New collaborative approaches for bin-packing problems

François Clautiaux

Université des Sciences et Technologies de Lille

Laboratoire d'informatique fondamentale de Lille

Thèse d'habilitation à diriger des recherches

présentée par

François Clautiaux

spécialité : informatique

New collaborative approaches for bin-packing problems

Philippe Baptiste	Directeur de recherche, Ecole Polytechnique	(Rapporteur)
Jacques Carlier	Professeur, Université de Technologie de Compiègne	(Examinateur)
Martine Labbé	Professeur, Université Libre de Bruxelles	(Rapporteur)
Andrea Lodi	Professeur, Université de Bologne	(Rapporteur)
Aziz Moukrim	Professeur, Université de Technologie de Compiègne	(Examinateur)
El-Ghazali Talbi	Professeur, Université de Lille I	(Garant de l'HDR)
Sophie Tison	Professeur, Université de Lille I	(Examinateur)

Numéro d'ordre 40399| Année 2010

Acknowledgements

Je souhaite tout d'abord remercier Philippe Baptiste, Martine Labbé et Andrea Lodi d'avoir accepté d'être rapporteurs pour mon habilitation à diriger des recherches. Merci aussi à Sophie Tison d'avoir accepté de participer au jury.

Cette habilitation n'aurait pas de raison d'être sans les étudiants chercheurs que j'ai encadrés. Merci tout d'abord à Ali Khanafer. Sa thèse représente une partie importante de ce document. Par son travail acharné, il a placé la barre haute pour ses successeurs. Merci également à Nadia Dahmani et Ines Bahri, qui ont toutes deux commencé leur travail sous mon encadrement, ainsi qu'à Matthieu Gérard (un jour on le convaincra de s'inscrire en thèse).

Au cours de mes travaux de recherche, j'ai eu le privilège de collaborer avec d'excellents chercheurs. Je me permets de distinguer Antoine Jouglet, Saïd Hanafi, ainsi que Claudio Alves, avec qui j'ai le plus travaillé. J'ai beaucoup appris auprès d'eux, tout en appréciant leurs immenses qualités humaines. Merci aussi à Jacques Carlier et Aziz Moukrim. En commençant ma carrière de chercheur sous leur direction, je n'avais aucune excuse en cas d'échec.

Je remercie également Rita Macedo, Valério de Carvalho, Jürgen Rietz, Manuel Iori, Mauro Dell'Amico, Christophe Wilbaut et Raïd Mansi, pour les fructueuses collaborations que nous menons et mènerons ensemble.

En tant que responsable de l'équipe DOLPHIN, El-Ghazali Talbi a fait le nécessaire pour me mettre dans les conditions qui m'ont permis de présenter cette habilitation. Qu'il soit assuré de mon gratitude à son égard.

Un grand merci aux collègues de l'IUT A de Lille 1, qui auront été d'un grand soutien depuis le début. Merci également aux collègues de l'équipe DOLPHIN qui m'ont soutenu. Je remercie en particulier Nouredine Melab pour m'avoir fait partager son expérience et pour les bons moments passés en sa compagnie.

Merci surtout à Gaëlle pour son infinie patience à mon égard.

Contents

Introduction

In this document, we propose new models and resolution schemes for problems that belong to the family of *Cutting and Packing* (C&P) problems [90]. The most "basic" NP-complete problems in the C&P field are the **bin-packing** and the **knapsack** problems. In the former, the objective is to find the minimum number of bins needed to pack all the items.

Problem 1 (Bin-packing Problem (BPP)) *Given a set I of items i of size c_i, what is the minimum number of bins of size C needed to pack all the items of I?*

The BPP is the main problem addressed here, but different knapsack problems also appear as subproblems throughout the document. In this second problem, all items cannot be packed in the containers, and the objective is to maximize the profit associated with the input items.

Problem 2 (Knapsack Problem (KP)) *Given a set I of items i of size c_i and profit p_i, and a knapsack of size C, find the subset of I whose total size is smaller than C which maximizes the total profit of the selected items.*

The rather simple structure of the classical BPP has made this problem one of the most popular to test new methods, or to prove theoretical results. For example, its variant of *cutting-stock* has been one of the first problems to be solved through column generation methods [50, 51], and BPP is one of the favorite subjects of approximability studies (see for example [38]). These basic (yet hard) C&P problems can be seen as *"laboratories"* in which new techniques are tested. Therefore, many results first proposed for these problems lead to improvements for the resolution of many others.

For solving hard multi-dimensional packing problems efficiently, the literature shows that the best results are achieved using meta-heuristics, mathematical programming and constraint programming (CP). The first family of resolution method is effective for large size instances (see [72] for example), while CP can be more efficient than heuristics for problems involving one bin only (rectangle placement problems) [10]. For some particular cutting problems, where there are many instances of each item type, mathematical programming can even be faster than greedy algorithms. This justifies the fact that most of our algorithms use different resolution methods in a collaborative way to tackle packing problems.

In this document, we propose new models and methodologies that we apply to three families of packing problems. In Chapter 1, we study decomposition methods and meta-heuristics based on so-called strategic oscillation. We apply these techniques to packing problems with different kinds of conflicts. In Chapter 2, we deal with the concept of *dual-feasible functions*, which are used to derive polynomial-time lower bounds for several bin-packing problems, and improve cuts for integer linear programs. In Chapter 3, we propose new models for two different packing problems in two dimensions. We use these models into methods that use a combination of operations research and constraint programming techniques.

Throughout the document, we follow a consistent methodology. A first feature of our work is to use techniques from different fields: most notably integer programming, constraint programming, meta-heuristics, and graph theory. Although *"hybrid"* would be too strong a word, we always use these different methods in a collaborative way, which improves the results that would be obtained by each method separately.

Chapter 1 is devoted to different kinds of decomposition methods and so-called *strategic oscillation*. We used these methods to design lower and upper bounding strategies for packing problems with conflicts. In particular, we show that these decomposition methods can lead to effective collaborative resolution schemes. Our techniques rely on two types of decompositions: *Dantzig-Wolfe decomposition* [39] of linear programs, and *tree-decomposition* [83] of graphs. We first propose a resolution method for the *bin-packing problem with pairwise conflicts* [49], based on the decomposition of the conflict graph into several clusters. Each cluster can be solved independently if a partition based on the decomposition is computed. This framework is exploited by a tabu search, which assigns and remove items to/from clusters. The second problem addressed is a new bin-packing problem with conflicts met in a multi-objective context. The number of bins is limited, and we need to minimize the number of conflicts in the bins. For this problem, we propose a method based on linear-programming and column generation. Our method makes a good use of heuristic and meta-heuristic methods for generating the initial basis, and the columns at each step of the process. Finally, we address the bin-packing problem with fragile items [5], in which conflicts are modelled by a level of fragility for each item. We propose a column generation scheme for solving this problem. We designed a dynamic programming scheme for generating iteratively the columns, and a meta-heuristic method for initializing the master problem. The three meta-heuristics proposed in this chapter are based on the concept of strategic oscillation, in which the search oscillates from feasible to unfeasible, or complete to incomplete solutions.

Chapter 2 is clearly connected to mathematical programming and heuristics. It is dedicated to so-called *dual-feasible functions* (DFF) [75], related to column-generation techniques for the bin-packing problem and duality. These functions are used to derive lower bounds for packing problems, but they can also strengthen valid cuts in integer programs (see for example [1]). This chapter gives an comprehensive overview of the

concept of DFF, and hints based on experimentation to determine the problems for which it can be generalized. In the first part of this chapter, we describe the concept of DFF and show how it is related to other concepts in the literature. Then we survey the literature in which dual-feasible functions are used (sometimes implicitly, sometimes under a different name), and stress the link with *superadditive* functions used in integer programming. We show that a few different techniques are sufficient to generate most functions of the literature. The second part of Chapter 2 deals with extensions of this concept to various bin-packing problems (most notably two-dimensional problems, and the addition of conflict-based constraints).

Chapter 3 deals with an important issue in two-dimensional cutting/packing problems: rectangle placement problems. This problem has been the subject of a large number of contributions [9, 10, 43–45, 54, 67, 79, 82]. To our knowledge, methods based on constraint programming are the most efficient exact algorithms for these problems (see [10] for example). This can be explained by the fact that linear relaxations of the integer models dedicated to these problems are generally weak. We propose new models for two different rectangle placement problems: the regular case, and the guillotine case. We first show that the regular case is tightly linked with cumulative scheduling problems. This allows us to use powerful results from the scheduling field (energetic reasoning [41], branching schemes, etc.). The obtained method relies on an effective combination of operations research and CP techniques. We also propose the first effective graph-theoretical model for the guillotine case, which captures the combinatorial structure of the patterns, and helps designing a CP-based resolution method. For both problems, the efficiencies of our methods, which are able to compete with the best methods of a large literature, rely on the strength of the new models, but also on the new propagation and pruning algorithms.

Decomposition methods and strategic oscillation for bin-packing problems with conflicts

The work described in this chapter has been published in an international journal [65]. Two reports [31, 64] are also submitted to international journals.

1.1 Introduction

In this chapter, we describe our lower and upper bounds for various bin-packing problems with conflicts using decomposition methods and meta-heuristics based on strategic oscillation.

Handling conflicts is one of the first additional constraints to be demanded in industrial applications. Incompatibilities can be modelled in many ways. In this document, we address three variants: hard conflicts, soft conflicts, and fragilities. Given the difficulty of these problems, and the time that would be entailed by an exact resolution, we focus on heuristic and lower bounding methods.

Packing problems with conflicts are generally harder to solve than the classical bin packing problem (BP). Whereas a simple Integer Linear Programming (ILP) formulation can find good solutions for BP and in many cases good lower bounds in a small amount of time, this is generally not the case with conflicts. Since these problems are difficult, a sensible way of addressing them is to decompose them into subproblems that will be hopefully easier to solve. We will focus on two particular decomposition methods: tree-decomposition of graphs and Dantzig-Wolfe decomposition of ILP.

The methods that we describe in this chapter share some similarities. The first, as hinted above, is to rely on decomposition methods. The second is to generate solutions using meta-heuristics based on so-called strategic oscillation. The idea is to oscillate between two sets of solutions: complete/incomplete for the first problem, over-constrained/relaxed for the second, and feasible/non-feasible for the third. We used this oscillation strategies because, given a neighborhood, the conflicts may forbid to travel simply from one good solution to a close one in the solution space (because

non feasible or non complete solutions are encountered on the shortest path between them).

The first problem considered is the classical bin-packing problem with conflicts (BPC). In this problem, conflicts between two items are forbidden, and are modelled with a graph. We show how *tree-decomposition* can be used to solve this problem. Applying this decomposition to BPC is not straightforward, since finding a partition of the items based on the tree decomposition is a hard problem. We propose several heuristics to address this problem, and a tabu search based on a construction/destruction scheme where items are assigned to and de-assigned from clusters.

The second problem considered is a new problem, which we name *min-conflict packing problem* (MCBP). We met this problem in the context of multi-objective optimization. The conflicts are of the same type as the first problem, but this time, the number of bins is limited, and the objective is to minimize the number of violated conflicts. We apply Dantzig-Wolfe decomposition to MCBP. Two difficulties arise: generating a good initial basis and iteratively generating the columns in an efficient manner. For the initial basis, we designed a tabu-search based on oscillation between solutions using different numbers of bins. We proposed heuristics, a local search method and two ILP models to generate the columns iteratively. The former are improved by the means of cuts added to the models.

The third problem features a different variant of conflicts. Each item has a fragility, and the total size of the items in a bin cannot be larger than the smallest fragility of an item in the bin. We used a methodology similar to the previous problem (column generation). Initial columns are generated by a meta-heuristic, a Variable Neighborhood Search (VNS), based on an oscillation between feasible and unfeasible solutions. The subproblem is solved through a new dynamic programming scheme. It outperforms our two ILP models and it finds a solution within a small amount of time for all our instances.

1.2 A tree-decomposition based resolution scheme for the bin-packing with conflicts

In this section, we deal with the classical bin-packing problem with conflicts. The method we propose is generic and can be used for both one- and multi-dimensional cases of the problem (the geometric constraints are handled by sub-routines). In this document, we focus on the two-dimensional case.

Problem 3 (Two-dimensional Bin-packing Problem with Conflicts (BPC)) *Let $I = \{1, \ldots, n\}$ be a set of rectangular items i of width w_i and height h_i, a bin B of width W and height H, and $G = (I, E)$ a conflict graph. Two items i and j are in conflict if $(i, j) \in E$. What is the minimum number of bins needed to pack all items of*

I in bins of type B in such a way that two conflicting items are not packed in the same bin, and no two items overlap?

The problem is defined with a conflict graph. In the following, we mainly use the compatibility graph $\bar{G} = (I, I \times I \setminus E)$. Our decomposition method will be applied on this graph.

Several heuristics have been proposed for the one-dimensional version of BPC [46, 49]. The most effective are based on classical *any-fit* algorithms originally designed for BP, and on the search of cliques in the graph. The first-fit decreasing algorithm sort the items by decreasing size, and pack the items one by one in this order in the first bin that can accommodate it. For the two-dimensional case, the same approach can be used. It leads to a larger computing time since verifying that an item can be packed into a bin is more difficult in two dimensions. Practically speaking, we use the algorithm bottom-left of Coffman [37]. Other heuristics can also be used (see the methods described in [71, 73] for example). Unfortunately, in many cases, these algorithms do not lead to interesting results, since they do not take into account the structure of the graph. In the sequel, we show how a decomposition method can help such a algorithms to find a better solution.

Some methods dedicated to the one-dimensional case of BPC [49] rely on maximal cliques or stable sets in the graph. Finding a stable set in the compatibility graph gives a subset of items that have to be packed in different bins. On the contrary, finding a clique gives a subset of compatible items that can be packed together. This notion can be generalized using the concept of *tree-decomposition* applied to the compatibility graph.

In a tree-decomposition, the graph is decomposed into clusters of vertices connected in a tree. Each cluster corresponds with a subproblem to solve. A property of this decomposition method is that each clique of the graph is contained entirely in at least one cluster. Consequently, even if some conflicting items remain inside the clusters, the compatibility graph associated with each subproblem should be denser, and thus algorithms designed for the classical BP should be more effective when applied to the different clusters.

Once a decomposition is computed, our method solves the problem related to each cluster separately, and merges the solutions found. The most crucial issue is that a given item/vertex can belong to several clusters of the decomposition. In a first phase, we assign each item to a unique cluster (and thus this item is removed from the other clusters). We show that finding the best partition of the items into the clusters is NP-hard and we describe several heuristics to find good solutions.

Finally a tabu search based on our framework and strategic oscillation is proposed. The idea is to alternate construction and destruction phases in which items are respectively assigned and de-assigned from the clusters. Our methods are tested against instances derived from the literature. Our computational experiments show the effec-

tiveness of our approach.

1.2.1 Tree-decomposition and graph triangulation

We now define the notion of *tree-decomposition*, which will be applied to the compatibility graph of the BPC in the sequel.

A tree-decomposition is a special mapping of a graph into a set of clusters linked in a tree.

Definition 1.2.1 *(Robertson and Seymour [83]) A tree-decomposition of a given graph* $G = (V, E)$ *is a pair* $(\mathcal{C}, T)$ *where* $T = (\mathcal{N}, A)$ *is a tree with node set* $\mathcal{N}$ *and edge set* A*, and* $\mathcal{C} = \{\mathcal{C}_i : i \in \mathcal{N}\}$*, is a family of subsets of* V *such that:*

1. $\cup_{i \in \mathcal{N}} \mathcal{C}_i = V$,
2. $\forall (v, w) \in E$, $\exists \mathcal{C}_i \in C$ *containing both items* v *and* w,
3. $\forall i, j, k \in \mathcal{N}$, *if* j *is on the path from* i *to* k *in* T, *then* $\mathcal{C}_i \cap \mathcal{C}_k \subseteq \mathcal{C}_j$.

Figure 1.1 shows a graph G with eight vertices, and a tree decomposition of G onto a tree with six nodes. The set of clusters is $\mathcal{C} = \{\mathcal{C}_1 = \{0, 1\}, \mathcal{C}_2 = \{1, 2, 5, 6\}, \mathcal{C}_3 = \{2, 3, 6\}, \mathcal{C}_4 = \{3, 6, 7\}, \mathcal{C}_5 = \{3, 4\}\}$.

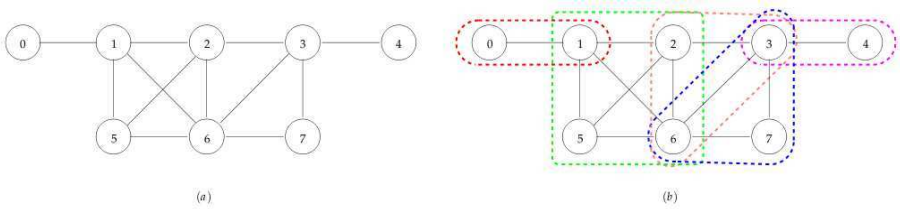


Figure 1.1: A graph G and a possible tree-decomposition for G

The *width* $w(\mathcal{C}, T)$ of a tree-decomposition is equal to $max_{i \in \mathcal{N}}(|\mathcal{C}_i| - 1)$. The *treewidth* $tw(G)$ of a graph G is defined as $min\{w(\mathcal{C}, T)\}$ where the minimum is taken over all tree-decompositions $(\mathcal{C}, T)$ of G. Whereas for some graph families, such as trees and series-parallel graphs, one can compute the treewidth in linear time, computing the treewidth of a general graph is a $\mathcal{NP}$-complete problem. Several papers are devoted to heuristics for this problem (see Koster *et al.* [68] or our paper [36]. The most famous (and simple) method is *Maximal Cardinality Search* (MCS) [85], which will be used in this document.

The notion of tree-decomposition is strongly connected with the class of *triangulated* graphs.

Definition 1.2.2 *A graph is* triangulated *if every cycle of length* > 3 *has a chord,* i.e. *an edge joining two non-consecutive vertices of a cycle.*

Tarjan and Yannakakis [85] showed that any triangulated graph contains at most n maximal cliques, and proposed an algorithm to enumerate these cliques in linear time. Computing a tree-decomposition for a graph is equivalent to finding a triangulation of this graph, *i.e.* finding a suitable set of edges to add to the graph to obtain a triangulated graph. Then, the clusters are obtained by enumerating in linear time the maximal cliques of the triangulated graph.

1.2.2 A general scheme for applying a tree-decomposition to the bin-packing problem with conflicts

In this section, we present our framework for applying tree-decomposition to the BPC. Once a tree-decomposition is obtained, which means that the set of clusters was identified, each item has to be assigned to a specific cluster to prevent items belonging to several clusters from being packed more than once. We call such an assignment a *cluster-separation* and show that finding the best cluster-separation is $\mathcal{NP}$-complete. Then we propose a first family of heuristics to find fast solutions for this problem.

Given a conflict graph $G = (I, E)$, let us denote by $\overline{G} = (I, I \times I \setminus E)$ the corresponding compatibility graph. Our method works as follows. The tree-decomposition is first applied to the compatibility graph $\overline{G}$. Each cluster is related to a set of items that induces a smaller and hopefully less dense subproblem than the original problem. Then each cluster is solved independently. If the density of the graph has been significantly decreased, algorithms dedicated to the classical bin-packing should be more effective. Finally, the partial solutions obtained are merged into a unique solution.

Now suppose the graph of Figure 1.1 is a graph of compatibility. We can notice that a vertex may belong to several clusters. For example, vertex 6 belongs to $\mathcal{C}_2$, $\mathcal{C}_3$ and $\mathcal{C}_4$. If the corresponding item is treated as many times as the vertex appears in a cluster, the solution obtained will be of weak quality.

Algorithm 1 shows a step-by-step description of the new approach. At line 1, the graph of compatibility is tree-decomposed. A cluster-separation is computed at line 2. The separated clusters are then solved as subproblems by the means of any resolution method at lines 4-5. Finally, at line 6, an improving heuristic is applied to the final solution. This last step is not mandatory, but it allows us to partially correct the effects of a bad cluster-separation.

When the partition of the item set is realized, we use a heuristic to solve the induced subproblem. We used an adaptation of the Bottom-Left algorithm [37].

Note that although only heuristics are used in this document, our framework allows exact methods to be used in each step of the algorithm (computing the decomposition, partitioning, packing, improving). This would lead to better results, but also to much more time-consuming methods.

Algorithm 1: A Tree-Decomposition based framework for solving BPC.

input : Set I of n items and $G = (I, E)$ graph of conflicts.
output: A Packing of the n items in a set B of bins.

1 $(\mathcal{C}, T) \longleftarrow TreeDecomposition(\overline{G})$;
2 $\mu(\mathcal{C}, T) \longleftarrow ClusterSeparation(\mathcal{C}, T)$;
3 $B \longleftarrow \emptyset$;
4 **foreach** $\mathcal{C}_i \in \mu(\mathcal{C}, T)$ **do**
5 $\quad B \longleftarrow B \cup ResolutionMethod(\mathcal{C}_i)$;
6 $B \longleftarrow ImprovingHeuristic(B)$;

1.2.3 The cluster-separation problem

An important issue in the new approach is to find a suitable partitioning of the items in the clusters. We call such a partitioning a *cluster-separation*.

Definition 1.2.3 *Given a BPC instance with a compatibility graph $\overline{G} = (I, \overline{E})$ and its tree-decomposition $(\mathcal{C}, T)$, a* cluster-separation *is a partition of the set of items I in the set of nodes C such that an item can be assigned to a node $\mathcal{C}_i$ only if it belongs to $\mathcal{C}_i$ in the tree-decomposition.*

A possible cluster-separation of the decomposition of Figure 1.1 is $\mathcal{C}_1 = \{0, 1\}, \mathcal{C}_2 = \{2, 5, 6\}, \mathcal{C}_3 = \emptyset, \mathcal{C}_4 = \{3, 7\}, \mathcal{C}_5 = \{4\}$.

The choice of the cluster-separation is the most crucial part of the algorithm. We say that a cluster separation is *compatible* with a given solution if, in this solution, two items assigned to two different clusters are never packed in the same bin. We state below that there always exists a cluster-separation that can lead to an optimal solution.

Proposition 1.2.1 *For any BPC instance D with a compatibility graph $\overline{G}$ and its tree-decomposition $(\mathcal{C}, T)$, there exists a cluster-separation μ of $(\mathcal{C}, T)$ that is compatible with an optimal solution for D.*

We call the problem of finding the best cluster-separation the *best-cluster-separation problem*. We now state that this problem is $\mathcal{NP}$-complete for an arbitrary graph by reducing the *partition problem* [48] to it.

Definition 1.2.4 *Let D be a BPC instance with a compatibility graph $\overline{G} = (I, E)$ and $(\mathcal{C}, T)$ its tree-decomposition and k an integer value. The best-cluster-separation problem consists in finding a cluster separation of $(\mathcal{C}, T)$ compatible with to a solution of value k, if it exists.*

Proposition 1.2.2 *The best-cluster-separation problem is $\mathcal{NP}$-complete.*

1.2.4 Greedy heuristics for the cluster separation problem

Since the best-cluster-separation problem is $\mathcal{NP}$-complete, a reasonable way of tackling this problem for dense graphs is to use heuristics. This phase is the most crucial of our approach since choosing a bad cluster-separation would lead to bad solutions.

The heuristics we propose belong to the family of greedy algorithms based on an initial sorting of the clusters. Consider a tree-decomposition $(\mathcal{C}, T)$. A cluster-separation can be computed as follows: 1) number the clusters according to a given ordering, and 2) for each cluster in the order chosen, assign all remaining items of the current cluster $\mathcal{C}_i$ to a new set S_i and remove these items from the subsequent clusters.

Several criteria have been proposed to explore the cluster tree associated to a tree-decomposition (see *e.g.* [61]). Two types of criteria were introduced in [61]: *local* and *global*. A local (resp. global) criterion evaluates the relevance of a candidate cluster without (resp. by) taking into account the interactions with other clusters. The authors also proposed two criteria, the *cluster size* (local) and the *cluster neighborhood size* (global) corresponding to the number of clusters connected to it.

In this document we introduce a new global criterion, the *demand* $\mathcal{D}(i)$ of an item i as to be the number of clusters that contain i. This criterion can be generalized and applied to clusters as follows: the demand of a cluster $\mathcal{C}_k$ is the sum of the demands of the items of $\mathcal{C}_k$: $\mathcal{D}(\mathcal{C}_k) = \sum_{i \in \mathcal{C}_k} \mathcal{D}(i)$. A cluster with a large demand shares many items with other clusters, and therefore this criterion identifies the *"central"* clusters of the decomposition. We also introduce another local criterion that we call *rand* corresponding to randomly sorting the clusters.

The choice of these simple heuristics may be justified by the fact that they do not entail a large computing time, since the use of any complicated heuristic for computing a cluster-separation would increase the computing time of algorithm 1.

1.2.5 A tabu search based on the tree decomposition and strategic oscillation

Local search algorithms are widely acknowledged as powerful tools for providing high-quality solutions to a wide variety of combinatorial problems. In the previous section, we have stated that the cluster separation problem was the core of our resolution approach. In order to improve the results obtained by the greedy heuristics, we designed a tabu search that focuses on the cluster separation phase.

In this section, we describe the tabu search algorithm, denoted as TS-TD in the following. Tabu search [52] has already been used to solve packing problems (see [56] for example), using a so-called oscillation strategy. We use this concept of oscillation by iteratively switching from construction to destruction phases.

1.2.5.1 Details of our tabu search

In our approach, **a solution** s is represented by a vector $\bar{v}$ of size n. Each element $\bar{v}_i$ of this vector records the current cluster to which item i is assigned. We denote by D_i the set of possible clusters that can accommodate i. For example, according to figure 1.1, the domain of item 3 is $D_3 = \{3, 4, 5\}$. The **solution space** of TS-TD is defined as the set of *complete* and *incomplete* solutions. A solution is said to be complete (resp. incomplete) when its variables are (resp. are not) all assigned. In our approach, any complete solution has to be feasible.

The **initialization phase** generates an initial non-complete solution vector by assigning each item i such that $|D_i| = 1$ (see Figure 1.2). The remaining items are set to -1 and will be assigned to clusters during the search.

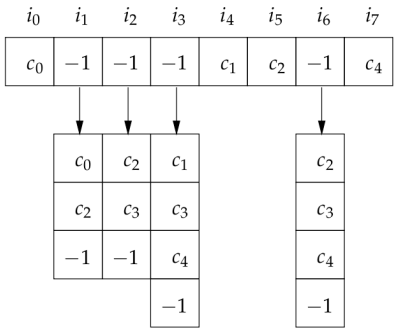


Figure 1.2: A vector representing a solution and its initialization according to figure 1.1.

A **move** is the assignment of a variable i to a value in its domain D_i (assign) or the value -1 (remove from the cluster). The existence of two types of movements is justified by the fact that our TS involves two phases, *construction* and *destruction*. The construction phase guides the TS toward a complete solution while the destruction phase desinstantiates some variables. Alternating these two phases plays the role of a diversification process in order to enable the TS to explore new regions of the search space.

Our **objective function** uses two terms. In a bin packing context, choosing solely the real objective function, which is to minimize the number of bins, is rather pointless, since many different solutions still in general have the same number of bins. It is often better to extend this coarser grained measure by the *gap* value, computed from the free area in each bin.

In our implementation, the **tabu list (TL)** is a set of moves classified tabu during some iterations (tabu tenure). The tabu tenure is a static value equal to the total number of possible moves. For example, the size of TL for the example of Figure 1.1 is equal to 14. Once TL is full, it will be resized in a such way that the oldest half is erased.

1.2.5.2 Strategic oscillation

Our **diversification** strategy consists in alternating the *construction* and *destruction* phases following some dynamic criteria based on the number of iterations. Figure 1.3 shows the behavior of our TS through the search process. We first start a construction phase. We keep running our local search until all items of s are assigned. A destruction phase is then applied on s by de-assigning some items in order to enable the next construction phase to explore new regions of the search space, and the procedure is repeated while no stopping criterion holds. This strategy may be controlled by two parameters: the *amplitude* a and the *frequency* f. The amplitude represents the maximal number of backward moves to be performed during a destruction phase. The value of the frequency is equal to the minimal number of complete solutions to reach during the search process.

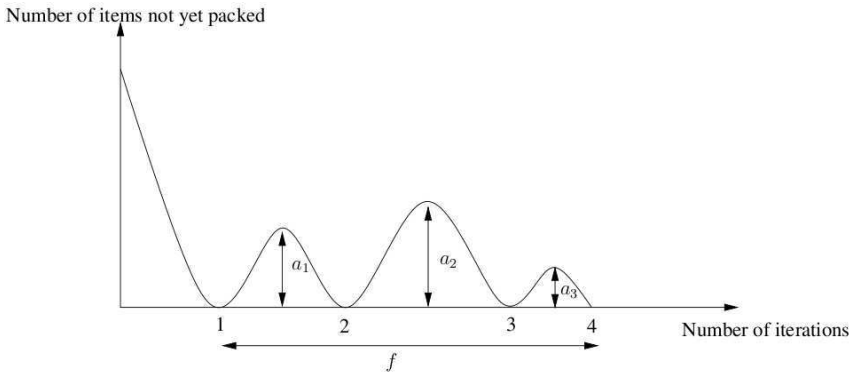


Figure 1.3: A search trajectory in the search space. The frequency f is the number of times a complete solution is obtained. The amplitude a represents the number of desinstatiations to perform in a destruction phase.

Our **intensification** phase consists in applying an *improving heuristic* on each complete solution we find during the search process. It is based on the progressive reduction of the number of bins used by the current solution. The idea is to destroy some bins and redistribute their contents to the remaining bins.

1.2.6 Time complexity of the method

For graphs of bounded treewidth, the time complexity of the algorithm is reduced compared to that of an equivalent construction heuristic.

The time complexity of computing a cluster-separation according to an ordering criterion depends on the number of items n, the number of clusters $|\mathcal{C}|$ in the tree-decomposition and the width $w(\mathcal{C}, T)$ of the tree-decomposition. If a local sorting algorithm is used, the time complexity of this phase is in $O(|\mathcal{C}| \times \log(w(\mathcal{C}, T)))$, which is in $O(n \times \log(n))$ for an arbitrary graph. For the global sorting strategies, the time complexity is $O(n^2)$, since an initial phase with this time complexity is needed to compute the size of the neighborhood of a cluster, or its demand.

For the two-dimensional case of BPC, the time complexity of BLC (a simple adaptation of the bottom-left heuristic) is $O(n^3)$. If the graph is such that there is an algorithm to find a tree-decomposition whose width is bounded by a constant, it would lead to a $O(1)$ time complexity in our framework (since the number of items in the subproblems would be a constant).

The time complexity of the improving heuristic is $O(|I^{\mathcal{B}}| \times n^2)$ where $I^{\mathcal{B}}$ is the number of items to repack. The time complexity is entailed by the maximum number of possible placements at any step of a packing. A possibility for reducing this time complexity is to consider a constant number of items in $I^{\mathcal{B}}$ and a number of open bins such that the number of packed items is also bounded by a small constant. In this case, the time complexity of this phase would also be $O(1)$. Unfortunately, experiments showed that it also dramatically weakens the effectiveness of the improving phase.

Let us summarize these results by studying the case of algorithm 1 applied using MCS to compute the tree-decomposition, a greedy algorithm based on a local criterion to compute the cluster-separation, BLC for solving the resulting clusters and the improving heuristic, the overall time complexity would be $O(m+n\times\log(n)+n\times w(\mathcal{C},T)^3)$.

If $w(\mathcal{C},T)$ is bounded by a constant, the time complexity becomes $O(m+n\times\log(n))$, whereas the original constructive algorithms are at least in $O(n^3)$. For huge instances, one can even avoid the sorting algorithm, which leads to a linear time complexity (although the quality of the solution obtained is expected to be weak).

1.2.7 Synthesis of the computational experiments

We tested our approach on the two-dimensional version of the BPC. The test cases were obtained from the classical two-dimensional bin-packing benchmarks. We generated conflict graph randomly following the method used in [46]. We used instances of size up to 120 items. We used the framework ParadisEO [16] to implement our tabu search.

For these instances, the tree-decomposition framework improves significantly the performance of the greedy algorithms with a slightly larger computing time. The tabu search outperforms the other methods, but needs a larger computing time.

We also generated huge instances (2000 items) to validate the effectiveness of the tree-decomposition. For this test, we used the greedy heuristics based on the tree-decomposition and we switched off the improvement heuristic that is used after the solutions are merged. For sparse conflict graphs, the method does not bring any improvement in term of computing time, and deteriorates the quality of the solution, since the width of the decomposition is close to the number of items. When the conflict graph is dense, the computing time is dramatically reduced, but there is a slight reduction in the quality of the method.

1.3 A column-generation algorithm for the min-conflict packing problem

In the context of a multi-objective bin-packing problem, we studied a subproblem that we name the *"min-conflict packing problem"*.

Problem 4 (Min-Conflict Packing Problem (MCPP)) *Given a set I of items, a bin type B, a value M and a conflict graph $G = (I, E)$ where $(i, j) \in E$ if i and j are not supposed to be packed in the same bin, compute the minimum number of conflicts that must occur if the set I is packed in M bins of type B.*

This problem is justified by the fact that resources (bins) are not always in infinite number, and that the decision maker may favor a solution violating some constraints if its cost is small. The work we present is inserted in a general multi-objective scheme, and we use the knowledge of solutions related to different numbers of bins in our resolution method.

We first propose a compact formulation for this problem. This model is weak and thus we propose a reformulation based on a column-generation scheme. The model itself is an adaptation of the set-covering model of Gilmore and Gomory [50, 51]. The difficulty arises in the pricing subproblem, a bilinear knapsack problem, which is harder to solve than the classical knapsack problem. We reformulate this problem with two different ILP models. These models demand too much time, so we designed a greedy heuristic and a local search method based on swaps in order to find good solutions in a faster manner. In our method, the ILP solver is run only if the heuristics were not able to find a column of reduced cost.

A tabu search is also proposed to generate a good initial basis for the LP. Like in the previous section, it is based on strategic oscillation. For this method, we do not oscillate from complete to incomplete, but from a number of bins to another. This is justified by the fact that our method is run in the context of multi-objective optimization, and thus we have recorded some good solutions related to different numbers of bins. When the number of bins is increased, we will explore non-feasible solutions for our present problem. When the number of bins is decreased, it leads to valid yet more constrained solution that will make a better usage of the space in the bins (disregarding the number of conflicts entailed).

1.3.1 A simple compact model

A simple linearization of a direct quadratic model for the min-conflict problem leads to the following ILP model. Variables x_{ik} are equal to 1 if i is packed in bin k, and 0 otherwise. Variables y_{ij} are equal to 1 when items i and j are in the same bin. For an item i, let $N(i)$ be the set of items j such that i and j are in conflict. Let also $N^+(i)$

be the items j of $N(i)$ such that $j > i$. This set is used to avoid counting a conflict twice in our models.

$$\min \sum_{i \in I} \sum_{j \in N^+(i)} y_{ij} \tag{1.1}$$

$$\sum_{k \in M} x_{ik} \geq 1, i \in I \tag{1.2}$$

$$\sum_{i=1}^{n} c_i x_{ik} \leq C, k \in M \tag{1.3}$$

$$y_{ij} \geq x_{ik} + x_{jk} - 1, i \in I, j \in N^+(i), k \in M \tag{1.4}$$

$$y_{ij} \in \{0,1\}, i \in I, j \in N^+(i) \tag{1.5}$$

$$x_{ik} \in \{0,1\}, i \in I, k \in M \tag{1.6}$$

Constraints (1.2) ensure that all items are packed, whereas constraints (1.3) verify the capacity constraint. Constraints (1.4) are sufficient to ensure that if i and j are packed together, then y_{ij} will be equal to one.

1.3.2 A set covering formulation

The linear relaxation of the ILP above is weak. Thus we used a formulation based on a set covering model and the decomposition of Dantzig-Wolfe [39]. The ILP is decomposed into a restricted master problem initialized with a set of columns, and optimized to determine the value of the dual variables. The dual information is passed to a subproblem that evaluates if there is a column that can be added to the master problem and improve the current solution. If there is such a column, the master problem is reoptimized, otherwise the process stops.

Let P be the set of possible *patterns*, *i.e.* the set of possible ways of packing items in a bin. Each possible pattern is described by a column $p = (a_{1p}, \ldots, a_{ip}, \ldots, a_{|I|p})^T$, where a_{ip} is equal to 1 if item i is in the pattern p, 0 otherwise. A new variable K_p is introduced, which corresponds with the number of conflicts in configuration p. The decomposition is a simple adaptation of the model of Gilmore and Gomory [50, 51] dedicated to the cutting-stock problem.

$$\min \sum_{p \in P} x_p K_p \tag{1.7}$$

$$s.t. \sum_{p \in P} a_{ip} x_p \geq 1, \forall i \in I \tag{1.8}$$

$$\sum_{p \in P} x_p \leq M \tag{1.9}$$

$$x_p \in \{0,1\}, \forall p \in P \tag{1.10}$$

Two families of constraints are kept in the master: constraints (1.8) ensure that all items are cut, whereas (1.9) verifies that the number of bins used is smaller than M. The capacity constraint is left to the subproblem.

The weak dual of (1.7)-(1.10) reads as follows.

$$\max \sum_{i \in I} \pi_i - \theta M \tag{1.11}$$

$$s.t. \sum_{i \in I} a_{ip} \pi_i - \theta \leq K_p, \forall p \in P \tag{1.12}$$

$$\pi_i \geq 0, \forall i \in I \tag{1.13}$$

$$\theta \geq 0 \tag{1.14}$$

Dual variables π are related to the demand constraints, whereas the dual variable θ is related to the number of bins.

1.3.3 Solving the pricing subproblem using local search and linear programming

In order to generate the best column to add to the current basis, the pattern to add is the one with the smallest reduced cost, *i.e.* the pattern p for which $\theta + K_p - \sum_{i \in I} \pi_i a_{ip}$ is minimized. The only constraint for a pattern p is the capacity constraint of the bin. This is equivalent to solving the following bilinear problem, called *pricing subproblem.*

$$\max \sum_{i \in I} (\pi_i a_i - \sum_{j \in N^+(i)} a_i a_j) - \theta \tag{1.15}$$

$$s.t. \sum_{i \in I} a_i c_i \leq C \tag{1.16}$$

$$a_i \in \{0, 1\}, \forall i \in I \tag{1.17}$$

It is a generalization of the knapsack problem, where each conflict between two items reduces the value of the solution by one. The knapsack problem with hard conflicts has been studied by, among others, Hifi and Michrafy [60]. In our problem, conflicts may occur, but they are penalized in the objective function.

Compared to the cutting-stock problem, the pricing subproblem is more difficult to solve. It transpired from our first experiments that solving the pricing to optimality at each step of the column generation algorithm leads to a large computing time. Thus we have developed a method that uses linear-programming, heuristics and local search.

1.3.3.1 Two ILP models for the pricing subproblem

A first way of solving the pricing subproblem is use a straightforward linearized version of (1.15)-(1.17). For this purpose, we introduce variables b_{ij} that are equal to 1 if $a_i = 1$

and $a_j = 1$, and to 0 otherwise.

$$\max \sum_{i \in I} (\pi_i a_i - \sum_{j \in N^+(i)} b_{ij}) - \theta \tag{1.18}$$

$$s.t. \sum_{i \in I} a_i c_i \leq C \tag{1.19}$$

$$b_{ij} \geq a_i + a_j - 1, \forall i \in I, j \in N^+(i) \tag{1.20}$$

$$b_{ij} \in \{0,1\}, \forall i \in I, j \in N^+(i) \tag{1.21}$$

$$a_i \in \{0,1\}, \forall i \in I \tag{1.22}$$

A better model can be proposed, using a different type of variables b_i for each item type i. Each variable b_i is equal to the number of items of $N^+(i)$ that are packed with i. Note that $a_i = 0$ implies $b_i = 0$. In this model the number of variables remains linear, whereas it is quadratic in the previous model.

$$\max \quad \sum_{i \in I} (\pi_i a_i - b_i) - \theta \tag{1.23}$$

$$s.t. \quad \sum_{i \in I} a_i c_i \leq C \tag{1.24}$$

$$\sum_{j \in N^+(i)} a_j - |N^+(i)| \times (1 - a_i) \leq b_i \quad \forall i \in I \tag{1.25}$$

$$a_i \in \{0,1\}, \quad \forall i \in I \tag{1.26}$$

$$b_i \in \mathbb{N}, \quad \forall i \in I \tag{1.27}$$

Both models can be improved by the means of cuts. We describe these cuts using the formalism of model (1.23)-(1.27). Since one of the constraints is a classical knapsack constraint, all techniques related to this constraint can be used. For example, if $N_{\max}$ is the maximum number of items that can be packed side by side, the following cut is valid.

$$\sum_{i \in I} a_i \leq N_{\max} \tag{1.28}$$

Other hand-tailored techniques can be used for this specific problem. **Note that these cuts are valid because we seek a solution of positive value.** In other words, these cuts may exclude the optimal solution if this solution has not a positive value.

First, note that if $b_i \geq \pi_i$, a better solution is obtained by removing item i (because it leads to more conflicts than its value of profit). This leads to the following family of cuts.

$$b_i \leq \pi_i - 1, \forall i \in I \tag{1.29}$$

This family of cuts can be generalized by considering sets of items. Let I_f be a set of items such that $\sum_{i \in I_f} \pi_i - \sum_{i \in I_f} |N(i) \cap I_f| \leq 0$. Clearly, this set cannot belong to

an optimal solution. Consequently, if we are able to detect such a set I_f, we add the following cut.

$$\sum_{i \in I_f} a_i \leq |I_f| - 1 \tag{1.30}$$

Another cut can be added. First we compute the minimum number $N_{\min}$ of items needed to obtain a sum of profit greater than or equal to θ.

$$\sum_{i \in I} a_i \geq N_{\min} \tag{1.31}$$

An estimation of $N_{\min}$ can be computed by sorting the items by decreasing value of π_i and disregarding the conflicts.

1.3.3.2 Heuristic solutions for the subproblem

Even with the cuts above, and even if the search is stopped as soon as a solution of positive cost is found, our two ILP models do not lead to fast solutions in many cases. Consequently, we developed heuristics and a local search method to fasten the column generation process.

The greedy heuristics are based on an initial ordering on the items : decreasing π_i/c_i, decreasing $(\pi_i + |N(i)|)/c_i$, decreasing $(\pi_i + \sum_{j \in N(i)} \pi_j)/c_i$ We compute the values of degrees in a dynamic fashion (*i.e.* they are updated each time an item is selected or rejected from the knapsack).

In the case where the greedy heuristics are not able to find a solution with negative reduced cost, a local search phase is applied. It is based on two simple operators: bit flip (select an unselected item or remove a selected item), and pairwise exchange (replace an item by another item). At each step of the local search algorithm, all possible bit-flip moves are tested. If none improves the value of the objective function, the swap moves are tested. The method stops when no improvement has been realized in the last iteration.

1.3.4 Computing the initial set of columns using tabu search and strategic oscillation

Since the pricing phase has a large computational cost, finding a good initial basis is crucial for our algorithm. Like in Section 1.2, we use a strategic oscillation in a tabu search. This time, the oscillation uses the fact that the search is run in the context of multi-objective optimization. The tabu search iterates from a number of bins to another, leading alternatively to unfeasible and over-constrained solutions.

1.3.4.1 Details of our tabu search

Given a set I of n items, a solution is a number of bin associated with each item of I. A solution is a complete packing in which the capacity constraint is not violated in any bin (the sum of the sizes of the items is not larger than the size of the bin). The number of bins is initially fixed to M.

The **search space** S is thus composed of all possible configurations meeting this constraint. A **neighbor** of a solution s is obtained by changing the bin associated to an item in such a way that the capacity constraint remains satisfied. A soft conflict (i, j) is said to be violated if i and j are packed in the same bin. Our **objective function** is the number of violated soft conflicts. When an item is moved from a bin to another, the number of conflicts is updated only for the two bins involved in the move.

We used three different families of heuristic algorithms for the **initialization phase** of the tabu search. The first is the family of *Any-Fit* algorithms. The second is based on *Soft Graph Coloring*, and the third on a simple *local search algorithm* that tries to improve iteratively an initial solution.

Our **tabu list** (TL) consists of a set of moves classified tabu during some iterations (tabu tenure). The tabu tenure is a static value. Once TL is full, it is resized in a such way that the oldest half is erased.

1.3.4.2 Strategic oscillation

We developed a **diversification strategy** based on two types of modifications (*top-down* or *bottom-up*). Let m be the number of bins used in the current solution s_m. In a top-down (resp. bottom-up) strategy, a solution s_m is replaced by a solution s_{m+d} (resp. s_{m-d}) where d is a possible value for the number of bins. These diversification strategies are controlled by a parameter called *distance* d. The diversification distance represents the number of bins to add or remove from the current solution depending on the chosen strategy.

In case of a top-down strategy, d bins are added to a solution s_m and thus a solution s_{m+d} is obtained by distributing the contents of m bins into $m + d$ bins by means of a classical bin packing heuristic. In case of a bottom-up strategy, d bins are removed from the solutions, and the items in these d bins are distributed among the remaining bins. The d bins are chosen following a given criterion, like the bins with the maximum number of conflicts. Consequently, our method oscillates between increase phases, in which it adds bins in order to reduce the number of conflicts, and decrease phases, in which bins are erased, and the algorithms focuses on the space used in the bins.

Figure 1.4 illustrates these two diversification strategies. The blue (resp. red) arrow shows a top-down (bottom-up) diversification of a solution s_l (resp. s_m) with a distance d_l (resp. d_m).

In the context of multi-objective optimization, solution found at the previous steps can be used to help diversifying effectively the search.

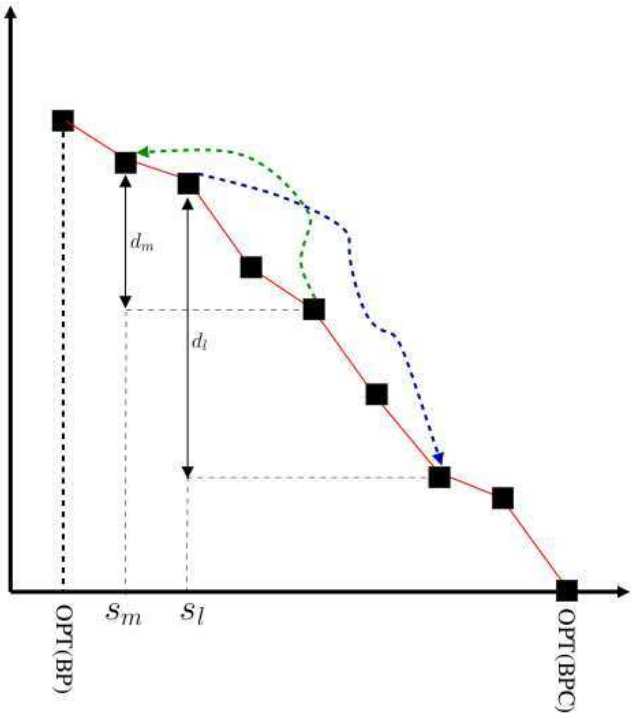


Figure 1.4: Example illustrating the two diversification strategies "top-down" and "bottom-up".

1.3.5 Synthesis of the computational experiments

We implemented the methods described above and tested them against benchmarks derived from the literature. We used instances with up to 120 items, for which we generated bounds for the whole Pareto front (more than 40 min conflict problems to solve in the worst case).

We were able to solve a large number of instances to optimality just by computing our bounds. However, our computational experiments confirm the fact that the pricing subproblem to solve is much harder than the classical knapsack problem, even using our second ILP model. Even with this faster model, generating the columns can take time. The best results were obtained using the heuristics and the local search, and then the second ILP model. We improved the results by stopping the ILP as soon as it gets an improving solution. Finding better algorithms for the pricing subproblem seems to be the main issue for improving the method. An efficient branch-and-price method cannot be designed before we are able to improve the pricing phase.

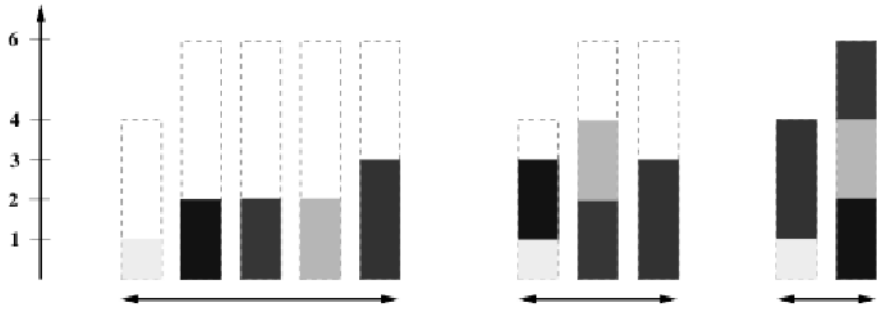


Figure 1.5: An instance of BPP-FO, a non-optimal solution with three bins and an optimal solution with two bins. Fragilities are represented by dotted rectangles, and sizes with grey rectangles.

1.4 A column generation algorithm for the bin-packing with fragile objects

In this section, we deal with a variant of packing in which conflicts are modelled in a different way. The problem is known in the literature as the *Bin Packing Problem with Fragile Objects* (BPP-FO). The BPP-FO arises in the telecommunication field and in particular in the allocation of cellular users to frequency channels (see Bansal et al. [5] and Chan et al. [24]).

Problem 5 (Bin-packing Problem with Fragile Objects (BPP-FO)) *Given a set I of items i of size c_i and fragility ψ_i, what is the minimum number of bins needed to pack all the items of I in such a way that in each bin, the sum of the sizes is smaller than the smallest fragility?*

An example of BPP-FO instance, and two solutions is given in Figure 1.5. More formally, let us denote $I(k)$ as the set of items assigned to a bin k, we need to ensure that

$$\sum_{i \in I(k)} c_i \leq \min_{i \in I(k)} \{\psi_i\} \tag{1.32}$$

for all possible bins k.

The literature on the BPP-FO is still small. Bansal et al. [5] present approximation schemes and probabilistic analysis. They consider approximations both with respect to the number of bins and to the fragility of a bin. They present results for the general BPP-FO and for a special case, denoted the *frequency allocation problem*, in which weight and fragility are strictly correlated one to the other. Chan et al. [24] consider instead the on-line version of the BPP-FO, in which an item arrives only after the previous item has been packed and the decision cannot be changed. They study the cases in which the ratio between the maximum and the minimum fragility is bounded

or unbounded, and present, for both cases, algorithms with asymptotic competitive ratios.

For the BPP-FO, we first propose simple compact models, which are able to solve exactly many random instances in a fast manner. However, for some classes of hard instances, these models are not sufficient anymore. For this reason, we propose a reformulation of the problem using the Dantzig-Wolfe decomposition and column generation. The pricing subproblem to solve is a *knapsack problem with fragile objects* described above.

Problem 6 (Knapsack Problem with Fragile Objects (KP01-FO)) *Given n items i with profit p_i, weight c_i and fragility ψ_i $(i = 1, \ldots, n)$ and a single uncapacitated bin, find the subset of items of largest total profit whose total weight is not larger than the fragility of any item in the bin.*

Being able to solve this problem efficiently is the most crucial issue in our column-generation algorithm. We propose two ILP models and a dynamic-programming scheme for KP01-FO. As we will see below, the dynamic-programming scheme outperforms both ILP models for all the instances we used.

For generating a suitable initial basis, we designed a variable-neighborhood search (VNS) method based on strategic oscillation. We oscillate from feasible to non-feasible solutions using different perturbation strategies. Our overall algorithm allows to find tight lower and upper bounds in a fast manner and close the integrality gap for many difficult instances that were not solved by the compact models within the allowed time limit.

1.4.1 Compact models

We first present two compact formulations for BPP-FO requiring a polynomial number of variables and constraints. We then discuss a third formulation using an exponential number of variables.

1.4.1.1 A simple formulation

Let us define $\psi_{\max} = \max_{i=1,\ldots,n}\{\psi_i\}$ to be the maximum fragility of an item. We define y_k as a binary variable taking value 1 if bin k is used, 0 otherwise $(k = 1, \ldots, n)$. We also define x_{ik} as a binary variable taking value 1 if item i is assigned to bin k, 0 otherwise $(i, k = 1, \ldots, n)$. The BPP-FO can be modeled as the following ILP:

$$\min \sum_{k=1}^{n} y_k \tag{1.33}$$

$$\sum_{k=1}^{n} x_{ik} = 1 \quad i = 1, \ldots, n \tag{1.34}$$

$$\sum_{j=1}^{n} c_j x_{jk} \leq \psi_{\max} + x_{ik}(\psi_i - \psi_{\max}) \quad i, k = 1, \ldots, n \tag{1.35}$$

$$x_{ik} \leq y_k \quad i, k = 1, \ldots, n \tag{1.36}$$

$$y_k \in \{0, 1\} \quad k = 1, \ldots, n \tag{1.37}$$

$$x_{ik} \in \{0, 1\} \quad i, k = 1, \ldots, n. \tag{1.38}$$

Constraints (1.34) impose that each item is assigned to a bin. Constraints (1.35) require that the sum of the weights of the items packed in a bin does not exceed the fragility of any item packed in the same bin (if item i is packed in bin k the right hand side of the constraint is equal to ψ_i, otherwise the constraint is redundant). Constraints (1.36) are used to tighten the model linear relaxation.

Model (1.33)–(1.38) is derived from the classical BPP compact model. It has a clear disadvantage that comes from the $O(n^2)$ Constraints (1.35), that model the non-linear restriction (1.32) by using a large value ($\psi_{\max}$). This value can worsen consistently the model linear relaxation, and makes the formulation very dependent from the fragility of the last item.

1.4.1.2 A better formulation

We recall that the items are sorted by non-decreasing values of ψ_i, breaking ties by non-increasing values of c_i. We define y_i as a binary variable taking value 1 *if item i is the item with smallest fragility in the bin in which it is packed*, 0 otherwise ($i = 1, \ldots, n$). We also define x_{ji} as a binary variable taking value 1 *if item j is assigned to the bin having item i as item with smallest fragility* (bin i for short in the following), 0 otherwise ($i = 1, \ldots, n, j = i + 1, \ldots, n$). The BPP-FO can be modeled as the following ILP:

$$\min \sum_{i=1}^{n} y_i \tag{1.39}$$

$$y_i + \sum_{j=1}^{i-1} x_{ij} = 1 \qquad i = 1, \ldots, n \tag{1.40}$$

$$\sum_{j=i+1}^{n} c_j x_{ji} \leq (\psi_i - c_i) y_i \qquad i = 1, \ldots, n \tag{1.41}$$

$$x_{ji} \leq y_i \quad i = 1, \ldots, n, j = i+1, \ldots, n \tag{1.42}$$

$$y_i \in \{0, 1\} \qquad i = 1, \ldots, n \tag{1.43}$$

$$x_{ji} \in \{0, 1\} \quad i = 1, \ldots, n, j = i+1, \ldots, n. \tag{1.44}$$

Constraints (1.40) impose that either an item is the smallest item in its bin, either it is assigned to a bin containing an item with smaller fragility. Constraints (1.41) require that the sum of the weights of the items packed in a bin does not exceed the smallest fragility in the bin. Constraints (1.42) are again used to tighten the model linear relaxation.

1.4.2 A set covering formulation

We present a model that builds upon the classical decomposition method by Gilmore and Gomory [50, 51]. We define a *pattern* as a feasible combination of items. We describe the pattern, say p, by a column $(a_{1p}, \ldots, a_{ip}, \ldots, a_{np})^T$, where a_{ip} takes value 1 if item i is in pattern p, 0 otherwise. Let P be the set of all valid patterns, i.e., the set of patterns p for which $\sum_{i=1}^{n} c_i a_{ip} \leq \min_{i=1,\ldots,n}\{\psi_i a_{ip}\}$.

Let also z_p be a binary variable taking value 1 if pattern p is used, 0 otherwise ($p \in P$). The BPP-FO can be modeled as the following Set Covering problem:

$$\min \sum_{p \in P} z_p \tag{1.45}$$

$$\sum_{p \in P} a_{ip} z_p \geq 1 \quad i = 1, \ldots, n \tag{1.46}$$

$$z_p \in \{0, 1\} \qquad \forall p \in P. \tag{1.47}$$

Constraints (1.46) impose that each item j is packed in at least one bin.

As the number of possible patterns may be very large, even solving the linear relaxation of Model (1.45)–(1.47) may be difficult. We approach this problem, as it is usually done in the literature, by means of a column generation method.

We initialize the model by a subset $\widetilde{P} \subseteq P$ of patterns. We then drop the integrality requirements (1.47) by replacing them with $z_p \geq 0$, $\forall p \in \widetilde{P}$. Note that we are allowed

to drop the conditions $z_p \leq 1$ since redundant (indeed we can always replace a solution in which there exists a $z_p > 1$ with a better one having $z_p = 1$). We finally associate dual variables π_i $(i = 1, \ldots, n)$ to constraints (1.46).

We operate in an iterative way. We solve the linear model just outlined and check if a pattern (i.e., a column) with negative reduced cost exists. If it exists, then we add it to the model and re-iterate, otherwise we proved the optimality of the (eventually fractional) solution obtained. The reduced cost of a pattern p is defined by

$$\overline{c}_p = 1 - \sum_{i=1}^{n} \pi_i a_{ip}. \tag{1.48}$$

A pattern is added to the model if it satisfies the fragility requirement and has a negative reduced cost. The existence of such pattern can thus be determined by solving a KP01-FO with objective function

$$\max \sum_{i=1}^{n} \pi_i a_{ip} \tag{1.49}$$

and subject to

$$\sum_{i=1}^{n} c_i a_{ip} \leq \min_{i=1,\ldots,n} \{\psi_i a_{ip}\} \tag{1.50}$$

$$a_{ip} \in \{0, 1\}, \forall i \in I \tag{1.51}$$

Note that one of the main issues in column generation is a long tail convergence, during which the value of the optimum is only marginally improved. Several methods have been proposed to deal with this issue. One of the most promising is the notion of *dual cuts* introduced by Valerio de Carvalho [86]. The idea is to add dual cuts to the master problem (columns in the primal) to exclude dual solutions that are dominated by others. This notion can be extended to the BPP-FO as follows.

Proposition 1.4.1 *For a given item i, if there exists a set $S \subset I$ such that $\sum_{j \in S} c_j \leq c_i$ and $\min_{j \in S} \psi_j \geq \psi_i$ then*

$$\sum_{j \in S} \pi_j \leq \pi_i \tag{1.52}$$

is a valid dual cut for (1.45)-(1.47).

Many cuts of this type can be applied. Practically speaking, we use the of Type I and II described in [86]. In the cuts of type I, the set S above is of size one. The cuts of type II consider two items in S. If the number of cuts of type II is too large, only a subset of them can be applied. Practically speaking, for the BPP-FO, this size is not so large, since the conditions for S is more restrictive than for the cutting-stock. Consequently, we applied all cuts of type II.

1.4.3 Solving the Pricing Subproblem using dynamic programming and linear programming

We solve the KP01-FO problem arising in the pricing phase by means of two ILP models and a dynamic programming algorithm. Insights in the computational results we obtained by using these three approaches will be given in Section 1.4.5 below.

ILP Models

We presented two compact ILP models to solve the BPP-FO, the latter (Model (1.39)–(1.44)) being better than the former (Model (1.33)–(1.38)). A similar result can be obtained for the KP01-FO.

A first ILP model can be obtained by using a binary variable x_i taking value 1 if item i is packed in the knapsack, 0 otherwise ($i = 1, \ldots, n$). The KP01-FO can be modeled as:

$$\max \sum_{i=1}^{n} \pi_i x_i \tag{1.53}$$

$$\sum_{j=1}^{n} c_j x_j \leq \psi_{\max} + x_i(\psi_i - \psi_{\max}), \quad \forall i \in I \tag{1.54}$$

$$x_i \in \{0, 1\} \quad \forall i \in I \tag{1.55}$$

The main difference with respect to the classical compact model for the KP01 is that here we need n constraints to impose the maximum capacity. Indeed, Constraints (1.54) impose that whenever an item i is packed in the knapsack, then the sum of the weights of all the items packed cannot exceed the value ψ_i.

A better ILP model can be obtained as follows. First, let us recall that items are sorted by non-decreasing values of ψ_i, breaking ties by non-increasing values of c_i. We define y_i as a binary variable taking value 1 *if item i is the item with smallest fragility in the knapsack*, 0 otherwise ($i = 1, \ldots, n$). We also define x_i as a binary variable taking value 1 *if item i is packed in the knapsack, but it is not the item with smallest fragility* ($i = 1, \ldots, n$). We obtain:

$$\max \sum_{i=1}^{n} \pi_i (x_i + y_i) \tag{1.56}$$

$$\sum_{i=1}^{n} y_i = 1 \tag{1.57}$$

$$\sum_{i=1}^{n} c_i x_i \leq \sum_{i=1}^{n} y_i (\psi_i - c_i) \tag{1.58}$$

$$x_i + \sum_{j=k}^{n} y_j \leq 1, \forall i \in I \tag{1.59}$$

$$x_i \in \{0, 1\}, \forall i \in I \tag{1.60}$$

$$y_i \in \{0, 1\}, \forall i \in I \tag{1.61}$$

Constraint (1.57) imposes the existence of just one item with smallest fragility inside the knapsack. Constraints (1.58) impose that the sum of the weights of the items in the bin does not exceed the fragility of the smallest item in the bin.

By the definition of the two sets of variables it follows that $x_i + y_i \leq 1$, for $i = 1, \ldots, n$. This simple consideration is generalized by constraints (1.59), that impose that if an item i is packed in the bin but is not the item with smallest fragility (i.e., if $x_i = 1$), then no other item having larger fragility can be the item with smallest fragility in the bin.

Dynamic programming

The KP01-FO can also be solved through dynamic programming. A trivial (but non efficient) method is to consider all possible values of fragility ψ_i in turn and each time solve the related KP01 with the set of items $\{i : c_i \leq \psi_i\}$ and a bin of size ψ_i. With this approach, in the worst case, the dynamic programming method is run n times (once for each different value of fragility), leading to a complexity of $O(n^2 * \psi_{\max})$.

A more efficient method can be used if the items are sorted by **decreasing** fragility. The algorithm is the following: construct the regular dynamic programming table related to a knapsack of size $\psi_{\max}$, and take the maximum value obtained among a set of *dominant* states. The validity of this method is based on the following proposition.

Note that the item with the largest fragility can be packed in a bin of any fragility. This can be generalized as follows: the k first items can be packed in a bin of fragility f_k. This can be repeated for any value of k. This means that the classical dynamic programming scheme can be used if states related to unreachable states are not considered.

Let $\psi(i, \alpha)$ be a dynamic programming state for KP01 related to the i first items and a knapsack of size at most α. To simplify the notation, we will consider that $\psi(i, \alpha) =$

$-\infty$ if $\alpha < 0$ or if the state (i, α) cannot be reached. We denote by $OPT(KP01{-}FO)$ the optimal value of the knapsack problem with conflicts.

Proposition 1.4.2 *If the items are sorted by decreasing fragility, we have:*

$$OPT(KP01{-}FO) = \max_{\alpha=0,\ldots,\psi_{\max}} \{\psi(\max\{i : \psi_i \geq \alpha\}, \alpha)\} \tag{1.62}$$

Each state (j, α) has to be explored only once. Using the same classical recursive formula as KP01, we obtain a complexity of $O(n \log n + \psi_{\max} * n)$ or $O(\psi_{\max} * n)$ if the items are initially sorted by decreasing fragility.

i	c_i	ψ_i	π_i
1	2	8	1
2	4	8	1
3	6	8	1
4	4	6	2
5	2	4	1

Tableau 1.1: A small instance of KP01-FO

In figure 1.6 we report the dynamic programming table used to solve the instance of BP01-FO of Table 1.1. Note that the states related to item 4 (resp. 5) and total size greater than ψ_4 (resp ψ_5) are forbidden. Following Proposition 1.4.2, we know that an optimal state can be found in the set of "rightmost" states in the table.

When item 1 is considered, only two states are possible: packing item 1 (profit 1) or not (profit 0). Once the first column is computed, the second can be computed by adding or not item 2 to each state of column 1. Each column is computed in turn following the same idea until all items have been considered.

1.4.4 Computing the initial set of columns using Variable-Neighborhood Search and strategic oscillation

In the two precedent sections, we proposed tabu search methods based on strategic oscillation. Another method that is able to implement this strategy is the *Variable Neighborhood Search* (VNS). We refer to Hansen et al. [57] for a recent and complete survey on VNS, and to Fleszar and Hindi [47] for a first successful application to the BPP.

The general idea behind this methodology is to iteratively modify the incumbent solution using a neighborhood which is initially small but becomes larger and larger during the iterations. Each new solution obtained in this way is optimized through local search and is possibly used to replace the incumbent one. We use the VNS to oscillate from feasible solutions to unfeasible solutions.

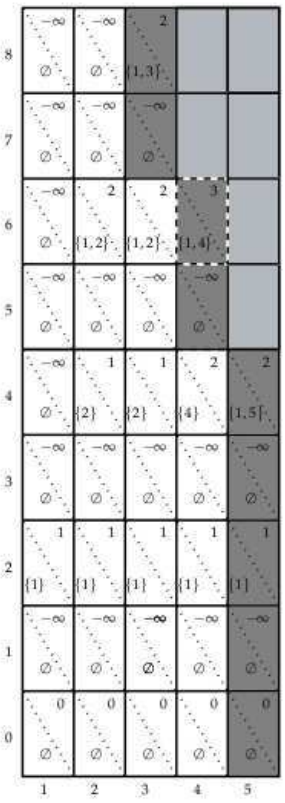


Figure 1.6: Table of states illustrating the dynamic programming procedure used to solve the instance of KP01-FO described in Table 1.1. Each line corresponds with a fragility level going up from 0 to ψ_{max} and each column i corresponds with the i first objects of I. Each cell corresponds with a state. It is split in two sub-cells: in the lower one we report the set of objects placed in the knapsack and in the upper one we report the corresponding profit. The dark gray cells represent dominant states (which contain an optimal solution), light grey cells are forbidden.

We start by **computing a heuristic solution**, say σ, using $U(\sigma)$ bins. The solution used as a starting point in the VNS is the one using the minimum number of bins among those found by different families of greedy algorithms.

We then enter a loop in which we modify σ using a **perturbation method** that depends on a parameter k, initially set to 1. We define the **neighborhood** $N_k(\sigma)$ as the set of solutions that are obtained by: (i) removing k bins from σ and (ii) reassigning the corresponding items in a way that possibly violates the fragility requirements but uses $U(\sigma) - 1$ bins. By using different criteria to perform steps (i) and (ii) we create a new solution, say σ', belonging to $N_k(\sigma)$.

The new solution σ' is possibly infeasible, because the sum of the weights of the items associated to one or more of the $U(\sigma) - 1$ bins may exceed the fragility of the most fragile items in the bins. We then try to minimize the sum of these weights excesses by using a set of **local search algorithms**. The new solution obtained after the local search application is denoted by σ''.

If we manage to restore feasibility in all bins of σ'', then we found a new heuristic

solution using one bin less. We thus update the incumbent solution and restart with $k = 1$. Otherwise, we reiterate the process until a maximum number of n_{iter} iterations has been elapsed. If σ'' remains infeasible after n_{iter} iterations, then we increase k by one unit, so as to perform a search in a larger solution space. Whenever k exceeds a given limit $k_{\max}$, we set again $k = 1$.

The algorithm is stopped whenever it finds an upper bound equal to the lower bound or after the maximum computing time is elapsed.

1.4.4.1 Strategic oscillation

As briefly discussed above, the aim of our **perturbation method** is to modify the incumbent solution σ so as to generate a new solution σ' belonging to the neighborhood $N_k(\sigma)$. This neighborhood is the set of solutions that can be obtained from σ by means of (i) the removal of k bins and (ii) the reassignment of the items originally packed in the removed bins to $U(\sigma) - 1$ bins. The $U(\sigma) - k$ bins that remain in σ after the removal of the k bins are copied directly into σ'; $k - 1$ new empty bins are opened in σ'; all the $U(\sigma) - 1$ bins obtained in this way are filled with the items originally to the k removed bins by accepting violations of the fragility requirements.

On the basis of computational outcome we use a two-level objective function to redistribute the removed items, which depends on the number of items conflicting with the fragility of a bin, and the weight excess in a bin.

The solution obtained after the execution of the perturbation method is usually infeasible, as some weight excess may exist in one or more bins. We try to **restore feasibility** by means of a local search procedure that *swaps* items between pairs of bins. We operate in a *first improvement* policy, *i.e.*, as soon as an improving move is found we perform it and re-iterate. We developed several types of swaps involving one, two, three and four items. We perform the swaps in non-decreasing order of their complexity. The local search phase is halted when all the weight excess has been removed from the bins or when no improving move is found for any type of swap.

This strategic oscillation proved to be very profitable also for the BPP-FO, because it allowed to quickly move from a solution to another by temporarily disregarding the fragility requirements, which may be particularly strict, hence facilitating local search.

1.4.5 Synthesis of the computational experiments

Since no benchmarks existed for the BPP-FO, we proposed a set of challenging instances based on our experimentations. These instances are derived from BP instance of the literature [84]. We generated instances where fragilities are strongly-correlated, weakly-correlated, and uncorrelated. We generated instances with up to 500 items. We kept the instances for which the compact model described in this section was no able to find a solution in five minutes.

Strongly correlated instances are very easy and can be all (but one) solved by the ILP model within the given time limit. Uncorrelated instances are instead more difficult. It is worth noting that for two classes the model is not even able to solve 20% of the instances to optimality. Weakly correlated instances are even more difficult, with three classes for which the 20% ratio of optimal solutions is not reached. The average percentage gaps are a bit higher than those noted for the uncorrelated ones.

Our overall algorithm outperforms the mathematical model, as it manages to obtain 78 optima out of 135, against the 51 obtained the model. Also the computational effort is smaller (163 seconds against 380) and the average percentage gap is reduced by a half. The mathematical model is very effective for solving the small instances with 50 items, but is weaker when solving the larger ones. The proposed algorithm can instead solve an interesting number of larger instances.

1.5 Conclusions, future works

We applied several decomposition methods (tree-decomposition and Dantzig-Wolfe decomposition) to packing problems with different kinds of conflicts (hard conflicts, soft conflicts and fragility). We showed that, using these decompositions and meta-heuristics based on strategic oscillation, we were able to design efficient methods for solving these difficult combinatorial problems.

Applying tree-decomposition to problems that do not only include graph constraints is not straightforward, but we have shown that it leads to interesting results for the bin-packing problem with conflicts. Our framework allows to make collaborate different kinds of methods, and can be easily parallelized. Using exact methods for solving the different subproblems could be viable if the clusters are not too large.

Concerning column generation, it transpires from our experiments that the min-conflict problem is much harder than the bin-packing with fragile objects. This is due to the difficulty to solve the pricing subproblem of the MCBP. On the contrary, in the BPFO, the pricing subproblem is tackled efficiently by our dynamic program scheme. Different issues arise for the creation of exact methods for these two problems. For the min-conflict problem, better ILP formulation (or better cuts) have to be proposed to fasten the resolution of the pricing subproblem. We are now studying the application of methods designed for the quadratic knapsack problem. For the bin-packing with fragile object, the difficulty is to find a branching scheme that does not break the structure of the subproblem, and allows our dynamic programming scheme to be used.

Concerning heuristic solutions, our experiments show that strategic oscillation is well suited to packing problems with conflicts. It allows the meta-heuristic to go from one good solution to another good solution quickly by using non-complete, or non-feasible solutions. It also allows efficient diversification and intensification phases, which are among the most important ingredients in a meta-heuristic based on local

search.

Dual-feasible functions and extensions

The work presented in this chapter has been published [28, 29, 33, 66] in several international journals. A report [31] will also be submitted soon to a journal.

2.1 Introduction

This chapter deals with fast lower bounds for several bin-packing problems. A large part of our work deals with the variant called *cutting-stock* problem. This problem is similar to the classical bin-packing problem: the main difference is that each item size is repeated many times. It can be written as follows.

Problem 7 (Cutting-Stock Problem (CSP)) *Given a set I of item types i of size c_i and demand b_i, how many rolls of size C are needed to cut b_i times each item i of I in such a way that the sum of their sizes in each bin is smaller than or equal to C?*

It can be modeled exactly like the bin-packing problem. However, the fact that items are repeated many times is a feature that helps linear programming based methods to solve this problem efficiently. The best model to date is due to Gilmore and Gomory [50,51] (see Chapter 1). This model has a strong linear relaxation (the famous MIRUP property[1], see [81]), and is able to take into account a large number of possible variants of bin-packing, since all constraints related to items in a given bin belong to the pricing subproblem, which dynamically generates feasible columns (packing patterns) for the model.

For large size instances, the computation of the LP relaxation of the model of Gilmore and Gomory can be expensive in time. Moreover, in some real-life applications, bin-packing lower bounds are computed repeatedly a large number of times. In these cases, fast alternative techniques have to be used. Several authors have focused on such techniques [18, 58, 69].

[1] The modified integer round-up property (MIRUP) for a linear integer minimization problem means that the optimal value of this problem is not greater than the optimal value of the corresponding LP relaxation rounded up plus one. Whether CSP has the MIRUP property or not remains an open problem in the general case.

In this chapter, we focus on lower bounding techniques based on so-called *dual-feasible functions* (DFF). This concept has been studied in the literature and successfully applied to the classical bin-packing problem. We focused on two aspects of DFF: the generation of useful DFF, and their generalization to more complicated packing problems.

Our first contribution is to survey the different dual-feasible functions that were used in the literature, some explicitly, other hidden behind a more complicated formalism. We also gather results concerning these DFF and superadditive functions, and give an insight into the simple frameworks that are generally used to generate DFF.

Our second contribution is a unified view and a simple formalism for dual-feasible functions applied to other packing problems. Then we give some applications of this concept to different variants of packing problems, namely the bin-packing problem, the bin-packing problem with conflicts, the bin-packing problem with fragile items, and the two-dimensional bin-packing problem with and without rotation.

2.2 Classical dual-feasible functions

The concept of dual-feasible function (DFF) has been used to improve the resolution of several cutting/packing (C&P) problems, and more generally any problem involving knapsack inequalities (scheduling problems, vehicle or network routing). It was used for the first time for deriving algorithmic lower bounds for bin-packing problems by Lueker [75]. Since then, several researchers have proposed new functions to improve the results obtained by the initial method.

2.2.1 Definitions and properties

Definition 2.2.1 *A function $f : [0,1] \to [0,1]$ is dual-feasible if for any finite set S of real numbers, we have*

$$\sum_{x \in S} x \leq 1 \Rightarrow \sum_{x \in S} f(x) \leq 1.$$

Dual-feasible functions are generally defined in $[0,1]$. However, when data are integer, using discrete values instead may lead to simpler formulations. Carlier and Néron [21–23] propose a discrete version of DFF. They use the designation of *redundant functions* to denote such functions. They are defined from $[0,C]$ to $[0,C']$ (C and C' strictly positive integers) instead of being defined from $[0,1]$ to $[0,1]$.

In this document, we consider discrete functions in general. We now define formally these discrete DFF. For the sake of simplicity, for a given integer value C, we define these function from $\{0,\ldots,C\}$ to $[0,1]$.

Definition 2.2.2 *For a given integer value C, a function $f : \{0,\ldots,C\} \to [0,1]$ is a discrete dual-feasible function if for any finite set S of values in $\{0,\ldots,C\}$, we have*

$$\sum_{x \in S} x \leq C \Rightarrow \sum_{x \in S} f(x) \leq 1.$$

In the following, unless it is clearly specified, all functions considered will be discrete. We now introduce a dominant subset of the DFF, called Maximal DFF (MDFF).

Definition 2.2.3 *A DFF f is a* Maximal Dual Feasible Function *(MDFF) if there does not exist any other DFF f' such that $\frac{f(x)}{f(C)} \leq \frac{f'(x)}{f'(C)}$ for all $x \leq C$ and there exists a value y such that $\frac{f(y)}{f(C)} < \frac{f'(y)}{f'(C)}$.*

Let us give a simple example of DFF, and how it is applied to produce a lower bound for the cutting-stock problem.

Example 2.2.1 *We consider an instance with a bin of size* 10*. The item set is composed of one item of size* 2*, ten items of size* 3*, one item of size* 7*, and ten items of size* 8*. A simple bound is obtained by summing the sizes of the items and divide it by the size of the bin. The result is rounded up since the number of bins is integer. $L_0 = \lceil (1*2+10*3+1*7+10*8)/10 \rceil = 12$. This bound can be improved by applying a discrete DFF to the instance as a preprocessing. We give below an example of discrete DFF defined from $\{0, \ldots, 10\}$ to $[0, 1]$. For each value x in $\{0, \ldots, 10\}$, we report the value of $f(x)$.*

x	0	1	2	3	4	5	6	7	8	9	10
$f(x)$	0	0	0	1/3	1/3	1/2	2/3	2/3	1	1	1

When this function is applied to the instance, the following new bound is obtained: $\lceil 1 * 0 + 10 * 1/3 + 1 * 2/3 + 10 * 1 \rceil = 14$.

Several properties characterize the MDFF functions. In the following, we rewrite a result proved by [23] using our formalism. Recall that a function f is said to be superadditive if for any x, y such that $f(x)$, $f(y)$ and $f(x+y)$ are defined, $f(x)+f(y) \leq f(x+y)$.

Proposition 2.2.1 *(Theorem 1. of [23]). A dual-feasible function is* maximal *if and only if $f(0) = 0$, f is monotonously increasing, f is superadditive, and f is such that for all $i = 1, \ldots, C/2$ it holds that $f(i) + f(C - i) = 1$.*

2.2.2 Data-dependent DFF

We introduced data-dependent dual-feasible functions (DDFF) in [20] to generalize a result proposed by Boschetti and Mingozzi [13]. The difference between DFF and DDFF is that DFF have to be valid for any instance, whereas DDFF are computed for a given instance (and thus cannot be applied to another instance). The definition of DDFF is the following.

Definition 2.2.4 *Let $I = \{1, \ldots, n\}$, $c_1, c_2, \ldots, c_n$ n integer values and C an integer such that $C \geq c_i$ for $i = 1, \ldots, n$. A* Data-Dependent DFF *(DDFF) f associated with C and $c_1, c_2, \ldots, c_n$ is a discrete application from I to $\{0, \ldots, C'\}$ such that*

$$\forall I_1 \subset I, \sum_{i \in I_1} c_i \leq C \Rightarrow \sum_{i \in I_1} f(i) \leq C'$$

Note that DFF are applied to indices. Two items of same size may have different images.

DDFF have proved to be effective when applied to two-dimensional bin-packing problems. They can lead to bounds that could not be reached by the application of DFF.

2.2.3 Applications of DFF

As explained above, the classical applications of DFF are related to the computation of lower bounds for the bin-packing problem. Formally, a bound is obtained as follows. Let f be a discrete DFF defined from $\{0, \ldots, C\}$ to $[0, 1]$, $\lceil \sum_{x \in S} f(x) \rceil$ is a lower bound for BPP.

Dual-feasible functions can also be used for problems in two or three dimensions by applying them independently on each dimension (see [44]). To our knowledge, DFF have only been computed for bin-packing problems.

Apart from computing fast lower bounds, any dual-feasible function can be used to generate valid inequalities for integer programs defined over the sets $S = \{x \in \mathbb{Z}_+^n : \sum_{j=1}^n a_{ij} x_j \leq b_i,\ i = 1, \ldots, m\}$ such that $b_i \geq a_{ij} \geq 0$ and $b_i > 0$ for any i, j.

Proposition 2.2.2 *Let $S = \{x \in \mathbb{Z}_+^n : \sum_{j=1}^n a_{ij} x_j \leq b_i,\ i = 1, \ldots, m\}$, with $b_i \geq a_{ij} \geq 0$ and $b_i > 0\ \forall i, j$. For any i, if $f : \{0, \ldots, b_i\} \to [0, 1]$ is a DFF, then $\sum_{j=1}^n f(a_{ij}) x_j \leq 1$ is a valid inequality for S.*

The term DFF is hardly used in the context of cuts for linear programs (with the notable exceptions of [4] and [2]). Note also that even if we focus on lower bounding strategies for bin-packing problems, our work on dual feasible functions (and the corresponding dual solutions of the cutting stock) have also allowed to propose techniques to stabilize column generation algorithms for this problem [29].

2.2.4 Discrete DFF and the model of Gilmore and Gomory

Discrete DFF are tightly linked with the model of Gilmore and Gomory [50, 51] for the cutting-stock problem. We already used this model in Chapter 1. We just recall some details.

A combination of items of I in a roll is called a *pattern*. Each possible cutting pattern is described by a column labelled p: $(a_{1p}, \ldots, a_{ip}, \ldots, a_{|I|p})^T$, where a_{ip} is the

number of items of width c_i in the pattern p. The model of Gilmore and Gomory [50,51] is the following model (already defined in Chapter 1) applied to the set of patterns P described just above.

$$\min \sum_{p \in P} z_p \tag{2.1}$$

$$\sum_{p \in P} a_{ip} z_p \geq b_i \quad i \in I \tag{2.2}$$

$$z_p \in \{0,1\} \quad \forall p \in P. \tag{2.3}$$

Here, a valid cutting pattern is such that

$$\sum_{i \in I} a_{ip} c_i \leq C, \qquad \forall p \in P \tag{2.4}$$

$$a_{ip} \in \mathbb{N}, \quad \forall p \in P, i \in I \tag{2.5}$$

As explained in Chapter 1, the columns of (2.1)-(2.3) are generated iteratively by solving the pricing subproblem. In our case, this problem is the classical knapsack problem.

The weak dual of (2.1)-(2.3) reads:

$$\max \quad \sum_{i \in I} b_i \pi_i \tag{2.6}$$

$$\text{s.t.} \quad \sum_{i \in I} a_{ip} \pi_i \leq 1, \forall p \in P \tag{2.7}$$

$$\pi_i \geq 0 \tag{2.8}$$

One can notice that the condition for a solution π to be valid for this dual is $\sum_{i \in I} a_{ip} c_i \leq C \implies \sum_{i \in I} a_{ip} \pi_i \leq 1$ for any pattern p, which is the definition of discrete dual-feasible functions.

An alternative definition can be given for discrete DFF associated with a size C: a function f is a discrete dual-feasible function if **for any instance** of cutting-stock where the size of the bins is C, there exists a valid solution π of (2.6)-(2.8) such that $f(i) = \pi_i$ for any value i in I.

An alternative definition of DDFF is that f is a DDFF dependent on a given instance D if there exists a valid dual solution π of (2.6)-(2.8) **applied to** D such that $f(i) = \pi_i$ for any value i in D. The main difference with DFF is that the dual problem here is less constrained than for the DFF, since some patterns do not belong to P, whereas for the DFF, all possible patterns related to the size C have to be considered.

This explains why the DFF are named *dual-feasible*.

2.3 A general point on view on DFF

We first propose a generalization of the concept of DFF to any problem that can be modeled with a set-covering model and a subproblem. This name this concept *Set-Covering-DFF* (SC-DFF).

Let $I = \{1, \ldots, n\}$ be a set of indices, and P the set of **all** possible *patterns* p related to a given problem $\mathbb{P}$. We denote by a_{ip} the number of items of type i that appear in pattern p and $v(p)$ the cost of pattern p. Note that P is of exponential, yet finite size.

Definition 2.3.1 *Let P be the set of **all possible patterns** defined for set I and a given problem $\mathbb{P}$. A Set-Covering-DFF (SC-DFF) for (I, P) is a mapping g defined from I to $\mathbb{R}^+$ such that*

$$p \in P \Rightarrow \sum_{i \in I} a_{ip} * g(i) \leq v(p) \tag{2.9}$$

The main difference with the classical DFF is that it applies to indices i instead of the sizes c_i. This is due to the fact that two items with the same size may be different (take the problem with conflicts for example).

In this formalism, geometric constraints of packing applications are modeled as a set of feasible patterns (the set of instance vector). Practically speaking, being able to characterize this (possibly exponential size) set without enumerating all its elements is crucial.

For most bin-packing problems, where the goal is to minimize the number of bins used, the value $v(p)$ is equal to 1 for all patterns p.

Now we give a definition of data-dependant set-covering DFF. This time the set of patterns $P(D)$ is restricted to those possible with the item sizes in D.

Definition 2.3.2 *Let I be a set of items and D an instance of problem $\mathbb{P}$. Let $P(D)$ be **the set of valid patterns related to instance** D. A Set-Covering-DDFF (SC-DDFF) related to instance D is a mapping g defined from I to $\mathbb{R}^+$ such that*

$$p \in P(D) \Rightarrow \sum_{i \in I} a_{ip} * g(i) \leq v(p) \tag{2.10}$$

Note that for two different instances of the same problem, a SC-DDFF can be valid for one and not for the other.

This formalism is compatible with constraints added during a branch-and-cut-and-price algorithm. This means that SC-DFF could be used in any node of a search tree, and not only at the root node as it is done up to now.

From now on, we will name CS-DFF the classical DFF (Cutting-Stock DFF). The term CS-MDFF will be used for Maximal CS-DFF.

Note that when the constraints of the initial problem are equality constraints (set-partitioning problem instead of set-covering problem), the only difference is that the

values of $f(i)$ can be negative. Practically speaking, this is rarely useful for packing problems, since in these particular cases, set-partitioning models can be relaxed into set-covering problem without modifying the value of an optimal solution. However, the concept of *"set-partitioning DFF"* could be interesting to introduce for some specific problems outside the field of bin-packing problems.

2.4 Computation of CS-DFF

Several papers propose CS-DFF ("classical" dual-feasible functions), sometimes implicitly when they are used to improve cuts in linear programs. Identifying the functions underlying complicated formulations is far to be easy in some cases. We surveyed the literature and gathered two literatures that were somehow disconnected. The goal was to propose a guide for generating CS-DFF.

2.4.1 Frameworks for creating valid CS-DFF

A simple way of combining two functions is to compute a linear combination, or to compose two CS-DFF (see [42] for example).

Proposition 2.4.1 *A composition, or a positive linear combination of superadditive functions is superadditive.*

More particularly, it is shown in [80] that if f and g are superadditive, then, for $\lambda \geq 0$, λf, $\lfloor f \rfloor$, $f+g$, $\min\{f,g\}$ are superadditive. Note that for $\lambda > 0$, $\max\{0, x-\lambda\}$ is also superadditive, whereas $x+\lambda$ is not.

A composition or a positive linear combination of CS-MDFF is also an CS-MDFF. Function $\min\{f,g\}$ is not maximal, unless $f=g$. As function $f(x) = \lfloor x \rfloor$ is not a CS-MDFF, in general $\lfloor f(x) \rfloor$ is not dominant either.

In this paragraph, we address the functions that apply on rational values. When x is rational, r_x will denote the fractional part of x ($r_x = x - \lfloor x \rfloor$). Practically speaking, if the data are integer, one can divide all values by a given rational to obtain rational values.

The following result shows that a way of associating two CS-DFF is to apply them separately to the integer part and to the fractional part of the values. This is valid if the conditions of Lemma 2.4.1 are verified.

Lemma 2.4.1 *Let f and g be two superadditive functions respectively defined on $[0,C]$ and $[0,1]$. If $f(x+1) - f(x) \geq v^*$ for all $x \in [0, C-1]$, and for all $y, y' \in [0,1]$ such that $y + y' > 1$, $g(y+y'-1) \geq g(y) + g(y') - v^*$, the function defined as follows*

$$h(x) = f(\lfloor x \rfloor) + g(r_x)$$

is superadditive on $[0,C]$.

The condition of the lemma is restrictive, since only strictly increasing functions f can lead to a superadditive function. However the conditions on function g are not too strong, and many functions can be used. For example, if g is a classical CS-DFF defined from $[0,1]$ to $[0,1]$, f has to be strictly increasing and such that $f(x)-f(x-1)>1$ for all x in $[0,C]$. We will show in the next section that functions proposed in [15] and [70] use this framework.

The ceiling function is not superadditive. However it can lead to superadditive functions if it is minored by a suitable value. We now generalize several results used implicitly in [87] and [70].

Lemma 2.4.2 *Let f be a superadditive function. If $\beta \geq 1$, $g(x) = \max\{0, \lceil f(x) \rceil - \beta\}$ is superadditive.*

In the literature, several superadditive and nondecreasing functions are proposed, which are not maximal. The following results aim at creating a CS-MDFF when one knows a non-maximal superadditive function f.

A dominating maximal CS-DFF $\widehat{f}$ can be built by keeping the images of the values smaller than $C/2$ and computing the images of the values larger than $C/2$ by symmetry. This is a generalization of what is done implicitly by Carlier *et al.* in [20].

Theorem 2.4.1 *Let f be a superadditive and nondecreasing discrete function defined from $\{0,\ldots,C\}$ to $[0,1]$, and such that $f(0)=0$. The following function is a maximal CS-DFF.*

$$\widehat{f}: \{0,\ldots,C\} \to [0,1]$$
$$x \mapsto \begin{cases} f(C) - f(C-x), & \text{for } C \geq x > \frac{C}{2}, \\ 1/2, & \text{for } x = \frac{C}{2}, \\ f(x), & \text{for } x < \frac{C}{2}. \end{cases}$$

Theorem 2.4.2 shows another way of obtaining an CS-MDFF from a non-maximal superadditive function f. The case we consider occurs when for some value x where f is not continuous, the value of $f(x)$ can be increased without modifying the other values. This technique leads to an improved function with some singular values x such that $\lim_{\varepsilon \to 0^-} f(x+\varepsilon) < f(x) < \lim_{\varepsilon \to 0^+} f(x+\varepsilon)$. This technique is implicitly used by [40,42,87] for example. Theorem 2.4.2 is used in the sequel to show that some functions are maximal.

For the sake of simplicity, for a given function f and a given value x^* for which f is defined, we define $\bar{f}^{x^*}$ the function defined as follows: $\bar{f}^{x^*}(x) = f(x)$ if $x \neq x^*$, and $\bar{f}^{x^*}(x^*) = f(x^*) + \varepsilon$ with ε a real value as small as needed. Note that when f is a CS-DFF, $\bar{f}^{x^*}$ may or may not be a CS-DFF. In the following theorem, for a given CS-DFF f, I_2 is the set of values x^* such that $\bar{f}^{x^*}$ is also a CS-DFF, *i.e.* the set of values for which the image can be increased. In the sequel we say that a function f is *right-continuous* in x if $\lim_{\varepsilon \to 0^+} f(x+\varepsilon) = f(x)$.

Theorem 2.4.2 *Let f be a superadditive and nondecreasing non-discrete function defined from $[0, C]$ to $[0, f(C)]$ such that $f(0) = 0$. We denote by I_2 the subset of values x from $[0, C]$ such that $\bar{f}^x$ is a CS-DFF, and I_1 the set of remaining values. We suppose that I_2 is a discrete set of values $\{x_1, \ldots, x_k\}$ and that f is right-continuous on each set $[0, x_1), (x_1, x_2), \ldots (x_k, 1]$ of I_1.*

For a given function g, the following function

$$h : [0, C] \to [0, f(C)]$$
$$x \mapsto \begin{cases} f(x) & \text{if } x \in I_1 \\ g(x) & \text{if } x \in I_2 \end{cases}$$

is a superadditive nondecreasing function if the following conditions are true.

1. $f(x) \leq g(x) \leq \lim_{\varepsilon \to 0^+} f(x + \varepsilon)$ *for any x in I_2*
2. $g(x) + g(y) \leq g(x + y)$ *if $x, y, x + y \in I_2$*
3. $g(x) + f(y) \leq g(x + y)$ *if $x \in I_2, x + y \in I_2$ and $y \in I_1$*

2.4.2 A comparative analysis of CS-DFF

In the following, we describe several CS-DFF that were used in the literature (explicitly or implicitly). These functions are defined for the values $\{0, \ldots, C\}$. The image domain will depend on the function. We will use the formalism that simplifies the most the presentation.

Fekete and Schepers [42] propose three dual-feasible functions. Two are maximal, but the third is not an CS-MDFF.

The first function f_0^k is used implicitly by Martello and Toth [78] in their L_2 lower bound for the bin-packing problem. Function f_0^k, with $k \in \{0, \ldots, \frac{C}{2}\}$, consists in removing all values of size less than a given threshold k, and symmetrically increasing to one the size of the large values.

$$f_0^k : \{0, \ldots, C\} \to [0, 1]$$
$$x \mapsto \begin{cases} 1, & \text{for } x > C - k, \\ x/C, & \text{for } k \leq x \leq C - k, \\ 0, & \text{for } x < k. \end{cases}$$

It has been shown that this function is superadditive and nondecreasing [2,35], and even maximal [35]. Only values $k \leq C/2$ such that $C - k$ is the size of a large item are interesting. Note than when k is small enough, f_0^k is equivalent to the identity function, and so the lower bounds for the bin-packing problem obtained using this function are never smaller than the initial continuous bound.

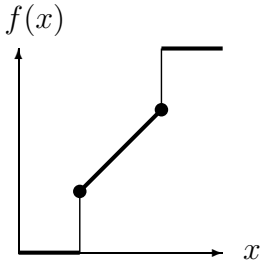


Figure 2.1: Function f_0^k for $C = 10$ and $k = 3$

The second function $f^k_{FS,1}$, $k \in \mathbb{N}^*$, can be seen as a special rounding procedure. It is an improvement on a function proposed by Lueker [75], using Theorem 2.4.2. The relative sizes of values equal to $C/(k+1)$, $C/(k+2)$, ..., $C/(k+1)$ are not modified.

$$f^k_{FS,1} : \{0, \ldots, C\} \to [0, 1]$$
$$x \mapsto \begin{cases} x/C, & \text{for } x(k+1)/C \in N, \\ \lfloor (k+1)x/C \rfloor \frac{1}{k}, & \text{otherwise.} \end{cases}$$

Proposition 2.4.2 *Function $f^k_{FS,1}$ is a maximal CS-DFF.*

The authors also propose a function $f^k_{FS,2}$, which is not superadditive (see [2]). Several functions of the literature dominate this function (see [14, 20]).

Boschetti and Mingozzi [14] and Boschetti [12] respectively propose bounds for the two- and three-dimensional bin-packing problems. For the two-dimensional bin-packing problem, they implicitly use function f_0^k and $f^k_{BM,1}$, an improved discrete version of $f^k_{FS,2}$. We do not report the formulation of this function, as a slightly improved version ([20]) is described next. In [35], it is shown that any iterative composition of $f_0^{k_i}$ and $f^{j_i}_{BM,1}$ is dominated by a function of the form $f^k_{BM,1} \circ f^l_0$.

Carlier *et al.* propose a slight improvement on the function of Boschetti [20], by enforcing the image of $\frac{C}{2}$ to be $1/2$. This function can also be obtained by applying Theorem 2.4.1 to function $\lfloor x/k \rfloor$. Note that as for f_0^k, when $k = 1$, this function is equivalent to the identity function. Let $k \in [1, C/2]$.

$$f^k_{CCM,1} : \{0, \ldots, C\} \to [0, 1]$$
$$x \mapsto \begin{cases} 1 - \frac{\lfloor (C-x)/k \rfloor}{\lfloor C/k \rfloor}, & \text{if } x > \frac{C}{2}, \\ 1/2, & \text{if } x = \frac{C}{2}, \\ \frac{\lfloor x/k \rfloor}{\lfloor C/k \rfloor}, & \text{if } x < \frac{C}{2}. \end{cases}$$

Proposition 2.4.3 *Function $f^k_{CCM,1}$ dominates $f^k_{FS,2}$.*

In [87], Vanderbeck uses a superadditive and nondecreasing function $f^k_{VB,1}$, $k \in \{2, \ldots, C\}$, to derive valid inequalities for the pattern minimization problem which are stronger than the rank 1 Chvátal-Gomory cuts [27]. His function states as follows.

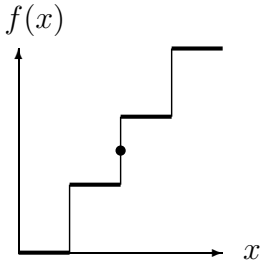


Figure 2.2: Function $f^k_{CCM,1}$ for $k = 3$

$$f^k_{VB,1} : \{0, \ldots, C\} \to [0, 1]$$
$$x \mapsto \frac{\max\{0, \lceil \frac{kx}{C} \rceil - 1\}}{k - 1}$$

Function $f^k_{VB,1}$ is a CS-DFF, and it is also superadditive (it is direct using Lemma 2.4.2). In [2], it is shown that $f^k_{FS,1}$ dominates $f^k_{VB,1}$.

Using Theorem 2.4.1, we have created a function $f^k_{VB,2}$ that dominates $f^k_{VB,1}$.

Proposition 2.4.4 *The following function is a maximal CS-DFF:*

$$f^k_{VB,2} : \{0, \ldots, C\} \to [0, 1]$$
$$x \mapsto \begin{cases} 1 - f^k_{VB,1}(C - x) & \text{if } x > 1/2 \\ 1/2 & \text{if } x = 1/2 \\ f^k_{VB,1}(x) & \text{if } x < 1/2 \end{cases}$$

For the two following functions, the discrete formalism leads to complicated formulations, so we use non-discrete functions instead. When integer data are used, one has to divide all values by a real value in $[1, C]$ before applying these functions.

In [15], Burdett and Johnson propose a function defined for rational values. When valid inequalities are considered, the data we have to deal with are often rational. If one wants to use this function for bounding (where data are in general integer), one can multiply the initial values by a rational constant. For a given value x, let r_x be the fractional part of x. The function of Burdett and Johnson is given next.

$$f_{BJ,1} : [0, C] \to [0, 1]$$
$$x \mapsto \lfloor x \rfloor / \lfloor C \rfloor + \max\left\{0, \frac{(r_x - r_C)/(1 - r_C)}{\lfloor C \rfloor}\right\}.$$

Any value $\alpha \neq 1$ can replace r_C in this function, but it appears [80] that the strongest inequality is obtained when $\alpha = r_C$. If $r_C = 0$, the function is equal to the identity function. In [80], the authors show that this function is superadditive. An alternative immediate proof of superadditivity derives directly from Lemma 2.4.1.

Proposition 2.4.5 *(directly deduced from Lemma 2.4.1 and [80]) Function $f_{BJ,1}$ is a maximal CS-DFF.*

Letchford and Lodi [70] propose another way of strengthening Chvátal-Gomory cuts [27] and Gomory fractional cuts [53] in linear programs. In the remainder, we suppose that the fractional part of C is such that $r_C > 0$. In [70], the authors do not precise that their improvement is based on a dual-feasible function. In this document, we explicitly formulate the dual-feasible function that underlies their method. As $f_{BJ,1}$, it is based on Lemma 2.4.1.

Proposition 2.4.6 *Let ζ be equal to $\lceil \frac{1}{r_C} \rceil - 1$. The following function is a CS-DFF.*

$$f_{LL,1} : [0, C] \to [0, 1]$$
$$x \mapsto \lfloor x \rfloor / \lfloor C \rfloor + \max \left\{ 0, \frac{\left\lceil \frac{r_x - r_C}{1 - r_C} \zeta \right\rceil}{(\zeta + 1) \lfloor C \rfloor} \right\}.$$

We use Lemma 2.4.2 to show that this function is superadditive. However, it is not maximal.

Proposition 2.4.7 *Function $f_{LL,1}$ implicitly used by Letchford and Lodi [70] is superadditive, but not maximal.*

This means that one can propose an improved version of this function by applying Theorem 2.4.1.

Proposition 2.4.8 *The following function is a maximal CS-DFF, and dominates $f_{LL,1}$.*

$$f_{LL,2} : [0, C] \to [0, 1]$$
$$x \mapsto \begin{cases} 1 - f_{LL,1}(C - x), & \text{if } x > C/2, \\ 1/2, & \text{if } x = C/2, \\ f_{LL,1}(x), & \text{if } x < C/2. \end{cases}$$

Note that Dash and Günlük [40] have proved that ζ can be replaced in $f_{LL,1}$ by any integer value k greater than ζ. We will use $f^k_{LL,1}$ for this extension.

A particular case of the so-called *extended 2-step Mixed-Integer Rounding (MIR)* inequalities of Dash and Günlük [40] leads to a cut that can be obtained by applying a CS-MDFF. This function also dominates $f^k_{LL,1}$, yet it is not equal to $f^k_{LL,2}$. Again, we suppose that the fractional part of C is such that $r_C > 0$.

Proposition 2.4.9 *Let k be an integer greater than or equal to $\lceil \frac{1}{r_C} \rceil - 1$. The following function is a CS-DFF.*

$$f^k_{DG,1} : [0, C] \to [0, 1]$$
$$x \mapsto \begin{cases} \lfloor x \rfloor / \lfloor C \rfloor + \frac{(r_x - r_C)/(1 - r_C)}{\lfloor C \rfloor} & \text{if } k \frac{1 - r_x}{1 - r_C} \in \mathbb{N} \text{ and } r_x > r_C, \\ \lfloor x \rfloor / \lfloor C \rfloor + \max \left\{ 0, \frac{\left\lceil \frac{r_x - r_C}{1 - r_C} k \right\rceil}{(k+1) \lfloor C \rfloor} \right\}, & \text{otherwise .} \end{cases}$$

This function is maximal, since it is obtained from $f^k_{LL,1}$ using Theorem 2.4.2.

Proposition 2.4.10 *Function $f^k_{DG,1}$ is increasing, superadditive, and symmetric and hence is a CS-MDFF.*

2.4.3 Summary of our literature study

In this paragraph, we sum up the different results stated in the current section. We recall several kind of results: dominance, maximality, and the different techniques underlying each family of functions.

In Table 2.1, we report a classification of the different functions. For each function, we recall the paper in which context it was proposed ('-' means that it is a trivial CS-DFF). We also report the type of application for which it has originally been designed (column Appli.: *lb* for lower bounding, and *cuts* for improved valid inequalities). Then we give informations for each function: if it is a CS-MDFF, if it explicitly uses Lemmas 2.4.1 and 2.4.2, or Theorem 2.4.1.

Functions $\lfloor \frac{x}{k} \rfloor$ $(k \neq 1)$, $f^k_{FS,2}$, $f^k_{LL,1}$ and $f^k_{VB,1}$ are not maximal, whereas $f^k_{BM,1}$ is *almost* maximal. Only the image of $\frac{C}{2}$ makes the latter non-maximal. All the other functions are maximal. Among all the functions considered, only $f^k_{FS,2}$ is not superadditive.

Functions $f^k_{BJ,1}$, $f^k_{LL,1}$ and its improved versions $f^k_{DG,1}$ $f^k_{LL,2}$ are the only ones to use Lemma 2.4.1. Note that these functions were originally proposed to derive cuts, which can explain the fact that the fractional part is treated apart.

Lemma 2.4.2 is used to modify the fractional part in $f^k_{LL,1}$, $f^k_{LL,2}$ and $f^k_{DG,1}$. It underlies function $f^k_{VB,1}$, and the function that dominates it, $f^k_{VB,2}$. Function $f^k_{FS,1}$ does not use explicitly Lemma 2.4.2, but its structure is close to it (it "almost" uses Lemma 2.4.2, since the integer values x are treated separately).

Theorem 2.4.1 is used by $f^k_{CCM,1}$, $f^k_{LL,2}$ and $f^k_{VB,2}$, and *almost* by $f^k_{BM,1}$ (only the image of $C/2$ has to be modified). Theorem 2.4.2 is used by $f^k_{FS,1}$ and $f^k_{DG,1}$ to obtain a maximal function, and by $f^k_{VB,1}$ to obtain a non-maximal function.

We have compared the different CS-DFF analyzed above against several types of instances for the one-dimensional bin-packing problem generated in a classical way. We used instances with up to 10000 items. Even for these large instances, each lower bound is computed in less than 1 second.

As expected, maximal functions lead to improved results compared to non-maximal functions. What is more surprising is the fact that function $f^k_{CCM,1}$ is strictly better than $f_{BM,1}$ for many test cases, although it only modifies the image of $\frac{C}{2}$. This can be explained by the fact that the instances are generated randomly, and thus items of size $\frac{C}{2}$ may appear several times in an instance.

Only the bounds based on the rounding function ($f^k_{BM,1}$ and $f^k_{CCM,1}$) are better than f^k_0 on average. This means that if one wants to use a unique function, $f^k_{CCM,1}$ would be this one (since it dominates the other rounding-based functions). But if one

Function	Paper	Appli.	CS-MDFF	lem. 2.4.1	lem. 2.4.2	thm. 2.4.1	thm. 2.4.2
identity	-	-	yes	no	no	no	no
f_0^k	[42]	lb	yes	no	no	no	no
$\lfloor \frac{x}{k} \rfloor$	-	-	no	no	no	no	no
$f_{FS,2}^k$	[42]	lb	no	no	no	no	no
$f_{BM,1}^k$	[14]	lb	almost	no	no	almost	no
$f_{CCM,1}^k$	[20]	lb	yes	no	no	yes	no
$f_{VB,1}^k$	[87]	cuts	no	no	yes	no	yes
$f_{FS,1}^k$	[42]	lb	yes	no	almost	no	yes
$f_{BJ,1}^k$	[15]	cuts	yes	yes	no	no	no
$f_{LL,1}^k$	[70]	cuts	no	yes	yes	no	no
$f_{DG,1}^k$	[40]	cuts	yes	yes	yes	no	yes
$f_{LL,2}^k$	[29]	-	yes	yes	yes	yes	no
$f_{VB,2}^k$	[29]	-	yes	no	yes	yes	no

Tableau 2.1: Summary of the properties of the functions analyzed in this document

is looking for the best results, he will have to use all the maximal CS-DFF described in this document.

2.5 Extensions of DFF for various bin-packing problems

In this section, we describe the extensions of DFF to other packing problems that we have proposed. The problems addressed are all bin-packing problems. We consider the cases where conflicts between items can occur (pairwise conflicts, or a so-called fragility constraint). We also consider several variants of the two-dimensional case.

2.5.1 DDFF for the bin-packing problem (BP-DDFF)

In the following, we present DDFF designed for the bin-packing problem. For our study, the only difference between bin-packing and cutting-stock will the fact that the number of items for each type is small and thus packing $\lfloor C/c_i \rfloor$ instances of a given item i may not be allowed. This means that the set P of valid patterns is different. In BP-DDFF, the number of times an item is repeated in the instance is taken into account.

2.5.1.1 Definition and link with CS-DFF

To our knowledge, the first implicit use of BP-DDFF for computing lower bounds for bin-packing problems is due to Boschetti and Mingozzi [13]. We originally defined this

concept under the name of *data-dependent dual-feasible functions (DDFF)* [20]. In this document, we will use the term *BP-DDFF* (DDFF defined for bin-packing problem). We now define formally this concept of BP-DDFF.

Definition 2.5.1 *[20] Let I be a set of items i of size c_i, and C a size of bin. A bin-packing DDFF (BP-DDFF) g associated with this instance is a mapping from I to $[0,1]$ such*

$$\sum_{i \in I_1 \subseteq I} c_i \leq C \Rightarrow \sum_{i \in I_1} g(i) \leq 1 \tag{2.11}$$

Any CS-DFF (or "classical" DFF) is tightly related to BP-DDFF.

Proposition 2.5.1 *Let $f : \{0, \ldots, C\} \to [0,1]$ be a discrete CS-DFF. Function $g : I \to [0,1]$ defined as follows: $g(i) = f(c_i)$ is a BP-DDFF for any instance.*

However, some BP-DDFF are not BP-DFF, and only apply on specific instances.

We now give an example of DDFF to illustrate the fact that DDFF can lead to values that would not be valid for a DFF.

Example 2.5.1 *Consider an instance with a bin of size 100, and two items: item 1 of size 5 and item 2 of size 96. A valid DDFF defined from I to $[0,100]$ can map item 1 to value 99 and item 2 to value 1. Note that with a DFF, the image of an item of size 1 cannot be larger than 1/100.*

2.5.1.2 A general BP-DDFF

The first BP-DDFF has been proposed by [20]. It uses a special parameter k. This method generalizes the work of [13] and gives a different viewpoint on the method. We now propose a way of computing a more generic family of BP-DDFF.

Let $I = \{1, \ldots, n\}$ be a set of indices, C an integer value, and $c_1, c_2, \ldots, c_n$ a list of integer values less than or equal to C, and J a subset of I. The following family of functions uses an arbitrary set of parameters $\alpha = \{\alpha_i \in \mathbb{N} : i \in I\}$. We denote by $KP(C, J, c, \alpha)$, the value of an optimal solution to the classical one-dimensional knapsack problem (Problem 2). The value C is the size of the bin, J the set of items i, each of size c_i, and α is a function that associates a profit to the items of J.

Formally, $KP(C, J, c, \alpha)$ can be stated as follows.

$$KP(C, J, c, \alpha) = \max_{J' \subseteq J, \sum_{i \in J'} c_i \leq C} \{\sum_{i \in J'} \alpha_i\}$$

Proposition 2.5.2 *The following function g_1 is a BP-DDFF defined for a given instance D.*

$$g_1 : I \to [0,1]$$
$$i \mapsto \begin{cases} 1 - KP(C - c_i, J, c, \alpha)/KP(C, J, c, \alpha) & \text{if } i \in J \\ \alpha_i / KP(C, J, c, \alpha) & \text{if } i \in I \setminus J \end{cases}$$

The values of the small items are equal to α_i, and the sizes of the bin and of the large items are computed by solving the knapsack problem described above. Then all values are divided by $KP(C, J, c, \alpha)$ to obtain a function in $[0, 1]$.

When applying a CS-DFF on D, removing an item may decrease the value of the lower bound obtained. When BP-DDFF are used, this observation does not hold anymore since the value of other items may be increased using the knapsack problem.

The knapsack problems involved are NP-hard in the general case. However they can be solved in pseudo-polynomial time using dynamic programming (see [78] for example). When the size of the bin is large, it may entail a large computing time. In this case, the set of parameters α should be chosen in a way to re-enable the resolution of the knapsack problem in a polynomial time. We investigated this idea in [35] by choosing $\alpha_i = 1, \forall i \in J$, similarly to what is implicitly done in [13]. The optimal value of the knapsack problem is then equal to the maximum number of items that can be put together in a knapsack of size C. It can be solved in linear time if the items are sorted by increasing order of size. However, the general form g_1 is more efficient for computing lower bounds than the method of [35]. The best results on average were obtained by using $\alpha_i = c_i,\ \forall i \in J$.

2.5.1.3 Practical usefulness

Practically speaking, these functions were able to improve the quality of the lower bounds for well-known two-dimensional benchmarks from the literature. For the one-dimensional benchmarks, the improvement is small. This is due to the fact that in two-dimensional benchmark the number of items of each type is generally smaller than in the one-dimensional case. Consequently, taking into account the fact that the number of small items is limited helps computing the quantity of lost space in the bins.

2.5.2 DDFF for the bin-packing with conflicts (BPC-DDFF)

We now propose DDFF for the bin-packing with conflicts. Note that we consider the generic case without specifying the geometric constraint applied to the problem. Similarly to BP-DDFF, we will name BPC-DDFF the DDFF designed for BPC. We only propose functions that depend on the data, since it seems difficult (if not impossible) to design effective methods that would be valid for any graph.

Both techniques we propose are based on graph concepts: graph triangulation for the first, tree-decomposition for the second. A formal definition of these concepts is given in Chapter 1.

2.5.2.1 Knapsack-based BPC-DDFF

The first BPC-DDFF is a generalization of function g_1 defined in Proposition 2.5.2 for BP.

Proposition 2.5.3 *Let h_1 be the function obtained from function g_1 where the knapsack problem KP is replaced by the disjunctive knapsack problem. Function h_1 is an BPC-DDFF.*

To our knowledge, no dynamic programming scheme exists for the disjunctive knapsack problem with general graph. A first solution to solve these problems is to use an ILP solver like ILOG cplex. We used other methods, based on graph concepts and a relaxation of the problem.

When a conflict graph G is considered, only stable sets of G can be solutions of the knapsack problem with conflicts (KPC). Thus a (possibly not practically tractable) solution for the KPC is to compute all maximal stable sets of the conflict graph, and then to solve for each stable set the associated knapsack problem. The maximum value obtained for all stable sets is the optimal value for KPC. This solution is tractable only if 1) the number of stable sets is small and 2) they can be computed with a small complexity. Neither of the two conditions are fulfilled when a random graph is considered.

For this method to be tractable, we relax our problem by removing edges to our conflict graphs in such a way that its complementary graph becomes *triangulated* (see Chapter 1). Tarjan and Yannakakis proved in [85] that any triangulated graph G has at most n maximal cliques. In addition, they described a linear algorithm to recognize a triangulated graph and to enumerate its maximal cliques. In our case, the compatibility graph $\overline{G}$ is rarely triangulated. Finding the minimum set of edges to add in order to obtain a triangulated graph is a NP-hard problem, so we use the heuristic called Maximum Cardinality Search (MCS) [85] to triangulate the initial compatibility graph.

In Figure 2.3, we give an example of triangulated compatibility graph and the corresponding knapsack problems to solve. To compute the size of the bin, the six knapsack problems have to be solved. To compute the size of item 10, only the knapsack problems KP_3 KP_4, KP_5 and KP_6 have to be solved.

2.5.2.2 A BPC-DDFF based on graph decomposition

Suppose the set I of items can be decomposed into two sets I_1 and I_2 of pairwise incompatible items. In this case, two different DFF f and g can be applied to I_1 and I_2, since the instance can be decomposed into two distinct sub-instances. Now, if there is a third set I_3 where each item is compatible with some items of I_1 and I_2, a lower bound can be obtained as follows: $\left\lceil \sum_{i \in I_1} f(i) + \sum_{i \in I_2} g(i) + \sum_{i \in I_3} \min\{f(i), g(i)\} \right\rceil$. This is true since each item of I_3 will be packed either with items of I_1, items or I_2 but not both. We have used this technique to derive lower bounds during a branch-and-bound method for the bin-packing problem [30].

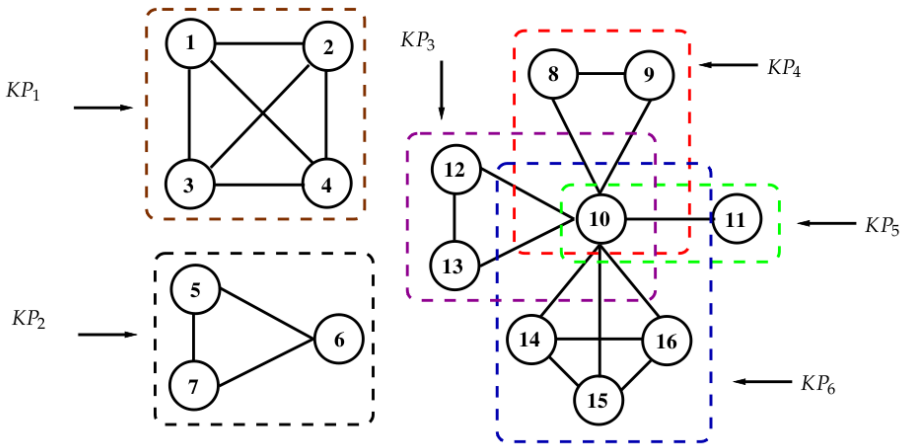


Figure 2.3: Knapsack problems to solve to compute the knapsack-based BPC-DDFF

Using our formalism, it is equivalent to applying the following BPC-DDFF, which depends on two CS-DFF f and g.

$$h(f,g) : I \rightarrow [0,1] \tag{2.12}$$

$$i \mapsto \begin{cases} f(c_i) & \text{if } i \in I_1 \\ g(c_i) & \text{if } i \in I_2 \\ \min\{f(c_i), g(c_i)\} & \text{otherwise} \end{cases} \tag{2.13}$$

This technique can be generalized by decomposing the graph into different intersecting subsets. Function h_2 is based on the concept of *tree-decomposition* (see Chapter 1), which captures the possible associations of items.

Let $\overline{G} = (I, I \times I \setminus E)$ be the compatibility graph of the instance, and $T = (S, A)$ a tree-decomposition of $\overline{G}$. The basic idea of h_2 is to assign a given DFF f_s to each node $s \in S$ of the tree-decomposition T.

Let F be a list of valid discrete DFF $f_1, \ldots, f_{|S|}$ defined from $\{0, \ldots, C\}$ to $[0,1]$, one for each node of the tree decomposition. For each vertex i in the graph we define S^i the set of nodes of the tree decomposition containing i. Clearly there is always a set of functions $f_1, \ldots, f_{|S|}$ that allows to dominate the application of a single DFF (e.g. $f_1 = f_2 = \ldots = f_{|S|}$).

Proposition 2.5.4 *The following function h_2 is a BPC-DDFF.*

$$h_2 : I \rightarrow [0,1] \tag{2.14}$$

$$i \mapsto \min_{s \in S^i}\{f_s(c_i)\} \tag{2.15}$$

An issue is to choose a suitable set of functions to be applied to the nodes of the tree decomposition. We use the following heuristic. For each node s, we compute the value of the bound associated with each function of our initial set F and we record the function that leads to the best value. This strategy may not be optimal but it leads to fast bounds.

If this technique is applied to an instance with the graph of Figure 2.3, since the graph is triangulated, the optimal tree-decomposition uses the six maximal cliques of the graph. Only one DFF is applied to items $1, 2, 3$ and 4. Since item 10 belongs to four clusters (four maximal cliques), four function will be applied to its size (one per cluster), and the minimum will be kept.

2.5.2.3 Practical usefulness

These functions helped improving the results for the two-dimensional case of BPC. For the one-dimensional case, the bounds were already tight. The results are somehow disappointing, since the functions are not sufficient to obtain competitive results (even if they improved the previous best combinatorial lower bounds). More complicated bounds (involving the resolution of a transportation problems) were used after the application of the DFF.

Considering the fact that the column-generation method returns good results for this problem, we think that there is room for improvement for this variant of bin-packing. Note that finding a DDFF for this problem is equivalent to finding a heuristic solution for the dual problem. This would require using techniques from both continuous and discrete optimization fields.

2.5.3 DFF for the bin-packing problem with fragile items (BPFI-DFF)

In this section, we propose DFF for the bin-packing problem with fragile items (see Chapter 1). Recall that the fragility of an item i is denoted ψ_i.

The only lower bound previously dedicated to this problem is the so-called "fractional lower bound". This bound is obtained by packing iteratively the items by increasing fragility, allowing several parts of an item to be packed in two consecutive bins (see [5]). It is a direct adaptation of the linear lower bound for the classical bin-packing problem.

2.5.3.1 Definition and properties

We first define formally the notion of *dual-feasible functions for BPFI* (BPFI-DFF).

Definition 2.5.2 *A mapping g defined from I to $[0, 1]$ is a BPFI-DFF if*

$$\sum_{i \in S \subseteq I} c_i \leq \min_{j \in S}\{\psi_j\} \implies \sum_{i \in S} g(i) \leq 1 \tag{2.16}$$

When a BPFI-DFF is computed, it can be directly used to derive a lower bound for BPFI.

Proposition 2.5.5 *If g is a BPFI-DFF and I a set of items to pack in a BPFI instance D, $L_g = \lceil \sum_{i \in I} g(i) \rceil$ is a valid lower bound for the minimum number of bins to use for D.*

The following proposition directly follows from Equation (2.16) and the definition of a CS-DFF.

Proposition 2.5.6 *If λ is a CS-DFF and g and BPFI-DFF, $\lambda \circ g$ is a BPFI-DFF.*

This means that a CS-DFF can be applied as a postprocessing when a BPFI-DFF is applied to the original instance, and may improve the results obtained.

The following results show a relation between superadditive functions and BPFI-DFF. Note that without loss of generality, we consider that the fragilities are strictly greater than 0.

Proposition 2.5.7 *Let λ be a superadditive and increasing function such that $\lambda(0) = 0$. The following function g is a BPFI-DFF.*

$$g : i \mapsto \lambda(c_i)/\lambda(\psi_i) \tag{2.17}$$

Note that all maximal CS-DFF described in this chapter are superadditive and increasing. Consequently, all useful functions defined for the cutting-stock can be used for the bin-packing with fragile items. If they are defined independently of a size of bin, they can be applied in a straightforward way. However some CS-DFF a defined according to a size of bin. In this case, $\psi_{\max}$ can be used as a fictive size of bin, or hand-tailored techniques can be applied to improve this value.

The result of Proposition 2.5.7 can be improved by increasing the image of the large items ($i \in I$, $c_i > \psi_i/2$) to the largest possible size that is allowed when the small items have been transformed using function λ.

Proposition 2.5.8 *Let λ be a superadditive and increasing function such that $\lambda(0) = 0$. The following function $\bar{g}$ is a BPFI-DFF.*

$$\bar{g} : i \mapsto \begin{cases} 1 - \max\limits_{\rho=0,\ldots,\psi_i - c_i} \{\frac{\lambda(\rho)}{\lambda(c_i+\rho)}\} & \text{if } c_i > \psi_i/2 \\ \frac{\lambda(c_i)}{\lambda(\psi_i)} & \text{if } c_i \leq \psi_i/2 \end{cases} \tag{2.18}$$

2.5.3.2 A specific family of BPFI-DFF

Creating BPFI-DFF from classical CS-DFF can be done using the results of Propositions 2.5.6,2.5.7 and 2.5.8. We now give an application of these theoretical results to a simple superadditive function: $\lfloor . \rfloor$. We also show a way of using two different BPFI-DFF to a given instance by using a decomposition method.

Corollary 2.5.1 *Let k be a given parameter ($1 \leq k < \min_{j \in I}\{\psi_j\}$), the following function is a BPFI-DFF.*

$$g_2^k : i \mapsto \frac{\lfloor c_i/k \rfloor}{\lfloor \psi_i/k \rfloor} \tag{2.19}$$

Note that using g_2^k yields bounds that are better than L_0 and L_1, since taking $k = 1$ leads to the same value as L_1. It can also be strictly greater than L_2. Take n items of width $C/2 + \varepsilon$ and fragility C. In this case, $L_2 = n/2 - 1$, while the bound obtained from $g_2^{C/2+\varepsilon}$ is equal to $\sum_{i \in I} 1/1 = n$ (which is the optimal result).

Function g_2^k can be improved by increasing the image of the large items ($i \in I$, $c_i > \psi_i/2$) to the largest possible remaining space when the other items have been transformed using g_2^k.

Corollary 2.5.2 *Let k be a given parameter ($1 \leq k < \min_{j \in I}\{\psi_j\}$), the following function is a BPFI-DFF.*

$$\bar{g}_2^k : i \mapsto \begin{cases} 1 - \max\limits_{\rho=1,\ldots,\psi_i - c_i} \{\frac{\lfloor \rho/k \rfloor}{\lfloor (c_i+\rho)/k \rfloor}\} & \text{if } c_i > \psi_i/2 \\ \frac{\lfloor c_i/k \rfloor}{\lfloor \psi_i/k \rfloor} & \text{if } c_i \leq \psi_i/2 \end{cases} \tag{2.20}$$

Let us give an example to illustrate the improvements that can be achieved using BPPFO-DFF.

Example 2.5.2 *Take an instance with 100 items of width 5 and fragility 8 and 100 items of width 2 and fragility 7. Let L_2 be the fractional bound.*

$L_2 = 92$ (the 28 first bins are used to pack the 98 first items of size 2, then one bin contains two items of size 2 and 3 units of an item of size 5. Finally, the remaining items of size 5 are fractionally packed in 63 bins.

$L_{g_2^k}$ with $k = 2$: $\lceil 100 * \frac{\lfloor 5/2 \rfloor}{\lfloor 8/2 \rfloor} + 100 * \frac{\lfloor 2/2 \rfloor}{\lfloor 7/2 \rfloor} \rceil = \lceil 100 * 1/2 + 100 * 1/3 \rceil = 84$

$L_{\bar{g}_2^k}$ with $k = 2$: $\lceil 100 * (1 - \frac{\lfloor 2/2 \rfloor}{\lfloor (5+2)/2 \rfloor}) + 100 * \frac{\lfloor 2/2 \rfloor}{\lfloor 7/2 \rfloor} \rceil = \lceil 100 * 2/3 + 100 * 1/3 \rceil = 100$

2.5.3.3 Knapsack-based BPFI-DDFF

Once again, function g_1 defined in Proposition 2.5.2 can be used. In this case, the knapsack problems to solve to compute the size of the bin and the size of the large items is a *knapsack problem with fragile items*. A dynamic programming method for this problem is described in Chapter 1.

2.5.3.4 Practical usefulness

The functions proposed lead to bounds that are close to the value returned by the model of Gilmore and Gomory. However, they are also close to the simple fractional lower bound. The small gap between this latter bound and the column-generation one leaves small room for improvements for the DFF.

2.5.4 DFF for two-dimensional bin-packing problems (2BPP-DFF)

We now consider the two-dimensional bin-packing problem with and without rotation. We show that CS-DFF can be used to derive new 2BPP-DFF. The result for the oriented case is due to Fekete and Schepers [44] (we just rewrite their result to fit our formalism). Our main contribution here is the improvement for the case with rotation.

Problem 8 (Two-dimensional Bin-Packing Problem (2BPP)) *Given a set I of rectangular items i of size (w_i, h_i), what is the minimum number of bins of size (W, H) needed to pack all the items of I in such a way that in each bin, the items can be packed inside the boundaries of the bin without overlapping? If the rotation of the items is allowed, we have a 2BPP with rotation (2BPP-R), otherwise, we have a 2BPP with fixed orientation (2BPP-O).*

We avoid the repetitive formal definition of a 2BPP-O-DFF and 2BPP-R-DFF. Just recall the fact that, using such a DFF, for any valid pattern P for 2BPP, the sum of the images of the items in P is smaller than 1.

2.5.4.1 DFF for the oriented case [44] (2BPP-O-DFF)

Fekete and Schepers [44] have shown that two DFF could be applied to each dimension of an instance of 2BPP-O to obtain a lower bound. Using our formalism, their result can be written as follows.

Proposition 2.5.9 *Let f and g be two discrete CS-DFF respectively defined from $\{0, \ldots, W\}$ to $[0, 1]$ and $\{0, \ldots, H\}$ to $[0, 1]$. The following function is a 2BPP-O-DFF.*

$$\varphi : i \mapsto f(w_i) * g(h_i) \tag{2.21}$$

If the identity function is used for f and g, the bound based on the surface of the bins is obtained. Note that the value of the image depends on the two dimensions independently. No actual two-dimensional DFF were derived in the literature. This can be explained by the fact that characterizing the set of feasible patterns is hard. Even verifying that a pattern is feasible is NP-complete (see Chapter 3).

To our knowledge, there are no lower bounding techniques for 2BPP that do not rely on techniques dedicated to the one-dimensional case. However, they have shown their effectiveness on hard two-dimensional instances. This has been confirmed by Caprara *et al.* [17]. In this paper, the authors have proposed a bilinear programming method for finding the best pair of DFF to apply to a bin-packing instance. This method leads to results that are close to those obtained by column-generation algorithms.

2.5.4.2 DFF for the case with rotation (2BPP-R-DFF)

The two following results are rewritings of those proposed in [33]. The first is a generalization of a bound of [14], the second is a truly original result.

The first result derives from a simple fact. For two given CS-DFF f and g, if a lower bound for the oriented case based on these two functions is run for all possible orientations of the items, and if the minimum is recorded, a valid lower bound is obtained. Of course, the bound obtained would need an exponential time, since it would lead to 2^n lower bounds to compute. Nevertheless a lower bound can be computed by considering the following relaxation: for each item i, keep the smallest image that it can have for its possible orientations. This leads to the following result.

Proposition 2.5.10 *(implicitly used in [14]) Let f and g be two discrete CS-DFF respectively defined from $\{0,\dots,W\}$ to $[0,1]$ and from $\{0,\dots,H\}$ to $[0,1]$. The following function is a 2BPP-R-DFF.*

$$\varphi_1 : i \mapsto \min\{f(w_i) * g(h_i), g(w_i) * f(h_i)\} \tag{2.22}$$

A better DFF, that dominates the previous one (if f and g are increasing and superadditive), is now described.

Proposition 2.5.11 *Let f and g be two CS-DFF defined as above. The following function is a 2BPP-R-DFF.*

$$\varphi_2 : i \mapsto \frac{f(w_i) * g(h_i) + g(w_i) * f(h_i)}{2} \tag{2.23}$$

The result is not intuitive, but is becomes obvious when the following relaxation is considered. From a $2BPP{-}R$, construct a $2BPP{-}O$ instance I' of size $2 * n$ where each item is repeated once for each of its orientation. Clearly, the value of an optimal solution for this new problem cannot be more than twice the value of an optimal solution for the original $2BPP{-}R$ instance. Take an optimal solution for $2BPP{-}R$ with z bins, keep the z bins and create z new bins by rotating the z first bins. You obtain a feasible solution for the new $2BPP{-}O$ instance with $2 * z$ bins (see Figure 2.4).

The last result only holds when the bin is a square. It can be adapted for the case where the bin is a rectangle by introducing dummy items (and thus constructing an

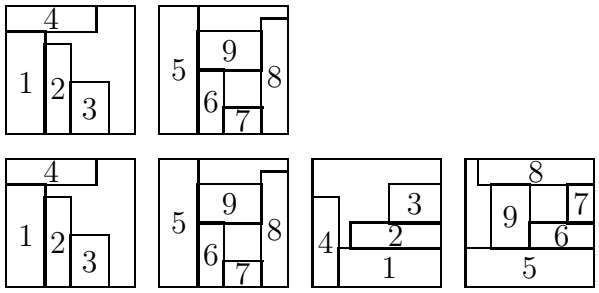


Figure 2.4: An optimal solution for $2BP|R$ with z bins (the two upper bins) and a solution for the $2BP|O$ relaxation using $2 * z$ bins (the two lower bins)

instance where the bins are square). Unfortunately, doing so breaks the dominance result between φ_2 and φ_1. Note that the fact that some items can only have one orientation is not taken into account in this result.

2.5.4.3 Practical usefulness

For the two-dimensional cases, the DFF lead to surprisingly good results when the number of bins is large. When the number of bins decreases, the geometric constraint becomes more important and the DFF are weaker. However, the results remain of good quality (see Chapter 3 for a discussion on the reduction of the computing time obtained using DFF). For the case with rotations, the bounds are rather simple, but dominate the other bounds from the literature and are practically tight when the bin is a square. However, in some cases, considering a rectangular bin can decrease the quality of the bound by a wide range.

2.6 Conclusions, future works

In this chapter, we focused on lower bounding techniques for bin-packing problem using the concept of dual-feasible functions. It is important to note that the link between the theory of superadditive functions and dual-feasible functions also helps improving cutting planes algorithms by enforcing some cuts with maximal functions. Therefore, we give a tool for improving many previous methods in a straightforward manner.

Successfully generalizing the concept of dual-feasible function for several packing problems hints that this methodology can be applied to problems that lie outside the field of C&P problems. We plan to study and analyze such solutions for new problems. We will focus on problems for which set-covering models and column generation give good lower bounds (variants of vehicule routing or staff scheduling problems for example). An interesting fact is that the definition of SC-DFF can directly be applied to a large variety of problems. One of the main difficulties is to handle the set of constraints defined in the subproblem, which can be much more complicated than the

one-dimensional knapsack problem involved in the cutting-stock problem. Characterizing the set of feasible patterns is also a challenging issue.

Mixing constraint-programming and OR techniques for solving rectangle placement problems

The work of this chapter has been published in an international journal [32] and in an international conference [34] (an extended version is submitted to INFORMS Journal on Computing).

3.1 Rectangle placement problems

When the two-dimensional bin-packing problem is addressed, verifying if a given subset of items can be packed into a bin is NP-complete (whereas for the one-dimensional case the answer is straightforward). This problem has been addressed under several names. In the following, we will use the term rectangle placement problem. Figure 3.1 is an example of solution for an instance of RPP with 12 items.

Problem 9 (Rectangle Placement Problem (RPP)) *Given a list of rectangular items, and a unique large rectangle, is it possible to pack all the items into the rectangle without overlapping?*

RPP not only occurs as a subproblem in two-dimensional packing problems (bin-packing or knapsack problems), but also alone when rectangular pieces of steel, wood, or paper have to be cut from a larger rectangle, and in many industrial applications (VLSI design for example).

When RPP has to be repeatedly solved in a more general optimization problem, researchers have focused on *avoiding* to solve this problem (see [19] for the knapsack problem or [30] for the bin-packing problem). For example, bounds based on DFF can be useful to detect non-feasible problems. As we will see in this chapter, considerable practical difficulties arise in cases where solving RPP is unavoidable.

Different variants of RPP have been studied in the literature. In this document, we focus on the regular ("unconstrained" rectangle packing problem), and on a classical variant called *guillotine-cutting problem*. In this specific case of RPP, the rectangles

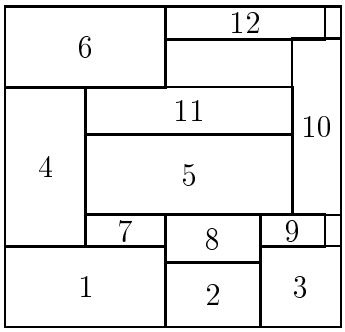


Figure 3.1: A solution for an instance of RPP

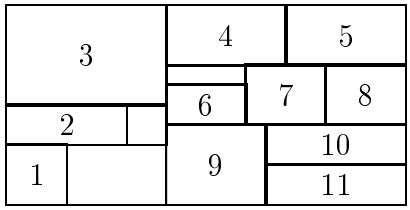


Figure 3.2: A guillotine pattern

have to be cut using *guillotine cuts* only. A guillotine cut is parallel to one of the sides of the rectangle, and must go from one edge all the way to the opposite edge of a currently available rectangle.

Problem 10 (Guillotine Cutting Problem (GCP)) *Given a list of rectangular items, and a large rectangle, is it possible to cut the items from the large rectangle without overlapping, and using only guillotine cuts?*

Figure 3.2 pictures an example of guillotine pattern. A first cut separates items 1, 2 and 3 from the others, then a cut separates 4 and 5 from items 6 to 11 and so on. Note that the pattern of Figure 3.1 is not guillotine, since no set of items can be separated from the others without cutting an item.

Although GCP is considered as a "cutting" problem, the guillotine constraint can also be relevant when one needs to pack items on shelves. Actually we have met this particular problem during an industrial contract on an automatic storage device.

Surprisingly enough, although its combinatorial structure seems easier (with a nice recursive pattern), GCP is harder to solve in practice than RPP for both heuristic and exact methods. For the former, most of the researchers have restricted the search space to so-called *two-stage patterns*, where the recursive structure is limited to a depth of two (strips are cut from the large rectangles, and then each strip is cut to obtain the final items). There are also works about three-stage patterns. These variants are popular because they are relevant from a practical point of view, and can be modelled

as effective ILP (see [76] for example). In our work we do not restrict our patterns to be two-stage.

RPP has been the topic of many research papers, in the OR literature under the name of *feasibility problem* [79,82], or *orthogonal packing problem* (in [43–45] for example), and in the constraint programming community [9,10,54,67]. In the CP community, RPP is much more addressed than GCP. This can be explained by the fact that the structure of GCP is far to be straightforward to capture in a CP model, and that most of the propagation algorithms are only effective when many rectangles have already been packed.

For both versions (RPP and GCP), our contributions are twofold: they concern new models and hybridization of CP and OR techniques based on these models. Our new models are able to capture effectively the structure of the problems addressed, but it transpires that an effective use of OR techniques is mandatory to be able to prune solutions and fasten the search.

For RPP (Section 3.3), we have exploited the fact that the problem is tightly linked to a classical cumulative scheduling problem. Our model creates two scheduling problems in addition to the original problem. This allows us to adapt several methods from the scheduling field to the packing field (energetic reasoning, ...).

For GCP (Section 3.4), we have proposed a brand new graph model, which captures the recursive structure of a guillotine pattern. This model leads to a CP algorithm, which uses the graph to check the guillotine constraint, and our RPP model to verify that the items are packed into the boundaries of the large rectangle.

3.2 Constraint programming

Constraint programming is a paradigm aimed at solving combinatorial problems that can be described by a set of variables, a set of possible values for each variable, and a set of constraints between the variables.

The set of possible values of a variable V is called the *variable domain*, denoted as $D(V)$. It might be, for example, a set of numeric or symbolic values $\{v_1, v_2, \ldots, v_k\}$, or an interval of consecutive integers $[\alpha..\beta]$. In the latter case the lower bound of $D(V)$ is denoted as $V^- = \alpha$ and the upper bound is denoted as $V^+ = \beta$.

A constraint between variables expresses which combinations of values for the variables are allowed. The question is whether there exists an assignment of values to variables, such that all constraints are satisfied. The power of the constraint programming method lies mainly in the fact that constraints can be used in an active process termed "constraint propagation" where certain deductions are performed, in order to reduce computational effort. Constraint propagation removes values from the domains, deduces new constraints, and detects inconsistencies.

Constraint propagation alone is rarely sufficient to solve hard problems. The

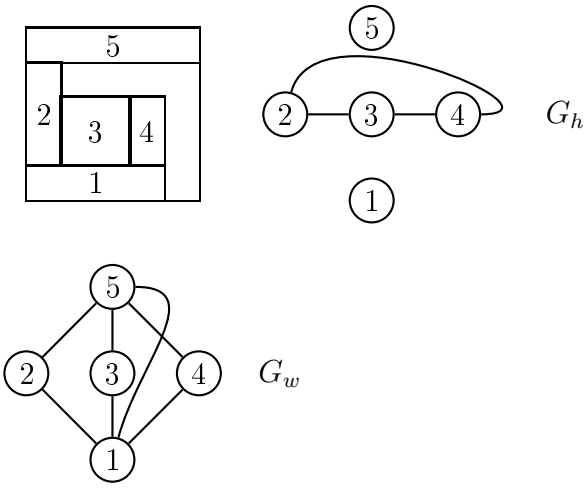


Figure 3.3: Modeling a rectangle placement with two interval graphs

constraint propagation algorithms are generally run at each node of an enumerative method.

3.3 The rectangle placement problem (RPP)

The first OR methods for the rectangle placement problem consist in packing items one by one in the bin [55, 79]. They rely on the so-called *bottom-left dominance rule* (see [26]), which states that each item can be packed in a leftmost and downward corner. Each item is either adjacent to another item, or a side of the bin.

In [43,45], Fekete *et al.* propose a new model for the feasibility problem. They show that a pair of interval graphs can be associated with any *packing class* (*i.e.*, a set of packings with common properties). The interest of this concept is that a large number of symmetries are removed, since only one packing is enumerated per class. A graph $G_d = (I, E_d)$ is associated with each dimension (I is the set of items) and $d \in \{w, h\}$ is the dimension considered. An edge is added in the graph G_w (respectively G_h) between two vertices i and j if the projections of items i and j on the horizontal (respectively vertical) axis overlap (see Figure 3.3). They also provide a branch-and-bound [45] to seek a pair of interval graphs with suitable properties, and then deduce a feasible packing. In comparison with classical methods, this method avoids a large number of redundancies, and outperforms the best previous OR method [79].

The model of Fekete and Schepers has been improved by considering a better representation of interval graphs (using so-called consecutive 1 matrices [62] or PQ-trees [63]). Note that, even with these refinements, the methods based on interval graphs are still outperformed by state-of-the-art CP methods. The CP community has studied the RPP with several different models. In the first, the decision variables are

the relative placement of each pair of items i and j (see [67,82] for example). Beldiceanu and Carlsson [9] have shown that with a good propagation (performed by the SWEEP algorithm), the non-overlapping constraints alone can lead to good results [11]. The branching scheme consists in testing each possible position in the bin for each item in turn.

Our work on rectangle placement lies between the OR and CP communities. We use a constraint-programming scheme, which is based on scheduling techniques, and add several pruning procedures based on OR techniques (resolution of small knapsack problems, and original applications of dual-feasible functions).

3.3.1 A constraint-based scheduling model for RPP

We now describe our constraint-based scheduling model for RPP. We first recall a classical model for the non-overlapping constraint, and then explain how it can be relaxed into a scheduling problem. This leads to a model that uses so-called cumulative constraints in addition to the non-overlapping constraints.

3.3.1.1 A basic constraint programming model

In constraint programming, RPP can be classically encoded in terms of variables and constraints. Two variables X_i and Y_i are associated with each item i. They represent the coordinates of i in the bin. We denote as $D(X_i) = [X_i^{min}, X_i^{max}]$ and $D(Y_i) = [Y_i^{min}, Y_i^{max}]$, respectively the domains of variables X_i and Y_i, in which X_i^{min}, X_i^{max}, Y_i^{min} and Y_i^{max} are the lower and upper bounds of the domains. Initially, the domains of these variables are respectively set to $[0, \ldots, W - w_i]$ and $[0, \ldots, H - h_i]$. For each pair of items i and j, we associate the following constraint: $[X_i + w_i \leq X_j]$ or $[X_j + w_j \leq X_i]$ or $[Y_i + h_i \leq Y_j]$ or $[Y_j + h_j \leq Y_i]$, which expresses the fact that items i and j cannot overlap in the bin.

This model is sufficient to ensure that the solution is valid, once the domains of variables have been reduced to only one value such that all constraints are satisfied. We will refer to this model as the "basic model". Generally, this model is practically ineffective to solve the problem. However, note that the use of the "sweep" algorithm of Beldiceanu and Carlsson [9], which propagates efficiently the above-described constraint, gives competitive results.

3.3.1.2 A new relaxation into a scheduling model

In a previous work, we considered a new relaxation for the RPP. We remove the constraints related to the height of the items and replace them by a classical cumulative constraint. Practically speaking, it means that several horizontal strips of an item are allowed not to be contiguous on the vertical axis. Each solution of the relaxed problem

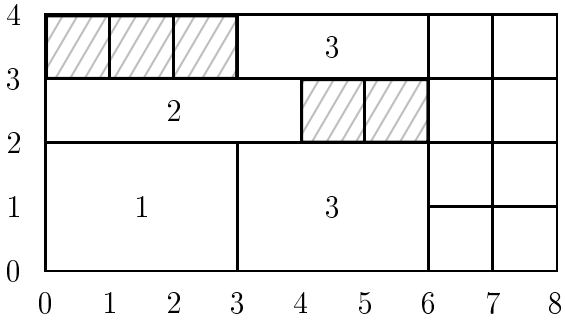


Figure 3.4: A relaxation of RPP into a CuSP

corresponds with a set of patterns for RPP. If there is no solution for the relaxed problem, there is no solution for RPP. A partial solution of the relaxed problem is pictured in Figure 3.4. This problem is known in the scheduling community as the *cumulative scheduling problem* (CuSP).

Problem 11 (Cumulative scheduling problem (CuSP)) *We are given a set of n activities $\{A_1, \ldots, A_n\}$ and a set of resources $\{R_1, \ldots, R_m\}$. Each activity A_i has a processing time, requires a particular amount of a resource R_k and has to be executed within a time window $[est_i, let_i)$. Resources have a given capacity that cannot be exceeded at any point in time. The resource can execute several activities, provided that the resource capacity is not exceeded. The problem to be solved consists in deciding when each activity is executed, while respecting the resource constraints, and without interruption.*

In the following, we show how this relaxation is used to enforce our CP model, and how constraint-based scheduling algorithms can be adapted to our models.

3.3.1.3 A constraint-based scheduling model

The relaxation described above can be applied to the width instead of the height. In this case, another CuSP is obtained. Actually, our method uses both CuSP (one for each dimension) to strengthen the original model.

We now describe formally the two CuSP addressed. The two considered resources are termed R_w and R_h. The resource capacity of R_w is equal to H and the resource capacity of R_h is equal to W. We define a set $\{A_1^w, \ldots, A_n^w\}$. Each activity A_i^w has a processing time w_i, requires an amount h_i of the resource R_w, and has to be executed within the time window $[0, W)$. Similarly, we define a set $\{A_1^h, \ldots, A_n^h\}$ of activities. Each activity A_i^h has a processing time h_i, requires an amount w_i of the resource R_h and has to be executed within the time window $[0, H)$.

We introduce a variable $start(A)$ for each activity A, representing the start time of A. Initially, the domain of variables $start(A_i^w)$ is set to $[0, \ldots, W - w_i]$ and the domain

of variables $start(A_i^h)$ is set to $[0, \ldots, H - h_i]$. Resource constraints represent the fact that activities require some amount of a resource throughout their execution. In our non-preemptive cumulative case, the resource constraints can be expressed as follows:

$$\forall t \in [0, \ldots, W], \sum_{A_i^w / start(A_i^w) \leq t < start(A_i^w) + w_i} h_i \leq H.$$

$$\forall t \in [0, \ldots, H], \sum_{A_i^h / start(A_i^h) \leq t < start(A_i^h) + h_i} w_i \leq W.$$

In other words, the sum of resource requirement of activities A_i^w (respectively A_i^h) executed at time t has to be lower than or equal to the resource capacity H (respectively W) of resource R_w (respectively R_h).

Finally, the scheduling problem is linked to the constraint programming model of the original RPP (see above) with the following constraints: for each item i, $[start(A_i^w) = X_i]$ and $[start(A_i^h) = Y_i]$. It is easy to see that once the variables $start(A_i^w)$ and $start(A_i^h)$ are instantiated in such a way that all constraints are satisfied, the corresponding solution is valid.

To describe our branch-and-bound algorithm, we use the $start(A_i^w)$ and $start(A_i^h)$ variables. We use a schedule-or-postpone method, which works as follows: at each step of the procedure, we choose an unscheduled activity and we schedule it as early as the previous activities scheduled on the same resource will allow. We obtained the best experimental results by working first on a given resource exclusively and then on the other. Note that our work has now been extended by [54], who used the same model, with an improved branching scheme based on the same ideas (in fact they use dichotomy instead of testing each value of X in turn).

Since RPP has been modeled as a scheduling problem, it is now possible to use powerful constraint-based scheduling propagation techniques specific to non-preemptive scheduling problems (see for instance [6]). These techniques allow us to tighten the domains of variables and to detect inconsistencies during the procedure. However, note that edge-finding propagation techniques [6] were not useful for our problem.

3.3.2 Two-dimensional energetic reasoning

We now describe the concept of energetic reasoning, originally developed by Erschler *et al.* [41, 74] to solve cumulative scheduling problems. We suggest a generalization of energetic reasoning, which allows the feasibility of orthogonal packing patterns to be tested, and new adjustments to be found.

3.3.2.1 Feasibility tests and bounds adjustments

For scheduling problems, deductions made using energetic reasoning are based on the consumption of resources by activities during given time intervals. For a given time

interval $[\alpha, \beta)$, $\alpha < \beta$, energy is supplied by a resource and consumed by an activity. The energy supplied by a resource of capacity C in this interval is equal to $(\beta - \alpha) \times C$, and the energy consumed by an activity of demand c_i is equal to $c_i \times \Delta_i$, where Δ_i is the part of activity i scheduled in $[\alpha, \beta)$. If the starting time of the activity is not yet fixed, we determine the mandatory energy consumption in interval $[\alpha, \beta)$. It is obtained by considering the positions in which the processing of the activity is minimal in $[\alpha, \beta)$. By considering the quantities of energy supplied and consumed within given intervals, the energetic approach aims at developing satisfiability tests and time-bound adjustments to ensure that either a given schedule is not feasible or to derive some necessary conditions that any feasible schedule must satisfy.

Unlike activities in scheduling problems, the position of an item has to be fixed with respect to both the horizontal and vertical dimensions. We therefore suggest the following generalization of energetic reasoning. Instead of considering an interval $[\alpha, \beta)$, we consider a rectangular window. We define the rectangular window $[\alpha, \beta, \gamma, \delta)$, $\alpha < \beta$ and $\gamma < \delta$, in which $[\alpha, \beta) \times [\gamma, \delta)$ is the area under consideration.

Energy is now supplied by the bin and consumed by items. Energy supplied by the bin in window $[\alpha, \beta, \gamma, \delta)$ is equal to $(\beta - \alpha) \times (\delta - \gamma)$. The energy consumed by an item can be computed considering the bottom left and top right positions according to the domains of its coordinate variables, in which the item's consumption is minimal (see Figure 3.5). Let $\widehat{w}_i(\alpha, \beta)$ and $\widehat{h}_i(\gamma, \delta)$ be respectively the width and the height of the mandatory part of item i in the window $[\alpha, \beta, \gamma, \delta)$ (see Figure 3.5). We have:

$$\widehat{w}_i(\alpha, \beta) = \max\left(0, \min\left\{w_i, \beta - \alpha, X_i^{min} + w_i - \alpha, \beta - X_i^{max}\right\}\right)$$

and

$$\widehat{h}_i(\gamma, \delta) = \max\left(0, \min\left\{h_i, \delta - \gamma, Y_i^{min} + h_i - \gamma, \delta - Y_i^{max}\right\}\right).$$

Energy consumed by item i in window $[\alpha, \beta, \gamma, \delta)$ is then $\widehat{E}_i(\alpha, \beta, \gamma, \delta) = \widehat{w}_i(\alpha, \beta) \times \widehat{h}_i(\gamma, \delta)$. Therefore, the total energy consumed by all items in window $[\alpha, \beta, \gamma, \delta)$ is $\widehat{E}(\alpha, \beta, \gamma, \delta) = \sum_{i \in I} \widehat{E}_i(\alpha, \beta, \gamma, \delta)$. As in basic energetic reasoning, the following proposition holds.

Proposition 3.3.1 *If there is a feasible packing, then* $\forall \alpha, \beta \in [0, W)$, $\forall \gamma, \delta \in [0, H)$, *such that* $\alpha < \beta$ *and* $\gamma < \delta$, *we have* $\widehat{E}(\alpha, \beta, \gamma, \delta) \leq (\beta - \alpha) \times (\delta - \gamma)$.

This means that for every possible window, energy supplied by the bin has to be at least as large as minimal energy consumed by items. To perform feasibility tests, we can test this inequality at each node of the search tree algorithm for all relevant windows in the bin. If there exists a window for which the inequality does not hold, then the considered node cannot lead to a feasible solution and can consequently be pruned.

The values of $\widehat{E}(\alpha, \beta, \gamma, \delta)$ can also be used to adjust domain variable bounds of X_i and Y_i. Let i be an item and let $[\alpha, \beta, \gamma, \delta)$ be a window such that $\beta < X_i^{max} + w_i$. We

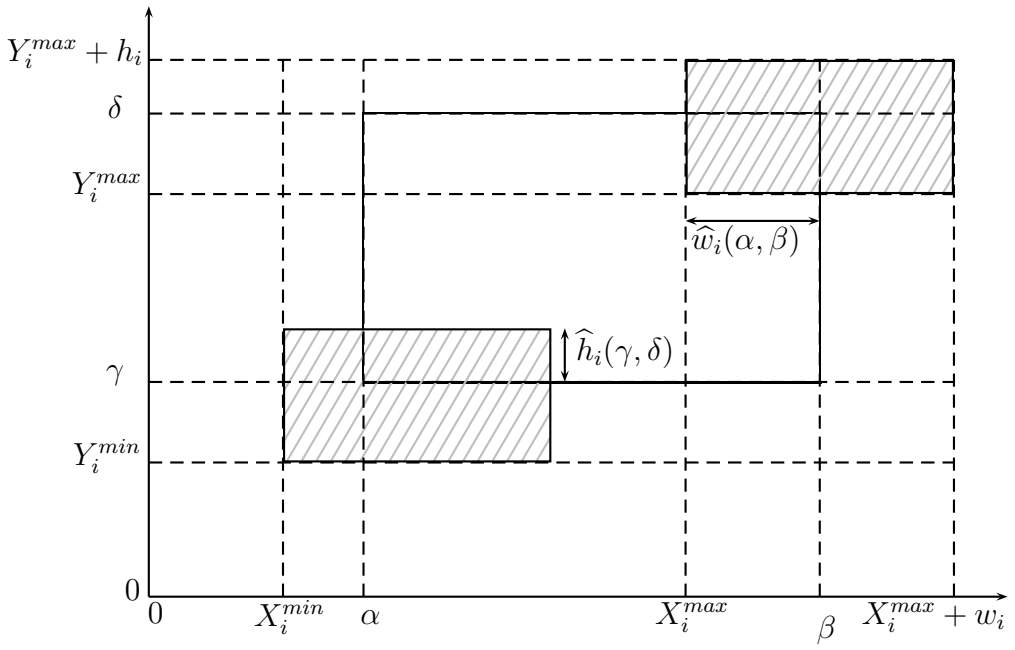


Figure 3.5: Mandatory part of item i in the rectangle $[\alpha, \beta, \gamma, \delta)$ by considering its two extreme positions.

verify whether i can be fully packed before β, *i.e.*, if i can be packed at a coordinate X_i such that $X_i + w_i \leq \beta$. If i is fully packed before β, then its energy consumption is obtained by considering its leftmost position. If this total energy consumption is greater than $(\beta - \alpha) \times (\delta - \gamma)$, it means that item i cannot be fully packed before β. In this case, we update the domain of X_i to take into account this fact.

Baptiste *et al.* [7] studied the cumulative scheduling problem and showed that it is sufficient to calculate energies for intervals belonging to a characterized set in order to find all possible deductions. We generalized their result to the two-dimensional case. Practically speaking, only a subset of possible intervals are used, since using all of them is too time-consuming.

3.3.3 Using DFF in feasibility tests

DFF have been defined in details in Chapter 2. In this section, we focus on the transformations applied to the partial instances of RPP to allow an effective usage of DFF. The bounds applied on these instances are described in details in Chapter 2. Note that when DFF are applied to the instance, a lower bound for the bin-packing problem is obtained. If this value is greater than one, then the instance has no solution.

The problem with classical DFF is that they cannot take into account the fact that some items are already packed. If DFF are applied on a partial solution, the value returned will be the same as initially. Consequently, we operate some modifications on the instance, based on the position of the packed items, to allow the DFF to produce

better pruning methods.

In the first technique, we use DFF to improve the feasibility test based on energetic reasoning. It relies on the fact that a small $2OPP$ instance is created when energetic reasonings are used. In the second technique, we merge items that have been packed to create a more constrained instance for which DFF may be able to show that there is no solutions.

3.3.3.1 Using BP-DFF in Energetic Reasoning

Consider virtual items $(\widehat{w}_i(\alpha,\beta),\widehat{h}_i(\gamma,\delta))$ obtained from the widths and the heights of items in window $[\alpha,\beta,\gamma,\delta)$. The set of items corresponding to the mandatory parts of the items of I in $[\alpha,\beta,\gamma,\delta)$ is denoted by $\widehat{I}(\alpha,\beta,\gamma,\delta)$.

Proposition 3.3.2 *Let (I,B) be a RPP. For four integer values α,β,γ and δ, and a domain variable for the x- and y-coordinates of the items of I, if the $2OPP$ problem defined by $\widehat{I}(\alpha,\beta,\gamma,\delta)$ and $\widehat{B}=(\beta-\alpha,\delta-\gamma)$ has no solution, then there is no solution for the original RPP with the current domain variables.*

Several methods can be used to show that a particular $2OPP$ problem obtained is not feasible, the quickest being to check that the continuous lower bound does not exceed one. This corresponds to the feasibility test described in previous section. In our case, we use the lower bounds based on DFF and described in Chapter 2. We could use preprocessing methods or an exact method. Nevertheless, experimentation has shown that the computation time required for this latter method is too great with respect to the reduction of the search space.

3.3.3.2 Using BP-DFF on a constrained instance

The idea is to aggregate the items which are packed side-by-side to create new instances which are more constrained than the initial instance (Figure 3.6). The lower bounds and the reduction procedures are applied to these instances to obtain better results. The method is based on a geometric observation. Consider the polygon ψ formed by the set I_1 of items packed in the bin. If I_1 is replaced in the initial instance D by another set I_1' such that items of I_1' can be packed in ψ, the following property holds:

Proposition 3.3.3 *If there is no feasible solution for $I \setminus I_1 + I_1'$ in B, then there is no feasible solution for I in B such that items of I_1 are packed in ψ.*

Given a set of packed items I_1, we create a set I_1' which is more constrained than I_1. The idea is to maximize the height or the width of the created items to obtain two instances (Figure 3.6). If several transformed items which are packed side by side are shallow they can be packed one above the other in the new instance. To avoid this situation we operate a second modification on the items. If the packed items have been

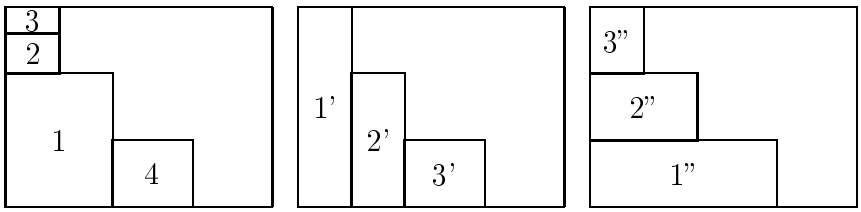


Figure 3.6: Computing suitable instances for applying the DFF

cut into vertical strips, we add to the new items a height equal to H and the height of the bin is updated to $2H$. So the bounds can take into account the fact that these items cannot be packed one above the other. If the packed items do not fit the width of the bin a dummy item with size (w^*, H) is created, w^* being the free width to the right of the packed items (Figure 3.6). Note that after the second transformation, item $3'$ cannot be packed above item $2'$. The same operation can be realized when the width is considered. We denote the new instance obtained as D''. The results are improved because the problem is more constrained as the number of large items increases.

3.3.4 Knapsack-based feasibility tests

We now show how reasoning can be done for pruning the search tree using the solution of *subset-sum* problems. The subset-sum problem is a particular case of the knapsack problem (Problem 2), where the size of each item is equal to its profit. It has been used by Boschetti and Mingozzi [13] in a preprocessing, and by Fekete and Schepers [44, 45] in their exact method.

In our method, the main idea is to make use of the information that is given by the domains of the variables in our constraint programming model. Similarly to energetic reasoning, our algorithm allows to prune partial solutions, but also to realize adjustments on the domains of the variables. An important part of our work is to avoid computing non-necessary information to realize our tests, using dominance rules.

Let P be a partial solution for a RPP instance, and x a given x-coordinate. We denote by H_x the sum of the heights of the items whose variable has been fixed ($X_i^{min} = X_i^{max}$) and such that $x \in [X_i, X_i + w_i)$. If $H_x < H$, additional items can be packed in $[x, x+1)$. However, if there is no unpacked item j such that $h_j < H - H_x$, then necessarily some area has been lost. Consequently, the value $H - H_x$ can be added to the total area of the items to strengthen any reasoning based on the remaining area.

This idea can be generalized when items can be packed in $[x, x+1)$ without the possibility of perfectly fitting the free space. Let I_x be the subset of items i whose variable X_i has not been fixed ($X_i^{min} < X_i^{max}$) and such that $x \in [X_i^{min}, X_i^{max} + w_i)$, *i.e.*, a piece of i could be packed in $[x, x+1)$. We want to determine the minimum

height loss in interval $[x, x+1)$. This can be done by solving a classical one-dimensional knapsack problem, where the size of the bin is $H - H_x$, the set of items I_x, each of size and profit h_i.

We finely tuned this method by: 1) considering intervals instead of all coordinates (time optimization); 2) performing all possible deductions; 3) taking into account the mandatory parts of the items.

3.3.5 Computational experiments: a synthesis

The main conclusions that can be drawn from our computational experiments is that including the cumulative constraints in the model and giving a priority in the branching scheme for one of the two dimensions is sufficient to lead to competitive results. These conclusions have been confirmed later by [54] in their computational experiments.

The two-dimensional energetic reasoning allows some additional deductions, but the computing time is in general too large compared to its effectiveness. This is due in part to our branching scheme, which first works on one dimension. However, for some instances, the computing time is reduced, in particular instances for which some solutions of CuSP do not have a corresponding RPP solution.

All our new feasibility tests reduce both the number of explored states and the computing time. The best compromise between the reduction of the search space and the time required seems to be the method used with one-dimensional energetic reasoning and subset-sum reasoning. The size of the bin is small in the instances we used. For larger bins, the subset-sum based methods have to be avoided, since their computing time becomes too large.

We have compared the best previous algorithms in the literature with our method [9,45]. For this purpose, we used difficult benchmarks with up to 20 items. Our method, using improving techniques, dramatically reduces the search space in comparison with all previous algorithms. Even in the absence of improving techniques, we are almost competitive with the previous approaches. All instances can now be solved in less than seven seconds, unlike the previous results, where some instances cannot be solved within one hour.

3.4 The guillotine-cutting problem (GCP)

In the literature to date we find two alternative methods for solving the guillotine cutting problem (see [59]). The first approach [25] consists in iteratively cutting the bin into two rectangles, using horizontal or vertical cuts, until all the required rectangles are obtained. The second approach [88] recursively merges items into larger rectangles, using so-called horizontal or vertical *builds* [89]. The most recent work on the subject is by Bekrar *et al.* [8], and provides an adaptation of the branch-and-bound method of

Martello *et al.* [77]. An adaptation of an RPP algorithm to the GCP is also proposed by Amossen and Pisinger [3].

To our knowledge, the only existing CP procedures are based on methods that check algorithmically at each node if the packed rectangles violate the guillotine constraint. Our implementation of this method did not lead to interesting results.

In this document, we use another approach. We propose a new graph-theoretical model for GCP. A first idea is to use a tree to represent a pattern. When a solution is found, it can indeed be modeled as a tree, where the leaves correspond with items and the inner vertices to cuts. This is a suitable representation for a final solution, but we considered that it was not suitable for building a solution, since the number of vertices is not known in advance, and represent different kinds of objects (cuts or items).

Another way of modeling a guillotine pattern with a graph would be to adapt the model of [43], using algorithms to detect whether the guillotine constraint is satisfied. This is not how we have chosen to proceed. Instead, we propose a novel graph-theoretical model that takes into account the specific combinatorial structure of the guillotine-cutting problem.

We first describe the new concept of *guillotine-cutting classes*, which models equivalent patterns for the GCP. Then we describe our new arc-colored oriented graph model, and show the equivalence between finding a suitable graph and finding a feasible solution for GCP. Finally, we study the combinatorial structure of our model, obtained by removing colors and directions of the arcs. These non-directed multi-graphs have a special structure, which is used to design efficient algorithm to recognize them and computing the patterns associated with them. Finally, we describe roughly the CP approach based on our model, and comment our computational experiments, which show that our model allows to improve previous results by a wide range.

3.4.1 Guillotine-cutting classes

In order to avoid equivalent patterns in the RPP, Fekete and Schepers [43] proposed the concept of *packing class*. Packing classes are general, and can model any pattern. When only guillotine patterns are sought, packing classes may not be suited to the problem, since two different packing classes may give rise to patterns having the same combinatorial structure. We introduce the concept of *guillotine-cutting class* to include all guillotine patterns. This takes into account the fact that exchanging the positions of two rectangular blocks of items does not change the combinatorial structure of the solution. The definition uses the notion of *builds* that we define below.

A build [89] involves creating a new item by combining two other items (see Figure 3.7). The result of a horizontal build of two items i and j, denoted $build(i, j, horizontal)$, is an item labeled $min\{i, j\}$, of width $w_i + w_j$ and height $\max\{h_i, h_j\}$. A vertical build is defined similarly.

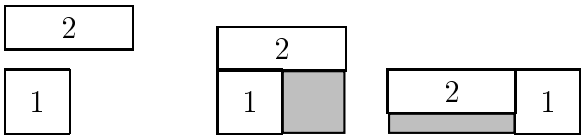


Figure 3.7: Vertical and horizontal builds of two items 1 and 2.

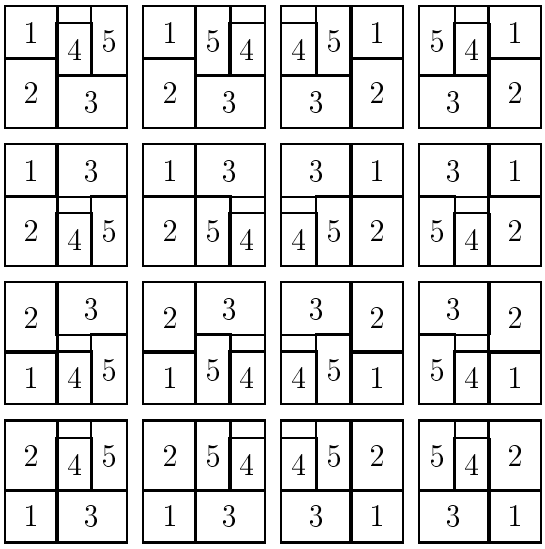


Figure 3.8: A guillotine-cutting class

Definition 3.4.1 *Two solutions belong to the same* guillotine-cutting *class if they can be obtained from a same sequence of horizontal and vertical builds.*

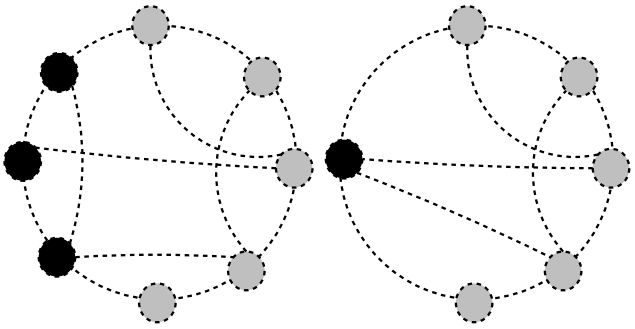
Figure 3.8 shows a guillotine-cutting class. Clearly, if one member of a guillotine-cutting class is feasible, then so are the other members. In this case we say that the guillotine-cutting class is feasible. This concept reduces dramatically the number of equivalent patterns in comparison with a direct application of the model of [43]. This is hardly surprising, since the concept of packing classes was not designed for this specific problem.

Note that there still remain redundancies. Two different sequences of builds may lead to solutions with the same combinatorial structure, for example when there is a partial pattern that can be obtained with either a horizontal or a vertical first cut. This means that a given pattern may belong to several guillotine-cutting classes.

3.4.2 A new graph-theoretical model

We study a new class of directed and arc-colored graphs, and show that such graphs can be associated with guillotine-cutting classes. We call these graphs *guillotine graphs*.

In order to define this new class, we introduce the concept of *circuit contraction*, analogous to the classical concept of arc contraction used in graph theory.



(a) initial graph (b) after cycle-contraction

Figure 3.9: Cycle-contraction

Definition 3.4.2 *Let $G = (V, E)$ be a graph, and $\mu = [v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}]$ a cycle of G. Contracting μ is equivalent to iteratively contracting each edge of μ.*

When referring to an undirected graph we use the term *cycle-contraction*. The same concept can be applied to directed graphs, in which case we use the term *circuit-contraction*. In Figure 3.9, contracting the black cycle in the left-hand graph leads to the right-hand graph. The index of the vertex obtained by contracting a circuit μ is the smallest index of an item in μ.

In our new model, a vertex is associated with each item i, and a circuit is associated with a list of horizontal or vertical builds. Let $G = (I, A)$ be a directed graph. We use a concept of arc coloring defined as follows. An arc coloring of a graph G is a mapping ξ from A to a set of k colors. In order to distinguish between horizontal and vertical builds, we equate horizontal builds with the color red, and vertical builds with the color green. Thus in this document we focus on bicoloring (and arc-bicolored graphs), *i.e.,* we consider a mapping from A to $\{red, green\}$.

We say that a circuit is *monochromatic* if all arcs of the circuit have the same color. In the graph, circuit-contracting a red (resp. green) circuit corresponds to a list of horizontal (resp. vertical) builds. When a circuit μ is contracted, the size associated with the residual vertex is the size of the item built, and its label is the smallest vertex label in μ. We now give a definition of *guillotine graphs*, which model guillotine patterns.

Definition 3.4.3 *Let G be an arc-bicolored directed graph. G is a* guillotine graph *if G can be reduced to a single vertex x by iterative contractions of monochromatic circuits with the following properties:*

1. *there are no steps in which a vertex belongs to two different monochromatic circuits*

2. *when a circuit μ is contracted, either the current graph is a circuit, or exactly two vertices of μ are of degree strictly greater than two.*

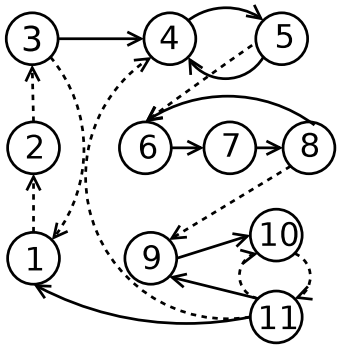


Figure 3.10: Modeling the pattern in Figure 3.2 with a dominant guillotine graph

Note that the definition can be directly generalized for higher dimensions (just by considering k colors instead of two).

Many equivalent graphs can be associated with a given guillotine-cutting class, and so we introduce different levels of dominance for these graphs.

In a *normal guillotine graph*, the two vertices x_i and x_j of degree greater than two in a monochromatic circuit μ are such that x_j follows x_i in μ, x_i cannot be the tail of any arc outside μ, and x_j cannot be the head of any arc outside μ.

Definition 3.4.4 *Let G be a guillotine graph. G is a normal guillotine graph if at any step of the iterative contraction process, in each monochromatic circuit $\mu = (x_1, x_2), \ldots, (x_{k-1}, x_k)$, if there are two vertices x_i and x_j of degree strictly greater than two, then $(x_i, x_j) \in \mu$ and $|N^+(x_i)| = 1$ and $|N^-(x_j)| = 1$.*

In *dominant guillotine graphs*, vertices have to be sorted by increasing index in any circuit.

Definition 3.4.5 *Let G be a normal guillotine graph. G is a* dominant guillotine graph *if in all graphs obtained by applying circuit-contractions to G, vertices in a monochromatic circuit are ordered by increasing index.*

Figure 3.10 shows the dominant graph that models the configuration of Figure 3.2.

Theorem 3.4.1 *If G is a dominant guillotine graph, G can be associated with a unique guillotine-cutting class. Moreover, for each normal sequence of builds, there is exactly one dominant guillotine graph.*

If several normal sequences of builds lead to the same guillotine pattern, then several graphs will be associated with the same pattern. This occurs in cases where a vertical and a horizontal cut produce items of the same size, irrespective of the order in which the two cuts are performed. Handling these symmetries is an issue that can only be done using algorithmic methods. However, it has to be noted that from a practical point of view, these cutting patterns are actually different, since they are related to two different sequences of cuts for an automatic cutting device.

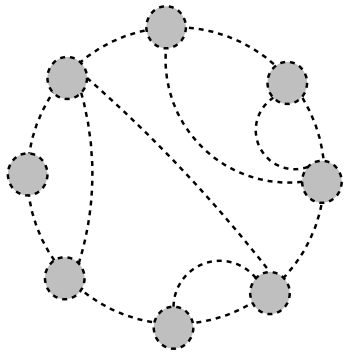


Figure 3.11: A cycle-contractable graph

3.4.3 Cycle-contractable graphs

We now look at the combinatorial structure of our model when colors and orientations are removed. This non-directed and uncolored version of guillotine graphs helps finding linear algorithms for recognizing guillotine graphs, and may be easier to use in heuristics or meta-heuristics. We name *cycle-contractable graphs* these undirected uncolored guillotine graphs.

We first define the cycle-contractable graphs, and show that they are undirected uncolored guillotine graphs.

Let $G = (V, E)$ be an undirected multigraph. If there is a Hamiltonian cycle $\mu = [v_1, v_2, \ldots, v_n, v_1]$, a corresponding ordering σ can be associated with the vertices of V ($\sigma(v_k) = k$) for $k = 1, \ldots, n$. Hereafter, when a graph G has a Hamiltonian cycle, we shall refer to any edge that is not included in the cycle as a *backward edge*.

Definition 3.4.6 *Let $G = (I, E)$ be an undirected multigraph. G is a* cycle-contractable graph *if G contains a Hamiltonian cycle μ with a corresponding ordering σ such that*

1. *G does not include two equivalent backward edges $[i, j]$ and $[i, j]$*
2. *G does not include two backward edges $[i, j]$ and $[k, l]$ such that $\sigma(i) < \sigma(k) \leq \sigma(j) < \sigma(l)$*

The graph in Figure 3.11 is a cycle-contractable graph. It can be depicted as a circle of vertices and non-crossing chords.

In the following, we show that dominant guillotine graphs and cycle-contractable graphs are similar. A first important property is that guillotine graphs contain a Hamiltonian circuit. This result will be used throughout this chapter.

Lemma 3.4.1 *If G is a guillotine graph, it contains a Hamiltonian circuit.*

This allows us to show that dominant guillotine graphs have the structure of cycle-contractable graphs. For this purpose we consider the graph obtained by removing the color and the orientation of the arcs of the considered guillotine graph.

Algorithm 2: Finding the Hamiltonian cycle in a cycle-contractable graph

Data: $G = (V, E)$: multigraph;
$\mu \leftarrow \emptyset$;
$L \leftarrow \emptyset$;
forall *edge that appears twice in* E **do** delete one of the two edges $[v_i, v_j]$;
forall i *such that* $|N(v_i)| = 2$ **do** $L \leftarrow L \cup \{v_i\}$;
repeat
  Let v_i be a vertex in L and let v_j and v_k be its two neighbors;
  $L \leftarrow L \setminus \{v_i\}$;
  if $[v_i, v_j]$ *is not backward* **then** $\mu \leftarrow \mu \cup \{[v_i, v_j]\}$;
  if $[v_i, v_k]$ *is not backward* **then** $\mu \leftarrow \mu \cup \{[v_i, v_k]\}$;
  $G \leftarrow G \setminus \{v_i\}$;
  if $[v_j, v_k] \notin G$ **then** $G \leftarrow G \cup \{[v_j, v_k]\}$;
  mark $[v_j, v_k]$ as *backward*;
  if $|N(v_j)| = 2$ **then** $L \leftarrow L \cup \{v_j\}$;
  if $|N(v_k)| = 2$ **then** $L \leftarrow L \cup \{v_k\}$;
until $n = 3$ *or* L *is empty*;
if $n > 3$ **then** exit with the **FAIL** status;
add each remaining edge in μ if it is not backward;
return μ;

Theorem 3.4.2 *An arc-uncolored undirected guillotine graph is a cycle-contractable multigraph.*

3.4.4 Computing the patterns associated with a cycle-contractable graph

We now propose an algorithm to recognize cycle-contractable graphs. When such graphs are considered, the first step is to determine which edges belong to the Hamiltonian cycle μ, and which edges are backward (*i.e.* edges that do not belong to μ). Algorithm 2 finds the cycle μ in linear time.

The validity of Algorithm 2 below is based on the two following lemmas, which directly induce a recursive algorithm for finding the Hamiltonian cycle if the graph is cycle-contractable. The idea is to remove all "double edges" and then to iteratively delete the vertices of degree two. If there is only one vertex at the end of the process, then the graph is cycle-contractable.

Lemma 3.4.2 *The graph G', obtained from the cycle-contractable graph G by performing one of the two following modifications,*

1. *removing an edge $[v_j, v_k]$ that appears twice in G;*

2. *deleting a vertex v_i of degree two and its two incident edges $[v_i, v_j]$ and $[v_i, v_k]$, and adding an edge $[v_j, v_k]$ if it is not already in the graph.*

is a cycle-contractable graph. Moreover, any edge belonging to the Hamiltonian cycle of G' and to G also belongs to the Hamiltonian cycle of G.

Lemma 3.4.3 *Let G be a cycle-contractable graph. If G has at least three vertices and no cycle of size two, it has at least one vertex of degree two.*

Proposition 3.4.1 *Algorithm 2 finds the Hamiltonian cycle of G if and only if G is cycle-contractable.*

We have shown that a given dominant guillotine graph leads to a unique cycle-contractable graph. In this section we show that a given cycle-contractable graph leads to exactly two guillotine-cutting classes. The first step is to deduce the only possible valid orientation of the edges. Then a choice remains for the coloring of the arcs. The two possible arc-colorings lead to two possible guillotine graphs, and thus to two possible guillotine patterns.

Depending on the ordering of the vertices in the cycles, not all cycle-contractable graphs give rise to dominant guillotine graphs. In order to avoid non-dominant solutions we introduce the *dominant cycle-contractable graphs*, which yield dominant guillotine graphs.

Definition 3.4.7 *Let G be a cycle-contractable graph. If for one of the two possible orientations, for all obtained arcs (x_i, x_j) of the Hamiltonian circuit ($j \neq 1$), then either $i < j$, or there is a backward arc (x_l, x_i) such that $l < j$, G is a dominant cycle-contractable graph.*

Proposition 3.4.2 *A cycle-contractable graph yields a dominant guillotine graph if and only if it is a dominant cycle-contractable graph.*

Algorithm 3 returns true if and only if the input cycle-contractable graph is dominant (checked using Proposition 3.4.2). In this case, the algorithm visits the vertices v of the obtained directed graph following the Hamiltonian circuit σ. Each time a new circuit is entered or leaved, the current color is changed.

Proposition 3.4.3 *Algorithm 3 colors the arcs of a dominant cycle-contractable graph G in such a way that the bicolored graph H obtained is a dominant guillotine graph.*

Corollary 3.4.1 *Given the color of one arc, there is only one valid coloring for a cycle-contractable graph.*

Theorem 3.4.3 *Each dominant cycle-contractable graph is related to two guillotine-cutting classes, and every dominant sequence of builds is related to one dominant cycle-contractable graph.*

Algorithm 3: Orienting and coloring a cycle-contractable graph

Data: $G = (V, E)$: a cycle-contractable graph;
Use Algorithm 2 to determine the backward edges;
Choose an orientation for the edges that is consistent with the hamiltonian cycle;
$test \leftarrow true$;
forall *arc* (v_i, v_j) *of the Hamiltonian circuit* **do**
  if $j < i$ *and* $\nexists k < j$ *s.t.* (v_k, v_i) *is a backward edge* **then** $test \leftarrow fail$;
if $test = fail$ **then**
  choose the other orientation for the edges;
  forall *arc* (v_i, v_j) *of the Hamiltonian circuit* **do**
    if $j < i$ *and* $\nexists k < j$ *s.t.* (v_k, v_i) *is a backward edge* **then** return **false**;
compute the corresponding ordering σ;
choose a color;
for $i : 1 \rightarrow n$ **do**
  $v \leftarrow \sigma(i)$;
  Let S^+ be the set of backward arcs a such that $a = (u, v)$;
  forall $a \in S^+$ **do** change the current color;
  Let S^- be the set of backward arcs a such that $a = (v, u)$;
  foreach *backward arc* a *of* S^- *by decreasing value of label* **do**
    color a with the current color;
    change the current color;
  $u = \sigma(i + 1)$;
  color the arc (v, u) with the current color;
return **true**;

Algorithm 4 computes the width and the height of the guillotine pattern associated with the guillotine graph G. First the ordering σ is computed using Algorithm 2. Then the vertices are considered following σ. Initially a dummy build b is created, with the current item only. When there is a backward arc, the new build associated with the corresponding circuit is computed and stored in b, and then pushed onto the top of S. At the end of the algorithm, S contains only one element, which corresponds to the guillotine pattern.

Proposition 3.4.4 *For a given initial color, Algorithm 4 computes the size of the pattern associated with the guillotine graph G.*

The following theorem summarizes the different complexity results related to cycle-contractable graphs and guillotine graphs.

Proposition 3.4.5 *Let G be a guillotine graph with at least two vertices. The number m of arcs in G is in $[n, 2n - 2]$, and the bounds are tight.*

Algorithm 4: Computing the size of the guillotine pattern related to a guillotine graph

Data: G: a valid guillotine graph;
σ: the corresponding ordering on the vertices ($\sigma(1) = 1$);
Let S be an initially empty stack of builds b_k;
for $i : 1 \rightarrow n$ **do**
    $v_j \leftarrow \sigma(i)$;
    Let b_j be a new build of size $w_j \times h_j$ and of label j;
    foreach *backward arc* (v_j, v_k) *of color* c *by decreasing value of* $\sigma^{-1}(v_k)$ **do**
        repeat
            remove from S its top element b_t;
            $b_j \leftarrow build(b_j, b_t, c)$;
        until b_j *has for label* v_k;
    push b_j on the top of S;
Let b_j be the iterative build of all elements of the stack;
return b_j;

Using this property, we deduce that the algorithms described above take $O(n)$ time and space.

Theorem 3.4.4 *Recognizing a cycle-contractable graph, and computing the two guillotine-cutting classes related to this graph takes $O(n)$ time and space.*

Note that our exact approach below uses the colored and directed version of our model. However, we are planning to use our model in a meta-heuristic and in this case, we will use the uncolored undirected version. Indeed when an arc is added to the graph, it may change the color of many other arcs. When the uncolored version of our model is considered, one just has to use the coloration algorithm described above to compute the new set of colors.

3.4.5 A constraint-programming approach

We designed an exact approach based on our new model for the guillotine-cutting problem. The basic idea of the method is to seek a guillotine graph corresponding to a configuration that fits within the boundary of the input bin. Our model is embedded into a *constraint-programming* scheme, which seeks a suitable set of arcs. The model is composed of two parts: a graph part, which verifies the guillotine constraint, and a rectangle placement part, which verifies that the rectangles can be packed into the bin. For the latter, we use the model described for the RPP.

3.4.5.1 Variables

We now describe how the guillotine-cutting problem can be modeled in terms of two sets of variables and constraints. The first variable set is related to the graph underlying the pattern, to ensure that the configuration is guillotine, while the second is related to geometric considerations, to ensure that the rectangles can be placed within the boundaries of the large rectangle.

The first set of variables is related to the arcs of the guillotine graph to be built. It specifies the *state* of each arc. The state of an arc is determined by its *existence*, its *orientation* (backward or forward) and its *direction* (horizontal or vertical).

Recall that a guillotine graph can be reduced to a single vertex by iterative contractions of monochromatic circuits. Thus, each time such a monochromatic circuit μ is found in the graph under construction, μ is contracted. In order to prevent contracted vertices from being revisited, a state is associated with each vertex, specifying whether or not it has been contracted, and giving its current dimensions. Thus, a vertex i represents an item or an aggregation of items. Its dimensions are either the dimensions of the original rectangle i, or the dimensions of a build of items corresponding to the contractions in the graph.

A valid dominant guillotine graph may lead to a guillotine-cutting class that does not fit within the boundaries of the bin. Consequently we use a second set of variables which represent the coordinates of a specific member of the guillotine-cutting class under construction. We use the model designed for the unconstrained rectangle placement problem (see Section 3.3.1.3).

3.4.5.2 Exploration of the search space

In our method, the branching scheme modifies only the graph variables directly: the values of the geometric variables X and Y are deduced from constraint propagation.

At each node of the search tree an arc must be chosen for possible inclusion in the graph. We use a depth-first strategy giving priority to the inclusion of backward arcs in the current partial Hamiltonian path $\sigma = \sigma_1, \ldots, \sigma_k$. The backward arc (σ_j, σ_i) is selected from among all the possible backward arcs, with j and i respectively the smallest and the largest index. If no backward arc is possible in σ, then σ is expanded by adding a forward arc between σ_k and another vertex.

3.4.5.3 Constraint-propagation techniques

During the search, constraint-propagation techniques are used to reduce the search space by eliminating non-relevant values from the domain of the variables. These techniques perform different deductions: they eliminate potential arcs that cannot lead to a dominant guillotine graph or to a valid solution; they eliminate potential coordinates that cannot lead to a valid solution; they add some arcs that are mandatory

for obtaining a dominant guillotine graph and a valid solution; they update the possible orientations or the backward status of arcs. These techniques are used to adjust the domains of graph variables to the domains of coordinate variables, and vice versa.

3.4.6 Computational experiments: a synthesis

We have compared our methods for GCP with algorithms in the literature, using 25 instances [59] derived from strip-cutting problems. In this problem, the width of the bin is fixed and the minimal feasible height for the bin must be determined. Therefore this problem leads to a set of feasible or unfeasible decision problems. The number of items in these instances is less than 25.

A first remark is that the computing time required by the algorithms is large compared to the time required by our methods to solve the non-guillotine version. In many cases, it is even more interesting to run the RPP solver before. We found that solving the feasibility problems by increasing value of height led to the fastest results. This suggests that our method is better at proving that a problem has no solutions than at finding a feasible solution.

We also compared our methods to the best method of the literature [59]. We were able to solve each test case in less than three seconds. For several instances, the difference in terms of nodes in the search tree is large. For example the IMVB method of [59] needs 40909 branching points to solve an instance, whereas our method needs only 293. On average, our method needs 15 times fewer nodes that the best approach BMVB of [59]. These results show that the additional information added by our model enhances the exploration of branches in a tree search.

3.5 Conclusions, future works

Our experimentations on RPP and GCP confirm that CP techniques cannot be ignored from the OR community when competitive results are sought. Moreover, methodology-wise, we have shown that coupling OR and CP, packing and scheduling techniques was of great interest.

Another strong conclusion is that even if CP models may be efficient for small or medium sizes of problems, they may fail to find a solution for large instances. Clearly, good heuristics and lower bounding procedures (based on DFF for example) remain crucial to avoid running an expensive exact RPP or GCP procedure.

In our future works, we will focus on GCP. It transpires from our experiments that even for medium size instances, exact methods can take a large computing time to find a solution. Another issue is that our CP-based exact method for this problem is hard to implement. Many constraints have to be hand-tailored and cannot be integrated "out of the shelf" in a CP algorithm. We plan to work on a more CP-oriented method based on the same ideas, which would mostly use constraints implemented in most CP

solvers. Using our new graph-theoretical model for designing heuristic methods is also a work in progress. The first works in this direction are already promising.

Conclusions and future work

In this document we have described new models and methods that we applied to various packing problems. The main feature of our work is to use in a collaborative way techniques from the mathematical programming, constraint programming, graphs, dynamic programming and meta-heuristic communities.

Our work on decomposition methods has confirmed the fact that these methods, when applied effectively, are helpful to solve hard combinatorial problems. We first showed that tree-decomposition can be applied successfully on some packing problems. It leads to a generic framework that can be used in many hybrid methods. For example, taking into account the structure and the size of the subproblems, a different exact or approximated method could be used for each cluster of the decomposition. We also studied different ways of helping column generation using heuristics (for generating an initial set of columns and for solving the pricing subproblem). Another conclusion is that strategic oscillation is a suitable tool for solving packing problems in which some patterns are excluded. It allows our meta-heuristics to travel from good solutions to good solutions in a fast manner by relaxing some constraints.

We now plan to focus our work on **hybrid methods**. A first perspective is a tighter collaboration between column-generation methods and meta-heuristics. It has been shown that re-optimization and multiple column generation reduced the computing effort by a wide range. Several questions arise from this statement. What are the wished properties of an initial pool of columns? How meta-heuristics can help applying suitable sets of dual cuts to stabilize column generation? Can multi-objective optimization help generating a suitable set of columns in the pricing phase? Another research path is to design a column-generation scheme based on the tree-decomposition, where the master problem would consist of assigning items to clusters. For packing problems, a direct application would lead to a better linear relaxation, but the pricing subproblem would be harder to solve.

A second perspective on hybrid methods is to study **heuristics based on mathematical programming** (so-called *matheuristics*). We are now designing such a method for the quadratic knapsack problem of Chapter 1, where the role of the local search is played by a mathematical programming based method. We are also working on generic matheuristics based on **pseudo-polynomial formulations**, which are improved in an iterative process. The first experiments on time-dependent formulations are already promising.

Our work on dual-feasible functions and their extensions clearly shows that this concept is useful for many different packing problems. It transpires that the effectiveness of the DFF depends on the constraints added to the original structure of the packing problem. If these constraints weaken the quality of the linear relaxation of the model of Gilmore and Gomory, the bounds obtained using DFF are expected to be weak. Another conclusion is that the difficulty to handle the additional constraints also has a large impact on the efficiency of the method. For the bin-packing with conflicts, for example, the column generation lower bound is strong, but our heuristics are not able to approximate this bound effectively. This is due to the fact that each optimal dual solution is highly data-dependent (because of the graph structure). For the other problems we addressed, the lower bounds are much more effective.

Up to now, techniques based on DFF are only used for computing an initial lower bound (at the root node of a branch-and-price method for example). When additional cuts are added, their results are weakened, and thus no effective exact methods can be based on DFF only. **Focusing on constraints related to cuts and branching constraints** seems to be one of the most challenging and useful research path (this would avoid solving repeatedly huge linear programs). Studying conflicts involving more than two items is the main issue to handle. An effective solution would be to design the branching scheme in such a way that the underlying conflict graph has suitable properties.

Another difficult challenge is to **generalize the DFF to other problems**, such as vehicle routing problems, where the structure of the "patterns" (routes) is more complicated. Our first experiments tend to show that this issue needs a large amount of work before any useful result is sought.

For rectangle placement problems, our work has confirmed that constraint programming is one of the most useful techniques to solve this family of highly combinatorial problems. We stressed the fact that OR techniques, such as DFF, are helpful to quickly determine that a pattern is not feasible, and to prune nodes in a search method. Our graph-theoretical model for the guillotine cutting problem has also proved to be useful in the design of an exact method. It allows to represent and gather patterns in a way that facilitates deductions and pruning during the search.

As a first perspective for this work, we plan to exploit our graph model in heuristic and meta-heuristics frameworks. Our first experiments tend to show that this is a viable approach, when the uncolored undirected model is used. A more CP-oriented implementation, using classical implemented constraints, would also help our work to be used and extended.

We are also studying real-life placement problems, and it transpires from our experience that focusing on the initial placement is not sufficient. One has to be aware that items will be packed and removed repeatedly, deeply modifying the structure of the initial packing pattern. Thus we plan to deal with the **dynamic versions of these problems**, where robustness and real-time re-optimization have to be studied. Exact

methods should be more difficult to apply. However, we believe that hybrid methods based on our packing models will be useful. We are also interested in an extension of our models into a **bi-level** context. We are now conducting a preliminary study on this subject, focusing on reformulations, graph models and dynamic programming schemes.

More generally, we are now applying our optimization techniques to problems that lie outside the field of C&P problems. We have already obtained results on a variant of the **vehicle routing problem, flight scheduling**, and **staff scheduling** problems. In particular, we are studying special cases of **multi-objective** problems, where additional constraints or objectives added by the decision maker involve hard subproblems for which we expect our hybrid resolution techniques to be useful.

References

[1] C. Alves, *Cutting and packing: Problems, models and exact algorithms*, Ph.D. thesis, Universidade do Minho, Portugal, 2005.

[2] C. Alves and J.M. Valério de Carvalho, *A branch-and-price-and-cut algorithm for the pattern minimization problem*, RAIRO Operations Research **42** (2009), no. 4, 435–452.

[3] R. Amossen and D. Pisinger, *Multi-dimensional bin packing problems with guillotine constraints*, Computers and Operations Research **37** (2010), no. 11, 1999–2006.

[4] R. Baldacci and M. Boschetti, *A cutting plane approach for the two-dimensional orthogonal non guillotine cutting stock problem*, European Journal of Operational Research **183** (2007), 1136–1149.

[5] N. Bansal, Z. Liu, and A. Sankar, *Bin-packing with fragile objects and frequency allocation in cellular networks*, Wireless Networks **15** (2009), 821–830.

[6] Ph. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling, applying constraint programming to scheduling problems*, International Series in Operations Research and Management Science, vol. 39, Kluwer, 2001.

[7] Ph. Baptiste, C. Le Pape, and W. Nuijten, *Satisfiability tests and time bound adjustments for cumulative scheduling problems*, Annals of Operational Research **92** (1999), 3305–3333.

[8] A. Bekrar, I. Kacem, C. Chu, and C. Sadfi, *An improved heuristic and an exact algorithm for the 2d strip and bin packing problem*, International Journal of Product Development **10** (2010), no. 1-3, 217,240.

[9] N. Beldiceanu and M. Carlsson, *Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints*, Principles and Practice of Constraint

Programming (CP'2001), Lecture Notes in Computer Science **2239** (2001), 377–391.

[10] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet, *A generic geometrical constraint kernel in space and time for handling polymorphic k -dimensional objects*, Principles and Practice of Constraint Programming - CP 2007, LNCS 4741, 2007, pp. 180–194.

[11] N. Beldiceanu, M. Carlsson, and S. Thiel, *Sweep synchronization as a global propagation mechanism*, Computers and Operations Research **33** (2006), no. 10, 2835–2851.

[12] M. Boschetti, *New lower bounds for the three-dimensional finite bin packing problem*, Discrete Applied Mathematics **140** (2004), 241–258.

[13] M. Boschetti and A. Mingozzi, *The two-dimensional finite bin packing problem. part I: New lower bounds for the oriented case*, 4OR, A Quarterly Journal of Operations Research **1** (2003), 27–42.

[14] ———, *The two-dimensional finite bin packing problem. part II: New lower and upper bounds*, 4OR, A Quarterly Journal of Operations Research **1** (2003), 135–147.

[15] C.A. Burdett and E. L. Johnson, *A subadditive approach to solve linear integer programs*, Annals of Discrete Mathematics **1** (1977), 117–144.

[16] S. Cahon, N. Melab, and E.G. Talbi, *ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics*, Journal of Heuristics **10** (2004), no. 3, 357–380.

[17] A. Caprara, M. Locatelli, and M. Monaci, *Bilinear packing by bilinear programming*, Integer Programming and Combinatorial Optimization, 11th International IPCO Conference, Berlin, Germany, June 8-10, 2005 (Michael Jünger and Volker Kaibel, eds.), Lecture Notes in Computer Science, vol. 3509, Springer, June 8-10 2005, pp. 377–391.

[18] A. Caprara, A. Lodi, and R. Rizzi, *On d−threshold graphs and d−dimensional bin packing*, Networks **44** (2004), no. 4, 266–280.

[19] A. Caprara and M. Monaci, *On the two-dimensional knapsack problem*, Operations Research Letters **32** (2004), 5–14.

[20] J. Carlier, F. Clautiaux, and A. Moukrim, *New reduction procedures and lower bounds for the two-dimensional bin-packing problem with fixed orientation*, Computers and Operations Research **34** (2007), no. 8, 2223–2250.

[21] J. Carlier and E. Néron, *A new LP-based lower bound for the cumulative scheduling problem*, European Journal of Operational Research **127** (2000), 363–382.

[22] ———, *On linear lower bounds for the resource constraint project scheduling problem*, European Journal of Operational Research **149** (2003), no. 2, 314–324.

[23] ———, *Computing redundant resources for cumulative scheduling problems*, European Journal of Operational Research **176** (2007), no. 3, 1452–1463.

[24] W.T. Chan, F.Y.-L. Chin, D. Ye, G. Zhang, and Y. Zhang, *Online bin packing of fragile objects with application in cellular networks*, Journal of Combinatorial Optimization **14** (2007), 427–435.

[25] N. Christofides and E. Hadjiconstantinou, *An exact algorithm for orthogonal 2-d cutting problems using guillotine cuts*, European Journal of Operational Research **83** (1995), 21–38.

[26] N. Christofides and C. Whitlock, *An algorithm for two-dimensional cutting problems*, Operations Research **25** (1977), 30–44.

[27] V. Chvátal, *Edmonds polytopes and a hierarchy of combinatorial problems*, Discrete Math. **4** (1973), 305–337.

[28] F. Clautiaux, C. Alves, and J. Valério de Carvalho, *A survey of dual-feasible functions for bin-packing problems*, Annals of Operations Research (to appear).

[29] F. Clautiaux, C. Alves, and J.M. Valério de Carvalho, *New ways of deriving dual cuts for the cutting-stock problem*, accepted in INFORMS Journal on Computing (2010).

[30] F. Clautiaux, J. Carlier, and A. Moukrim, *A new exact method for the two-dimensional bin-packing problem with fixed orientation*, Operations Research Letters **35** (2007), no. 3, 357–364.

[31] F. Clautiaux, M. Dell'Amico, M. Iori, A. Khanafer, and E.-G. Talbi, *The bin-packing problem with fragile objects*, Research report (2010).

[32] F. Clautiaux, A. Jouglet, J. Carlier, and A. Moukrim, *A new constraint programming approach for the orthogonal packing problem*, Computers and Operations Research **35** (2008), no. 3, 944–959.

[33] F. Clautiaux, A. Jouglet, and J. El Hayek, *A new lower bound for the non-oriented two-dimensional bin-packing problem*, Operations Research Letters **35** (2007), no. 3, 365–373.

[34] F. Clautiaux, A. Jouglet, and A. Moukrim, *A new graph-theoretical model for k-dimensional guillotine-cutting problems*, Experimental algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May/June 2008, LNCS, vol. 5038, 2008.

[35] F. Clautiaux, A. Moukrim, and J. Carlier, *Improving lower bounds for a two-dimensional bin-packing problem by generating new data-dependent dual-feasible functions*, International Journal on Production Research **47** (2009), no. 2, 537–560.

[36] F. Clautiaux, A. Moukrim, S. Negre, and J. Carlier, *Heuristic and meta-heuristic methods for computing graph treewidth*, RAIRO Operations Research **38** (2004), 13–26.

[37] E. G. Coffman, M. R. Garey, and D. S. Johnson, *Approximation algorithms for bin packing - an updated survey*, Algorithms design for computer system design (G. Ausiello, M. Lucertini, and P. Serafini, eds.), Spring-Verlag, New York, 1984, pp. 49–106.

[38] E.G. Coffman, G. Galambos, S. Martello, and D. Vigo, *Bin packing approximation algorithms: Combinatorial analysis*, Handbook of Combinatorial Optimization (D.-Z. Du and P.M. Pardalos, eds.), Kluwer Academic Publishers, 1999, pp. 151–208.

[39] G. B. Dantzig and P. Wolfe, *Decomposition principle for linear programs*, Operations Research **8** (1960), 101–111.

[40] S. Dash and O. Günlük, *Valid inequalities based on simple mixed-integer sets*, Mathematical Programming **105** (2006), 29–53.

[41] J. Erschler, P. Lopez, and C. Thuriot, *Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement*, Revue d'Intelligence Artificielle **5** (1991), no. 3, 7–32.

[42] S. Fekete and J. Schepers, *New classes of fast lower bounds for bin packing problems*, Mathematical Programming **91** (2001), 11–31.

[43] ———, *A combinatorial characterization of higher-dimensional orthogonal packing*, Mathematics of Operations Research **29** (2004), 353–368.

[44] ———, *A general framework for bounds for higher-dimensional orthogonal packing problems*, Mathematical Methods of Operations Research **60** (2004), 311–329.

[45] S. Fekete, J. Schepers, and J. Van Der Ween, *An exact algorithm for higher-dimensional orthogonal packing.*, Operations Research **55** (2007), no. 3, 569–587.

[46] Albert E. Fernandes Muritiba, Manuel Iori, Enrico Malaguti, and Paolo Toth, *Algorithms for the Bin Packing Problem with Conflicts*, INFORMS Journal on Computing (2009), ijoc.1090.0355.

[47] K. Fleszar and K.S. Hindi, *New heuristics for one-dimensional bin-packing*, Computers and Operations Research **29** (2002), 821–839.

[48] M. R. Garey and D. S. Johnson, *Computers and intractability, a guide to the theory of NP-completeness*, Freeman, New York, 1979.

[49] M. Gendreau, G. Laporte, and F. Semet, *Heuristics and lower bounds for the bin packing problem with conflicts*, Computers and Operations Research **31** (2004), 347–358.

[50] P. Gilmore and R. Gomory, *A linear programming approach to the cutting stock problem*, Operations Research **9** (1961), 849–859.

[51] ———, *A linear programming approach to the cutting stock problem - part II*, Operations Research **11** (1963), 863–888.

[52] F. Glover and M. Laguna, *Tabu search*, Kluwer Academic Publishers, 1998.

[53] R. Gomory, *Outline of an algorithm for integer solutions to linear programs*, Bulletin of the American Mathematical Society **64** (1958), 275–278.

[54] B. O'Sullivan H. Simonis, *Search strategies for rectangle packing*, Principles and Practice of Constraint Programming (CP 2008), 2008, pp. 52–66.

[55] E. Hadjiconstantinou and N. Christofides, *An exact algorithm for general, orthogonal, two-dimensional knapsack problem*, European Journal of Operational Research **83** (1995), 39–56.

[56] S. Hanafi and A. Fréville, *An efficient tabu search approach for the 0–1 multidimensional knapsack problem*, European Journal of Operational Research **106** (1998), 659–675.

[57] P. Hansen, N. Mladenović, and J.A. Moreno Pérez, *Variable neighbourhood search: methods and applications*, Annals of Operations Research **175** (2010), 367–407.

[58] M. Haouari and A. Gharbi, *Fast lifting procedures for the bin packing problem*, Discrete Optimization **2** (2005), no. 3, 201–218.

[59] M. Hifi, *Exact algorithms for the guillotine strip cutting/packing problem*, Computers and Operations Research **25** (1998), no. 11, 925–940.

[60] ———, *Reductions strategies and exact algorithms for the disjunctively constrained knapsack problem*, Computers and Operations Research **34** (2005), 2657–2673.

[61] P. Jégou, S. N. Ndiaye, and C. Terrioux, *Computing and exploiting tree-decompositions for solving constraint networks*, Principles and Practice of Constraint Programming (CP-2005), 2005, pp. 777–781.

[62] C. Joncour and A. Pêcher, *Consecutive ones matrices for multi-dimensional orthogonal packing problems*, Electronic Notes in Discrete Mathematics International Symposium on Combinatorial Optimization (Hammamet Tunisie), 2010, p. to appear (Anglais).

[63] C. Joncour, A. Pêcher, and P. Valicov, *MPQ-trees for orthogonal packing problem*, Tech. report, LABBRI, 2010.

[64] A. Khanafer, F. Clautiaux, S. Hanafi, and E.-G. Talbi, *A multi-objective bin-packing problem with conflicts*, To appear in Computers and Operations Research (2012).

[65] A. Khanafer, F. Clautiaux, and E.-G. Talbi, *Tree-decomposition based heuristics for the two-dimensional bin packing problem with conflicts*, To appear in Computers and Operations Research (2010).

[66] A. Khanafer, F. Clautiaux, and E.G. Talbi, *New lower bounds for bin packing problems with conflicts*, European Journal of Operational Research **2** (2010), no. 206, 281–288.

[67] R. E. Korf, M. D. Moffitt, and M. E. Pollack, *Optimal rectangle packing*, To appear in Annals of Operations Research (2010).

[68] A. Koster, H. Bodlaender, and S. van Hoesel, *Treewidth: Computational experiments*, Fundamenta Informaticae **49** (2001), 301–312.

[69] M. Labbé, G. Laporte, and H. Mercure, *Capacitated vehicle routing on trees*, Operations Research **39** (1991), no. 6, 16–22.

[70] A. N. Letchford and A. Lodi, *Strengthening chváal-gomory cuts and gomory fractional cuts*, Operations Research Letters **30** (2002), 74–82.

[71] A. Lodi, S. Martello, and M. Monaci, *Two-dimensional packing problems: a survey*, European Journal of Operational Research **141** (2002), 241–252.

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

[73] ———, *Recent advances on two-dimensional bin packing problems*, Discrete Applied Mathematics **123** (2002), 379–396.

[74] P. Lopez, J. Erschler, and P. Esquirol, *Ordonnancement de tâches sous contraintes : une approche énergétique*, RAIRO Automatique, Productique, Informatique Industrielle **26** (1992), no. 6, 453–481.

[75] G. S. Lueker, *Bin packing with items uniformly distributed over intervals [a,b]*, Proc. of the 24th Annual Symposium on Foundations of Computer Science (FOCS 83), IEEE Computer Society, 1983, pp. 289–297.

[76] R. Macedo, C. Alves, and J.M. Valério de Carvalho, *Arc-flow model for the two-dimensional guillotine cutting stock problem*, Computers and Operations Research **37** (2010), no. 6, 991–1001.

[77] S. Martello, M. Monaci, and D. Vigo, *An exact approach to the strip-packing problem*, INFORMS Journal on Computing **15** (2003), 310–319.

[78] S. Martello and P. Toth, *Knapsack problems - algorithms and computer implementation*, Wiley, Chichester, 1990.

[79] S. Martello and D. Vigo, *Exact solution of the two-dimensional finite bin packing problem*, Management Science **44** (1998), 388–399.

[80] G. L. Nemhauser and L.A. Wolsey, *Integer and combinatorial optimization*, Wiley, New York, 1998.

[81] C. Nitsche, G. Scheithauer, and J. Terno, *New cases of the cutting stock problem having mirup*, Mathematical Methods of Operations Research **48** (1998), no. 1, 1432–2994.

[82] D. Pisinger and M. Sigurd, *Using decomposition techniques and constraint programming for solving the two-dimensional bin packing problem*, INFORMS Journal on Computing **19** (2007), no. 1, 36–51.

[83] N. Robertson and P. Seymour, *Graph minors. II algorithmic aspects of tree-width*, Journal of Algorithms **7** (1986), 309–322.

[84] A. Scholl, R. Klein, and C. Jurgens, *BISON: a fast hybrid procedure for exactly solving the one-dimensional bin-packing problem*, Computers and Operations Research **24** (1997), no. 7, 627–645.

[85] R. E. Tarjan and M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM Journal on Computing, vol. 13, 1984, pp. 566–579.

[86] J.M. Valério de Carvalho, *Using extra dual cuts to accelerate column generation*, INFORMS Journal on Computing **17** (2005), no. 2, 175–182.

[87] F. Vanderbeck, *Exact algorithm for minimizing the number of setups in the one-dimensional cutting stock problem*, Operations Research **46** (2000), no. 6, 915–926.

[88] K.V. Viswanathan and A. Bagchi, *Best-first search methods for constrained two-dimensional cutting stock problem*, Operations Research **41** (1993), 768–776.

[89] P. Y. Wang, *Two algorithms for constrained two-dimensional cutting stock problems*, Operations Research **31** (1983), 573–586.

[90] G. Wäscher, H. Haussner, and H. Schumann, *An improved typology of cutting and packing problems*, European Journal of Operational Research **183** (2007), 1109–1130.

Index of problems

Title New collaborative approaches for bin-packing problems

Abstract This document describes new models and methodologies that we apply to three families of packing problems. We first study decomposition methods and meta-heuristics based on so-called strategic oscillation. We apply these techniques to packing problems with different kinds of conflicts. We also deal with the concept of *dual-feasible functions*, which are used to derive polynomial-time lower bounds for several bin-packing problems. Finally, we propose new models for two different rectangle placement problems. We used these models into a constraint programming framework.

Keywords operations research, cutting and packing, mathematical programming, heuristics,meta-heuristics, constraint-programming, decomposition methods, dual-feasible functions

Titre Nouvelles approches collaboratives pour des problèmes de conditionnement

Résumé Ce document décrit de nouvelles modélisations et approches de résolution que nous appliquons à des problèmes de découpe et de conditionnement. Nous étudions dans un premier temps plusieurs techniques de décomposition alliées à différentes méta-heuristiques basées sur des stratégies d'oscillation. Nous étudions ensuite le concept de fonctions dual-réalisables qui permettent d'obtenir des évaluations par défaut polynomiales pour des problèmes de conditionnement. Finalement, nous proposons des modèles originaux pour des problèmes de placement de rectangles. Nous utilisons ces modèles dans des méthodes de programmation par contraintes.

Mots-clés recherche opérationnelle, découpe et conditionnement, programmation mathématique, heuristiques, méta-heuristiques, programmation par contraintes, méthodes de décomposition, fonctions dual-réalisables