High-Multiplicity Scheduling and Packing Problems

Michaël Gabay

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques et Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Michaël Gabay

Thèse dirigée par **Nadia Brauner**

préparée au sein **Laboratoire G-SCOP**
et de **L'école doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

High-Multiplicity Scheduling and Packing Problems

Theory and Applications

Thèse soutenue publiquement le **20 octobre 2014**,
devant le jury composé de :

M. Yves Crama
Professeur, Université de Liège, Belgique, Président
M. Alessandro Agnetis
Professeur, Université de Sienne, Italie, Rapporteur
M. François Clautiaux
Professeur, Université de Bordeaux, France, Rapporteur
M. Pierre Lemaire
Maître de conférences, Grenoble-INP Génie Industriel, France, Examinateur
M. Francis Sourd
Chercheur HDR, Manager Recherche et Développement, Sun'R Smart Energy, France, Examinateur
M^{me} Nadia Brauner
Professeur, Université Joseph Fourier, France, Directrice de thèse

Remerciements

Je tiens tout d'abord à exprimer mes remerciement à Nadia Brauner qui a dirigé mes travaux de thèse pendant ces trois années. Merci pour les très riches et nombreuses discussions et réflexions que nous avons eu ensemble et un grand merci de m'avoir fait confiance et de ne jamais m'avoir freiné dans mon enthousiasme permanent à explorer des nouvelles questions et des nouveaux domaines. Au-delà même de cela, merci de m'y avoir encouragé et d'avoir fait en sorte de réunir les conditions humaines et matérielles à la réussite de ces travaux.

Pour continuer, je souhaite remercier tous les membres de mon jury: Alessandro Agnetis et François Clautiaux, en la qualité de rapporteurs ; Yves Crama, en tant que Président du Jury ; Pierre Lemaire et Francis Sourd, en la qualité d'examinateurs ; d'avoir accepté avec enthousiasme d'évaluer ces travaux et de m'avoir fait l'honneur de leur présence dans ce jury.

Je voudrai également exprimer ma gratitude au Professeur Gerd Finke pour tout le bonheur que j'ai eu à travailler avec lui sur le problème le plus frustrant que j'ai étudié jusqu'à présent.

Je tiens également à remercier tous les collègues et collaborateurs français ou d'ailleurs avec qui j'ai pu interagir durant cette thèse, sur le plan de la recherche, de l'enseignement ou de simples discussions occasionnelles. En particulier, je souhaite remercier Alexander Grigoriev, Christophe Rapine, Hadrien Cambazard et Sofia Zaourar avec qui j'ai eu grand plaisir à interagir et collaborer.

Je souhaite également remercier les membres du Laboratoire G-SCOP dans leur intégralité pour le formidable environnement de travail qu'ils s'impliquent tous à développer. Merci à l'A-DOC toutes générations confondues et en particulier à mes deux comparses de l'A-DOC, Maud et Anne-Laure. Un grand clin œil à tous les (ex-)doctorants et non permanents du laboratoire : mes ex co-bureaux Julien, Yohann, Diana, Ariel et Florence ; ceux qui, malgré le saignement discontinu de leurs oreilles en ma présence, ont continué à jouer de la guitare avec moi, Clément et Marine ; ceux qui remplissent le frigo, qui Bang, qui coinchent, qui traînent à la cafette ou ailleurs, je ne me risquerai pas à les énumérer ayant bien trop peur d'en oublier un(e) mais je les remercie sincèrement pour tous les formidables moments passés ensembles !

Merci à Zi Hui qui m'a soutenu ardemment ; qui a, par son amour, égayé mon quotidien, par ses efforts, allégé mes tâches et, par ses questions judicieuses, m'a permis de trouver les réponses à nombre des questions que je me suis posé durant cette thèse.

Merci enfin à ma famille, en particulier à mon grand-père qui m'a inspiré depuis mon enfance et qui continuera encore à m'inspirer pendant de très nombreuses années et à ma mère qui m'a toujours soutenu, n'a jamais questionné mes choix et à tout fait pour que je puisse mener mes études dans les meilleures conditions possibles.

Contents

Introduction (français)

Nous sommes ka-tet,
Nous sommes un en plusieurs,
La multiplicité faite unité.

Roland Deschain, La Tour Sombre, Stephen King

Résumé : Nous nous intéressons dans cette thèse aux problèmes d'ordonnancement et de conditionnement en high-multiplicity. Nous présentons dans ce chapitre les concepts de l'ordonnancement high-multiplicity et de cet encodage. Nous exposons quelques unes des difficultés relatives à l'utilisation d'un tel encodage et nous détaillons le contenu de cette thèse.

Ordonnancement et encodage High-Multiplicity

Dans l'étude dans problèmes d'ordonnancement, on considère généralement que les paramètres des différentes tâches sont spécifiés séparément pour chaque tâche. Néanmoins, lorsque le problème comprend des tâches identiques, il est naturel de les exprimer de manière agrégée, en exprimant directement les caractéristiques d'un groupe de tâches identiques et le nombre de tâches (la *multiplicité*) de ce groupe. Cet encodage compact d'une instance d'un problème est appelé encodage high-multiplicity et permet de réduire significativement la taille de la description d'une instance pour les problèmes comportant des tâches identiques. À l'extrême, pour un problème comportant n tâches identiques, la taille de l'encodage passe de $\mathcal{O}(n)$ à $\mathcal{O}(\log n)$ lorsqu'on utilise l'encodage high-multiplicity.

Utiliser un tel encodage permet de réduire la taille du problème mais ne le rend pas plus facile pour autant, bien au contraire. Si on s'intéresse au point de vue de l'analyse de complexité pour ces problèmes, la difficulté est caractérisée en rapport avec la taille de l'encodage d'une instance. En conséquence, un même algorithme utilisé dans les deux cas (encodage high-multiplicity et encodage non high-multiplicity) sera plus onéreux lorsqu'un encodage high-multiplicity est considéré.

Une seconde conséquence intéressante est que spécifier les dates de début de toutes les tâches et leurs machines ne constitue généralement pas un certificat polynomial pour les problèmes d'ordonnancement high-multiplicity puisque le nombre de tâches peut être exponentiel (plus précisément, pseudo-polynomial) en la taille de l'instance high-multiplicity. Alors même que

ceci permet souvent de prouver immédiatement qu'un problème d'ordonnancement non high-multiplicity est dans $\mathcal{NP}$, démontrer l'appartenance à $\mathcal{NP}$ d'un problème high-multiplicity est généralement beaucoup plus complexe et nécessite une analyse fine et structurelle des propriétés des solutions optimales. Les certificats polynomiaux pour les problème d'ordonnancement high-multiplicity font généralement usage de propriétés de dominances permettant de regrouper des tâches par groupes afin de donner une description compacte d'un ordonnancement.

Les problèmes d'ordonnancement high-multiplicity se présentent dans de nombreux contextes industriels comportant des opérations manufacturières répétitives, des maintenances ou des familles (lots) de tâches identiques.

L'encodage high-multiplicity est également utile dans d'autres domaines que l'ordonnancement. Il intervient dans les problèmes de conditionnement mais également dans d'autres problèmes plus éloignés de l'ordonnancement. En particulier, on peut citer les problèmes de graphes, par exemple pour la conception de circuits électroniques (VLSI layouts design), de tournées de véhicules ou de voyageurs de commerces avec visites multiples. Un état de l'art portant sur les problèmes utilisant un encodage high-multiplicity est présenté au chapitre 1.3.

Il existe de nombreuses approches permettant de traiter des problèmes high-multiplicity. Par exemple, pour commencer, on peut essayer de trouver un algorithme polynomial pour le problème concerné. Néanmoins, la tâche est généralement très ardue et l'existence d'un algorithme polynomial pour le problème avec un encodage non high-multiplicity ne permet pas d'obtenir directement un algorithme polynomial pour la version high-multiplicity et ne garantie pas non plus l'existence d'un tel algorithme.

Le premier objectif lors de l'étude d'un problème high-multiplicity est généralement de démontrer son appartenance à $\mathcal{NP}$ et donc de trouver un certificat polynomial. Pour ce faire, on analyse généralement les structures des solutions optimales et on recherche des dominances. Les certificats obtenus utilisent généralement des structures de groupes. Par exemple, le problème d'ordonnancement $P|pmtn|C_{max}$ est un problème d'ordonnancement très simple qu'on résout dans le cas non-high-multiplicity avec la règle de McNaughton (McNaughton 1959). Remarquons qu'on peut trouver une solution optimale pour ce problème dans laquelle toutes les tâches d'une même famille sont affectées consécutivement sur une même machine et, si nécessaire, sur des machines consécutives. On peut alors décrire cette solution en spécifiant la date de début de chaque *famille* de tâches et l'indice de la première machine utilisée. On obtient ainsi un certificat polynomial pour la variante high-multiplicity du problème.

Objectifs et organisation de la thèse

L'objectif de cette thèse est d'exposer des techniques permettant d'étudier et résoudre des problèmes high-multiplicity et également de contribuer plus généralement à l'étude des problèmes d'ordonnancement et de conditionnement. Nous présentons différentes approches permettant d'étudier des problèmes en high-multiplicity et des contributions analytiques, algorithmiques et numériques sur des problèmes de conditionnement.

Dans le premier chapitre, nous présentons les enjeux de l'étude des problèmes utilisant un encodage high-multiplicity, un exemple introductif détaillé et le plan détaillé du mémoire.

La suite de la thèse est divisée en deux partie. La première traite des problèmes d'ordonnancement et la seconde des problèmes de conditionnement (dont certains sont également des prob-

lèmes d'ordonnancement mais les outils utilisés dans cette partie correspondent généralement à ceux de la littérature de packing).

Dans le chapitre 2, nous étudions un problème d'ordonnancement high-multiplicity à une machine, avec des instants de début et fin de tâches interdits. Nous démontrons que ce problème est dans $\mathcal{NP}$, même avec un encodage high-multiplicity et proposons un algorithme polynomial pour le cas où le nombre de types de tâches est plus grand que le nombre d'instants d'indisponibilité et un algorithme FPT, qui est polynomial lorsque le nombre d'instants d'indisponibilités est fixé.

Dans le chapitre 3, nous étudions le problème d'ordonnancement de tâches couplées identiques. Ce problème est un cas extrême de problème d'ordonnancement high-multiplicity puisqu' il ne comporte qu'un seul type de tâche. Toutes les tâches sont donc identiques et il suffit alors de spécifier 4 entiers pour spécifier une instance complète. Ce problème est néanmoins très difficile et demeure ouvert depuis plus de 15 ans. Nous présentons des bornes inférieures et supérieures pour ce problème et étudions finement différentes structures de solutions optimales. Nous utilisons ces structures pour créer des algorithmes polynomiaux ou de faible complexité permettant d'obtenir d'excellentes solutions. Nous décomposons également ces structures et expliquons pourquoi l'existence de celles-ci laisse très peu d'espoir quant à l'existence même d'un certificat polynomial pour ce problème.

Dans les chapitres 4 et 5, nous étudions le problème de bin stretching qui est une version semi-en-ligne du problème d'ordonnancement sur machine parallèle identiques $Pm||C_{max}$. Nous proposons dans le chapitre 5, un algorithme d'approximation à performance garantie. L'algorithme proposé améliore les meilleurs algorithmes connus pour ce problème.
Dans le chapitre 5, nous proposons un algorithme permettant de calculer des bornes inférieures pour ce problème et nous utilisons celui-ci pour calculer une borne inférieure améliorée. C'est la première fois que les bornes inférieures sont améliorées sur ce problème, en particulier du fait de la quasi-impossibilité d'utiliser les techniques de bornes classiques à cause de la connaissance initiale. L'approche proposée dans ce chapitre est générale et transposable à d'autres problèmes d'ordonnancement ou de conditionnement en-ligne ou semi-en-ligne.

Dans le chapitre 6, nous introduisons et étudions une variante du problème de vector bin packing. Nous proposons et expérimentons de nombreuses heuristiques sur celui-ci permettant de construire des solutions réalisables. Ce problème est inspiré du challenge EURO/ROADEF 2012 auquel j'ai participé en collaboration avec Sofia Zaourar.

Nous présentons dans le chapitre 7, une réduction générale pour les problèmes de conditionnement. Nous étudions les structures sous-jacentes communes à ces problèmes et présentons un algorithme de réduction polynomial. L'algorithme est très général et applicable à de nombreux problème de conditionnement comme le bin packing, vector bin packing, bin packing multidimensionnel, etc. Par ailleurs, l'algorithme est polynomial en la taille de l'encodage de l'instance (high-multiplicity ou non) et peut s'intégrer naturellement et à des algorithmes de résolution exacte pour les problèmes de conditionnement.

Nous terminons cette thèse par un résumé et une mise en perspective des travaux réalisés ainsi que des propositions de pistes de recherches à privilégier pour les futurs travaux portant sur l'étude des problèmes avec un encodage high-multiplicity.

Les différents chapitres de cette thèse sont indépendants. Les chapitres sont rédigés en anglais à l'exception de cette introduction et de la conclusion, présentée dans les deux langues.

Chapter 1

High-Multiplicity Scheduling

We are ka-tet. We are one from many.

Roland Deschain,
The Dark Tower,
Stephen King

Résumé : Nous présentons dans ce chapitre les concepts de l'ordonnancement high-multiplicity et de cet encodage. Nous expliquons quelques unes des difficultés relatives à l'utilisation d'un tel encodage et nous les exposons sur un exemple. Nous présentons un état de l'art ainsi que la plan de cette thèse.

Abstract: In this thesis, we are interested in solving scheduling and packing problems, focusing on problems in which the input is encoded using multiplicities. In this chapter, we introduce the concept of high-multiplicity scheduling. We present and emphasize the extent of high-multiplicity encoding on a small example problem which we call *The 7 Wonders Problem*. Then, we present a literature review on high-multiplicity scheduling and the outline of the thesis[1].

1.1 High-Multiplicity: *Problems and challenges*

In scheduling problems, it is often assumed that the parameters of the jobs are specified separately for each job. However, when there are identical jobs it is natural to group the jobs and specify them all at once by describing the features of a representative job and the number of such jobs. For instance, if one has to produce 50 000 items with features (f_1, f_2, f_3), it is natural

[1] We assume that the reader is familiar with classical concepts of complexity analysis. If not, the reader can refer to the very well written book from Garey and Johnson (1979) for a first approach on complexity or to Papadimitriou (2003) for more details on specific concepts, especially on less usual complexity classes such as EXP. We also assume that the reader is familiar with scheduling and, to a lesser extent, with packing problems. The reader can refer to the book from Pinedo (2012) for an introduction to scheduling and a detailed study of common problems. Graham's 3-fields notations as well as other notations used in this thesis are presented in Appendix A.

to describe this as "Produce 50 000 items with features (f_1, f_2, f_3)" while it is not natural to ask to "Produce item (f_1, f_2, f_3) and item (f_1, f_2, f_3), etc." with (f_1, f_2, f_3) repeated 50 000 times. The compact way to state a problem is called *high-multiplicity encoding* because each kind of job is described only once by specifying the features of a single representative job and a *multiplicity* which is the number of occurrences of the job. This representation allows to provide compact problem statements and yields drastic decrease in the size of the encoding of a problem instance.

We denote by n the total number of jobs in the problem and by s the number of different types of jobs. While the input size of a problem is roughly $\mathcal{O}(n)$ when all jobs are specified separately, it can get as low as $\mathcal{O}(\log n)$ when multiplicities are used.

The reduction of the input size induces several changes in the complexity study of problems with compact encoding. For instance, an algorithm which is polynomial in the number of jobs n is then exponential (more precisely, pseudo-polynomial) on an input with multiplicities. Another consequence is that specifying a schedule by giving all jobs starting times and the allocation of jobs to the machines is exponential in the input size.

While most scheduling problems specified using a job-by-job encoding can be proven easily to belong to $\mathcal{NP}$ by providing a schedule as a certificate, proving that a high-multiplicity scheduling problem belongs to $\mathcal{NP}$ may be a hard matter. Let us recall a definition of $\mathcal{NP}$:

Definition 1.1 (Verifier based definition of $\mathcal{NP}$)**.** A decision problem P is in $\mathcal{NP}$ if it has a polynomial-size certificate which can be verified in polynomial time.

This means that a decision problem is in $\mathcal{NP}$ if, when the answer to this problem is Yes, we can provide a proof whose length is reasonable and which can be verified in reasonable time compared to the input size. Such a proof is called a polynomial certificate. In scheduling problems, a certificate is usually a schedule specifying the assignment of all jobs to the machines and their starting times. However, in high-multiplicity scheduling problems, this certificate is not polynomial in the input size. Therefore, in order to prove that a high-multiplicity scheduling problem belongs to $\mathcal{NP}$ we have to find more compact certificates. This is usually achieved by using dominance properties on the considered problems such as "all jobs of a given group can be scheduled consecutively" or "the order of the jobs does not matter".

High-multiplicity scheduling problems occur in several industrial domains involving repetitive manufacturing operations, periodic maintenance, or families of identical jobs. Moreover, when designing approximation algorithms to solve problems, it is often interesting to aggregate data and group similar (but not necessarily identical) jobs, *i.e.* to transform a problem into a similar problem with multiplicities.

High-multiplicity does not only occur in scheduling. For instance, it can be encountered in graph theory with problems such as the traveling salesman problem with multiple visits, vehicle routing problems, VLSI layouts design, etc. It is also encountered in big data where the amount of data is so massive that even processing it in linear time cannot be considered. In this case, data is first aggregated and then processed, in the same way as it is done in approximation algorithms which aggregate jobs. In Section 1.3, we provide references on these types of problems.

There are several ways to cope with high-multiplicity scheduling problems. One of them is to find a polynomial algorithm to solve the problem. Thus, we do not need to find a certificate since the input can be provided as such. However, this may not be satisfactory since knowing the optimal value of a problem is often worthless without information on the optimal solution. Hence the aim is rather to find both a polynomial algorithm and an efficient way to describe an optimal solution.

In general, the first matter is to find a polynomial certificate for the problem. Finding a certificate can often be achieved by grouping jobs and machines. For instance, $P|pmtn, HM|C_{max}$ (where HM means that high-multiplicity encoding is used, see Appendix A) is a very simple scheduling problems which can be solved using McNaughton's wrap-around rule (McNaughton 1959) and there exists an optimal schedule which can be described in polynomial size: observe that there is an optimal solution in which all jobs from each family are scheduled consecutively and on consecutive machines. We can describe this solution by giving the starting time and the machine of the first job from each family.

In the following of this chapter, we present: an introductory example to high-multiplicity in Section 1.2, a literature review in Section 1.3 and the outline of the thesis in Section 1.4.

1.2 An introductory example: *The 7 Wonders Problem*

In this section, we present and analyze a simple optimization problem occurring in a famous board game. Throughout this analysis, we introduce several complexity concepts, including the encoding of the input. The aim of this example is to introduce a few basic and advanced features of complexity analysis with multiplicities, the challenges, and to understand the importance of high-multiplicity encoding and why it should be considered as the standard when this encoding makes sense.

7 Wonders is a board game created by Antoine Bauza and edited by Repos Production. It is based on the draft principle, well known from Magic[2] players. In 7 Wonders, in the end of a game, each player calculates his victory points. The player who has the highest score (total of victory points) wins the game. The score of a player is obtained by summing up his victory points obtained by building monuments and wonders, making war and developing sciences.

Source: http://boardgamegeek.com/image/842338/7-wonders
Author: garyjames
License: Creative Commons Attribution-Share Alike 3.0 Unported

Figure 1.1: A game of 7 Wonders

All scores are trivial to compute, except sometimes science. Sciences are partitioned into 3 kinds, each one corresponding to a symbol: geometry, writing and engineering. The science score is equal to the sum of the squares of the number of cards possessed for each science plus 7 times the number of groups of 3 different symbols. For instance, if a player has 4 geometry cards, 2 writing cards and 3 engineering cards, his science score is $4^2 + 2^2 + 3^2 + 7 \times 2 = 43$, with one more engineering symbol his score would be $4^2 + 2^2 + 4^2 + 7 \times 2 = 50$ and with an additional writing symbol, the score would be $4^2 + 3^2 + 3^2 + 7 \times 3 = 55$. However, there is a subtlety: there are special cards (guilds) providing science *jokers*. Each of these cards counts as an additional science card in the field chosen by the player. For instance, with the previous setup $(4, 2, 3)$ and one joker, the player can assign his joker to yield any of the three following configuration: $(5, 2, 3)$ (score 52), $(4, 3, 3)$ (score 55) or $(4, 2, 4)$ (score 50). Hence the best choice is to assign this card to the second kind, writing. With one joker, the problem is easy but there are several such jokers and with n jokers, there are $(n+1)(1+\frac{n}{2})$ different solutions (with all jokers assigned). One can test all these combinations in quadratic time in n which is easy to do with a computer but harder for a player, even with few jokers.

[2]Magic : The Gathering is a collectible card game created by Richard Garfield and edited by Wizards of the Coast.

In the game, the number of jokers is limited and they are not exactly all the same but we will suppose that they are all identical and that the number of jokers can be large. In the following, we mathematically define the problem and analyze algorithms to solve it.

1.2.1 Problem definition

We denote by v_1, v_2 et v_3 the number of cards that the player has for each science and by n his number of joker cards. The aim of the player is to assign his jokers in order to maximize his score. The first step in studying an optimization problem is to define the corresponding decision problem:

7 WONDERS

Input : v_1, v_2 and v_3 the number of cards possessed for each science, n the number of jokers possessed, K a threshold.

Output : YES if and only if there are positive integers x_1, x_2, x_3 such that $x_1 + x_2 + x_3 \leq n$ and $f(x_1, x_2, x_3) \geq K$.

With $f(x_1, x_2, x_3) = (v_1 + x_1)^2 + (v_2 + x_2)^2 + (v_3 + x_3)^2 + 7 \times \min(v_1 + x_1, v_2 + x_2, v_3 + x_3)$. The values of x_1, x_2 and x_3 are the number of jokers respectively assigned to type 1, 2 and 3 and $f(x_1, x_2, x_3)$ is the corresponding score. Remark that f is strictly increasing in all of its variables, hence, for any optimal solution, $x_1^* + x_2^* + x_3^* = n$.

We denote by $v_{max} = \max(v_1, v_2, v_3)$. The problem input is given by 4 integers: v_1, v_2, v_3 and n. The size of this input is $\mathcal{O}(\log(v_{max}) + \log(n))$. Informally, a polynomial algorithm is an algorithm whose running time is bounded by a polynomial of the input size. An algorithm is strongly polynomial if the number of operations performed by this algorithm does not depend on the magnitude of the numbers in the input. For the 7 Wonders problem, since the number of values in the input is constant, an algorithm is strongly polynomial if its number of operations can be bounded by a constant. Our aim is to find *efficient* algorithms to solve this problem.

1.2.2 First approach: compute all values

This problem can be solved by computing all feasible values of f. Since $x_1^* + x_2^* + x_3^* = n$ in an optimal solution, we restrict the computations to all positive integers x_1, x_2, x_3 such that $x_1 + x_2 + x_3 = n$. There are $(n+1)(1 + \frac{n}{2})$ such solutions.

The optimal solution can be computed by evaluating $(n+1)(1 + \frac{n}{2})$ times f with the different values of x_1, x_2, x_3. Since f can be evaluated in $\mathcal{O}(\log(v_{max}) + \log(n))$ time, the overall complexity of this algorithm is $\mathcal{O}(n^2(\log(v_{max}) + \log(n)))$ (or $\mathcal{O}(n^2)$ if we consider that arithmetical operations can be computed in constant time). The first obvious remark is that this algorithm is not strongly polynomial since the number of computations depends on the magnitude of n. But is it polynomial?

1.2.2.1 Player's point of view

With our definition of the 7 Wonders problem, it is obvious that this algorithm is not polynomial in the input size since n is exponential in $\log n$. However, one can argue that the player needs to *manipulate* n cards, hence *"in the real world"*, the input size is $\mathcal{O}(n + v_{max})$ and the algorithm is

polynomial in the *"real world"* input size. We remark that in this case, polynomial is not efficient: there are indeed n different items but asking the player to do n^2 operations on these items is not reasonable. Even though the items exist and the player manipulates them, it is not reasonable to assume that they can easily be manipulated individually. For instance, once a player has decided the values of x_1, x_2 and x_3, he will most likely place his x_1 first jokers on the first stack, his x_2 second jokers on the second stack and the rest on the third stack. The player assigns several cards at once, he manipulates *batches* of cards.

Since all jokers have the same properties, the only thing that matters is the number of jokers. Having jokers j_7, j_{14} and j_{42} does not matter; what matters is only their total number, the *multiplicity* of the "joker" item. This yields the 4 integer input which we defined for this problem. This is what we call the *high-multiplicity* encoding: for each class of items, specify the properties of the items in the class and the multiplicities of such items. The input obtained is more natural and also more compact. In the 7 Wonders problem, this input size is $\mathcal{O}(\log(v_{max}) + \log(n))$. The algorithm testing all combinations is exponential in the input size. More precisely, it is *pseudo-polynomial*, meaning that if the input was coded in unary, the algorithm would be polynomial in the input size. For instance, the input $v_1 = 2, v_2 = 1, v_3 = 3, n = 4$ coded in unary could be represented as: 11 2 333 $jjjj$.

We have seen that this algorithm is not polynomial in the input size but what can we do then?

1.2.2.2 Refine analysis

A smart player will remark that there is a limited number of cards in the game. Let K be the total number of cards. Science and jokers are cards, hence there is obviously less than K cards for each science and K jokers in the game. The input size of the decision problem is then upper bounded by $\mathcal{O}(K)$. Under the reasonable assumption that the total number of cards K is either a constant or is bounded by a constant, the input size of the problem is bounded by $\mathcal{O}(1)$ and the complexity of the previous algorithm is now $\mathcal{O}(1)$. Does it mean that we were mistaken in the previous complexity analysis ?

No, it does not: what we have shown here is that the problem is *fixed-parameter tractable* which means that once we have fixed a parameter (in our case the total number of cards in the game), the problem can be solved in polynomial time. In the previous case, K was not fixed, and the input n could be as large as one wants it to be. Still, does it mean that it is now easy to manipulate the cards ?

Once again, this is not the case. It means that we can guarantee that the player will not spend more than a constant (possibly large) amount of time to try all combinations. One can think that such a fact does not really help but there is another way to interpret this fact: it means that we can compute once and for all a table containing the optimal assignment and score for all feasible values of v_1, v_2, v_3 and n. Then in the end of a game, the player only has to refer to this table to know his score, requiring no additional effort. The computational time spent to compute this table depends on the magnitude of K.

We have shown that this problem is fixed-parameter tractable which is a first improvement over the first results. The next problem is to determine whether this problem can be solved in polynomial time in the input size when K is not fixed.

1.2.3 A strongly polynomial algorithm

In this section, we obtain a strongly polynomial algorithm for the 7 Wonders Problem by refining the analysis. We present the whole process to obtain such an algorithm but the main concern here is to show how we can further analyze a problem to obtain such an algorithm and not the computations.

We model the 7 Wonders optimization problem as follows:

$$\begin{aligned}
\max \quad & (v_1+x_1)^2+(v_2+x_2)^2+(v_3+x_3)^2+7\times\min(v_1+x_1,v_2+x_2,v_3+x_3) && (1.1)\\
\text{s.t.} \quad & x_1+x_2+x_3\leq n && (1.2)\\
& x_1,x_2,x_3\geq 0 && (1.3)\\
& x_1,x_2,x_3\in\mathbf{Z} && (1.4)
\end{aligned}$$

We reformulate the objective function:

$$\max \quad x_1^2+x_2^2+x_3^2+2v_1x_1+2v_2x_2+2v_3x_3+7\times\min(v_1+x_1,v_2+x_2,v_3+x_3) \qquad (1.5)$$

Without loss of generality, we assume that $v_1\geq v_2\geq v_3$. Notice that there is an optimal solution verifying $v_2+x_2^*\geq v_3+x_3^*$. Indeed, if $v_2+x_2^*<v_3+x_3^*$, then the solution $x_1=x_1^*$, $x_2=x_2^*+(v_3+x_3^*-v_2-x_2^*)$, $x_3=x_2^*+x_3^*-x_2$ is feasible and its cost is not smaller. Same goes for x_1 and x_2: $v_1+x_1^*\geq v_2+x_2^*$. We strengthen the problem formulation using this dominance. Hence, $v_1+x_1\geq v_2+x_2\geq v_3+x_3$ and $\min(v_1+x_1,v_2+x_2,v_3+x_3)=v_3+x_3$. Now, we have the following formulation:

$$\begin{aligned}
\max \quad & x_1^2+x_2^2+x_3^2+2v_1x_1+2v_2x_2+2v_3x_3+7x_3 && (1.6)\\
\text{s.t.} \quad & v_1+x_1\geq v_2+x_2 && (1.7)\\
& v_2+x_2\geq v_3+x_3 && (1.8)\\
& x_1+x_2+x_3=n && (1.9)\\
& x_1,x_2,x_3\geq 0 && (1.10)\\
& x_1,x_2,x_3\in\mathbf{Z} && (1.11)
\end{aligned}$$

This is a quadratic function to optimize over a convex domain. In the following, we solve this problem by using a relaxation in which the integrality constraint (1.11) is removed.

Earlier on, we noticed that the solutions verifying $v_1+x_1\geq v_2+x_2\geq v_3+x_3$ are dominating. Now, we notice that if $v_1+x_1\geq v_2+x_2>v_3+x_3$ and $x_2\neq 0$, then $x_1\leftarrow x_1+1$, $x_2\leftarrow x_2-1$ is feasible and yields a strictly better solution. Therefore, in an optimal solution, either $x_2=0$ or $v_2+x_2=v_3+x_3$. We can substitute x_2 in the problem formulation.

Now, notice that we can split the value of x_1 into two parts. The first part is the minimum increase caused by x_3 and constraints (1.7) and (1.8). The second part is the "additional" increase. Both parts are contributing to increase the objective on the $(v_i+x_i)^2$ but the first one also contributes by increasing the number of different symbols. These different kinds of increases are illustrated Figure 1.2.

We reformulate the objective of the optimization problem:

$$z(x)=(v_1+s+\max(0,t-(v_1-v_3)))^2+(v_2+\max(0,t-(v_2-v_3)))^2+(v_3+t)^2+7(v_3+t)$$

Where $x=(s,t)$ is a feasible solution.

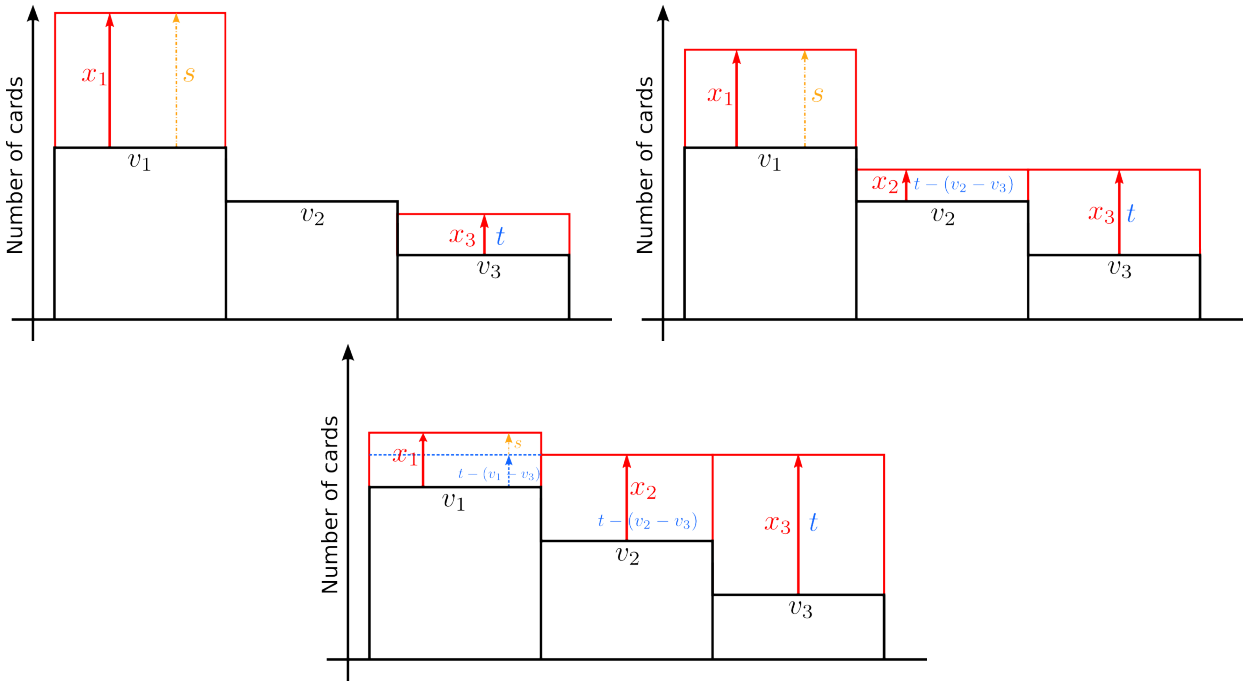


Figure 1.2: The different kinds of increases.

This yields a piecewise definition of z:

$$z(x) = \begin{cases} (v_1+s)^2 + v_2^2 + (v_3+t)^2 + 7(v_3+t) & \text{if } 0 \le t \le v_2 - v_3 \\ (v_1+s)^2 + 2(v_3+t)^2 + 7(v_3+t) & \text{if } v_2 - v_3 < t \le v_1 - v_3 \\ (v_3+s+t)^2 + 2(v_3+t)^2 + 7(v_3+t) & \text{if } v_1 - v_3 < t \end{cases}$$

Notice that $v_2 - v_3 \le t \le v_1 - v_3$, corresponds to $x_3 = t$ and $x_2 = t - v_2 + v_3$. In this domain, both of the values of v_2 and v_3 are increased at the same time. In order to increase t by δt, we need to use $2\delta t$ jokers. A similar result applies to the third domain of t.

We proceed to the following variable substitutions on the 3 domains: $t = r$, $t = v_2 - v_3 + \frac{u}{2}$ and $t = v_1 - v_3 + \frac{v}{3}$. Moreover, since $x_1 + x_2 + x_3 = n$ in any optimal solution, we can also substitute s:

$$z(x) = \begin{cases} (v_1+n-r)^2 + v_2^2 + (v_3+r)^2 + 7(v_3+r) & \text{if } 0 \le t \le v_2 - v_3 \\ (v_1+n-v_2+v_3-u)^2 + 2(v_2+\frac{u}{2})^2 + 7(v_2+\frac{u}{2}) & \text{if } v_2 - v_3 < t \le v_1 - v_3 \\ (n-v_1+v_2+v_3-\frac{2v}{3})^2 + 2(v_1+\frac{v}{3})^2 + 7(v_1+\frac{v}{3}) & \text{if } v_1 - v_3 < t \end{cases}$$

The function z is defined over a convex domain of $\mathbf{R}_+$ and is continuous and differentiable except on $v_2 - v_3$ and $v_1 - v_3$ but its derivatives have finite limits to the right and to the left.

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \begin{cases} 4r + 2(v_3 - v_1 - n) + 7 & \text{if } t \le v_2 - v_3 \\ 3u - 2n - 2v_1 + 4v_2 - 2v_3 + \frac{7}{2} & \text{if } v_2 - v_3 < t \le v_1 - v_3 \\ \frac{4}{3}v - \frac{4}{3}n + \frac{8}{3}v_1 - \frac{4}{3}v_2 - \frac{4}{3}v_3 + \frac{7}{3} & \text{if } v_1 - v_3 < t \end{cases}$$

The derivatives of z are strictly increasing, hence the maximum of z is obtained on an extreme point of its domains. Hence, z is maximum either on $t = 0$ (always feasible) or on $t = v_2 - v_3$

(feasible if $n \geq v_2 - v_3$), or on $t = v_1 - v_3$ (feasible if $n \geq 2v_1 - v_2 - v_3$) or on the largest value of t which is feasible. Let $t_{\max}$ be this value. By using the integrality constraints, we compute $t_{\max}$:

$$t_{\max} = \begin{cases} n & \text{if } n \in [0\ ;\ v_2 - v_3] \\ v_2 - v_3 + \lfloor \frac{1}{2}(n - v_2 + v_3) \rfloor & \text{if } n \in [v_2 - v_3\ ;\ 2v_1 - v_2 - v_3] \\ v_1 - v_3 + \lfloor \frac{1}{3}(n - 2v_1 + v_2 + v_3) \rfloor & \text{if } n \geq 2v_1 - v_2 - v_3 \end{cases}$$

We enforce the problem formulation by adding a constraint $t \leq t_{\max}$. This does not change any of the previous results and we obtain the optimal solution of the new relaxed problem:

$$z(x^*) = \max(z(t) | t \in \{0,\ v_2 - v_3,\ v_1 - v_3,\ t_{\max}\} \text{ and } t \leq t_{\max})$$

For all $t \in \{0,\ v_2 - v_3,\ v_1 - v_3,\ t_{\max}\}$ verifying $t \leq t_{\max}$ the values assigned to x_1, x_2 and x_3 are integral. Hence, the optimal solution of the relaxation is feasible for the original problem and therefore is an optimal solution to the original problem.

In order to find out the optimal solution of the problem, we only have to evaluate $t_{\max}$ and at most 4 values of z corresponding to the 4 strategies: "all jokers on stack 1", "$v_2 - v_3$ jokers on stack 3, the rest on stack 1", "$v_1 - v_3$ jokers on stack 3, $v_1 - v_2$ jokers on stack 2, the rest on 1" and "Place as many jokers as possible to match the sizes of the three stacks and place the remaining $\{0, 1$ or $2\}$ jokers on stack 1". This algorithm is strongly polynomial and runs in linear time in the input size. Its computational complexity is $\mathcal{O}(\log(v_{max}) + \log(n))$ (or $\mathcal{O}(1)$ if we suppose that reading integers and performing arithmetical operations are done in constant time). The player can apply this algorithm very quickly and easily. We can study this problem further and eliminate cases but this would not improve the complexity of the algorithm.

1.2.4 Conclusion

We solved a simple problem and analyzed the complexity of proposed algorithms. On this example, even though one can argue that a high-multiplicity encoding is "too compact", it is actually the only natural encoding of the input and only a polynomial time algorithm in the size of the high-multiplicity input yields an efficient solution to this problem. The fact that all items exist individually in reality and we have to manipulate them is not an acceptable reason to discard high-multiplicity encoding.

Remark that the approaches used to solve the problem are quite different depending on the encoding. Using a unary encoding of the sizes, the natural approach is algorithmic and problem centered. Using a high-multiplicity encoding, the approach is more mathematical and arithmetic oriented.

1.3 Literature review

The term high-multiplicity has been first coined by Hochbaum and Shamir (1991). In their paper, they define a high-multiplicity scheduling problem as:

Definition 1.2 (Hochbaum and Shamir (1991))**.** A *high-multiplicity* scheduling problem consists of many jobs which can be partitioned into few groups, where all the jobs within each group are identical. The number of jobs of a specific type is called the *multiplicity*.

In this thesis, we are interested in problems in which the input is described using a high-multiplicity encoding. A high-multiplicity encoding is an encoding in which all identical jobs are specified at once by giving the parameters of a representative job and the number of such jobs (the *multiplicity*). This is a slightly more general definition than the one from Hochbaum and Shamir (1991) in the sense that we do not require that there are *few* groups. Remark that a job-by-job schedule is not a polynomial certificate for such a problem since the number of jobs can be pseudo-polynomial in the input size. More specifically, with such an input, provided the number of jobs is not bounded, by increasing the multiplicities we can always build an input such that the number of jobs is pseudo-polynomial in the input size.

Notice that these definitions can be generalized to other kinds of problems. Especially, they apply directly to packing problems.

In their paper, Hochbaum and Shamir (1991) were the first to introduce formally high-multiplicity decision problems, although previous studies have already identified such problems and the related issues.

In the following sections, we present a state-of-the-art on high-multiplicity scheduling.

1.3.1 Early results

In this section, we introduce early results on high-multiplicity problems. These results were obtained by their authors before the definition of high-multiplicity by Hochbaum and Shamir (1991). They are dealing with high-multiplicity scheduling problems as well as the more general class of problems with succinct encoding of the input.

In the scheduling literature, Psaraftis (1980) proposed a dynamic programming approach to sequence groups of identical jobs on a single machine in order to minimize the total processing cost. Using the existence of identical jobs, they were able to significantly improve the complexity from Held and Karp (1962). Dessouky et al. (1990) considered the problem of scheduling identical jobs on uniform parallel machines and studied this problem with the objectives $f_{\max}$ and $\sum f_i$ with non-decreasing functions. They proposed algorithms to solve $Q|p_j = 1|f_{\max}$, $Q|p_j = 1|\sum f_j$ and improved these algorithms on several particular cases. They consider an input, with size $\mathcal{O}(n+m)$. Their algorithms are polynomial in this input size but we can see that, with a slight modification of their model, their results can be generalized to be polynomial in the size of a high-multiplicity encoding. Especially, $Q|p_j = 1|C_{\max}$ can be solved in $\mathcal{O}(M \log M)$ time, where M is the number of different machine speeds.

In the graph theory literature, Lengauer (1982) studied the complexity of problems occurring in very large scale integrated circuitry (VLSI) design when the circuit is given using a compact representation. Galperin and Wigderson (1983), motivated by the same applications, introduced succinct representations of graphs and studied the relation between the complexity of the "classical" version of a problem and the complexity of the same problem using a succinct representation. These works were followed up by several other works by Lengauer and Wanke (1988), Lozano and Balcázar (1990), Papadimitriou and Yannakakis (1986), Turán (1984), with both the ideas of solving these problems and finding succinct representations for graphs with given properties. Among other applications, Cosmadakis and Papadimitriou (1984) studied a generalization of the traveling salesman problem, where each city is visited several times.

1.3.2 High-Multiplicity scheduling: definition and framework

In their paper, Hochbaum and Shamir (1991) introduce the concept of high-multiplicity scheduling and propose a strongly polynomial algorithm for $1|p_j = 1, HM|\sum w_j U_j$ by reducing the problem to a transportation problem. They also claim that $1|p_j = 1, HM|\sum w_j T_j$ is polynomial, which they prove in Hochbaum et al. (1992). In Hochbaum and Shamir (1990), they extend their results to $1|p_j = 1, r_j, HM|\sum w_j U_j$ and highlight the difficulties of high-multiplicity scheduling problems.

Following these papers, high-multiplicity scheduling became an active field in scheduling. Several authors studied high-multiplicity scheduling problems. See Table 1.1 for a small survey. Brauner et al. (2005) remarked that traditional frameworks are not sufficient to characterize algorithms for high-multiplicity scheduling problems. Especially because there might be efficient ways to solve such problems while it might not be easy to provide a schedule which is polynomial in the input size. Moreover, while the definition of a decision problem is clear, the definition of an optimization problem can vary from one author to another with surprising consequences. For instance, Clifford and Posner (2001) require an algorithm for an optimization problem to output a schedule composed of starting dates and groups of jobs and/or machines, with the jobs in a group being processed for the exact same duration on the machines of the group. As a result, they claim that $Q_2|pmtn, HM|\sum C_j$ is in $EXP\backslash P$ for optimization because they are able to give an input in which all jobs are processed for different durations on each machine. This result may hold with their definition of an optimization problem but there are actually several other efficient ways to describe a schedule in polynomial space such as giving the sequence of jobs or a function which computes any job starting time in polynomial time.

Brauner et al. (2005, 2007) provide a detailed framework to cope with such issues and classify algorithms for high-multiplicity scheduling problems. This framework is very useful since it removes the ambiguity over the definitions of a recognition and an optimization problem. It emphasizes the fact that there are many ways to compute and describe a schedule and allows to refining the complexity analysis of high-multiplicity scheduling problems by taking these facts into account.

We briefly present the main tools of their framework. Let I be an instance of a scheduling problem. On this problem and on instance I, let $\mathcal{F}_I$ be the set of feasible solutions (schedules) and $f_I : \mathcal{F}_I \to \mathbf{R}$ the objective function. Brauner et al. (2005) define the following problems:

RECOGNITION PROBLEM – $\mathcal{SP}_1$:
Input: An instance I and $K \in \mathbf{R}$.
Output: Yes if there is a schedule $S \in \mathcal{F}_I$ such that $f_I(S) \leq K$. No otherwise.

EVALUATION PROBLEM – $\mathcal{SP}_2$:
Input: An instance I.
Output: The minimum value of f_I over $\mathcal{F}_I$.

OPTIMIZATION PROBLEM – $\mathcal{SP}_3$:
Input: An instance I.
Output: A schedule $S \in \mathcal{F}_I$ which minimizes f_I over $\mathcal{F}_I$.

There are several ways to describe a schedule. Depending on the way considered, they propose two different classifications of algorithms solving problems from $\mathcal{SP}_3$. The class is devoted to schedules defined in extension; for instance, a set of starting times is a schedule in extension.

Definition 1.3 (Brauner et al. (2005))**.** An algorithm $\mathcal{A}$ is a *list-generating* algorithm for $\mathcal{SP}_3$ if,

for every instance of $\mathcal{SP}_3$, $\mathcal{A}$ successfully outputs the values $(\pi^1, S(\pi^1)), (\pi^2, S(\pi^2)), \ldots, (\pi^n, S(\pi^n))$ where S is an optimal schedule and $(\pi^1, \pi^2, \ldots, \pi^n)$ is a permutation of the job-set.

Then, let $|I|$ be the input size, one can classify the algorithms as follows:

Definition 1.4 (Brauner et al. (2005))**.** Let $\tau(i)$ be the running time required by algorithm $\mathcal{A}$ to output the first i elements of the schedule and $\tau(0) = 0$. A list-generating algorithm $\mathcal{A}$ for $\mathcal{SP}_3$ runs in:

- *polynomial total time* if $\tau(n)$ is polynomially bounded in n and $|I|$
- *polynomial incremental time* if $\tau(j) - \tau(j-1)$ is polynomially bounded in j and $|I|$ for $j = 1, 2, \ldots, n$
- *polynomial delay* if $\tau(j) - \tau(j-1)$ is polynomially bounded in $|I|$ for $j = 1, 2, \ldots, n$
- *polynomial time* if $\tau(n)$ is polynomially bounded in $|I|$

Polynomial total time corresponds to the classical definition of polynomial time for non-high-multiplicity scheduling problems.
Polynomial incremental time and polynomial delay are stronger requirements. Especially, polynomial delay is a key requirement when all jobs have to be processed separately in a manufacturing process: when a machine is available we ask which is the next job to be processed and get the answer in polynomial time.
We can interpret polynomial incremental time as a relaxation of polynomial delay where we have the additional ability to look at the history (the definition is actually larger than this).
For a list-generating algorithm, polynomial time only makes sense for non-high-multiplicity scheduling problems since for any high-multiplicity scheduling problem we can build an instance I such that n is pseudo-polynomial in $|I|$.

A list-generating algorithm can be used to generate an optimal schedule. Yet, we cannot generally derive a polynomial time algorithm for $\mathcal{SP}_2$ from a polynomial delay list-generating algorithm since the schedule generated is pseudo-polynomial in the input size.

An immediate result from these definitions is that a polynomial time list-generating algorithm for a non-high-multiplicity scheduling problem immediately yields a polynomial total time algorithm for the high-multiplicity variant of this problem. In such cases, the challenge is to find more efficient algorithms, such as polynomial incremental time or polynomial delay list-generating algorithms.

When a schedule S is not described in extension but using a mapping they propose other tools:

Definition 1.5 (Brauner et al. (2005))**.** A *pointwise algorithm* for $\mathcal{SP}_3$ is an algorithm $\mathcal{A}$ such that:

- on the input (I, j), algorithm $\mathcal{A}$ outputs $S(j)$, the starting time of job j, for $j = 1, 2, \ldots, n$
- $\{S(j) : j \in \{1, 2, \ldots, n\}\}$ defines an optimal schedule for I

Remark that the target set of S is not defined and the second point of the definition is somehow vague but thanks to this very general definition, it covers many ways to describe a schedule.

The efficiency of a pointwise algorithm is then characterized by the classical definition of polynomial time: an algorithm is *pointwise polynomial* if the functions S can be evaluated in polynomial time.

We sum this up as:

Definition 1.6. An algorithm is *pointwise polynomial* if it provides an implicit representation of an optimal schedule which can be used to compute the machine(s) and position in the sequence of any job in polynomial time.

For instance, a pointwise polynomial algorithm can group jobs and machines and provide the sequence obtained.

From a pointwise polynomial algorithm, we can easily derive a polynomial delay list-generating algorithm. In Brauner et al. (2007), they further refine this definition for several usual target sets of S.

1.3.3 Main results

Table 1.1 summarizes a few results on high-multiplicity scheduling problems from the literature. In the presentation of these results, we integrate them in the framework from Brauner et al. (2005). Column $\mathcal{SP}_3$, "Pointwise Polynomial" either means that the algorithms are pointwise polynomial or that they can be adapted to obtain a pointwise polynomial algorithm.

Table 1.1 is not an inventory of all work on high-multiplicity scheduling problems but rather on common problems, well studied in the non high-multiplicity case and easy to denote. In the following, we highlight a few interesting results from Table 1.1 and other results which do not appear in this table.

Problem $1|r_{jk}, HM|C_{\max}$ was studied by Brauner et al. (2007). In this problem, jobs are grouped according to their release dates and processing times. Jobs in a same group have the same processing times and their release dates are given by a linear function for the group. The Earliest Release Date (ERD) is optimal for this problem. Hence, it immediately gives a polynomial delay list-generating algorithm. Moreover, given a job, by analyzing all groups, we can easily find, in polynomial time, the position of this job in an optimal sequence, yielding a pointwise polynomial algorithm. Using any of these two algorithms, we can compute the optimal makespan in pseudo-polynomial time. However, the problem of computing the optimal makespan in polynomial time for this problem is still open.

There are substantial differences between "classical" and high-multiplicity scheduling problems. For instance, while it is often easy to prove that a scheduling problem belongs to $\mathcal{NP}$, it is difficult to prove that a high-multiplicity scheduling problem is in $\mathcal{NP}$ in many cases. The previous example, $1|r_{jk}, HM|C_{\max}$, is a good illustration of this fact. An even more interesting illustration is the cutting stock problem. In this problem, we have paper rolls, all with the same width, which can be cut into several pieces and there are demands on paper rolls with given width. The aim is to cut paper rolls in order to satisfy the demands and minimize the trim loss. This problem is actually a high-multiplicity bin packing problem. Hence, it has been known to be $\mathcal{NP}-hard$ since its introduction by Eisemann (1957). However, it is only very recently that this problem was proven to be in $\mathcal{NP}$ by Eisenbrand and Shmonin (2006). They proved similar results as Carathéodory bounds (Carathéodory 1907) but on integer cones and applied their result to integer programming. Then, using the integer program from Gilmore and Gomory (1961) they proved that there exists an optimal solution using a number of patterns which is polynomial in the size of the input. Such an approach can be used to prove that other high-multiplicity scheduling problems belong to $\mathcal{NP}$ by formulating them as integer programs using patterns (column generation approach).

High-multiplicity scheduling problems have also been investigated with the aim of designing approximation algorithms. Agnetis (1997) proposed an approximation algorithm for the no-wait

Problem	$\mathcal{SP}_1$ (Recognition)	$\mathcal{SP}_2$ (Evaluation)	$\mathcal{SP}_3$ (Optimization)	Evaluation complexity	Reference
$1\|r_j, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s \log s)$	Brauner et al. (2007)
$1\|r_{jk}, HM\|C_{\max}$	?	?	Pointwise polynomial	?	Brauner et al. (2007)
$1\|\text{Fs}, s = 2\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	Weakly polynomial	Billaut and Sourd (2009)
$1\|coup - Task, a_j = b_j = 1, L_j = L\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(\log n)$	Orman and Potts (1997)
$1\|coup - Task, a_j = a, b_j = b, L_j = L\|C_{\max}$	?	?	?	?	Chapter 3, Ahr et al. (2004), Baptiste (2010), Gabay et al. (2012a)
$1\|coup - Task, a_j = a, b_j = b, L_j = L, cycl\|Ct_{\min}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(\log L)$	Lehoux-Lebacque et al. (2009)
$1\|\text{Fse}, HM\|C_{\max}$	$\mathcal{NP} - Complete$	$\mathcal{NP} - Complete$	$\mathcal{NP} - Complete$	–	Rapine and Brauner (2013)
$1\|\text{Fse}, s > k, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(sk + k^4)$	Chapter 2, Brauner and Rapine (2010)
$1\|\text{Fse}, fixed - k, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(\log \gamma_k + \log n)$	Chapter 2, Gabay et al. (2013f)
$1\|p_j = 1, HM\|\sum w_j U_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s \log s)$	Hochbaum and Shamir (1990)
$1\|p_j = 1, HM\|\sum w_j T_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	Weakly polynomial	Hochbaum et al. (1992)
$1\|s_{jk}, fixed - s, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(poly(\log n))$	van der Veen and Zhang (1996)
$P\|pmtn, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s)$	Brauner et al. (2007)
$P\|HM\|\sum C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s^2)$	Clifford and Posner (2001)
$P\|pmtn, HM\|\sum C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s^2)$	Clifford and Posner (2001)
$Pm\|r_i, HM\|\sum w_i C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	Polynomial	Granot et al. (1997)
$P\|s = 2, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}((\log L)^3)$	McCormick et al. (2001)
$Q2\|pmtn, HM\|\sum C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s \log s)$	Brauner et al. (2007)
$Q\|pmtn, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}((s+h)\log(s+h))$	Clifford and Posner (2001)
$Q\|HM\|\sum C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s(h+s)\log(h+s))$	Clifford (1997)
$Qm\|r_i, p_j = 1\|\sum w_{ij} C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	Polynomial	Granot et al. (1997)
$R\|pmtn, HM\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	Polynomial	Clifford and Posner (2001)
$R\|M_J\|\sum C_j$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	Polynomial	Clifford and Posner (2001)
$F2\|no - wait, HM\|Ct_{\min}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(s^2)$	Brauner et al. (2007)
$O\|p_{ij} = 1, t_{ijk} = T\|C_{\max}$	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(\log L)$	Rayward-Smith and Rebaine (1992)
Total deviation JIT	?	?	Total polynomial	$\mathcal{O}(poly(n,s))$	Kubiak and Sethi (1991, 1994)
Maximum deviation JIT	$co - \mathcal{NP}$	?	Total polynomial	$\mathcal{O}(n \log n)$	Brauner and Crama (2004), Steiner and Yeomans (1993)
Maximum deviation JIT, fixed s	$\mathcal{P}$	$\mathcal{P}$	Polynomial delay	$\mathcal{O}(\log n)$	Brauner and Crama (2004)
Lot-size scheduling with two types	$\mathcal{P}$	$\mathcal{P}$	Pointwise polynomial	$\mathcal{O}(N^2)$	Kovalyov et al. (2002)
Cutting Stock	$\mathcal{NP} - Complete$	$\mathcal{NP} - Complete$	$\mathcal{NP} - Complete$	–	Eisenbrand and Shmonin (2006)

Table 1.1: Complexity of some High-Multiplicity scheduling problems. We denote by L, the biggest integer of the input, by N the number of integers in the input and by s and h the number of job and machine types.

flow shop problem with high-multiplicity. The performance of this algorithm improves with the number of identical jobs per class. Clifford and Posner (2000) investigated earliness-tardiness scheduling problems and proposed approximation algorithms. Some algorithms from Clifford and Posner (2001) for minimizing the sum of completion times are improved in Filippi and Romanin-Jacur (2009). They also propose asymptotically optimum approximation algorithms for $P|M_J|C_{\max}$, $Q|M_J|C_{\max}$ and $R|M_J|C_{\max}$ (where M_J means that multiplicities are allowed on the jobs but not on the machines). Filippi (2010) designed an asymptotically optimal algorithm for $R|M_J|\sum w_jC_j$. Serna and Xhafa (2008) present parallel approximation algorithms for two problems proven to be polynomial by Granot et al. (1997) (these problems cannot be solved efficiently in parallel unless $\mathcal{P} = \mathcal{NC}$ where $\mathcal{NC}$ is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors).

The high-multiplicity bin packing (cutting stock) problem has been thoroughly investigated by Filippi and Agnetis (2005) and Stille (2008). Filippi and Agnetis (2005) proposed an algorithm which provides a solution requiring at most $s - 2$ bins more than the optimal solution (where s is the number of different weight values). Stille (2008) proposed a combinatorial algorithm computing a solution using at most one more bin than the optimal solution. Jansen and Solis-Oba (2010) obtained a similar result using an integer programming approach, an adaptation of Lenstra's theorem (Lenstra 1983) and rounding. All these algorithms are polynomial when s, the number of distinct types, is fixed.

In graph theory, Balcázar et al. (1992, 1996) provided further results on combinatorial problems on succinct graphs and proposed a framework to study these problems. On the high-multiplicity traveling salesman problem, Grigoriev and van de Klundert (2006) lead a "sensitivity" analysis. They provided results on the improvement of the average tour length when all multiplicities are multiplied by a common factor and how and when the structure of an optimal tour can be derived from tours with smaller multiplicities.

Other fields of applications of high-multiplicity scheduling are batch scheduling (Selvarajah and Steiner 2006), scheduling in robotic cells (Brauner 2008, Crama and Van De Klundert 1997) or telecommunication networks (Ciaschetti et al. 2007, Detti 2008, Detti et al. 2005, 2009).

The opportunities and issues related to high-multiplicity scheduling problems are also often identified even if the specific term "high-multiplicity" is not used. For instance, in problems with similar jobs (Drozdowski and Lawenda 2008), identical jobs (Ahr et al. 2004, Baptiste et al. 2004, Brucker et al., Drozdowski and Lawenda 2008), or unit-time jobs (Kubiak and Timkovsky 1996, Timkovsky 1998).

Most polynomial algorithms are obtained by scheduling groups of jobs. Several authors, such as van der Veen and Zhang (1996), also study the fix parameter tractability of problems. A very important result in such approach is the theorem from Lenstra (1983), proving that it can be decided in polynomial time whether an integer program has feasible solutions when the number of variables is fixed. As a consequence, by binary search, an integer program can be solved in polynomial time when the number of variables is fixed. Eisenbrand (2003) further improved this result by showing that when both the number of variables and constraints are fixed, an integer program can be solved in linear time.

1.4 Outline of the thesis

The aim of this thesis is to provide insights on how to cope with high-multiplicity scheduling problems. We also seek to contribute to scheduling and packing in general. We solve some high-

multiplicity scheduling problems with different techniques and approaches. In the first part of the thesis we work on scheduling problems and focus on the scheduling aspects. The analysis of these problems mostly relies on scheduling tools. In the second part, we work both on scheduling and packing problems but we focus on the packing aspects and use packing approaches to cope with these problems.

In Chapter 2, we study a single machine scheduling problem with forbidden start and completion times. We prove that this problem is in $\mathcal{NP}$, even with a high-multiplicity encoding of the input and we propose both a polynomial-time algorithm for a particular case and a polynomial-time algorithm for the general case when the number of unavailabilities is fixed.

In Chapter 3, we study the identical coupled-task scheduling problem. This problem is somehow a very "pure" high-multiplicity scheduling problem as the input is very short (4 integers) and all jobs are identical. Yet this is a hard combinatorial problem and its complexity status is open for more than 15 years. We provide lower and upper bounds for this problem. We show that optimal solutions of this problem use very complicated structure and that finding such solutions is very challenging, regardless of the encoding of the input. We present a polynomial algorithm which is capable of computing solutions using some of these structures.

In Chapters 4 and 5, we study the bin stretching problem which is a semi-online variant of the scheduling problem $Pm||C_{max}$. Chapter 5, we propose an approximation algorithm with performance guarantee for this problem. The algorithm improves the best known upper bound for this problem. We do not consider the high-multiplicity setting since jobs have to be scheduled one by one. However, our algorithm relies on techniques of partitioning the jobs and the machines into few groups and packing them according to these groups. In Chapter 5, we improve the best known lower bound for the bin stretching problem by modeling the problem of finding lower bounds as a two-player game and solving this game using game theory and computer science techniques. This approach is different from "traditional" approaches used in online scheduling to improve lower bounds since it does not use layering techniques but is computational.

In Chapter 6, we introduce and study a variant of the vector packing problem and experiment several heuristics to build feasible solutions for this problem. This problem has been inspired by the Google EURO/ROADEF Challenge 2012 in which Sofia Zaourar and I took part and were qualified for the final phases.

In Chapter 7, we study packing problems in general and present a reduction algorithm which is very general and can be applied to many packing problem such as bin packing, vector packing, multidimensional bin packing and others. This reduction algorithm is well suited to be integrated in assignment based branch and bound approaches for packing problems, branching heuristics or heuristics and can be computed in polynomial time in the number of bins and the input size, even with a high-multiplicity encoding of the input.

The Chapters of this thesis are self-contained and can be read separately and in any order.

Part I

Scheduling

Chapter 2

High-Multiplicity Scheduling on One Machine with Forbidden Start and Completion Times

Résumé : Nous présentons dans ce chapitre le problème d'ordonnancement sur une machine avec indisponibilités des opérateurs et encodage high-multiplicity de l'instance. Nous exposons un certificat polynomial pour ce problème et un algorithme polynomial pour les instances dans lesquelles le nombre de types de tâches est supérieur au nombre d'indisponibilités. Nous montrons enfin que ce problème peut être résolu en temps polynomial lorsque le nombre de périodes d'indisponibilités est un paramètre fixé.

Abstract: We are interested in a single machine scheduling problem where jobs can neither start nor end on some specified dates, and the aim is to minimize the makespan[1]. This problem models the situation where an additional resource, subject to unavailability constraints, is required to start and to finish a job. We consider in this chapter the high-multiplicity version of the problem, when the input is given using a compact encoding. We present a polynomial time algorithm for large diversity instances (when the number of different processing times is greater than the number of forbidden instants). We also show that this problem is Fixed-Parameter Tractable when the number of forbidden instants is fixed, regardless of jobs characteristics.

2.1 Introduction

We consider a scheduling problem on one machine where a set of instants is given, such that no job is allowed to start or to complete at any of these instants. We refer to such an instant as a *forbidden start & end instant* (Fse). Forbidden instants may arise when jobs need some additional

[1]The results presented in this chapter are joint work with Christophe Rapine. They were presented during an invited talk in Maastricht (Gabay 2014) and in conferences (Gabay et al. 2013b,c). The content of this chapter is the same as the article [Gabay et al., 2013f] which we have submitted to an international journal for publication.

resources at launch and completion and these resources are not continuously available. This may be the case if the additional resources are shared with other activities. For example, consider the situation where the jobs are processed by an automated device during a specified amount of time, but a qualified operator is required on setup and completion. While the device is continuously available, the operators have days off and other planed activities. On these days, jobs can be performed by the device, but none can start or complete. We encountered this problem in chemical industry through a collaboration with the *Institut Français du Pétrole*. In their problem, jobs were chemical experiments whose durations typically last between 3 days and 3 weeks. A chemist is required on jobs start and completion to control the process. Each intervention of the chemist can be performed within an hour, but requires of course a chemist to be available and present in the laboratory. For more details on this application, we refer the reader to Brauner et al. (2009b) and Rapine et al. (2012).

Notice that, contrary to a classical unavailability constraint, the machine can be processing a job during an Fse instant, as long as it started its execution before the forbidden instant and will complete after it. We restrict to integer values for the data and to schedules where all the jobs start and complete at integer instants. The objective is to minimize the makespan $C_{\max}$. Using Graham notations, the problem is denoted by $1|\text{Fse}|C_{\max}$. As an example, consider the instance where instants 3, 4, 6 and 9 are Fse instants and 5 jobs have to be scheduled: a and b of duration 1, c and d of duration 2 and e of duration 4. On Figure 2.1 and 2.2, forbidden instants are represented on the time axis by dashed rectangles. The sequence of jobs (a, e, c, b, d) leads to an idle-free schedule represented Figure 2.1; the makespan of this schedule is 10. On Figure 2.2, we have represented the scheduling of the jobs according to the LPT sequence (e, d, c, b, a), that is in non-increasing order of the processing times. In order to respect the forbidden instants, two idle slots are used in the schedule. One can check that the SPT sequence (a, b, c, d, e) leads to a worse schedule, of makepsan 14.

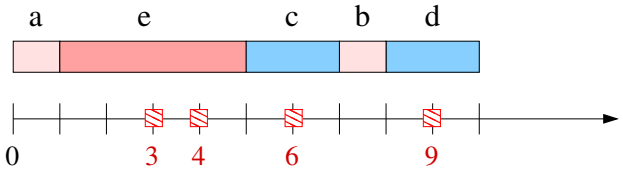


Figure 2.1: Sequence (a, e, c, b, d). The schedule is idle-free and completes at time 10.

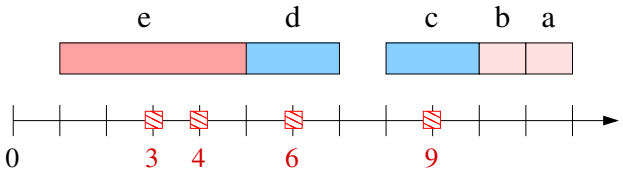


Figure 2.2: Sequence (e, d, c, b, a). The schedule completes at time 12.

The problem of scheduling jobs on a single machine where a set of time slots is forbidden for starting or completing the jobs has been first investigated by Billaut and Sourd (2009). They considered the case where it is forbidden to start new jobs on some given time slots, namely the Fs instants (for *forbidden start*). They proved that minimizing the makespan is polynomially

solvable if the number of forbidden start times is fixed, and $\mathcal{NP}$-hard in the strong sense if this number is part of the input. Their algorithm runs in time $\mathcal{O}(n^{2k^2+2k-1})$, where k denotes the number of Fs instants and n is the number of jobs. They also established that if there are at least $2k(k+1)$ distinct processing times of the jobs in the instance, then an idle-free schedule exists. Rapine and Brauner (2013) generalized this results: they established that having $k+1$ distinct processing times is a sufficient condition to ensure the existence of an idle-free schedule in presence of k Fse instants. Such an optimal schedule can be found in $\mathcal{O}(k^3 n)$. As a consequence, the overall complexity to solve the problem for a fixed number of forbidden instants is reduced to $\mathcal{O}(n^k)$. Chen et al. (2013) consider the same problem with a different objective function, namely the total completion time.

High-multiplicity encoding

The number of *types* of jobs, that is the number of different job durations, play a central role in the above mentioned results. Hence, it is natural to consider a compact encoding where similar jobs are grouped together. The problem then falls in the field of high-multiplicity Scheduling introduced by Hochbaum and Shamir (1991). Compared to a traditional encoding, where each job is described, in a high-multiplicity (HM) encoding, each *type* is described only once, along with its multiplicity (the number of jobs of this type). Thus the size of a HM encoding depends linearly on the number of types but only logarithmically on the number of jobs. As a consequence, a polynomial time algorithm under the standard encoding may become exponential under a HM encoding of the input, which is the case of the previously mentioned algorithms. HM scheduling and more generally HM combinatorial optimization has become an active domain in recent years, see (Brauner et al. 2005, Clifford and Posner 2001, Filippi and Agnetis 2005, Filippi and Romanin-Jacur 2009).

The goal of this chapter is to explore the complexity of problem $1|\text{Fse}|C_{\max}$ under a high-multiplicity encoding of the input. We show that essentially the main results established in the literature under a standard encoding remain valid under a HM encoding. Specifically, we propose in Section 2.2 a polynomial time algorithm for large diversity instances, that is when the number of types is greater than the number of Fse instants. In Section 2.3, we also prove that the general problem remains polynomial when k is fixed. We first introduce the following notations which will be used in the remaining of the chapter.

Notations

Throughout the chapter k denotes the number of Fse instants in the instance. Let γ_i be the i-th Fse instant with $\gamma_1 < \gamma_2 < \cdots < \gamma_k$. We denote by $\mathcal{F} = \{\gamma_1, \ldots, \gamma_k\}$ the set of the Fse instants. Two jobs are of different types if and only if their processing times are different. The number of types of jobs in the instance is denoted by s. Without loss of generality, we index the types by decreasing order of the processing times of their jobs. The set of jobs to schedule is represented in a HM encoding by a multiplicity vector $(m_1, \ldots, m_s)$, together with a processing times vector $(p_1, \ldots, p_s)$, where m_i and p_i are respectively the number of jobs of the ith type and its corresponding processing time. The number of jobs is $n = \sum_{i=1}^{s} m_i$. The instance of Figures 2.1 is thus represented by the processing times vector $(4, 2, 1)$ and the multiplicity vector $(1, 2, 2)$. A job is said to *cross* an Fse instant γ_i if it starts its processing before γ_i and ends after γ_i. For instance in Figure 2.1, the job e crosses the first two Fse instants.

We denote by $|x|$ the size of the input under a HM encoding. We have $|x| = \mathcal{O}(s \log n + s \log p_1 +$

$k \log \gamma_k$). Hence $|x|$ can be $\mathcal{O}(s \log n)$ while the algorithm proposed in Rapine and Brauner (2013) runs in time $\mathcal{O}(k^3 n)$, which can be exponential with respect to $|x|$.

In HM scheduling, it may not be obvious to determine whether schedules can be described with a compact encoding, *i.e.* polynomial in $|x|$. For the problem we consider, it is readily that the schedule of the jobs between two forbidden instants is immaterial (provided unnecessary idle-times are not inserted). As a consequence any schedule has a polynomial encoding as a sequence of k vectors $(m_i^1, \ldots, m_i^s)$ and k pairs (j_i, s_i), where m_i^j is the number of jobs of type j scheduled between γ_{i-1} and γ_i; j_i is the job crossing γ_i and s_i its starting time.

An instance is denoted by $x = (N, \mathcal{F})$ where N is the set of jobs. We say that an instance is of *large diversity* if $s > k$, that is, if the number of distinct types is greater than the number of forbidden instants. In the reverse situation, we say that the instance is of *small diversity*.

2.2 A polynomial time algorithm for large diversity instances

In this section, we design a polynomial algorithm for large diversity instances. Rapine and Brauner (2013) proved that, in such cases, there exists and idle-free schedule:

Theorem 2.1 (Rapine and Brauner (2013))**.** *If $s > k$ and $0, p(N) \notin \mathcal{F}$, then there exists a feasible schedule without idle time.*

They also presented an algorithm, called L-partition, finding an idle-free schedule for large diversity instances in $\mathcal{O}(k^3 n)$ time, where $n = \sum_{i=1}^{s} m_i$ is the number of jobs. Although linear in the number of jobs, this algorithm is not polynomial with a high-multiplicity encoding, except if the multiplicity of each type is bounded by a constant. In particular if only one job is associated with each type, the L-partition algorithm runs in time $\mathcal{O}(k^3 s)$. We use this fact in our approach.

To design a polynomial time algorithm under a HM encoding, we need to schedule more than one job at a time. We also need an efficient way to decompose the problem. Consider a large diversity instance $x = (N, \mathcal{F})$. Notice that Theorem 2.1 ensures that an optimal schedule is idle free, assuming that neither instant 0 nor instant $p(N)$ is forbidden. A schedule is said *partial* if only a subset of the jobs is scheduled. We introduce the following definition:

Definition 2.1. A partial schedule π is an *optimal prefix* if there exists an optimal schedule of the form $\pi\sigma$.

Consider a partial schedule π completing at time t. Looking at the definition, deciding if π is an optimal prefix may request to compute an optimal schedule for the whole instance. However, by Theorem 2.1, a sufficient condition for π to be an optimal prefix is that π is idle-free, and that the remaining instance $x' = (N', \mathcal{F}' = \mathcal{F} \cap [t, +\infty[)$ is a large diversity instance. Indeed, it guarantees the existence of an idle-free schedule σ for the remaining jobs to schedule after time t.

If we are able to find an optimal prefix π, the problem is reduced to finding an optimal schedule starting at time t on the remaining set N' of jobs. We can then look again for an optimal prefix π' on the remaining large diversity instance x'. However, for this decomposition to be efficient, we need to bound the number of times an optimal prefix is searched for. We say that a prefix π is *efficient* if it is optimal and crosses at least one forbidden instant. It is then immediate that at most k efficient prefixes need to be computed to build an optimal schedule.

Algorithm 2.1 finds an (efficient) optimal prefix. The main idea of the algorithm is to put aside initially one job of each of the $k+1$ largest types. Let B be this set of jobs. This reserve B is

Algorithm 2.1: Optimal Prefix Algorithm

Input: a large diversity instance $(N,\mathcal{F})$ with types indexed by decreasing order of processing times p_j.
Output: an optimal prefix π

```
{Update the multiplicities to keep one job of each of the first k+1 types in the set B}
set m_i = m_i − 1 for i = 1 to k+1
i = 1 ; t = 0 ; π = ∅ ;
while i ≤ s and t + m_i p_i < γ_1 do
   {Append the m_i jobs of type i to π}
   π = π(i, m_i) ; t = t + m_i p_i ; i = i + 1 ;
end while
if i > s then
   return π {Only k+1 jobs remain to schedule}
end if
{Append as many jobs of type i as possible, before γ_1}
α = ⌈(γ_1 − t)/p_i⌉ − 1 ; π = π(i, α) ; t = t + α p_i ;
{Extend π to complete after time γ_1}
for l = 1 to k+1 such that t + p_l ≥ γ_1 do
   if t + p_l ∉ F then
      {Use one job from B to cross γ_1}
      return π(l, 1)
   end if
end for
for l = 2 to k+1 such that t + p_l < γ_1 do
   if t + p_l + p_1 ∉ F then
      {Use one job from B and one job of type 1 to cross the forbidden instant(s)}
      return π(l, 1)(1, 1)
   end if
end for
```

used to ensure that the remaining instance is of large diversity. Notice that we can afford to use one of these jobs each time a forbidden instant is crossed. We call *additional* jobs the set $A = N\backslash B$. The algorithm iteratively schedules all the additional jobs of type 1, then all the additional jobs of type 2, and so on. Recall that types are indexed in decreasing order of the processing times, thus we simply follow a LPT sequence for the additional jobs. We keep scheduling additional jobs as long as they fit before the first forbidden instant γ_1. When this process halts on some index i, either only the jobs from the set B remain to schedule, or there is not enough room left before γ_1 to schedule all the additional jobs of the ith type. In the latter case, the algorithm schedules as much jobs of type i as possible before γ_1. Then, it tries to cross the forbidden instant γ_1. In order to keep a large diversity instance, we ensure that each job of B scheduled allows to cross at least one forbidden instant. This way the algorithm outputs an efficient prefix. In the other case, all additional jobs have been scheduled and the partial schedule returned is optimal but not efficient, since the first Fse instant is not crossed. However, we are in the situation where the remaining large diversity instance contains only one job per type, and we have exactly $k+1$ types. We can use the L-partition algorithm to solve it efficiently, in time $\mathcal{O}(k^4)$. The correctness

of the algorithm is summarized in the following lemma:

Lemma 2.1. *Given a large diversity instance $x = (N, \mathcal{F})$, Algorithm 2.1 delivers an optimal prefix π. In addition, if $x' = (N', \mathcal{F}')$ is the remaining instance to schedule, then x' is a large diversity instance and:*

1. *either $|\mathcal{F}'| < |\mathcal{F}|$, that is π is an efficient prefix,*
2. *or $|N'| = |\mathcal{F}| + 1$ and all the remaining jobs have distinct processing times.*

Proof. Let $(N', \mathcal{F}')$ be the instance remaining to schedule at the end of Algorithm 2.1. Recall that B denotes a set with exactly one job of the $k+1$ largest types of N and $A = N \backslash B$ is the set of the additional jobs. If only the set B remains to schedule at the end of the algorithm, then we are clearly in the second case of our claim: $|N'| = k+1$. Otherwise the algorithm has stopped the first loop on a type i such that all the additional jobs cannot be scheduled before γ_1. At this point, there is at least one unscheduled job of type i remaining in A, and possibly another in B, if $i \leq k+1$. Let $t < \gamma_1$ be the current completion time of the schedule, and consider the partition $B = \mathcal{S} \cup \mathcal{L}$ defined by $\mathcal{L} = \{j \in B \mid t + p_j \geq \gamma_1\}$ and $\mathcal{S} = B \backslash \mathcal{L}$. Notice that $\mathcal{L}$ is not empty as $t + p_i \geq \gamma_1$; in particular a job of type 1 belongs to $\mathcal{L}$. By construction the prefix algorithm tries to extend π in order to complete after the first forbidden instant γ_1. We have to prove that it will always succeed, and that $(N', \mathcal{F}')$ is a large diversity instance. We denote by s' the number of distinct types of jobs in the remaining instance x' and by $k' = |\mathcal{F}'|$ the number of FSE instants appearing after time t.

In the following, we show that if π completes after the lth forbidden instant, at most l jobs of B have been scheduled in π. As a consequence, $s' \geq |B| - l > k - l \geq k'$ and $(N', \mathcal{F}')$ is a large diversity instance. Consider the last two loops of the algorithm. If one job of $\mathcal{L}$ can be scheduled, the property clearly holds as π completes after time γ_1. If there is no such job, then for all jobs j of $\mathcal{L}$, $t + p_j$ is a forbidden instant while any job of $\mathcal{S}$ can be scheduled before time γ_1. Therefore a simple counting argument, illustrated Figure 2.3, ensures that there exists a job $s \in \mathcal{S}$ which can be scheduled at time t immediately followed by a job of type 1. If $t + p_s + p_1 \geq \gamma_2$, *i.e.* π completes after time γ_2, we are done. Otherwise, we have $t + p_1 < \gamma_2$. In this case $k' = k - 1$, while we apparently use 2 jobs of B. However, instant $t + p_1$ is forbidden; in fact we have $t + p_1 = \gamma_1$ and as a consequence $i = 1$. As we noticed, there is at least one unscheduled job of type i in A. Since $i = 1$, we can use an additional job of type 1, instead of using a job of type 1 from B. We have $s' \geq s - 1$ which completes the proof. □

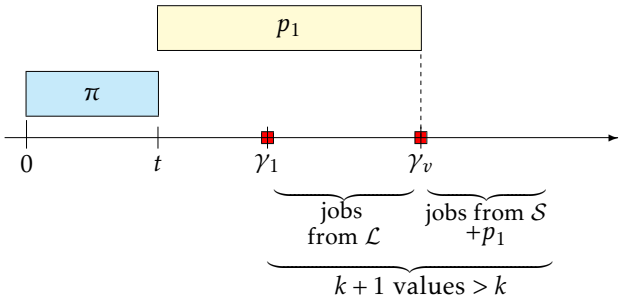


Figure 2.3: Counting argument: the first FSE instants can be crossed using one or two jobs from B.

In order to deliver an optimal schedule, we iteratively call the prefix algorithm on the remaining instance as long as we obtain an efficient prefix. Otherwise, we are in the second

case of Lemma 2.1, which corresponds to the basis of the recursion: we simply solve instance $x' = (N', \mathcal{F}')$ using the L-partition algorithm. Since N' contains at most $(k+1)$ jobs, the running time of the L-partition algorithm on this instance is in $\mathcal{O}(k^4)$. We have the following theorem:

Theorem 2.2. *Problem* $1|\text{Fse}|C_{\max}$ *is polynomial under HM encoding for large diversity instances, and can be solved in time* $\mathcal{O}(sk + k^4)$

Proof. From the above discussion, we only need to establish the time complexity of the algorithm, its correctness being a direct consequence of Lemma 2.1. We use the classic convention that basic operations on integers (addition, division…) are performed in constant time. Then the time complexity of Algorithm 2.1 is in $\mathcal{O}(k+s)$, which is in $\mathcal{O}(s)$ for large diversity instances. To solve Problem $1|\text{Fse}|C_{\max}$, we call Algorithm 2.1 on the set of unscheduled jobs as long as there is still some forbidden instants in the future or that this set is not reduced to B. Thus we have at most k calls to Algorithm 2.1, possibly followed by a call to L-partition algorithm on an instance containing at most $k+1$ jobs. Therefore the overall complexity is in $\mathcal{O}(sk + k^4)$. □

If 0 or $p(N)$ are in $\mathcal{F}$, then the same transformation as in Rapine and Brauner (2013) allows to obtain an optimal schedule. Note that, even under a traditional encoding of the instance, the optimal prefix algorithm has a better time complexity than the L-partition algorithm which runs in time $\mathcal{O}(k^3 n)$.

2.3 A polynomial time algorithm for a fixed number of Fse

In this section we establish that the problem $1|\text{Fse}|C_{\max}$ can be solved in polynomial time under a HM encoding of the instances if the number of forbidden instants is fixed, that is, if k is not part of the input. This result extends a theorem from Rapine and Brauner (2013) which establishes that $1|k - \text{Fse}|C_{\max}$ is polynomial under a standard encoding, that is, its complexity is polynomially bounded in n, the number of jobs (but not in s, the number of types). The rest of the section is devoted to proving the following theorem:

Theorem 2.3. *The problem* $1|\text{Fse}|C_{\max}$ *is Fixed Parameter Tractable for parameter k, even under high-multiplicity encoding of the input.*

Notice that if the instance is of large diversity, the optimal prefix algorithm (Algorithm 2.1, Section 2.2) can deliver an idle-free (and thus optimal) schedule in time $\mathcal{O}(s)$ for any fixed number k of forbidden instants. Hence we can focus on the case of small diversity instances. Our idea is to formulate the problem on small diversity instances as an integer linear program (ILP) with a fixed number of variables and constraints. Such an ILP can be solved in polynomial time, due to the following result from Eisenbrand (2003):

Theorem 2.4 (Eisenbrand (2003))**.** *An integer program of binary encoding length l in fixed dimension, which is defined by a fixed number of constraints, can be solved with* $\mathcal{O}(l)$ *arithmetic operations on rational numbers of binary encoding length* $\mathcal{O}(l)$.

For small diversity instances, we have by definition $s \leq k$. Thus, if the number of variables and the number of constraints in our ILP formulation are bounded by a polynomial in k, Theorem 2.4 implies that $1|\text{Fse}|C_{\max}$ is FPT with respect to parameter k. Clearly, to obtain such a formulation, we can not afford to introduce one decision variable (such as the completion time) or one constraint (such as avoiding to complete on a forbidden instant) for each job. Instead, as

already discussed in the HM encoding of a solution, see Section 2.1, we represent a solution by the number of jobs of each type scheduled between two consecutive forbidden instants. However, this representation of the solution is only suitable for idle-free schedules, since otherwise one has also to give the starting time of each block of jobs. To formulate the problem as an ILP, we take advantage of the fact that any large diversity instance admits an idle-free schedule, see Theorem 2.1. More precisely, we transform a small diversity instance I into a large diversity instance I' by adding dummy jobs as follows. Given an instance I composed of s types, I' is constituted of the following types:

- *Real jobs*. They are the jobs of I. We denote by p_i and m_i the processing time and the multiplicity of the type i, for $i = 1, \ldots, s$.
- *Optional jobs*. We add $k+1$ types to ensure that there exists an idle free schedule. For $i = s+1$ to $s+k+1$, type i has a processing time $p_i = i - s$ and its multiplicity is unbounded.

The number of jobs of the instance I' is unbounded due to the optional jobs. However, as their name suggests, a schedule π' for I' does not need to schedule all the *optional* jobs. More precisely, we do not request to schedule any optional job once all the *real* jobs have been processed and all the forbidden instants have been crossed. We denote by $\widetilde{C}_{\max}(\pi')$ the completion time of the last *real* job of π'. We have the following property:

Property 2.1. *There exists a schedule π for the instance I with makespan $C_{\max}(\pi)$ if and only if there exists an idle-free schedule π' for the instance I' such that $\widetilde{C}_{\max}(\pi') = C_{\max}(\pi)$.*

Proof. Given a schedule π' for I', we immediately obtain a valid schedule for the instance I by replacing the optional jobs by idle times with the same duration. The jobs of I are processed at the same dates as in π', and thus clearly the makespan is equal to $\widetilde{C}_{\max}(\pi')$. Conversely, consider a schedule π for instance I. We have to prove that for each idle period $[u, v]$ occurring in π, we can sequence optional jobs to obtain an idle-free schedule. Since π is feasible, u and v cannot be forbidden instants. Thus, if the idle period is short, that is $v - u \leq k + 1$, we can simply schedule an optional job of duration $v - u$. Otherwise, we have $v \geq u + (k + 2)$. Since there are k forbidden instants in the instance, at least one instant in the time interval $[u + 1, u + k + 1]$ is not forbidden. Let t be the last forbidden instant before $u + 1 + k$ which is not forbidden. In π', at time u, we schedule an optional job of duration $t - u \leq k + 1$. By immediate induction we can fill the remaining idle period $[t, v]$ with optional jobs. □

Based on Property 2.1, we show that we can use an ILP with a fixed number of variables and constraints to find an idle-free schedule π' minimizing the completion time of the last real job. We denote by $s' \geq k + 1$ the number of types (real and optional) in the instance I'. By construction I' is of large diversity, that is $s' > k$, and thus we know that an idle-free schedule π' exists. To bound the completion time of the last real job, we use the property (see Rapine and Brauner (2013)) that any list scheduling algorithm produces a schedule with makespan at most $Q = 2k + \sum_{i=1}^{s} m_i p_i$. Thus Q is an upper bound on the completion time of the last real job in an optimal schedule for I'. As a consequence we can assume without loss of generality that $\gamma_k \leq Q - 2$, since the last Fse instants can be discarded till this inequality holds. We also add a very large optional job of processing time $p_{s+k+2} = \gamma_k + 1$. This job allows to cross all the remaining Fse instants if the schedule finishes before the last one. Finally, for the ease of presentation, we introduce the notation $\gamma_{k+1} = Q + p_{s+k+2} + 1$.

As already discussed, we can represent an idle-free schedule by giving the number of jobs of each type sequenced between any two consecutive forbidden instants (or alternatively by giving the cumulative number of jobs completed before any forbidden instant) and the jobs crossing forbidden instants. We have the following decision variables:

m_{ij}	number of jobs of type i completed by time γ_j for $i = 1,\dots,s'$ and $j = 1,\dots,k+1$.
$S_{jf} = 1$	if a job crosses exactly the instants γ_j till γ_{f-1} (included), for $j = 1,\dots,k$ and $f = j+1,\dots,k+1$.
$= 0$	otherwise
$x_{ij} = 1$	if a job of type i crosses the instant γ_j and this job does not cross the previous Fse instant, for $i = 1,\dots,s'$ and $j = 1,\dots,k$.
$= 0$	otherwise
$y_j = 1$	if all real jobs have been completed by time γ_j, for $j = 1,\dots,k$.
$= 0$	otherwise
$\widetilde{C}_{\max}$	completion time of the last real job.

The variables m_{ij} are non-negative integers, S_{jf}, x_{ij}, y_j are boolean variables and $\widetilde{C}_{\max}$ is a non-negative real. We also define variable W_j as the total work completed by time γ_j for $j = 1,\dots,k+1$. Notice that we do not distinguish real from optional jobs in the definition of W_j, that is W_j is simply a short-hand for $\sum_{i=1}^{s'} p_i m_{ij}$. Also notice that W_j does not take into account the processing time of a job started but not yet completed, that is a job that would cross the forbidden instant γ_j. Hence a job crossing the forbidden instants γ_j but not γ_{j-1} must start at time W_j in an idle-free schedule.

The following linear formulation finds an idle-free schedule minimizing the completion time of the last real job for the instance I':
Minimize $\widetilde{C}_{\max}$, subject to the constraints:

- All Fse are crossed, which is equivalent to require that variables S_{jf} define a unique path from 1 to $(k+1)$:

$$\sum_{f=2}^{k+1} S_{1f} = 1 \tag{2.1}$$

$$\sum_{j=1}^{k} S_{j,k+1} = 1 \tag{2.2}$$

$$\sum_{j=1}^{f-1} S_{jf} = \sum_{l=f+1}^{k+1} S_{fl} \qquad \forall f = 2,\dots,k \tag{2.3}$$

- A job crosses γ_j as its first Fse instant if and only if $S_{jf} = 1$ for some index $f > j$:

$$\sum_{i=1}^{s'} x_{ij} = \sum_{f=j+1}^{k+1} S_{jf} \qquad \forall j = 1,\dots,k \tag{2.4}$$

- For each type, the variable m_{ij} is increasing with j. In addition, if a job of type i crosses γ_j, then the number of jobs of type i completed should increase by at least one after the next

forbidden instant following the completion of the job.

$$m_{i,j+1} \geq m_{ij} \qquad \forall i = 1,\dots,s' \quad \forall j = 1,\dots,k \tag{2.5}$$

$$m_{if} \geq m_{ij} + x_{ij} + S_{jf} - 1 \qquad \forall i = 1,\dots,s' \quad 1 \leq j < f \leq k+1 \tag{2.6}$$

- Schedule all the real jobs:

$$m_{i,k+1} = m_i \qquad \forall i = 1,\dots,s \tag{2.7}$$

- Set $y_j = 0$ if all the real jobs are not completed before the instant γ_j:

$$\sum_{i=1}^{s} m_{ij} \geq y_j \sum_{i=1}^{s} m_i \qquad \forall j = 1,\dots,k \tag{2.8}$$

- Definition of the work W_j:

$$W_j = \sum_{i=1}^{s'} m_{ij} p_i \qquad \forall j = 1,\dots,k+1 \tag{2.9}$$

- All the work W_j must be completed by time γ_j:

$$W_j \leq \gamma_j - 1 \qquad \forall j = 1,\dots,k+1 \tag{2.10}$$

- The amount of work completed can not increase between instants γ_j and γ_{f-1} if a job crosses these instants, that is $S_{jf} = 1$:

$$W_{f-1} \leq W_j + Q(1 - S_{jf}) \qquad \forall 1 \leq j < f \leq k+1 \tag{2.11}$$

- If $S_{jf} = 1$ and a job of type i crosses γ_j, then this job should complete in the time interval $[\gamma_{f-1} + 1, \gamma_f - 1]$:

$$W_j + \sum_{i=1}^{s'} p_i x_{ij} \geq \sum_{f=j+1}^{k+1} (\gamma_{f-1} + 1) S_{jf} \qquad \forall j = 1,\dots,k \tag{2.12}$$

$$W_j + \sum_{i=1}^{s'} p_i x_{ij} \leq \gamma_j - 1 + \sum_{f=j+1}^{k+1} (\gamma_f - \gamma_j) S_{jf} \qquad \forall j = 1,\dots,k \tag{2.13}$$

- The makespan should be equal to the first W_j such that $y_j = 1$:

$$\widetilde{C}_{\max} \geq W_1 \tag{2.14}$$

$$\widetilde{C}_{\max} \geq W_j - y_{j-1}Q \qquad \forall j = 2,\dots,k+1 \tag{2.15}$$

Constraints (2.1)-(2.2)-(2.3) are classical flow conservation equations. They impose all the forbidden instants to be crossed in an idle-free schedule. If a job crosses the forbidden instants γ_j up to γ_{f-1}, Constraint (2.4) ensures that exactly one variable x_{ij} is set to 1 to represent the type of this job; Reciprocally if one job crosses γ_j and not the preceding forbidden instant, Constraint (2.4) ensures that exactly one variable S_{jf} is set to 1, to represent the set of Fse instants crossed by the job. Constraint (2.6) forces the number of completed jobs of type i to increase by at least one between forbidden instants γ_j and γ_f if a job of type i crosses exactly all the Fse instants from γ_j to γ_{f-1}. Notice that in this case we have $x_{ij} = 1$ and $S_{jf} = 1$, which imposes that $m_{if} > m_{ij}$. As we know that an optimal schedule sequences the last real job before instant γ_{k+1}, we can impose through Constraint (2.7) that all the real jobs are completed by this time.

Constraint (2.10) ensures that the completion time of the last job completing before the instant γ_j does not coincide with this instant. Constraint (2.11) prevents from scheduling some jobs between forbidden instants crossed by a same job: if variable S_{jf} is equal to 1, then the constraint boils down to $W_{f-1} \leq W_j$. Due to Constraint (2.5), W_l is increasing with the index l. Hence we have $W_j = W_{j+1} = \cdots = W_{f-1}$: The work achieved by time γ_{f-1} is still W_j. On the contrary if S_{jf} is equal to zero, the constraint becomes redundant.

Constraints (2.12) and (2.13) prevent a job crossing the forbidden instant γ_j from completing on another forbidden instant. If $S_{jf} = 1$ for some index $f \leq k$ and the crossing job is of type i ($x_{ij} = 1$), the constraints force $\gamma_{f-1} + 1 \leq W_j + p_i \leq \gamma_f - 1$. Notice that if $f = k+1$, Constraint (2.13) becomes redundant. Finally if $S_{jf} = 0$ for all indices $f > j$, both constraints are redundant since all the variables x_{ij} are zero due to Constraint (2.4) already discussed, and the right hand sides are then equal respectively to 0 and $\gamma_j - 1$. Thus (2.12) states that W_j is non negative and (2.13) gives Constraint (2.10).

Finally, consider Constraint (2.15), and let l be the first index such that $y_l = 1$. We claim that this constraint imposes at the optimum that $\widetilde{C}_{\max} = W_l$. Indeed, if $y_{j-1} = 1$ the constraint yields $\widetilde{C}_{\max}$ positivity and if $y_{j-1} = 0$, it boils down to $\widetilde{C}_{\max} \geq W_j$. Setting $y_j = 1$ for all $j \geq l$ is feasible and dominant. Since we are minimizing $\widetilde{C}_{\max}$, the inequality $\widetilde{C}_{\max} \geq W_l$ is tight. We claim that W_j is precisely equal to the completion time of the last real jobs in an optimal solution. Indeed, once this last job has been scheduled, if there are some forbidden instants remaining, they can all be crossed by using the optional job $s + k + 2$. This optional job clearly crosses all the remaining forbidden instants, and in particular the instant γ_l. This shows that the value of W_l, and thus of $\widetilde{C}_{\max}$ at the optimum, is equal to the completion time of the last real job.

This integer program delivers an optimal solution to the instance I' and, using Property 2.1, we can convert it into an optimal solution to the original instance I. Moreover, the number of decision variables of the ILP is in $\mathcal{O}(k^2)$ and the number of constraints is in $\mathcal{O}(k^3)$. Thus, we can apply Theorem 2.4, which proves Theorem 2.3.

2.4 Conclusion

In this chapter, we have generalized to high-multiplicity the results from Rapine and Brauner (2013): we have shown that large diversity instances can be solved in polynomial time also with

a high-multiplicity encoding of the input. We proposed an algorithm solving this problem in $\mathcal{O}(sk + k^4)$ time, improving the complexity of the previous algorithm from Rapine and Brauner (2013) even if the input is not provided using a compact encoding.

We modeled $1|\text{Fse}|C_{\max}$ as an integer program and used the existence of an idle-free schedule for large diversity instances to avoid modeling the completion time for each job. The resulting integer program has a fixed number of constraints and variables. Therefore, by Eisenbrand's theorem, $1|\text{Fse}|C_{\max}$ is fixed-parameter tractable, even under high-multiplicity encoding of the input. Such an approach could be used on other high-multiplicity scheduling problems to classify them.

Further research can investigate small diversity instances. Especially, it would be interesting to determine whether this problem remains polynomial when s is close to k, in particular if $s = k$.

Other optimization criteria such as minimizing the mean flow time can be investigated as well. Chen et al. (2013) have already studied a similar problem, with one operator non-availability period. Further investigations of these problems would be interesting and likely to have industrial applications.

Chapter 3

The Identical Coupled-Task Scheduling Problem

Résumé : Nous présentons dans ce chapitre le problème d'ordonnancement de tâches couplées identiques. Une tâche couplée est une tâche composée de deux opérations séparées par une durée fixe. Dans le cas identique, il n'y a qu'un seul type de tâches et l'encodage d'une instance du problème est extrêmement compact (4 entiers). Nous étudions les propriétés de ce problème et en montrons la grande difficulté, en partie due à la présence de structures complexes dans les solutions optimales. Nous proposons des algorithmes polynomiaux ou de faible complexité permettant de calculer des solutions réalisables et utilisant certaines des structures complexes apparaissant dans les solutions optimales de ce problème.

Abstract: In this chapter, we are interested in the single-machine identical coupled-task scheduling problem[1]. A coupled-task is a two-operation job where the operations of a same job are separated by a fixed amount of time. In the identical coupled-task scheduling problem, all jobs are identical and the objective is to find a feasible schedule on one machine which minimizes the makespan. The complexity status of the coupled-task scheduling problem has been settled for several particular cases but the case where all tasks are identical remains open. Recently, Lehoux-Lebacque et al. (2009) proved that the cyclic case is polynomial. We tried to generalize their results to the finite case and found out that the optimal solutions have very different structures in the finite case. In this chapter, we present underlying structures of optimal solutions and explain how they work and why they give good solutions. We also provide lower and upper bounds for this problem. The upper bounds are obtained by building feasible solutions with special structures. The last upper bound which is presented uses very elaborated structures. Computing these solutions in polynomial time was challenging since it required a very good understanding of the structures in order to be able to ensure feasibility without actually building the schedule.

3.1 Coupled-task scheduling

In this chapter, we study the identical coupled-task scheduling problem on one machine. A coupled-task is a two-operation job defined by three durations: a_j the processing time of the first operation, b_j the processing time of the second operation and L_j the separation time. If a coupled-task j is scheduled at time t, then the task is processed during $[t;t+a_j]$ and $[t+a_j+L_j;t+a_j+L_j+b_j]$. We can schedule other operations during $[t+a_j;t+a_j+L_j]$. Figures 3.1 and 3.2 illustrate a single coupled-task and a schedule of 5 coupled-tasks. We consider the single-machine problem and the objective is to minimize the makespan. This problem is denoted by $1|coup-task|C_{\max}$.

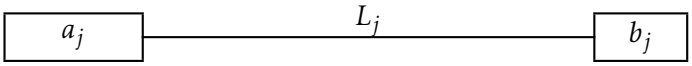


Figure 3.1: A single coupled-task

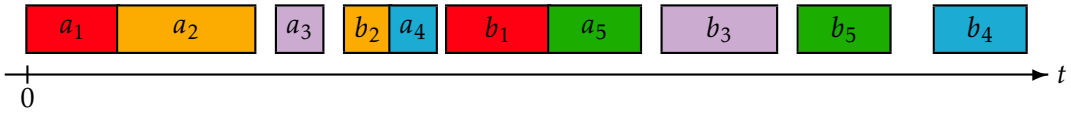


Figure 3.2: A schedule of 5 coupled-tasks on a single machine

The coupled-task scheduling problem has been introduced by Shapiro (1980) in order to model the problem of scheduling jobs on a radar system: to detect an object, the radar emits a pulse in a direction and then, after a fixed amount of time, the radar listens to the echo of the pulse. If an echo is detected, the radar system uses it to compute the position, the direction and the speed of the object using Doppler effect. This problem also has applications in medicine (Condotta and Shakhlevich 2014).

The problem $1|coup-task|C_{\max}$ has been shown to be $\mathcal{NP}$-Complete by Shapiro (1980). This analysis has been refined by Orman and Potts (1997) who settled the complexity of many sub-cases. However, the complexity in the case where all tasks are identical remains open, even if we do not consider the high-multiplicity encoding.

In this chapter, we consider this case. We denote by n the number of tasks and by integers a, b and L the characteristics of the tasks. Since all tasks are identical, we have: $\forall j = 1,\dots,n$, $a_j = a$, $b_j = b$ and $L_j = L$. In the sequel, we use the notations a and b to denote both the type of operations and their lengths. We also use L to denote both the gap and its length.

The identical coupled-task scheduling problem is denoted by $1|coup-task, a_j = a, b_j = b, L_j = L|C_{\max}$. Remark that an instance of this problem is entirely specified by giving the four integers a, b, L and n. Hence, this is a high-multiplicity scheduling problem. Without loss of generality, we assume that $a > b$ (if $a < b$ we can swap these values and if $a = b$, a greedy schedule is optimal) and $L > a$. The size of the input is $\mathcal{O}(\log L + \log n)$.

3.1.1 Related work

The coupled-task scheduling problem was originally introduced by Shapiro (1980). He proved that the problem is $\mathcal{NP}$-complete and proposed and compared 3 heuristics.

[1] The results presented in this chapter are joint work with Gerd Finke. Some of these results were presented during an invited talk in Poznań (Gabay 2011), conferences (Gabay et al. 2012a,b,c) and a technical report (Gabay et al. 2011) on some preliminary results has been written. Other results on a similar problem with unit tasks and close values of the separation times were obtained with Wojciech Wojciechowicz (from Poznań University) and are not presented here.

In the case where the time separating operations is a minimum time instead of an exact time, Gupta (1996) has shown that the multi-operation scheduling problem with time-lags is $\mathcal{NP}$-hard in the strong sense, even with only two operations per job.

Orman and Potts (1997) settled the complexity of several sub-cases of this problem. They considered the cases where all jobs have common attributes and settled the complexity of all these cases except for the identical coupled-task scheduling problem. Table 3.1 sums up their results. In this table, we give the maximal polynomial, minimal $\mathcal{NP}$-hard and open subproblems. The results are symmetric: $a_j = a$, $L_j = L$, b_j is the same as a_j, $L_j = L$, $b_j = b$.

Strongly $\mathcal{NP}$-Hard	a_j; L_j; b_j
	$a_j = L_j = b_j$
	$a_j = a$; L_j; $b_j = b$
	$a_j = a$; $L_j = L$; b_j
Open	$a_j = a$; $L_j = L$; $b_j = b$
Polynomial	$a_j = L_j = p$; b_j
	$a_j = b_j = p$; $L_j = L$

Table 3.1: Complexity of some coupled-task scheduling problems (Orman and Potts 1997)

Blazewicz et al. (2010) showed that the identical coupled-task scheduling problem with unit processing times ($a = b = 1$) and strict precedence constraints is $\mathcal{NP}$-complete in the strong sense.

Ahr et al. (2004) have elaborated an exact algorithm for the identical coupled-task scheduling problem. The complexity of this algorithm is $\mathcal{O}(nr^{2L})$ where $r \leq \sqrt[a-1]{a}$ holds. Baptiste (2010) improved this result by proving that for fixed a, b and L, the problem can be solved in $\mathcal{O}(log(n))$. The constant in the algorithm from Baptiste (2010) is exponential in L and therefore this algorithm is not polynomial in the input size. Baptiste (2010) also showed that for integer inputs, there is always an optimal solution in which all operations are starting on integer times.

In the cyclic case, we have to schedule an infinite number of tasks and the goal is to maximize the throughput rate. Brauner et al. (2009a) showed that the coupled-task scheduling problem is equivalent to a one-machine no-wait robotic cell problem and they solved the cyclic production case by adapting the algorithm from Ahr et al. (2004). They carried out some computational experiments and they were able to compute the optimal solution on random instances with L up to 43 and $\lfloor L/a \rfloor$ up to 8.

Lehoux-Lebacque et al. (2009) made a breakthrough on the identical coupled-task scheduling problem and proved that this problem is polynomial in the cyclic case. We will use some of their ideas and transpose them to the finite case.

The reader can refer to Blazewicz et al. (2012), Tanas et al. (2011) for detailed surveys on coupled-task scheduling problems. However, we remark that Blazewicz et al. (2012) report in Section 3.1 and in Table 1 that the identical coupled-task scheduling problem is polynomial by the algorithm from Ahr et al. (2004) and Baptiste (2010). This algorithm is indeed polynomial when a, b and L are fixed but not in the general case where they are part of the input.

3.1.2 Outline

In Section 3.2, we present a general fixed-parameter tractability result for the coupled-task scheduling problem. In Section 3.3, we recall the main results from Lehoux-Lebacque et al. (2009) for the cyclic case.

Based on the ideas from Lehoux-Lebacque et al. (2009), we move forward to the finite case. Surprisingly, the finite case seems to be very different from the cyclic case. In Section 3.4, we present the finite case, show the diversity of optimal solutions and highlight the lack of general structure of these solutions. We also explain the structures of different classes of optimal solutions.

In Section 3.5, we present lower bounds for this special case.

Finally, in Section 3.6 we present a combinatorial problem which occurs when we focus on finding solutions with a particular structure. We show that this problem can be solved in polynomial time and we present a polynomial algorithm which finds good solutions for the identical coupled-task scheduling problem. This algorithm finds optimal solutions in many cases and these solutions are all but trivial. They use clever structures and take advantage of the mandatory idle times in a schedule.

3.2 Fixed-parameter tractability

Based on the proof from Baptiste (2010) that there exists an optimal solution with integer starting times of the tasks, we prove that coupled-task scheduling is polynomial when the number of tasks is fixed.

Theorem 3.1. *The coupled-task scheduling problem is fixed-parameter tractable with regard to the number of tasks. This result holds even with a high-multiplicity encoding of the input.*

Proof. The proof is based on the LP used in Baptiste (2010) to prove the existence of optimal solutions in which all tasks are starting on integer times. We recall this LP: let $S_{i,j}$ be the starting time of operation j of task i, $i = 1,\dots,n$; $S_{i,1}$ is the starting time of a_i and $S_{i,2}$ is the starting time of b_i. Consider a sequence of operations and let $\nu(i,j)$ be the first operation scheduled after operation (i,j) (renumber the tasks so that b_n is the last operation). The sequence is feasible if and only if the following LP has a feasible solution. Moreover, any feasible solution of the LP is a feasible schedule (and conversely):

$$\begin{aligned}
\min \quad & S_{n,2} && && (3.1)\\
\text{s.t.} \quad & S_{i,2} = S_{i,1} + L_i + a_i && \forall i = 1,\dots,n && (3.2)\\
& S_{i,1} + a_i \leq S_{\nu(i,1)} && \forall i = 1,\dots,n && (3.3)\\
& S_{i,2} + b_i \leq S_{\nu(i,2)} && \forall i = 1,\dots,n-1 && (3.4)\\
& S_{i,j} \geq 0 && \forall i = 1,\dots,n,\ \forall j = 1,2 && (3.5)
\end{aligned}$$

The LP has $\mathcal{O}(n)$ variables and constraints. The largest number is $L_{\max} = \max_{i=1,\dots,n}(a_i, b_i, L_i)$. This LP can be solved in polynomial time by any polynomial algorithm for linear programming. Moreover, since the constraint matrix is totally unimodular, there is an optimal integral solution. We can add the integrality constraint and solve this LP in $\mathcal{O}(\log L_{\max})$ time using the algorithm from Eisenbrand (2003).

Since b_i cannot precede a_i, there are $s_n = \frac{(2n)!}{2^n}$ possible sequences. Notice that the number of sequences is smaller when tasks are identical since we can set that a_i always precedes a_{i+1}. Then, the number of sequences only depends on how a_i's are placed relatively to b_j's for $i - \lfloor \frac{L}{a} \rfloor \leq j < i$ ($[i - \lfloor \frac{L}{a} \rfloor; i[$ is the set of indices of the tasks which have a chance to be nested with task i). So the number of sequences is upper bounded by $(\frac{L}{a})^n$. In both cases, $s_n = \mathcal{O}(1)$ since n is fixed.

Therefore, by trying all sequences, we obtain an $\mathcal{O}(\log L_{\max})$ algorithm to solve the coupled-tasks scheduling problem with fixed n. □

We can also show this result by using a single integer program as we did in the previous chapter. However, the approach provided here is more direct and does not rely on the results from Lenstra (1983) or Eisenbrand (2003) except to show that the complexity is low.

3.3 Cyclic case

In this section, we present the cyclic case and the results from Lehoux-Lebacque et al. (2009) who solved this case and found a polynomial size representation of a schedule. All the results and definitions presented in this section are coming from Lehoux-Lebacque et al. (2009).

In the cyclic case, there is a single type of coupled-task with parameters a, b and L and an infinite numbers of copies of this task. A *cycle* C is a finite sequence of a's, b's and idle times that can be repeated, forming a feasible cyclic schedule. C contains necessarily the same number of a's and b's.

The cycle time $\lambda(C)$ is the ratio of the length of C over the number of coupled-tasks in C (*i.e.* the number of operations in C divided by 2). Two cycles C_1 and C_2 are *equivalent* if they have the same cycle times, $\lambda(C_1) = \lambda(C_2)$. Cycle C_1 *dominates* C_2 if their cycle times verify $\lambda(C_1) \leq \lambda(C_2)$. The dominance is *strict* if $\lambda(C_1) < \lambda(C_2)$. An *optimal* cycle is a cycle which dominates all other cycles.

Lehoux-Lebacque et al. (2009) found out that a crucial part in coupled-task scheduling is to have numerous operations during each separation time. In order to describe the structures of operations within a given task, they defined windows and profiles:

Definition 3.1 (Profile, Window)**.** Let k be a coupled-task. We denote by $W_k = a_k S_k b_k$ the *window* defined by coupled-task k (where S_k is the set of operations scheduled between a_k and b_k). We denote by $\overline{S_k}$ the sequence S_k in which each subsequence bb is converted to $\bar{a}b$. If S_k terminates with a b, then this b also becomes an $\bar{a}$ since it is followed by b_k.
We denote by β the number of (ba) in $\overline{S_k}$ and by α the number of remaining a's and $\bar{a}$'s.
The pair (α, β) is the *profile* of window W_k.

The idea of converting bb into $\bar{a}b$ comes from the fact that if 2 b's are consecutive, then there is an intrinsic idle time of at least $(a - b)$ between these two b's. Hence, the length of the first b plus the idle time is at least equal to a.

Figure 3.3 illustrates an example window, with profile $(2, 2)$. The setup of this window is: $a_0 ababbab_0 = a_0 a(ba)\bar{a}(ba)b_0$. Hence there are two ba's and two (a or $\bar{a}$)'s in this window, which corresponds to a profile $(2, 2)$.

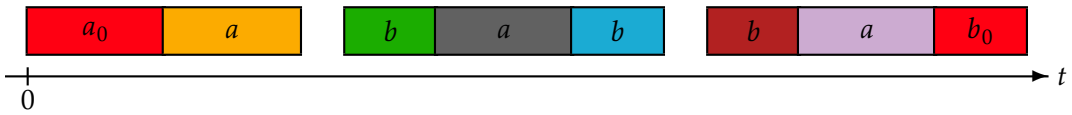


Figure 3.3: A window of profile $(2, 2)$

A window with profile (α, β) contains $\alpha + 2\beta$ operations and $\alpha a + \beta(b + a) \leq L$. Moreover, a profile is feasible if and only if α and β are non-negative integers and $\alpha a + \beta(b + a) \leq L$. The slack is denoted by $\gamma = L - \alpha a - \beta(b + a)$; this is the value of the non-intrinsic idle time in a window of profile (α, β). We denote by (α, β, γ) the *extended profile* of a window.

Based on a profile, there are several ways to define a cycle in which all windows have the same profile. For instance, one can simply build a first window $W = aa^{\alpha}(ba)^{\beta}b = a^{\alpha+1}(ba)^{\beta}b$. Then extend this sequence to $Z = a^{\alpha+1}(ba)^{\beta}b^{\alpha+1}(ab)^{\beta}$. A cycle is then obtained by repeating Z $\beta+1$ times, using the earliest placement strategy for the tasks. The cycle is $Z^{\beta+1}$ and the idle times are at different locations in each occurrence of Z. We denote this cycle by $C(\alpha,\beta)$. Notice that in order to describe the cycle $C(\alpha,\beta)$, we only need to give the two integers (α,β). These cycles have a polynomial certificate. Lehoux-Lebacque et al. (2009) have shown that all cycles having a same profile are equivalent and their cycle times can be computed in polynomial time. Moreover, they have shown that there is an optimal solution using a single profile and this optimal profile can be computed in polynomial time by Algorithm 3.1.

In the cyclic case, the aim is to minimize the cycle time. Hence, it is natural to try to have as many operations as possible in a window. Let $M^* = \max_{\alpha,\beta\in\mathbf{Z}^+}\{\alpha+2\beta:\ \alpha a+\beta(a+b)\le L\}$ the maximum number of operations in a window. A feasible profile (α,β) is called *saturated* if $(\alpha+1)a+\beta(a+b)>L$ and *tight* if $\alpha+2\beta=M^*$. A tight profile is saturated but a saturated profile is not necessarily tight. All saturated profiles are tight if and only if there is an α such that the profile $(\alpha,0)$ is feasible and tight.

Lehoux-Lebacque et al. (2009) proved that $C(\alpha^*,\beta^*)$ is an optimal cycle. The values of α^*, β^* and the cycle time $\lambda(C)$ are given by Algorithm 3.1 whose time complexity is $\mathcal{O}(log(L)^2)$. For the sake of simplicity, in the following, we will refer to $C(\alpha^*,\beta^*)$ as ***the*** optimal cycle although there is not a unique optimal cycle.

Algorithm 3.1: Optimal cyclic profile

Input: $a,b,L\in\mathbf{Z}^+ \quad (a>b,\ L\ge a+b)$
Output: An optimal profile $(\alpha^*,\beta^*,\gamma^*)$ and its cycle time $\lambda(C(\alpha^*,\beta^*))$

Let $\beta^*=\lfloor\frac{L}{a+b}\rfloor$ and $R=L-\beta^*(a+b)$
if $R<a$ **then**
  $\alpha^*=0$ and $\gamma^*=R$
else
  $\alpha^*=1$ and $\gamma^*=R-a$
end if

if $\gamma^*\ge(\beta^*+1)(a-b)$ **then**
  $\alpha^*=\lfloor\frac{L}{a}\rfloor$, $\beta^*=0$ and $\gamma^*=L-\alpha^*a$ $\quad\{\gamma^*$ satisfies $b>\gamma^*>a-b\}$
end if

$\lambda=\frac{(\beta^*+1)(2L+a+b)-\gamma^*}{(\beta^*+1)(1+\alpha^*+2\beta^*)}$

return $(\alpha^*,\beta^*,\gamma^*),\lambda$

Notice from this algorithm that the optimal profile is either of the form of $(\alpha,0)$ or $(0,\beta)$ or $(1,\beta)$.

In their proof, Lehoux-Lebacque et al. (2009) use the following dominance property:

Property 3.1. *A cycle $C(\alpha,\beta,\gamma)$ is dominated by:*

1. $C(\alpha+2,\beta-1,\gamma-(a-b))$ *if* $\beta\ge1$ *and* $\gamma\ge(\beta+1)(a-b)$

2. $C(\alpha-2, \beta+1, \gamma+(a-b))$ *if* $\alpha \geq 2$ *and* $\gamma \leq (\beta+1)(a-b)$

This dominance is strict for $\gamma > (\beta+1)(a-b)$ *in the first case and for* $\gamma < (\beta+1)(a-b)$ *in the second case.*

As a consequence, if $\gamma^* = (\beta^*+1)(a-b)$, then for any integer i, all feasible profiles (α^*-2i, β^*+i) yield an optimal cycle. In other words, all tight profiles are optimal. Notice that in such cases, Algorithm 3.1 outputs the profile $(\lfloor \frac{L}{a} \rfloor, 0, a-b)$.

In the cyclic case, there are two critical properties in order to have a polynomial certificate, an optimal algorithm and to prove its optimality. The first one is that there is an optimal solution which uses a unique profile (the profile is a window invariant) and the second one is that in a cycle, the order of the operations does not matter ([...]*aba*[...] and [...]*baa*[...] yield equivalent cycles).

3.4 Finite case

It is natural to try to generalize the results from the cyclic case to the finite case in which a number of task is given. In this chapter, we will see that the finite case is very different from the cyclic case. Extending the optimal cycle to have a start and an end is not enough to obtain an optimal solution, even when the number of tasks tends to infinity. In the following section, we present our results on the finite case. We present upper and lower bounds and we show that different solutions structures appear in the finite case.

In order to solve instances we implemented 3 approaches, based on mixed integer linear programming (MILP), dynamic programming (DP) and constraint programming (CP). The first two approaches are detailed in my master's thesis (Gabay et al. 2011). The third approach is not detailed here but the sources are available in a public repository[2]. A common weakness of these approaches is that we need to have good knowledge of the structure of an optimal schedule in order to improve their efficiency. While we have such knowledge for the cyclic case, we do not know about strong non-trivial dominance properties for the finite case. We will see afterwards that as some surprising solutions come up, we cannot really expect to have simple dominance properties.

For the sake of clarity and since all jobs are identical, we suppose that jobs are scheduled by increasing order of their indexes. In the following, we present counter-intuitive optimal solutions. These solutions are also counter-examples to the existence of simple and optimal placement strategies. We present a class of simple strategies and show that a solution within this class can be computed in polynomial time. Then, we expose improved solutions and their structures and also show lower bounds and asymptotic results. Finally, we present an original class of solutions exploiting the structures of the problem and achieving optimality in some cases.

3.4.1 Pure strategy

3.4.1.1 Definitions and properties

The first strategy which we consider is what we call the *pure strategy*. The idea is to chose a feasible profile and generate schedules using only this profile. The selected profile can be the optimal cyclic profile or any other one. Figure 3.4 illustrates a schedule using the pure strategy with profile $(0, 4)$. In this case, this yields an optimal solution.

[2] https://github.com/mgabay/Coupled-Tasks

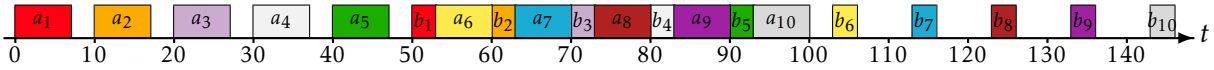


Figure 3.4: The optimal solution for $a = 7$, $b = 3$, $L = 43$, $n = 10$. This is the pure strategy solution using the optimal cyclic profile: $(0, 4, 3)$.

Notice, that there are no b's in the first window. Still, we will leave an idle time of length b in place of the operations b so that the profile fits in the next windows. We call these idle times *ghost b's*.

The pure strategy consists in selecting a profile and then repeating it until all tasks have been placed. If α or β is equal to zero in the chosen profile, then all tasks are placed as soon a possible. Otherwise, we also have to chose whether a's or ba's are placed first. As we will see later, we do not have to consider different placement strategies such as mixing up a's and ba's.

Our interest in this strategy is motivated by the fact that it is asymptotically optimal when using the optimal cyclic profile (this comes immediately from the cyclic case results). But we also have a stronger result in the finite case:

Theorem 3.1 (Asymptotic guarantee). *The pure strategy using the optimal cyclic profile is a* $(1 + \frac{2}{k})$*-approximation, where* $k = \left\lfloor \frac{n}{(\beta^*+1)(M^*+1)} \right\rfloor$ *is the number of complete cycles used.*

Before we prove this theorem, we introduce the concept of block and explain why complete cycles are interesting.

Definition 3.2 (block)**.** Let t_j^x be the starting time of operation x of task j.
We define the *block* of task i as the set of all operations scheduled within $[t_i^a; t_k^b + b]$ where $t_k^b = a + L + \max_{t_j^a < t_i^b} t_j^a$. The task k is the last task starting before b_i.

For instance, on Figure 3.4, the block of task 4 starts on time 30, ends on time 126 and is made up of operations $a_4, a_5, b_1, a_6, b_2, a_7, b_3, a_8, b_4, a_9, b_5, a_{10}, b_6, b_7, b_8$. We recall that $M^* = \alpha^* + 2\beta^*$. Observe that a block i contains 2 to $2M^* + 2$ operations and if it contains $2M^* + 2$ operations, then the windows of the task i to j are tight. So if a block contains $2M^* + 2$ operations, we call it a *tight block*.

Since the two tasks defining a block are nested the length of a block is in $\{a+L+b, \ldots, a+2L+b\}$. Let l_i be the length of the block i. We denote by $g_i = (a + 2L + b) - l_i$ the *gain* of this block.

In the cyclic case, a cycle which is generated by profile (α, β, γ) is made up of $\beta + 1$ disjoint blocks and the length of this cycle is $(\beta + 1)(a + 2L + b) - \gamma$. That is, on $\beta + 1$ block, we only gain γ compared to $(\beta + 1)$ times the maximum size of a block. Notice that if all operations are placed leftmost in the first window, then the gain in the first block is γ while it is 0 in the next β disjoint blocks. This is interesting in the finite case since we are not guaranteed that the number of tasks n is divisible by $(\beta + 1)(\alpha + 2\beta + 1)$, the number of tasks in the cycle (α, β). Hence, if we use a cycle in the finite case, we use the leftmost placement of the tasks so that the whole gain is obtained in the first block of the cycle.
Now, we can prove Theorem 3.1:

Proof of Theorem 3.1. Let us first consider an instance $I = \{a, b, L, n\}$, with $n = k(\beta^* + 1)(M^* + 1)$. We denote by $l_C = (a + 2L + b)(\beta^* + 1) - \gamma^*$ the length of the optimal cycle. Observe that a schedule of n tasks and length p is also a cycle with cycle time p/n. So $Opt(I) \geq kl_C$ (otherwise it would give a better cycle than the optimal).
Let $P(I)$ denote the length of a solution using the pure strategy with the optimal cyclic profile

(we do not mind the order of operations here). The pure strategy solution is made up of k cycles and an extension in which the tasks started in the last window are finished. Hence, $P(I) \leq (kl_C - b) + (L + b) = kl_C + L$. Therefore:

$$\frac{P(I)}{Opt(I)} \leq 1 + \frac{L}{kl_c} < 1 + \frac{1}{k}$$

Eventually, if n is different, let $k = \lfloor \frac{n}{(\beta^*+1)(M^*+1)} \rfloor$, $n^+ = (k+1)(\beta^*+1)(M^*+1)$ and $n^- = k(\beta^*+1)(M^*+1)$. We have:

$$\frac{P(\{a,b,L,n\})}{Opt(\{a,b,L,n\})} \leq \frac{P(\{a,b,L,n^+\})}{Opt(\{a,b,L,n^-\})} \leq \frac{(k+1)l_C + L}{kl_C} < 1 + \frac{2}{k}$$

□

As you can see in this proof, the analysis is gross and we can easily improve the approximation ratio. However, we will not be doing this since the matter in this problem is rather the difference between the value of a solution and the optimal one than the ratio. It is easy to get good solutions for this problem but obtaining an optimal solution is much harder.

Notice that even though the pure strategy is asymptotically optimal, we can build an infinite family of problems in which the pure strategy is never optimal. For instance, with $a = 10$, $b = 9$ and $L = 82$, the optimal cyclic profile is $(8, 0, 2)$. Now, for all positive integers k, let $n_k = k(M^*+1)+1$. The makespan of the pure strategy using the optimal cyclic profile is $k(a+2L+b-\gamma^*)+(a+L+b) = 181k+101$ while the solution using $k-1$ times the profile $(8,0)$ and then once the profile $(0,4)$ (we can denote it by $(8,0)^{k-1}(0,4)$) has a makespan of: $(k-1)(a+2L+b-\gamma^*)+(a+2L+b-\gamma^*-\beta)+(a+L+b) = 181k+97$ which is always smaller than the makespan of the pure strategy.

Now, let us come back to the placement. Contrary to the cyclic case, the order of the elements in a block is a major concern in the finite case. Figures 3.5 and 3.6 illustrate the solutions obtained with different placements of the tasks.

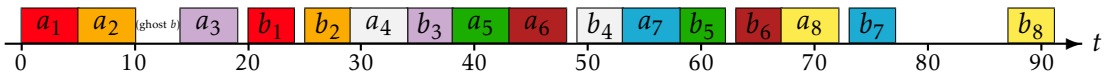


Figure 3.5: $a = 5$, $b = 4$, $L = 15$, $n = 8$, profile $(1,1)$, a first. Makespan: 91.

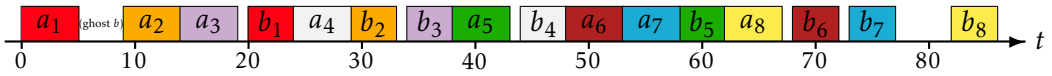


Figure 3.6: $a = 5$, $b = 4$, $L = 15$, $n = 8$, profile $(1,1)$, ba first. Makespan: 86.

Actually, for a given profile, we can compute the best makespan for a pure strategy using this profile and the best ordering of operations. In the following, we give an analytic formula of this makespan.

Let $\{a, b, L, n\}$ be an instance and (α, β) the feasible profile selected. Let $M = \alpha + 2\beta$ the number of operations in a window, $\gamma = L - \alpha a - \beta(a+b)$ the slack, $k = \lfloor \frac{n}{(\beta+1)(M+1)} \rfloor$ the number of complete cycles, $n' = n - k(\beta+1)(M+1)$ the number of remaining tasks, $k' = \lfloor \frac{n'}{M+1} \rfloor$ the number of remaining disjoint blocks, $r = n' - k'(M+1)$ the number of tasks in the extension, $r' = r - (\alpha + 1)$ and $r'' = r - (\alpha + \beta + 1)$. We recall that $\mathbf{1}$ is the indicator function. The length of the best pure strategy

solution using this profile is equal to $((\beta+1)k+k')(a+2L+b)-(k+\mathbf{1}_{k'\neq 0})\gamma+ext$, where:

$$ext=\begin{cases}\beta(a+b)+\gamma\mathbf{1}_{k'\neq 0 \text{ and } \beta\neq 0} & \text{if } r=0 \qquad (ba \textit{ first})\\ ra+L+b & \text{if } 1\leq r\leq \alpha+1 \qquad (a \textit{ first})\\ ra+L+(r'+1)b+\gamma\mathbf{1}_{k'\neq 0 \text{ and } r'\geq k'} & \text{if } \alpha+2\leq r\leq \alpha+\beta+1 \qquad (a \textit{ first})\\ a+r''(b+a)+2L+b+\gamma\mathbf{1}_{k'\neq 0 \text{ and } r''>k'} & \text{if } \alpha+\beta+2\leq r\leq M \qquad (ba \textit{ first})\end{cases}$$

Any other placement than the one indicated at the end of the lines yields a schedule with non-smaller makespan.

Remark that if the profile is not saturated, we can increase α by at least 1. Let S_1 be the original schedule and let S_2 be the schedule with α increased by 1. Let us consider the values of k, k' and r associated to the second profile. When α is decreased by 1, γ is increased by a but after $(M+1)k+k'$ blocks, there are $r+(M+1)k+k'$ tasks remaining instead of r. Hence, the length of S_1 is increased by at least $((M+1)k+k')a-ka=(Mk+k')a\geq 0$ compared to S_2. Eventually, if $k=k'=0$, then the profile actually used may not be saturated but we can consider that we used the corresponding saturated profile and that n is just too small to have the complete profile. Therefore, we only need to consider pure strategy solutions using saturated profiles.

3.4.1.2 Example

We have already seen a few examples of the pure strategy on Figures 3.4, 3.5 and 3.6. In the previous section, we provided an example proving that this strategy is not always optimal. On the other side, this strategy provides good or even optimal solutions. In Table 3.2, we consider an example where $a=5$, $b=3$, $L=100$ and $n=1$ to 48. We have solved these instances to optimality using integer and constraint programming. On all the cases in this table, the pure strategy yields an optimal solution when the right profile is used. On the first column of Table 3.2, we provide all saturated profiles for this instance. The second column gives the number of tasks in blocks using these profiles. The remaining columns are different instances of the problem (different values of n). OPT denotes that the pure strategy using the profile in the line is optimal for the value of n in the column. For instance, for $n=23$, the pure strategy using profile $(12,5)$ is optimal.

Obviously, when n is smaller than or equal to $\lfloor\frac{L}{a}\rfloor$, the pure strategy using profile $(\lfloor\frac{L}{a}\rfloor,0)$ is optimal (its length is equal to the trivial lower bound $na+L+b$).
When n is smaller than M^*+1, we have a similar result: the pure strategy using the saturated profile with largest α and smallest $\alpha+2\beta+1$ greater than n is optimal (this minimizes β and hence the size of the extension).

In the example Table 3.2, there are 13 saturated profiles and the number of tasks in their blocks goes from 21 to 25. The optimal cyclic profile is $(0,12,4)$ and there are three tight profiles. All optimal solutions found using solvers were pure strategy solutions. Notice however that for all values of n presented Table 3.2, there are less than two disjoint blocks in an optimal solution.

Actually, we believe that for all problems, when the number of tasks is smaller than or equal to $2M^*+2$, there is a saturated profile such that the pure strategy using this profile gives an optimal solution. The reason is that, with less than $2M^*+3$ tasks, there cannot be a transition. Moreover, there is no β-increasing sequence of size 2. Both of these concepts are detailed later.

Notice on the example that for $n=33,39,43,44,47$ and 48, two profiles with different number of tasks yield optimal solutions. For larger values of n, tightness is a very important property. We do not have optimal solutions on this example for n larger than 48 but we have computed all pure strategy solutions for all n smaller than 10^9. We observed that for n larger than 49, only

α	β	γ	$1+\alpha+2\beta$	$n \le 21$	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
0	12	4	25 (tight)						OPT																						
2	11	2								OPT	OPT																				
4	10	0						OPT				OPT	OPT	OPT																OPT	OPT
5	9	3	24																												
7	8	1					OPT								OPT	OPT														OPT	OPT
8	7	4	23																												
10	6	2														OPT	OPT														
12	5	0				OPT												OPT	OPT	OPT						OPT	OPT	OPT	OPT		
13	4	3	22																												
15	3	1			OPT																OPT	OPT				OPT	OPT				
16	2	4	21																												
18	1	2																				OPT	OPT								
20	0	0		OPT																				OPT	OPT						

Table 3.2: Optimal pure strategy for $a = 5$, $b = 3$, $L = 100$, $n = 1$ to 48

tight profiles yield the best pure strategy solutions. Up to $n = 1925$, the 3 tight profiles are used and 1 to the 3 of them give the best solution for a given n. For instance, for $n = 1007$ to 1014 the three tight profiles yield a best pure strategy solution.

For n greater than 1925, only one profile yields the best pure strategy solution. This profile is $(0, 12)$, the optimal cyclic profile.

3.4.1.3 Summary

The pure strategy is a natural approach towards solving the identical coupled-task scheduling problem. It takes into account the compact input and returns solutions with simple structures. The pure strategy solutions are built using a greedy algorithm but are still more sophisticated than a basic greedy algorithm which places each task as soon as possible (corresponding to the pure strategy using profile $(\lfloor \frac{L}{a} \rfloor, 0)$). Moreover, by combining the pure strategy with the cyclic case algorithm, we obtain a polynomial algorithm with an asymptotic performance guarantee of 1.

Computing the makespan of a pure strategy schedule can be done in $\mathcal{O}(\log L + \log n)$ time by using the formula provided page 44. Using this we can design both a polynomial delay algorithm and a pointwise polynomial algorithm to compute and output the best pure strategy solution for a given profile.

We remark that the number of saturated profiles for a given problem is of the order of $\mathcal{O}(L/a)$. Even enumerating all saturated profiles is not polynomial in the input size! If one intends to compute the length of the pure strategy solutions for all profiles in order to determine the best pure strategy solution, this can be done in $\mathcal{O}(\frac{L}{a}(\log L + \log n))$ which is not polynomial in the input size. This is however a much better complexity compared to any algorithm which considers all tasks separately.

3.4.2 β-increasing sequences

In the pure strategy, we have a positive gain in the first block, the $(\beta + 2)^{th}$ block and so on, there is a positive gain once every $\beta + 1$ block. This gain is obtained in the first block of the cycle and then in the following block there is an idle time appearing before the first b, then the second and so on until it is after the β^{th} b and there is a positive gain again. Hence there is no gain in the second, the third,..., the $(\beta + 1)^{th}$ but thanks to these block there is a gain in the $(\beta + 2)^{th}$ block. So these blocks are of use when there are $\beta + 2$ blocks or more. However, the blocks following the last block with a positive gain are not since they have the longest length and do not even allow to have a positive gain later. So we are interesting in making use of these blocks in order to obtain positive gain from them whenever we can. Such constructions exist, for instances in a pure strategy solution using a tight profile (α, β), if there are k additional blocks and the profile $(M^* - 2k - 2, k - 1, \gamma)$ is feasible, then we can start the schedule with k blocks of profile $(M^* - 2k - 2, k - 1)$ and then use the pure strategy with profile (α, β). In the new schedule the gain is increased by $\gamma \geq 0$. Moreover, since the k blocks where additional, it means that $k - 1 < \beta$ so in the schedule we use two profiles and β is non-decreasing between consecutive blocks. We call such a sequence, a β-increasing sequence. More precisely, we say that a sequence is β-increasing if the profile in the different blocks have non-decreasing values of β and β is increased at some point.

Figure 3.7 and Table 3.3 represent a same example of a β-increasing sequence: in the first block, the profile is $(2, 3)$, while in the other blocks, it is $(0, 4)$. Notice that this construction is

Block index	Block configuration	Profile	Gain
1	$aaa([ghost\ b]a)([ghost\ b]a)([ghost\ b]a)(idle = 3)b\ldots$	$(2,3)$	3
2	$a([ghost\ b]a)(idle = 4)(ba)(ba)(ba)b\ldots$	$(0,4)$	0
3	$a(ba)(ba)(idle = 4)(ba)(ba)b\ldots$	$(0,4)$	0
4	$a(ba)(ba)(ba)(idle = 4)(ba)b\ldots$	$(0,4)$	0
5	$a(ba)(ba)(ba)(ba)(idle = 4)b\ldots$	$(0,4)$	4

Table 3.3: A β-increasing sequence; $a = 10$, $b = 9$, $L = 80$, $n = 46$

only at the cost of delaying 1 operation: in order to increase β, we only have to omit an a in the next window and shift the next a by $(a - b)$ to the left. Table 3.3 highlights how the idle times move: between two consecutive blocks, the idle time "crosses" a b and is then located before the next b. When the idle time is after the last ba of the window, there is a gain and in the following block the idle time is located before the first ba. Let $B = \alpha^* + 2\beta^* + 1 = M^* + 1$, the number of tasks in a tight block. Notice that, in the example, $n \equiv 1 \pmod{B}$ and the construction can be reproduced for any $n = (5k + 5)B + 1$ (for any non-negative integer k): we schedule 1 block with profile $(2,3)$, then $5k+4$ blocks with profile $(0,4)$ and the last task as soon as possible. This results in a schedule with an improved makespan compared to any pure strategy solution.

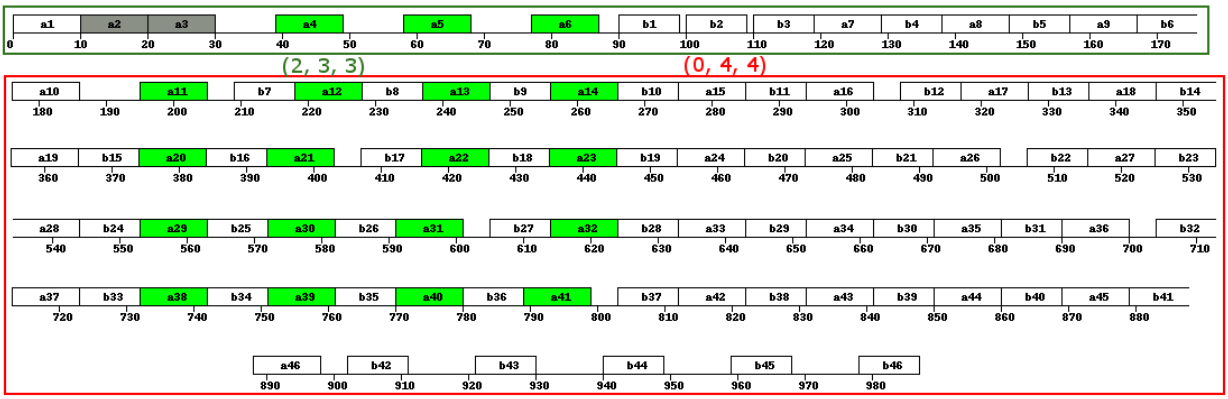


Figure 3.7: A β-increasing sequence; $a = 10$, $b = 9$, $L = 80$, $n = 46$. Optimal cyclic profile: $(0,4)$. The blocks are $a_1 \ldots b_6$, $a_{10} \ldots b_{14}$, $a_{19} \ldots b_{23}$, $a_{28} \ldots b_{32}$, $a_{37} \ldots b_{41}$ and the extension $a_{46} \ldots b_{46}$.

This type of transformation shows that if we have a profile (α, β, γ) with $\alpha \geq 2$, we can transform it to $(\alpha - 2, \beta + 1, \gamma + a - b)$. The second profile has the same number of tasks as the first one. As a consequence, if the original profile is tight, then the new one is tight as well. The new profile has an increased gain but it has an increased cycle time as well.

When the number of operations is large enough, in optimal schedules, most of the blocks are tight. So one may be interesting in finding the best β-increasing sequence using tight blocks. Observe that the optimal cyclic profile is the tight profile maximizing $\frac{\gamma}{\beta+1}$, hence it is of one of the three forms: $(\alpha^*, 0)$, $(0, \beta^*)$ or $(1, \beta^*)$. When $\beta^* > 0$, this problem can be modeled with the

following integer program:

$$\max \quad \sum_{i=0}^{c} x_i(\gamma^* - i(a-b)) \tag{3.6}$$

$$\text{s.t.} \quad \sum_{i=0}^{c} x_i(\beta^* - i + 1) \leq \lfloor \frac{n}{M^*+1} \rfloor \tag{3.7}$$

$$x_i \in \mathbf{Z}^+, \quad i = 0, \ldots, c-1 \tag{3.8}$$

where $c = \lfloor \frac{1}{a-b}(L - \alpha^* a - \beta^*(a+b)) \rfloor$; $c+1$ is the number of tight profiles, the profile $(\alpha^* + 2c, \beta^* - c)$ is the tight profile with the smallest value of β. The variables x_i denote the number of times the whole cycle $C(2i, \beta^* - i)$ is used in the schedule.

The resulting problem is a knapsack problem. The capacity of the knapsack is the maximum number of tight blocks in a schedule, the profits are the gain of the different tight profiles and the weights are the lengths of the cycles.

More precisely the problem is an unbounded knapsack problem with a specific instance (if $a-b=1$, this is a so-called inverse strongly correlated instance). The general knapsack problem is weakly $\mathcal{NP}$-Hard but the instances here are specific (weights are consecutive, profits are linear) so we cannot simply conclude on the complexity of this problem. However, we remark that the number of items in this problem is not polynomial in the input size of the coupled-tasks scheduling problem. The problem is non-trivial and if we give items separately even the problem statement is not polynomial in the input size (but we can shrink it to polynomial size by giving the input of the identical coupled-task scheduling problem). Eventually, solving this problem does not even guarantee to obtain an optimal solution.

3.4.3 Non-tight sequences

In this section, we present other structures appearing in optimal solutions for the finite case. We have already seen that we can increase β with a low cost, obtaining additional gains while we keep using tight blocks. However, there are other phenomenons occurring in the finite case and they can make it worthwhile to use non-tight blocks at some point in the schedule or even for the whole schedule. In Section 3.4.1 we have seen that for small values of n but this also occurs for larger values of n. The main reason for that is the extension: the number of tasks is not always a multiple of $\alpha^* + 2\beta^* + 1$ and so we have additional tasks to schedule; if these tasks are simply appended to a schedule, then we lose the opportunity to use the additional idle times involved. This is illustrated in the next section.

3.4.3.1 Transitions

A first way to use the additional idle time is to make a transition between two blocks. On the example considered in this section, $a = 10$, $b = 9$, $L = 82$, $n = 19$. There are 2 blocks and an additional task ($n = 2(M^* + 1) + 1$).

In Figure 3.8, we see that, when we use the pure-strategy with the optimal cyclic profile, there is a significant idle time within the last task. Using a pure-strategy with a different profile, Figure 3.9, we use some of this time to finish other tasks and obtain an improved schedule. Can we further use the idle time which is available within this last task ?

Having some empty space available at the end of the schedule is good to finish some tasks but starting an a would increase the makespan. Instead of considering that we have an additional

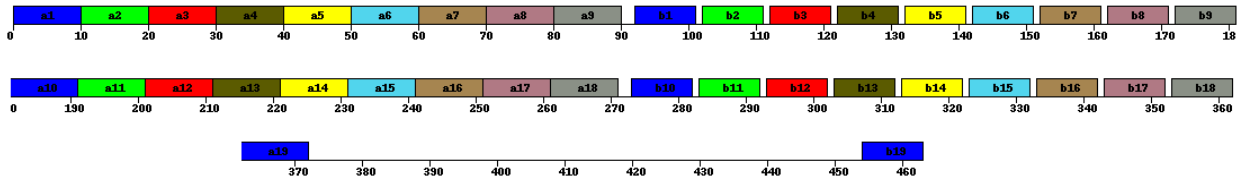


Figure 3.8: $a = 10$, $b = 9$, $L = 82$, $n = 19$: pure strategy solution with profile $(\alpha^*, 0) = (8, 0)$. Makespan: 463.

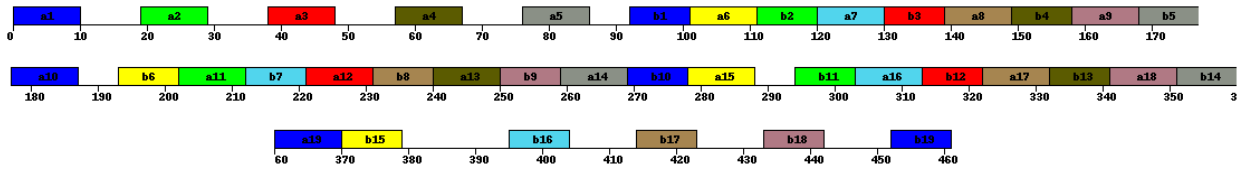


Figure 3.9: $a = 10$, $b = 9$, $L = 82$, $n = 19$: pure strategy solution with profile $(0, M^*/2) = (0, 4)$. Makespan: 461.

task which is processed last, let us consider that this task is inserted in between some blocks (in our example, between the first and second blocks). On Figure 3.8 this would only postpone the jobs of the second block, while on Figure 3.9, the schedule would remain exactly the same. Now that this task is in the middle of the schedule, we can both finish tasks within this window and start new tasks. Since we do not have to use a tight profile within this window, we can adjust the jobs to reset the gain of the previous block as if we had a complete cycle and also to change the profile to another profile with a decreased value of β. Figure 3.10 illustrates the optimal solution for our example. We say that we have a transition window between blocks 1 and 2. Using this structure, we are able to reach a full gain in both the first and the last block. In Figure 3.8 the gain is 2 in the first block and 2 in the second; in Figure 3.9 the gain is 6 in the first block and 0 in the second; in Figure 3.10 the gain is 5 in the first block and 4 in the last.

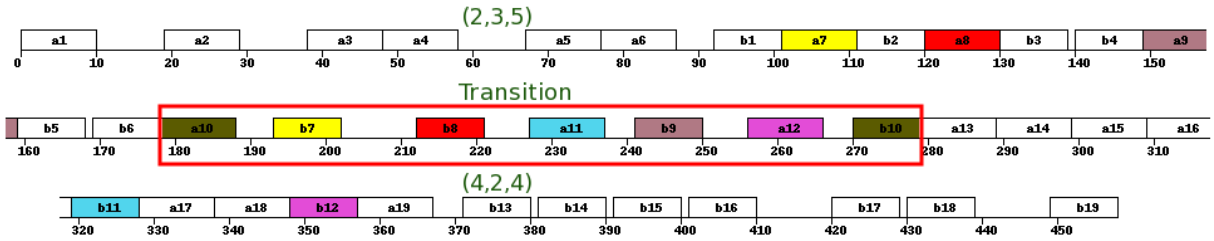


Figure 3.10: $a = 10$, $b = 9$, $L = 82$, $n = 19$: an optimal schedule. Makespan: 459.

With two blocks, it seems that additional task separates the two blocks and allows changing the profile. Seeing transitions that way is tempting because the profiles are then changing between the blocks and not within. However, the transition task is actually the first job of the second block. Using this point of view helps understanding what really happens in these solutions: the extension is fixed – after the last complete block the remaining task's a's are scheduled consecutively – and within the last a's and the last b's of the schedule, there is idle time which we can use. In transitions, we postpone some tasks and add ghosts to anticipate the end of a cycle and change profiles. Yet, the time used to postpone or "ghostify" operations does not exceed

the idle time available in the extension. As a consequence, we are able to reset cycles (replace operations as in the first block of a cycle) and also change the profile. This leads to the feeding problem which we detail in Section 3.6.3.

For larger values of n, we still use these transitions as it is shown Figures 3.11 and 3.12 in which we see them within tasks 10, 19 and 28. Notice that a schedule of 37 tasks with makespan 819 can also be obtained by appending a block of profile $(8,0)$ to the schedule Figure 3.11. For $n = 46$, the optimal makespan is 1000 and can also be obtained by appending a block of profile $(8,0)$ to the schedule Figure 3.12.

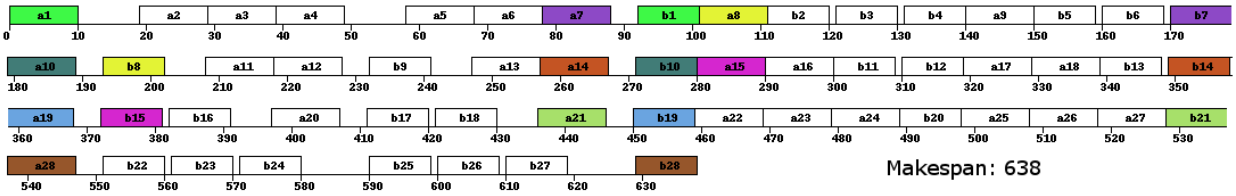


Figure 3.11: $a = 10$, $b = 9$, $L = 82$, $n = 28$: an optimal schedule. Makespan: 638.

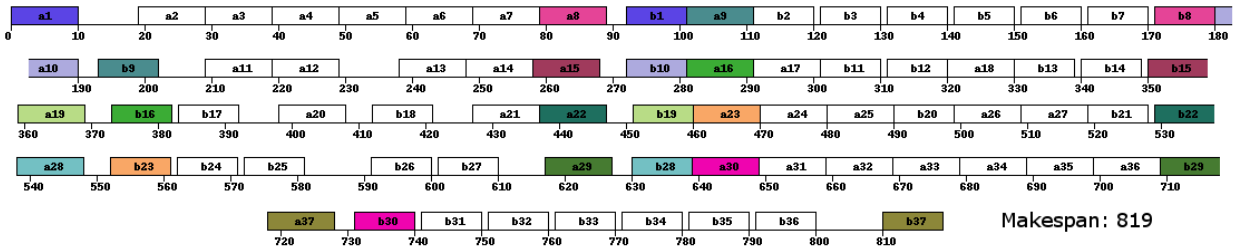


Figure 3.12: $a = 10$, $b = 9$, $L = 82$, $n = 37$: an optimal schedule. Makespan: 819.

In theses examples, the optimal cyclic profile is $(8,0)$. When $\beta^* = 0$, we are in a setup which favors the use of the optimal cyclic profile since a cycle is made up of a single and independent block. Indeed, if we have a whole cycle with positive β in the schedule, we can remove this cycle and replace it by $\beta + 1$ blocks $(\alpha^*, 0)$ with an increased gain. Yet, we observe that different profiles than $(\alpha^*, 0)$ are used in optimal solutions. The reason behind that is that we are able to take advantage of the idle time in the extension. On the three examples, $a = 10$, $b = 9$, $L = 82$ and $n \equiv 1 \pmod{(M^* + 1)}$ so we can see how solutions evolve when there is an extension and we increase the number of blocks. In the following, we will keep using this family of examples and witness interesting results.

When $\beta^* = 0$, we have the following upper bound: let $C(n)$ be the optimal solution for the problem with n tasks, $C(n + M^* + 1) \leq C(n) + (a + 2L + b - \gamma^*)$. This bound corresponds to appending a block of profile $(\alpha^*, 0)$ to the optimal solution with n tasks. It holds because a block of profile $(\alpha^*, 0)$ is independent (all a's are contained within the first window) and of length $a + 2L + b - \gamma^*$. Knowing these results one can expect that if $\beta^* = 0$ and $C(n + M^* + 1) = C(n) + (a + 2L + b - \gamma^*)$ for some n, then for all positive integers k, we have: $C(n + k(M^* + 1)) = C(n) + k(a + 2L + b - \gamma^*)$.

Very surprisingly, this property is wrong. On this example, $a = 10$, $b = 9$, $L = 82$, with 7 blocks ($n = 64$) the expected makespan would be 1362 while we can make a schedule with makespan 1361. We will see the details of these results Section 3.6.3. However, we conjecture that this property is right when the number of blocks is greater than $\frac{L}{a}$. We discuss these matters in

details Section 3.6.3.

3.4.3.2 No tight block

Because of Theorem 3.1, when the number of blocks is large enough most blocks are tight. However, and especially on the boundaries, we may use non-tight blocks. One can think that we have to use tight blocks as soon as we have more than 2 blocks. However, we invite the reader to have a look at Figure 3.13. On this example, the optimal profile is $(0,4)$ and n is not so small: we have 3 blocks and some remaining tasks. For this example, there is no optimal solution which uses a tight block (the proof is computational). The blue rectangle is not a block in the sense of Definition 3.2. However if we consider the ghosts (which overlap with b_{16} since the cycle is shortened) then it is a $(5,1)$ block. Otherwise, it is a $(6,0)$ block and it ends with b_{22}. The following block is then $(7,0)$ but this time, the block is complete. Either way, there is no tight block in this schedule.

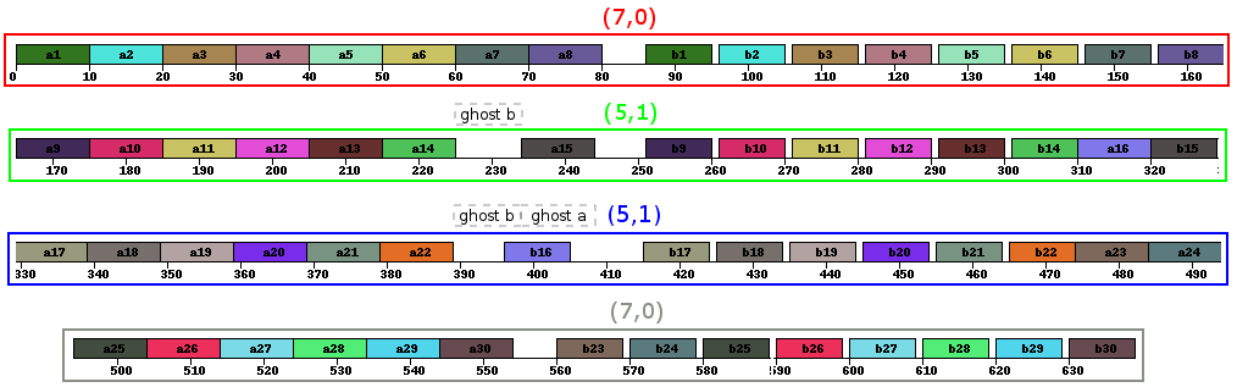


Figure 3.13: $a = 10$, $b = 9$, $L = 76$, $n = 30$: the optimal solution uses no tight block.

So, even the very powerful *tight* structures may not be used in an optimal solution. Seeing such structures with more than two blocks is quite surprising and is one of the things which makes the finite case much different from the cyclic case.

In Section 3.6.3, we elaborate a strategy which is aware of these structures and we discuss the construction of the schedule Figure 3.13.

3.4.3.3 Complexity?

The two phenomenons (transitions and no tight blocks) presented in this Section may be surprising but they are representative of the hardness of the identical coupled-task scheduling problem: there are many ways to organize tasks before and after a window and if we want to reach full gain and have the jobs fit together we have to shift them and make sure that they do not overlap. Moreover, if we want to get the best of such structures, we have to check different profiles and check if they fit together. This is a strong insight that this problem is not in $\mathcal{P}$ and maybe not even in $\mathcal{NP}$ since none of the following items is polynomial in the input size:

- Enumerating all tight profiles

- Schedule the operations of 1 window by considering tasks in this window separately
- A solution whose size is polynomial in the number of blocks (the number of blocks is roughly $\mathcal{O}(\frac{na}{L})$ which is in general non-polynomial in the size of the input)

However, we keep in mind that $\mathcal{O}(\frac{na}{L})$ is a much better complexity than $\mathcal{O}((\log n)^L)$ and propose some upper-bounding algorithms whose complexity are polynomial in $\mathcal{O}(\frac{na}{L})$.

3.5 Lower bounds

In this section, we present a lower bounds for the identical coupled-task scheduling problem. Having good lower bounds for this problem is very difficult because we need to ensure that blocks can be connected and also we have threshold effects (increasing n by 1 can increase the makespan by a up to $a+L+b$).

Property 3.2 (Lower bound)**.** *Let x be the maximum number of disjoint windows in a schedule. The length of this schedule is at least: $x\times(a+L+b)+b\times\max(0,2n-x(M^*+2))$. Moreover, $x \geq \left\lceil \frac{2n}{2+M^*+\lfloor\frac{L}{a}\rfloor} \right\rceil \geq \left\lceil \frac{n}{M^*+1} \right\rceil$.*

Proof. This lower bound corresponds to a solution in which all windows contain the maximum number of operations and the number of operations out of the windows is the smallest possible.

There are $2n$ operations to schedule and each window contains at most M^* operations. Moreover, each window is followed by at most $\lfloor\frac{L}{a}\rfloor \leq M^*$ operations b before the beginning of the next one. Hence, $x\times(1+M^*+1+\lfloor\frac{L}{a}\rfloor) \geq 2n \Rightarrow x \geq \frac{2n}{2+M^*+\lfloor\frac{L}{a}\rfloor} \geq \frac{n}{M^*+1}$ and x is integer therefore we can round up. □

Notice that there are only b's between 2 consecutive windows and 2 consecutive b's are always separated by $a-b$. Hence, their length is actually $b+(a-b)=a$ (the first b is counted with the window).

We improve the lower bound and state it as a piecewise linear function of the number of windows. The lower bound is defined for $x \in \{\left\lceil \frac{2n}{2+M^*+\lfloor\frac{L}{a}\rfloor} \right\rceil, \ldots, n\}$ and is equal to:

$$LB(x) = \begin{cases} x(a+L+b)+a(2n-x(M^*+2)) & \text{if } x \leq \frac{2n}{M^*+2} \\ x(a+L+b) & \text{otherwise} \end{cases}$$

We can now use this bound to prove the following result.

Corollary 3.1. *If $(\alpha,0,\gamma)$ is a tight profile, $\gamma \geq a-b$ and n is divisible by M^*+1 then $C^*_{max} = na + \frac{n}{M^*+1}(L+b)$. An optimal schedule is obtained by repeating blocks with profile $(\alpha,0)$.*

Proof. If $(\alpha,0,\gamma)$ is tight and $\gamma \geq a-b$, then the lower bound $LB(x)$ is increasing on all its domain. Its minimum is obtained for the smallest feasible value of x which is $x = \frac{n}{M^*+1}$. Its value is $na + \frac{n}{M^*+1}(L+b)$. Moreover, if $(\alpha,0)$ is tight, repeating blocks using this profile yields a schedule with makespan $\frac{n}{M^*+1}((\alpha+1)a+L+b) = \frac{n}{M^*+1}((M^*+1)a+L+b) = na + \frac{n}{M^*+1}(L+b)$. □

Corollary 3.1 can also be proven by noticing that if $(\alpha,0,\gamma)$ is a tight profile with $\gamma \geq a-b$, then $(\alpha,0)$ is the optimal cyclic profile. Finally, we obtain Corollary 3.1 from the following lower bound:

Property 3.3 (Cyclic Lower Bound)**.** *Let λ^* be the optimal cycle time, $n\lambda^* \leq C_{max}$. We recall that* $\lambda^* = \frac{(\beta^*+1)(2L+a+b)-\gamma^*}{(\beta^*+1)(1+\alpha^*+2\beta^*)}$.

Proof. If $C_{max} < n\lambda^*$, then using the schedule as a cycle, we obtain $\lambda = \frac{C_{max}}{n} < \lambda^*$ which contradicts the optimality of λ^*. □

The two lower bounds are gross and do not account for the connections between tasks and the threshold phenomenons. In order to have good lower bounds, we need to account for these and also for useful idle times – the ones which are used to shorten blocks or change profiles.

The following lower bounds accounts for some of these phenomenons:

Theorem 3.2. *Let $M' = \frac{L+a-b}{b}$, $z' = \left\lfloor \frac{n}{M'+1} \right\rfloor$ and $r' = n - z' \times (M'+1)$. The following lower bound holds: $C_{\max} \geq z' \times ((M'+1)b + L + a) + \mathbf{1}_{r'\neq 0} \times (r'b + L + a)$ where $\mathbf{1}_{r'\neq 0} = 1$ if $r' \neq 0$ and 0 otherwise.*

Proof. This proof relies on the fact that the identical coupled-task scheduling problem is easy when $a = b$. Indeed, the greedy solution which consists in scheduling each task as soon as possible is optimal. The reader can refer to Orman and Potts (1997) for a complete proof.

Let $(S_i)_{i=1,\ldots,n}$ be the starting times of the tasks in a feasible solution of the identical coupled-task scheduling problem with input (a, b, L, n). Consider the identical coupled-task scheduling problem with input $(b, b, L + a - b, n)$. $(S_i)_{i=1,\ldots,n}$ is also a feasible solution to this problem: this is the exact same schedule but the last $(a - b)$ time units of each operation a become idle times. Moreover, the makespan is unchanged. Therefore, $OPT(a, b, L) \geq OPT(b, b, L+a-b, n) = z' \times ((M'+1)b + L + a) + (r'b + L + a)\mathbf{1}_{r'\neq 0}$. □

3.6 Upper bounds

First of all, we remark that while there may be non-tight blocks in optimal solutions, tight blocks still have an overwhelming advantage over other blocks provided the number of disjoint blocks is not too small. For instance, we have the following property:

Property 3.4. *Consider a solution S to an instance of the identical coupled-task scheduling problem in which the block $(\alpha, 0)$ is tight. Let $B_1, \ldots, B_l$ the consecutive disjoint blocks in S: $S = B_1 B_2 \ldots B_l$. If there exists an index i such that the block B_i is tight with a profile different than $(\alpha, 0)$, let j be the smallest such index. Let B_α be the $(\alpha, 0)$ block in which a's are leftmost and let B'_j be the block B_j in which the b's in the first window have been removed. Then, the solution $S' = B_\alpha^{j-1} B'_j \ldots B_l$ is feasible and $C_{\max}(S') \leq C_{\max}(S)$.*

Proof. Consider a solution which does not verify this property. Remove the first $M^* + 1$ tasks of the schedule and append a block B_α to the schedule. Let η be the number of operations a in B_1. Let t be the end of the $(M+1)^{th}$ operation a. We have: $t \geq (\eta a + L + b) + (M + 1 - \eta)a = (M+1)a + L + b$ which is exactly the length of the block B_α. Yet, by time t, in S, $M + 1$ operations a have been scheduled and at most M operations b have been scheduled. Therefore, the makespan of the new schedule is smaller than or equal to the makespan of the previous schedule. □

The same result can be applied to the end of the schedule. So, if $(\alpha, 0)$ is tight then any optimal schedule can be modified to be of the form $B_\alpha^k a^r b^r$ for some k and r or $B_\alpha^k B S B'$ where B and B' are two tight blocks with $\beta > 0$ (it is possible that S is empty and B and B' are not disjoints).

3.6.1 Pure strategies and β increasing sequences

We already identified 2 classes of upper bounds. The first ones are the pure strategy solutions. These solutions are presented Section 3.4.1 and we show that given a profile, we can compute the makespan of the pure strategy solution using this profile in polynomial time.

The second class of upper bounds are β increasing sequences. Section 3.4.2, we show that given an instance, we can create a knapsack problem whose solution gives a schedule using different complete profiles and where the value of β between blocks are non-decreasing. These strategies allow improving on pure-strategy schedules, however the size of the knapsack problem is not polynomial in the input size and these knapsack problems are non-trivial. However, since the values of the weights and profits are regular, we can give a polynomial representation of these problems and it is very likely that they can be solved in polynomial time.

In the sequel we will see a different kind of structure and how to build good feasible schedules based on it.

3.6.2 Understanding transitions

Going back to Figure 3.8, it is clear that, in order to have a schedule with improved makespan, we have to focus on using the idle time between a_{19} and b_{19}. We can use this space by increasing β in the previous block. Obviously, since there are only two complete blocks on this example, the best β-increasing sequence is $(8,0)(0,4)Ext$ where Ext denotes the extension (a_{19} to b_{19} and what's in between). The makespan is then 459 which is better than Figure 3.9 but not better than Figure 3.10 whose makespan is 458. By increasing β we used 4 time slots instead of 0 in the extension. Yet, there are still 4 time slots available. The schedule Figure 3.10 makes a much better use of the last window. However, there is this weird window in the center which we called a *transition*.

Let us focus on what really matters. The aim is to have the shortest possible extension which means having a last window of the form a_{19}\{some other jobs\}b_{19}. In the general case, the aim is to have a last window of the form a^r\{some other jobs\}b^r with no idle time between the first a's. Obviously, we do not want to have any operation a in the {some other jobs} part because it would make the extension longer. So all operations in the {some other jobs} part are b's. Moreover, when there are two consecutive b's, they are always separated by an intrinsic idle time of at least $a-b$. Hence, they actually consume at least a time units in the schedule. This is why we talk about time slots: we possibly put at most $\left\lfloor \frac{L}{a} \right\rfloor - (r-1)$ additional operations within the extension.

The transition Figure 3.10 somehow overlaps with the extension: either we consider that we have an extension or transition(s). We focus on the extension. So, in the example Figure 3.10, we have two blocks and an extension. Understanding what is occurring in the extension is kind of tricky. A first requirement is to know which are the profiles involved. These profiles are the ones in the last window before the extension and in the first after. More precisely, for the second profile, since the extension is part of the block, it starts with the last operation of the extension and we have to compute the dual of the operations to get the profile. In the example, the second part is $b_{10}a_{13}a_{14}a_{15}a_{16}b_{11}a_{17}a_{18}b_{12}a_{19}$; its dual is $abbbbabbab = a\bar{a}\bar{a}\bar{a}ba\bar{a}bab$ which is a window of profile $(4,2)$.

Now, to understand transitions, we have to account for the ghosts as illustrated Figure 3.13. The aim of a transition is mostly to terminate a cycle earlier. In order to terminate a cycle, we have to reset the position of the idle time. After a block with positive gain g, in the next block, the first b and all following tasks are postponed by g. In the second block, all tasks after the

second b are postponed by g and so on, as illustrated Table 3.3. Therefore, in order to reset the gain, we have to make sure that it is not propagated: we have to remove all a's which are after a b and have all the other ones starting at the same position in the block as in the block with positive gain. Then if we also want to change profile, we can remove and reschedule other a's.

So the idea is that if we want to change the profile then we transform a $b_0a_1a_2$ into an aba by omitting a_1 and scheduling a_2 $a-g$ units after the end of b_0. The cost is one time slot per ba. Moreover, let $\beta_{\max}$ be the largest β of a window in the schedule. There are $\beta_{\max}$ ghost b's in the schedule so there is an initial cost of $\beta_{\max}$ slots. We discuss feasibility issues on the next section and show that provided we do not use more slots than available, then the transitions are feasible.

So basically, we compute the extension: $r = n \pmod{(M+1)}$. We want to have windows containing M operations (ghosts included) and a last block of the form $a^r\{\text{some } b\text{'s}\}b^r$. We are aiming at making the best use of these time slots in order to maximize the total gain. The number of time slots available is equal to $M+1-r$ and the cost is equal to $\beta_{\max}+\sum x_i$ where x_i is the number of times we have postponed the idle time in the first window of the i^{th} block. In case we terminate a cycle immediately after its beginning, $x_i = \beta_i$ (and this is what we will be aiming at).

Notice that we used M instead of M^*. We have $M = \alpha + 2\beta$ where (α,β) is a saturated (not necessarily tight) profile. The example Figure 3.13 is simply a case where $M \neq M^*$ (we used $M = 7$ and $M^* = 8$). In this example, there is a single tight profile which is $(0,4)$. So basically, if we use this profile, it consumes half of the available time slots at once and since it is the unique tight profile, we cannot switch to a profile with the same number of operations anyway.

3.6.3 Feeding problem

In this section, we define and solve the feeding problem which is a formalized version of the problem described in the previous section.

The feeding problem is the following: Suppose you are working for an organization whose goal is to deliver food to people. You have a working budget C and your aim is to deliver as much food as possible during a time horizon z. A truck can handle one delivery during a period. In the beginning, you do not have any truck. Buying a truck costs 1 and buying the food and the fuel for a truck's delivery costs 1. How can you maximize the number of deliveries ?

The feeding problem is a packing problem with costs: the budget is C, bins have capacity z, opening a bin costs 1 and adding an item to a bin costs 1. The aim is to maximize the number of items in bins.

Lemma 3.1. *The optimal solution to the feeding problem is equal to* $C - \left\lceil \frac{C}{z+1} \right\rceil$.

Proof. If we open $k \leq C$ bins, then the remaining money is $C-k$ and the total available capacity is kz. So the optimal solution with k bins is to assign $\min(C-k, kz)$ items to the bins and its value is $\min(C-k,kz)$. This function is maximized on $\tilde{k} = \frac{C}{z+1}$.
The optimal solution to the feeding problem is equal to $f^* = \max_{k=0,\dots,C} \min(C-k,kz)$. Let $m = \left\lfloor \frac{C}{z+1} \right\rfloor$ and $r = C - m(z+1)$.
If $r = 0$, then $f^* = z\tilde{k} = C - \tilde{k}$. Otherwise, $C - \left\lceil \frac{C}{z+1} \right\rceil = zm + r - 1 \geq zm$. Hence,

$$f^* = \max_{k=0,\dots,C} \min(C-k,kz) = \max(z\left\lfloor \frac{C}{z+1} \right\rfloor, C - \left\lceil \frac{C}{z+1} \right\rceil) = C - \left\lceil \frac{C}{z+1} \right\rceil$$

□

The optimal solution corresponds to the following greedy strategy: while the budget is not completely used, open a new bin and fill it as much as possible. So this problem can be solved in polynomial time and we can describe a solution in polynomial size: there are $k = \left\lceil \frac{C}{z+1} \right\rceil - 1$ full bins and 1 bin which is filled to $C - k(z+1) - 1$. There may be several optimum solution but, as we will see, they all yield a feasible schedule.

Now, back to the coupled-tasks scheduling problem, why is this problem relevant ? Let us consider the values of M such that $aM \leq L$. We aim at making the best schedule using blocks of $M+1$ tasks and taking advantage of the additional available slots in the last block. Let $r = n \pmod{(M+1)}$, there are $\left\lfloor \frac{L}{a} - r + 1 \right\rfloor = C$ slots available in the last block. This is the budget. The time horizon $z = \left\lfloor \frac{n}{M+1} \right\rfloor$ is equal to the number of complete blocks.

Let us have a look at a solution for the feeding problem. We use the example with $a = 10$, $b = 9$ and $L = 82$. Table 3.14, we present the optimal solution to the corresponding feeding problem when $n = 19$. The optimal schedule for this example appears Figure 3.10. Table 3.14, we denote by β_i the number of deliveries which we carry out during time period i, needless to say, it is not by chance that we use the notation β.

	Bin 1	Bin 2	Bin 3	Bin 4	β
$t = 1$	used	used	used		3
$t = 2$	used	used			2

Figure 3.14: An optimal solution to the feeding with $C = 8$, $z = 2$.

The value of the optimal solution to this feeding problem is 5 and we used the whole budget $(3 + 3 + 2)$. In the first period, we proceed to 3 deliveries and 2 in the second period. In the scheduling problem, we have 8 slots available. Figure 3.10, we start with profile $(2, 3)$ and then we move on to $(4, 2)$. In the first window, we are missing 3 operations b since there can be no b in the first window. So 3 slots are used. Then, we want to terminate the cycle, so we need to omit β_1 a's in the next window. This uses 3 additional slots. Eventually, we move on to profile $(4, 2)$ and we are missing β_2 b's. So 2 additional slots are used and the total cost is equal to $3+3+2 = 8$. If the schedule is feasible, we have a first block with gain 5, a second with gain 4 and the extension. The makespan of this schedule is equal to $2(a+2L+b)-\gamma_1-\gamma_2+(ra+L+b) = 2\times183-5-4+101 = 458$.

Solving the feeding problem yields a solution taking the best possible advantage of the extension's idle time. The fixed opening costs in the feeding problem corresponds to the ghost b's while the delivery costs corresponds to omitting the a's in the next block (either for an early termination of the cycle or because we are in the extension). This is how related are these two problems and why β in the two problems are corresponding to each others. Moreover, maximizing the gain is the same as maximizing the number of deliveries. Hence, provided the solution to the feeding problem is feasible, it yields a solutions making the best use of the time available in the extension. This solution is not guaranteed to be optimal but we conjecture that it is in many cases, especially if $\beta^* = 0$.

Theorem 3.3. *Any feasible solution to the feeding problem corresponding to an instance of the identical coupled-task scheduling problem yields a feasible solution to the identical coupled-task scheduling problem in which the gain in block i is equal to $L - Ma + \beta_i(a - b)$ (with β_i taken from an optimal solution of the feeding problem).*

Proof. Using the solution to the feeding problem, we provide a solution to the identical coupled-task scheduling problem with the same structure of gains and prove that this solution is feasible.

We sort β's by non-increasing order: $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_z$. Let $\kappa_1 = \beta_1$ and $\kappa_{i+1} = \kappa_i - 1$ for $i = 1, \ldots, \beta_1 + 1$. For $i = 1, \ldots, \beta_1$, we denote by $k_i = |\{j : \beta_j \geq \kappa_i\}| - 1$ and $k_{\beta_1+1} = M + 1 - r - 2\beta_1 - \sum_{i=1}^{\beta_1} k_i$. Let $\alpha_0 = M + 1 - r$ the number of available time slots in the last block. Remark that because of the fixed costs, $\beta_1 \leq 2\alpha_0$.

In the first window, we use the following placement of the operations (b's are ghosts and idle times are parenthesized):

$$aa^{r-1}baa^{k_1}baa^{k_2}\ldots baa^{k_{\beta_1}}a^{k_{\beta_1+1}}(\gamma_1)b$$

Then we finish the block accordingly and in the next block, if $k_1 \geq 1$, we have:

$$aa^{r-1}(\gamma_1)b(a-\gamma_1)aa^{k_1-1}(\gamma_1)b(a-\gamma_1)aa^{k_2-1}\ldots(\gamma_1)b(a-\gamma_1)aa^{k_{\beta_1}-1}a^{k_{\beta_1+1}}(\gamma_2)b$$

Otherwise, if $k_1 = 0$ and $k_2 \geq 1$, we have:

$$aa^{r-1}(\gamma_1)b(a)b(a-\gamma_1)aa^{k_2-1}\ldots(\gamma_1)b(a-\gamma_1)aa^{k_{\beta_1}-1}a^{k_{\beta_1+1}}(\gamma_2)b$$

Basically, after each block, we decrement all k_i's by 1 and if $k_i \geq 0$, then in the next block we have $[\ldots]b(a-\gamma_1)aa^{k_1}[\ldots]$ and otherwise we have $[\ldots]b(a)[\ldots]$.

In the second part of a block, we finish the block according to the profile. There is no idle time apart the intrinsic idle times between consecutive b's. So, in any position which does not hold an operation b, we place an a as soon as possible. This creates some $\bar{a}$ in the next window. For instance, if $k_1 = 0$ and $k_2 \geq 1$, the beginning of the third window will be $aa^{r-1}(\gamma_2)b(a-b)b(a-b)b(a-b)b\ldots$, which is actually $aa^{r-1}\bar{a}\bar{a}\bar{a}(\gamma_2)b\ldots$ and causes no overlapping between tasks.

Therefore, provided the first window is feasible then the schedule is feasible and with the same structure of gains as the feeding problem.

In order to show that the first window is feasible, since $aM \leq L$, we only have to ensure that $r - 1 + 2\beta_1 + \sum_{i=1}^{\beta_1}(k_i - 1) + k_{\beta_1+1} \leq M$. In the feeding problem, we have $C = M + 1 - r$ and the cost of the solution we are using to make the schedule is β_1 for the fixed opening costs and $\sum_{i=1}^{z}\beta_i = \sum_{i=1}^{\beta_1}k_i = \beta_1 + \sum_{i=1}^{\beta_1}(k_i - 1)$ for the delivery costs. The remaining money is equal to $C - 2\beta_1 - \sum_{i=1}^{\beta_1}(k_i - 1) = k_{\beta_1+1}$. Since the solution to the feeding problem is feasible, $k_i \geq 0$ for all $i = 1, \ldots, \beta_1 + 1$ and $2\beta_1 + \sum_{i=1}^{\beta_1}(k_i - 1) + k_{\beta_1+1} \leq C$ which yields:

$$M \geq r - 1 + 2\beta_1 + \sum_{i=1}^{\beta_1}(k_i - 1) + k_{\beta_1+1}$$

□

These results give a set of good, non-trivial solutions using complicated structures. Moreover, we can compute the makespan of the best such schedule in polynomial time using Algorithm 3.2.

Using this algorithm, we obtain the optimal solution for the examples Figures 3.12, 3.11, 3.10 as well as Figure 3.13 in which the optimal solution does not use tight blocks.

Section 3.4.3.1, we claimed that the "extension" property is wrong. Now that we have Theorem 3.3, we can see why this is wrong. The solutions of the feeding problems for different values of $n \equiv 1 \pmod{(M+1)}$ are given Table 3.15. Notice that for $z = 3$ to 6, the value of the optimal

Algorithm 3.2: Makespan of the "feeding" schedule

Input: $a, b, L, n \in \mathbf{Z}^+ \quad (a > b,\ L \geq a + b)$
Output: Makespan of the "feeding" schedule

Let $M = \lfloor \frac{L}{a} \rfloor$, $r = n \pmod{(M+1)}$, $l = a + 2L + b$ and $\gamma_0 = L - aM$.
Let $C = M + 1 - r$ and $z = \lfloor \frac{n}{M+1} \rfloor$
if $r = 0$ **then**
    // No mandatory extension
    // returns the makespan of a schedule with no extension ($(\alpha, 0)$ pure-strategy)
    return $z \times (l - \gamma_0)$
end if

$g = C - \lceil \frac{C}{z+1} \rceil$
return $z \times (l - \gamma_0) - g \times (a - b) + (ra + L + b)$

solution is the same. Actually, the optimal solution for $z = 3$ is unique and is also optimal for $z = 4, 5, 6$. We have an increased gain once the number of bins in an optimal solution is decreased. That is for $z = 7$. Because of Theorem 3.3, the solution of the feeding problem is feasible and is by one unit shorter than the solutions of $z = 6$ to which a block of profile $(\alpha^*, 0)$ is appended. On this example, we were able to check the solutions up to $n = 46$ ($z = 5$) and the solutions of the feeding problem were optimal.

The feeding problem solutions are very interesting, especially when $\beta^* = 0$. So far, we do not have any counter-examples where $\beta^* = 0$ and feeding problem solutions are not optimal. Especially, these solutions are very tailored for this case since a block $(\alpha, 0)$ can always be appended to a schedule and that is what is done when the whole budget is used. Without having identified these structures yet, we have shown in Gabay et al. (2011) that they are giving optimal solutions when $a - b = 1$ and $z \equiv 2$ or $z \equiv 3 \pmod{(M+1)}$ and $\beta^* = 0$.
We conjecture that the feeding problem solutions are giving optimal solutions for the identical coupled-task scheduling problem when $\beta^* = 0$.

	Bin 1	Bin 2	Bin 3	Bin 4	β
$z=3$	used	used			2
	used	used			2
	used	used			2
$z=4$	used	used			2
	used	used			2
	used				1
	used				1
$z=5$	used	used			2
	used				1
	used				1
	used				1
	used				1
$z=6$	used				1
	used				1
	used				1
	used				1
	used				1
	used				1

	Bin 1	Bin 2	Bin 3	Bin 4	β
$z=7$	used				1
	used				1
	used				1
	used				1
	used				1
	used				1
	used				1
$z\geq 7$	used				1
	used				1
	used				1
	used				1
	used				1
	used				1
	used				1
$t\geq 8$					0

Figure 3.15: Solutions of the feeding problems for $a=10$, $b=9$, $L=82$ and different $n\equiv 1 \pmod{(M+1)}$.

3.7 Conclusion

In this chapter, we presented the identical coupled-task scheduling problem and optimal solutions for the finite case. We highlighted the structures of these solutions and have shown that they use several structures. We also showed that these structures are related to combinatorial problems.

The number of different structures and their differences, are strong insights that this problem is difficult. Especially, it is unlikely to find a general simple polynomial algorithm since it will have to go through many cases, each one accounting for solutions with a special kind of structure. The feeding problem and the upper bound based on this problem is an interesting example of one of these structures.

In Gabay et al. (2011), we presented other numerical examples and analyzed special cases based on which profile is optimal in the cyclic case. In the end, we emitted the conjecture that this problem is not in $\mathcal{NP}$.

There are other classes of solution and upper bounds which are not described here. Especially, a promising one which we are currently working on is related to solving a knapsack problem corresponding to the problem of packing cycles and dealing with the extension at the same time.

With our now better understanding of this problem, we believe that if this problem is in $\mathcal{NP}$, then a certificate would consist in telling in which one of several cases we are plus one or a few profiles. It is not unlikely that such a certificate exists and that at most $\mathcal{O}(\log(M^*))$ profiles are used in an optimal solution. However, even $\mathcal{O}(M^*)$ is not polynomial in the input size and we refer the reader to Table 3.2, page 45, to see that even finding the best pure strategy in polynomial

time will be a tough job. We believe that the identical coupled-task scheduling problem is not in $\mathcal{P}$ and we would not be surprised if it were proven to belong to $EXPSPACE \backslash PSPACE$.

Part II

Packing

Chapter 4

Online Performance Guaranteed Algorithm for the Bin Stretching Problem

Résumé : Nous proposons un algorithme à performance garantie pour le problème d'ordonnancement semi-en-ligne sur m machine et dont la durée totale est connue. Ce problème se présente également comme un problème de bin packing en-ligne dont on connaît le nombre minimum de récipients de capacités unitaires nécessaires pour placer l'ensemble des objets. L'objectif dans le problème de bin stretching est alors, en utilisant ce nombre de récipients, de minimiser la taille du plus grand récipient (les capacités des récipients sont extensibles). L'algorithme proposé a une performance de $26/17 \approx 1.5294$ surpassant ainsi le meilleur algorithme connu dont la performance était de $11/7 \approx 1.5714$. Il repose sur des techniques de classification des objets et des récipients et l'application de règles de placement des objets par priorités.

Abstract: In this chapter we present an improved upper bound for the bin stretching problem[1]. We present an algorithm with performance guarantee $26/17 \approx 1.5294$ for this problem. Our algorithm improves the previous best known algorithm from Kellerer and Kotov (2013) whose stretching factor was $11/7 \approx 1.5714$. The algorithms has 2 stages and uses bunch techniques: we aggregate bins into batches sharing a common purpose.

4.1 Introduction

In bin packing problems, a set of items is to be packed into identical bins of size one; the goal is to minimize the number of bins. We are interested in the online variant of this problem: the items arrive consecutively and each of them must be packed irrevocably into a bin, without

[1] The results presented in this chapter are joint work with Vladimir Kotov. They were presented in conference (Gabay et al. 2014b). The content of this chapter is the same as the revised version of the article [Gabay et al., 2013d] which we have submitted to an international journal for publication. I also took part in research on a related problem, with Hans Kellerer and Vladimir Kotov (Kellerer et al. 2013). These latter results are not presented in the thesis.

any knowledge on future items. Recent research has focused on studying scenarios where some information is known in advance.

We consider the online problem where we know in advance that the items can be packed into m bins of size 1. The objective is to pack the items on arrival into m stretched bins, *i.e.* bins of size at most $\beta = 1 + \alpha$ where β is called the stretching factor. Formally speaking, a bin stretching algorithm is defined to have a stretching factor β if, for every sequence of items that can be assigned to m bins of unit size, the algorithm successfully packs the items into m bins of size at most β. The goal is to find an algorithm with the smallest possible stretching factor.

This problem was introduced by Azar and Regev (2001). They described a practical application of transferring files on a remote system and remarked that this problem is equivalent to the online makespan minimization problem on identical parallel machines with known value of the optimal makespan.

Graham (1966, 1969) gave the first deterministic online algorithm for this online scheduling problem. He showed that the famous List scheduling algorithm is $(2 - 1/m)$-competitive. A long list of improved algorithms has since been published, the best one is due to Fleischer and Wahl (2000).

For the semi-online case, the algorithm is provided with some information on the job sequence or has some extra ability to process it such as decreasing order (Cheng et al. 2012, Graham 1969, Seiden et al. 2000), known total processing time (Albers and Hellwig 2012, Angelelli et al. 2004, Cheng et al. 2005, Kellerer and Kotov 2013), or known number of necessary bins (Azar and Regev 2001) as in our case.

Notice that the bin stretching problem is different from the semi-online scheduling problem with known total processing time. A simple proof of this statement is that Albers and Hellwig (2012) proved that 1.585 is a lower bound for the semi-online scheduling problem with known total processing time while Kellerer and Kotov (2013) developed an algorithm with stretching factor $11/7 \approx 1.5714 < 1.585$ for the online bin stretching problem. Until recently, 4/3 was the best known lower bound for the bin stretching problem. This bound is obtained with 2 bins, on input $(1/3, 1/3, 1)$ or $(1/3, 1/3, 2/3, 2/3)$ and can be generalized to any number of bins Azar and Regev (2001). A better lower bound of $19/14 \approx 1.3571$ for 3 bins is given in Gabay et al. (2013a).

Generalizations of the bin stretching problem includes bin stretching with different machine speeds. The case with 2 uniform machines was studied in Dósa et al. (2011) and Ng et al. (2009).

In this chapter we present an algorithm that uses bunch techniques and provides a stretching factor $26/17 \approx 1.5294$.

4.1.1 Problem definition and notation

We are given a set of m identical unit size bins and a sequence of n items. Item j has a weight $w_j > 0$ and each item has to be assigned online to a bin. We define the weight of a bin B, denoted by $w(B)$, as the sum of the weights of all items assigned to B. In the course of the algorithm, we define some structures made up of one or several bins. For a given structure S, we denote by $w(S)$ the sum of the weights of all items packed into the bins composing S and $|S|$ is the number of bins in S.

The number m of bins is given as part of the initial input and it is certified that all items can fit into m bins. However, we have no more information in the initial input (the total number of items n is unknown until the end of the input).

We divide the items into 4 disjoint classes as in Table 4.1 and Figure 4.1. Items with weight in $(0; \frac{9}{34}]$ are called *tiny*, items in $(\frac{9}{34}; \frac{9}{17}]$ are called *small*, items in $(\frac{9}{17}; \frac{13}{17}]$ are called *medium* and

items in $(\frac{13}{17};1]$ are called *large*.

Item class	*tiny*	*small*	*medium*	*large*
Item weight	$(0;9/34]$	$(9/34;9/17]$	$(9/17;13/17]$	$(13/17;1]$

Table 4.1: Item classes

In the sequel, we design an algorithm with stretching factor $\frac{26}{17}$. Hence, each bin has a capacity $\frac{26}{17}$ and we say that an item j *fits* into a bin B (or equivalently that packing item j into bin B is *feasible*) if $w(B)+w_j \leq \frac{26}{17}$.

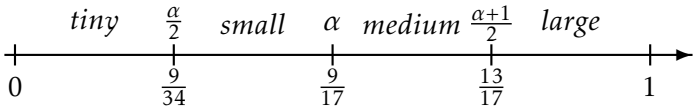


Figure 4.1: Item types for a stretching factor of $\beta = 1+\alpha = \frac{26}{17}$

4.1.2 Algorithm overview

We design a two-stage algorithm. In the first stage, we *open* the bins and create *bunches* which we use to fit the items. In the second stage, we fit the items into the remaining *non-reduced* bins and bunches.

In the algorithm, we use different types of bin structures and qualify them as *open*, *closed* or *reduced*. A structure is a group of one or several bins associated with a qualifier. We say that a bin is *open* if it can be used during current stage of the algorithm. A bin is *closed* once it contains enough items. The *closed* status simply means that the function of the bin changes. Closed bins can be reopened and converted into a new structure anytime. Finally, a bin is *reduced* if it will not be used anymore. Any reduced structure S has the property that the sum of the weights of its items is greater than its number of bins: $w(S) \geq |S|$ and for any bin $B \in S$, $w(B) \leq \frac{26}{17}$. Notice that if all bins have been reduced then there is no item remaining and the stretching factor of the current solution is at most $\frac{26}{17}$.

We denote respectively by sB, mB and lB single bins whose first goal is to contain *small*, *medium* and *large* items. $\mathcal{TB}$ and $\mathcal{LB}$ denote bunches intended to contain respectively *tiny* and *large* items. These bins and bunches can also contain different items as we will see later.

A bunch is a group of 4 bins. The aim of these structures is to help fitting items with more flexibility and then reduce them when structure's total weight is greater than or equal to 4. When a new bunch is created, we first assign a single bin to the bunch, then a second one, a third one and eventually the fourth bin. Once 4 bins have been assigned to a bunch, the bunch is complete and its status changes to *closed*. Otherwise, the bunch is incomplete and is denoted by $\mathcal{TB}^i$ where $i \leq 3$ is the number of bins currently assigned to the bunch.

In the following sections, we describe the different stages of the algorithm and show that any incoming item is packed into a *non-reduced* bin where it fits. This proves Theorem 4.1.

Theorem 4.1. *The algorithm further described in this chapter has a stretching factor of* 26/17.

This means that the algorithm never fails and all the weights of the bins are at most $\frac{26}{17}$. In the following sections, we describe the algorithm as a set of priority rules and prove its correctness.

4.2 Stage 1

At the beginning of the first stage, all bins are empty. Along the first stage, we open bins and organize them into different structures. When an item arrives, Algorithm 4.1 indicates in which structure it should be packed.

Algorithm 4.1: Packing item j

```
1  Let k = 1 and c = class(j)
2  while j is not packed AND all rules in Table 4.2 for class c have not been tried do
3      if the required structure for rule k of class c exists and is feasible then
4          Pack item j according to rule k of class c
5          Transform the structure into the new structure given Table 4.2
6      else
7          k ← k + 1
8  if j has not been packed then
9      return Fail ;                                              // Goto Stage 2
10 return Success
```

Item	Pack in	New structure
large	1. *open* $\mathcal{LB}$	*reduced* $\mathcal{LB}$ or *open* $\mathcal{LB}$
	2. *closed* $\mathcal{TB}$	*open* $\mathcal{LB}$
	3. *open* $\mathcal{TB}^1$	*reduced bin* or *open* lB
	4. *open* $\mathcal{TB}^i$	*reduced bin* and *open* $\mathcal{TB}^{i-1}$
	5. *empty*	*open* lB
medium	1. *open* mB	*reduced bin*
	2. *empty*	*open* mB
small	1. *open* sB	*reduced bin* or *open* sB
	2. *empty*	*open* sB
tiny	1. *open* lB	*reduced bin* or *open* lB
	2. *open* $\mathcal{TB}^3$	*open* $\mathcal{TB}^3$ or *closed* $\mathcal{TB}$
	3. *open* $\mathcal{TB}^i$	*open* $\mathcal{TB}^i$ or *open* $\mathcal{TB}^{i+1}$
	4. *empty*	*open* $\mathcal{TB}^1$

Table 4.2: Stage 1 priority rules

When the new structure is "*reduced X or Y*", it simply means that if $w(B) > 1$ then we reduce B and otherwise, we obtain Y. For instance, if the current item is *small* and an *open sB* exists, then the item is packed into an *open sB*. If the weight of the bin becomes greater than 1, then reduce it. Otherwise the bin remains an *open sB* and further *small* items can still be packed in it. If no *open sB* exists but there is an empty bin, then pack the current item into an empty bin which becomes an *open sB*. If there is no empty bin, then the algorithm goes into Stage 2.

If there is no item remaining, the current solution is feasible and has a stretching factor

smaller than or equal to $\frac{26}{17}$. Remark that the empty bin is in every set of rules. Hence, by the end of Stage 1, there is no empty bin remaining.

Algorithm 4.2 explains how items are packed into bunches. *Closed* bunches are made up of 4 bins added one after another. Notice that an *open* $\mathcal{TB}$ bunch contains only *tiny* items and has been assigned at most 3 bins.

Algorithm 4.2: Packing *tiny* item j into bunch $\mathcal{TB}^i$

```
// We have an open bunch TB^i composed of bins B_1,...,B_i with i ∈ {1,2,3}
// B_k is the k^th bin assigned to the bunch.
Let k = 1
while j is not packed AND k ≤ i do
    if w(B_k) + w_j ≤ 9/17 then
        Pack item j into B_k
    else
        k ← k + 1
if j has not been packed then
    // k = i + 1
    if there is no empty bin remaining then
        return Fail ;                                   // Goto Stage 2
    Assign an empty bin to the bunch as B_{i+1} and assign j to this bin
else if B_3 contains two items then
    // any two tiny items fit into B_3 with total weight smaller than 9/17
    if there is no empty bin remaining then
        return Fail ;                                   // Goto Stage 2
    Assign an empty bin to the bunch as B_4 and close the bunch
return Success
```

We apply these building rules and obtain the corresponding structures. We give the details of some rules in which there are two structures in the "New structure" field:

- Rule 4 for a *large* item: we pack the item into B_1, the first bin of the bunch. We reduce B_1 and the other bins are renamed: B_2 becomes B_1 and B_3 (if exists) becomes B_2. Notice that since rule 3 was not applied, $i \geq 2$ and $w(B_1) > 9/34$ so any large item fits into B_1 and the weight of B_1 is then greater than 1.
- Rule 2 for a tiny item: we apply the bunch building rules described in Algorithm 4.2. If the item is packed into B_1 or B_2, we obtain $\mathcal{TB}^3$. Otherwise, we obtain a *closed* $\mathcal{TB}$.
- Rule 3 for a tiny item: we apply the bunch building rules described in Algorithm 4.2. If the item is packed into B_l with $l \leq i$, we obtain $\mathcal{TB}^i$. Otherwise, we obtain $\mathcal{TB}^{i+1}$. Notice that, since rule 2 for a tiny item was not applied, we have: $i + 1 \leq 3$.

Remark that for any $\mathcal{TB}$ bunch, each bin (except B_4) contains at least two items. Denote j and

k, the two items in B_3, we have:

$$w(\mathcal{TB}) = (w(B_1) + w_j) + (w(B_2) + w_k) > \frac{18}{17}$$

Once a bunch is *closed*, sort its bins by decreasing order of the weights: $w(B_1) \geq w(B_2) \geq w(B_3) \geq w(B_4) = 0$. Then, the following property holds:

Property 4.1. *When a bunch is closed, we have:*

$$w(B_1) \geq w(B_2) \geq \frac{6}{17}$$

Proof. $w(B_1) + w(B_2) + w(B_3) > \frac{18}{17}$. Hence the largest weight of a bin is greater than the mean: $w(B_1) \geq \frac{6}{17}$. Both of the two remaining bins are containing at least two items. One precedes the other in the original ordering. W.l.o.g suppose that B_2 was before B_3. Let j and k be two items from B_3. If $w_j \geq \frac{3}{17}$ and $w_k \geq \frac{3}{17}$ then $w(B_3) \geq \frac{6}{17}$. Otherwise, $\min(w_j, w_k) < \frac{3}{17}$ and did not fit into B_2, hence $w(B_2) > \frac{9}{17} - \frac{3}{17} = \frac{6}{17}$. □

If a *closed* bunch is reopened (as an $\mathcal{LB}$) during Stage 1, items are packed into the first bin in which they fit, by increasing order of bin indices. Remark that in a *closed* $\mathcal{TB}$, the remaining capacity in each bin is larger than 1. Hence, we can fit one *large* item into each bin and then $w(\mathcal{LB}) > \frac{18}{17} + 4 \times \frac{13}{17} > 4$ and the bunch can be reduced.

Now it remains to state the reduction rules. For any structure composed of a single bin, reduce it once its weight exceeds 1. $\mathcal{LB}$ structures are reduced once they contain 4 *large* items.

Using the priority rules, one can now easily verify the following properties:

Lemma 4.1. *Anytime during Stage 1, the following properties hold:*

(*i*) *all the weights of the bins are smaller than or equal to* $\frac{26}{17}$

(*ii*) *there is at most one open mB*

(*iii*) *there is at most one open sB*

(*iv*) *there is at most one open* $\mathcal{LB}$

(*v*) *there is at most one open bunch*

(*vi*) *there is either no open lB or no bunch (neither open nor closed)*

(*vii*) *(Except rules 2 and 3 for a tiny item) packing an item into the first existing structure is always feasible and results in one of the corresponding structures stated Table 4.2.*

Note that the exception on property (*vii*) from Lemma 4.1 is related to the fact that rules 2 and 3 for a *tiny* item may require an additional empty bin. In such case, if there is no empty bin, the algorithm goes into Stage 2.

Remark that Property (*i*) from Lemma 4.1 proves Theorem 4.1 if the input ends before the algorithm goes into Stage 2.

4.3 Stage 2

In the second stage, there is no empty bin remaining (except B_4 bins in bunches). We use the remaining space in the open and closed bins and bunches to pack the items. Moreover, there is either no *open lB* or no bunch. We deal with both of these cases separately. In the following, we rely on the following property:

Property 4.2. *At any step, let $\mathcal{S}_r$ be the set of reduced bins, $|\mathcal{S}_r| = r$. The total weight of the items which are not packed into $\mathcal{S}_r$ is at most $m - r$.*

Proof. If a structure $\mathcal{S}$ is reduced then $w(\mathcal{S}) \geq |\mathcal{S}|$. We sum this up on all reduced structures and obtain: $w(\mathcal{S}_r) \geq r$. Let $\mathcal{I}$ be the set of all items and $\mathcal{I}_r$ the set of items packed into the reduced bins. $w(\mathcal{S}_r) = \sum_{i \in \mathcal{I}_r} w_i = w(\mathcal{I}_r)$. Since all items can be packed into m bins with capacity 1, $w(\mathcal{I}) \leq m$. Hence $w(\mathcal{I}) - w(\mathcal{I}_r) \leq m - r$. □

4.3.1 All bunches have been reduced

If there is no non-reduced bunch remaining, then there are no *open* $\mathcal{TB}^i$ or *closed* $\mathcal{TB}$ or *open* $\mathcal{LB}$ remaining. At the end of Stage 1, we have some of the following structures:

Reduced bins	*Reduced* $\mathcal{LB}$	
Open lB	*Open mB* (0 or 1)	*Open sB* (0 or 1)

Algorithm 4.3: Packing item j in Stage 2 (no non-reduced bunch remaining)

```
if item j fits in an open lB then
    Pack item j into the largest bin open lB in which it fits
else
    Pack item j into the largest bin in which it fits
Let B be the bin in which j has been packed.
if w(B) ≥ 1 then
    Reduce B
```

Algorithm 4.3 indicates how an item is packed during Stage 2. Remark that any *small* or *tiny* item can be packed into any non-reduced bin. Hence, while some *lB* are remaining, *open mB* or *open sB* are only used to pack *medium* or *large* items.

Lemma 4.2. *If there is no open or closed bunch at the beginning of Stage 2, then Algorithm 4.3 does not fail and the weight of all bins is smaller than or equal to $\frac{26}{17}$.*

Proof. Suppose that a remaining item j cannot be packed into the remaining open bins. For any non reduced bin B_i, the following inequalities hold:

$$w(B_i) > \frac{9}{17} \tag{4.1}$$

$$w_j + w(B_i) > \frac{26}{17} \tag{4.2}$$

Inequality (4.2), together with the fact that the weight of a non reduced bin is smaller than 1, give $w_j > \frac{9}{17}$. Therefore j is *medium* or *large*.

If there is 0 or 1 open bin remaining, then (4.2) contradicts Property 4.2. Hence, there are at least two open bins remaining.

Suppose there is no *open* lB remaining. Then, there are exactly two bins remaining: B_1, an *open* mB and B_2, an *open* sB. We sum up inequalities (4.1) and (4.2) and get: $w(B_1)+w(B_2)+w_j > 35/17 > 2$ which contradicts Property 4.2. Therefore, there are some *open* lB's remaining.

Remark that during Stage 1, *tiny* items can only be packed within lB bins or $\mathcal{TB}$ bunches. Since there were no bunches remaining at the beginning of Stage 2, all bunches have been reduced to *reduced* $\mathcal{LB}$. Moreover, there are some *open* lB's remaining. Hence, during Stage 2, all *tiny* items were packed into lB bins. Therefore, *tiny* items have been packed only into bins containing *large* items.

In any feasible solution to the bin packing problem, any bin containing a *large* item can only hold a few additional *tiny* items. Let p be the total number of *large* items and l the number of *large* items already packed.

We denote by $B^1,\dots,B^l$, the bins containing *large* items in the current solution. Because of the preceding remark, we know that $B^{l+1},\dots,B^m$ contain no *tiny* item. Hence we can pack j and all items from $B^{l+1},\dots,B^m$ into $m-l$ bins of capacity 1. Therefore, we have:

$$m-l \geq w_j + \sum_{i=l+1}^{m} w(B_i) \tag{4.3}$$

Additionally, all bins which are not containing *large* items have been reduced (and hence their weights are greater than 1), except maybe an *open* mB and an *open* sB, hence:

$$\sum_{i=l+3}^{m} w(B_i) \geq m-l-2 \tag{4.4}$$

and, by inequalities (4.1) and (4.2), we have:

$$w_j + w(B_{l+1}) + w(B_{l+2}) \geq \frac{9}{17} + \frac{26}{17} > 2 \tag{4.5}$$

By summing up inequalities (4.4) and (4.5), we obtain:

$$w_j + \sum_{i=l+1}^{m} w(B_i) > m-l \tag{4.6}$$

This contradicts inequality (4.3). Therefore, there is no such item j. □

We have proved in this case that the algorithm never fails and always returns a solution using at most m bins, filled to at most $\frac{26}{17}$.

Remark that if we define the classes as in Figure 4.1: $(0;\frac{\alpha}{2}]$ (*tiny*), $(\frac{\alpha}{2};\alpha]$ (*small*), $(\alpha;\frac{1+\alpha}{2}]$ (*medium*) and $(\frac{1+\alpha}{2};1]$ (*large*), then all previous results hold for any $\alpha > 0.5$.

4.3.2 There are some non-reduced bunches

We now show that Lemma 4.2 still holds if there are some non-reduced bunches remaining at the end of Stage 1. In this case, there is no *open* lB remaining. Stage 2 starts with some of the following structures:

Reduced bins	*Reduced* $\mathcal{LB}$	
Open mB (0 or 1)	*Open sB* (0 or 1)	
Open $\mathcal{TB}^i$ (0 or 1)	*Open* $\mathcal{LB}$ (0 or 1)	*Closed* $\mathcal{TB}$

During Stage 2, closed bunches are reopened and used to pack some of the remaining items. In the meantime, some *buffer* bins are used to pack the other items. These buffers will receive the smaller items while the larger ones will be packed in the bunches.

Current *buffer* is called $\mathcal{X}$. Along with this buffer, we use up to 3 other single bins: $\mathcal{Z}_1$, $\mathcal{Z}_2$ and $\mathcal{Z}_3$. If there is an *Open* $\mathcal{TB}^i$ at the beginning of Stage 2 we assign its bins to $\mathcal{Z}_1$ and possibly $\mathcal{Z}_2$ and $\mathcal{Z}_3$, by decreasing order of their weights. Whenever we have no $\mathcal{X}$ (Stage 2 is beginning or $\mathcal{X}$ is reduced), the first existing structure among the following becomes $\mathcal{X}$:

$$\textit{open sB},\ \textit{open mB},\ \mathcal{Z}_3,\ \mathcal{Z}_2,\ \mathcal{Z}_1,\ \textit{closed } \mathcal{TB}$$

In all but the last case, we get $\mathcal{X}$ by renaming a bin. In the last case, we denote by B_1, B_2, B_3, B_4 the bins from the bunch, $w(B_1) \geq w(B_2) \geq w(B_3) \geq w(B_4)$. We assign: $\mathcal{X} \leftarrow B_4$, $\mathcal{Z}_1 \leftarrow B_1$, $\mathcal{Z}_2 \leftarrow B_2$ and $\mathcal{Z}_3 \leftarrow B_3$ and the bunch is disbanded.

If we cannot get a new $\mathcal{X}$, then only a few bins are remaining. Stage 2 is terminated and the algorithm goes into a last stage, detailed in Section 4.3.2.2.

During Stage 2, an additional type of bunch, denoted by $\mathcal{MB}$ is used. The main purpose of these bunches is to receive *medium* items.

The process is then very similar to Stage 1: items are packed into bins according to priority rules and bins are reduced. Priority rules are given Table 4.3. There is however a slight difference with Table 4.2: it should be read as "Pack item j into structure $\mathcal{S}$ if $\mathcal{S}$ exists and packing item j into $\mathcal{S}$ is feasible and results in the new structure indicated Table 4.3". This difference only concerns rule (1) for *large* items and the reason is that $\mathcal{Z}_1$ was part of a (possibly open) bunch. Therefore, at the end of Stage 1, its weight was smaller than 9/17 and any item can be packed into $\mathcal{Z}_1$. However, we only pack an item into $\mathcal{Z}_1$ if we can reduce it afterwards.
If $\mathcal{Z}_1$ is reduced, then $\mathcal{Z}_1 \leftarrow \mathcal{Z}_2$ and $\mathcal{Z}_2 \leftarrow \mathcal{Z}_3$ (if exists).

Item	Pack in	New structure
large	1. $\mathcal{Z}_1$	*reduced bin*
	2. *open* $\mathcal{LB}$	*reduced* $\mathcal{LB}$ or *open* $\mathcal{LB}$
	3. *closed* $\mathcal{TB}$	*open* $\mathcal{LB}$
	4. $\mathcal{X}$	*reduced bin* or $\mathcal{X}$
medium	1. *open mB*	*reduced bin*
	2. $\mathcal{X}$	*reduced bin* or $\mathcal{X}$
	3. *open* $\mathcal{MB}$	*reduced* $\mathcal{MB}$ or *open* $\mathcal{MB}$
	4. *closed* $\mathcal{TB}$	*open* $\mathcal{MB}$
small / *tiny*	1. $\mathcal{X}$	*reduced bin* or $\mathcal{X}$
	2. *open* $\mathcal{MB}$	*reduced* $\mathcal{MB}$ or *open* $\mathcal{MB}$

Table 4.3: Stage 2 priority rules

When an item is assigned to a single bin structure, if the weight of the bin becomes greater than 1, then the bin is reduced.

When an item is assigned to an *open* $\mathcal{LB}$, we try to pack it into B_3, then B_2, B_1 and eventually B_4. Once B_4 contains an item, we reduce the bunch. As seen in Stage 1, the weight of the structure is greater than 4.
When a *medium* item is assigned to a *closed* $\mathcal{TB}$, it is packed into B_3. When an item is assigned to an *open* $\mathcal{MB}$ we try to pack it into B_3, then B_2 and eventually B_4. Since B_4 was empty at the end of Stage 1, we can pack any two *medium* items into B_4. When B_4 contains 2 items, we reduce B_2, B_3, B_4 and $\mathcal{X}$ and $\mathcal{X} \leftarrow B_1$. The following property shows that these bins can indeed be reduced:

Property 4.3. *Once B_4 from an open $\mathcal{MB}$ contains two items, $w(\mathcal{X}) + w(B_2) + w(B_3) + w(B_4) > 4$.*

Proof. During Stage 2, at least one item j which did not fit into B_3 has been packed into B_2. Hence, by Property 4.1:

$$\begin{aligned} w(B_3) + w(B_2) &= (w(B_3) + w_j) + (w(B_2) - w_j) \\ &> 26/17 + 6/17 = 32/17 \end{aligned}$$

Therefore, $\max(w(B_3), w(B_2)) > 16/17$. Moreover, B_4 is containing two items k and l (with l the last item packed). Neither k, nor l fit into B_3 or B_2 and l does not fit into $\mathcal{X}$. Hence:

$$\begin{aligned} &w(\mathcal{X}) + \min(w(B_3), w(B_2)) + w(B_4) \\ &\quad \geq (w(\mathcal{X}) + w_l) + (\min(w(B_3), w(B_2)) + w_k) \\ &\quad > 26/17 + 26/17 = 52/17 \end{aligned}$$

Eventually, summing this up with $\max(w(B_3), w(B_2))$ gives:

$$w(\mathcal{X}) + w(B_2) + w(B_3) + w(B_4) > 4$$

□

Remark that there is no assumption on the classes of the items packed into $\mathcal{X}$, B_2 and B_3 in Property 4.3.

4.3.2.1 Termination stage

Stage 2 is completed, either when the input is over or no packing rule is feasible (or we cannot get a new $\mathcal{X}$ – in such case, refer to section 4.3.2.2). In the following, we consider the different cases and show that we can always fit remaining items into non-reduced bins with a 26/17 stretching factor.

If the algorithm finishes before an item cannot be packed according to priority rules, then all items have been packed and none of the bins capacities exceeds 26/17. If all bins have been reduced, then the sum of all the weights of the bins is greater than m and hence all items have been packed.
Otherwise, no rule can be applied to pack the current item. Table 4.4 sums up the possibly remaining structures depending on the current item. Remark that for a *large* item, if Z_2 exists, then $w(Z_1) > \frac{9}{34} > \frac{4}{17}$ and since $w(Z_1) \leq \frac{9}{17}$, we can apply rule 1 for a *large* item. Hence if current item is *large* and no rule can be applied, then there is no $\mathcal{Z}_2$ remaining. The reader can easily verify remaining configurations for the other classes of items.

Table 4.4 does not take *open mB* into account. We deal with this case as follows: if an *open mB* is remaining, then no *medium* item came during Stage 2. Hence, there is no $\mathcal{MB}$ bunch.

Current item	Remaining bins
large	*open* $\mathcal{MB}$, $\mathcal{X}$, $\mathcal{Z}_1$
medium	*open* $\mathcal{LB}$, $\mathcal{X}$, $\mathcal{Z}_1$, $\mathcal{Z}_2$, $\mathcal{Z}_3$
small	*open* $\mathcal{LB}$
tiny	*open* $\mathcal{LB}$

Table 4.4: Remaining structures depending on the current item

Moreover, the current item j is *large* since any *tiny* or *small* item would fit into $\mathcal{X}$ and any *medium* into *open mB*. Therefore, there is no $\mathcal{Z}_2$. The remaining bins are $\mathcal{X}$, *open mB* and possibly $\mathcal{Z}_1$. The remaining items are packed according to Subsection 4.3.2.4.

If a bunch is remaining, we denote its bins by B_1, B_2, B_3 and B_4. Depending on the current configuration, we reduce some of the remaining bins as detailed in Algorithm 4.4.

Algorithm 4.4: Termination Stage

```
1  if there is an open LB containing 3 large items and no Z_3 then                    // case 1
2      Reduce B_1, B_2 and B_3
3  else if there is an open LB containing 3 large items and Z_3 then                  // case 2
4      if current item j fits into X then
5          Pack j into X
6          if w(X) ≥ 1 then
7              Reduce X, B_1, B_2 and B_3
8      else
9          Pack j into Z_1
10         Reduce X, Z_1, B_1, B_2 and B_3
11 else if there is an open MB remaining then                                         // case 3
12     if current item j fits into B_2 then
13         Pack j into B_2
14         Resume Stage 2
15     else
16         Pack j into B_1
17         Reduce B_1, B_2 and B_3
18 else if there is an open LB containing 1 or 2 large items remaining then          // case 4
19     Consider this bunch as an open MB and resume Stage 2
20 else
       // case 5
       // There is no bunch and at most 4 remaining bins
21     Use the rules given Section 4.3.2.3 to 4.3.2.5 to terminate
```

In the following, we explain why Algorithm 4.4 works and show that the remaining bins are in one of the configurations treated in the next subsections.

1. If there is an *open* $\mathcal{LB}$ containing 3 *large* items and no $\mathcal{Z}_3$. Reducing B_1, B_2 and B_3 is feasible because the bunch contains 3 *large* items and was a *closed* $\mathcal{TB}$ bunch before being reopened; hence its weight was greater than 18/17. Therefore:

$$w(B_1) + w(B_2) + w(B_3) \geq 18/17 + 3 \times 13/17 = 57/17 > 3$$

   Then, we have at most 4 bins remaining: $B_4, \mathcal{X}, \mathcal{Z}_1$ and $\mathcal{Z}_2$.

2. If there is an *open* $\mathcal{LB}$ containing 3 *large* items and $\mathcal{Z}_3$, we pack all coming items into $\mathcal{X}$ until it is reduced and then we reduce B_1, B_2 and B_3 as previously. Otherwise, current item j does not fit into $\mathcal{X}$. Since $\mathcal{Z}_3$ exists, we can use Property 4.1 for $\mathcal{Z}_1$:

$$w_j + w(\mathcal{X}) + w(\mathcal{Z}_1) + (w(B_1) + w(B_2) + w(B_3)) > 26/17 + 6/17 + 57/17 > 5$$

   We pack j into $\mathcal{Z}_1$ and reduce $\mathcal{X}$, $\mathcal{Z}_1$, B_1, B_2 and B_3. Then, $\mathcal{Z}_1 \leftarrow \mathcal{Z}_2$, $\mathcal{Z}_2 \leftarrow \mathcal{Z}_3$ and we have exactly 3 bins remaining: $\mathcal{Z}_1$, $\mathcal{Z}_2$ and B_4.

3. If there is an *open* $\mathcal{MB}$ remaining, then j (the current item) is *large*. If j fits into B_2 we pack it into B_2 and resume with priority rules. Property 4.3 still holds. Otherwise, B_2 contains an item which does not fit into B_3. Hence, $w(B_2) + w(B_3) > \frac{26}{17} + \frac{6}{17} = \frac{32}{17}$. Once j is packed into B_1, $w(B_1) + w(B_2) + w(B_3) > \frac{6}{17} + \frac{13}{17} + \frac{32}{17} = 3$ and we can reduce B_1, B_2 and B_3. There are at most 3 remaining bins: B_4, $\mathcal{X}$ and $\mathcal{Z}_1$.

4. If there is an *open* $\mathcal{LB}$ containing 1 or 2 *large* items remaining, we consider this bunch as an *open* $\mathcal{MB}$ and keep on applying priority rules and eventually previous point (3).

5. Otherwise, there is no bunch. There are at most 4 bins remaining: $\mathcal{X}, \mathcal{Z}_1, \mathcal{Z}_2$ and $\mathcal{Z}_3$.

After these reductions, we have at most 4 bins remaining. Let b be the number of remaining bins. In each cases, we explain how to use the remaining bins and then consider j, an item which does not fit into any of the remaining bins. We show that w_j plus the sum of the weights of the remaining bins is strictly greater than b, contradicting Property 4.2.

The cases with 0 or 1 bin remaining are trivial so we only deal with the other cases.

4.3.2.2 We cannot get a new $\mathcal{X}$

If we cannot get a new $\mathcal{X}$, then remaining bins are possibly an *open* $\mathcal{MB}$ and an *open* $\mathcal{LB}$. We keep on applying priority rules. However, when an item is packed into *open* $\mathcal{MB}$, we try to pack it into B_3, then B_2, then B_1 and eventually B_4. Hence, if the *open* $\mathcal{MB}$ is reduced, its 4 bins are reduced.

Once there is a single structure remaining, if it is the *open* $\mathcal{LB}$, then we reduce the bins and finish as presented Subsection 4.3.2.1.

Otherwise there is an *open* $\mathcal{MB}$ remaining. We keep on applying priority rules and suppose some item j cannot be packed.

The item j cannot be packed. Hence B_1 and B_4 both contain an item which fits into neither B_2, nor B_3. Denote those items k and l. By Property 4.1: $w(B_1) - w_k \geq \frac{6}{17}$. Moreover, Property 4.3

holds. Therefore, B_4 contains a single item. Therefore, either l or j (or both) is *large*. Without loss of generality, suppose j is *large*, then:

$$\begin{aligned} & w(B_1) + w(B_2) + w(B_3) + w(B_4) + w_j \\ & \quad \geq (w(B_1) - w_k) + (w(B_2) + w_k) + (w(B_3) + w_l) + w_j \\ & \quad > 6/17 + 26/17 + 26/17 + 13/17 \\ & \quad > 4 \end{aligned}$$

Which is a contradiction.

4.3.2.3 4 bins remaining

If there are 4 remaining bins, the possibly remaining bins are detailed Table 4.5. We rename those bins L_1, L_2, L_3 and L_4. Remark that $w(L_2), w(L_3), w(L_4) \leq \frac{9}{17}$ at the beginning of this step. Hence we can fit at least one item in any of those three bins.

New names	L_1	L_2	L_3	L_4
Old names	$\mathcal{X}$	B_4	$\mathcal{Z}_2$	$\mathcal{Z}_1$
	$\mathcal{X}$	$\mathcal{Z}_3$	$\mathcal{Z}_2$	$\mathcal{Z}_1$

Table 4.5: Renaming scheme

Pack any fitting item into L_1, otherwise L_2, then L_3 and eventually into L_4. Suppose j is an item which does not fit into any of the remaining bins. Denote k_i the last item packed into L_i and remark that, for $i = 2, 3, 4$, k_i does not fit into L_f for all $f < i$.

If the weight of a bin is greater than 1, then:

$$\begin{aligned} & w(L_1) + w(L_2) + w(L_3) + w(L_4) + w_j \\ & \quad > 1 + 26/17 + 26/17 \\ & \quad > 4 \end{aligned}$$

Otherwise, all the weights of the bins are smaller than one. Hence $w_j > \frac{9}{17}$. Moreover, at the beginning of this step, $w(L_3) + w(L_4) > \frac{9}{17}$.

$$\begin{aligned} & w(L_1) + w(L_2) + w(L_3) + w(L_4) + w_j \\ & \quad \geq (w(L_1) + w_{k_3}) + (w(L_2) + w_{k_4}) + \\ & \qquad (w(L_3) + w(L_4) - w_{k_3} - w_{k_4}) + w_j \\ & \quad > 26/17 + 26/17 + 9/17 + 9/17 \\ & \quad > 4 \end{aligned}$$

Which is a contradiction.

4.3.2.4 3 bins remaining

Remark that if there are 3 bins remaining, $\mathcal{Z}_1$ is among them and $w(\mathcal{Z}_1) \leq \frac{9}{17}$. Rename it L_3 and the other bins are renamed L_1 and L_2. Pack any fitting item into L_1, otherwise L_2 and eventually

L_3. Suppose that the item j does not fit into any of them and let k be the last item packed into L_3. There is at least one such item since $w(\mathcal{Z}_1) \leq \frac{9}{17}$ in the beginning.

$$\begin{aligned} & w(L_1) + w(L_2) + w(L_3) + w_j \\ & \quad \geq (w(L_1) + w_j) + (w(L_2) + w_k) \\ & \quad > 26/17 + 26/17 \\ & \quad > 3 \end{aligned}$$

Which is a contradiction.

4.3.2.5 2 bins remaining

In this case, denote one bin by L_1 and the other bin by L_2. Pack any fitting item into L_1, otherwise into L_2. If j does not fit into L_2, then $w(L_2) > \frac{9}{17}$.

$$w_j + w(L_1) + w(L_2) > 26/17 + 9/17 > 2$$

Which is a contradiction.

4.4 Complexity

We represent a bin and its content using a stack plus its current weight and use a dedicated data structure (a stack) for each kind of structure used in the algorithm. The overall space used is $\mathcal{O}(m)$.

In order to pack any given item during Stage 1, we need to check its class and try to pack it in at most 5 different structures with at most 3 bins tested for each one. Hence, any item is packed in $\mathcal{O}(1)$ time. Therefore the overall complexity of the first stage is bounded by $\mathcal{O}(n)$.

During Stage 2, we need to sort the structures. Each structure has at most 4 bins. Hence, a structure is sorted in $\mathcal{O}(1)$ time and we have at most $\frac{m}{4}$ structures to sort. Therefore, we sort all of them in $\mathcal{O}(m)$ time. In order to pack any item, we need to check its class and try at most 4 different structures. Hence, any item is packed in $\mathcal{O}(1)$ time and the overall complexity of this stage is bounded by $\mathcal{O}(n)$.

Same goes for the termination stage. Moreover, additional operation, like renumbering the bins, are performed but there is a fixed number of different additional operations and all of them are performed in constant time.

Eventually, when $m \geq n$, at most n bins are used. Hence, the overall time and space complexity of the algorithm is $\mathcal{O}(n)$.

4.5 Summary and future work

The presented algorithm has a stretching factor of $\frac{26}{17}$ and runs in linear time. Notice that this bound is tight with the input $m = 2$ and the items: $\{\frac{13}{17}, \frac{13}{17}\}$.

The techniques of combining bins into bunches with certain properties and analyzing the bunches has been successfully applied to other online and offline packing problems, see e.g. Babel et al. (2004), Kellerer and Kotov (2003).

It seems reasonable to hope that better worst-case behavior can be achieved by refining this approach. Based on this scheme, it might be possible to reduce the gap between lower and upper

bound for both known total sum and bin stretching problems. Improving lower bounds is also a challenging task.

Chapter 5

Lower Bounds for Online Problems

Application to Bin Stretching

Résumé : Depuis son introduction et jusqu'à présent, la meilleure borne inférieure connue pour le problème de bin stretching en-ligne était la simple borne de 4/3. Ceci s'explique en partie par la quasi-impossibilité d'utiliser les techniques classiques d'ordonnancement par couches, classiques en ordonnancement en-ligne pour prouver des bornes inférieures, du fait que la solution optimale du problème soit connue par avance. Dans ce chapitre nous améliorons cette borne pour la première fois. Nous utilisons des techniques de théorie des jeux et des sciences informatiques pour mener une recherche exhaustive et nous obtenons une nouvelle borne inférieure de valeur $19/14 \approx 1.357$.

Abstract: In this chapter we present the first improvement over the classical lower bound for the bin stretching problem[1]. The lower bound is obtain by means of computation but can be verified manually (the certificate is provided in Appendix B.1). In order to obtain this bound, we model the online bin stretching problem as a game and use game theory techniques coupled with computer science and combinatorial optimization techniques to solve this game. The same techniques can be applied in order to compute lower bounds for other online or semi-online problems. We also present a first lower bound on the expected competitive ratio of randomized algorithms for the bin stretching problem.

5.1 Introduction

In the online bin stretching problem, we are given a sequence of items defined by their weights $w_i \in [0;1]$. They all have to be packed into m bins with infinite capacities. We know in advance

[1] The results of this chapter were presented in a conference (Gabay et al. 2014b) and an article (Gabay et al. 2013a) has been submitted to an international journal for publication.

that all the items can be packed into m bins with unit size. The items become available and are packed in the order of the sequence, without any knowledge on the number of remaining items and their weigths except that all items fit into m bins with unit size. The value of a solution is equal to the size of the most stretched bin, which is the maximum between 1 and the size of the largest bin. An algorithm with *stretching factor* c for the online bin stretching problem is an online algorithm which successfully packs into m bins of size c, any sequence of items fitting into m unit sized bins. That is, for any instance I, the algorithm outputs a solution with value at most c. The aim is to find an algorithm having a stretching factor as small as possible.

This problem is equivalent to the scheduling problem $Pm|online-list|C_{max}$ where we additionally know that the optimal makespan is smaller than or equal to a given value C ($Pm|online-list, known-OPT|C_{max}$ is a subcase of this problem). The parameter $online-list$ means that, as soon as a job is presented, all its characteristics are known (its processing time in our case) and this job has to be scheduled before the next job is seen. The reader can refer to Borodin and El-Yaniv (1998), Fiat and Woeginger (1998) for more details about online algorithms and computation and to Pruhs et al. (2004) for online scheduling problems.

The bin stretching problem has been introduced by Azar and Regev (2001). They proposed an algorithm of stretching factor 1.625 and proved that 4/3 is the optimal stretching factor with two bins. Other algorithms with improved stretching factor have then been proposed by Kellerer and Kotov (2013), Gabay et al. (2013e), Böhm et al. (2014) who respectively proposed algorithms with stretching factors $11/7 \approx 1.5714$, $26/17 \approx 1.5294$ and then 1.5. The best known upper bound with 3 bins is 1.375 and is due to Böhm et al. (2014).

The upper bound on the competitive ratio (the stretching factor) for this problem has been improved while, in the meantime, the best known lower bound remained the same: 4/3. In this chapter, we present new lower bounds for this problem, for both deterministic and randomized algorithms.

For the deterministic problem we derive a lower bound of value $19/14 \approx 1.3571$ with 3 bins, leaving a gap of 1/56 between the best known lower bound and upper bound for this problem. For the randomized case, we present a lower bound with value 7/6. This lower bounds holds for any number of bins greater than or equal to 2.

In the following section, we define worst-case competitive analysis and present the classical 4/3 lower bound.

5.1.1 A lower bound

An online algorithm A is *c-competitive* if, for any instance I, A provides a solution with value at most c times greater than the optimal value, *i.e.* for all instance I, we have $A(I) \leq c \times OPT(I)$. For the bin stretching problem, this yields $A(I) \leq c$ (we are guaranteed that $OPT(I) = 1$).

Our objective is to improve lower bounds on c for a given problem. Ultimately, the aim is to find the smallest competitive ratio c^* among all online algorithms for the problem. This corresponds to finding the largest value c^* such that for any online algorithm A, there exists an instance I for which $A(I) \geq c^* \times OPT(I)$.

We now present the classical online scheduling lower bound for makespan minimization, adapted to the bin stretching problem. Consider the problem with 2 bins ($m = 2$) and the two following sequences of items in the input:

$$\pi = \left(\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}\right) \qquad \pi' = \left(\frac{1}{3}, \frac{1}{3}, 1\right)$$

Obviously, both of these sequences of items can be packed into two unit sized bins. Consider a c-competitive deterministic online algorithm A for the bin stretching problem. Algorithm A must pack both of these sequences of items with stretching factor at most c.

Either A packs both of the first two items, of size $\frac{1}{3}, \frac{1}{3}$, in the same bin or in different bins. In the first case, with the sequence π, the smallest bin is filled to at least 4/3, hence $c \geq \frac{4}{3}$. Otherwise, with sequence π', the smallest bin is filled to at least 4/3, hence $c \geq \frac{4}{3}$. In both case, $c \geq \frac{4}{3}$. Therefore, the stretching factor of any online algorithm is greater than or equal to $\frac{4}{3}$.

Azar and Regev (2001) generalized this bound to any number of bins. This bound, however, has not been improved ever since.

Our aim is to improve this lower bound. Obviously, we cannot work with all possible algorithms and instances. Yet, in order to prove that a lower bound is valid, we need to prove that it is valid for all deterministic algorithms. We remark that on a given input, considering all assignments for all items is the same as considering all algorithms. In the following, we model the problem of finding lower bounds as a game and restrict the choices of the adversary. This restriction limits the set of considered instances.

5.1.2 Contribution

We derive a new worst-case lower bound, with value $19/14 \approx 1.3571$. In order to obtain this bound, we model the problem as a two-player, zero-sum game. Then, we use the so-called adversary method in which a malicious, omnipotent, adversary is playing against the algorithm to derive improved lower bounds. In online scheduling litterature, layering techniques are often used to derive lower bounds for deterministic algorithms, see e.g. Albers (1999), Bartal et al. (1994), Rudin and Chandrasekaran (2003). However, since the optimum is known in advance in the bin stretching problem, this approach is very unlikely to work. We use an automated approach based on the minimax algorithm (Neumann 1928), with alpha-beta pruning (Pearl 1982) to solve the game where the adversary has restricted choices on items weights. Moreover, to comply with the known feasibility of the corresponding bin packing problem with unit sized bins, we use constraint programming to compute feasible decisions of the adversary.

The algorithm outputs a decision tree as a proof. All decisions of the adversary are provided in this tree, for all decisions of any algorithm. The proof for the 19/14 lower bound is provided in Appendix B.1.

Similar approaches have already been applied to other problems, see e.g. Gormley et al. (2000). This computational approach relies on several classical tools of computer science and combinatorial optimization and can be generalized and applied to any online or semi-online problem. In this chapter, we demonstrate how we apply it to the bin stretching problem and how the different components are connected together.

By applying Yao's minimax principle (Yao 1977), we also obtain a lower bound with value 7/6 for the expected competitive ratio of any randomized algorithm on the bin stretching problem. The reader can refer to Epstein and van Stee (2003) for several applications of Yao's principle on scheduling problems.

5.1.3 Outline

In Section 5.2, we model the problem of finding lower bounds for bin stretching algorithms as a game. Then, in Section 5.3, we present the algorithm and cuts we use to solve this game

and compute lower bounds. Finally, in Section 5.4, we present a lower bound on randomized algorithms for the bin stretching problem.

5.2 The bin stretching game

We model the problem of finding lower bounds for the bin stretching problem as the following two-player, zero-sum infinite game:

BIN STRETCHING GAME

Player 1 chooses a positive integer m. Then, successively, until Player 1 chooses Stop:

1. Player 1 (the *adversary*) chooses a feasible weight defining an item or Stop.
2. Player 2 (the *algorithm*) selects an integer $i \in \{1,\dots,m\}$ and packs the item into the bin B_i.

The payoff of Player 1 is equal to $\max(1, \max_{i=1,\dots,m} w(B_i))$, where $w(B_i) = \sum_{j \in B_i} w_j$.

Let w_j be the weight selected by Player 1 on iteration j. The weight w_j is feasible if and only if the bin packing problem with m bins of unit capacities and items with weights $w_1,\dots,w_j$ is feasible. The bin packing problem is strongly $\mathcal{NP}$-hard (Garey and Johnson 1979). However, we can consider that the adversary is an oracle and can easily compute this problem.

Additionally, this is a game with complete information which means that both players know all the decisions taken and recall the history of the game.

The payoff of Player 1 is c, the stretching factor, while the payoff of Player 2 is $-c$. This game is a minimax game where Player 1 aims at maximizing c while Player 2 aims at minimizing c. An algorithm for the bin stretching problem defines a behavior for Player 2. The worst-case competitive ratio of an algorithm is equal to the supremum of c when Player 2 acts according to the algorithm. The supremum on the payoff of Player 1 in this game is equal to the value c^*.

It is easy to see that this game is infinite since the adversary can provide the input $w_j = 1/2^j$, for $j = 1,\dots,\infty$. Hence, we cannot explore all feasible choices of the adversary unless we restrain them. To cope with this issue, we actually consider that Player 1 has the following behavior: at the beginning of a game, Player 1 chooses a positive integer C. Then, all the weights choosen by Player 1 are in $\{1/C, 2/C,\dots,1\}$ (and he can choose Stop as well). Considering this subset of adversaries, the game is finite: Player 1 has at most mC choices before the game is over.

In order to prove that a value c is a lower bound on c^*, it is "sufficient" to show that for any algorithm, there is an instance such that the stretching factor of the algorithm is greater than or equal to c. We cannot consider all algorithms but, on a given instance, there is a finite number of decisions for Player 2 and considering all decisions is actually the same as considering all algorithms. Hence, we only need to show that, for any decision of Player 2, there is a sequence of decisions from Player 1 leading to a solution with value at least c. Figure 5.1 illustrates this for the 4/3 lower bound. All decisions from Player 2 are considered while only one decision for each branch is provided for Player 1.

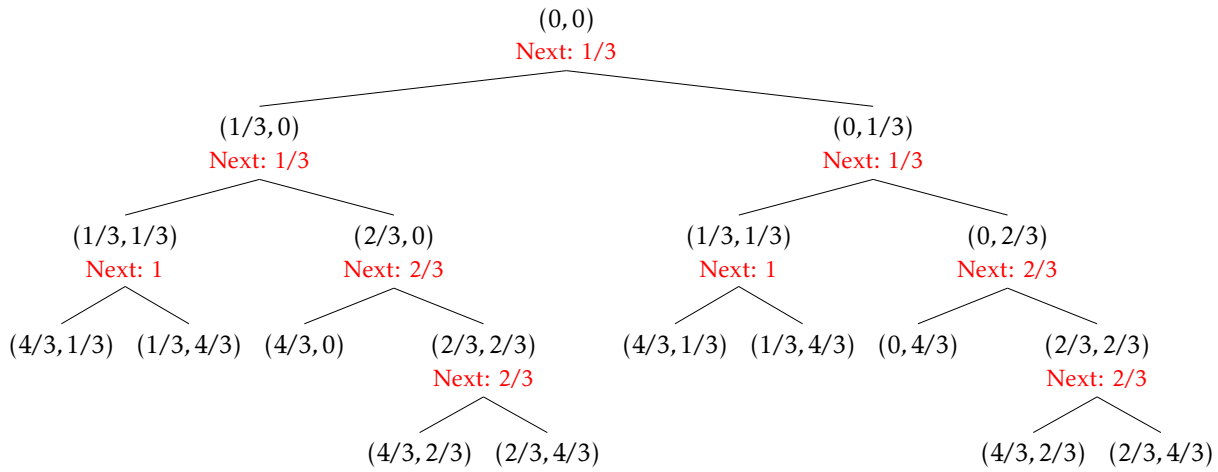


Figure 5.1: 4/3 lower bound decision tree. Player 1 decisions are the "Next: w_i". The pairs (w_1, w_2) are corresponding to the space used in the bins.

5.3 Implementation

In order to solve the game, that is, find a strategy for Player 1 maximizing c, we implement the minimax algorithm (depth-first search) for the game previously described. We apply the alpha-beta pruning with several additional cuts. Remark that considering unit capacities and weights in $\{1/C, 2/C, \ldots, 1\}$ is the same as considering the capacities of the bins to be C and weights in $\{1, \ldots, C\}$. Hence, we represent an item by an integer in $\{1, \ldots, C\}$ and a bin by a list of integers, corresponding to the items in the bin.

5.3.1 Decisions on items weights and assignments

In order to decide whether an item can be proposed by the adversary, we apply simple lower and upper bounding results on the corresponding bin packing problem, including the additional new item. Some of these are described in the following paragraphs.

Let $w_1, \ldots, w_j$ be the weights of the items up to step j, sorted in non-increasing order. We verify that $\sum_{i=1}^{j} w_i \leq mC$ and $w_m + w_{m+1} \leq C$. Let $k = \max\{i | w_i > C/2\}$ ($k = 0$ if there are no such items) and $l = \max\{i | w_i = C/2\}$ ($l = k$ if there are no such items), we also ensure that $2k + l \leq 2m$. If any of the previous inequalities is not verified, then the weight is infeasible.

Then, if the problem was not proven infeasible, we compute the best fit decreasing heuristic on the input. If it is feasible, then the new item is accepted. Otherwise, we need an exact approach to determine whether current item is feasible.

At this step, we can also compute refined lower bounds such as L_2 and L_3 from Martello and Toth (1990b). However, we choose to not compute these bounds since, in our experiments, sub-problems are small and it is computationally more efficient to immediately solve these problems with an exact solver. With larger number of bins, one should consider computing these bounds before computing the exact solution of the problem.

In our case, we use constraint programming to solve the bin packing problem. This choice was motivated by the small sizes of the problems that have to be solved. We did not implement

a dedicated approach since the time spent checking feasibility is dominated by the time spent in the rest of the algorithm.

In general, for a semi-online problem, one can use any approach, including integer programming, branch and bound or any exact dedicated approach to determine whether a move for the adversary is feasible. We can also use heuristic approaches with the risk of not being able to find a lower bound because of a missed feasible move. However, when a move is validated, it has to be really feasible in order to ensure the correctness of the results of the algorithm.

Eventually, it is not necessary to verify feasibility for all items: once an item is proven feasible, all smaller items are feasible as well. Hence, by considering adversary choices by decreasing order of the weights of the items, we only have to find the first feasible item; then all other choices are smaller items, hence they are feasible.

5.3.2 Cuts

The size of the minimax tree is exponential in m and C. Hence, we have to find a way to cut branches in order to be able to compute optimal solutions of the restricted game. The first step to reduce the minimax tree is to break symmetries on the game: permutations on the bins are actually corresponding to identical solutions. Moreover, from Player 2 point of view, the items in the bins do not matter. Only the bin sizes matter. So, the configuration $((6,3),(4,5),\emptyset)$ is actually the same as $((7,1,1),\emptyset,(3,6))$. However, these two configuration are different from Player 1 point of view since he needs to ensure that the resulting bin packing problem will be feasible. Yet, to both Player 1 and Player 2, the following configurations are equivalent: $((6,5,1,2),(7,7),\emptyset)$, $((6,1,7),(5,2,7),\emptyset)$. These two nodes can actually be described as: $(\{(0,1),(14,2)\},\{(1,1),(2,1),(6,1),(7,2)\})$ which is the same node to both players. In the first set of pairs, the weights of the bins and their multiplicities are given, while the second set of pairs denotes the weights of the items and their multiplicities. All nodes having the same encoding are equivalent.

We use this encoding to represent a (partial) solution and we take advantage of it in two ways: when Player 2 packs an item, the number of edges to explore is equal to the cardinality of the first set of the pair, which is less than or equal to m. Moreover, we use memoization (Michie 1968) (and compression) to store and recall the results of the nodes we have already computed and bin packing problems which have already been non-trivially solved. Since we use an alpha-beta pruning, which is further described, we also have to store the values of α and β on the node, in order to be able to determine whether the value of a node shall be recomputed when it is recalled.

We apply an alpha-beta pruning to the minimax algorithm. The idea is to maintain a lower bound α and an upper bound β on the stretching factor. The pruning works as follows: on a maximizer node, once it is known that the solution of this node will be better than the solution of another node having the same parent (this parent is a minimizer), it is not necessary to explore any other choice. And similarly for minimizer nodes.

Since the adversary is computing a solution against all algorithms, we can consider several particular algorithms. Especially, we can consider the algorithm packing all remaining items into the currently smallest bin. We do not know the remaining items, but we know that the sum of their weights cannot exceed $mC - \sum_{k=1}^{j} w_k$. Let B_i be the smallest bin, if $w(B_i) + mC - \sum_{k=1}^{j} w_k \leq \alpha$ then we can immediately proceed to a β cut-off.

Additionally, we are aiming at strictly improving known lower bounds and we know that some competitive ratio can be achieved by some deterministic algorithms. So, we start the exploration with a lower bound which is equal to the best known lower bound ($\alpha = \lfloor C\tilde{c} \rfloor$, where $\tilde{c}$ is

the best known lower bound on c^*) and an upper bound which is equal to the competitive ratio of the best algorithm ($\beta = 26C/17$).

For most values, the lower bound will not be increased, so we improve this approach by dividing it into two steps: in the first step, we determine whether the lower bound can be improved. If so, in a second step, we determine the new best lower bound. Otherwise, we go on to the next value. Thus, we start with $\alpha = \lfloor C\tilde{c} \rfloor$ and $\beta = \lfloor C\tilde{c} \rfloor + 1$; such close values allow very early cut-offs. When the algorithm is over, the value is either α or β. In the first case, the lower bound cannot be improved for current values of m and C. It is over, we can try a new set of parameters m, C. In the other case, we know that $\frac{\lfloor C\tilde{c} \rfloor + 1}{C}$ is a new, strictly larger lower bound. We re-run the algorithm with $\beta = 26C/17$ to see if we can further improve this new lower bound.

5.3.3 Results

We implemented the algorithm in Python and used Choco (Jussien et al. 2008) as a constraint programming solver to solve bin packing problems (we also implemented approaches using integer programming). The source code of our implementation is available online[2].

Running the program with parameters $m = 3$ and $C = 14$, we obtain the lower bound $19/14 \approx 1.357$. We backtracked the results and verified them manually. The proof is provided in Appendix B.1.

We used PyPy interpreter on a computer running Linux and equipped with an Intel Core i7-2600K Processor (clock speed 3.40GHz) and 4GB of RAM to compute lower bounds with our algorithm. Some experimental results are presented Table 5.2. Using our approach, we were able to compute the results for $m = 3$ and C up to 20, and $m = 4$ and C up to 12. Using the two-step approach, we were able to prove that neither $m = 3$, $C = 21$, nor $m = 4$, $C = 13$ allow to increase the lower bound. With larger values of m, we are only able to compute results with small C and we do not get improved lower bounds. For larger values of C, the number of nodes is too large and we are facing time and memory issues. We remark that the limiting factor is the combinatorial explosion (see column #nodes, Table 5.2) and not any algorithmic factor. Optimizing the code or running it on faster computers would barely allow to compute solutions for the next values of C.

The results presented Table 5.2 (except the last column) concern the single step approach, with pruning parameters initialized to $\alpha = \lfloor 4C/3 \rfloor$ and $\beta = 26C/17$. The last column gives the number of nodes in the first stage of the two-steps approach, with $\alpha = \lfloor C\tilde{c} \rfloor$ and $\beta = \lfloor C\tilde{c} \rfloor + 1$.

The column #calls corresponds to the number of times an item feasibility was verified. The column #exact is the number of calls to the exact method (that is when the item was not proven to be feasible or infeasible by a heuristic or a lower bound). The combinatorial explosion is very well illustrated in column #nodes where we can see that even with many efficient cuts, we cannot tackle much larger problems. Table 5.2 also shows that the time spent verifying items feasibility is negligible compared to the whole time spent. Time spent is approximately linear in the number of nodes, except for the largest instances ($\approx 2 \times 10^7$ nodes) since the computer is out of memory and swaps, making the algorithm very inefficient.

In order to improve this algorithm, the first step would be to reduce memory usage by restraining the amount of memoized data. Then, we could use a breadth-first search and for each depth, use a heuristic to select a sample of least promising nodes. Exploring these nodes in depth, will allow some early cut-offs. Another approach is to set $C = 1$ and select random weights in $]0;1]$. Then, we can run the algorithm on many random samples of items, hopefully resulting in

[2] https://github.com/mgabay/Bin-Stretching-Lower-Bounds

m	C	c	*feasibility check*			*overall*		*first step*
			#calls	*#exact*	*time* (s)	*#nodes*	*time* (s)	*#nodes*
3	10	—	4 687	37	0.4	49 055	1.4	41 753
	11	—	14 802	141	1.2	168 380	3.4	141 176
	12	—	9 125	63	0.5	118 925	2.2	98 186
	13	—	32 538	209	1.1	458 183	5.8	384 052
	14*	**19/14**	82 868	644	2.2	1 240 619	14.0	286 845
	15	—	55 929	344	1.2	890 291	10.1	702 449
	16	—	196 835	1142	3.3	3 384 144	35.5	2 901 483
	17	23/17	207 133	1 804	4.0	3 728 386	40.8	1 620 468
	18	—	303 725	1 646	4.3	5 692 383	57.2	4 652 427
	19	—	1 045 692	4 958	21.0	21 262 246	1225.1	18 653 870
	20	27/20	977 992	6 191	21.9	20 283 070	1046.7	11 446 232
4	7	—	6 622	50	0.4	50 642	1.6	39 946
	8	—	28 099	182	1.1	254 344	4.5	193 474
	9	—	30 991	98	0.8	331 112	4.8	266 926
	10	—	127 063	721	2.6	1 442 281	19.5	1 106 147
	11	—	503 560	3114	7.5	6 365 822	81.7	5 195 618
	12	—	491 497	1974	5.8	6 718 232	89.7	5 158 805
	13	—	1 540 000	>8000	>41.6	>22 900 000	>3600	19 956 339

Figure 5.2: Numerical results on some inputs. Column 3, c is the best lower bound on the competitive ratio obtained for the instance. "—" means that the 4/3 lower bound was not improved.

an improved lower bound. We ran several tests using random weights distributions but we did not obtain improved lower bounds with this approach.

5.4 Lower bound on randomized algorithms

In this section, we present a lower bound on the expected competitive ratio for any randomized algorithm for the bin stretching problem. This bound is a simple generalization of the results from the 4/3 deterministic lower bound of Azar and Regev (2001).

We recall that a randomized algorithm can take its decisions at random, according to probability distributions. While the previous worst case analysis holds, when designing a random algorithm, the aim is to minimize the expected competitive ratio rather than the worst case competitive ratio.

Theorem 5.1. *Any randomized algorithm for the online bin stretching problem, has an expected competitive ratio of at least* 7/6*, for any number of machines* $m \geq 2$.

Proof. We use Yao's minimax principle and consider a randomized adversary against a deterministic algorithm. Yao's principle states that a lower bound $\tilde{c}$ for the competitive ratio of deterministic algorithms on a fixed distribution over inputs is also a lower bound for any randomized algorithms.

Let m be the number of machines. We consider the input from Azar and Regev (2001), with an additional distribution of probabilities:

- with probability p, the input is m items of weight 1/3, followed by m item of weight 2/3;
- with probability $1-p$, the input is m items of weight 1/3, followed by an item of weight 1.

Both of these inputs are obviously feasible.

Any deterministic algorithm either packs the m first items in different bins or at least two of them are in the same bin. In the first case, the first input yields solutions with value at least 1, while the second input yields solutions with value at least 4/3. Otherwise, the first input yields solutions with value at least 4/3, while the second input yields solutions with value at least 1.

Hence, the performance of any deterministic algorithm packing the m first items in different bins is at least $p \times 1 + (1-p) \times 4/3$, while the other deterministic algorithms yield solutions with value at least $p \times 4/3 + (1-p) \times 1$. The minimum of both of these values is maximized for $p = 1/2$. In such case, the performance of any deterministic algorithm is at least 7/6 on this input.

Hence, by Yao's principle, 7/6 is a lower bound on the competitive ratio of any randomized algorithm for the online bin stretching problem. □

5.5 Conclusion

By modeling the bin stretching problem as a game and solving this game with computer science techniques we provided a first improved lower bound. For the 3 machines case, the new 19/14 lower bound reduces the gap between lower and upper bounds by more than a factor of two compared to the 4/3 lower bound.

Based on the tree, it is not obvious to find out whether this bound can be generalized to any number of bins $m \geq 4$.

We also presented a first lower bound for randomized algorithms for the bin stretching problem. This bound of 7/6 holds for any number of bins $m \geq 2$.

The approach can be generalized and applied to many other packing or scheduling, online or semi-online problems. Compared to layering techniques, for multiple machines problem, there is however a trade-off on the generality of the bound: the lower bound cannot easily be generalized to any number of machines.

Because of the initial knowledge that the bin packing problem with the instance is feasible, there is little hope that layering techniques could work. Future research should focus on finding more general lower bounds. For instance, how to design a computational approach whose results could be generalized for all values of m. Reducing the search space to explore only special types of structured tree would postpone the combinatorial explosion but then it would be less likely to find improved lower bounds. Another subject for further research is to find good distributions of randomized item weights. By imposing a structure on the distribution of the weights of the items and running the algorithm on many input it is maybe feasible to improve the lower bounds.

Chapter 6

Vector Bin Packing with Heterogeneous Bins

Résumé : Nous proposons un modèle pour le problème de placement d'objets dans des récipients hétérogènes. Ce modèle est particulièrement adapté à la modélisation du placement de processus dans des centres de traitement des données. Nous implémentons un grand nombre d'heuristiques simples pour ce problème et les utilisons pour résoudre des instances classiques des problèmes de bin packing et vector bin packing, des jeux d'instances générés aléatoirement pour le problème de placement dans des récipients hétérogènes et des instances issues du challenge ROADEF 2012 pour le problème de réaffectation de machines. Nous présentons les résultats sur ces instances et également des propriétés du problème de réaffectation de machines. Ces propriétés nous permettent d'adapter nos heuristiques à ce problème et demeurent par ailleurs intéressantes dans le cadre d'une approche en vue d'optimiser ce problème.

Abstract: In this chapter, we introduce a generalization of the vector bin packing problem, where the bins have variable sizes[1]. This generalization can be used to model virtual machine placement problems. In particular, we study the machine reassignment problem. We propose several greedy heuristics for the vector bin packing problem with heterogeneous bins and show that they are flexible and can be adapted to handle additional constraints. We present structural properties of the machine reassignment problem. These properties can be useful to anyone who is interested in the machine reassignment problem. In our case, we use them to adapt our heuristics to this problem. We present numerical results on vector bin packing benchmarks, randomly generated instances for the vector bin packing problem with heterogeneous bins and Google realistic instances for the Machine Reassignment Problem.

[1] The results presented in this chapter are joint work with Sofia Zaourar. Preliminary results were presented during an invited talk in Poznań (Gabay 2012) and in conference (Gabay and Zaourar 2012), both times from the ROADEF challenge point of view. An article (Gabay and Zaourar 2013) has been submitted for an international journal for publication. The content of this chapter is the same as the revised version of the article.

6.1 Introduction

In service hosting and virtualized hosting, services or virtual machines must be assigned to clustered servers. Each server has to provide enough resources, such as CPU, RAM or disk, in order to have all of its processes running. The machine reassignment problem, proposed by Google for the ROADEF/EURO challenge 2012[2], is such an assignment problem, with additional constraints and a cost function to minimize.

In this chapter, we propose a modeling framework for these packing problems and a greedy heuristic framework to find feasible assignments. Several classical bin packing heuristics are adapted and new variants are proposed; a worst-case complexity analysis of these algorithms is also provided.

We present some structural properties of the machine reassignment problem. These properties are used to adapt our heuristics to the machine reassignment problem. Finally, we provide experimental results on vector bin packing benchmarks, on a new benchmark for the vector bin packing problem with heterogeneous bins and on realistic instances for the machine reassignment problem.

6.1.1 Bin Packing Problems

In the classical Bin Packing (BP) problem, we are given a set $\mathcal{I} = \{I_1, \ldots, I_n\}$ of n items, a capacity $C \in \mathbb{N}$ and a size function $s : \mathcal{I} \to \mathbb{N}$. The goal is to find a feasible assignment minimizing the number of bins used. A feasible assignment of the items into N bins is a partition $P_1, \ldots, P_N$ of the items, such that for each P_k, the sum of the sizes of the items in P_k does not exceed the capacity C. In the decision version of this problem, the number of bins N is part of the input and the objective is to decide whether all the items can be packed using at most N bins. This problem is known to be strongly NP-hard (Garey and Johnson 1979).

Garey et al. (1976) introduced a generalization of this problem, called Vector Bin Packing (VBP) or d-Dimensional Vector Packing (d-DVP). In this problem, the weights of the items are described by a d-dimensional vector: $(s_i^1, \ldots, s_i^d)$ and bins have a capacity C in all dimensions. A feasible assignment of the items into N bins is a partition $P_1, \ldots, P_N$ of the items such that for each P_k, on each dimension, the sum of the sizes of the items in P_k does not exceed the capacity:

$$\forall k \in \{1, \ldots, N\},\ \forall j \in \{1, \ldots, d\},\ \sum_{i \in P_k} s_i^j \leq C$$

Vector bin packing is often used to model virtual machine placements (Lee et al. 2011, Panigrahy et al. 2011, Stillwell et al. 2010). In such cases, all machines are supposed to have the same capacities. This could be the case when a new computer cluster is built. However, as it grows and servers are renewed, new machines are introduced and the cluster becomes heterogeneous.

Hence, we are interested in a further generalization of this problem, where each bin has its own vector of capacities $(c_k^1, \ldots, c_k^d)$ and the goal is to find a feasible packing of the items. We call this problem the Vector Bin Packing with Heterogeneous Bins (VBPHB) problem. This problem has not been studied yet to the best of our knowledge. VBPHB can be used to model previously mentioned virtual machine placement problems in a realistic heterogeneous environment.

[2] http://challenge.roadef.org/2012/en/

6.1.2 Machine Reassignment Problem

The machine reassignment problem was proposed by Google for the 2012 ROADEF Challenge. This challenge was based on problems occurring in Google's data centers and realistic instances were provided. In the machine reassignment problem, a set of processes needs to be (re)assigned to a set of machines. There are d resources and each machine (resp. process) has its own capacity (resp. requirement) for each resource. There are also additional constraints presented in Section 6.4. The aim is to find a feasible assignment minimizing a weighted cost.

In the challenge, an initial feasible solution was provided. Therefore, local search based heuristics were a natural (and successful) approach. Local search aims at improving iteratively a given solution by applying small modifications, and is well-suited to quickly improve solutions of very large-scale problems. See Aarts and Lenstra (1997) for a detailed survey on local search.

When using local search, the search space is limited by the initial solution and the set of accepted moves. This space can be enlarged by running several local searches with different parameters and starting with diversified initial solutions. Feo and Resende (1989, 1995) designed the Greedy Randomized Adaptive Search Procedure (GRASP) which is an optimization algorithm combining local search with diversified initial solutions. GRASP is an iterative process where one successively creates a new feasible solution, then optimizes it using a local search algorithm. When applying a GRASP heuristic to the machine reassignment problem, VBPHB arises as a subproblem for generating new initial solutions. We explain how we can handle additional constraints and find new feasible assignments by solving VBPHB problems in Section 6.4.

Competitors in ROADEF challenge have exposed their approaches and results in Gavranović et al. (2012), Lopes et al. (2014), Mehta et al. (2012), Portal (2013). Saber et al. (2014) introduced and studied the generalization of the Machine Reassignment Problem to multi-objective optimization.

6.1.3 Outline

In this chapter, we study different heuristics to solve VBPHB and point out properties of the machine reassignment problem. In Section 6.2, we define VBPHB and present related work on vector bin packing. In Section 6.3, we propose several heuristics for this problem. In Section 6.4, we discuss structural properties of the machine reassignment problem and adapt our heuristics to this problem. Experimental results are reported Sections 6.3.5 and 6.4.5.

6.2 Vector Bin Packing Problem with Heterogeneous Bins

An instance of the vector bin packing problem with heterogeneous bins is defined by C an $\mathbb{N}^{N\times d}$ capacity matrix and S an $\mathbb{N}^{n\times d}$ size matrix, where c_k^j (resp. s_i^j) denotes the capacity of bin k (resp. the size of item i) in dimension j.

We define the following index sets: $\mathcal{B} = \{1,\dots,N\}$ for the bins, $\mathcal{I} = \{1,\dots,n\}$ for the items, and $\mathcal{D} = \{1,\dots,d\}$ for the dimensions.

The problem is to find a feasible assignment $x \in \mathbb{N}^{n\times N}$ of the items into the bins such that:

$$\sum_{i \in \mathcal{I}} s_i^j x_{i,k} \leq c_k^j \qquad \forall j \in \mathcal{D},\ \forall k \in \mathcal{B} \tag{6.1}$$

$$\sum_{k \in \mathcal{B}} x_{i,k} = 1 \qquad \forall i \in \mathcal{I} \tag{6.2}$$

$$x_{i,k} \in \{0,1\} \qquad \forall i \in \mathcal{I},\ \forall k \in \mathcal{B} \tag{6.3}$$

Inequality (6.1) models the capacity constraints while constraints (6.2) and (6.3) ensure that each item is assigned to a bin.

Let $c = max_{j,k} c_k^j$; observe that by adding one dimension and having bins of capacities c in all dimensions, you can transform any instance of the vector bin packing problem with heterogeneous bins in an instance of the vector bin packing problem by adding artificial items with requirements $(c - c_k^1, \ldots, c - c_k^d, c)$ and extending all other items requirements with 0 in dimension $d+1$. Yet, we introduce the vector bin packing problem with heterogeneous bins because these problems are actually very different. Indeed, vector bin packing with heterogeneous bins is a natural framework to model heterogeneous data centers for instance but it also naturally accounts for rare resources. For instance, few machines may be equipped with GPUs so using them for processes which do not require GPUs can easily lead to infeasible solutions. In vector bin packing problem, all bins are identical and if we introduce artificial items, we conceal the fact that there are rare resources which may not be wasted.

6.2.1 Related work

Since VBPHB is a generalization of bin packing, this problem is strongly NP-hard. Moreover, Chekuri and Khanna (1999) proved that 2-DVP is APX-hard and showed $d^{1/2-\epsilon}$ hardness of approximation. Woeginger (1997) proved that there is no asymptotic PTAS (unless P=NP). Hence, as a generalization of d-DVP, the optimization version of VBPHB (where the different bins are types on bins, each one with a cost) is APX-hard and cannot have an asymptotic PTAS.

Maruyama et al. (1977) generalized classical bin packing heuristics into a general framework for VBP.

There are many theoretical results for the vector bin packing problem: Kou and Markowsky (1977) studied lower and upper bounds and showed that the worst case performance ratio for the generalization of some classical bin packing algorithms is larger than d, where d is the dimension. Yao (1980) proved that any $o(n \log n)$ time algorithm has a worst case performance ratio bigger than d. Bansal et al. (2006) proposed a randomized $(\log d + 1 + \epsilon)$-approximation. Their algorithm is polynomial for fixed d. Spieksma (1994) proposed two lower bounds for 2-DVP and a branch-and-bound algorithm using these bounds. Caprara and Toth (2001) analyzed several lower bound for 2-DVP and showed that the lower bound obtained by the linear programming relaxation of the (huge) integer programming formulation they propose, dominates all these bounds. Chang et al. (2005) used 2-DVP to model a packing problem where steel products have to be packed into special containers and they proposed a heuristic. Caprara et al. (2003) showed that there is a PTAS for d-DVP if all items sizes are totally ordered. Shachnai and Tamir (2003) studied Data Placement problem as an application of VBP and proposed a PTAS for a subcase of VBP. Karp et al. (1984) studied VBP where all items sizes are drawn independently from the uniform distribution over [0,1]. They proved that the expected wasted space by the optimal solution is

$\Theta(n^{\frac{d-1}{d}})$ and proposed an algorithm that tries to pack two items in each bin and has the same expected wasted space.

Stillwell et al. (2010) implemented and compared several heuristics for VBP with additional real-world constraints in the case of virtualized hosting platforms. They found out that the algorithm which is performing the best is the choose pack heuristic from Leinberger et al. (1999) with items sorted by decreasing order of the sum of their requirements.

Brandao and Pedroso (2013) generalized the arc-flow formulation from Valério de Carvalho (1999) for bin packing and cutting stock to the vector bin packing problem and proposed improvements for this approach. They experimented their approach on academic benchmarks and closed many open instances.

Other works have considered the variable sized bin packing problem in which there are several types of bins and the aim is to minimize the sum of bin costs. Han et al. (1994) studied the 2-dimensional vector bin packing problem in which items have specific requirements for each bin and proposed exact and heuristic approaches along with a process to improve lower bounds.

In the classical First Fit Decreasing (FFD) heuristic, one has to select the *largest* item and then pack it into a bin. Hence, if one generalizes this heuristic to the multidimensional case, it has to be determined how to measure and compare items. Panigrahy et al. (2011) presented a generalization of the classical First Fit Decreasing (FFD) heuristic to VBP and experimented several measures. A promising measure is the $DotProduct$ which defines the *largest* item as the item that maximizes some weighted dot product between the vector of remaining capacities and the vector of requirements for the item.

6.3 Heuristic framework

We generalize the classical First Fit Decreasing (FFD) and Best Fit Decreasing (BFD) heuristics to VBPHB. Algorithm 6.1 is the classical BFD algorithm. Panigrahy et al. (2011) proposed a different approach of this algorithm which focuses on the bins, as detailed in Algorithm 6.2. In order to use these algorithms in multidimensional packing problems, one needs to define an ordering on bins and items.

This ordering can be defined using a measure: a size function which returns a scalar for each bin and item. In the following sections we propose several measures based on the remaining capacities of the bins and decisions made.

Since orderings are based on a measure, both the orderings of items and bins may change in the course of the algorithm. Observe that if the order is unchanged then item centric (Algorithm 6.1) and bin centric (Algorithm 6.2) heuristics give the same results (either both are infeasible or both are feasible and return the same solution).

Remark that any greedy algorithm for this problem can be reduced to a best fit item centric heuristic by computing the next decision of the algorithm in the measure and returning size 2 for chosen item, size 0 for chosen bin and size 1 for other bins and items.

6.3.1 Measures

In order to sort items and bins, we define a measure. Let $i \in \mathcal{I}$, $k \in \mathcal{B}$, $j \in \mathcal{D}$. We define $\mathcal{I}_r$ as the set of unpacked items and $\mathcal{B}_r$ as the set of remaining bins (unless we are using a bin centric approach, $\mathcal{B}_r = \mathcal{B}$). We denote by r_k^j the remaining capacity of bin k in dimension j, by $C(j)$

Algorithm 6.1: BFD Item Centric

```
while There are unpacked items do
    Compute sizes
    Pack the biggest item into the smallest feasible bin
    if the item cannot be packed then
        return Failure
return Success
```

Algorithm 6.2: BFD Bin Centric

```
while The list of bins is not empty do
    Compute sizes
    Select b the smallest bin
    while An unpacked item fits into b do
        Compute sizes
        Pack the biggest feasible item into b
    Remove b from the list of bins
if An item has not been packed then
    return Failure
return Success
```

the total remaining capacity in dimension j and by $R(j)$ the total requirement in dimension j: $C(j) = \sum_{k \in \mathcal{B}_r} r_k^j$ and $R(j) = \sum_{i \in \mathcal{I}_r} s_i^j$.
A natural idea to define a scalar size from vector size is to take a weighted sum of the vector components. We define the following sizes:

$$S_{\mathcal{B}}(k) = \sum_{j \in \mathcal{D}} \alpha_j r_k^j \qquad \forall k \in \mathcal{B}$$

$$S_{\mathcal{I}}(i) = \sum_{j \in \mathcal{D}} \beta_j s_i^j \qquad \forall i \in \mathcal{I}$$

where α and β are two scaling vectors. We propose three different scaling coefficients: $\frac{1}{C(j)}$, $\frac{1}{R(j)}$ and $\frac{R(j)}{C(j)}$. The first ratio normalizes based on bins capacities. The second ratio normalizes based on items requirements. The last coefficient takes both remaining capacities and requirements into account and normalizes on the rarity of resources.

We can also define the size of an item by choosing its maximal normalized requirement over the resources. We obtain the priority measure:

$$S_{\text{prio}}(i) = \max_{j \in \mathcal{D}} \frac{s_i^j}{C(j)} \qquad \forall i \in \mathcal{I}$$

Sizes are either computed once and for all, before the first run of the algorithm, in such

case we say that the measure and the resulting heuristics are *static*, or at every iteration of the algorithm. In this latter case, we have *dynamic* measures and heuristics.

For static measures, the ordering is fixed and Algorithms 6.1 and 6.2 become first fit heuristics. Observe that for a same static measure, Algorithms 6.1 and 6.2 return the same results.

Both the *static* and *dynamic* heuristics are considered in this chapter. In the following, we choose $\alpha = \beta$. As a consequence, the measure S has the following property:

Property 6.1. *If $\alpha = \beta$ and $S_{\mathcal{B}}(k) < S_{\mathcal{I}}(i)$ then the item i does not fit into the bin k.*

Proof. If $S_{\mathcal{B}}(k) < S_{\mathcal{I}}(i)$, then $\sum_{j\in\mathcal{D}} \alpha_r(r_k^j - s_i^j) < 0$. Since both r and s are positive, $r_k^j < s_i^j$ for some j. □

6.3.2 Bin balancing

In Section 6.3.1, we presented measures which yield different heuristics when combined with Algorithms 6.1 and 6.2. However, since bin capacities are different, it is hard to predict which resource, bin or item will be the bottlenecks. Moreover, we can take advantage of the fact that we are only interested in finding feasible assignments. Instead of packing as many items as possible in a bin, we can try to balance the load. The Permutation Pack and Choose Pack heuristics from Leinberger et al. (1999) use such an approach to pack items. We propose another approach: using the item centric heuristic, pack current item into the first feasible bin. Then, move this bin (or a subset of the bins) to the end of the list of bins. This approach is detailed in Algorithm 6.3. Line 7, l_B is updated by one of the two following ways:

- *Single bin balancing:* Used bin is moved to the end of the list
- *Bin balancing:* All bins tried (including the successful bin) are moved to the end of the list in the same order: let l be the new list. We have:
  $l(1) = l_B(j+1), \ldots, l(N-j) = l_B(N), l(N-j+1) = l_B(1), \ldots, l(N) = l_B(j)$
  (this is actually achieved by a simple modulo)

Algorithm 6.3: Bin Balancing Heuristics

```
1  Sort l_B (bins list) and l_I (items list)
2  while There are unpacked items do
3      Let I be the biggest unpacked item
4      for j = 1 to N do
5          if item I can be packed into l_B[j] then
6              Pack I into l_B[j]
7              Update l_B
8              break
9      if I has not been packed then
10         return Failure
11 return Success
```

The main idea of this algorithm is that once an item is assigned to a bin, we try to assign the following items to other bins, in order to prevent critical bins from being overwhelmed too early.

6.3.3 Dot Product

We generalize the *DotProduct* heuristic from Panigrahy et al. (2011). In this heuristic we select the feasible pair (i,k) maximizing the (weighted) dot product $s_i \cdot r_k$ (resp. $\sum_{j\in\mathcal{D}} \alpha_{i,k} s_i^j r_k^j$) and pack item i into bin k.

We propose three variants of this heuristic: maximize the dot product ($\alpha_{i,k} = 1$), or the weighted dot product with $\alpha_{i,k} = (\|s_i\|_2 \|r_k\|_2)^{-1}$ or $\alpha_{i,k} = \|r_k\|_2^{-2}$.

On the first iteration, we compute dot products for all feasible pairs, then store these values. On the following iterations, only the dot products concerning the bin where an item has just been packed are computed. The worst case time and space complexity for initializing sizes is $\mathcal{O}(dnN\log(nN))$. The complexity of computing costs afterwards is at most $\mathcal{O}(dn)$ and the list can be maintained in $\mathcal{O}(n\log(nN))$.

This heuristic maximizes the *similarity* of a bin and an item (the scalar projection of the item sizes onto the bin remaining capacities). Moreover, we need to be able to compare these dot products for all pairs of bins and items. On one hand, if we do not scale the vectors, then we maximize both the similarity and the size used. On the other hand, if we normalize both sizes and capacities, we minimize the angle between the two vectors. Eventually, if we re-scale by $\frac{1}{\|r_k\|_2^2}$, then we focus on maximizing the scalar projection of the item and maximize similarity.

6.3.4 Complexity

We denote $p = \max(n,N)$. In the worst case scenario, both the item centric and the bin centric algorithms behave as shown in Algorithm 6.4. Hence, the overall time complexity is $\mathcal{O}(dp^2 + p^2\log p)$. The space complexity is $\mathcal{O}(p^2 + dp)$ for the dot product and $\mathcal{O}(dp)$ for other measures.

Algorithm 6.4: Worst-case heuristics behavior

```
1 Initialize sizes                                  // O(dp²) (DotProduct)
2 for i = 1 to p do                                 // p×
3     Compute sizes                                 // O(dp) (for given measures)
4     Sort lists                                    // O(p log p)
5     Pick an item                                  // O(1)
6     Pack it                                       // O(dp)
```

Obviously, one shall not implement the algorithm as described by Algorithm 6.4. When using a static measure, bins and items should only be sorted at the beginning of the algorithm. The overall complexity will be $\mathcal{O}(dp^2)$.

Moreover, when checking whether an item fits into a bin, we can stop on the first dimension where the remaining capacity is smaller than the size of the item. Furthermore, when using one of the measures described in Section 6.3.1 or any other measure verifying Property 6.1, we can use this property to avoid checking feasibility when $S_{\mathcal{I}}(i) > S_{\mathcal{B}}(k)$. These optimizations, however, do not improve the worst-case complexity of the algorithm.

6.3.5 Experiments

We experimented all described heuristics on academic bin packing and vector bin packing benchmarks as well as new instances which we generated. Since this problem is new in the literature, there were no benchmark available apart from machine reassignment instances whose properties are mostly unknown (it was claimed that they were generated according to Google statistics

and then obfuscated). However, since it is a generalization of vector bin packing problem, we led experiments using academic benchmarks for these problems. In the following, we present the heuristics and experiments on classical benchmark for bin and vector packing as well as experiments on our new benchmark for vector bin packing with heterogeneous bins. We first present aggregated results of our heuristics, then the new benchmark with detailed results for each heuristic.

6.3.5.1 Heuristics

We use the heuristics presented in previous sections with following measures: *none* (static, items and bins are kept as provided in the input), static shuffle, static $1/C$, static $1/R$, static R/C, dynamic shuffle, dynamic $1/C$, dynamic $1/R$ and dynamic R/C. We use these measures with the item centric, bin centric, bin balancing and single bin balancing heuristics. This gives 31 heuristics since item centric and bin centric heuristics are the same for static measures. We also use the 3 variants of the dot product heuristic, for a total of 34 heuristics.

Heuristics are implemented in Python and the focus was made on simplicity rather than efficiency. For this reason, we do not report running times. However, even with these simple implementations, all heuristics (except dot-product) run in less than 0.1 second on every instance.

Since the vector bin packing problem with heterogeneous bins is defined as a decision problem, we implemented heuristics dedicated to this approach. So the number of bins and the bins capacities are fixed when a heuristic is called.

However, academic benchmarks for vector bin packing and bin packing problems are focused on optimization, so we implemented a very simple optimization procedure. Observe that all heuristics described in this chapter are very simple and efficient so a natural way towards solving a problem efficiently is to combine them all in a best-of-many algorithm: call all heuristics on the problem and keep the best result.

We implemented a binary search procedure on the number of bins. For a fixed number of bins, heuristics are called until one succeeds or they all fail. Each heuristic (including random) is only called 0 or 1 time for a given number of bins. Obviously, one can implement these heuristics without using binary search and with a much better performance. Especially for item centric heuristics whose number of bins can actually be computed in a single pass.

For vector bin packing instances, let C_j be the capacity of all bins in dimension j. In each dimension of a vector bin packing problem, we have a bin packing problem whose minimum number of bins is smaller than or equal to the minimum number of bins in the vector bin packing problem. If we formulate all these bin packing problems using the assignment based formulation and compute and round up the maximum of the linear programming relaxation over all dimensions, we obtain a lower bound on the vector bin packing problem. This lower bound is equal to:

$$l_\infty = \max_{j \in \mathcal{D}} \left\lceil \frac{\sum_{i \in \mathcal{I}} r_i^j}{C_j} \right\rceil$$

We computed this lower bound on all open instances of the vector bin packing benchmark and used it to compare our heuristics. This lower bound, combined with our heuristics allows us to solve and prove optimality on 54 out of the 77 open instances from Brandao and Pedroso (2013). For all open instances in which this lower bound combined with our heuristics was not sufficient to close the gap, we also computed the lower bound using optimum integer solutions of the bin packing problems and in all cases the two bounds were equal.

6.3.5.2 Vector Bin Packing Benchmark

Brandao and Pedroso (2013) gathered instances on bin packing, cutting stock and vector bin packing from the literature. In our benchmark, we use all the instances on vector bin packing used in their benchmark and some of the bin packing instances they have gathered. We present our results on this benchmark Table 6.1. The benchmark is made up with bin packing, and 2 and 20-dimensional vector bin packing problems.

HARD28 data set is a selection of 28 very difficult instances from Schoenfield (2002). The BPP FLK data set was proposed by Falkenauer (1996) and is composed of random uniform instances and triplets instances in which each bin is filled with three items in an optimal solution. The SCHOLL dataset, from Scholl et al. (1997), is composed of random instances with expected number of bins smaller than or equal to 3; expected number of items per bin equal to 3, 5, 7, 9; and difficult instances with 200 items. Caprara and Toth (2001) proposed the 2CBP data set, a set of two-dimensional vector packing instances with several sizes and classes detailed in their paper. On this benchmark, we solved and proved optimality on 52 out of the 70 instances left open by Brandao and Pedroso (2013). Brandao and Pedroso (2013) elaborated the 20CBP data set, a set of 40 20-dimensional instances obtained by concatenating instances of the same classes from Caprara and Toth (2001). We solved and proved optimality on 2 out of 7 of the instances which remained open in this class.

	instances				optimality		% gap on OPT		%gap on LB	
data set	*#dim*	*#inst*	*#kopt*	$n_{\max}$	*#opt*	*#new*	*mean*	*max*	*median*	9^{th} *decile*
HARD28	1	28	28	200	5	–	1.20%	1.72%	1.40%	1.63%
BPP FLK	1	160	160	1000	7	–	4.30%	10.00%	3.89%	8.55%
SCHOLL	1	1210	1210	500	839	–	0.99%	16.67%	0.00%	1.00%
2CBP	2	400	330	201	249	52	2.62%	33.33%	0.00%	8.00%
20CBP	20	40	33	201	20	2	3.03%	14.04%	0.49%	12.50%

#dim is the number of dimensions, *#inst* is the number of instances in the benchmark, *#kopt* is the number of instances whose optimum was previously known, $n_{\max}$ is the maximum number of items in the benchmark, *#opt* is the number of optimum solutions obtained with our heuristics, *#new* is the number of new optimum found, *mean* and *max* are the average and maximum gap in percent between our solutions and the optimum when it is known, *median* and 9^{th} *decile* are the values of the median and the 9^{th} decile gaps in percent between our solutions and the lower bound (which is the optimum when it is known and the LP relaxation lower bound otherwise).

Table 6.1: Results on academic benchmark

The results of the benchmark are presented Table 6.1. We observe that the combinations of heuristics performs very well on all instances. The gap is usually very small and most of the highest %gap values are observed on instances with very few bins.

6.3.5.3 Vector Bin Packing with Heterogeneous Bins Benchmark

We generated 5 classes of instances for the vector bin packing problem with heterogeneous bins. For each of these classes, we generated 100 feasible instances for each configuration with 10, 30 and 100 bins and 2, 5 and 10 dimensions. The whole test bed contains 4500 generated instances. In this section, we say that x% of bin k has been used if the average usage of the bin is more than x, *i.e.* $\sum_{\substack{j\in\mathcal{D} \\ \text{s.t. } c_k^j\neq 0}} (\frac{c_k^j - r_k^j}{c_k^j} - \frac{x}{100}) \geq 0.$

In the following we present the instance classes together with analogies to process assignments in data centers in order to show the diverse phenomena instances were made to account for.

Instances. In the first class of instances (*Random uniform*), bin capacities are chosen independently using a uniform distribution on [10;1000]. Then, items sizes are independently drawn from a uniform distribution on $[0; 0.8 \times r_k^j]$ until at least 80% of the bin capacity is used.

These instances account for diversified machines and processes. They could represent data centers with heterogeneous machines and processes. This especially makes sense when few resources are considered (CPU, RAM, disk, bandwidth) however, as the number of resource grows, it is unlikely that such a distribution accounts for realistic instances.

The second class of instances (*Random uniform with rare resources*) is the same as the first one, except that after generating the capacities of a bin, the capacity in dimension d is set to 0 with probability 0.75. Last dimension is a rare resource.

As we explained Section 6.2, in a data center, rare resources model rare components in machines such as GPUs or physical random number generators for instance.

In the third class of instances (*Correlated capacities*), for each bin, an integer $b \in [10;1000]$ is uniformly generated. Then, each capacity $j \in \mathcal{D}$ is set to $0.9 \times b + X_j$ where X_j is an exponentially distributed random variable with rate parameter $1/(0.1 \times b)$ (standard deviation is equal to 10% of b). Items sizes are generated as in the first and second classes.

This class accounts for a set of machines which are gradually renewed. The characteristics of the machines are improving at a constant rate.

Bins in the fourth class (*Correlated capacities and requirements*) are generated as in the third one. In this class, items are generated similarly to the bins, with $b' \in [1; 0.8 \times r_k^j]$ and until at least 80% of the bin capacity is used or we failed 100 times to generate a feasible item.

This class is the same as the previous one except that requirements are now growing as the set of machines is upgraded.

In the fifth class of instances (*Similar items and bins*), bin capacities are chosen uniformly and independently on [10;1000]. For each item, size in dimension j is set to $X_j + c_k^j/5$ where X_j is an exponentially distributed random variable with rate parameter $1/(0.2 \times c_k^j/5)$ (standard deviation is equal to 20% of $c_k^j/5$). Items are generated until at least 70% of the bin capacity is used or we failed 100 times to generate a feasible item.

In these instances, items are similar to the bins they are contained within. This could occur if new machines are bought to host bunches of similar new processes.

In classes 1 to 4, on average 85% of the bins capacities are used. In class 5, 79% of bins capacities are used on average. Table 6.2 gives detailed description of average resource usage and number of items in the set of generated instances.

Results. We benchmarked all the 34 heuristics on these instances. In Tables 6.3, 6.4, 6.5, 6.6, 6.7, we provide detailed results with the number of feasible solutions obtained on each of the different

		Class 1 (unif)			Class 2 (rare)			Class 3 (cor. cap.)			Class 4 (cor. cap/req)			Class 5 (similar)		
N	d	n	u	$u_{\max}$	n	u	$u_{\max}$	n	u	$u_{\max}$	n	u	$u_{\max}$	n	u	$u_{\max}$
10	2	36	85.8%	87.6%	34	86.1%	88.6%	36	85.5%	87.6%	33	86.9%	89.0%	39	79.6%	81.3%
10	5	36	85.1%	89.4%	36	85.2%	90.4%	36	85.0%	89.6%	38	84.7%	88.5%	39	79.4%	82.8%
10	10	36	84.6%	90.6%	36	84.7%	91.0%	36	84.5%	90.5%	40	82.6%	87.9%	39	78.5%	82.9%
30	2	107	85.8%	87.0%	102	85.5%	87.1%	108	85.7%	86.9%	99	87.0%	88.1%	119	79.6%	80.6%
30	5	108	85.1%	87.7%	109	85.1%	88.0%	109	85.1%	87.8%	111	84.3%	86.7%	119	79.2%	81.2%
30	10	109	84.6%	88.1%	109	84.6%	88.5%	109	84.6%	88.4%	121	82.5%	85.5%	118	78.5%	81.0%
100	2	357	85.8%	86.4%	342	85.7%	86.6%	358	85.7%	86.4%	333	86.9%	87.5%	397	79.4%	79.9%
100	5	363	85.1%	86.6%	363	85.1%	86.9%	363	85.0%	86.5%	373	84.2%	85.6%	397	79.2%	80.1%
100	10	365	84.7%	86.6%	364	84.7%	86.9%	364	84.6%	86.7%	407	82.5%	84.2%	394	78.7%	80.1%
Average:		3.6	85.2%	87.8%	3.6	85.2%	88.2%	3.6	85.1%	87.8%	3.7	84.6%	87.0%	4.0	79.1%	81.1%

Each line corresponds to one of the instance sizes. N is the number of bins and d is the number of resources in all these instances. For each class of instances, we give n the average number of items, u the average usage over all resources and $u_{\max}$ the average maximum usage of a resource.

Table 6.2: Average bin usage in generated instances

classes for all different sizes. On instances of class 1, we also present the percent of items packed (we keep packing items even if we know that the solution will be infeasible). Figure 6.2 shows the total number of success of all heuristics on the whole benchmark.

		Instances										
	#bins	10	10	10	30	30	30	100	100	100		
	#resources	2	5	10	2	5	10	2	5	10		
	Avg #items	36	36	36	107	108	109	357	363	365	Results	
Heuristic	Sorting	#feasible									total	%feasible
Static	none	6	0	0	10	0	0	48	0	0	64	7.1%
	shuffle	7	0	0	9	0	0	57	0	0	73	8.1%
	$1/C$	87	1	21	**100**	0	0	**100**	0	0	309	34.3%
	$1/R$	87	1	20	**100**	0	0	**100**	0	0	308	34.2%
	R/C	88	1	22	**100**	0	0	**100**	0	0	311	34.6%
Item Centric (dynamic)	shuffle	0	0	0	0	0	0	0	0	0	0	0.0%
	$1/C$	81	1	**33**	99	0	0	**100**	0	0	314	34.9%
	$1/R$	77	2	29	99	0	0	**100**	0	0	307	34.1%
	R/C	85	1	27	100	0	0	**100**	0	0	313	34.8%
Bin Centric (dynamic)	shuffle	2	0	0	0	0	0	0	0	0	2	0.2%
	$1/C$	89	2	22	**100**	0	0	**100**	0	0	313	34.8%
	$1/R$	87	2	20	**100**	0	0	**100**	0	0	309	34.3%
	R/C	89	**3**	22	**100**	0	0	**100**	0	0	314	34.9%
Bin Balancing	none	1	0	0	0	0	0	0	0	0	1	0.1%
	static shuffle	3	0	0	0	0	0	0	0	0	3	0.3%
	dynamic shuffle	1	0	0	0	0	0	0	0	0	1	0.1%
	static $1/C$	40	0	15	13	0	0	9	0	0	77	8.6%
	dynamic $1/C$	43	2	20	24	0	0	15	0	0	104	11.6%
	Static $1/R$	38	0	14	23	0	0	14	0	0	89	9.9%
	dynamic $1/R$	47	0	20	22	0	0	12	0	0	101	11.2%
	static R/C	48	0	14	21	0	0	12	0	0	95	10.6%
	dynamic R/C	43	0	13	13	0	0	14	0	0	83	9.2%
Single Bin Balancing	none	0	0	0	0	0	0	0	0	0	0	0.0%
	static shuffle	2	0	0	1	0	0	0	0	0	3	0.3%
	dynamic shuffle	1	0	0	0	0	0	0	0	0	1	0.1%
	static $1/C$	71	0	15	97	0	0	**100**	0	0	283	31.4%
	dynamic $1/C$	68	0	17	93	0	0	**100**	0	0	278	30.9%
	Static $1/R$	73	1	13	97	0	0	**100**	0	0	284	31.6%
	dynamic $1/R$	66	0	20	94	0	0	**100**	0	0	280	31.1%
	static R/C	79	0	15	94	0	0	**100**	0	0	288	32.0%
	dynamic R/C	80	0	10	96	0	0	**100**	0	0	286	31.8%
Dot Product	non weighted	**91**	2	31	**100**	0	0	**100**	**38**	0	**362**	**40.2%**
	$1/C^2$	88	2	31	**100**	0	0	**100**	14	0	335	37.2%
	$1/(s \times C)$	17	0	11	11	0	0	43	0	0	82	9.1%

Table 6.3: Results of all heuristics on instances of class 1 (uniform)

	Instances										
	#bins	10	10	10	30	30	30	100	100	100	
	#resources	2	5	10	2	5	10	2	5	10	
	Avg #items	34	36	36	102	109	109	342	363	364	Results
Heuristic	Sorting	#feasible									total / %feasible
Static	none	18	0	0	0	0	0	1	0	0	19 / 2.1%
	shuffle	20	0	0	1	0	0	0	0	0	21 / 2.3%
	$1/C$	67	5	36	40	0	0	50	0	0	198 / 22.0%
	$1/R$	68	6	32	40	0	0	47	0	0	193 / 21.4%
	R/C	69	3	29	65	0	0	**98**	0	0	264 / 29.3%
Item Centric (dynamic)	shuffle	9	0	0	0	0	0	0	0	0	9 / 1.0%
	$1/C$	52	7	35	13	0	0	5	0	0	112 / 12.4%
	$1/R$	50	3	32	6	0	0	0	0	0	91 / 10.1%
	R/C	30	0	30	7	0	0	22	0	0	89 / 9.9%
Bin Centric (dynamic)	shuffle	8	0	0	0	0	0	0	0	0	8 / 0.9%
	$1/C$	76	6	**41**	**79**	0	0	97	0	0	**299** / **33.2%**
	$1/R$	**80**	**9**	33	73	0	0	**98**	0	0	293 / 32.6%
	R/C	68	3	31	70	0	0	97	0	0	269 / 29.9%
Bin Balancing	none	9	0	0	0	0	0	0	0	0	9 / 1.0%
	static shuffle	12	0	0	0	0	0	0	0	0	12 / 1.3%
	dynamic shuffle	7	0	0	0	0	0	0	0	0	7 / 0.8%
	static $1/C$	45	1	24	12	0	0	0	0	0	82 / 9.1%
	dynamic $1/C$	53	0	26	10	0	0	5	0	0	94 / 10.4%
	Static $1/R$	50	1	19	9	0	0	1	0	0	80 / 8.9%
	dynamic $1/R$	58	0	22	15	0	0	0	0	0	95 / 10.6%
	static R/C	42	1	18	0	0	0	0	0	0	61 / 6.8%
	dynamic R/C	35	0	20	2	0	0	0	0	0	57 / 6.3%
Single Bin Balancing	none	16	0	0	1	0	0	1	0	0	18 / 2.0%
	static shuffle	14	0	0	0	0	0	0	0	0	14 / 1.6%
	dynamic shuffle	6	0	0	0	0	0	0	0	0	6 / 0.7%
	static $1/C$	57	0	28	31	0	0	42	0	0	158 / 17.6%
	dynamic $1/C$	59	2	33	12	0	0	11	0	0	117 / 13.0%
	Static $1/R$	58	1	28	30	0	0	42	0	0	159 / 17.7%
	dynamic $1/R$	60	0	31	9	0	0	0	0	0	100 / 11.1%
	static R/C	51	1	19	35	0	0	88	0	0	194 / 21.6%
	dynamic R/C	48	1	18	35	0	0	82	0	0	184 / 20.4%
Dot Product	non weighted	69	5	28	51	**1**	0	90	0	0	244 / 27.1%
	$1/C^2$	37	1	28	25	0	0	71	0	0	162 / 18.0%
	$1/(s \times C)$	57	0	9	20	0	0	10	0	0	96 / 10.7%

Table 6.4: Results of all heuristics on instances of class 2 (uniform with rare resources)

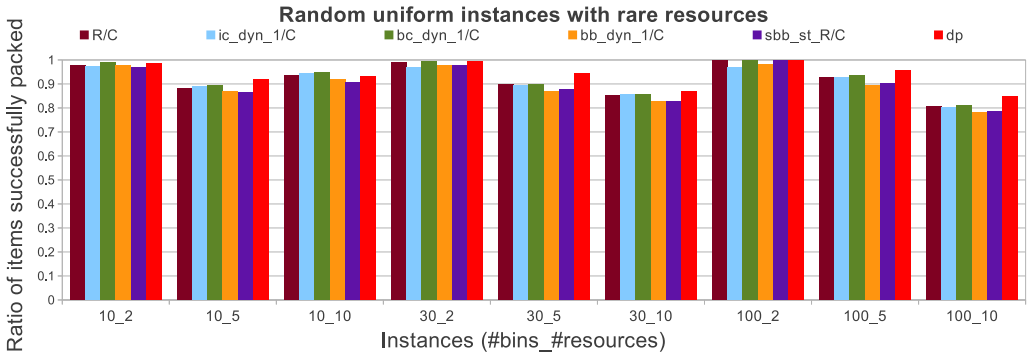


Figure 6.1: Random uniform instances with rare resources, average ratios of items packed

		Instances									Results	
	#bins	10	10	10	30	30	30	100	100	100		
	#resources	2	5	10	2	5	10	2	5	10		
	Avg #items	36	36	36	108	109	109	358	363	364		
Heuristic	Sorting	#feasible									total	%feasible
Static	none	31	0	0	48	0	0	95	0	0	174	19.3%
	shuffle	27	0	0	52	0	0	91	0	0	170	18.9%
	$1/C$	95	3	0	**100**	5	0	**100**	38	0	341	37.9%
	$1/R$	95	2	0	**100**	2	0	**100**	40	0	339	37.7%
	R/C	95	5	0	**100**	2	0	**100**	38	0	340	37.8%
Item Centric (dynamic)	shuffle	4	0	0	1	0	0	0	0	0	5	0.6%
	$1/C$	95	3	0	**100**	4	0	**100**	37	0	339	37.7%
	$1/R$	94	3	0	**100**	3	0	**100**	28	0	328	36.4%
	R/C	94	6	0	**100**	7	0	**100**	47	0	354	39.3%
Bin Centric (dynamic)	shuffle	11	0	0	8	0	0	5	0	0	24	2.7%
	$1/C$	93	2	0	**100**	3	0	**100**	24	0	322	35.8%
	$1/R$	93	2	0	**100**	1	0	**100**	17	0	313	34.8%
	R/C	97	4	0	**100**	2	0	**100**	46	0	349	38.8%
Bin Balancing	none	7	0	0	0	0	0	0	0	0	7	0.8%
	static shuffle	6	0	0	0	0	0	0	0	0	6	0.7%
	dynamic shuffle	4	0	0	0	0	0	0	0	0	4	0.4%
	static $1/C$	66	0	0	83	0	0	98	0	0	247	27.4%
	dynamic $1/C$	74	0	0	83	0	0	99	0	0	256	28.4%
	Static $1/R$	68	0	0	76	0	0	99	0	0	243	27.0%
	dynamic $1/R$	85	1	0	91	0	0	97	0	0	274	30.4%
	static R/C	71	0	0	85	0	0	99	0	0	255	28.3%
	dynamic R/C	70	0	0	80	0	0	98	0	0	248	27.6%
Single Bin Balancing	none	16	0	0	15	0	0	35	0	0	66	7.3%
	static shuffle	16	0	0	10	0	0	34	0	0	60	6.7%
	dynamic shuffle	8	0	0	3	0	0	0	0	0	11	1.2%
	static $1/C$	86	0	0	99	0	0	**100**	1	0	286	31.8%
	dynamic $1/C$	88	1	0	**100**	0	0	**100**	1	0	290	32.2%
	Static $1/R$	87	0	0	97	0	0	**100**	1	0	285	31.7%
	dynamic $1/R$	86	1	0	99	0	0	**100**	0	0	286	31.8%
	static R/C	89	0	0	99	0	0	**100**	0	0	288	32.0%
	dynamic R/C	87	0	0	98	0	0	**100**	0	0	285	31.7%
Dot Product	non weighted	92	**8**	0	**100**	**26**	0	**100**	54	0	380	42.2%
	$1/C^2$	**99**	4	0	**100**	21	0	**100**	**99**	0	**423**	**47.0%**
	$1/(s \times C)$	30	0	0	48	0	0	80	2	0	160	17.8%

Table 6.5: Results of all heuristics on instances of class 3 (correlated capacities)

		Instances										
	#bins	10	10	10	30	30	30	**100**	**100**	**100**		
	#resources	2	5	10	2	5	10	2	5	10		
	Avg #items	33	38	40	99	111	121	333	373	407	Results	
Heuristic	Sorting	#feasible									total	%feasible
Static	none	50	46	22	82	75	60	95	97	88	615	68.3%
	shuffle	50	45	30	86	69	56	98	94	85	613	68.1%
	$1/C$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	$1/R$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	R/C	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
Item Centric (dynamic)	shuffle	17	5	3	4	2	0	0	0	0	31	3.4%
	$1/C$	**100**	99	**96**	**100**	**100**	**100**	**100**	**100**	**100**	**895**	**99.4**%
	$1/R$	**100**	99	**96**	**100**	**100**	**100**	**100**	**100**	**100**	**895**	**99.4**%
	R/C	**100**	99	**96**	**100**	**100**	**100**	**100**	**100**	**100**	**895**	**99.4**%
Bin Centric (dynamic)	shuffle	27	15	13	18	2	2	20	1	0	98	10.9%
	$1/C$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	$1/R$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	R/C	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
Bin Balancing	none	21	7	4	2	0	0	0	0	0	34	3.8%
	static shuffle	19	11	6	2	1	1	0	0	0	40	4.4%
	dynamic shuffle	10	7	4	0	0	0	0	0	0	21	2.3%
	static $1/C$	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**	893	99.2%
	dynamic $1/C$	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**	893	99.2%
	static $1/R$	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**	893	99.2%
	dynamic $1/R$	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**	893	99.2%
	static R/C	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**	893	99.2%
	dynamic R/C	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**	893	99.2%
Single Bin Balancing	none	36	16	14	56	33	20	66	47	31	319	35.4%
	static shuffle	34	18	11	54	23	13	77	57	26	313	34.8%
	dynamic shuffle	16	14	5	9	2	3	3	0	0	52	5.8%
	static $1/C$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	dynamic $1/C$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	static $1/R$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	dynamic $1/R$	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	static R/C	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
	dynamic R/C	**100**	99	95	**100**	**100**	**100**	**100**	**100**	**100**	894	99.3%
Dot Product	non weighted	99	97	93	**100**	**100**	99	**100**	**100**	**100**	888	98.7%
	$1/C^2$	**100**	99	**96**	**100**	**100**	**100**	**100**	**100**	**100**	**895**	**99.4**%
	$1/(s \times C)$	21	7	3	11	2	1	1	1	0	47	5.2%

Table 6.6: Results of all heuristics on instances of class 4 (correlated capacities and requirements)

		Instances										
	#bins	10	10	10	30	30	30	100	100	100		
	#resources	2	5	10	2	5	10	2	5	10		
	Avg #items	39	39	39	119	119	118	397	397	394	Results	
Heuristic	Sorting	#feasible									total	%feasible
Static	none	16	0	0	12	0	0	10	0	0	38	4.2%
	shuffle	10	0	0	1	0	0	13	0	0	24	2.7%
	$1/C$	37	0	0	47	0	0	66	0	0	150	16.7%
	$1/R$	35	0	0	49	0	0	75	0	0	159	17.7%
	R/C	37	0	0	46	0	0	65	0	0	148	16.4%
Item Centric (dynamic)	shuffle	3	0	0	0	0	0	0	0	0	3	0.3%
	$1/C$	31	0	0	37	0	0	55	0	0	123	13.7%
	$1/R$	31	0	0	24	0	0	27	0	0	82	9.1%
	R/C	37	0	0	42	0	0	74	0	0	153	17.0%
Bin Centric (dynamic)	shuffle	11	0	0	5	0	0	1	0	0	17	1.9%
	$1/C$	38	0	0	45	0	0	50	0	0	133	14.8%
	$1/R$	33	0	0	31	0	0	40	0	0	104	11.6%
	R/C	41	0	0	55	0	0	78	0	0	174	19.3%
Bin Balancing	none	2	0	0	0	0	0	0	0	0	2	0.2%
	static shuffle	4	0	0	0	0	0	0	0	0	4	0.4%
	dynamic shuffle	5	0	0	0	0	0	0	0	0	5	0.6%
	static $1/C$	19	0	0	1	0	0	0	0	0	20	2.2%
	dynamic $1/C$	14	0	0	4	0	0	1	0	0	19	2.1%
	Static $1/R$	22	0	0	1	0	0	1	0	0	24	2.7%
	dynamic $1/R$	18	0	0	6	0	0	1	0	0	25	2.8%
	static R/C	24	0	0	5	0	0	1	0	0	30	3.3%
	dynamic R/C	18	0	0	4	0	0	0	0	0	22	2.4%
Single Bin Balancing	none	11	0	0	2	0	0	0	0	0	13	1.4%
	static shuffle	13	0	0	5	0	0	1	0	0	19	2.1%
	dynamic shuffle	8	0	0	2	0	0	0	0	0	10	1.1%
	static $1/C$	33	0	0	44	0	0	67	0	0	144	16.0%
	dynamic $1/C$	30	0	0	36	0	0	59	0	0	125	13.9%
	Static $1/R$	33	0	0	46	0	0	72	0	0	151	16.8%
	dynamic $1/R$	28	0	0	23	0	0	49	0	0	100	11.1%
	static R/C	33	0	0	38	0	0	71	0	0	142	15.8%
	dynamic R/C	32	0	0	43	0	0	68	0	0	143	15.9%
Dot Product	non weighted	**96**	0	11	**100**	0	0	**100**	0	0	307	34.1%
	$1/C^2$	85	0	4	**100**	0	0	**100**	0	0	289	32.1%
	$1/(s \times C)$	95	**76**	**98**	99	**22**	**87**	**100**	**5**	**30**	**612**	**68.0%**

Table 6.7: Results of all heuristics on instances of class 5 (similar items and bins)

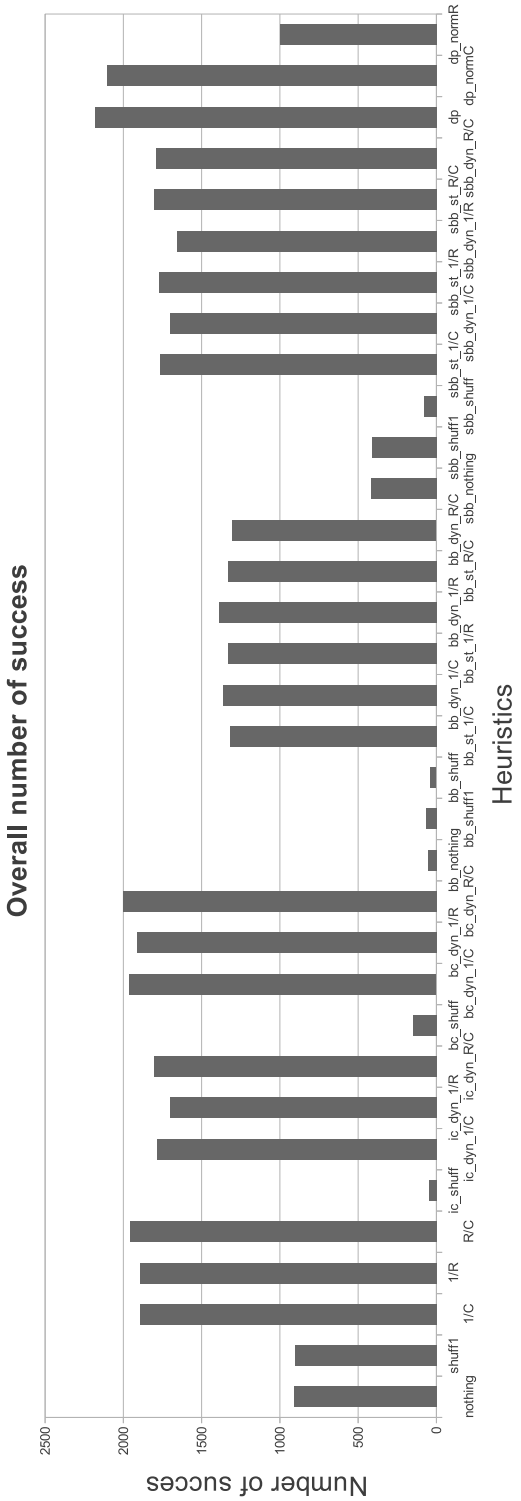


Figure 6.2: Total number of feasible assignments obtained by each of the 34 heuristics, on the 4500 generated instances

In Table 6.3, on random uniform instances, we observe that static best fit, dynamic item centric, dynamic bin centric and dot product heuristics are roughly achieving the same performance. Yet, the non weighted dot product heuristic slightly outperforms other heuristics. This is especially the only heuristic providing feasible solutions with 100 bins and 5 resources. With a rare resource, we observe Table 6.4 that bin centric heuristics are providing slightly better results than other heuristics. Since bin centric heuristics focus on the bins rather than the items, they can make better use of the bins, especially if they first consider bins with null rare resources before considering bins providing rare resources. This helps avoiding to get stuck later on in the algorithm if items with null requirements in the rare resource were packed in bins providing this resource.

Paradoxically, with 10 bins, instances with 10 resources are easier to solve than instances with 5 resources. The reason is that the higher the number of resources, the lower is the probability that an item fits into a different bin other than its initial bin. With very few bins, this actually guides the heuristic.

In Figure 6.1, we give the ratio of items packed in one of the best heuristic of each class for instances with rare resources. Observe that even though heuristics may not provide feasible solutions, 90% of the items are packed on average and the average percentage of items packed depends more on the class of instance than the heuristic used. Yet, we observe significant differences between heuristics results, especially on instances with 10 and 30 bins and 2 resources.

In Table 6.5, we observe that when bin capacities are correlated, heuristics perform very well on instances with few resources while their performances drastically decrease as the number of resources increases. The dot product normalized by bin capacities accounts for these correlations and achieves much better performance than other heuristics.

When both capacities and item sizes are correlated, the problem is almost the same as the single dimensional bin packing decision problem. Table 6.6, we report results for all heuristics on this case. We observe that all heuristics, except random heuristics and the third dot product, are performing very well and achieve an over 99% success rate. For the third dot product, we remark that since items and bins are normalized, all items and bins are roughly the same to this heuristic, resulting in an almost random assignment which explains why these results are close to random assignment results. The harder instances in this case are the ones with few bins and many resources because with more bins it is very likely to have more similar (and exchangeable) items.

On similar instances, Table 6.7, observe that the third dot product heuristic significantly outperforms all other heuristics. This performance is explained by the measure: notice that on the initial configuration (all items remaining and empty bins), the normalized dot product of an item with its initial bin will be close to 1 with high probability and the normalized dot product with other bins will be smaller than 1. Other heuristics are blind to this similarity criterion.

On this benchmark we observe that as the number of resources grows, the problem quickly becomes much harder. Moreover, dynamic item centric, dynamic bin centric and dot product heuristics outperform bin balancing heuristics in terms of number of feasible solutions found. Figure 6.2, we report the total number of success of all heuristics on the benchmark. Observe that the non-weighted dot product heuristic has the highest total number of success but does not significantly outperform other heuristics.

Since all these heuristics are very fast to compute, one can consider applying all of them to

problem instances, as we did for the vector bin packing benchmark. By combining all heuristics, one can expect a slight improvement over the results of the best heuristic for the case considered but above all, one can expect good results without having to carefully analyze properties of the instances (which may be a very difficult and computationally expensive task) in order to chose the best heuristic according to the situation.

6.4 Application to the Machine Reassignment Problem

The machine reassignment problem[3] is a simplified version of problems encountered with data centers: several processes are assigned to different servers, in several data centers, all over the world. The system needs to be robust to energy or machine failures. Moreover, some processes depend on each other and hence have to run on machines which are *close to* each other. Occasionally they consider moving processes to different servers in order to increase system performance. In the machine reassignment problem, system performance is modeled by an aggregated cost and the aim is to minimize it.

The vector bin packing problem with heterogeneous bins is a subproblem of the machine reassignment problem: any feasible assignment for the machine reassignment problem is a feasible VBPHB assignment for the problem defined with items sizes being processes requirements and bins capacities being the machines capacities. Yet, there are some additional constraints in the machine reassignment problem:

- *Conflict constraints*: processes are partitioned into services and two processes of the same service cannot be assigned to the same machine.
- *Transient usage constraints*: when a process is moved from one machine to another, some resources (such as disk space) remain used on the first machine. Thus, the process consumes its requirement of these transient resources on both its initial and final machines.
- *Spread constraints*: Machines are partitioned into locations and each service s needs to have its processes spread over a minimum number of distinct locations, denoted $spreadMin(s)$.
- *Dependency constraints*: Machines are partitioned into neighborhoods and if a service s^a depends on a service s^b, then any process from s^a has to run on some machine having in its neighborhood a machine running a process from s^b.

The goal of the machine reassignment problem is to find a feasible assignment minimizing a weighted cost.

In order to use a diversified multi-start approach, we need to get various, diversified, initial feasible solutions. We only consider feasibility and not solution costs. In this section, we highlight some structural properties of the machine reassignment problem and show how our heuristics can be adapted to this problem and its constraints.

We will use the following notations in the remainder of this section: we denote by $\mathcal{M}$ the set of machines, $\mathcal{N}$ the set of neighborhoods, $\mathcal{P}$ the set of processes, $\mathcal{R}$ the set of resources, $\mathcal{TR} \subseteq \mathcal{R}$ the set of transient resources and $\mathcal{S}$ the set of services. $\mathcal{N}$ is a partition of $\mathcal{M}$ and $\mathcal{S}$ is a partition of $\mathcal{P}$.

The function $N : \mathcal{M} \to \mathcal{N}$ maps each machine to its neighborhood. The function $M : \mathcal{P} \to \mathcal{M}$ is the assignment: it maps each process to its machine. M_0 denotes the initial assignment.

[3] http://challenge.roadef.org/2012/en/

$R(p,r)$ is the requirement of resource $r \in \mathcal{R}$ for the process $p \in \mathcal{P}$. We denote by $C(m,r)$ the capacity of resource $r \in \mathcal{R}$ for the machine $m \in \mathcal{M}$. The two functions $R(r)$ and $C(r)$ are shorthands for $\sum_{p\in\mathcal{P}} R(p,r)$ and $\sum_{m\in\mathcal{M}} C(m,r)$, the overall requirement and capacity on resource r. We denote the initial amount of resource r consumed on machine m by

$$U_0(m,r) = \sum_{\substack{p\in\mathcal{P} \\ s.t.\ M_0(p)=m}} R(p,r).$$

In this problem, we have an initial feasible solution which is used to define transient usage constraints. We will rely on this initial solution to derive properties and set a few process assignments.

In the following subsections, we present several properties of the machine reassignment problem and explain how we can use them to ensure that a feasible solution exists in the search space.

6.4.1 Transient usage constraints

Our heuristics can easily be adapted to integrate transient usage constraints. Indeed we can take them into account as follows: when initial bin capacities are set, let r_1 be a non-transient resource and r_2 be a transient resource. For each machine $m \in \mathcal{M}$, we set its capacity in resource r_1 to $C(m,r_1)$ while we set its capacity in resource r_2 to $C(m,r_2) - U_0(m,r_2)$. Then, process requirements depend on machines: for all $r \in \mathcal{R}\setminus\mathcal{TR}$ and all machines, they are equal to $R(p,r)$, while for all $r \in \mathcal{TR}$ they are equal to 0 for the machine $M_0(p)$ and to $R(p,r)$ otherwise. When a process is assigned to its initial machine, the capacity constraints on transient resources are always satisfied.

These constraints can be taken into account when sizes are computed. We can decide, for instance, that processes with huge requirements on some transient resources will not be moved. Moreover, observe that if a process is moved from its initial machine, then for all of its transient resources, the space used is lost. Hence, we have the following property:

Property 6.2. *For each process $p \in \mathcal{P}$, if there is a transient resource $r \in \mathcal{TR}$ such that $R(p,r) > C(r) - R(r)$, then in every feasible assignment, p has to be assigned to its initial machine.*

Proof. Let p be a process and r a transient resource such that $R(r) > C(r) - R(p,r)$. If process p is moved, since r is transient, a space $R(p,r)$ on machine m cannot be used by any process. Hence, the total available space for all processes in resource r is $C(r) - R(p,r)$, which is smaller than the total requirement. Therefore, any assignment M with $M(p) \neq M_0(p)$ is not feasible. □

Using Property 6.2, we can determine that some processes cannot be moved. In such cases, we can fix them to their initial machines. If we are interested in moving a set of processes P, then we obtain the following corollary:

Corollary 6.1. *Let $P \subseteq \mathcal{P}$ be a subset of processes. If there is a transient resource $r \in \mathcal{TR}$ such that $\sum_{p\in P} R(p,r) > C(r) - R(r)$, then in every feasible assignment, at least one process from P is assigned to its initial machine.*

In a greedy approach, Property 6.2 and Corollary 6.1 can be used with C and R, the *residual* capacities and requirements. Moreover, thanks to these properties, one can fix items or conclude – before being unable to pack an item – that an intermediate solution (a partial assignment) is infeasible. Notice that Property 6.2 and its corollary can also be used for optimization purposes.

6.4.2 Conflict constraints

In order to satisfy conflict constraints, when trying to assign a process p from a service s to a machine m, one just needs to check that there is no process from service s which is already assigned to m.

6.4.3 Spread constraints

A simple way to make sure that these constraints are satisfied is the following: for each service $s \in \mathcal{S}$, take a subset of processes $P \subseteq s$ such that $|P| = spreadMin(s)$, and assign all processes of P to distinct locations. To make sure that there is a feasible solution, we use the initial solution to choose a subset of processes which will be assigned to their initial machines.

6.4.4 Dependency constraints

Dependency constraints are difficult constraints to cope with, because they bound processes to each other and can be cyclic. We propose to take advantage of these constraints to decompose the problem into smaller subproblems where all dependency constraints are satisfied. More precisely, let $g \in \mathcal{N}$ be a neighborhood, $m_1, m_2 \in g$ and $p \in \mathcal{P}$. Remark that if M is a feasible assignment with $M(p) = m_1$, then, setting $M(p) = m_2$ does not violate any dependency constraint. We can even generalize this property to all the processes from any neighborhood into any other neighborhood:

Property 6.3. *Let M be a feasible assignment. Denote by P_n the set of processes assigned to neighborhood $n \in \mathcal{N}$: $P_n = \{p \in \mathcal{P} : M(p) \in n\}$. Any assignment M' such that $\forall n \in \mathcal{N}$, $\forall p_1, p_2 \in P_n$, $N(M'(p_1)) = N(M'(p_2))$, satisfies all dependency constraints.*

Proof. Let $s^a, s^b \in \mathcal{S}$, s^a depends on s^b. Let $p \in s^a$. The assignment M is feasible, hence $\exists p' \in s^b$ such that $M(p') \in N(M(p))$. Moreover $p, p' \in P_{N(M(p))}$. Therefore $p' \in N(M'(p))$. □

Property 6.3 implies that if one takes all processes from a given neighborhood and reassign all of them to a same neighborhood, then the new assignment satisfies all dependency constraints.

We use Property 6.3 with $M = M_0$ to decompose the problem into several subproblems where we either try to find an assignment for all processes from a given neighborhood into itself, or into another. In this latter case, recall that all transient resources used by the processes are lost. Hence, we have to make sure that Corollary 6.1 does not immediately induce that there is no feasible assignment. Moreover, such reassignment also implies that every process will be moved, possibly resulting in huge move costs.

6.4.5 Experiments

In this section, we apply several variants of VBPHB heuristics to machine reassignment problems.

Test problems. We use the 30 instances (sets A, B and X) provided during ROADEF/EURO challenge. They are realistic instances, randomly generated according to real-life Google statistics.

The largest instances contain up to 5,000 machines, 50,000 processes and 12 resources. More details on the instances can be found on the challenge web page[4].

Implemented heuristics. Combining the above ideas to handle the additional constraints, our algorithm proceeds as follows. First, some processes are assigned to their initial machines in order to satisfy the spread constraints. In our experiments, on average 26% of the processes are assigned during this phase. Then, we decompose the problem into smaller independent subproblems. We define a subproblem by selecting all processes initially assigned to a neighborhood and the aim is to find a feasible assignment of these processes into this neighborhood. This makes dependency constraints automatically satisfied by any feasible assignment of the subproblems. We apply our various VBPHB heuristics to each neighborhood. Conflict and transient usage constraints are checked on the fly. Finally, the subproblems assignments are combined to form the global assignment.

We implemented the different types of VBPHB heuristics: item centric, bin centric and bin balancing. For each type, we used several measures, including the static $1/C$, $1/R$ and R/C measures, and the dynamic dot product and process priority measures. We also combined these measures with random orderings. In this case, we report the average results over 50 runs.

We implemented these heuristics in C++ using efficient data structures. Although there is still room for code optimization, we will see below that most heuristics are already very fast.

Results. In order to compare the different heuristics even on instances where they do not find feasible assignments, we report the percentage of assigned processes. A reported 100% means that the heuristic found a feasible solution. Table 6.8 and 6.9 present the results on all instances for each heuristic. The second and the third rows of the tables are describing the sorting used, Rand means random and Prio means priority. The processes (Proc) are the items and the machines (Mach) are the bins. The percent of assigned items is in bold font when the assignment is feasible. If no heuristic finds a feasible assignment, the best percentage of assigned items is green and italicized.

Observe that all heuristics find feasible solutions and assign a high percent of processes in average. Observe also that on instances where feasible solutions are found, the different heuristics are complementary.

Regarding run times, bin balancing variants with static measures are the fastest: they take less than one second to solve all the instances. Bin centric static variants take a few seconds. As expected, the slowest variants are the ones using dynamic measures. In particular, the bin centric dot product heuristic does not scale well to large instances.

In terms of number of feasible solutions found, the best heuristics are the item centric heuristics with priorities on processes and machines ordered randomly or normalized by bins capacities. Observe that all heuristics assign almost all processes on almost all instances. Moreover, even heuristics with the lowest percent of assigned items are useful. For instance, the bin balancing heuristic with normalizations $1/R$ on processes and $1/C$ on machines is the only heuristic which finds a feasible assignment on instance b_3. Notice that the heuristics with the highest percent of assigned processes in average (bin balancing "prio proc–rand mach" and "rand proc–$1/C$ mach") are also the ones with some of the smallest numbers of feasible instances. These two heuristics might however be very useful to provide almost feasible solutions if a repairing algorithm such as a feasibility focused local-search is available.

We remark that for instance a_1_4, since each neighborhood is reduced to one machine, our

[4] http://challenge.roadef.org/2012/files/Roadef-results.pdf

	Item Centric Heuristics				**Bin Centric Heuristics**											
	Prio Proc Rand Mach		Prio Proc 1/*C* Mach		Rand Proc Rand Mach		1/*C* Proc 1/*C* Mach		1/*R* Proc 1/*C* Mach		1/*C* Proc Rand Mach		Rand Proc 1/*C* Mach		Dot Prod Proc 1/*C* Mach	
instance	%	time	%	time	%	time	%	time	%	time	%	time	%	time	%	time
a_1_1	97	0.00	98	0.00	*99*	0.00	88	0.00	88	0.00	89	0.00	98	0.00	88	0.00
a_1_2	92.1	0.02	*92.9*	0.02	91.9	0.00	79.3	0.00	79.3	0.00	79.8	0.00	92.2	0.00	80.1	0.13
a_1_3	*97.5*	0.00	97.2	0.00	96.7	0.00	94	0.00	94.7	0.00	96	0.00	96.5	0.00	95.2	0.01
a_1_4	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00
a_1_5	*99.9*	0.01	97.2	0.01	96.7	0.00	74.4	0.00	78	0.00	76	0.00	94.8	0.00	81.7	0.02
a_2_1	96.4	0.02	96.9	0.02	98.2	0.00	**100**	0.00	**100**	0.00	**100**	0.00	97.9	0.00	**100**	0.73
a_2_2	96.4	0.01	96.8	0.01	96.7	0.00	*98.3*	0.00	*98.3*	0.00	98	0.00	96.8	0.00	97.8	0.01
a_2_3	96.9	0.01	97.1	0.01	97.1	0.00	98.6	0.00	98.7	0.00	98.7	0.00	96.7	0.00	*99.1*	0.01
a_2_4	*97.5*	0.01	96.7	0.01	96.4	0.00	95	0.00	95.3	0.00	92.8	0.00	96.2	0.00	94.4	0.01
a_2_5	94.2	0.01	95.3	0.01	95.1	0.00	89.1	0.00	88.3	0.00	87.7	0.00	95.2	0.00	89.4	0.01
b_1	97.1	0.25	97.5	0.26	97.2	0.01	81.5	0.01	81.5	0.01	82.3	0.01	96.7	0.00	74.7	0.69
b_2	87.8	0.24	86	0.24	85.8	0.01	57.6	0.01	57.1	0.01	57.9	0.01	86.6	0.01	57.5	0.80
b_3	99.9	3.26	99.9	3.43	99.9	0.02	99.4	0.02	99.4	0.03	99.7	0.02	99.9	0.02	96.2	13.67
b_4	**100**	1.37	99.9	1.39	99.9	0.06	96.8	0.09	97.4	0.09	96.6	0.09	**100**	0.06	89.3	36.21
b_5	**100**	14.64	**100**	16.00	**100**	0.05	**100**	0.05	**100**	0.05	**100**	0.05	**100**	0.05	98.4	65.07
b_6	**100**	8.54	**100**	8.77	**100**	0.07	73.3	0.11	71.1	0.12	71.6	0.12	**100**	0.07	72.3	128.21
b_7	**100**	9.17	**100**	9.41	**100**	0.92	**100**	1.47	**100**	1.45	**100**	1.28	**100**	0.90	**100**	1886.06
b_8	99.8	15.38	**100**	17.73	99.9	0.05	**100**	0.07	**100**	0.07	**100**	0.06	99.9	0.05	99.9	61.17
b_9	97.8	6.63	97.7	6.87	98.6	0.31	80	0.50	79.5	0.51	83.1	0.50	98.5	0.31	86.8	397.66
b_10	**100**	7.33	**100**	7.54	**100**	0.92	**100**	1.56	**100**	1.55	**100**	1.32	**100**	0.91	**100**	1368.69
x_1	*97.2*	0.25	97.1	0.25	96.8	0.01	80	0.01	80.8	0.01	81.1	0.01	96.7	0.00	79.5	0.63
x_2	86.1	0.24	85.7	0.24	85.9	0.01	58.6	0.01	58.4	0.01	58	0.01	86	0.01	56.5	0.84
x_3	99.9	3.24	99.9	3.43	99.9	0.02	99.1	0.02	98.8	0.03	99.5	0.02	99.9	0.02	90.7	14.79
x_4	**100**	1.15	**100**	1.18	**100**	0.06	96.2	0.09	95.2	0.09	98.9	0.08	**100**	0.05	89.2	29.94
x_5	**100**	14.53	**100**	15.91	**100**	0.05	**100**	0.05	**100**	0.05	**100**	0.05	**100**	0.05	96.3	66.74
x_6	**100**	8.27	**100**	8.48	**100**	0.07	70.6	0.11	69.1	0.12	70.8	0.12	**100**	0.07	71.9	124.91
x_7	**100**	9.63	**100**	9.87	**100**	0.95	**100**	1.52	**100**	1.52	**100**	1.33	**100**	0.95	**100**	2012.85
x_8	**100**	15.44	**100**	17.83	**100**	0.05	**100**	0.06	**100**	0.06	**100**	0.06	**100**	0.05	98.1	58.98
x_9	98.2	6.46	98.6	6.68	99	0.31	89.6	0.51	89.6	0.51	91.7	0.49	99.1	0.31	91.2	381.08
x_10	**100**	7.12	**100**	7.33	**100**	0.92	**100**	1.53	**100**	1.50	**100**	1.31	**100**	0.90	**100**	1310.05
avg/sum	97.7	133.2	97.7	142.9	97.7	4.8	90	7.81	90	7.77	90.3	6.95	97.6	4.81	89.1	7960.00
#feasible	**12**		**12**		11		10		10		10		**12**		6	

Table 6.8: Results of bin centric heuristics using different measures, on ROADEF/EURO challenge machine reassignment instances. For each variant, the percentage of processes successfully assigned (column "%") and the CPU time (in seconds) are reported.

neighborhood decomposition makes all the heuristics find the initial solution.

If we combine the different heuristics, a feasible assignment is found on 16/30 instances. Out of the 16 feasible instances, 2 are from the instance set A and 8 from each of the sets B and X. We observe that our heuristics are likely to find solutions when R(r)/C(r) is below 85% on average.

It was claimed by the challenge organizers that instances from the set X were generated similarly to instances from the set B. More precisely, that for a given i, instances B_i and X_i are similar. Observe that our heuristics have very similar results on instances with same indices in sets B and X, confirming that these instances have indeed the same structures.

	Bin Balancing Heuristics													
	1/*C* Proc 1/*C* Mach		1/*R* Proc 1/*C* Mach		1/*C* Proc Rand Mach		Rand Proc Rand Mach		Prio Proc 1/*C* Mach		Prio Proc Rand Mach		Rand Proc 1/*C* Mach	
instance	%	time	%	time	%	time	%	time	%	time	%	time	%	time
a_1_1	97	0.00	95	0.00	91	0.00	*99*	0.00	*99*	0.00	*99*	0.00	*99*	0.00
a_1_2	76.6	0.00	77.3	0.00	77.5	0.00	89.8	0.00	88.6	0.02	90.2	0.02	89.8	0.00
a_1_3	94.5	0.00	93.7	0.00	95	0.00	95.8	0.00	96.2	0.00	96.1	0.00	95.9	0.00
a_1_4	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00	**100**	0.00
a_1_5	81.8	0.00	85.4	0.00	77.7	0.00	96.5	0.00	95.7	0.01	96.3	0.01	96.4	0.00
a_2_1	62.7	0.00	61.5	0.00	61.5	0.00	98.6	0.00	98.6	0.02	98.7	0.02	98.7	0.00
a_2_2	96	0.00	97.3	0.00	97.1	0.00	96	0.00	95.6	0.01	96.1	0.01	96	0.00
a_2_3	97.2	0.00	97.2	0.00	97.4	0.00	96.6	0.00	95.9	0.01	96.5	0.01	96.5	0.00
a_2_4	93.8	0.00	88.1	0.00	90.9	0.00	96.2	0.00	96.1	0.01	96.5	0.01	96.2	0.00
a_2_5	84.7	0.00	85.6	0.00	87.2	0.00	95.2	0.00	*95.9*	0.01	95.3	0.01	95.3	0.00
b_1	82.8	0.00	82.5	0.00	83.4	0.00	96.8	0.00	*97.6*	0.25	97.3	0.25	96.8	0.00
b_2	58.5	0.00	59.8	0.00	60.9	0.00	92.9	0.00	90	0.24	92	0.24	*93.1*	0.00
b_3	99.8	0.02	**100**	0.02	99.5	0.02	99.8	0.01	99.9	3.37	99.8	3.44	99.8	0.01
b_4	92.6	0.02	94.7	0.02	94	0.02	**100**	0.01	**100**	1.31	**100**	1.32	**100**	0.01
b_5	**100**	0.03	**100**	0.03	**100**	0.03	99.9	0.03	99.9	15.05	99.9	16.06	99.9	0.03
b_6	**100**	0.02	**100**	0.02	**100**	0.02	**100**	0.02	**100**	8.66	**100**	8.71	**100**	0.02
b_7	**100**	0.03	99.9	0.03	99.9	0.03	99.7	0.03	99.7	8.57	99.7	8.65	99.7	0.03
b_8	**100**	0.04	**100**	0.04	**100**	0.04	**100**	0.03	**100**	16.02	99.9	17.94	**100**	0.03
b_9	72.6	0.27	72.4	0.27	73.9	0.27	99.9	0.03	99.9	6.42	*99.9*	6.45	99.9	0.02
b_10	**100**	0.02	**100**	0.03	**100**	0.03	99.9	0.03	99.9	6.38	99.9	6.41	99.9	0.03
x_1	82.1	0.00	84.2	0.00	82	0.00	96.6	0.00	95.8	0.25	96.6	0.25	96.6	0.00
x_2	62.8	0.00	60.8	0.00	59.8	0.00	92.4	0.00	*94.3*	0.24	92.8	0.24	92.9	0.00
x_3	99.7	0.02	**100**	0.02	99.4	0.02	99.8	0.01	99.8	3.34	99.8	3.43	99.8	0.01
x_4	95.2	0.02	92.3	0.02	95.9	0.02	**100**	0.01	**100**	1.10	**100**	1.10	**100**	0.01
x_5	**100**	0.03	**100**	0.03	**100**	0.03	99.9	0.03	99.9	14.95	99.9	15.98	99.9	0.03
x_6	**100**	0.02	**100**	0.02	**100**	0.02	**100**	0.02	**100**	8.39	**100**	8.43	**100**	0.02
x_7	99.7	0.04	99.8	0.04	99.6	0.04	99.6	0.04	99.6	9.03	99.6	9.08	99.6	0.03
x_8	**100**	0.04	**100**	0.04	**100**	0.04	**100**	0.03	99.9	16.06	99.9	18.08	**100**	0.03
x_9	73.1	0.25	74.7	0.24	77.6	0.23	99.9	0.03	99.9	6.26	*99.9*	6.29	*99.9*	0.02
x_10	**100**	0.02	**100**	0.02	**100**	0.03	99.9	0.03	99.9	6.22	99.9	6.25	99.9	0.03
avg/sum	90.1	0.92	90.1	0.92	90	0.93	98	0.41	97.9	132.19	*98.1*	138.6	*98.1*	*0.38*
#feasible	10		11		9		7		6		5		7	

Table 6.9: Results of bin balancing heuristics using different measures, on ROADEF/EURO challenge machine reassignment instances. For each variant, the percentage of processes successfully assigned (column "%") and the CPU time (in seconds) are reported.

6.5 Conclusion

We introduced the vector bin packing problem with heterogeneous bins, a generalization of the vector bin packing problem which accounts for many real-life problems. We proposed a family of heuristics for the VBPHB, including adaptation of the well-known first fit and best fit bin heuristics, and some new variants taking advantage of the multidimensional resources and variable bin sizes. These heuristics are flexible and easy to implement. Thanks to their efficiency, we can combine all of them and apply them all on any instance, getting the best of each heuristic without having to analyze instances to pick the heuristic which is the most likely to be successful.

Even though the heuristics were designed for the vector bin packing problem with hetero-

geneous bins, they can also be used to solve vector bin packing problems. By combining our heuristics with a simple lower bounding procedure, we were able to solve and prove optimality on 54 out of the 77 instances left open in vector bin packing benchmark from Brandao and Pedroso (2013).

We analyzed the machine reassignment problem and presented some of its properties. Based on these properties, we adapted our heuristics and we were able to generate feasible solutions which can be used as starting points for optimization algorithms.

In future works, one can experiment more sophisticated measures, possibly based on a relaxation of this problem, or the *permutation pack* and *choose pack* heuristics from Leinberger et al. (1999).

When considering greedy or constructive assignment based approaches, one can also reason on partial solutions and infer that some items have to be packed in a subset of the remaining bins. We can use constraint programming to implement such an approach: propagate decisions taken by the heuristic, then take next decisions using updated domains.

The source code related to this chapter and our algorithms for ROADEF Challenge are open-source and freely available[5,6].

[5] https://github.com/mgabay/ROADEF2012-J19

[6] https://github.com/mgabay/Variable-Size-Vector-Bin-Packing

Chapter 7

A Reduction Algorithm for Packing Problems

Résumé : Nous proposons un algorithme de réduction très général et applicable à de très nombreux problèmes de placement d'objets. Nous exposons une propriété de dominance et nous proposons un algorithme, de complexité polynomiale en le nombre de récipients et la taille de l'instance, trouvant des ensembles non triviaux permettant d'appliquer cette dominance. Ces réductions peuvent s'intégrer très simplement dans des approches de résolution dont les décisions sont les affectations des objets. La réduction est également applicable, avec une faible complexité, à des instances high-multiplicity. Des expériences préliminaires montrent la force de cette réduction dans le cadre d'un algorithme de résolution pour le problème de vector bin packing hétérogène.

Abstract: In this chapter, we present a reduction algorithm for packing problems[1]. This reduction is very generic and can be applied to almost any packing problem such as bin packing, multi-dimensional bin packing, vector bin packing (with or without heterogeneous bins), etc. It is based on a dominance applied in the compatibility graph[2] of a partial solution and can be computed in polynomial time in the input size and the number of bins, even on instances with high-multiplicity encoding of the input.

7.1 Introduction

We are interested in combinatorial packing problems in general. There are usually two ways to solve such problems. The first one is to focus on the assignment of the items: one has to decide in which bin each item will be packed. The second one is focused on patterns: given all

[1] The results presented in this chapter are joint work with Hadrien Cambazard and Yohann Benchetrit. They were presented in conference Gabay et al. (2014a).

[2] We expect the reader to be familiar with basic graph theory, matching and network flows. If not, we refer the reader to chapters 1 and 2 from Lovász and Plummer (1986) which cover all the concepts used in this chapter.

the feasible packings of items in bins, which patterns are used in an optimum solution and how many times? This second approach is the generalization of Gilmore and Gomory (1961) approach for the cutting-stock problem. Pattern based exact algorithms are usually more efficient when the number of patterns is limited or there are items with large multiplicities while assignment based algorithms are usually better for heuristics and when the number of different items is large.

In this chapter, we focus on assignment based approaches and propose a reduction algorithm based on a dominance property. In an assignment based solver for a packing problem, this property can be used to fix the assignment of many items at once and reduce the problem to a smaller packing problem. This is especially well suited to be integrated in branch-and-bound approaches for packing problems.

In packing problems, Dual Feasible Functions have been extensively used to obtain lower bounds (Alves et al. 2014, Clautiaux et al. 2010). In this chapter, we are interested in reduction procedures which ensure that optimality is preserved in the reduced problem. Such reductions have already been proposed by authors on specific packing problems, see e.g. Carlier et al. (2007), Huegler and Hartman (2002), Martello and Toth (1990b). Carlier et al. (2007) introduced the notion of Identically Feasible Function (IFF) to properly state the idea of reductions. The authors then propose IFF for 2-dimensional bin packing problems to remove small items and increase the size of large items. In Khanafer et al. (2012), they investigate the use of tree-decomposition techniques to identify subproblems to be solved independently in a 2-dimensional packing problem with conflicts. Notice that reductions procedures or problem separation techniques were proposed very early by Martello and Toth (1990b).

Reduction algorithms are often critical for the success of packing algorithms, see e.g. Kellerer et al. (2004), Martello and Toth (1990a) for presentation of reduction algorithms on the knapsack and the bin packing problems. In multi-dimension or with additional constraints, it is even more critical to be able to reduce packing problems to the smallest possible core problems. In this chapter, we present a reduction algorithm which can be used on whole instances of packing problems and can also be applied in the course of an assignment based algorithm for packing problems.

7.2 Definitions

We define a *pure packing problem* as a packing problem in which the capacities of the bins and the weights of the items are non-negative and the only constraints are the capacity constraints. The results presented in this chapter are however much more general than this case. A number of other constraints can be added to the problem and if we simply take them into account when we compute the compatibility graph, then all the results for pure packing problems will still hold. For instance, we can add the following types of constraints:

- Variable item weights: if the weights of an item depends on the bin it is assigned to.
- Conflict constraints between items: if there are sets of conflicting items such that two items in a same set cannot be in a same bin.
- Incompatibility constraints between items and bins: if some items are incompatible with some bins, for instance because the items are fragile or heavy.

Basically, the results can be generalized to most constraints which do not involve both set of bins and set of items (for instance, spread or dependency constraints involve both set of bins and set of items).

A partial solution of a packing problem is a solution in which some but not all the items have been packed. Given a partial solution of a packing problem, observe that packing the remaining items is an instance of the same packing problem but in which bins have variable sizes. Pure packing problems have the nice property that for any feasible assignment, any subset of the bins and of the items in these bins is a partial solution of this problem and a feasible solution to the packing problem defined by only these bins and items. So if a partial solution is not feasible, then there is no feasible solution of the whole instance having this partial solution as a subset.

This is not necessarily true for "non-pure" packing problems such as the Machine Reassignment Problem (see Chapter 6 for an extended description of this problem). In this problem, we also have to spread the items: a subset of a feasible solution may violate the spread constraint. However, the partial solution property holds for the underlying vector bin packing problem with heterogeneous bins.

We say that a partial solution P is feasible if it is a feasible solution to the pure packing problem defined with only the bins and items appearing in P. We say that it is *g-feasible* (g stands for globally) if P is a subset of a feasible solution to the underlying pure packing problem. Given a bin b (resp. a subset of bins X) we denote by b_P (resp. X_P) the set of items contained within this bin (resp. this subset of bins) in partial solution P.

Given a feasible partial solution, in the remaining subproblem, not all items fit into all bins. Let b be a bin, we say that an item $i \notin b_P$ is *compatible with* or *fits into* the bin b if the set of items $b_P \cup \{i\}$ fits into the bin b (starting from P, packing i into b gives another feasible partial solution).

Let $\mathcal{I}$ be the set of items and $\mathcal{B}$ be the set of bins. In a partial solution P, we denote by $\mathcal{I}_P$ the set of unpacked items and by $\mathcal{B}_P$ the set of bins in which at least one item from $\mathcal{I}_P$ fits. Let $X \subseteq \mathcal{B}_P$ be a subset of bins, we denote by $\Gamma(X) = \{i \in \mathcal{I}_P : \exists b \in X \text{ s.t. } i \text{ fits into } b\}$, the set of items which are compatible with these bins.

We have the following property:

Property 7.1. *Given a partial solution (possibly with no item assigned) P, for any $X \subseteq \mathcal{B}_P$ such that there is a feasible packing of $\Gamma(X)$ into X, let P_X be a partial feasible solution extending P and in which all items from $\Gamma(X)$ are assigned to X. P is g-feasible if and only if P_X is g-feasible.*

Proof. If P_X is g-feasible, since P is a subset of P_X, P is obviously g-feasible.
Suppose P is g-feasible and consider a feasible solution S (S is a solution to the complete pure packing problem) which is an extension of P. Notice that bins from X contain only the items which they were assigned in P and some items from $\Gamma(X)$. If we remove the items in $\Gamma(X)$ from S, we obtain a partial solution P' which is clearly g-feasible. Moreover, in P', the bins from X are assigned exactly the same items as in P. So packing $\Gamma(X)$ into X is feasible in P'. We pack these items as in P_X and obtain a feasible solution S' containing P_X. Therefore, P_X is g-feasible. □

Property 7.1 gives a decomposition of packing problems into subproblems. It states that if we can find a subset of bins such that there is a feasible assignment of their compatible items within these bins, then we can pack these items in the bins and remove both the bins and items from further considerations. A valid set X is illustrated Figure 7.1, page 120.

Hence, if we can find such a set of bins and a feasible packing, we can assign all of them at once and reduce the problem to a smaller subproblem. However, given a set, determining whether it verifies the property is a hard matter. For instance, for $X = \mathcal{B}$ and $P = \emptyset$ we have the initial packing problem and even the (single dimension) bin packing problem is strongly $\mathcal{NP}$-hard and does not have a PTAS (unless $\mathcal{P} = \mathcal{NP}$ of course).

We propose a reduction algorithm based on flow-computation in the graph of items and bins compatibilities. Given a partial or an empty solution, it finds a feasible set X and a feasible assignment of the items from $\Gamma(X)$ into X. In Section 7.5, we present the reduction algorithm with single item assignments. It also proves that any packing problem in which the number of bins is greater than or equal to the number of items can be reduced in polynomial time to a packing problem with fewer bins than items. In Section 7.6, by using network flows, we generalize the results of Section 7.5 to more complex assignments. We present preliminary results and additional definitions in Section 7.4. In the next section, we expose a quick discussion explaining complexity matters in these problems and why the results presented in this chapter are holding for both decision and optimization packing problems.

7.3 Discussion

We have not stated yet whether we are considering decision or optimization problems and which are the complexity parameters. In this section, we seek to clarify these matters.

7.3.1 Decision or Optimization ?

The problem description may let the reader think that we are focusing on decision problems. In decision problems, the number of bins is given. So checking bins' and items' compatibilities is straightforward. Moreover, we do not have to consider adding or removing bins: $\mathcal{B}$ is known in advance. In our case, we are considering both decision and optimization problems. In optimization problems, in any assignment based algorithm, if an item does not fit in any bin, then we have to add a new bin (the partial solution is not g-feasible with the current number of bins). So we can modify the algorithm to immediately add a new bin and extend $\mathcal{B}$ with it once an item is compatible with none of the bins. The results of the dominance properties and the reductions applied are not modified since adding a bin neither adds nor removes any edge from the other bins. One can argue that there are maybe several types of bins but, in this case, in exact algorithms we "only" have to branch to select which kind of bin is added. The algorithm can be adapted by simply branching earlier on the new bins. Moreover, in order to minimize the number of bins, many algorithms repeatedly solve the problem with a fixed number of bin. In this case, we are actually solving several decision problems.

In the following, we assume that the number of bins is fixed and equal to $|\mathcal{B}|$ but the results can easily be generalized to optimization problems.

7.3.2 Complexity

Regarding complexity, the algorithm is polynomial in the input size and in $|\mathcal{B}|$. If we have a high-multiplicity encoding of the input, it is legitimate to expect a high-multiplicity encoding of the output. For instance an output can be specified by giving the patterns used (each pattern being a way to fill a bin) and for each pattern how many items of each type are contained in the bin and how many times the pattern is used. Such an encoding is compact but not necessarily polynomial in the input size. Indeed, $|\mathcal{B}|$ is not necessarily polynomial in the input size when a high-multiplicity encoding is used. However, we are interested in algorithms here (and in solving $\mathcal{NP}$-hard problems…) and, above all, we are considering *assignment based algorithms*. Meaning that $|\mathcal{B}|$ is small enough to consider such approaches. Finally, observe that even assignment based algorithms can assign many items at once.

In the end, the algorithm may not be polynomial in the input size but it is polynomial in the size of the space which the programmer decided to allocate to his algorithm.

In the following, we assume that given a partial solution P, an item i and a bin b, we have access to an oracle which tells us whether i fits into b. However, we remark that for hyper-boxes geometric packing problems (strip packing, multi-dimensional bin packing,...) this problem is $\mathcal{NP}$-hard. Yet, for algorithms which do not consider reorganizing the contents of the bins in partial solutions, we can usually find out in polynomial time whether an item fits into a given bin.

7.4 Preliminaries

In the following, we use the bipartite compatibility graph of partial assignments. In this graph, each unpacked item is a vertex in the first partition and each bin in which at least one item fits is a vertex in the second partition. We denote this graph by $G_P = (\mathcal{I}_P, \mathcal{B}_P, E_P)$; the set of vertices is denoted by $V_P = \mathcal{I}_P \cup \mathcal{B}_P$. Let $u \in \mathcal{I}_P$ and $v \in \mathcal{B}_P$, the edge $\{u, v\}$ is in E_P if the item u fits into the bin v. Figure 7.1 illustrates the compatibility graph of a vector packing problem with two dimensions and also gives a set X satisfying Property 7.1. We remark that if we use constraint programming, the compatibility graph is obtained directly from the variables domains after filtering.

In a graph $G = (V, E)$, we denote by $\delta(u) = \{\{u, v\} : v \in V \text{ and } \{u, v\} \in E\}$ the set of edges incident to $u \in V$; we denote by $d(u) = |\delta(u)|$ the degree of a vertex $u \in V$ and by $\Gamma(X) = \{v \in V \setminus X : \exists u \in X \text{ s.t. } \{u, v\} \in E\}$ the neighbors of a set of vertices $X \subseteq V$. For singletons, we use $\Gamma(v)$ instead of $\Gamma(\{v\})$. In the following we will also use directed graphs but d, δ and Γ *always* refer to the (underlying) undirected graph.

In the next section, we use Hall's theorem which is recalled below:

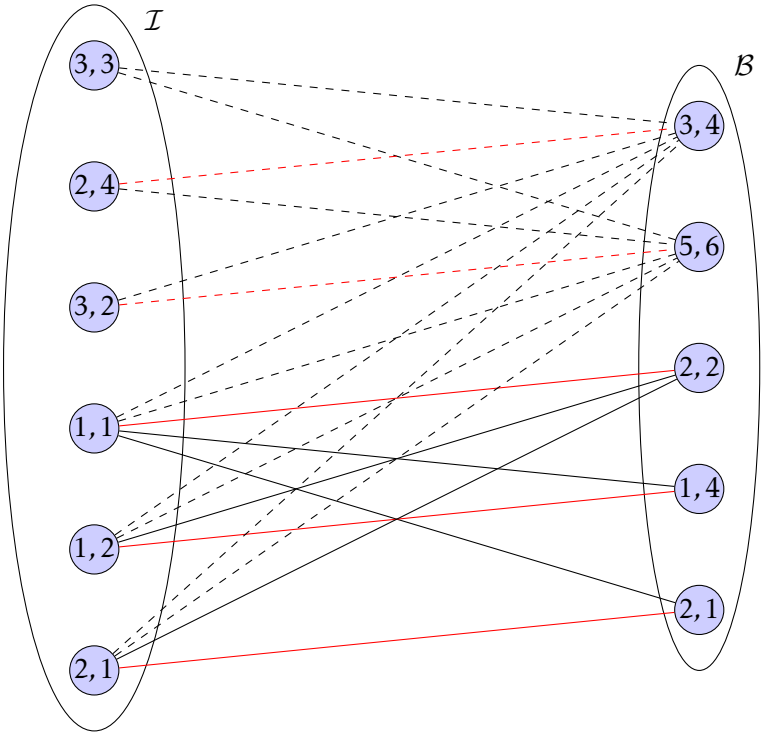
Theorem 7.1 (Hall's Theorem)**.** *A bipartite graph $G = (U, V, E)$ has a matching saturating U if and only if $\forall X \subseteq U$, $|X| \leq |\Gamma(X)|$.*

Now, let us consider the compatibility graph G_P for a given P and derive some basic properties from this graph. Let $u \in \mathcal{I}_P$, if $d(u) = 0$ then the item u cannot be packed. Hence, the partial solution P is g-infeasible. If $d(u) = 1$, then $\delta(u) = \{\{u, v\}\}$ for some $v \in \mathcal{B}_P$ and if the number of bins cannot be increased, the item u has to be packed into the bin v.

We said that we are only interested in bins in which at least one of the remaining item fits. Notice that if we added the other bins to the graph their degrees would be 0 so they would be isolated vertices which are modeling nothing useful. For a bin $v \in \mathcal{B}_P$, if $d(v) = 1$ then only one item $u \in \mathcal{I}_P$ can be assigned to the bin v. In any pure packing problem, it is dominant to pack this item into v. So we can assign item u to v and remove u and v from the compatibility graph.

Observe that, when an item is assigned to a bin, we can update the compatibility graph and ensure that all degrees are greater than or equal to 2 in linear time in the size of $\mathcal{I}_P$. When the graph is updated, we only remove edges and vertices but never add others. In the following, we do not assume that all degrees are greater than 1 since it can result in loss of generality for optimization problems (when an item u is assigned because $d(u) = 1$).

We first present the reduction when there are no multiplicities on the items and at most one item is assigned to a bin. Then, we generalize the reduction algorithm to account for multiplicites on the items and perform reductions with assignments of more than one item into a bin. In this general case, the compatibility graph will be slightly different. We present the changes in Section 7.6.



The labels in the vertices from $\mathcal{I}$ are the requirements of the items in each dimension and the labels in the vertices from $\mathcal{B}$ are the remaining capacities of the bins in each dimension. Edges are denoting compatibilities. The three vertices $\{(2,2),(1,4),(2,1)\}$ from $\mathcal{B}$ form a set X satisfying Property 7.1. Plain edges are edges from $\delta(X)$, other edges are dashed. A maximum matching in this graph is given in red.

Figure 7.1: The bipartite compatibility graph of a vector packing problem.

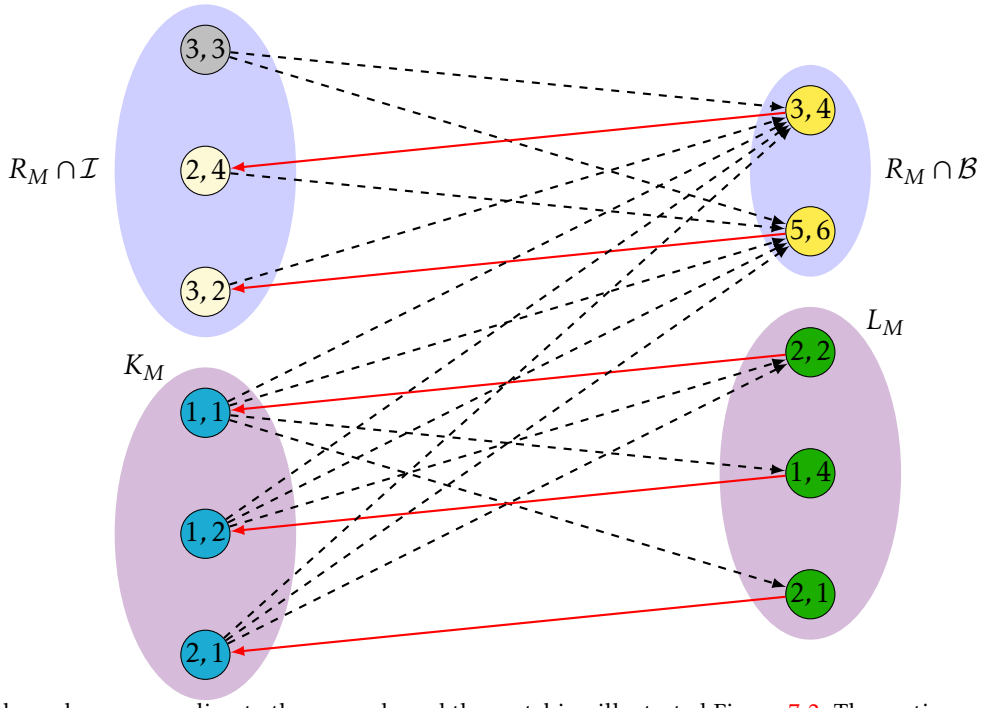
7.5 Matching-based reduction algorithm

In this section, we show that given the compatibility graph we can find a set X satisfying Property 7.1 and which is maximum for one-to-one assignments. A one-to-one assignment is an assignment in which each item is assigned to one bin and at most one item is assigned to a bin.

In order to find this set, we compute a maximum matching $M \subseteq E_P$ in the compatibility graph G_P. We orient the edges from G_P as follows: let $u \in \mathcal{I}_P$ and $v \in \mathcal{B}_P$ two adjacent vertices, i.e. $e = \{u, v\} \in E_P$, if $e \in M$ we orient e from v to u, otherwise e is oriented from u to v. We denote by $D_M = (\mathcal{I}_P, \mathcal{B}_P, A_M)$ the new directed graph. Let U_M be the set of all vertices from $\mathcal{I}_P$ which are not saturated by M, we denote by R_M the set of vertices reachable from U_M in D_M (we have $U_M \subseteq R_M$), and by $K_M = \mathcal{I}_P \setminus R_M$ the items which are unreachable from U_M and $L_M = \mathcal{B}_P \setminus R_M$ the bins which are unreachable from U_M. Figure 7.2 illustrates the orientation and these sets, in the oriented graph obtained from the matching Figure 7.1.
We have the following result:

Theorem 7.2. *$L_M = \mathcal{B}_P \setminus R_M$ is a set satisfying Property 7.1 and the matching M gives a feasible assignment of $\Gamma(L_M)$ into L_M.*

Proof. This proof is based on the proof from König's theorem (König 1931). König's theorem states that in a bipartite graph, the cardinality of a maximum matching is equal to the cardinality



The oriented graph corresponding to the example and the matching illustrated Figure 7.2. The vertices are labeled as in Figure 7.2. The gray vertex $(3,3)$ is the only node in the set U_M. All dashed edges are oriented from left to right and all red edges are edges from the matching and oriented from the right to the left.

Figure 7.2: An oriented compatibility graph

of a minimum vertex cover.

If M saturates $\mathcal{I}_P$, then $U_M = R_M = \emptyset$ and $L_M = \mathcal{B}_P$. Moreover, assigning each item to its corresponding bin in the matching is clearly a feasible assignment of all items.

Otherwise, M does not saturate $\mathcal{I}_P$. By definition of R_M, there is no arc from R_M to L_M. Moreover, there is no arc from L_M to R_M since any vertex from $R_M \cap \mathcal{I}_P$ is either not saturated by M (in U_M) or matched with a vertex in $R_M \cap \mathcal{B}_P$. So $\Gamma(L_M) \subseteq K_M$.

Finally, by definition of U_M and K_M, all vertices in K_M are saturated by M. Moreover, let $u \in \mathcal{I}_P$ and let $e = \{u, v\} \in M$ if $v \in R_M$, then by definition of R_M, $u \in R_M$. Hence, the matching M saturates all vertices of K_M using edges from $\mathcal{B}_P \setminus R_M = L_M$ to K_M. Therefore $K_M \subseteq \Gamma(L_M)$ and hence $\Gamma(L_M) = K_M$. Moreover, assigning each item from K_M to its corresponding bin in the matching is feasible since only one item is assigned to each bin and edges are denoting feasible assignments. □

Based on Theorem 7.2, we can compute a feasible set X of one-to-one assignments using Algorithm 7.1.

Algorithm 7.1: Matching-based reduction algorithm

```
1 Compute a maximum matching M in G_P;
2 Compute R_M;
3 return M, X = B_P \ R_M;                    // The matching gives the assignment
```

The reduction assigns any item in K_M to its bin matched by M. Then, L_M and K_M are taken

out of consideration. We denote by P' the new partial assignment. If K_M was empty, then $P' = P$. We have the following property:

Property 7.2. *The set X returned by Algorithm 7.1 is maximum for one-to-one assignments.*

Proof. Let $Q \in \mathcal{B}_P$ be a feasible set for Property 7.1 with one-to-one assignments. Observe that a one-to-one assignment is an assignment of one item into each bin, so it is a matching. Since Q is a feasible set, there is a one-to-one assignment, hence a matching, of $\Gamma(Q)$ into Q. Let W be a second feasible set for Property 7.1 with one-to-one assignments. Clearly, there is a matching saturating vertices from $\Gamma(W)$ with edges from $\Gamma(W)$ to W. Hence, there is a matching saturating vertices from $\Gamma(W \cup Q)$ with edges from $\Gamma(W \cup Q)$ to $W \cup Q$ and $|\Gamma(W \cup Q)| \leq |W \cup Q|$.

Therefore, in order to show that a feasible set X with one-to-one assignments is maximum, it is sufficient to show that it is maximal with respect to the union. Moreover, observe that for any set $W \in \mathcal{B}_P$, disjoint from Q, if $Q \cup W$ is a feasible set with one-to-one assignments, then there is a matching saturating vertices from $\Gamma(W) \setminus \Gamma(Q)$ with edges from W to $\Gamma(W) \setminus \Gamma(Q)$. Hence, $|W| \geq |\Gamma(W) \setminus \Gamma(Q)|$. Therefore, it is sufficient to show that once the reduction has been applied, in the new compatibility graph there is no set $X \subseteq \mathcal{B}_P$ such that $|X| \geq |\Gamma(X)|$.
Once the reduction has been performed and items from K_M have been assigned to the bins from L_M, we denote by P' the extended partial assignment and we have:

Lemma 7.1. *In $G_{P'}$, $\forall X \in \mathcal{B}_{P'}$, $|X| < |\Gamma(X)|$.*

Proof of Lemma 7.1. Let M' be the restriction of M to $G_{P'}$; M' is a maximum matching of $G_{P'}$. Clearly, M' is not a perfect matching and $L_{M'} = K_{M'} = \emptyset$.

Hence, the vertices of $\mathcal{B}_{P'} = R_{M'} \cap \mathcal{B}_{P'}$ are saturated by M'. By Hall's theorem, $\forall X \subseteq \mathcal{B}_{P'}$, $|X| \leq |\Gamma(X)|$. Suppose there exists $X \in \mathcal{B}_P$ such that $|X| \geq |\Gamma(X)|$, then $|X| = |\Gamma(X)|$ and X is saturated by M'. Therefore, M' is a perfect matching of $G_{P'_{|X \cup \Gamma(X)}}$ (the restriction of $G_{P'}$ to $X \cup \Gamma(X)$). Hence, $\Gamma(X) \cap U_{M'} = \emptyset$, there is no arc from $\mathcal{B}_{P'} \setminus X$ into $\Gamma(X)$ and obviously there is no arc from $\mathcal{I}_{P'} \setminus \Gamma(X)$ into X. Which contradicts $L_{M'} = \emptyset$ and $K_{M'} = \emptyset$. □

By Lemma 7.1, $\mathcal{B}_P \setminus R_M$ is maximal with respect to the union and hence, it is maximum for one-to-one assignments. □

As a consequence of Lemma 7.1, in a single run of the algorithm, we have proceeded to all feasible one-to-one assignments.

The algorithm computes a maximum matching and marks the vertices of the graph in a single pass. Therefore, the overall complexity is the complexity of the maximum matching algorithm which is $\mathcal{O}(|\mathcal{I}|^{2.5})$ if we use Hopcroft-Karp's matching algorithm (Hopcroft and Karp 1973).
A second consequence is the following:

Corollary 7.1. *For pure packing problems in which we can compute the compatibility graph in polynomial time: any instance with more bins than items ($|\mathcal{I}| \leq |\mathcal{B}|$) can be reduced to an instance with more items than bins ($|\mathcal{I}| > |\mathcal{B}|$) in polynomial time.*

On consecutive runs of the algorithm, the efficiency of the maximum matching algorithm can be improved by updating the previously computed maximum matching and using it for a hot start of the matching algorithm.

Observe that for the special case of the one-dimensional bin packing problem finding the set X is trivial. Indeed, an item which fits into a bin fits into all bins with smaller weight. Hence, the

compatibility graph is (doubly) convex. The maximum matching can be obtained by simply sorting the bins and the items, or even in linear time with the algorithm from Steiner and Yeomans (1996).

7.6 Generalized reduction algorithm

In this section, we extend previous results to other assignments than one-to-one assignments and instances which are specified using a high-multiplicity encoding of the input.

First, observe that a matching in a bipartite graph $G = (U, W, E)$ is an integer flow in the same graph extended with a source s connected to all vertices from U and a sink t connected to all vertices from W, and all edges from E being oriented from U to W. We now use this oriented compatibility graph $D = (V, A)$ (resp. $D_P = (V_P, A_P)$) in place of the previous one. We denote by $c(u,v)$ (resp. $c^P(u,v)$) the capacity of the arc (u,v) in the compatibility graph (resp. the compatibility graph of the partial solution P).

In order to generalize the results to high-multiplicity, we need to ensure that the size of the graph is polynomial in the input size and $|\mathcal{B}|$. In order to achieve this, we simply merge vertices from a same item type into a single vertex. Suppose that $\mathcal{I}$ is now the set of different item types and m_i the multiplicity of item $i \in \mathcal{I}$. In a partial solution P, we now consider that $\mathcal{I}_P$ is the set of remaining different item types and m_i^P denote the residual multiplicity of the item i (the number of items of type i which are not already packed in P). For an item $i \in \mathcal{I}$, we set $c(s,i) = m_i$ (resp. $c^P(s,i) = m_i^P$) and $c(i,b) = +\infty \ \forall b \in \mathcal{B}$ with $(i,b) \in A$.

Now, suppose you expand this graph and replicate m_i times each vertex $i \in \mathcal{I}$. Let $b \in \mathcal{B}$ and $N = \{i \in \mathcal{I} : (i,b) \in A\}$ (since Γ is the neighborhood in the undirected graph, this is also $\Gamma(b) \cap \mathcal{I}$). We denote by κ_b the maximum number such that any subsets of items in N of size at most κ_b can be packed in b; $\kappa_b = \max\{k : \forall J \subseteq N \text{ with } |J| = k,\ J \text{ fits into } b\}$. We call κ_b the *robust capacity* of the bin b. The capacity of the arc (b,t) is set to $c(b,t) = \kappa_b$. We denote by $\kappa = (\kappa_1, \ldots, \kappa_{|\mathcal{B}|})$ the vector of robust capacities of the bins. We define κ_b^P and capacities of the arcs similarly for partial solutions. Similarly to one-to-one assignments, we define a κ-assignment as an assignment in which at most κ_b items are assigned to the bin b.

The compatibility graph now looks like the graph Figure 7.3. An example of a compatibility graph is illustrated Figure 7.4.

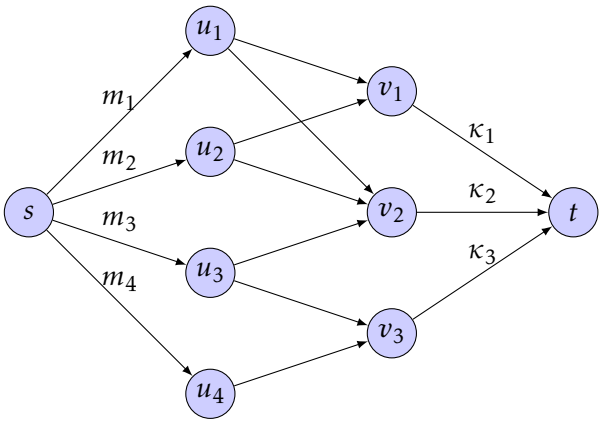


Figure 7.3: A generalized compatibility graph

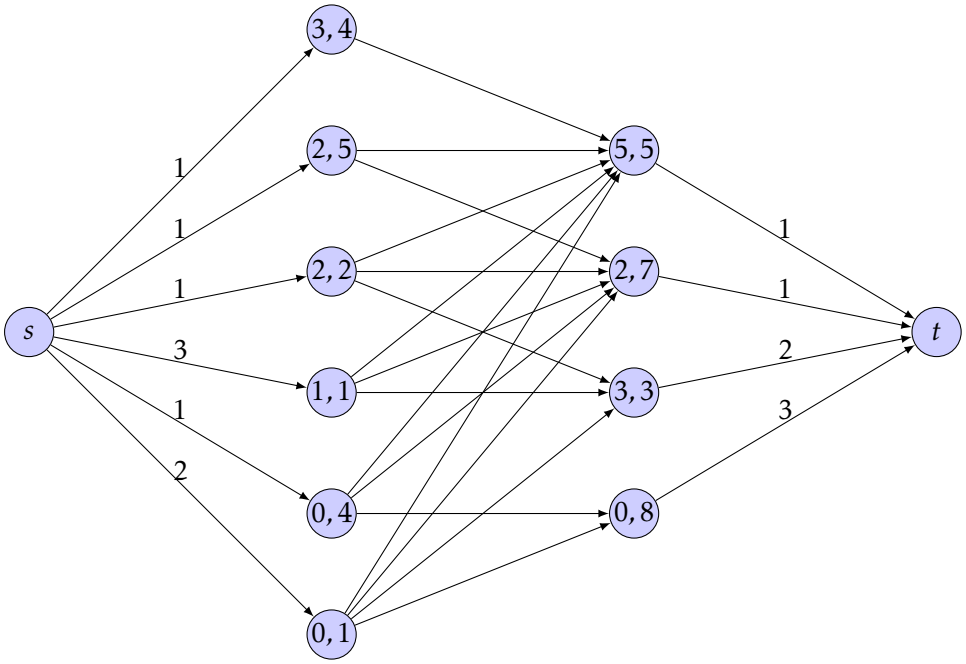


Figure 7.4: An example of a generalized compatibility graph for a 2-dimensional vector packing problem

Observe that determining the values of κ is easy for single dimension bin packing problems: Sort the items by decreasing order of the weights and let r be the remaining space in bin b. Let w_i be the weight of item i and $j = \min\{k : \sum_{i=1}^{k} m_i w_i > r\}$. Then, $\kappa_b = \sum_{i=1}^{j-1} m_i + \max\{k : k \leq m_j \text{ and } k \times w_j + \sum_{i=1}^{j-1} m_i w_i \leq r\}$.

For vector bin packing problems, we compute κ_b by applying the same procedure on all dimensions: $\kappa_b = \min \kappa_b^j$ where κ_b^j is the value of κ_b for the bin packing problem on dimension j.

For other packing problems, this problem can be $\mathcal{NP}$-hard but as we will see, only the diversity of combinations is depending on the values of κ's. So setting them to 1 will simply limit solutions to one-to-one assignments. Additionally, any heuristic giving a lower bound on the values of κ's will yield feasible reductions. So, for instance, in a 2-dimensional bin packing problem, if we create a rectangle whose width is the largest width among the items compatible with b and whose height is the largest height among the items compatible with b; then any heuristic giving a feasible number of such rectangles which can be packed, gives a lower bound on κ_b. And we can set the capacity $c(b, t)$ to this lower bound.

Finally, we can use fixed-parameter tractability results: when the number of items is fixed, for a set of items of the given size, we can usually determine in polynomial time whether the items fit. One can use such results when the number of compatible items is small but the number of combinations of a fixed number of items is quickly impracticable if we consider combinations of more than 3 items and bins with many compatible items.
Algorithm 7.2 gives the generalized reduction algorithm.

The construction used for Algorithm 7.1 is similar to the construction used in a proof from König's theorem (König 1931). König's theorem states that in a bipartite graph, the cardinality of a maximum matching is equal to the cardinality of a minimum vertex cover. Computing R_M as we did in Section 7.5 is the same as computing a minimum vertex cover from a maximum

Algorithm 7.2: Generalized reduction algorithm

1 Compute a maximum flow f in D_P;
2 Compute R_f, the set of all vertices reachable from s in the flow residual graph;
3 **return** f, $X = \mathcal{B}_P \setminus R_f$; `// The flow gives the assignment`

matching. In Algorithm 7.2, we compute the set of reachable vertices in the residual graph which is actually the same as computing a minimum cut based on a maximum flow. It is very natural to compute similar dual elements since König's theorem is a special case of the max-flow min-cut theorem.

In Algorithm 7.2, we mark R, the set of all vertices which can be reached from s in the residual graph. Clearly, R contains U, the set of unsaturated vertices from $\mathcal{I}$. Observe that with unit multiplicities, the set R is the same set as in Algorithm 7.1. We remark that with unit multiplicities, if we remove s, t and the edges with infinite capacities which have a positive flow from the residual graph, then the residual graph is exactly the directed graph defined in Section 7.5.

In fact, the proof of correctness of the algorithm is entirely based on Theorem 7.2.

Theorem 7.3. *Algorithm 7.2 gives a set X satisfying Property 7.1 and a feasible assignment. Moreover, X is maximum for κ-assignments.*

Proof. From D_P we create the following graph $G'_P = (I', B', E)$: let I' be the set of vertices in which each vertex $i \in \mathcal{I}_P \cap V$ is replicated $c(s,i)$ times; let B' be the set of vertices in which each vertex $b \in \mathcal{B}_P \cap V$ is replicated $c(b,t)$ times; s and t do not belong to G'_P. Let $E = \{\{u,v\} : (u,v) \in A \text{ or } (v,u) \in A\}$. A maximum flow f in D immediately gives a maximum matching M in G'_P by dividing the flow in units.

Clearly, if $\forall b \in \mathcal{B}_P\ \kappa_b = 1$, then $G'_P = G_P$, the compatibility graph defined by the same instance in which items are replicated instead of given with multiplicities. So Theorem 7.2 proves the theorem.

If κ is not a 1 vector, then the bins are also replicated. Since the bin b is replicated exactly κ_b times and any combination of κ_b items, which are compatible with b, fits into b, the assignment given by the flow is feasible and verifies Property 7.1. Moreover, any κ-assignment of k items can be divided in k one-to-one assignments of one item into a bin by replicating each bin at most a number of times equal to its robust capacity, and conversely. Moreover, the assignment obtained in G'_P is a maximum one-to-one assignment by Theorem 7.2 therefore it is a maximum κ-assignments in D_P. □

The complexity of Algorithm 7.2 is equal to the complexity of computing a maximum flow in D_P. This is polynomial in the size of D_P whose size is polynomial in the instance size and $\mathcal{B}$. In practical implementations, we can keep previously computed flows for a hot start of the maximum flow algorithm.

Finally, we can generalize Corollary 7.1:

Corollary 7.2. *For pure packing problems in which we can compute the compatibility graph in polynomial time: any instance s.t. $\sum_{b\in\mathcal{B}} \kappa_b \geq \sum_{i\in\mathcal{I}} m_i$ can be reduced to an instance with $\sum_{b\in\mathcal{B}'} \kappa_b < \sum_{i\in\mathcal{I}'} m_i$ and $\mathcal{B}' \subset \mathcal{B}$, $\mathcal{I}' \subseteq .\mathcal{I}$, in polynomial time in the input size and the number of bins, even with high-multiplicity encoding of the input.*

7.7 Experiments

We implemented the reduction and led preliminary experiments on vector bin packing problems with heterogeneous bins (VBPHB).

In order to measure the efficiency of this reduction, we embedded it in a VBPHB solver and ran it on every node of the branch and bound. We implemented a simple VBPHB solver using constraint programming (with `Choco 2` solver) and only the `bin_packing` constraint (Shaw 2004) on each dimension. We implemented the generalized reduction in a dedicated constraint. We used backtrackable structures for the compatibility graph but we did not implement hot starts for the flow algorithm which is a simple implementation of Ford-Fulkerson maximum flow algorithm (Ford and Fulkerson 1956).

We generated random uniform instances for this problem in which capacities of the bins and in the different dimensions are independent and all bins are full in all dimensions. We also generated a second set of instances in which each item is removed with a probability 0.1. We generated 100 instances of each class for the tests.

In order to have a fair evaluation of the algorithms with and without the reduction, we ran the solver using the default branching heuristic and the branching heuristic using lexical ordering.

On instances with 10 bins and 2 dimensions, there are 4.48 and 4.26 items per bin in average. We observe that the number of nodes is reduced by 17% in average when the reduction is applied on each node. The results are similar for the two branching heuristics. On the same instances with 10% of removed items, we observe on average that the number of nodes is decreased by 10%. There are however huge variations depending on the instances.

The results are much more significant on the overall number of nodes: if we sum up the number of nodes explored on all instances, this number is decreased by 68% for the first class and 88% for the second class. This means that the reduction is very powerful on hard instances.

On instances with 7 bins, 3 dimensions and 10% of items removed, the average number of nodes is reduced by 19% while the overall number of nodes (as well as the overall number of backtrack and time spent) is reduced by more than 99%.

If we do not remove 10% of the items, some instances cannot be solved within an hour without the reduction while if we use the reduction, it takes a total time of 2.2s to solve the 100 instances from the set.

On larger instances, with more bins or dimensions, one cannot expect to solve instances with this solver without the reductions.

There is much room for optimization in the implementation but the results show already that the reduction is very powerful and can be computed efficiently.

7.8 Conclusion

In this chapter, we proposed a reduction algorithm for packing problems. This reduction algorithm is polynomial in the number of item types and the number of bins. Moreover, it can easily be used in practice with a strong efficiency.

The algorithm is very general and can be applied to any packing problem. It can be used as is in heuristics and exact algorithms for pure packing problems and can be used for heuristics or lower bounds computations in "non-pure" packing problems.

In any packing problem, the number of bins used in an optimal solution is obviously smaller than the number of items. However, if we are given an instance with more bins than items, the

instance is not necessarily feasible and we cannot easily remove bins from this instance while guaranteeing the feasibility. A consequence of our reduction algorithm is that for any pure packing problem in which we can verify in polynomial time whether an item fits into a bin, any instance with more bins than items can be reduced in polynomial time (in the input size and the number of bins) to an instance in which the number of bins is strictly smaller than the number of items. Although it is obvious that such a reduction exists, finding one is not trivial when there is more than one dimension and, to the best of our knowledge, it was not known whether this could be done in polynomial time, even when high-multiplicity encoding is not considered.

Implementing the reduction algorithm is straightforward for pure packing problems and with immediate benefits. For other packing problems the reduction can be implemented for heuristic methods or in the branching heuristic for exact methods. Moreover, it is unnecessary to apply it in every node of the branch-and-bound. Calling it once in a while should be sufficient and spare computation time.

Further research may focus on improving the reduction algorithm by extending the compatibility graph. Observe that if a bin is compatible with one big and many small items, the value of κ_b is likely to be small because of the big item. It is maybe possible to account for such phenomenons by modifying the compatibility graph. Furthermore, we can intricate this question within a branch-and-bound algorithm: if by assigning an item we can proceed to a large reduction then it is very interesting to immediately branch and assign the item. Another way to deal with this problem and improve reductions is to merge items: for instance, in the expanded compatibility graph (with no multiplicities), if two items have the same compatibilities and by summing their requirements (merging the items) the compatibilities are unchanged, then we can merge these two items into a new one. If we consider multiplicities, we can also consider splitting a type into two types. So we can obtain a new compatibility graph by repeatedly merging two items until there is no pair which can be merged without removing edges. Then we can apply the reduction and we obtain a set which still verifies Property 7.1 and which may not have been found by using the robust capacities only.

It would also be interesting to see whether the reduction algorithm can be improved by specializing it to a particular packing problem.

Chapter 8

Conclusion

In this thesis, we presented the concerns occurring in high-multiplicity scheduling and we studied related high-multiplicity scheduling and packing problems. We have shown how difficult it is to study high-multiplicity problems and that it raises many interesting questions. On example problems, we provided tools to study high-multiplicity problems and emphasized both the importance and the difficulty of obtaining polynomial algorithms and certificates. The complexity study of these problems is often hard with only classical complexity tools since it is often not even possible to show whether the problems belong to $\mathcal{NP}$.

On the scheduling problem with forbidden start and completions times, we described a polynomial certificate in which we group jobs and use symmetries of solutions of the problem. We have shown that the problem is polynomial for large diversity instances. The algorithm combines the polynomial algorithm for the case where the input is assumed to use a non high-multiplicity encoding with a preprocessing algorithm which schedules batches of processes. It either finds a solution or reduces the problem to a subproblem in which the multiplicities are 0 or 1. We have also shown that this problem is fixed-parameter tractable when the number of unavailabilities is fixed.

The identical coupled-task scheduling problem is a very singular problem. This problem illustrates an extreme case in high-multiplicity scheduling. We gave insights showing that this problem is very hard, even if we suppose that the number of tasks is polynomial in the input size. Yet, its complexity status remains open for both cases. We studied the properties of this problem and proposed algorithms to obtain good feasible solutions. The feeding algorithm is a secondary problem which was obtained when we adopted a completely different point of view on the identical coupled-task scheduling problem. Based on this problem, we derive an algorithm with low complexity and provide solutions whose structures are so complicated that it seemed at first to be impossible to obtain these solutions with a non-enumerative algorithm. The identical coupled-task scheduling problem is of greater interest than the underlying problem. We believe that significant progress on this problem cannot be achieved without a complete understanding of this problem and either very smart lower bounds or great advances on handling complexity classes containing $\mathcal{NP}$. In both cases, settling the complexity status of this problem would be a great progress for coupled-task scheduling and high-multiplicity scheduling in general.

On the bin stretching problem, we used approximation and high-multiplicity scheduling techniques to solve a semi-online scheduling problem. We proposed an algorithm with new best performance by classifying items, bins and grouping bins together to form powerful struc-

tures. By mixing game theory and computer science techniques, we also developed an exhaustive search algorithm to find improved lower bounds and for the first time on this problem we found a lower bound which is better than the classical 4/3 lower bound for online packing problems. Combining these two results, we see that we have reduced the gap between the best lower and best upper bound on this problem by almost 30%.

In 2012, I took part in the ROADEF challenge which was proposed by Google. With my teammate, Sofia Zaourar, we have quickly seen that local search algorithms would perform very well on this problem but we also knew that when it comes to local search, with the same moves implemented, the best algorithm is usually the one which is implemented at the lowest machine level. We decided however to experiment different, constructive approaches. We abstracted the problem of finding new feasible solutions from scratch to the vector bin packing problem with heterogeneous bins and additional constraints. We developed heuristics for this problem and obtained new feasible solutions; some among them being almost impossible to obtain with local search heuristics without global constructing moves. For example, on instances $B10/X10$ of the challenge, we found out that we can switch the neighborhoods. Actually, we found feasible solutions for all assignments of any neighborhood into any other neighborhood. Unfortunately, these solutions were not very useful on challenge instances because the process move costs were too high. We also implemented a matheuristic approach using a variable neighborhood search and an integer program to solve the subproblems but we finally had to stick to local search in our final program.

Our study of the vector bin packing problem with heterogeneous bins presents however an interesting packing problem and our experiments have shown an almost surprisingly very good performance achieved by a best-of-many heuristic on this hard packing problem. By studying how to adapt these heuristics to the machine reassignment problem, we also exposed interesting properties of this problem.

Finally, we presented a simple, polynomial, reduction algorithm for packing problems. Since this algorithm is based on the weighted compatibility graph of the packing, it accounts for the multiplicities of the items and can be applied directly to many types of packing problem. The reduction algorithm can be generalized and applied with many additional constraints and when there are constraints preventing from generalizing, the algorithm can be adapted to compute lower bounds or guide branching algorithms. The algorithm is efficient in practice and gives good reductions of the instance set. Moreover, as the number of dimensions increases, the compatibility graph usually gets sparser making the algorithm even more likely to find large reductions.

Further research on high-multiplicity could be inspired from graph theory approaches since stating items multiplicity is very similar to stating capacities in a network graph for instance. We can push further the analogy with network flows. The worst case performance of Ford-Fulkerson algorithm is $\mathcal{O}(nmC)$ where n is the number of vertices in the graph, m the number of edges and C the maximum capacity. This complexity is pseudo-polynomial and considered impractical by most researchers. Yet, many of these researchers are still considering that in scheduling the jobs are always given separately and do not consider high-multiplicity encoding of the input. For the exact same reasons as they consider that $\mathcal{O}(nmC)$ is not a reasonable complexity, they shall consider that any non-high-multiplicity encoding is unreasonable when the multiplicities are not very small.

Cyclic scheduling is also a topic which shares common problems with high-multiplicity sche-

duling. This topic is not discussed in depth in this thesis but it shares many of the concerns of high-multiplicity scheduling. Especially, what is the size of an optimal cycle and does it have a polynomial encoding? We have seen such an example on the identical coupled-task scheduling problem. Investigating cyclic scheduling problems with regards to high-multiplicity constraints will be an interesting source of work for the future. I have already started research on this topic with colleagues from Maastricht university on a lot sizing problem with switching costs (Oosterwijk et al. 2014) but our results are not discussed in the thesis.

We have seen that the classical complexity classes $\mathcal{P}$ and $\mathcal{NP}$ are not very well adapted to analyze high-multiplicity problems. Some works have already been led by Brauner et al. (2007) to propose a new framework to describe the complexity of algorithms. However, there is still a lack of knowledge on how to classify the difficulty of these problems since many high-multiplicity problems are not proven to be in $\mathcal{NP}$ and may not belong to it. Further work can seek to classify high-multiplicity problems with respect to the above-$\mathcal{NP}$ classes: PSPACE, EXPTIME and EXPSPACE.

In order to solve a high-multiplicity problem (and any problem in general) it is necessary to understand the structures of its solutions very well. Yet, when we are not able to have a complete understanding of these structures but *need* to solve the problem, we still have to design approaches which are capable of handling and solving the problem. Further research in this direction can be led to extend the work presented in this thesis. In life as in computer science, when you cannot be clever enough, you have to be able to try many things. The translation in computer science is that if you cannot find the structures, you can still massively increase the computing power to try many things. There are several ways of achieving a massive increase in computing power. One of them is to use computing clusters. Computing clusters offer the flexibility of programming on CPUs but are very expensive. Another way is to use heterogeneous computing which takes advantages of processors on other devices than CPUs such as graphic cards (GPU computing). In GPU computing however there are multiple additional constraints on the programs. The programming paradigm imposed by the SIMD (Single Instruction on Multiple Data) architectures makes it very difficult to use GPU computing for combinatorial algorithms which are often sequential and hard to implement in parallel apart from using branch-and-bound approaches. We have started experiments using GPUs and MIC coprocessors (MIC stands for Many Integrated Core which is basically a union of a SIMD architecture with x86 processors) to implement combinatorial optimization problems in parallel. We are currently working on the knapsack and the bin packing problem and are looking for new original ways to implement efficient parallel algorithms on these architectures. Progress in this area would be of interest for high-multiplicity problems as well as combinatorial optimization and computing in general.

Conclusion (français)

Nous avons présenté dans cette thèse l'ordonnance high-multiplicity et y avons étudié divers problèmes d'ordonnancement et de placement d'objets. Nous avons montré qu'il est difficile d'étudier des problèmes d'ordonnancement high-multiplicity et, en particulier, qu'il peut être très difficile de les classifier au regard des classes de complexité classiques, $\mathcal{P}$ et $\mathcal{NP}$. Cette difficulté singulière est due au fait qu'il peut être difficile (voire impossible) d'exhiber un certificat de taille polynomiale en la taille de l'instance pour ces problèmes.

Concernant le problème d'ordonnancement avec instants interdits, nous avons exhibé un certificat polynomial en regroupant les tâches et en exploitant les symétries du problème. Nous avons montré que ce problème est polynomial, même avec un encodage high-multiplicity de l'instance, lorsque le nombre de types de tâches est supérieur strictement au nombre d'instants interdits. L'algorithme que nous proposons combine l'algorithme polynomial pour le cas où l'instance n'utilise pas un encodage high-multiplicity avec un algorithme de preffixage trouvant un préfixe accolable à l'ordonnancement, cette fois avec une complexité polynomiale en la taille de l'encodage high-multiplicity. Nous avons également montré que ce problème est FPT avec comme paramètre fixé le nombre d'instants interdits.

Le problème des tâches couplées identiques est un problème très particulier. Il illustre toute la difficulté de l'ordonnancement multi-opérations et l'ordonnancement high-multiplicity. Nous avons montré que ce problème est difficile, indépendamment de l'encodage de l'entrée. La complexité de ce problème demeure néanmoins ouverte. Nous avons étudié les propriétés de ce problème et avons proposé des algorithmes polynomiaux ou de faible complexité permettant d'obtenir de bonnes solutions réalisables. Le problème d'alimentation que nous proposons est d'un intérêt tout à fait particulier puisque nous transformons le problème d'utilisation des temps morts en un problème tout à fait différent et obtenons un algorithme performant grâce à cette abstraction. En particulier, les solutions obtenues présentent des structures complexes que nous doutions pouvoir obtenir avec un algorithme non énumératif.

Nous avons proposé un algorithme polynomial à performance garantie pour le problème de bin stretching online. L'algorithme utilise des techniques communes à l'ordonnancement high-multiplicity et aux algorithmes d'approximation: les objets et les récipients sont classés en un petit nombre de types distincts, puis les affectations sont décidées en fonction des types en présence. En mixant les techniques des sciences informatiques et de la théorie des jeux, nous avons par ailleurs développé un algorithme nous permettant d'améliorer la borne inférieure pour ce problème pour la première fois depuis son introduction. Si l'on combine ces deux résultats, on constante que nous avons réduit l'écart entre la meilleure borne inférieure connue et la meilleure borne supérieure connue pour ce problème de presque 30%.

En 2012, j'ai participé au challenge ROADEF proposé par Google. Avec ma coéquipière, Sofia

Zaourar, nous nous sommes rapidement rendu compte que les algorithmes de recherche locale étaient très bien adapté pour ce problème. Néanmoins, nous savions également que, compte tenu du fait que les autres compétiteurs verront également ceci, il nous faudrait optimiser notre code au plus bas niveau possible et régler les paramètres avec une granularité fine afin d'être compétitifs. Néanmoins, afin de pouvoir contribuer scientifiquement, nous avons pris la décision de tester d'autres approches pour résoudre ce problème. Nous voulions proposer une approche constructive pour ce problème et avons donc étudié comment obtenir de nouvelles solutions réalisables. Nous avons abstrait le problème d'obtenir de nouvelle solution réalisables en un problème de placement d'objets dans des récipients hétérogènes pluri-contraints. Nous avons développé des heuristiques pour ce problème et obtenu de nouvelles solutions réalisables diversifiées. Ces solutions n'étaient malheureusement pas efficaces sur le challenge en raison des coûts élevés pour déplacer des processus. Néanmoins, le problème de placements d'objets dans des récipients hétérogènes modélise naturellement les problèmes de placements de machines virtuelles sur des serveurs. Nous avons proposé de nombreuses heuristiques très simple et avons également testé l'approche best-of-many. Nous avons constaté que nos heuristiques ont d'excellentes performances sur les instances de la littérature du problème de vector bin packing et des performances correctes sur le problème de placement d'objets dans des récipients hétérogènes. Nous avons étudié comment généraliser ces heuristiques à d'autres problèmes et avons exposé quelques une des propriétés structurelles du problème de réaffectation de machines permettant cette généralisation.

Dans le dernier chapitre, nous avons présenté une réduction polynomiale pour les problèmes de placement d'objets. Cette réduction est très générale, adaptable à un très grand nombre de problèmes de placements d'objets et est basée sur le calcul d'un flot dans le graphe des compatibilités des objets. La réduction est polynomiale en le nombre de types d'objets et le nombre de récipients, elle peut donc être utilisée même avec des instances ayant un encodage high-multiplicity. Pour les problèmes ayant des contraintes additionnelles complexes, elle peut être utilisée pour calculer des bornes inférieures ou pour guider un algorithme de branchement. Des expériences ont montré que la réduction est efficace et performante en pratique.

Dans de future recherches, on pourra s'intéresser aux graphes compactes et aux approches utilisées dans les graphes. L'ordonnancement cyclique est également très proche de l'ordoannancement high-multiplicity en de nombreux aspects. En particulier, sur la taille de la sortie, par exemple, qu'advient-il si le cycle est très long, même si les tâches ne sont pas regroupées par types dans l'instance ?

Nous avons pu constater dans cette thèse que les classes de complexité classiques ne conviennent pas vraiment à l'étude des problèmes high-multiplicity. Les travaux de Brauner et al. (2007) ont permis d'améliorer ces classes en proposant une classification plus affinée de la complexité des algorithmes. Il reste néanmoins de nombreux efforts à fournir concernant la complexité des problèmes. Soit en affinant la classification, soit en investiguant d'avantage les classes de complexités contenant $\mathcal{NP}$, en particulier PSPACE, EXPTIME et EXPSPACE.

Nous avons constaté que pour espérer résoudre un problème high-multiplicity, il faut en avoir une compréhension structurelle totale. Néanmoins, que peut-on faire lorsqu'on n'a pas atteint ce niveau de compréhension et qu'on a néanmoins besoin de résoudre de tels problèmes ? Une idée peut-être d'augmenter massivement la puissance de calcul, ce qui peut être fait en utilisant des grappes de calculs qui offrent la flexibilité de programmer sur des CPU mais avec de nombreux problèmes de connexion réseau. Ces grappes sont par ailleurs, généralement très onéreuses. Un autre moyen est d'utiliser des moyens de calcul hétérogènes et utilisant d'autres

processeurs de calcul que des CPU, comme les cartes graphiques GPU. Ces dernières présentent l'avantage d'offrir un excellent rapport puissance de calcul/prix mais les inconvénients d'être basés sur une architecture SIMD (Single Instruction on Multiple Data) rendant très difficile le portage d'algorithmes combinatoires sur ces équipements. Nous avons démarré des travaux sur ces problématiques, sur GPU et coprocesseurs Xeon Phi. Nous travaillons actuellement sur des problèmes de sac-à-dos et de bin-packing et recherchons de nouvelles manières originales d'utiliser ces architectures.

Appendices

Appendix A

Notations

A.1 Notations in this thesis

In this thesis, we use the following notations:

s	number of distinct types of jobs
h	number of distinct types of machines
m_j	multiplicity of job j (the number of jobs j to process)
n	total number of jobs: $n = \sum_{j=1}^{s} m_j$
m	total number of machines

A.2 Common functions

- $\mathbf{1}$ is the *indicator* function: $\mathbf{1}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}$, we also use $\mathbf{1}_{expr} = \begin{cases} 1 & \text{if } expr \text{ is true} \\ 0 & \text{otherwise} \end{cases}$
- $\lfloor \ \rfloor$ is the *floor* function: $\lfloor 5.7 \rfloor = 5$
- $\lceil \ \rceil$ is the *ceiling* function: $\lceil 5.2 \rceil = 6$
- $[\ ]$ is the *round* function: $[5.2] = 5$, $[5.7] = 6$
- $x^+ = \max(x, 0)$

A.3 Graham's three-field notations

Graham's notations were introduced in Graham et al. (1977). These notations are a convenient way to present any scheduling problem using three fields: $\alpha|\beta|\gamma$. Information related to the encoding of the input is denoted in the β field. In the following, we present the meaning of these fields and some classical attributes for each one of them.

Using these notations, many scheduling problems complexity are referenced on Knust's website (Brucker and Knust). These results can be searched using the engine from Christophe Dürr (Dürr).

The first field, α, denotes the machines setup. If a number or m is added, then this means that the number of machines is fixed and is equal to m instead of being part of the problem input.

- 1: there is a single machines
- P: parallel identical machines
- Q: parallel machines with proportionnal speeds. The processing time of job i on machine j is the processing time p_i divided by the machine speed s_j
- R: parallel unrelated machines. The processing time of job i on machine j is p_{ij}.
- F: flow shop problem
- J: job shop problem
- O: open shop problem
- Xm: the setup is X and the number of machines is fixed and equal to m.

The second field, β, denotes the constraints:

- r_j: release dates. Job j cannot be started before time r_j.
- $\tilde{d}_j$: deadlines job j cannot finish after time $\tilde{d}_j$.
- $pmtn$: preemption is allowed. Jobs may be preempted and execution resumed later, possibly on a different machine
- $size_j$: job j needs simultaneously $size_j$ machines for its execution.
- $prec$: jobs have precedence relations
- $sp-tree, tree, intree, outtree, chain$: specific precedence relations
- M_J: multiplicities on the jobs are allowed but not on the machines.
- HM: input is provided using a high-multiplicity encoding.

The last field, γ, is the objective function. The aim of the problem is to minimize it. The objective can be altered using weights w_j. An optimization criterion is regular if it is non-decreasing in the completion times $C_1, \ldots, C_n$.

- $C_{\max}$: makespan
- $L_{\max}$: algebraic lateness
- $T_{\max}$: tardyness
- f_{max}: a bottleneck regular criterion
- $\sum C_j$: sum of completion times
- $\sum U_j$: number of tardy jobs
- f_{sum}: a sum regular criterion

Remark that some objectives, such as $L_{\max}, T_{\max}$ and $\sum U_j$ implies that jobs have due-dates d_j. Usually, d_j is not specified in the β field since it is induced by the objective function. In tha same way, w_j are weights on the jobs, hence, for instance, while $R|p_j = 1|\sum C_j$ is a high-multiplicity scheduling problem, $R|p_j = 1|\sum w_j C_j$ is not.

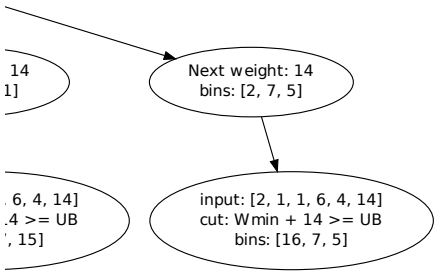
Appendix B

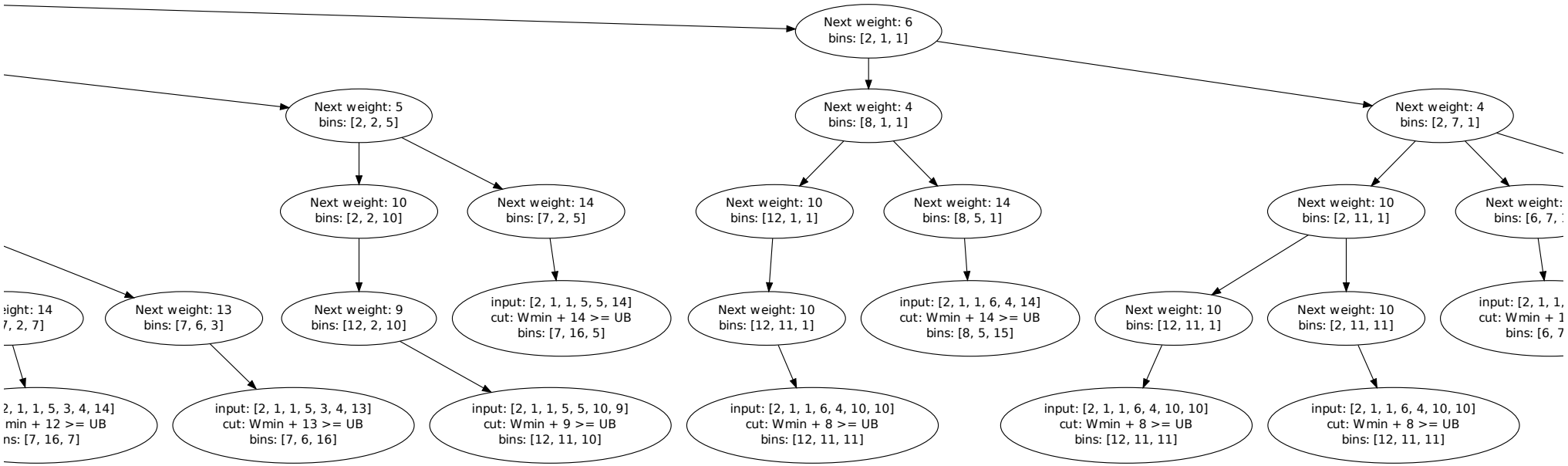
Bin Stretching

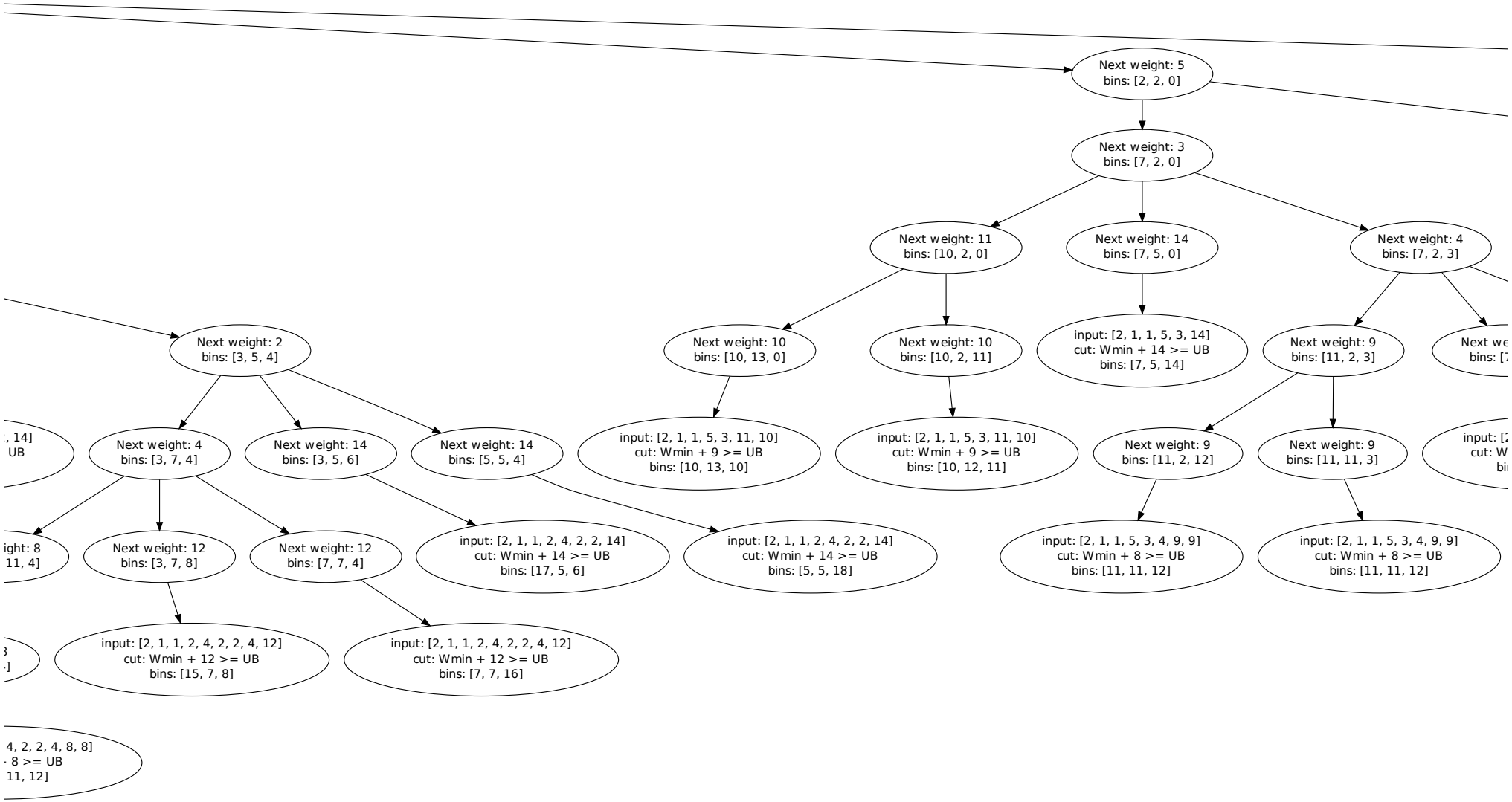
B.1 Proof of the lower bound

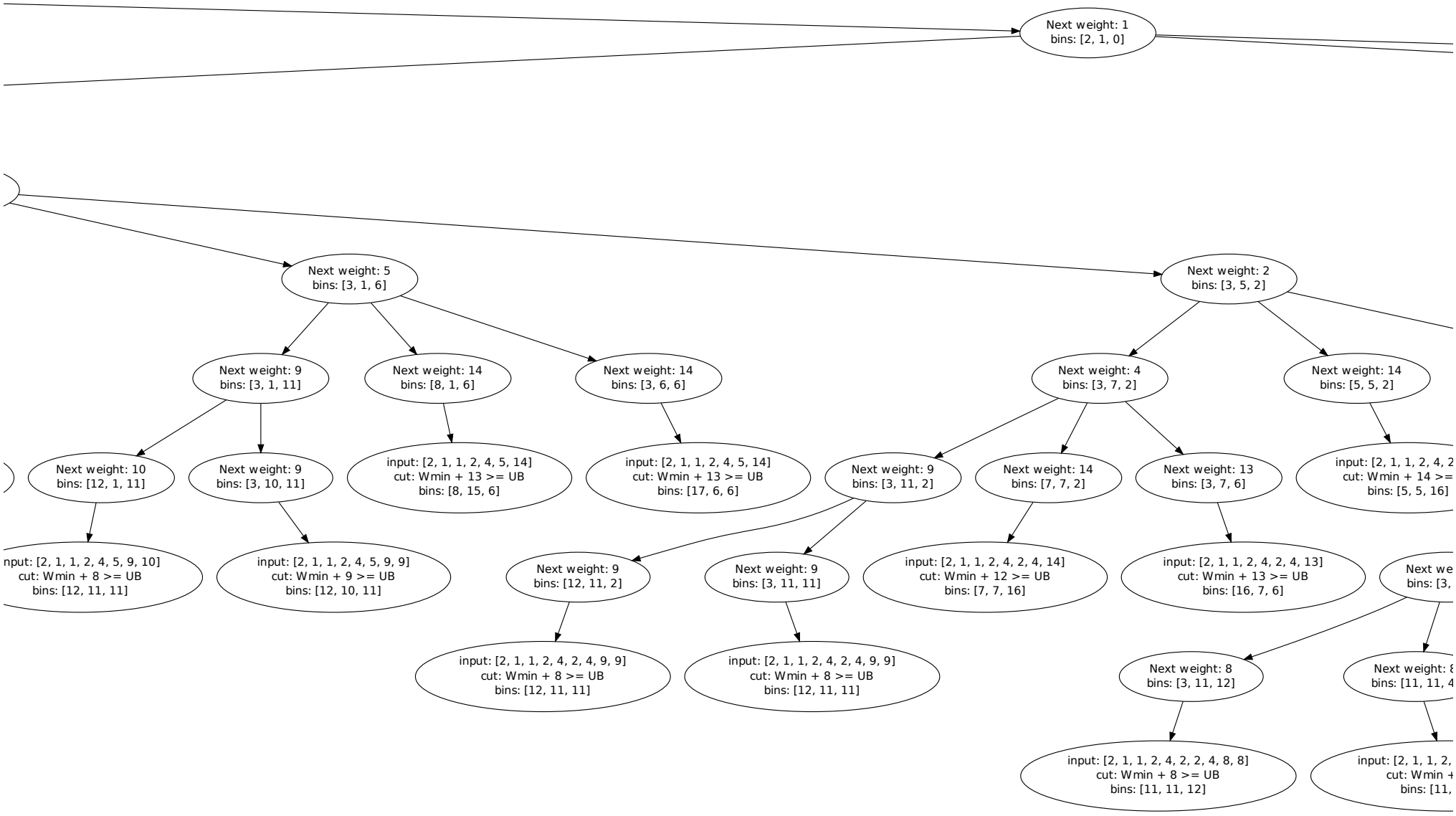
The following tree proves the 19/14 lower bound for the bin stretching problem. This lower bound was obtained using our algorithm with parameters $m = 3$ and $C = 14$.

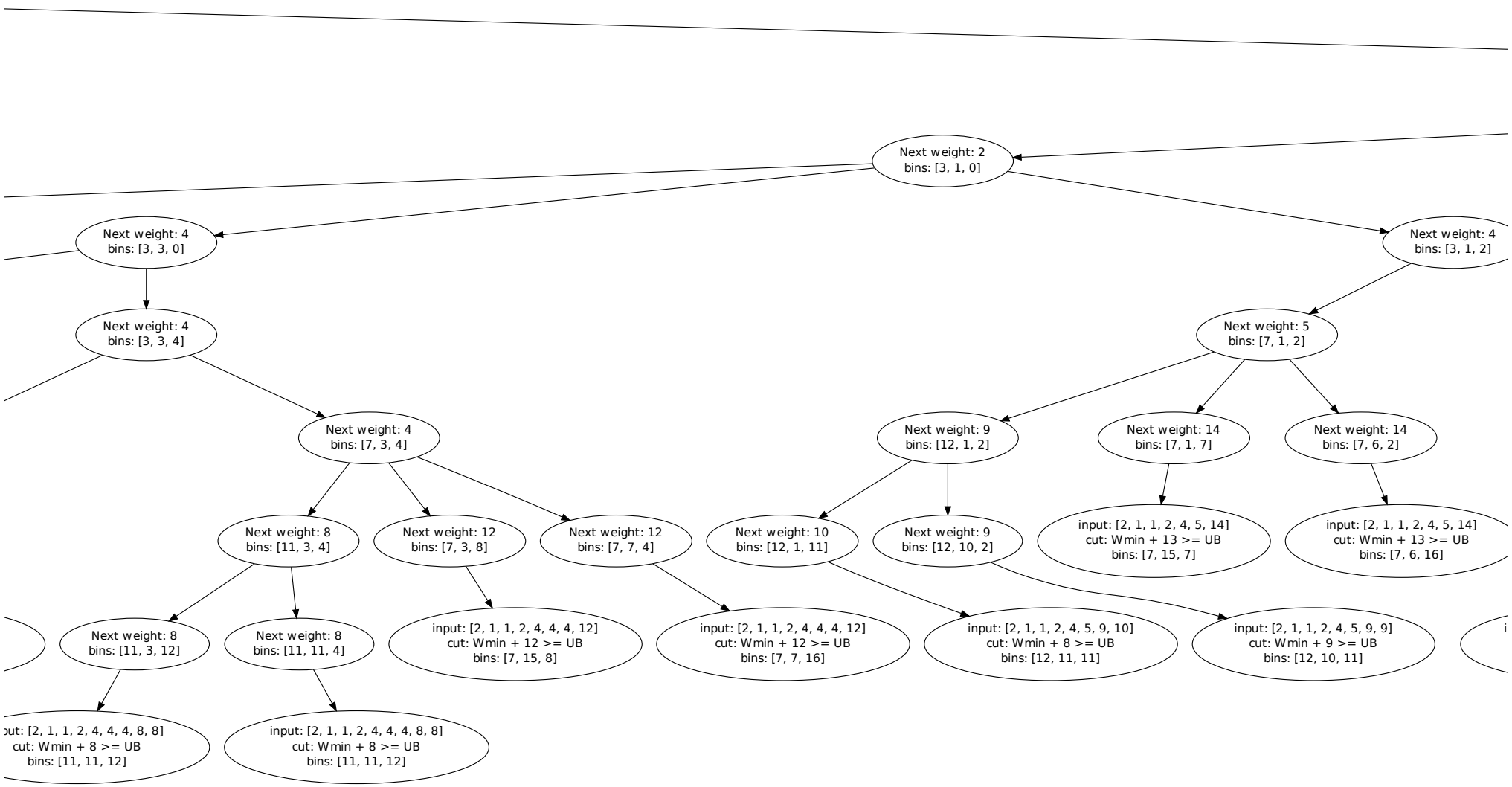
In the proof, a single decision of the adversary is provided for each decision of the algorithm. We do not explore branches where the algorithm packs the item in a bin, making it larger than or equal to 19. Moreover, we stop exploring a branch when there is a feasible item making all algorithms fail. We denote these latter nodes by "`cut: Wmin+`w_j`>=UB`". We recall the input sequence on the leaves. The next items are not added to this sequence. For instance, if we have a leaf "`input: [2,1,7] / cut: Wmin + 3 >= UB`" then the whole input sequence is [2,1,7,3].

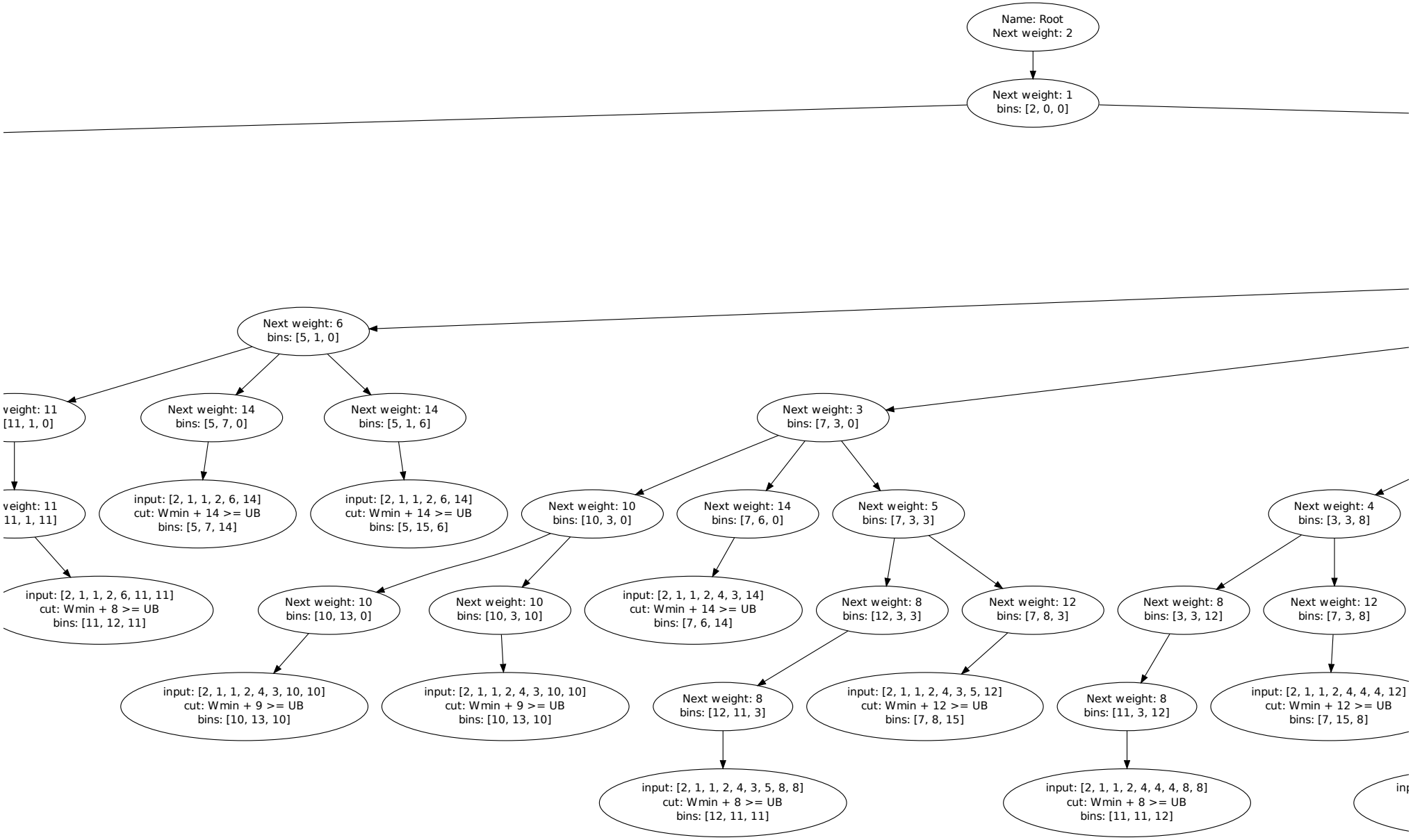

14
1]
Next weight: 14
bins: [2, 7, 5]
6, 4, 14]
.4 >= UB
', 15]
input: [2, 1, 1, 6, 4, 14]
cut: Wmin + 14 >= UB
bins: [16, 7, 5]

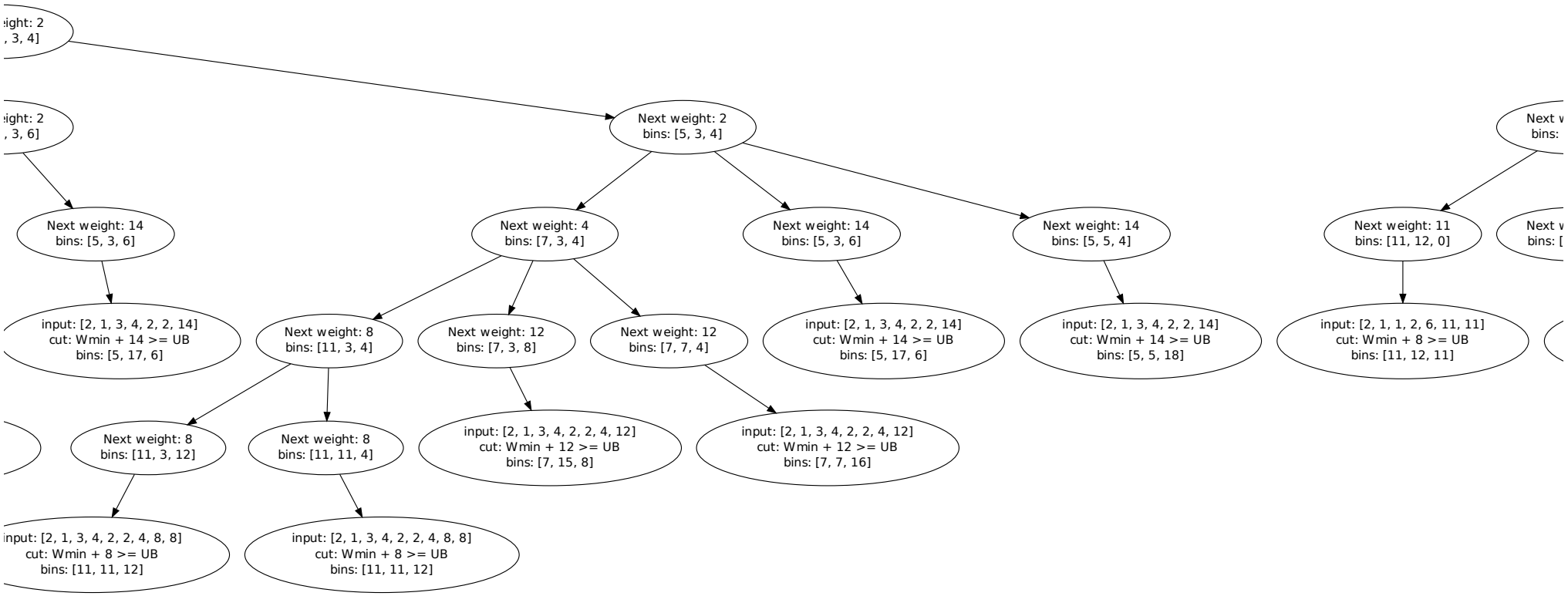

Next weight: 6
bins: [2, 1, 1]
Next weight: 5
bins: [2, 2, 5]
Next weight: 4
bins: [8, 1, 1]
Next weight: 4
bins: [2, 7, 1]
Next weight: 10
bins: [2, 2, 10]
Next weight: 14
bins: [7, 2, 5]
Next weight: 10
bins: [12, 1, 1]
Next weight: 14
bins: [8, 5, 1]
Next weight: 10
bins: [2, 11, 1]
Next weight: 13
bins: [7, 6, 3]
Next weight: 9
bins: [12, 2, 10]
input: [2, 1, 1, 5, 5, 14]
cut: Wmin + 14 >= UB
bins: [7, 16, 5]
Next weight: 10
bins: [12, 11, 1]
input: [2, 1, 1, 6, 4, 14]
cut: Wmin + 14 >= UB
bins: [8, 5, 15]
Next weight: 10
bins: [12, 11, 1]
Next weight: 10
bins: [2, 11, 11]
input: [2, 1, 1, 5, 3, 4, 13]
cut: Wmin + 13 >= UB
bins: [7, 6, 16]
input: [2, 1, 1, 5, 5, 10, 9]
cut: Wmin + 9 >= UB
bins: [12, 11, 10]
input: [2, 1, 1, 6, 4, 10, 10]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
input: [2, 1, 1, 6, 4, 10, 10]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
input: [2, 1, 1, 6, 4, 10, 10]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]

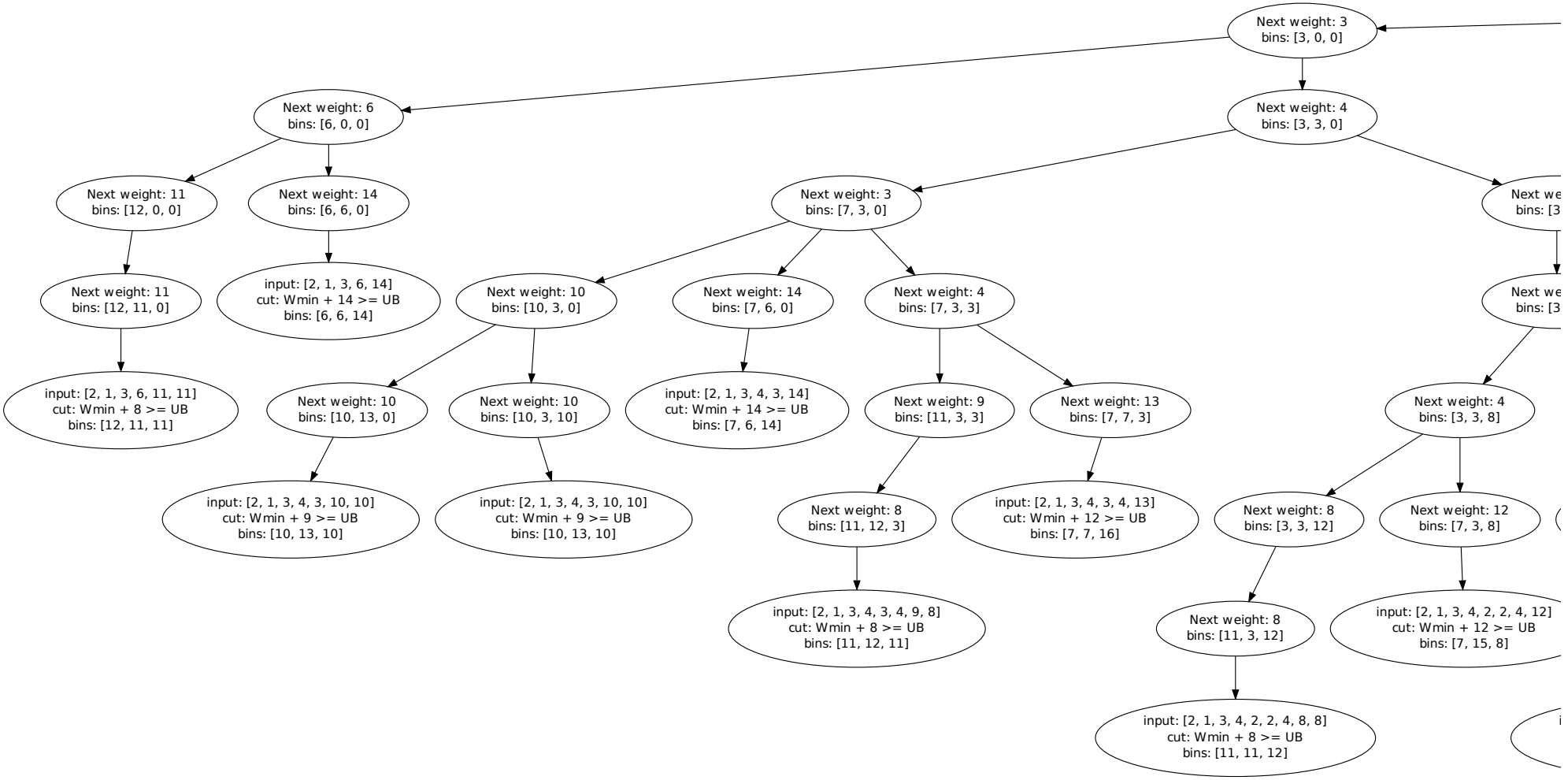

Next weight: 5
bins: [2, 2, 0]
Next weight: 3
bins: [7, 2, 0]
Next weight: 11
bins: [10, 2, 0]
Next weight: 14
bins: [7, 5, 0]
Next weight: 4
bins: [7, 2, 3]
Next weight: 2
bins: [3, 5, 4]
Next weight: 10
bins: [10, 13, 0]
Next weight: 10
bins: [10, 2, 11]
input: [2, 1, 1, 5, 3, 14]
cut: Wmin + 14 >= UB
bins: [7, 5, 14]
Next weight: 9
bins: [11, 2, 3]
Next weight: 4
bins: [3, 7, 4]
Next weight: 14
bins: [3, 5, 6]
Next weight: 14
bins: [5, 5, 4]
input: [2, 1, 1, 5, 3, 11, 10]
cut: Wmin + 9 >= UB
bins: [10, 13, 10]
input: [2, 1, 1, 5, 3, 11, 10]
cut: Wmin + 9 >= UB
bins: [10, 12, 11]
Next weight: 9
bins: [11, 2, 12]
Next weight: 9
bins: [11, 11, 3]
Next weight: 12
bins: [3, 7, 8]
Next weight: 12
bins: [7, 7, 4]
input: [2, 1, 1, 2, 4, 2, 2, 14]
cut: Wmin + 14 >= UB
bins: [17, 5, 6]
input: [2, 1, 1, 2, 4, 2, 2, 14]
cut: Wmin + 14 >= UB
bins: [5, 5, 18]
input: [2, 1, 1, 5, 3, 4, 9, 9]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]
input: [2, 1, 1, 5, 3, 4, 9, 9]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]
input: [2, 1, 1, 2, 4, 2, 2, 4, 12]
cut: Wmin + 12 >= UB
bins: [15, 7, 8]
input: [2, 1, 1, 2, 4, 2, 2, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 7, 16]


Next weight: 1
bins: [2, 1, 0]
Next weight: 5
bins: [3, 1, 6]
Next weight: 2
bins: [3, 5, 2]
Next weight: 9
bins: [3, 1, 11]
Next weight: 14
bins: [8, 1, 6]
Next weight: 14
bins: [3, 6, 6]
Next weight: 4
bins: [3, 7, 2]
Next weight: 14
bins: [5, 5, 2]
Next weight: 10
bins: [12, 1, 11]
Next weight: 9
bins: [3, 10, 11]
input: [2, 1, 1, 2, 4, 5, 14]
cut: Wmin + 13 >= UB
bins: [8, 15, 6]
input: [2, 1, 1, 2, 4, 5, 14]
cut: Wmin + 13 >= UB
bins: [17, 6, 6]
Next weight: 9
bins: [3, 11, 2]
Next weight: 14
bins: [7, 7, 2]
Next weight: 13
bins: [3, 7, 6]
input: [2, 1, 1, 2, 4, 2
cut: Wmin + 14 >=
bins: [5, 5, 16]
nput: [2, 1, 1, 2, 4, 5, 9, 10]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
input: [2, 1, 1, 2, 4, 5, 9, 9]
cut: Wmin + 9 >= UB
bins: [12, 10, 11]
Next weight: 9
bins: [12, 11, 2]
Next weight: 9
bins: [3, 11, 11]
input: [2, 1, 1, 2, 4, 2, 4, 14]
cut: Wmin + 12 >= UB
bins: [7, 7, 16]
input: [2, 1, 1, 2, 4, 2, 4, 13]
cut: Wmin + 13 >= UB
bins: [16, 7, 6]
Next we
bins: [3,
input: [2, 1, 1, 2, 4, 2, 4, 9, 9]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
input: [2, 1, 1, 2, 4, 2, 4, 9, 9]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
Next weight: 8
bins: [3, 11, 12]
Next weight: 8
bins: [11, 11, 4
input: [2, 1, 1, 2, 4, 2, 2, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]
input: [2, 1, 1, 2,
cut: Wmin +
bins: [11,


Next weight: 2
bins: [3, 1, 0]
Next weight: 4
bins: [3, 3, 0]
Next weight: 4
bins: [3, 1, 2]
Next weight: 4
bins: [3, 3, 4]
Next weight: 5
bins: [7, 1, 2]
Next weight: 4
bins: [7, 3, 4]
Next weight: 9
bins: [12, 1, 2]
Next weight: 14
bins: [7, 1, 7]
Next weight: 14
bins: [7, 6, 2]
Next weight: 8
bins: [11, 3, 4]
Next weight: 12
bins: [7, 3, 8]
Next weight: 12
bins: [7, 7, 4]
Next weight: 10
bins: [12, 1, 11]
Next weight: 9
bins: [12, 10, 2]
input: [2, 1, 1, 2, 4, 5, 14]
cut: Wmin + 13 >= UB
bins: [7, 15, 7]
input: [2, 1, 1, 2, 4, 5, 14]
cut: Wmin + 13 >= UB
bins: [7, 6, 16]
Next weight: 8
bins: [11, 3, 12]
Next weight: 8
bins: [11, 11, 4]
input: [2, 1, 1, 2, 4, 4, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 15, 8]
input: [2, 1, 1, 2, 4, 4, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 7, 16]
input: [2, 1, 1, 2, 4, 5, 9, 10]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
input: [2, 1, 1, 2, 4, 5, 9, 9]
cut: Wmin + 9 >= UB
bins: [12, 10, 11]
ut: [2, 1, 1, 2, 4, 4, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]
input: [2, 1, 1, 2, 4, 4, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]


Name: Root
Next weight: 2
Next weight: 1
bins: [2, 0, 0]
Next weight: 6
bins: [5, 1, 0]
Next weight: 3
bins: [7, 3, 0]
Next weight: 14
bins: [5, 7, 0]
Next weight: 14
bins: [5, 1, 6]
input: [2, 1, 1, 2, 6, 14]
cut: Wmin + 14 >= UB
bins: [5, 7, 14]
input: [2, 1, 1, 2, 6, 14]
cut: Wmin + 14 >= UB
bins: [5, 15, 6]
Next weight: 10
bins: [10, 3, 0]
Next weight: 14
bins: [7, 6, 0]
Next weight: 5
bins: [7, 3, 3]
Next weight: 4
bins: [3, 3, 8]
input: [2, 1, 1, 2, 4, 3, 14]
cut: Wmin + 14 >= UB
bins: [7, 6, 14]
Next weight: 10
bins: [10, 13, 0]
Next weight: 10
bins: [10, 3, 10]
Next weight: 8
bins: [12, 3, 3]
Next weight: 12
bins: [7, 8, 3]
Next weight: 8
bins: [3, 3, 12]
Next weight: 12
bins: [7, 3, 8]
input: [2, 1, 1, 2, 4, 3, 10, 10]
cut: Wmin + 9 >= UB
bins: [10, 13, 10]
input: [2, 1, 1, 2, 4, 3, 10, 10]
cut: Wmin + 9 >= UB
bins: [10, 13, 10]
Next weight: 8
bins: [12, 11, 3]
input: [2, 1, 1, 2, 4, 3, 5, 12]
cut: Wmin + 12 >= UB
bins: [7, 8, 15]
Next weight: 8
bins: [11, 3, 12]
input: [2, 1, 1, 2, 4, 4, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 15, 8]
input: [2, 1, 1, 2, 4, 3, 5, 8, 8]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
input: [2, 1, 1, 2, 4, 4, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]


ight: 2
, 3, 4]
ight: 2
, 3, 6]
Next weight: 2
bins: [5, 3, 4]
Next w
bins:
Next weight: 14
bins: [5, 3, 6]
Next weight: 4
bins: [7, 3, 4]
Next weight: 14
bins: [5, 3, 6]
Next weight: 14
bins: [5, 5, 4]
Next weight: 11
bins: [11, 12, 0]
Next w
bins: [
input: [2, 1, 3, 4, 2, 2, 14]
cut: Wmin + 14 >= UB
bins: [5, 17, 6]
Next weight: 8
bins: [11, 3, 4]
Next weight: 12
bins: [7, 3, 8]
Next weight: 12
bins: [7, 7, 4]
input: [2, 1, 3, 4, 2, 2, 14]
cut: Wmin + 14 >= UB
bins: [5, 17, 6]
input: [2, 1, 3, 4, 2, 2, 14]
cut: Wmin + 14 >= UB
bins: [5, 5, 18]
input: [2, 1, 1, 2, 6, 11, 11]
cut: Wmin + 8 >= UB
bins: [11, 12, 11]
Next weight: 8
bins: [11, 3, 12]
Next weight: 8
bins: [11, 11, 4]
input: [2, 1, 3, 4, 2, 2, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 15, 8]
input: [2, 1, 3, 4, 2, 2, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 7, 16]
input: [2, 1, 3, 4, 2, 2, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]
input: [2, 1, 3, 4, 2, 2, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]


Next weight: 3
bins: [3, 0, 0]
Next weight: 6
bins: [6, 0, 0]
Next weight: 4
bins: [3, 3, 0]
Next weight: 11
bins: [12, 0, 0]
Next weight: 14
bins: [6, 6, 0]
Next weight: 3
bins: [7, 3, 0]
Next we
bins: [3
Next weight: 11
bins: [12, 11, 0]
input: [2, 1, 3, 6, 14]
cut: Wmin + 14 >= UB
bins: [6, 6, 14]
Next weight: 10
bins: [10, 3, 0]
Next weight: 14
bins: [7, 6, 0]
Next weight: 4
bins: [7, 3, 3]
Next we
bins: [3
input: [2, 1, 3, 6, 11, 11]
cut: Wmin + 8 >= UB
bins: [12, 11, 11]
Next weight: 10
bins: [10, 13, 0]
Next weight: 10
bins: [10, 3, 10]
input: [2, 1, 3, 4, 3, 14]
cut: Wmin + 14 >= UB
bins: [7, 6, 14]
Next weight: 9
bins: [11, 3, 3]
Next weight: 13
bins: [7, 7, 3]
Next weight: 4
bins: [3, 3, 8]
input: [2, 1, 3, 4, 3, 10, 10]
cut: Wmin + 9 >= UB
bins: [10, 13, 10]
input: [2, 1, 3, 4, 3, 10, 10]
cut: Wmin + 9 >= UB
bins: [10, 13, 10]
Next weight: 8
bins: [11, 12, 3]
input: [2, 1, 3, 4, 3, 4, 13]
cut: Wmin + 12 >= UB
bins: [7, 7, 16]
Next weight: 8
bins: [3, 3, 12]
Next weight: 12
bins: [7, 3, 8]
input: [2, 1, 3, 4, 3, 4, 9, 8]
cut: Wmin + 8 >= UB
bins: [11, 12, 11]
Next weight: 8
bins: [11, 3, 12]
input: [2, 1, 3, 4, 2, 2, 4, 12]
cut: Wmin + 12 >= UB
bins: [7, 15, 8]
input: [2, 1, 3, 4, 2, 2, 4, 8, 8]
cut: Wmin + 8 >= UB
bins: [11, 11, 12]

Bibliography

E. Aarts and J. K. Lenstra. *Local search in combinatorial optimization*. John Wiley & Sons, 1997.

A. Agnetis. No-wait flow shop scheduling with large lot sizes. *Annals of Operations Research*, 70: 415–438, 1997.

D. Ahr, J. Békési, G. Galambos, M. Oswald, and G. Reinelt. An exact algorithm for scheduling identical coupled tasks. *Mathematical Methods of Operations Research*, 59(2):193–203, 2004.

S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473, 1999.

S. Albers and M. Hellwig. Semi-online scheduling revisited. *Theoretical Computer Science*, 443: 1–9, 2012.

C. Alves, J. V. de Carvalho, F. Clautiaux, and J. Rietz. Multidimensional dual-feasible functions and fast lower bounds for the vector packing problem. *European Journal of Operational Research*, 233(1):43–63, 2014.

E. Angelelli, A. Nagy, M. Speranza, and Z. Tuza. The on-line multiprocessor scheduling problem with known sum of the tasks. *Journal of Scheduling*, 7(6):421–428, 2004.

Y. Azar and O. Regev. On-line bin-stretching. *Theoretical Computer Science*, 268(1):17–41, 2001.

L. Babel, B. Chen, H. Kellerer, and V. Kotov. Algorithms for on-line bin-packing problems with cardinality constraints. *Discrete Applied Mathematics*, 143(1):238–251, 2004.

J. L. Balcázar, A. Lozano, and J. Torán. The complexity of algorithmic problems on succinct instances. In *Computer Science*, pages 351–377. Springer, 1992.

J. L. Balcázar, R. Gavaldà, and O. Watanabe. Coding Complexity: The Computational Complexity of Succinct Descriptions. 1996.

N. Bansal, A. Caprara, and M. Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 697–708. IEEE, 2006.

P. Baptiste. A note on scheduling identical coupled tasks in logarithmic time. *Discrete Applied Mathematics*, 158(5):583–587, 2010.

P. Baptiste, P. Brucker, S. Knust, and V. Timkovsky. Ten notes on equal-execution-time scheduling. *4 OR*, 2:111–127, 2004.

Y. Bartal, H. Karloff, and Y. Rabani. A better lower bound for on-line scheduling. *Information Processing Letters*, 50(3):113–116, 1994.

J.-C. Billaut and F. Sourd. Single machine scheduling with forbidden start times. *4OR*, 7(1): 37–50, 2009.

J. Blazewicz, K. Ecker, T. Kis, C. Potts, M. Tanas, and J. Whitehead. Scheduling of coupled tasks with unit processing times. *Journal of Scheduling*, pages 1–9, 2010.

J. Blazewicz, G. Pawlak, M. Tanas, and W. Wojciechowicz. New algorithms for coupled tasks scheduling–a survey. *RAIRO-Operations Research*, 46(04):335–353, 2012.

M. Böhm, J. Sgall, R. van Stee, and P. Veselý. Better algorithms for online bin stretching. *arXiv:1404.5569*, 2014.

A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*, volume 53. Cambridge University Press, 1998.

F. Brandao and J. P. Pedroso. Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression. *arXiv preprint*, *arXiv:1310.6887*, 2013.

N. Brauner. Identical part production in cyclic robotic cells: Concepts, overview and open questions. *Discrete Applied Mathematics*, 156(13):2480–2492, 2008.

N. Brauner and Y. Crama. The maximum deviation just-in-time scheduling problem. *Discrete Applied Mathematics*, 134(1):25–50, 2004.

N. Brauner and C. Rapine. Polynomial time algorithms for makespan minimization on one machine with forbidden start and completion times. *Cahiers Leibniz 181*, 2010.

N. Brauner, Y. Crama, A. Grigoriev, and J. Van De Klundert. A framework for the complexity of high-multiplicity scheduling problems. *Journal of combinatorial optimization*, 9(3):313–323, 2005.

N. Brauner, Y. Crama, A. Grigoriev, and J. van de Klundert. Multiplicity and complexity issues in contemporary production scheduling. *Statistica Neerlandica*, 61(1):75–91, 2007.

N. Brauner, G. Finke, V. Lehoux-Lebacque, C. Potts, and J. Whitehead. Scheduling of coupled tasks and one-machine no-wait robotic cells. *Computers & Operations Research*, 36(2):301–307, 2009a. Scheduling for Modern Manufacturing, Logistics, and Supply Chains.

N. Brauner, G. Finke, V. Lehoux-Lebacque, C. Rapine, H. Kellerer, C. Potts, and V. Strusevich. Operator non-availability periods. *4OR*, 7(3):239–253, 2009b.

P. Brucker and S. Knust. Complexity results for scheduling problems. http://www.informatik.uni-osnabrueck.de/knust/class/.

P. Brucker, B. Jurisch, and M. Jurisch. Open shop problems with unit time operations. *Z. Oper. Res., 37(1):59-73, 1993.*

A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001.

A. Caprara, H. Kellerer, and U. Pferschy. Approximation schemes for ordered vector packing problems. *Naval Research Logistics (NRL)*, 50(1):58–69, 2003.

C. Carathéodory. Über den variabilitätsbereich der koeffizienten von potenzreihen, die gegebene werte nicht annehmen. *Mathematische Annalen*, 64(1):95–115, 1907.

J. Carlier, F. Clautiaux, and A. Moukrim. New reduction procedures and lower bounds for the two-dimensional bin packing problem with fixed orientation. *Computers & Operations Research*, 34(8):2223–2250, 2007.

S. Y. Chang, H.-C. Hwang, and S. Park. A two-dimensional vector packing model for the efficient use of coil cassettes. *Computers & operations research*, 32(8):2051–2058, 2005.

C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 185–194, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. ISBN 0-89871-434-6.

Y. Chen, A. Zhang, and Z. Tan. Complexity and approximation of single machine scheduling with an operator non-availability period to minimize total completion time. *Information Sciences*, 251:150–163, 2013.

T. Cheng, H. Kellerer, and V. Kotov. Semi-on-line multiprocessor scheduling with given total processing time. *Theoretical computer science*, 337(1):134–146, 2005.

T. Cheng, H. Kellerer, and V. Kotov. Algorithms better than LPT for semi-online scheduling with decreasing processing times. *Operations Research Letters*, 40(5):349–352, 2012.

G. Ciaschetti, L. Corsini, P. Detti, and G. Giambene. Packet scheduling in third-generation mobile systems with utra-tdd air interface. *Annals of Operations Research*, 150(1):93–114, 2007.

F. Clautiaux, C. Alves, and J. V. de Carvalho. A survey of dual-feasible and superadditive functions. *Annals of Operations Research*, 179(1):317–342, 2010.

J. Clifford. *Machine Scheduling with High Multiplicity*. PhD thesis, Ohio State University, Colombus, 1997.

J. Clifford and M. Posner. Parallel machine scheduling with high multiplicity. *Mathematical programming*, 89(3):359–383, 2001.

J. J. Clifford and M. E. Posner. High multiplicity in earliness-tardiness scheduling. *Operations Research*, 48(5):788–800, 2000.

A. Condotta and N. Shakhlevich. Scheduling patient appointments via multilevel template: A case study in chemotherapy. *Operations Research for Health Care*, 2014. ISSN 2211-6923.

S. Cosmadakis and C. Papadimitriou. The traveling salesman problem with many visits to few cities. *SIAM Journal on Computing*, 13:99–108, 1984.

Y. Crama and J. Van De Klundert. Cyclic scheduling of identical parts in a robotic cell. *Operations Research*, 45(6):952–965, 1997.

M. I. Dessouky, B. J. Lageweg, J. K. Lenstra, and v. d. S. Velde. Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica*, 44(3):115–123, 1990.

P. Detti. Algorithms for multiprocessor scheduling with two job lengths and allocation restrictions. *Journal of Scheduling*, 11(3):205–212, 2008.

P. Detti, A. Agnetis, and G. Ciaschetti. Polynomial algorithms for a two-class multiprocessor scheduling problem in mobile telecommunications systems. *Journal of Scheduling*, 8(3):255–273, 2005.

P. Detti, C. Hurkens, A. Agnetis, and G. Ciaschetti. Optimal packet-to-slot assignment in mobile telecommunications. *Operations Research Letters*, 37(4):261–264, 2009.

M. Drozdowski and M. Lawenda. Scheduling multiple divisible loads in homogeneous star systems. *Journal of Scheduling*, 11(5):347–356, 2008.

G. Dósa, M. G. Speranza, and Z. Tuza. Two uniform machines with nearly equal speeds: unified approach to known sum and known optimum in semi on-line scheduling. *Journal of Combinatorial Optimization*, 21(4):458–480, 2011.

C. Dürr. The scheduling zoo. http://www-desir.lip6.fr/~durrc/query/.

K. Eisemann. The trim problem. *Management Science*, 3(3):279–284, 1957.

F. Eisenbrand. Fast Integer Programming in Fixed Dimension. In G. Battista and U. Zwick, editors, *Algorithms - ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 196–207. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20064-2.

F. Eisenbrand and G. Shmonin. Carathéodory bounds for integer cones. *Operations Research Letters*, 34(5):564–568, 2006.

L. Epstein and R. van Stee. Lower bounds for on-line single-machine scheduling. *Theoretical Computer Science*, 299(1):439–450, 2003.

E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1): 5–30, 1996.

T. A. Feo and M. G. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

A. Fiat and G. J. Woeginger. *Online algorithms: The state of the art*. Springer Berlin Heidelberg, 1998.

C. Filippi. An approximate algorithm for a high-multiplicity parallel machine scheduling problem. *Operations Research Letters*, 38(4):312–317, 2010.

C. Filippi and A. Agnetis. An asymptotically exact algorithm for the high-multiplicity bin packing problem. *Mathematical programming*, 104(1):21–37, 2005.

C. Filippi and G. Romanin-Jacur. Exact and approximate algorithms for high-multiplicity parallel machine scheduling. *Journal of Scheduling*, 12(5):529–541, 2009.

R. Fleischer and M. Wahl. On-line scheduling revisited. *Journal of Scheduling*, 3(6):343–353, 2000.

L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

M. Gabay. Identical coupled task scheduling problem, polynomial solution for the cyclic case - empirical analysis for the finite case. Seminar of the scheduling group, Institute of computing science, Poznań University of Technology, Poland, Oct. 2011.

M. Gabay. A GRASP approach to the machine reassignment problem. Seminar of the scheduling group, Institute of computing science, Poznań University of Technology, Poland, Oct. 2012.

M. Gabay. High-Multiplicity Scheduling Problems. Seminar, Department of Quantitative Economics, Maastricht University, Netherlands, Feb. 2014.

M. Gabay and S. Zaourar. A GRASP approach for the machine reassignment problem. In *EURO 2012 - 25th European Conference on Operational Research*, Vilnius, Lituanie, July 2012.

M. Gabay and S. Zaourar. Variable size vector bin packing heuristics - Application to the machine reassignment problem. *HAL preprint hal-00868016*, Sept. 2013.

M. Gabay, G. Finke, and N. Brauner. Identical coupled task scheduling problem: the finite case. Technical report, Sept. 2011.

M. Gabay, G. Finke, and N. Brauner. Tâches couplées identiques : le cas fini. In *13ème congrès annuel de la Société française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF)*, Angers, France, Apr. 2012a.

M. Gabay, G. Finke, and N. Brauner. Identical Coupled Task Scheduling Problem: The Finite Case. In *ISCO 2012, 2nd International Symposium on Combinatorial Optimization*, Athens, Greece, Apr. 2012b.

M. Gabay, G. Finke, and N. Brauner. Scheduling of coupled tasks with high multiplicity. In *EURO 2012, 25th European Conference on Operational Research*, Vilnius, Lituanie, July 2012c.

M. Gabay, N. Brauner, and V. Kotov. Computing Lower Bounds for Online Optimization Problems: Application to the Bin Stretching Problem. *HAL preprint hal-00921663*, Dec. 2013a.

M. Gabay, N. Brauner, and C. Rapine. Complexité paramétrée pour le problème d'ordonnancement sur une machine avec instants de début et fin interdits. In *14ème congrès annuel de la Société française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF)*, Troyes, France, Feb. 2013b.

M. Gabay, N. Brauner, and C. Rapine. Parametrized complexity for single machine scheduling with forbidden start and completion times. In *EURO|INFORMS 2013 - 26th European Conference on Operational Research*, Rome, Italie, July 2013c.

M. Gabay, V. Kotov, and N. Brauner. Semi-Online Bin Stretching with Bunch Techniques. *HAL preprint hal-00869858*, Oct. 2013d.

M. Gabay, V. Kotov, and N. Brauner. Semi-Online Bin Stretching with Bunch Techniques. *Les Cahiers Leibniz*, 208:1–10, 2013e.

M. Gabay, C. Rapine, and N. Brauner. High Multiplicity Scheduling on One Machine with Forbidden Start and Completion Times. *HAL preprint hal-00850824*, Aug. 2013f.

M. Gabay, H. Cambazard, and Y. Benchetrit. A dominance criterion for packing problems. In *IFORS 2014 - 20th Conference of the International Federation of Operational Research Societies*, Bercelona, Spain, July 2014a.

M. Gabay, V. Kotov, and N. Brauner. Algorithme d'Approximation pour le Bin Stretching Semi-Online. In *ROADEF - 15ème congrès annuel de la Société française de recherche opérationnelle et d'aide à la décision*, Bordeaux, France, Feb. 2014b.

H. Galperin and A. Wigderson. Succinct representations of graphs. *Information and Control*, 56 (3):183–198, 1983.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co. New York, NY, USA, 1979. ISBN 0716710447.

M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory*, 21:257–298, 1976.

H. Gavranović, M. Buljubašić, and E. Demirović. Variable neighborhood search for google machine reassignment problem. *Electronic Notes in Discrete Mathematics*, 39:209–216, 2012. {EURO} Mini Conference.

P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

T. Gormley, N. Reingold, E. Torng, and J. Westbrook. Generating adversaries for request-answer games. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–565. Society for Industrial and Applied Mathematics, 2000.

R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45 (9):1563–1581, 1966.

R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics. v5*, pages 287–326, 1977.

F. Granot, J. Skorin-Kapov, and A. Tamir. Using quadratic programming to solve high multiplicity scheduling problems on parallel machines. *Algorithmica*, 17(2):100–110, 1997.

A. Grigoriev and J. van de Klundert. On the high multiplicity traveling salesman problem. *Discrete optimization*, 3(1):50–62, 2006.

J. Gupta. Comparative evaluation of heuristic algorithms for the single machine scheduling problem with two operations per job and time-lags. *Journal of Global Optimization*, 9(3):239–253, 1996.

B. T. Han, G. Diehr, and J. S. Cook. Multiple-type, two-dimensional bin packing problems: Applications and algorithms. *Annals of Operations Research*, 50(1):239–261, 1994.

M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial & Applied Mathematics*, 10(1):196–210, 1962.

D. Hochbaum and R. Shamir. Minimizing the number of tardy job units under release time constraints. *Discrete Applied Mathematics*, 28(1):45–57, 1990.

D. S. Hochbaum and R. Shamir. Strongly polynomial algorithms for the high multiplicity scheduling problem. *Operations Research*, 39(4):648–653, 1991.

D. S. Hochbaum, R. Shamir, and J. G. Shanthikumar. A polynomial algorithm for an integer quadratic non-separable transportation problem. *Mathematical Programming*, 55(1-3):359–371, 1992.

J. E. Hopcroft and R. M. Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

P. A. Huegler and J. C. Hartman. Problem reduction for one-dimensional cutting and packing problems. Technical report, 2002.

K. Jansen and R. Solis-Oba. An opt+ 1 algorithm for the cutting stock problem with constant number of object lengths. In *Integer Programming and Combinatorial Optimization*, pages 438–449. Springer, 2010.

N. Jussien, G. Rochart, X. Lorca, et al. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008.

R. M. Karp, M. Luby, and A. Marchetti-Spaccamela. A probabilistic analysis of multidimensional bin packing problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 289–298. ACM, 1984.

H. Kellerer and V. Kotov. An approximation algorithm with absolute worst-case performance ratio 2 for two-dimensional vector packing. *Operations Research Letters*, 31(1):35–41, 2003.

H. Kellerer and V. Kotov. An efficient algorithm for bin stretching. *Operations Research Letters*, 41(4):343–346, 2013.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

H. Kellerer, V. Kotov, and M. Gabay. A Best Possible Algorithm for Semi-Online Scheduling. *Submitted*, 2013.

A. Khanafer, F. Clautiaux, and E.-G. Talbi. Tree-decomposition based heuristics for the two-dimensional bin packing problem with conflicts. *Computers & Operations Research*, 39(1):54–63, 2012.

L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM Journal of Research and development*, 21(5):443–448, 1977.

M. Y. Kovalyov, M. Pattloch, and G. Schmidt. A polynomial algorithm for lot-size scheduling of two type tasks. *Information processing letters*, 83(4):229–235, 2002.

W. Kubiak and S. Sethi. A note on "level schedules for mixed-model assembly lines in just-in-time production systems". *Management Science*, 37(1):121–122, 1991.

W. Kubiak and S. P. Sethi. Optimal just-in-time schedules for flexible transfer lines. *International Journal of Flexible Manufacturing Systems*, 6(2):137–154, 1994.

W. Kubiak and V. Timkovsky. Total completion time minimization in two-machine job shops with unit-time operations. *European journal of operational research*, 94(2):310–320, 1996.

D. König. Graphs and matrices (in hungarian). *Mat Fiz Lapok 38*, pages 116–119, 1931.

S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, 2011.

V. Lehoux-Lebacque, N. Brauner, and G. Finke. Identical coupled task scheduling: polynomial complexity of the cyclic case. *Cahiers Leibniz 179*, 2009.

W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 404–412. IEEE, 1999.

T. Lengauer. The complexity of compacting hierarchically specified layouts of integrated circuits. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 358–368. IEEE, 1982.

T. Lengauer and E. Wanke. Efficient Solution of Connectivity Problems on Hierarchically Defined Graphs. *SIAM Journal on Computing*, 17(6):1063–1080, 1988.

H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of operations research*, pages 538–548, 1983.

R. Lopes, V. W. Morais, T. F. Noronha, and V. A. Souza. Heuristics and matheuristics for a real-life machine reassignment problem. *International Transactions in Operational Research*, 2014.

L. Lovász and M. D. Plummer. Matching theory. *Elsevier*, 1986.

A. Lozano and J. L. Balcázar. The complexity of graph problems for succinctly represented graphs. In *Graph-Theoretic Concepts in Computer Science*, pages 277–286. Springer, 1990.

S. Martello and P. Toth. *Knapsack problems*. Wiley New York, 1990a.

S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990b.

K. Maruyama, S. Chang, and D. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer & Information Sciences*, 6(2):131–149, 1977.

S. T. McCormick, S. R. Smallwood, and F. C. Spieksma. A polynomial algorithm for multiprocessor scheduling with two job lengths. *Mathematics of Operations Research*, 26(1):31–49, 2001.

R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959.

D. Mehta, B. O'Sullivan, and H. Simonis. Comparing solution methods for the machine reassignment problem. In *Principles and Practice of Constraint Programming*, pages 782–797. Springer, 2012.

D. Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

J. v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

C. Ng, Z. Tan, Y. He, and T. Cheng. Two semi-online scheduling problems on two uniform machines. *Theoretical Computer Science*, 410(8):776–792, 2009.

T. Oosterwijk, M. Gabay, A. Grigoriev, and V. J. Kreuzen. High Multiplicity Scheduling with Switching Costs for few Products. OR2014 - International Conference of the German Operations Research Society, Aachen, Germany, 2014.

A. Orman and C. Potts. On the complexity of coupled-task scheduling. *Discrete Applied Mathematics*, 72(1-2):141–154, 1997.

R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for vector bin packing. Technical report, Microsoft Research, 2011.

C. H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

C. H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71(3):181–185, 1986.

J. Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, 1982.

M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

G. M. Portal. An algorithmic study of the machine reassignment problem. Master's thesis, Universidade Federal do Rio Grande do Sul, 2013. URL http://hdl.handle.net/10183/54137.

K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In J. Y. Leung, editor, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

H. N. Psaraftis. A Dynamic Programming Approach for Sequencing Groups of Identical Jobs. *Operations Research*, 28(6):1347–1359, 1980.

C. Rapine and N. Brauner. A polynomial time algorithm for makespan minimization on one machine with forbidden start and completion times. *Discrete Optimization*, 10(4):241–250, 2013.

C. Rapine, N. Brauner, G. Finke, and V. Lebacque. Single machine scheduling with small operator-non-availability periods. *Journal of Scheduling*, 15(2):127–139, 2012.

V. Rayward-Smith and D. Rebaine. Open shop scheduling with delays. *RAIRO - Informatique théorique et applications*, 26(5):439–448, 1992.

J. Rudin and R. Chandrasekaran. Improved Bounds for the Online Scheduling Problem. *SIAM Journal on Computing*, 32(3):717–735, 2003.

T. Saber, A. Ventresque, X. Gandibleux, and L. Murphy. Genepi: A multi-objective machine reassignment algorithm for data centres. In M. Blesa, C. Blum, and S. Voß, editors, *Hybrid Metaheuristics*, volume 8457 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2014.

J. E. Schoenfield. Fast, exact solution of open bin packing problems without linear programming. *Draft, US Army Space and Missile Defense Command, Huntsville, Alabama, USA*, 2002.

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

S. Seiden, J. Sgall, and G. Woeginger. Semi-online scheduling with decreasing job sizes. *Operations Research Letters*, 27(5):215–221, 2000.

E. Selvarajah and G. Steiner. Batch scheduling in a two-level supply chain – a focus on the supplier. *European Journal of Operational Research*, 173(1):226–240, 2006.

M. Serna and F. Xhafa. Parallel approximation to high multiplicity scheduling problems via smooth multi-valued quadratic programming. *RAIRO-Theoretical Informatics and Applications*, 42(02):237–252, 2008.

H. Shachnai and T. Tamir. Approximation schemes for generalized 2-dimensional vector packing with application to data placement. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 165–177. Springer, 2003.

R. Shapiro. Scheduling coupled tasks. *Naval Research Logistics Quarterly*, 27(3):489–498, 1980.

P. Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming–CP 2004*, pages 648–662. Springer, 2004.

F. C. Spieksma. A branch-and-bound algorithm for the two-dimensional vector packing problem. *Computers & operations research*, 21(1):19–25, 1994.

G. Steiner and J. Yeomans. A linear time algorithm for maximum matchings in convex, bipartite graphs. *Computers & Mathematics with Applications*, 31(12):91–96, 1996.

G. Steiner and S. Yeomans. Level schedules for mixed-model, just-in-time processes. *Management Science*, 39(6):728–735, 1993.

W. M. Stille. *Solution Techniques for specific Bin Packing Problems with Applications to Assembly Line Optimization*. PhD thesis, TU Darmstadt, 2008.

M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and Distributed Computing*, 70(9):962–974, 2010.

M. Tanas, J. Blazewicz, and K. Ecker. Survey of scheduling of coupled tasks with chains and in-tree precedence constraints. 2011.

V. Timkovsky. Is a unit-time job shop not easier than identical parallel machines? *Discrete applied mathematics*, 85(2):149–162, 1998.

G. Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

J. Valério de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629–659, 1999.

J. A. van der Veen and S. Zhang. Low-complexity algorithms for sequencing jobs with a fixed number of job-classes. *Computers & operations research*, 23(11):1059–1067, 1996.

G. J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, 1997.

A. C.-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977.

A. C.-C. Yao. New algorithms for bin packing. *Journal of the ACM (JACM)*, 27(2):207–227, 1980.

List of Figures

List of Tables

List of Algorithms

High-Multiplicity Scheduling and Packing Problems

Abstract: High-Multiplicity encoding is a natural encoding of data. In scheduling, it simply consists in stating once the characteristics of each type of tasks and the number of tasks of this type. This encoding is very compact and natural but it is generally supposed in scheduling that all tasks are specified separately. High-Multiplicity scheduling, when considered, raises many complexity issues because of the small input size. The aim of this thesis is to provide insights on how to cope with high-multiplicity scheduling problems. We also seek to contribute to scheduling and packing in general. We expose different techniques and approaches and use them to solve specific scheduling and packing problems. We study the high-multiplicity single machine scheduling problem with forbidden start and completion times and show that this problem is polynomial with large diversity instances. We present the identical coupled-task scheduling problem and display many difficulties and issues occurring in high-multiplicity scheduling on this problem. We improve the best upper and lower bounds on the bin stretching problem. We study the vector packing problems with heterogeneous bins and propose heuristics for this problem. Finally, we present a general reduction algorithm for packing problems which can be applied in polynomial time, even with high-multiplicity encoding of the input.

Keywords: High-Multiplicity · Scheduling · Packing · Complexity · Algorithms · Forbidden Start and Completion times · Coupled-Task · Bin Packing · Bin Stretching · Vector Bin Packing · Online Algorithms

Résumé: L'encodage High-Multiplicity est un encodage naturel des données consistant, en ordonnancement, à réunir les tâches similaires et, pour chaque type, décrire les caractéristiques d'une seule tâche et le nombre de tâches de ce type. Cet encodage est très compact et lorsqu'il est considéré, pose de nombreux problème pour analyser la complexités des problèmes considérés. L'objectif de cette thèse est de proposer des techniques permettant de traiter les problèmes high-multiplicity. Nous exposons celles-ci à travers divers exemples. Nous étudions le problème d'ordonnancement high-multiplicity avec indisponibilités des opérateurs et montrons que celui-ci est polynomial lorsque le nombre de type de tâches est supérieur au nombre d'instants d'indisponibilités. Nous étudions les problème d'ordonnancement de tâches couplées identiques et montrons sur ce problème de nombreuses difficultés majeures de l'ordonnancement high-multiplicity. Nous améliorons les meilleures bornes supérieures et inférieures sur le problème de bin stretching. Nous étudions le problème de vector packing avec des récipients hétérogènes et proposons des heuristiques pour celui-ci. Enfin, nous proposons un algorithme de réduction très général pour les problèmes de placement d'objets. Cet algorithme peut être appliqué en temps polynomial en le nombre de types d'objets et de récipients.

Mot-clés: Multiplicités · Ordonnancement · Placement d'objets · Complexité · Algorithmes · Indisponibilité des opérateurs · Tâches couplées · Bin Packing · Bin Stretching · Vector Bin Packing · Algorithmes en-ligne