Zero half and MIR. Detailed results, log files and datasets are available online[2].

6.1. p-dimensional vector packing

In the p-dimensional vector packing problem, the set S of valid patterns is defined as follows:

$$A = \begin{bmatrix} w_1^1 & \dots & w_m^1 \\ \vdots & & \vdots \\ w_1^p & \dots & w_m^p \end{bmatrix} \qquad L = \begin{bmatrix} W^1 \\ \vdots \\ W^p \end{bmatrix} \qquad S = \{\mathbf{x} \in \mathbb{Z}_{\geq 0}^m : A\mathbf{x} \leq L\} \tag{13}$$

[1] `http://www.dcc.fc.up.pt/~fdabrandao/code`

[2] `http://www.dcc.fc.up.pt/~fdabrandao/research/vpsolver/results/`

where A is the matrix of weights and L is the vector of capacities. The set S is the set of valid packing patterns that satisfy all the following knapsack constraints:

$$w_1^1 x_1 + w_2^1 x_2 + \ldots + w_m^1 x_m \leq W^1 \tag{14}$$

$$w_1^2 x_1 + w_2^2 x_2 + \ldots + w_m^2 x_m \leq W^2 \tag{15}$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

$$w_1^p x_1 + w_2^p x_2 + \ldots + w_m^p x_m \leq W^p \tag{17}$$

$$x_i \geq 0, \text{ integer}, \ i = 1, \ldots, m, \tag{18}$$

Two-constraint bin packing (2CBP) is a 2-dimensional vector packing problem. We used the proposed arc-flow formulation to solve to optimality 330 of the 400 instances from the DEIS-OR's two-constraint bin packing test dataset[3], which was proposed by Caprara and Toth (2001). Table 1 summarizes the results. This dataset has several sizes n for each class, each pair (class, size) having 10 instances.

Table 1: Results for 2-dimensional vector packing.

	$n \in \{24, 25\}$			$n \in \{50, 51\}$			$n \in \{99, 100\}$			$n \in \{200, 201\}$		
class	N^{bb}	T^{tot}	#open	N^{bb}	T^{tot}	#open	N^{bb}	T^{tot}	#open	N^{bb}	T^{tot}	#open
1	0.0	0.12	0	0.0	1.62	0	0.0	66.96	5	0.0	7,601.35	7
2	0.0	0.01	10	0.0	0.04	10	0.0	0.21	10	0.0	6.93	10
3	0.0	0.01	10	0.0	0.02	10	0.0	0.05	10	0.0	0.20	10
4	0.0	21.97	10	-	-	-	-	-	-	-	-	-
5	0.0	10.62	10	-	-	-	-	-	-	-	-	-
6	0.0	0.02	0	0.0	0.06	1	0.0	0.31	5	0.0	4.75	8
7	0.0	0.03	0	0.0	0.14	1	0.3	1.69	7	31.4	14.01	3
8	0.0	0.01	10	0.0	0.02	10	0.0	0.07	10	0.0	0.24	10
9	0.0	0.10	0	0.0	0.66	1	0.0	28.11	10	-	-	-
10	0.0	0.02	0	0.0	0.10	0	0.0	0.66	0	226.9	155.43	0

n - total number of items; N^{bb} - average number of nodes explored in the branch-and-bound procedure; T^{tot} - average run time in seconds; #open - number of previously open instances solved; "-" means that the subclass was not solved by the proposed algorithm.

Note that among the instances presented in Table 1 there were 188 instances with no previously known optimum. The arc-flow formulation allowed the solution of 330 instances out of the 400 instances; hence, there remain 70 open instances. The graph compression algo-

[3] `http://www.or.deis.unibo.it/research_pages/ORinstances/ORinstances.htm`

rithm is remarkably effective on all of this subset of two-constraint bin packing instances. In many cases, graph compression allowed the removal of more than 90% of the vertices and arcs; without it, it would not be viable to solve many of these instances within a reasonable amount of time.

One may ask why so many of these instances could not be solved before. A reasonable explanation may be the fact that, for example, in instances from classes 2, 3 and 8 (none of the instances from these classes had been solved before), the lower bound provided by the linear relaxation of assignment-based formulations is rather loose. Also, in some of these instances, the average number of items per bin in the optimal solution is reasonably large, which leads to an extremely large number of possible patterns. Results presented in Caprara and Toth (2001) show that solving the linear relaxation of many of these instances using Gilmore-Gomory's model and column-generation was not possible within 100,000 seconds (more than 27 hours). The computer they used is much slower than the one used here, but this shows how hard it can be to compute a linear relaxation of Gilmore-Gomory's model as the number of patterns increases and the subproblems become harder to solve. With the arc-flow formulation, graph compression leads to very large reductions in the graph size and allows us to represent all these patterns in reasonably small graphs.

The classes 4 and 5 are hard even for our arc-flow formulation due to a large number of items that fit in a single bin, which leads to a huge number of valid packing patterns that are difficult to represent in a compact way. Most of the instances that remain open belong to these two classes. The seven subclasses that we did not solve (appearing as "-" in Table 1) contain at least one instance that takes more than 12 hours to be solved exactly. The average run time in the 330 solved instances was 4 minutes, and none of these instances took longer than 5 hours to be solved exactly.

In order to test the behavior of the arc-flow formulation in instances with the same characteristics and more dimensions, we created 20-dimensional vector packing instances by combining the ten 2-dimensional vector packing instances of each subclass (class, size) into

one instance. Table 2 summarizes the results. The arc-flow formulation allowed the solution of 33 out of the 40 instances. The same subclasses that were solved in the 2-dimensional case were also solved in the 20-dimensional case. Moreover, some 20-dimensional instances were easier to solve than the original 2-dimensional ones due to the reduction in the number of valid packing patterns. Seven instances were not solved within a 12 hour time limit; these are instances in which the patterns are very long. The average run time for the remaining 33 solved instances was 48 seconds, and none of these instances took longer than 23 minutes to be solved exactly.

Table 2: Results for 20-dimensional vector packing.

	$n \in \{24, 25\}$		$n \in \{50, 51\}$		$n \in \{99, 100\}$		$n \in \{200, 201\}$	
class	N^{bb}	T^{tot}	N^{bb}	T^{tot}	N^{bb}	T^{tot}	N^{bb}	T^{tot}
1	0	0.09	0	0.81	0	36.28	0	1,374.18
2	0	0.01	0	0.01	0	0.01	0	0.01
3	0	0.01	0	0.01	0	0.01	0	0.01
4	0	50.27	-	-	-	-	-	-
5	0	73.20	-	-	-	-	-	-
6	0	0.01	0	0.02	0	0.05	0	0.19
7	0	0.01	0	0.03	0	0.06	0	0.19
8	0	0.01	0	0.01	0	0.03	0	0.10
9	0	0.05	0	0.37	0	12.80	-	-
10	0	0.02	0	0.11	0	0.91	0	14.52

n - total number of items; N^{bb} - number of nodes explored in the branch-and-bound procedure; T^{tot} - run time in seconds; "-" means that the instance was not solved by the proposed algorithm.

6.2. Bin packing and cutting stock

Standard bin packing and cutting stock are one-dimensional vector packing problems whose set S of valid patterns is defined as follows:

$$A = \begin{bmatrix} w_1 & \dots & w_m \end{bmatrix} \qquad L = \begin{bmatrix} W \end{bmatrix} \qquad S = \{\mathbf{x} \in \mathbb{Z}^m_{\geq 0} : A\mathbf{x} \leq L\} \tag{19}$$

We used the arc-flow formulation to solve a large variety of bin packing and cutting stock test datasets. OR-Library[4] provides two bin packing test datasets that were proposed by Falkenauer (1996). The first dataset (BFLK_u) is composed of uniform instances, where

[4] `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`

items have randomly generated weights, and the second dataset (BFLK_t) is composed of triplets instances, where each bin is completely filled with three items in the optimal solution. Each of these is further divided into subclasses of varying sizes. Scholl et al. (1997) provides three datasets that are available online[5]. The first of them (Scholl C1) is composed of randomly generated instances whose expected number of items per bin is not larger than 3. The second test dataset (Scholl C2) is composed of instances whose expected average number of items per bin is 3, 5, 7 or 9. Finally, the third test dataset (Scholl C3) is composed of 10 difficult instances with a total number of items $n = 200$ and bins of capacity $W = 100,000$. Schoenfield (2002) provides a bin packing test dataset (Hard28) composed of 28 instances selected from a huge testing. Among these 28 instances, 5 are non-IRUP (see, e.g., Baum and L. E. Trotter 1981), so the integer programming solver has to use branch-and-cut to reduce the gap and prove optimality. The remaining instances are IRUP, yet very hard for heuristics. ESICUP[6] provides two test datasets collected from Schwerin and Wäscher (1997) and Wäscher and Gau (1996) (SCH/WAE and WAE/GAU, respectively). We generated cutting stock datasets (CFLK_u and CFLK_t) from Falkenauer's bin packing test datasets by multiplying the demand of each item by one million, hence creating instances with one billion items; note that, however, multiplying the demand of each item by one million has very little effect on the model size. Two additional large test datasets for cutting stock problems are available online[7]. The first dataset (Cutgen) is composed of 1800 randomly generated instances of 18 classes. These instances were generated using the problem generator proposed by Gau and Wäscher (1995). The second dataset (Fiber) was taken from a real application in a chemical fiber company in Japan. Finally, a cutting stock test dataset (1Dbar) was obtained from 1D-bar relaxations of the two-dimensional bin packing test dataset of Lodi et al. (1999). Tables 3 and 4 present some characteristics of these datasets. The results are summarized in Table 5. No instance took longer than 5 hours to be solved to optimality. The average run time for the 4,114 instances was 13 seconds; 97% of these instances were

[5] http://www.wiwi.uni-jena.de/entscheidung/binpp/

[6] http://paginas.fe.up.pt/~esicup/

[7] http://www-sys.ist.osaka-u.ac.jp/~umetani/benchmark.html

solved in less than 1 minute each.

The run time is usually very dependent on the graph size, as we show in Section 6.7. The hardest BPP and CSP instances for our method usually combine very small items and very large capacities, which tend to lead to huge graphs, due to the number of patterns that need to be represented. The performance of other methods such as Valério de Carvalho (1999)'s are very dependent on the capacity limits; in our method, graph compression attenuates this problem. We were able to solve easily all the instances from the third Scholl's dataset, which have capacity $100,000$, since the items are not very small; in Valério de Carvalho's model there would be around $80,000$ constraints, while in our model there are around $2,000$. Brandão (2012) compares the performance of our method with Valério de Carvalho's arc-flow formulation, and even when the capacities are small the gains of using graph compression are substantial.

Table 3: Demand characteristics of BPP/CSP datasets.

dataset	type	m^{avg}	m^{min}	m^{max}	n^{avg}	n^{min}	n^{max}	b^{avg}	b^{min}	b^{max}
BFLK_u	BPP	75.56	58	81	467.50	120	1,000	5.92	1	24
BFLK_t	BPP	117.61	46	203	232.50	60	501	1.74	1	18
Scholl C1	BPP	63.78	31	100	212.50	50	500	2.96	1	20
Scholl C2	BPP	97.58	28	350	212.50	50	500	2.31	1	22
Scholl C3	BPP	199.00	197	200	200.00	200	200	1.01	1	3
Hard28	BPP	161.75	136	189	181.43	160	200	1.12	1	3
SCH/WAE	BPP	45.09	39	49	110.00	100	120	2.44	1	10
WAE/GAU	BPP	49.65	33	64	129.41	57	239	2.60	1	38
CFLK_u	CSP	75.56	58	81	4.68E8	1.20E8	1.00E9	5.92E6	1.0E6	2.40E7
CFLK_t	CSP	117.61	46	203	2.33E8	6.00E7	5.01E8	1.74E6	1.0E6	1.80E7
Fiber	CSP	10.79	4	20	418.79	155	1,121	42.82	1	555
Cutgen	CSP	22.46	8	40	1,283.37	100	4,000	56.81	1	504
1Dbar	CSP	60.00	20	100	2,140.19	93	7,478	35.67	1	100

Average, minimum and maximum values for: m - number of different items; n - number of items; b - demand.

Table 4: Size characteristics of BPP/CSP datasets.

dataset	type	W^{avg}	W^{min}	W^{max}	w^{avg}	w^{min}	w^{max}	r^{avg}	r^{min}	r^{max}
BFLK_u	BPP	150.00	150	150	60.35	20	100	0.40	0.13	0.67
BFLK_t	BPP	1,000.00	1,000	1,000	333.33	250	499	0.33	0.25	0.50
Scholl C1	BPP	123.33	100	150	58.56	1	100	0.49	0.01	1.00
Scholl C2	BPP	1,000.00	1,000	1,000	195.44	13	627	0.20	0.01	0.63
Scholl C3	BPP	100,000.00	100,000	100,000	27,503.03	20,000	35,000	0.28	0.20	0.35
Hard28	BPP	1,000.00	1,000	1,000	386.73	1	800	0.39	0.00	0.80
SCH/WAE	BPP	1,000.00	1,000	1,000	174.81	150	200	0.17	0.15	0.20
WAE/GAU	BPP	10,000.00	10,000	10,000	1,577.48	2	7,332	0.16	0.00	0.73
CFLK_u	CSP	150.00	150	150	60.35	20	100	0.40	0.13	0.67
CFLK_t	CSP	1,000.00	1,000	1,000	333.33	250	499	0.33	0.25	0.50
Fiber	CSP	7,080.00	5,180	9,080	930.23	500	2,000	0.14	0.06	0.39
Cutgen	CSP	1,000.00	1,000	1,000	344.59	10	800	0.34	0.01	0.80
1Dbar	CSP	98.00	10	300	35.14	1	100	0.41	0.00	1.00

Average, minimum and maximum values for: W - capacity; w - item sizes; r - relative item sizes.

Table 5: Results for the standard BPP/CSP.

dataset	type	#inst.	$\#v$	$\#a$	$\%v$	$\%a$	N^{bb}	T^{tot}
BFLK_u	BPP	80	107.16	2,620.26	19%	14%	1.81	0.34
BFLK_t	BPP	80	125.35	4,987.63	80%	75%	1.10	0.92
Scholl C1	BPP	720	72.32	1,309.33	35%	36%	0.00	0.15
Scholl C2	BPP	480	596.99	30,824.36	33%	37%	0.10	43.43
Scholl C3	BPP	10	1,810.20	80,180.10	96%	95%	0.00	12.17
Hard28	BPP	28	789.46	27,284.00	19%	26%	102.57	29.69
SCH/WAE	BPP	200	210.11	3,553.57	70%	70%	0.00	0.62
WAE/GAU	BPP	17	6,235.06	128,212.76	33%	37%	2.59	1,641.09
CFLK_u	CSP	80	108.69	2,657.41	17%	13%	0.00	0.36
CFLK_t	CSP	80	122.06	5,032.21	81%	75%	0.00	0.83
Fiber	CSP	39	253.38	1,631.79	73%	71%	0.00	0.29
Cutgen	CSP	1,800	339.85	4,204.77	58%	51%	0.06	1.98
1Dbar	CSP	500	87.25	2,111.17	10%	7%	0.00	0.42

#inst. - number of instances; $\#v, \#a$ - average number of vertices and arcs in the final arc-flow graph; $\%v, \%a$ - average percentage of vertices and arcs removed by the graph compression method. N^{bb} - average number of nodes explored in the branch-and-bound procedure; T^{tot} - average run time in seconds.

6.3. Cardinality constrained bin packing and cutting stock

One of the BPP variants is the cardinality constrained bin packing in which, in addition to the capacity constraint, the number of items per bin is also limited. Similarly, one of the variants of the CSP is cutting stock with cutting knife limitation in which there is a limit on the number of pieces that can be cut from each roll due to a limit on the number of knives. BPP and CSP with cardinality constraints can be seen as special cases of the 2-dimensional vector packing problem. The set S of valid packing patterns for these problems is defined as follows:

$$A = \begin{bmatrix} w_1 & \dots & w_m \\ 1 & \dots & 1 \end{bmatrix} \qquad L = \begin{bmatrix} W \\ C \end{bmatrix} \qquad S = \{\mathbf{x} \in \mathbb{Z}^m_{\geq 0} : A\mathbf{x} \leq L\} \tag{20}$$

Cardinality constrained bin packing is strongly NP-hard for any cardinality larger than 2 (see, e.g., Epstein and van Stee 2011); for cardinality 2, the cardinality constrained bin packing problem can be solved in polynomial time as a maximum non-bipartite matching problem in a graph where each item is represented by a node and every compatible pair of items is connect by an edge.

We solved using the arc-flow formulation every instance from the bin packing and cutting stock datasets with cardinalities between 2 and the minimum cardinality limit that allowed the optimum to be the same with or without cardinality constraints. Table 6 summarizes the results for each dataset. No instance took longer than 8 hours to be solved to optimality. The average run time for the 14,568 instances was 13 seconds; 99% of these instances were solved in less than 1 minute each.

Graph compression reduces substantially the graph sizes and usually leads to graphs of size comparable to the size without cardinality constraints. In fact, there are instances in which cardinality constraints help to reduce the final graph size, thus leading to easier models. The arc-flow model allowed us to solve the cardinality constrained BPP/CSP as easily as the standard BPP/CSP. We are not aware of any good method in the literature for solving

Table 6: Results for the cardinality constrained BPP/CSP.

dataset	type	#inst.	$C^{\max}$	$\#v$	$\#a$	$\%v$	$\%a$	N^{bb}	$\mathrm{T}^{\mathrm{tot}}$
BFLK_u	BPP	160	3	53.04	789.81	79%	79%	0.00	0.11
BFLK_t	BPP	160	3	64.13	2,610.92	92%	86%	2.88	0.58
Scholl C1	BPP	1,189	4	42.49	451.39	80%	81%	0.00	0.06
Scholl C2	BPP	2,529	10	301.93	8,927.75	89%	90%	0.05	14.71
Scholl C3	BPP	30	4	624.70	27,181.63	99%	98%	0.00	5.73
Hard28	BPP	56	3	100.73	1,991.96	94%	94%	25.13	0.82
SCH/WAE	BPP	1,000	6	74.90	924.78	89%	90%	0.00	0.20
WAE/GAU	BPP	131	18	6,833.89	103,768.11	91%	90%	0.73	999.58
CFLK_u	CSP	160	3	52.85	792.15	79%	79%	0.04	0.10
CFLK_t	CSP	160	3	62.08	2,632.82	93%	86%	0.00	0.44
Fiber	CSP	279	12	83.23	525.35	94%	90%	0.00	0.07
Cutgen	CSP	7,299	18	253.15	2,459.78	93%	89%	0.06	1.26
1Dbar	CSP	1,415	8	59.81	965.96	82%	79%	0.05	0.25

#inst. - number of instances; $C^{\max}$ - maximum cardinality limit; $\#v, \#a$ - average number of vertices and arcs in the final arc-flow graph; $\%v, \%a$ - average percentage of vertices and arcs removed by the graph compression method. N^{bb} - average number of nodes explored in the branch-and-bound procedure; $\mathrm{T}^{\mathrm{tot}}$ - average run time in seconds.

the cardinality constrained BPP/CSP in general.

6.4. Graph coloring

The graph coloring problem is a combinatorial NP-hard problem in which one has to assign a color to each vertex of a graph in such a way that no two adjacent vertices share the same color and by using the minimum number of colors (see, e.g., Garey and Johnson 1979).

Graph coloring can be reduced to vector packing in several ways. Let variables x_i of constraints (14)-(17) represent whether or not vertex i appears in a given pattern (each pattern corresponds to a set of vertices that can share the same color). Considering each color as a bin and each vertex as an item with demand one, the following reductions are valid:

- Adjacency constraints: For each pair of adjacent vertices i and j, there is an adjacency constraint $x_i + x_j \leq 1$. Each adjacency constraint can be represented by a dimension k of capacity $W^k = 1$, with $w_i^k = w_j^k = 1$.

- Degree constraints: Let $\deg(i)$ and $\mathrm{adj}(i)$ be the degree and the list of adjacent vertices of vertex i, respectively. For each vertex i, there is a constraint $\deg(i)x_i + \sum_{j \in \mathrm{adj}(i)} x_j \leq \deg(i)$. Each constraint can be represented by a dimension k of capacity $W^k = \deg(i)$,

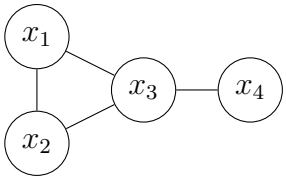
with $w_i^k = \deg(i)$ and $w_j^k = 1$ for every $j \in \text{adj}(i)$.

- Clique constraints: For each clique C, there is a constraint $\sum_{i \in C} x_i \leq 1$. Each clique constraint can be represented by a dimension k of capacity $W^k = 1$, with $w_i^k = 1$ for every $i \in C$. Bron and Kerbosch (1973)'s algorithm can be used to decompose the graph into maximal cliques.

Using any of the three reductions above, a vector packing solution with z bins corresponds to a graph coloring solution with z colors. Different reductions result in different vector packing instances that can be harder or easier to solve. According to our experiments, it is usually a good idea to choose reductions that lead to vector packing instances with fewer dimensions. Figure 8 illustrates the three reductions defined above.

Figure 8: Graph coloring reductions to vector packing.

Graph coloring instance:



Adjacency constraints:

$$\begin{aligned} x_1+x_2 &\leq 1, \\ x_1+x_3 &\leq 1, \\ x_2+x_3 &\leq 1, \\ x_3+x_4 &\leq 1, \\ x_i \in \{0,1\},\ i &= 1..4 \end{aligned}$$

Degree constraints:

$$\begin{aligned} 2x_1+x_2+x_3 &\leq 2, \\ 2x_2+x_1+x_3 &\leq 2, \\ 3x_3+x_1+x_2+x_4 &\leq 3, \\ 1x_4+x_3 &\leq 1, \\ x_i \in \{0,1\},\ i &= 1..4 \end{aligned}$$

Clique constraints:

$$\begin{aligned} x_1+x_2+x_3 &\leq 1, \\ x_3+x_4 &\leq 1, \\ x_i \in \{0,1\},\ i &= 1..4 \end{aligned}$$

Note that, in graph coloring, the lengths of the patterns are usually very long, especially when the graphs are sparse. However, there are problems that can be reduced to graph coloring problems with reasonably short patterns, and thus it may be possible to solve them using the proposed arc-flow model. Some computational results for this problem and to timetabling reduced to graph coloring are given in the online supplement of this paper. Despite not being the most appropriate method for this type of problems, these applications show the flexibility of our method.

6.5. Bin packing with conflicts

The bin packing problem with conflicts (BPPC) is one of the most important bin packing variants. This problem consists of the combination of bin packing with graph coloring. In

addition to the capacity constraints, there are compatibility constraints. This problem can be solved as a vector packing problem with $c + 1$ dimensions, where c is the number of dimensions used to model conflicts. The set S of valid packing patterns for this problem can be defined as follows:

$$A = \begin{bmatrix} w_1 & \dots & w_n \\ \alpha_1^1 & \dots & \alpha_n^1 \\ \vdots & & \vdots \\ \alpha_1^c & \dots & \alpha_n^c \end{bmatrix} \qquad L = \begin{bmatrix} W \\ \beta^1 \\ \vdots \\ \beta^c \end{bmatrix} \qquad S = \{\mathbf{x} \in \mathbb{Z}^n_{\geq 0} : A\mathbf{x} \leq L\} \tag{21}$$

The conflicts (the last c rows of A and L) can be modeled using any of the graph coloring reductions to vector packing presented in Section 6.4. In our experiments, we used degree constraints for modeling constraints in this problem.

In order to test the arc-flow formulation in the BPPC, we used the dataset proposed by Fernandes-Muritiba et al. (2010). This dataset was created from the bin packing dataset of Falkenauer (1996), adding conflict graphs with several densities. Tables 7 and 8 summarize the results for each class and density, respectively. Fernandes-Muritiba et al. (2010) solved some of theses instances using a branch-and-price algorithm under a time limit of 10 hours. Sadykov and Vanderbeck (2013) solved all the instances within an one-hour time limit using a branch-and-price algorithm. As opposed to our method, both branch-and-price algorithms of Fernandes-Muritiba et al. (2010) and Sadykov and Vanderbeck (2013) included algorithms tailored for bin packing with conflicts, namely special purpose algorithms form solving the subproblem. The instances of class u1000 were the most difficult for our method. Note that instances of this class have 1000 items and, in the literature, instances with 200 items are already considered difficult even in the one-dimensional case. Nevertheless, no instance took longer than 50 minutes to be solved exactly. The average run time of our method in the 800 instances was 2 minutes and 80% of these instances were solved in less than 1 minute each. In this problem, due to the high number of dimensions, a large part of the run time is spent building the arc-flow graph.

Table 7: Results for the BPP with conflicts.

class	#inst.	n	d	$d^{\max}$	$\#v$	$\#a$	T^{pp}	T^{lp}	T^{bb}	N^{bb}	T^{tot}
u120	100	120	84.96	121	359.99	3,012.12	0.18	0.06	0.18	0.07	0.43
u250	100	250	175.21	251	1,167.75	9,393.41	2.21	0.37	1.29	0.00	3.86
u500	100	500	350.48	501	3,486.90	27,440.63	32.40	1.87	10.88	0.00	45.16
u1000	100	1,000	702.21	1001	11,316.90	88,323.32	643.68	13.24	104.79	0.00	761.72
t60	100	60	42.22	61	99.69	693.14	0.03	0.01	0.02	0.00	0.06
t120	100	120	84.01	121	298.36	2,627.20	0.20	0.05	0.27	3.70	0.52
t249	100	249	174.42	250	1,045.20	10,300.08	2.82	0.32	1.65	3.12	4.78
t501	100	501	352.26	502	3,796.35	36,809.01	53.90	2.56	17.50	0.00	73.95

Table 8: Results for the BPP with conflicts (grouped by density).

density	#inst.	d	$d^{\max}$	$\#v$	$\#a$	T^{pp}	T^{lp}	T^{bb}	N^{bb}	T^{tot}
0%	80	1.00	1	113.47	12,288.20	0.28	0.24	1.24	0.00	1.76
10%	80	69.76	222	814.02	20,034.35	18.35	0.62	4.01	0.00	22.98
20%	80	140.60	420	2,077.11	24,327.59	75.96	1.38	11.45	0.00	88.79
30%	80	211.31	632	3,540.24	29,554.71	161.19	2.77	50.21	8.61	214.18
40%	80	280.51	818	4,951.05	35,266.04	233.93	4.73	70.75	0.00	309.40
50%	80	350.02	1,001	5,862.62	39,442.55	227.11	6.55	17.02	0.00	250.68
60%	80	351.00	1,001	4,456.75	29,639.79	126.87	4.03	10.46	0.00	141.36
70%	80	351.00	1,001	3,020.04	19,390.99	56.31	1.95	4.06	0.00	62.32
80%	80	351.00	1,001	1,643.91	10,220.38	16.17	0.73	1.37	0.00	18.28
90%	80	351.00	1,001	484.70	3,084.05	3.10	0.09	0.16	0.00	3.35

#inst. - number of instances; n - number of items; d - average number of dimensions; $\#v, \#a$ - average number of vertices and arcs in the final arc-flow graph; T^{pp} - average time spent building the graph; T^{lp} - average time spent in the linear relaxation of the root node; T^{bb} - average time spent in the branch-and-bound procedure; N^{bb} - average number of nodes explored in the branch-and-bound procedure; T^{tot} - average run time in seconds.

Sadykov and Vanderbeck (2013) also propose harder instances, most of which we were not able to solve in less than 12 hours. In order to solve these instances, they used branch-and-bound to solve the subproblems, instead of the dynamic programming algorithm that they used on the Fernandes-Muritiba et al. (2010)'s dataset. A possible reason for this is that the dynamic programming search space is difficult to represent in a compact form for these harder instances. A limitation of our method is the combination of large capacities and long patterns, which can be difficult to represent in a compact way. In practice, branch-and-price can overcome this problem by using a tailored algorithm in the sub-problem. This is not possible in our method; but when it generates reasonably small graphs, it usually outperforms more complex approaches, such as branch-and-price algorithms.

6.6. Other applications

In the online supplement of this paper, two additional applications through reductions to vector packing are presented, namely cutting stock with binary patterns and cutting stock with binary patterns and forbidden pairs. Moreover, the formulation and the graph compression algorithm are very flexible and can be applied to other problems that may not be directly reducible to vector packing, such as multiple-choice vector packing (see, e.g., Brandão and Pedroso 2013).

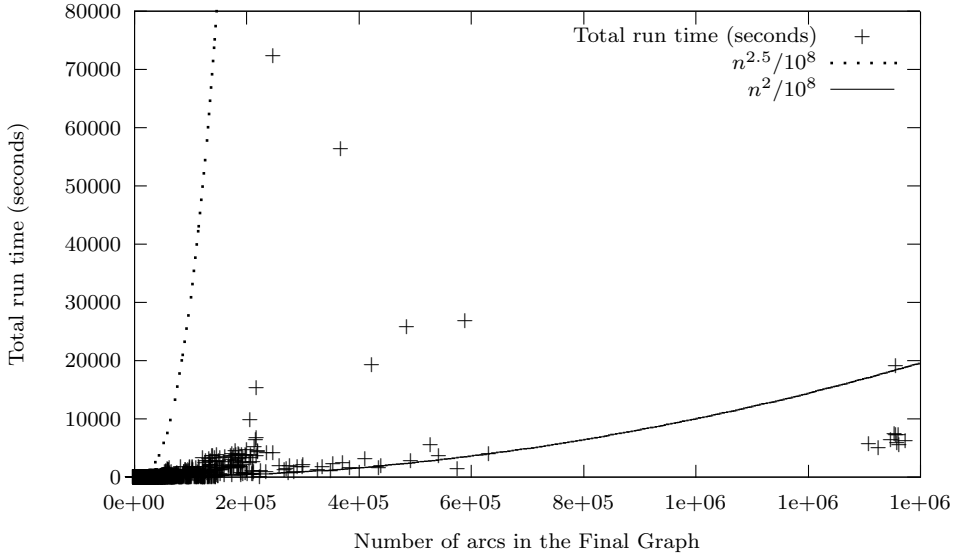
6.7. Run time analysis

Using the proposed method, we solved sequentially 23,153 benchmark instances in 9 days, spending 33 seconds per instance, on average. These benchmark instances belong to several different problems. The same method was used to tackle all the instances without any problem-specific adjustment. The linear relaxations are very strong in every problem we considered. The largest absolute gap (i.e., absolute difference between the optimal solution value and the value of the continuous relaxation of the model) we found in all the instances from benchmark test datasets was 1.0027.

For instances that could be solved by our method (all except 77 instances), Figure 9 shows the relation between the number of arcs in the final arc-flow graph and the total run time.

The two curves $n^2/10^8$ and $n^{2.5}/10^8$ show an approximation of the run time (in seconds) for algorithms with complexities $\Theta(n^2)$ and $\Theta(n^{2.5})$, with very low constant factors. The large majority of the observed run times for our method appear between these two curves. Many of them are very close to the quadratic run time, which is favorable since solving the arc-flow model is NP-hard. The few instances that lead to run times far from quadratic are mainly instances where the number of items that fit in each bin is large (e.g., more than 10) and hence the total number of patterns is huge. The arc-flow graphs for the 77 instances that were not solved in a reasonable amount of time had millions of arcs and the corresponding models are too large to be solved optimally using current state-of-the-art mixed integer programming solvers in a reasonable amount of time.

Figure 9: Run time analysis (Gurobi).



Very good results were also obtained using non-commercial MIP solvers such as `COIN-OR`[8]. Note that the non-commercial solvers are usually inferior to commercial solvers and hence they may not be able to solve the large models as easily as `Gurobi`[9].

[8] `http://www.coin-or.org/`
[9] `http://www.gurobi.com`

7. Conclusions

The method presented in this paper proved to be a very powerful tool for solving several cutting and packing problems. The model is equivalent to Gilmore and Gomory's, thus providing a very strong linear relaxation. Nevertheless, it replaces column-generation by the generation of a graph able to represent one permutation for each valid packing pattern. These are implicitly enumerated through the construction of a compressed graph, which is proven to hold all the paths from the source to the target that are required for determining the optimum solution of the original problem.

This method can be used for solving several problems through reductions to vector packing; examples include bin packing, cutting stock, and bin packing with conflicts. Without any problem-specific adjustment, we solved most of the known benchmark instances of these problems on a standard desktop computer, spending less than one minute per instance, on average. The linear relaxations are very strong for every problem we considered: the largest absolute gap we found in all the instances solved was 1.0027.

The proposed graph compression algorithm is simple and proved to be very effective on a large variety of problems. The major limitation of our method is the combination of large capacities and long patterns, which can be difficult to represent in a compact way. Nevertheless, when the graphs generated are reasonably small, the proposed method usually outperforms more complex approaches such as branch-and-price algorithms.

Acknowledgments

We would like to thank the three anonymous referees for their constructive comments, which led to a clearer presentation of the material.

F. Brandão thanks Fundação para a Ciência e Tecnologia (FCT), Portugal for the Ph.D. Grant SFRH/BD/91538/2012.

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows - theory, algorithms and applications*. Prentice-Hall.

Baum, S. and L. E. Trotter, J. (1981). Integer rounding for polymatroid and branching optimization problems. *SIAM Journal on Algebraic Discrete Methods*, 2(4):416–425.

Brandão, F. (2012). Bin Packing and Related Problems: Pattern-Based Approaches. Master's thesis, Faculdade de Ciências da Universidade do Porto, Portugal.

Brandão, F. and Pedroso, J. P. (2013). Multiple-choice Vector Bin Packing: Arc-flow Formulation with Graph Compression. Technical Report DCC-2013-13, Faculdade de Ciências da Universidade do Porto, Portugal.

Bron, C. and Kerbosch, J. (1973). Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577.

Caprara, A. and Toth, P. (2001). Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Appl. Math.*, 111:231–262.

Epstein, L. and van Stee, R. (2011). Improved Results for a Memory Allocation Problem. *Theor. Comp. Sys.*, 48(1):79–92.

Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30.

Fernandes-Muritiba, A. E., Iori, M., Malaguti, E., and Toth, P. (2010). Algorithms for the bin packing problem with conflicts. *INFORMS Journal on Computing*, 22(3):401–415.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Gau, T. and Wäscher, G. (1995). CUTGEN1: A problem generator for the standard one-dimensional cutting stock problem. *European Journal of Operational Research*, 84(3):572 – 579.

Gilmore, P. C. and Gomory, R. E. (1961). A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research*, 9:849–859.

Lodi, A., Martello, S., and Vigo, D. (1999). Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11(4):345–357.

Sadykov, R. and Vanderbeck, F. (2013). Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):244–255.

Schoenfield, J. (2002). Fast, Exact Solution of Open Bin Packing Problems without Linear Programming. draft, US Army Space & Missile Defence Command, Huntsville, 20 Alabama.

Scholl, A., Klein, R., and Jürgens, C. (1997). Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7):627 – 645.

Schwerin, P. and Wäscher, G. (1997). The Bin-Packing Problem: A Problem Generator and Some Numerical Experiments with FFD Packing and MTP. *International Transactions in Operational Research*, 4(5-6):377–389.

Valério de Carvalho, J. M. (1999). Exact solution of bin-packing problems using column generation and branch-and-bound. *Ann. Oper. Res.*, 86:629–659.

Valério de Carvalho, J. M. (2002). LP models for bin packing and cutting stock problems. *European Journal of Operational Research*, 141(2):253–273.

Wäscher, G. and Gau, T. (1996). Heuristics for the integer one-dimensional cutting stock problem: A computational study. *OR Spectrum*, 18:131–144.

Wäscher, G., Haußner, H., and Schumann, H. (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130.

Wolsey, L. A. (1977). Valid inequalities, covering problems and discrete dynamic programs. In P.L. Hammer, E.L. Johnson, B. K. and Nemhauser, G., editors, *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pages 527 – 538. Elsevier.