Systèmes interactifs pour la résolution de problèmes complexes.

Marin Bougeret

Université de Grenoble

Thèse

Pour obtenir le grade de

Docteur de l'Université de Grenoble

Spécialité : Informatique

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

Marin Bougeret

le 15 octobre 2010

Systèmes interactifs pour la résolution de problèmes complexes

---

Thèse dirigée par Denis Trystram et codirigée par Pierre-François Dutot

Jury

Pierre	Fraignaud	Directeur de recherche au CNRS	Rapporteur/Président
Maxim	Sviridenko	Chercheur à IBM Watson, NY	Rapporteur
Philippe	Baptiste	Directeur de recherche au CNRS	Examinateur
Eva	Crück	Ingénieur de recherche pour la DGA	Examinateur
Denis	Trystram	Professeur à Grenoble INP	Directeur
Pierre-François	Dutot	Maître de conférences à l'UPMF	Co-Directeur
Guochuan	Zhang	Professeur à l'Université de Zhejiang	Invité

Thèse préparée au sein du Laboratoire d'Informatique de Grenoble, dans l'École Doctorale de Mathématiques, Sciences et Technologies de l'Information, Informatique

Résumé

Cette thèse concerne l'utilisation de l'interaction entre un algorithme et un expert pour la résolution de problèmes complexes, typiquement NP-difficiles. Plusieurs définitions de l'expert sont possibles. L'objectif étant d'obtenir des algorithmes dont les performances sont garanties, ce travail est centré sur les interactions avec un expert de type "oracle", plutôt que "humain". Ainsi, on s'intéresse à des compromis entre performance, coût (typiquement temps d'exécution), et quantité d'information donnée par l'oracle. Le premier objectif de cette thèse est de comprendre quel est l'état de l'art des différentes techniques interactives dans différents domaines (algorithmique distribuée et online, complexité, optimisation combinatoire). Le second objectif est centré sur l'optimisation combinatoire, et plus particulièrement les problèmes d'ordonnancement et d'empaquetage. Nous proposons un formalisme interactif pour le contexte des problèmes d'optimisations (offline). Le but est de montrer en quoi ce formalisme facilite la conception d'algorithmes d'approximation, en le situant par rapport aux techniques classiques de conception de schémas d'approximation, et en l'utilisant pour fournir de nouveaux résultats sur des problèmes d'ordonnancement et d'empaquetage. Nous avons principalement abordé deux problèmes : le "discrete Resource Sharing Scheduling Problem ($dRSSP$)" et le problème du "Multiple Strip Packing" (MSP). Le $dRSSP$ est un problème d'hybridation d'algorithmes. Etant donné un ensemble d'algorithmes (appelé un "portfolio"), un nombre fini de ressources (des processeurs par exemple), et un ensemble représentatif d'instances (appelé "benchmark"), le but est de distribuer ces ressources aux algorithmes afin de minimiser le temps nécessaire à la résolution de toutes les instances du benchmark, en exécutant les algorithmes en parallèle selon le modèle dit du "space sharing" . Nous avons étudié l'impact de plusieurs questions à poser à l'oracle, ainsi que comment communiquer efficacement avec ce dernier (signifiant que la réponse de l'oracle est courte), aboutissant à plusieurs schémas d'approximation. Le MSP est une extension du problème célèbre du "Strip Packing" consistant à placer des rectangles dans un nombre fixé de boîtes, en minimisant la hauteur atteinte. Nous avons fourni plusieurs algorithmes/schémas d'approximation pour différentes variantes de ce problème, dans lesquelles les boîtes ont des largeurs égales/différentes, ou les rectangles doivent être placés de façon "continue" ou non (correspondant alors à un problème classique d'ordonnancement de tâches parallèles). D'une manière générale l'utilisation de l'interactivité permet d'isoler la difficulté des problèmes, et donc de les étudier différemment.

Abstract

This thesis focuses on algorithm-expert interaction for solving hard problems. Several definitions of an "expert" are possible. Our work concerns interactions with an oracle (rather than human) expert, as we are looking for theoretical performance guarantee. Thus, we are interested in tradeoffs between performance, cost (typically running time), and length of information provided by the oracle. The first objective of this thesis is to understand what is the related work and the common oracle techniques in different domains (distributed and online computing, complexity, combinatorial optimization). The second objective is centered on combinatorial optimization, and more precisely on scheduling and packing problems. We aim at showing how this interactive setting is helpful for the design of approximation algorithms, and of course to provide new results on scheduling and packing problems using these techniques. We mainly focused on two problems : the discrete Resource Sharing Scheduling Problem ($dRSSP$) and the Multiple Strip Packing (MSP). The $dRSSP$ comes from the community of hybridation of algorithms. Given a set of algorithms (often called a portfolio), a fixed amount of resources (processors for example), and a (finite) benchmark of instances to solve, the goal is to distribute the resources among the processors to minimize the cost for solving the whole benchmark, using a "space sharing" model for running the algorithms in parallel. We studied the impact of different questions to ask to the oracle, and how to communicate "efficiently" (meaning that the oracle answer is short) with the oracle, leading to several approximation schemes. MSP, which is an extension of the well known strip packing problem, consists in packing rectangles into a fixed number of strips, minimizing the height of the packing. We provided approximation schemes/algorithms for different variants of MSP where strips have equal/different widths, and where rectangles must be packed continuously or not (corresponding then to scheduling parallel jobs). It turns out that interactive techniques point out the difficulty of the problems, and are helpful to study problems in a different ways.

Remerciements

Je commence par Laila pour son support au quotidien (pour la thèse et pour le reste !), et pour m'avoir inculqué le minimum vital de confiance en soi. Je poursuis dans la famille avec mon père, qui en plus d'avoir éradiqué avec patience les horreurs grammaticales de ce manuscrit, a toujours été là pour moi pendant la dernière ligne droite*.

Je passe maintenant à mes deux chers encadrant. Non pas que je ne veuille pas vous appeler chefs (j'ai beaucoup de respect de vous, j'espère que cela se voit à des kilomètres), mais plutôt que j'ai ressenti tellement plus de bienveillance que de hiérarchie ! Merci de la liberté judicieusement bornée que vous m'avez accordée, merci de m'avoir remonté le moral dans les moments difficiles†, merci à toi Pf pour le filtre anti-pipologique permanent‡, merci à toi Denis pour toutes nos discussions musicales, et pour tes conseils paternels§. Je souhaite à des thésards d'avoir la chance d'être entre vos mains, et j'espère qu'on va continuer à travailler ensemble !

Enfin, merci aux amis du labo, à mes co-bureaux Slim (tu es d'une patience infinie, c'est bon pour la recherche), Erik (on l'a toujours pas eu ce donut), Kelly (arrête de travailler), à Marc (grâce à qui je sais dire maintenant "j'ai pas compris" dès qu'il faut), à Nicolas, Christophe, Gael .. (j'en oublie désolé), et aux chefs qui ont alimentés des débats non constructifs à la cafèt (Jean-Louis, Olivier, Greg ..) Une mention particulière à mes amis de Tours, aux parents de Laila, à Geneviève : cela m'a touché que vous soyez venus m'écouter. Et merci à celui qui me montre ce qu'est la vraie vie d'un musicien !

*une dernière ligne droite étant un laps de temps d'une durée de 8 ans (ou plus)

†vive les preuves fausses

‡et les bières à la gare

§je sais il ne faut pas faire de matching, le divan m'attend

Cette thèse a été financée par la Délégation Générale pour l'Armement (DGA) et le Centre National de la Recherche Scientifique (CNRS).

Table des matières

Chapitre 1

Introduction

Le thème de cette thèse est l'étude des systèmes interactifs pour la résolution de problèmes complexes. Les problèmes cibles sont en particulier des problèmes d'ordonnancement et d'empaquetage (ou problèmes dits de "packing"). L'idée de base est de supposer que l'on dispose d'un *expert*, pouvant répondre à certaines questions. L'objectif est alors de concevoir des algorithmes interagissant avec cet expert, en posant des "bonnes" questions pour résoudre les problèmes plus efficacement.

On s'intéresse aux algorithmes ayant des garanties de performance, typiquement sur leur temps d'exécution et la qualité des solutions qu'ils calculent. Ainsi, la fiabilité de l'expert est une des premières questions à aborder. Sans aucune hypothèse sur l'expert (comme par exemple avec un expert humain), il sera difficile de prouver que les performances d'un algorithme interactif sont meilleures que celles des algorithmes sans expert. Les travaux utilisant des expert humains, comme par exemple le "framework" classique pour la résolution interactive dénommé HuGS (pour "Human-Guided Simple Search", [Anderson et al., 2000, Klau et al., 2010]), évaluent donc généralement les algorithmes en mesurant leurs performances sur des instances "représentatives", comme par exemple dans [Lesh et al., 2005, Klau et al., 2002]. Une deuxième démarche pour profiter d'un expert non fiable concerne les problèmes d'optimisation multicritères [Alves and Clímaco, 2007, Marcotte and Soland, 1986]. Dans de tels problèmes on s'intéresse aux solutions Pareto optimales (telles qu'il n'existe pas une autre solution meilleure dans tous les critères), qui sont parfois extrêmement nombreuses. L'expert humain est alors souvent utilisé pour "naviguer" dans cet ensemble de Pareto, en choisissant interactivement les solutions qu'il préfère. Enfin, une autre approche existante pour augmenter la fiabilité des algorithmes interactifs avec humain consiste à résoudre plusieurs fois une même instance avec des experts différents. Cette technique est principalement mise en place dans des mécanismes (tels [Anderson, 2008, Von Ahn et al., 2006]) distribuant un grand nombre d'instances (par exemple l'association de mots à des images) à un grand nombre d'humains. Ce type de démarche est généralement nommé "distributed thinking", ou "human computing/computation" [Von Ahn, 2007].

Première approche de la problématique

On considérera donc dorénavant que l'expert est un oracle. Un oracle est capable de répondre correctement et en temps constant à n'importe quelle question. Notre

problématique est donc la construction d'algorithmes interactifs avec oracle ayant des garanties de performance (sur le temps de calcul et la qualité des solutions produites) sur toutes les instances. Les premières questions à aborder sont les suivantes :

- quel est le modèle d'interaction (l'algorithme pose-t-il des questions à n'importe quel instant, l'information est-elle donnée entièrement au début du calcul) ?
- quelles sont les "bonnes" informations à demander à l'oracle ?
- quel est l'intérêt de résultats obtenus avec un oracle ?

Plan

L'objectif du Chapitre 2 est de déterminer quels sont les résultats utilisant l'interactivité avec oracle existant dans les différents domaines. Nous aborderons d'abord en Section 2.2 et 2.3 les domaines de l'algorithmique distribuée et online. Nous présenterons les résultats autant que possible sous le point de vue interactif, et en gardant à l'esprit les questions mentionnées précédemment. Nous découvrirons que le modèle d'interaction est généralement simple, et que l'oracle est souvent assimilé à un ajout d'information "initial". De nombreux résultats de ces sections concernent donc des compromis entre performance de l'algorithme interactif, quantité d'information fournie par l'oracle, et coût de production d'une solution. En Section 2.4 nous présenterons brièvement les principales classes de complexité interactives et les résultats qui leurs sont associés, puisque ces résultats étudient typiquement des variations subtiles de protocoles interactifs, et nous éloignent donc de notre problématique de construction d'algorithmes interactif (avec un modèle d'interaction simple).

L'objectif du Chapitre 3 est d'aborder l'optimisation offline sous cet angle original de l'interactivité (et en particulier des compromis performance, quantité d'information, temps de calcul). La problématique est donc re-precisée : nous choisissons de restreindre l'interaction à un ajout d'information initial, et de mesurer la performance des algorithmes *via* la mesure classique de rapport d'approximation. Nous définirons donc dans les Sections 3.2 et 3.3 un formalisme approprié à ce cadre, et nous étudierons dans la Section 3.4 les liens entre les techniques interactives et les techniques classiques de conception d'algorithmes (et en particulier de schémas d'approximation). Les conclusions sur l'optimisation offline et les perspectives sont données en Section 3.5.

Dans le Chapitre 4 (et 5 pour une version en anglais) sont présentés les conclusions générales, ainsi que les nouveaux résultats obtenus sur des problèmes d'ordonnancement et d'empaquetage. Ces résultats figurent en détail dans les chapitres 6 à 12. Enfin, l'annexe située en Page 199 regroupe quelques définitions classiques de problèmes d'ordonnancement et des notions basiques de la théorie de l'approximation.

Niveau de détail

Les résultats cités dans le Chapitre 2 ne concernent pas directement les problèmes qui nous intéressent, et seront donc présentés en omettant les détails techniques, l'objectif étant plutôt de comprendre les démarches interactives mises en jeu. Les preuves rappelées dans le Chapitre 3 seront détaillées, et souvent reformulées du point de vue interactif.

Première partie

Etude méthodologique de l'utilisation d'un oracle

Chapitre 2

Tour d'horizon de l'interaction quantifiée avec expert oracle

2.1 Introduction

2.1.1 Présentation de deux démarches possibles

On considère dans tout notre travail que l'interaction s'effectue avec un expert de type *oracle*. Conformément au sens classique donné à la notion d'oracle, nous considérerons que l'oracle peut répondre correctement à n'importe quelle question en temps constant. Naturellement, les résultats utilisant un expert si puissant doivent prendre en compte à quel point l'oracle a aidé à la construction de la solution. Deux démarches principales semblent possibles pour mesurer l'aide apportée par l'oracle.

On d'une part définir une notion de *taille* de l'information donnée, et alors étudier des compromis entre la *mesure de performance* d'un algorithme interactif, la *quantité d'information* fournie par l'oracle, et le *coût* de l'algorithme (typiquement le temps d'exécution). Les résultats typiques exhibent une information permettant d'atteindre une certaine performance (résultats dits positifs), ou montrent qu'une taille minimum d'information est nécessaire pour atteindre une performance fixée (résultats dits négatifs).

On peut d'autre part étudier le nouveau problème P' consistant à "retrouver séparément l'information donnée par l'oracle". Dans ce cas on a déplacé l'étude vers le problème P' qui sera considéré à son tour de façon "classique", en fournissant un algorithme pour le résoudre ou en montrant qu'il appartient à une certaine classe de problèmes.

La deuxième démarche sera plutôt utilisée dans le Chapitre 3. En effet, pour le cas de l'algorithmique distribuée et online où l'oracle fournit souvent simplement une partie inconnue de l'instance, le problème consistant à "retrouver cette information séparément" n'a en général pas de sens, puisque par définition les parties inconnues de l'instance sont *impossibles* à retrouver.

Les résultats des Sections 2.2 et 2.3 s'inscrivent donc plutôt dans la deuxième démarche de quantification.

2.1.2 Définitions

Les notions d'oracle, de mesure de performance, quantité d'information et coût seront évidemment précisées selon les domaines et les problèmes. Par contre, nous pouvons dès à présent donner quelques définitions très générales qui seront utilisées dans les Sections 2.2 et 2.3 afin de présenter les résultats de façon unifiée. Introduisons donc les notations suivantes, qui correspondront à des définitions précises dans chacune des parties.

Notation 2.1. *On note (A, r) l'algorithme interactif composé d'un algorihme (au sens classique) A demandant une requête r à l'oracle.*

Notation 2.2. *Soit A un algorithme. On note $p(A)$ la performance de l'algorithme A. Cette notation est étendue en notant $p(A, r)$ la performance de l'algorithme interactif (A, r).*

Nous supposons que l'on souhaite minimiser la mesure de performance p, comme par exemple lorsque p est un rapport (ou ratio) d'approximation [Hochbaum, 1997].

Définition 2.3. *Un résultat positif consiste à fournir un algorithme A_0, une requête r_0 (pouvant être "vide", signifiant que A_0 est un algorithme classique, non interactif), et à majorer $p(A_0, r_0)$ (ou donner f tel que $p(A_0, r_0) \in \mathcal{O}(f)$ par exemple).*

Dans le contexte des algorithmes interactifs, un résultat négatif dépend de deux paramètres : il consiste à montrer qu'une performance n'est pas atteignable pour tout un ensemble d'algorithmes et d'oracles.

Définition 2.4. *Soit $\mathcal{A}$ un ensemble d'algorithmes, et $\mathcal{R}$ un ensemble de requêtes. Soit $\mathcal{LB}(\mathcal{A}, \mathcal{R}) = \min_{A \in \mathcal{A}, r \in \mathcal{R}} p(A, r)$. Un résultat négatif consiste à minorer $\mathcal{LB}(\mathcal{A}, \mathcal{R})$ (ou à donner f tel que $\mathcal{LB}(\mathcal{A}, \mathcal{R}) \in \Omega(f)$ par exemple).*

Bien évidemment, un résultat négatif est plus ou moins puissant selon la taille des ensembles $\mathcal{A}$ et $\mathcal{R}$.

Notation 2.5. *Nous utiliserons $\mathcal{U}$ pour désigner l'ensemble de tous les algorithmes.*

Enfin, le dernier point caractérisant un résultat concerne le type d'information utilisée. Certains résultats ne fournissent qu'un seul point dans le compromis performance - quantité d'information - coût, alors que d'autres fournissent tout un ensemble de points de ce compromis. Cette différence se justifie généralement par l'utilisation d'une information *paramétrable* ou non.

Définition 2.6. *Informellement, une information est paramétrable lorsque l'on peut contrôler (choisir) sa taille.*

2.1.3 Plan du chapitre

Les objectifs sont à présent d'étudier quels sont, dans les différents domaines, les résultats avec oracle proches de notre problématique de construction d'algorithmes interactifs ayant des performances garanties. Nous aborderons donc les différentes définitions utilisées pour la notion d'oracle, les informations typiques qui lui sont demandées, et les différentes conséquences de ces résultats avec oracle.

Les sections 2.2 et 2.3 traitent de l'utilisation d'un oracle en algorithmique distribuée et online. Ces domaines n'étant pas *a priori* notre cible principale (voir le Chapitre 3 où l'utilisation d'un oracle en optimisation offline sera abordée plus en détail), nous présenterons simplement dans ces sections des résultats liés à l'utilisation d'un oracle, en comparant autant que possible les démarches existantes. La Section 2.4 dresse à un aperçu de l'utilisation de l'interactivité en complexité, l'objectif étant de cerner les résultats proches de notre problématique.

2.2 Utilisation de l'interactivité en algorithmique distribuée

L'objectif de cette partie est de donner un aperçu de l'utilisation d'un oracle dans le contexte de l'algorithmique distribuée. Nous illustrerons quelques résultats représentatifs utilisant un oracle à travers plusieurs problèmes classiques, tels le problème de la diffusion ("broadcast") et sa variante du reveil ("wake up"), ou de la coloration.

2.2.1 Introduction

On s'intéresse aux résultats de compromis entre la mesure de performance d'un algorithme interactif, son coût, et la taille de l'information fournie par l'oracle. Il nous faut donc définir tous ces termes, ainsi que la notion de système distribué et d'algorithme distribué. L'objectif n'étant pas centré sur la problématique de l'algorithmique distribuée, nous ne définirons pas nécessairement en détail les modèles et les conditions précises d'application des travaux présentés. Nous ne détaillerons pas non plus les différents modèles de calcul distribué (selon les différentes hypothèses de synchronisation ou de primitives de communication par exemple). De nombreux ouvrages (tel par exemple [Ghosh, 2007]) traitent en détail de ce type de questions. Seront uniquement introduits (informellement) dans cette section les concepts de base de l'algorithmique distribuée et les notations correspondantes.

Définition 2.7. *On appelle système distribué un ensemble de n machines reliées par un réseau (un graphe) $\mathcal{G} = (V, E)$, avec $V = \{1, \ldots, n\}$ représentant l'ensemble des n machines (parfois appelées nœuds), et $E \subset V^2$ représentant l'ensemble des m liens entre les machines. Une arête $(i, j) \in E$ signifie que les machines i et j sont reliées dans le réseau.*

Rappelons que l'une des problématiques d'un système distribué est que les machines ne connaissent pas la totalité du graphe $\mathcal{G}$ (encore une fois sous réserve de variantes).

L'objectif est donc de définir un algorithme distribué, s'exécutant "en parallèle" sur chaque nœud, permettant aux n machines de coopérer (en faisant des calculs locaux, et en échangeant des messages) pour résoudre un problème fixé. La *mesure de performance* d'un algorithme distribué sera définie selon les exemples. Le *coût* d'un algorithme distribué est généralement défini comme le temps de calcul maximal autorisé "sur chaque nœud" (c'est à dire de calcul "local"). Cependant, ce paramètre ne sera en général pas central dans les exemples abordés, et on ne cherchera donc pas spécialement

à minimiser le temps d'exécution des algorithmes. Celui-ci sera soit illimité, soit borné par un polynôme.

Etant donné que nous considérerons à plusieurs reprises les problèmes du *broadcast* et du *wakeup*, nous allons dès à présent en donner une définition informelle (une définition exacte nécessiterait de spécifier entièrement le modèle de calcul distribué utilisé).

Définition 2.8 (Problème du *broadcast*, *wakeup*)**.** *Le problème du broadcast (ou problème de la diffusion) consiste à diffuser à l'ensemble des n nœuds du réseau $\mathcal{G}$ un message initialement détenu par un nœud (appelé source).*

Le problème du wakeup (ou problème du réveil) est similaire à celui du broadcast, avec la contrainte qu'un nœud ne peut pas émettre le message tant qu'il ne l'a pas reçu.

Enfin, nous pouvons dès à présent définir une notion d'oracle, correspondant à la majorité des travaux qui seront abordés ensuite.

Définition 2.9 (Oracle)**.** *Pour toute instance I d'un problème distribué (typiquement un graphe $\mathcal{G}$), l'oracle fournit un ensemble $r(I) = \{s_1, \ldots, s_n\}$ de chaînes de bits (présentes au début du calcul), la chaîne s_i étant fournie uniquement au nœud i.*

On assimile donc l'oracle à l'information qu'il fournit.

Notation 2.10. *La notation générale (A, r) introduite précédemment d'un algorithme interactif utilisant un oracle r correspond ici à un algorithme (distribué) A ayant pour chaque I l'information $r(I)$ répartie sur les différents nœuds.*

La définition de la taille de l'information de l'oracle sera précisée selon les exemples.

Nous allons à présent étudier différentes démarches possibles mettant en jeu la notion d'oracle. Nous nous intéresserons d'abord aux résultats positifs dans les sections 2.2.2 et 2.2.3. Nous aborderons ensuite dans les sections 2.2.4. et 2.2.5 des résultats négatifs étudiant la quantité minimum d'information à fournir pour atteindre une performance fixée. La Section 2.2.6 traite des "informative labeling scheme", qui constituent une approche légèrement différente dans laquelle le but n'est pas directement de construire un algorithme distribué, mais simplement d'étiqueter intelligemment les nœuds du réseau.

Remarque 2.11. *Il existe de nombreux résultats en algorithmique distribuée utilisant des oracles "détecteur de pannes" [Chandra and Toueg, 1996]. Chaque noeud dispose d'un détecteur local qu'il peut questionner pour avoir des informations. La quantité d'aide reçue de l'oracle correspond ici à l'appartenance de l'oracle à une catégorie donnée, les catégories les plus célèbres étant définies par les propriétés de complétude et d'exactitude (ultimement) forte / faible. Une des démarches classique consiste à trouver le détecteur le plus faible permettant de résoudre un problème, d'écrire un algorithme avec ce détecteur, puis d'implémenter le détecteur en question. Même si cette démarche semble proche de l'étude des informations non paramétrables (voir Section 2.2.2), la notion de détecteur de panne paraît difficile à réduire à un ajout d'information initial comme dans la définition 2.9, et sort donc de notre cadre. De plus, les plus célèbres utilisations de ces oracles permettent de résoudre des problèmes qui étaient auparavant impossibles (par exemple le problème du consensus en asynchrone dans [Chandra and Toueg, 1996]), et non pas d'améliorer des performances de façon "quantifiable".*

2.2.2 Application 1 : étude du gain avec une information non paramétrable

Cette partie concerne les résultats où l'on ne contrôle pas la quantité d'information demandée. Cela signifie donc que soit l'information est demandée en entier, soit elle ne l'est pas du tout.

Ce type de résultat permet de comprendre ce que l'on peut améliorer (resp. perdre) en ayant (ou non) la connaissance d'une information particulière. Les résultats visés ici n'étant pas des compromis entre performance, temps de calcul et taille d'information, il n'est donc pas nécessaire de définir la taille de l'information reçue. Les résultats de ce type consistent souvent à :

- fournir un résultat positif avec un algorithme A_0 et un oracle r_0
- fournir un résultat négatif, *i.e.* borner $\mathcal{LB}(\mathcal{A}, r_1)$, avec r_1 et $\mathcal{A}$ fixés

Ces deux points permettent donc de montrer en quoi la connaissance de r_0 est bénéfique par rapport à la seule connaissance de r_1. De nombreux travaux s'intéressent au cas $r_1 = \emptyset$ et $\mathcal{A} = \mathcal{U}$, et montrent donc en quoi la connaissance de r_0 permet d'obtenir de meilleurs résultats pour le problème considéré.

Un premier exemple concerne le problème de la diffusion dans un réseau de type radio, c'est-à-dire où chaque nœud peut lors d'une étape soit écouter, soit émettre vers tous ses voisins. Il est démontré dans [Kowalski and Pelc, 2007] que lorsque chaque nœud connaît le graphe $\mathcal{G}$, la diffusion peut être réalisée en $O(D + log^2(n))$, avec D le diamètre du graphe, la mesure d'une exécution étant le nombre d'étapes synchrones. Or, il est prouvé dans [Clementi et al., 2001] que $\mathcal{LB}(\mathcal{U}, \emptyset) \in \Omega(nlog(D))$ lorsque les nœuds ne connaissent que leur identité (autrement dit tout algorihme distribué sans oracle nécessite en pire cas $\Omega(nlog(D))$ étapes).

Des exemples courants d'informations non paramétrables (*i.e.* d'oracle r) sont par exemple le nombre de nœuds du graphe, une majoration du nombre de nœuds du graphe, ou l'orientation dans le cas où $\mathcal{G}$ est un anneau.

Remarquons que pour ce type de situations, le formalisme oracle ne semble pas forcement nécessaire, puisque l'on peut simplement considérer que l'on compare deux variantes d'un même problème, avec des hypothèses différentes. Ce type de comparaison est donc extrêmement courant (*cf* par exemple [Fich and Ruppert, 2003] où sont listés de nombreux résultats d'impossibilité), et l'on retiendra simplement que du point de vue de la comparaison entre informations, on se pose ici la question de savoir si une information "domine" une autre sur l'ensemble de tous les algorithmes.

2.2.3 Application 2 : étude du gain avec une information paramétrable

Nous allons à présent aborder un exemple tiré de [Awerbuch et al., 1990] où l'on étudie le bénéfice obtenu grâce à une information paramétrable, au sens où l'on peut demander à l'oracle une quantité plus ou moins grande d'information.

Considérons le problème du réveil. L'information $r_0(\rho)$ considérée dans [Awerbuch et al., 1990] est la connaissance du graphe sur un rayon ρ autour de chaque point. Il est prouvé dans [Awerbuch et al., 1990] que (la mesure d'une exécution étant définie comme le nombre total de messages de taille bornée)

- il existe A_0 tel que $p(A_0, r_0(\rho)) \in \mathcal{O}(\min(m, n^{1+\epsilon/\rho}))$ (où m est le nombre d'arêtes de $\mathcal{G}$ et ϵ est constant)
- $\mathcal{LB}(\mathcal{U}, r_0(\rho)) \in \Omega(\min(m, n^{1+\epsilon'/\rho}))$ (avec ϵ' constant)

On peut donc tirer plusieurs conséquences de ces deux résultats. Le résultat positif (seul) nous permet bien sûr de mesurer le gain de performance obtenu grâce à une connaissance plus ou moins importante du réseau. La borne inférieure (seule) permet de savoir si l'information r_0 peut être utile. En effet, si $\mathcal{LB}(\mathcal{U}, r_0(\rho))$ est "grand", ou diminue lentement avec ρ, alors on peut conclure que demander à rayon fixé les voisins de chaque nœud n'est pas "efficace". Enfin, les deux résultats ensemble montrent que pour tout ρ fixé, A_0 est "le meilleur" algorithme utilisant $r_0(\rho)$, ce qui correspond à une analyse de "thightness" classique.

Ce type de résultat permet donc d'avoir une idée plus fine de l'impact d'une information. Cependant, concernant les résultat négatifs, il serait intéressant d'étudier des bornes inférieures $\mathcal{LB}(\mathcal{A}, \mathcal{R})$ où $\mathcal{R}$ n'est pas seulement réduit à une seule information. En effet, nous avons vu que l'on pouvait conclure qu'une information particulière n'était pas efficace, mais il serait bien sûr plus intéressant de savoir qu'aucune information (de l'ensemble $\mathcal{R}$) ne l'est. Ainsi, un candidat naturel d'ensemble d'informations "équivalentes" sera par exemple $\mathcal{R}_x$ l'ensemble de toutes les informations de taille x, la notion de taille devant être préalablement définie. Comme nous allons le voir dans les Section 2.2.4 et 2.2.5, des résultats négatifs sur les ensembles $\mathcal{R}_x$ permettent de caractériser la difficulté d'un problème en montrant qu'un problème est insensible à l'information (ou "information insensitive" comme il est défini dans [Fraigniaud et al., 2007]), signifiant qu'il faudrait pratiquement une information aussi grande que le codage direct d'une solution optimale pour obtenir une bonne performance. Ces résultats permettent également de comparer les problèmes en évaluant la quantité d'information minimale à fournir pour atteindre une performance fixée.

2.2.4 Application 3 : "information (in)sensitivness"

Dans cette partie nous abordons la notion de problème "information sensitive", que pouvons traduire par "sensible à l'information".

Selon la définition introduite dans [Fraigniaud et al., 2007], un problème est sensible à l'information si une petite quantité d'information suffit à obtenir une performance bien meilleure que sans oracle. A l'inverse, un problème est insensible à l'information si aucune information (et aucun algorithme utilisant cette information) de taille "raisonnable" (par exemple plus petite que la taille nécessaire pour coder une solution optimale) ne permet d'atteindre une performance correcte. Ainsi, un résultat de sensibilité à l'information est prouvé grâce à un résultat positif classique, alors qu'un résultat d'insensibilité nécessite de considérer des bornes inférieures sur tous les algorithmes et toutes les informations d'une certaine taille. Ce type de bornes inférieures est donc plus général que celles présentées précédemment.

Nous allons maintenant illustrer cette notion à travers un exemple tiré de [Fraigniaud et al., 2007] : la 3-coloration distribuée d'un cycle. Dans ce problème chacun des n nœuds d'un cycle $\mathcal{G}$ doit choisir une des trois couleurs disponibles, telle que deux voisins n'aient pas la même couleur. La mesure d'une exécution est ici définie (selon le modèle $\mathcal{LOCAL}$, *cf* [Peleg, 2000a]) comme le nombre d'étapes (synchrones)

nécessaires pour trouver une coloration, chaque nœud pouvant lors d'une étape envoyer et recevoir des messages de taille non bornée, et effectuer un nombre non borné de calculs locaux. La taille d'une information donnée par l'oracle est la somme des tailles des chaînes de bits présentes sur les nœuds.

Les auteurs montrent que ce problème est insensible à l'information, en prouvant que :

- $\mathcal{LB}(\mathcal{U}, \{r | r \notin \Omega(n)\})$ n'est pas constant
- $\mathcal{LB}(\mathcal{U}, \{r | r \notin \Omega(n/log^k n)\})$ n'est pas dans $o(log^* n)$ ($log^k n$ désignant la k-ième itération de log)

Le premier de ces résultats permet de conclure qu'il n'est pas possible d'atteindre une performance constante à moins que l'oracle ne donne une information de taille comparable à celle qui code directement une 3-coloration (nécessitant $O(n)$ bits). Comme il est expliqué dans [Fraigniaud et al., 2007], le second résultat montre qu'une quantité importante d'information est nécessaire pour obtenir une performance meilleure que sans oracle, car il existe un algorithme sans oracle de coût $\Theta(log^* n)$ [Cole and Vishkin, 1986].

On voit donc le premier intérêt des résultats de bornes inférieures sur de "grands" ensembles d'informations. Remarquons que dans cet exemple seulement deux points de la courbe "quantité d'information - performance" sont nécessaires pour conclure que le problème est insensible à l'information. Il existe d'autres travaux (par exemple [Fusco and Pelc, 2008])) où les résultats positifs et négatifs sont valables pour tout un ensemble de tailles d'informations.

2.2.5 Application 4 : Comparaison de problèmes

Nous étudions dans cette partie une deuxième utilisation des bornes inférieures sur des ensembles d'informations de même taille, à travers un exemple tiré de [Fraigniaud et al., 2006] où sont étudiés les problèmes du réveil et de la diffusion dans un graphe quelconque. La mesure d'une exécution est ici définie comme le nombre total de message échangés. La taille d'une information donnée par l'oracle est définie comme la somme des tailles des chaînes de bits présentes sur les nœuds.

Les auteurs montrent les résultats suivants pour le problème du réveil :

- il existe un algorithme A_0 et une information $r_0 \in \mathcal{O}(nlog(n))$ tels que $p(A_0, r_0) \in \mathcal{O}(n)$
- $\mathcal{LB}(\mathcal{U}, \{r | r \notin \Omega(nlog(n))\}) \notin \mathcal{O}(n)$

alors que pour le problème de la diffusion,

- il existe un algorithme A_1 et une information $r_1 \in \mathcal{O}(n)$ tels que $p(A_1, r_1) \in \mathcal{O}(n)$
- $\mathcal{LB}(\mathcal{U}, \{r | r \in o(n)\}) \notin \mathcal{O}(n)$

La partie la plus facile de ces résultats concerne les résultats positifs. Par exemple r_0 est simplement un arbre couvrant (ayant pour racine le nœud source), nécessitant donc $\mathcal{O}(nlog(n))$ bits pour être codé, et A_0 utilise cet arbre pour diffuser le message de la source vers les feuilles. Notons que r_1 est en fait aussi un arbre couvrant dont le codage peut être amélioré en utilisant le fait qu'un nœud peut envoyer des messages (et donc signaler sa position dans l'arbre) avant d'avoir reçu le message détenu par la source.

Ces résultats pris séparément montrent que r_0 et r_1 sont quelque part les "meilleures" informations possibles. Ensemble, ces résultats permettent de comparer

les deux problèmes, au sens où le problème du réveil est "plus difficile" que celui de la diffusion, puisque la quantité d'information nécessaire pour atteindre une performance fixée ($\mathcal{O}(n)$) est plus grande (d'un facteur logarithmique) pour le problème du réveil.

2.2.6 Application 5 : "informative labeling schemes"

Une autre voie possible pour étudier des problèmes distribués vise à développer des méthodes et des structures permettant de stocker à moindre coût (le coût étant par exemple la taille des données stockées) des informations sur les nœuds pour rendre le réseau plus facilement "utilisable".

L'idée des "informative labeling schemes" (introduits dans [Peleg, 2000b]) est que, au lieu de simplement étiqueter les nœuds $v \in V$ entre 1 et $n = |V|$, on donne à chaque nœud v une étiquette $\lambda(v)$ (définie en connaissant le graphe en entier) permettant de déduire des informations (adjacence de deux nœuds par exemple) uniquement en regardant ces valeurs. On ne cherche donc *a priori* ici un algorithme distribué, mais uniquement une fonction λ d'attribution des étiquettes, ainsi qu'un algorithme de "décodage" A calculant l'information désirée uniquement à partir des valeurs $\lambda(v)$ de certains nœuds v. Par exemple, on peut chercher un "distance labeling scheme" (*i.e.* des étiquettes qui nous renseignent sur la distance entre les nœuds) (λ, A) tel que pour tout couple (v, v'), $A(\lambda(v), \lambda(v')) = dist(v, v')$ avec $dist(v, v')$ la distance minimale entre v et v' (ou plus faiblement tel que $1 \leq \frac{A(\lambda(v),\lambda(v'))}{dist(v,v')} \leq r$ avec r fixé).

Ainsi, trois paramètres principaux caractérisent de tels résultats (voir par exemple [Peleg, 2000b] pour plus de détails) :

- la taille de l'étiquetage $|\lambda|$ (avec par exemple $|\lambda| = \max_{v \in V}\{|\lambda(v)|\}$ avec $|\lambda(v)|$ le nombre de bits nécessaires pour étiqueter v
- le temps de calcul de A
- éventuellement le ratio d'approximation ou "stretch" (r dans l'exemple ci-dessus) pour certains problèmes de minimisation intractables.

Considérons par exemple le problème de la construction d'un "distance labeling" (présenté ci-dessus) dans un graphe quelconque. Un étiquetage trivial de taille $O(n^2)$ permettant un ratio 1 consisterait à coder sur chaque nœud la matrice d'adjacence. Ainsi, l'algorithme A de décodage serait simplement un algorithme de plus court chemin classique, puisque le codage du graphe entier serait disponible sur tous les nœuds. Les auteurs de [Peleg, 2000c] fournissent pour tout $k \in \mathbb{N}^*$ un étiquetage de taille $O(kn^{1/k}log(n)log(D))$ (où D est le diamètre du graphe) et un algorithme de décodage s'exécutant en temps polynomial permettant de calculer la distance minimale entre deux nœuds quelconques avec un ratio $r = \sqrt{8k}$.

Tout comme dans les cas précédents (voir Section 2.2.4), il existe également des résultats négatifs sur la taille minimum d'étiquetage permettant d'atteindre une performance fixée, comme par exemple dans [Gavoille et al., 2001].

Une des différences entre les "informatives labeling schemes" et les autres situations abordées est qu'ici l'oracle ne sert pas à "aider" un algorithme distribué particulier, mais à rendre le réseau lui-même plus facile à exploiter grâce aux informations données dans les étiquettes des nœuds. Cependant, ces deux situations se rejoignent parfois lorsque les "informative labeling schemes" sont utilisés par des algorithmes distribués, comme par exemple dans [Peleg, 1999] où les auteurs montrent (entre autres) comment

utiliser un "distance labeling scheme" pour construire un algorithme efficace pour le problème de "connection setup in circuit-switched network".

2.2.7 Bilan

L'ajout d'information permet souvent une analyse plus fine que celles se limitant au modèle trop pessimiste dans lequel aucune information sur le réseau n'est disponible. L'oracle est donc souvent utilisé pour obtenir des informations supplémentaires sur le réseau. Les informations paramétrables, dont on peu contrôler la taille, permettent de caractériser des compromis entre performance et quantité d'information fournie par l'oracle. Il semble que l'approche par ajout d'information soit un outil puissant, puisqu'elle reflète de nouvelles distinctions entre les problèmes. En effet, les résultats négatifs abordés (visant à déterminer la quantité minimale d'information à ajouter pour atteindre une performance fixée) montrent que certains problèmes sont insensibles à l'information, ou qu'une légère variation de quantité d'information permet de distinguer des problèmes "proches", tels les problèmes de diffusion et de réveil.

2.3 Utilisation de l'interactivité en algorithmique online

L'objectif de cette partie est de donner un aperçu de l'utilisation d'un oracle dans le contexte de l'algorithmique online, en particulier pour les problèmes d'ordonnancement et d'équilibrage de charges.

2.3.1 Introduction

Cette partie se situe dans le contexte de l'algorithmique online, dans lequel par définition l'instance n'est pas entièrement connue de l'algorithme. A chaque étape t, l'algorithme reçoit la t^{ieme} partie de l'instance $\sigma(t) \in X$ (appelée la requête t), et doit alors prendre une décision. Nous adoptons la définition suivante d'un algorithme online, tirée de [Borodin and El-Yaniv, 1998] :

Définition 2.12. *Un algorithme online (déterministe) est une suite de fonctions $g_t : X^t \to A_t$, où A_t est l'ensemble des actions possibles en réponse à la requête t.*

Remarque 2.13. *Dans la définition 2.12, on a en général $A_t = A$, pour tout t.*

Rappelons que l'on s'intéresse aux résultats de compromis entre la mesure de performance (que l'on veut minimiser) d'un algorithme interactif, son temps d'exécution, et la taille de l'information fournie par l'oracle. Dans cette section, la notion de mesure de performance d'un algorithme (éventuellement interactif) A sera classiquement le rapport (ou ratio) de compétitivité [Sleator and Tarjan, 1985].

Définition 2.14. *Un algorithme A à un ratio de compétitivité r_A (pour un problème de minimisation par exemple) s'il existe une constante $c \geq 0$ telle que pour tout instance I*

$$A(I) \leq r_A Opt(I) + c$$

où $A(I)$ désigne la valeur de la solution construite par A sur I et $Opt(I)$ désigne la valeur de l'optimal ***offline*** *de I.*

Le ratio de compétitivité est dit strict (ou absolu) quand $c = 0$.

Cela signifie que $Opt(I)$ est calculée en connaissant entièrement l'instance I, alors que l'algorithme ne connaît à chaque instant qu'une partie de I. La notion d'oracle dans cette partie (ainsi que de "taille" de l'information qu'il fournit) est plus variée que dans les exemples abordés en algorithmique distribuée, et sera détaillée selon les exemples.

Tout comme dans la Section 2.2, les résultats sont présentés par ordre croissant de généralité. En sections 2.3.2 et 2.3.3 sont abordés des résultats positifs liés à l'étude d'une information *particulière* (*i.e.* dépendant du problème), paramétrable ou non. Nous discuterons en Section 2.3.4 des difficultés liées à la définition d'une notion de taille, et à l'obtention de résultats négatifs généraux. Une possibilité pour contourner ces difficultés est alors présentée en Section 2.3.5. Nous conclurons en Section 2.3.6, en donnant également quelques variantes existantes concernant l'étude du online avec ajout d'information.

2.3.2 Application 1 : étude du gain avec une Information non paramétrable

Tout comme en algorithmique distribuée, de nombreux travaux concernent le gain lié à la présence d'une information "additionnelle". Ces résultats sont souvent regroupés sous le nom d'algorithmique *semi-online*.

Tout comme en algorithmique distribuée, les résultats de ce type consistent souvent à :

- fournir un résultat positif avec un algorithme A_0 et un oracle r_0
- fournir un résultat négatif, *i.e.* borner $\mathcal{LB}(\mathcal{A}, r_1)$, avec r_1 et $\mathcal{A}$ fixés

Ces deux points permettent donc de montrer en quoi la connaissance de r_0 est bénéfique par rapport à la seule connaissance de r_1. De nombreux travaux s'intéressent au cas $r_1 = \emptyset$ et $\mathcal{A} = \mathcal{U}$, et montrent donc en quoi la connaissance de r_0 permet d'obtenir de meilleurs résultats sur pour le problème considéré.

Par exemple, de nombreux résultats de ce type ont été fournis pour la version online du problème $P||C_{max}$ (voir en Annexe pour une introduction à cette notation classique). Ce problème consiste à ordonnancer n tâches de longueur quelconque sur m machines identiques, en minimisant la date de fin de la dernière tâche (voir plus de détails en annexe, Chapitre A). Dans sa version online, chaque tâche n'est rendue visible que lorsque la précédente a été ordonnancée. Notons que le fait de connaître la durée p_x de la tâche visible courante est déjà une information supplémentaire. On parlera dans ce cas de tâches (ou d'ordonnancements) "clairvoyantes". Rappelons que dans la mesure de compétitivité on compare l'algorithme à l'optimal offline, signifiant que l'optimal est calculée en connaissant **toute** l'instance, c'est-à-dire dans ce cas le nombre de tâches, leur taille, ainsi que leur ordre d'arrivée.

Pour un nombre de machines quelconques, et des tâches clairvoyantes, il est montré dans [Albers, 1997a] que $\mathcal{LB}(\mathcal{U}, \emptyset) \geq 1,852$. Or, il est prouvé que l'on peut atteindre un ratio de $1,725$ en connaissant la somme de la durée de toutes les tâches [Angelelli et al., 2004]. On peut également citer le cas à deux machines pour lequel la

borne inférieure de $3/2$ (de plus atteinte par l'algorithme simple de "list scheduling" [Graham, 1966]) est par exemple dépassée dans [Angelelli et al., 2003] (grâce un algorithme de ratio $4/3$) lorsque la somme des tailles des tâches et une borne supérieure sur leur taille sont connues.

De nombreux résultats utilisant une information non paramétrable existent pour beaucoup d'autres problèmes d'ordonnancement et d'équilibrage de charges. Les informations les plus courantes pour ce genre de problèmes sont entre autres : la connaissance des tailles des tâches, de celle de la plus grande, de la valeur de l'optimal, ou du nombre total de tâches. Tout comme le cas des informations non paramétrables en algorithmique distribuée, la notion d'oracle n'est pas forcément ici nécessaire, puisque l'on peut simplement considérer que l'on compare deux variantes d'un même problème. Dans ce type de situation on ne quantifie donc pas l'information donnée. Nous allons aborder dans la Section 2.3.3 quelques exemples d'étude d'une information paramétrable, où cette fois-ci tout un ensemble de situations "intermédiaires" sont considérées.

2.3.3 Application 2 : Etude du gain avec une information paramétrable

Dans le cadre de l'amélioration des bornes du online pur, une des solutions liée à l'ajout d'information est la notion de *lookahead*. L'idée d'un lookahead "de taille" l est de considérer que l'algorithme voit à chaque instant les l prochaines requêtes (ou tranches de temps suivantes), le cas $l = 1$ correspondant au contexte online pur. Ainsi, le lookahead peut être considéré comme une fenêtre "glissante" sur le futur, de taille fixée. Nous allons à présent étudier des résultats utilisant le lookahead à travers trois problèmes classiques de l'algorithmique online. Rappelons que la mesure de performance considérée est le ratio de compétitivité en pire cas. Ainsi, il nous suffit pour ces exemples de définir uniquement la performance d'un algorithme sur une instance pour définir sa performance sur le problème.

Dans le problème très classique de gestion des pages [Sleator and Tarjan, 1985], on considère un cache de taille k (c'est-à-dire un tableau à k cases), et une séquence de requêtes arrivant online. Une requête correspond à la demande d'une page $p_i \in N$. A chaque requête, deux situations sont possibles. Si p_i est déjà présente dans le cache, le coût pour servir la requête est de 0, et l'on passe à la requête suivante. Sinon, l'algorithme doit stocker p_i dans une des cases, et donc décider quelle autre page doit être effacée du cache, entraînant un coût de 1. La performance d'un algorithme sur une séquence est alors défini comme la somme des coûts pour servir toutes les requêtes de la séquence.

Il est bien connu que le ratio de compétitivité des algorithmes LRU (last recently used) et $FIFO$ est égal à k, et que ce résultat est en quelque sorte optimal étant donné que $\mathcal{LB}(\mathcal{U}, \emptyset) = k$ (voir [Sleator and Tarjan, 1985]). Comme il est remarqué dans [David and Borodin, 1990], l'utilisation "directe" du lookahead dans lequel l'algorithme, lorsqu'il doit prendre une décision pour la page p_i, voit les pages $(p_{i+1}, \ldots, p_{i+l})$, est inutile. En effet, on peut alors construire un adversaire qui, en répétant l fois chaque requête, rend ce lookahead inutile. L'auteur de [Albers, 1997b] étudie donc la notion de lookahead *fort* de taille l (noté $Lk(l)$) : lorsque l'algorithme répond à la requête p_i, il voit les pages $(p_{i+1}, \ldots, p_{i+x})$ avec $x = \min\{s > i \ t.q. \ |\{p_i, \ldots, p_s\}| = l + 1\}$. Il est

prouvé dans [Albers, 1997b] qu'une adaptation simple de LRU utilisant un lookahead fort de taille l a un ratio de compétitivité de $k-l$, et que cet algorithme est optimal au sens où $\mathcal{LB}(\mathcal{U}, Lk(l)) \geq k-l$.

On peut citer d'autres exemples typiques de résultat utilisant la notion de lookahead en ordonnancement. Rappelons que dans ces deux problèmes d'ordonnancement online, une requête (c'est-à-dire l'arrivée d'une tâche) ne doit pas nécessairement être traitée (c'est-à-dire ordonnancée) dès son arrivée. A un instant donné, on peut donc avoir un ensemble de tâches, révélées dans le passé, en attente d'être ordonnancées. L'évènement déclencheur d'une prise de décision est plutôt le début d'une période d'inactivité d'un processeur. Les auteurs de [Zheng et al., 2008] étudient le problème noté $1|Online-time, p_j = p|\sum U_i$ dans la notation classique de [Graham et al., 1979]. Dans ce problème on considère une machine, et des tâches de taille identique égale à p. Les tâches ont chacune une date d'arrivée et d'expiration, et l'on souhaite maximiser le nombre de tâches complétées avant expiration. Les auteurs considèrent un lookahead de taille l (noté $lk(l)$) défini comme la vision des lp prochaines unités de temps. Pour la variante du problème avec préemption et redémarrage (*i.e.* on peut interrompre l'exécution d'une tâche mais il faudra reprendre son calcul depuis le début), il est démontré que $\mathcal{LB}(\mathcal{U}, lk(l)) \geq \frac{\lfloor l \rfloor + 2}{\lfloor l \rfloor + 1}$. Pour la variante sans préemption, les auteurs fournissent un algorithme A_0 tel que $p(A_0, lk(1)) = 3/2$, surpassant ainsi la borne inférieur de 2 [Goldman et al., 1997] pour les algorithmes sans lookahead. Un autre lookahead classique est étudié dans [Coleman and Mao, 2002], et consiste à donner à l'algorithme la vision des l prochaines tâches (*i.e.* de leur durée).

Pour donner un peu d'intuition sur l'obtention de ces résultats, on peut dire qu'en général les résultats négatifs avec lookahead sont obtenus avec des constructions "classiques" d'adversaire. L'adversaire devant lutter contre un algorithme ayant à chaque instant i la vision des requêtes $(r_i, \ldots, r_{i+l})$ est souvent construit en décidant de la requête r_{i+l+1} (c'est-à-dire le premier évènement invisible pour l'algorithme) en fonction de la réaction de l'algorithme à la requête r_i. Les résultats positifs sont bien entendu de nature très variée. On peut cependant noter que certaines des techniques de conception d'algorithmes (technique du localement glouton, rétrospective, *etc.*) citées dans la classification de [Hochbaum, 1997] semblent s'étendre naturellement à l'utilisation d'un lookahead. On peut par exemple citer la catégorie des algorithmes dits *localement gloutons* qui à chaque instant prennent la "meilleure" décision en fonction de l'état du système et de la requête visible. Adapté avec un lookahead, les algorithmes de ce type calculent donc la meilleure décision en fonction de l'état du système et des requêtes visibles grâce au lookahead. Certains des algorithmes proposés dans les deux exemples précédents [Zheng et al., 2008, Coleman and Mao, 2002] peuvent être classés dans ce type. Etant donnés, à un instant fixé, les tâches déjà disponibles et en attente X, les tâches futures visibles Y, et l'état des machines, ces algorithmes calculent d'abord la meilleure solution σ (relativement à la fonction objectif en jeu) en considérant que toutes les tâches de $X \cup Y$ sont disponibles. Puis, si la première tâche de $X \cup Y$ devant être ordonnancée dans σ est effectivement disponible (c'est-à-dire appartient à X), alors cette tâche est ordonnancée comme dans σ. Sinon, on passe simplement à "l'instant" suivant.

On peut également remarquer que dans le compromis "coût d'un algorithme, temps d'exécution de celui-ci, et quantité d'information fournie", le critère du temps d'exécu-

tion est délaissé dans les exemples ci-dessus. En effet, pour de nombreux problèmes la non-connaissance du futur impacte bien plus l'algorithme qu'un manque de temps de calcul pour produire les solutions à chaque requête. Par exemple, les problèmes d'ordonnancement (sans précédence) visant à optimiser le nombre de tâches complétées ou la somme des tailles des tâches complétées sont souvent dans $\mathcal{P}$, ce qui suggère que la difficulté des problèmes mentionnés précédemment n'est pas dans le temps de calcul nécessaire pour construire un optimal "local" à chaque requête.

2.3.4 Difficultés liées aux résultats négatifs et à la mesure de l'information de l'oracle en online

Différents types d'informations.

Ayant maintenant grâce aux sections précédentes une première intuition des informations considérées en algorithmique online, nous allons proposer une classification des différents types d'informations utilisées.

Indépendemment du fait qu'elles soient ou non paramétrables, il semble que deux éléments caractérisent une information : son *mode de délivrance* et sa *nature*. Nous proposons donc les définitions suivantes.

Définition 2.15 (Mode de délivrance d'une information)**.** *Une information est statique (ou délivrée statiquement) si celle-ci est donnée (en une fois) à l'algorithme, au début du calcul. Une information est dynamique (ou délivrée dynamiquement) si celle-ci est donnée à l'algorithme à chaque requête.*

Remarque 2.16. *Lorsque l'on considère une information dynamique, la partie d'information délivrée à chaque requête est en général différente selon les requêtes.*

Remarque 2.17. *On pourrait encore diversifier les modes de délivrance, en imaginant par exemple que l'algorithme ne reçoive des informations que lors de certaines étapes, éventuellement choisies par l'algorithme. Autrement dit on pourrait considérer des algorithmes qui "demandent" des informations à des instants quelconques.*

Définition 2.18 (Nature d'une information)**.** *Une information est de nature "découverte" si elle sert uniquement à dévoiler une partie de l'instance future (*i.e. *si la connaissance totale de l'instance permettrait de calculer "très facilement" cette information). Sinon, on dit que l'information est de nature "combinatoire".*

Remarque 2.19. *On pourrait raffiner la définition précédente en considérant par exemple une information de nature k-découverte, si la connaissance des k prochaines requêtes permettait de calculer cette information.*

Une information combinatoire concerne typiquement une solution optimale. En ordonnancement par exemple, demander où est ordonnancée la tâche (requête) courante dans une solution optimale est une information combinatoire. L'utilisation d'une information de nature combinatoire pour des résultats positifs est très courante dans un contexte offline, car celles-ci peuvent être ensuite reconstruites, moyennant du temps de calcul (voir Chapitre 3). Elles sont par contre moins courantes en algorithmique online, car il est plus difficile de justifier leur intérêt en pratique. En effet, un résultat utilisant une information de découverte (par exemple la vision des k prochaines requêtes)

suggère une modification du système réel, qui devrait donc fournir cette information. Un résultat utilisant une information combinatoire (par exemple la connaissance du traitement optimal de la première requête) sera plus difficile à réutiliser sans oracle, puisque en pratique on ne peut pas par exemple "rejouer" l'algorithme (pour essayer tous les traitements possibles de cette première requête).

Les différentes informations classiquement utilisées sont classées selon ces définitions dans le Tableau 2.3.4.

	Statique	Dynamique
Découverte	"Semi-online classique"	"Lookahead"
Combinatoire	Ressemble à l'optimisation offline, voir Chapitre 3	Voir Section 2.3.5, ou [Emek et al., 2009]

FIG. 2.20: Résumé des principales informations utilisées en algorithme online. Chaque cas de figure peut être encore dérivé selon que l'information est paramétrable ou non.

Nous allons à présent étudier en quoi ces situations très différentes rendent difficile l'élaboration d'une définition de taille d'information.

Définition de la taille.

Commençons par rappeler en quoi définir la taille de toute information est important. La sensibilité à l'information (introduite en algorithme online, voir [Fraigniaud et al., 2007]) pose la question de la quantité d'information à ajouter afin de pouvoir résoudre efficacement un problème. Autrement dit, il s'agit d'étudier $\mathcal{LB}(\mathcal{U}, \mathcal{R}_x)$, avec $\mathcal{R}_x$ l'ensemble de toutes les informations de *taille* x. La notion de taille permet donc de définir de façon naturelle de grands ensembles d'informations "équivalentes". Un autre avantage d'une notion générale de taille (à la différence d'une taille définie uniquement pour un "lookahead" précis) serait de pouvoir comparer les problèmes selon la quantité d'information nécessaire pour atteindre des performances fixées, comme l'ont fait les auteurs de [Fraigniaud et al., 2006] pour deux problèmes classiques de l'algorithmique distribuée. Du point de vue de la comparaison des informations, on remarque que les résultats abordés dans les Sections 2.3.2 et 2.3.3 montrent qu'une information est meilleure qu'une autre sur un grand ensemble d'algorithmes (en général tous), mais que l'on ne peut pas montrer qu'une information est meilleure que *beaucoup* d'autres. Encore une fois une notion générale de taille d'information serait donc utile.

La définition naturelle de "taille" comme étant le nombre de bits nécessaires pour écrire la chaîne considérée s'adapte bien aux informations statiques. Cependant, les extensions naïves de cette mesure aux informations dynamiques ne sont pas satisfaisantes. Par exemple, définir la taille comme le nombre maximal de bits nécessaires pour écrire (lors d'une requête) le morceau de l'information dynamique considéré semblerait "injuste", au sens où une information dynamique serait beaucoup plus puissante qu'une information statique de même taille. On ne peut non plus sommer (sur toutes les requêtes) les bits utilisés pour écrire l'information dynamique, puisque l'on obtiendrait

dans de nombreux cas (au moins) la taille totale de l'instance. De plus, si l'on considère par exemple une information $i(k)$, étant la connaissance de la plus grande tâche parmi les k prochaines, on remarque que l'obtient une information statique, mais qui concerne une tranche de futur plus ou moins grande. Il faudrait donc quelque part une mesure qui soit croissante en l'étendue du futur concerné (ici k). En effet, il faudrait pouvoir comparer un résultat utilisant $i(1)$ et $i(n)$, où n est le nombre total de requêtes (remarquons que de tels résultats n'ont pas besoin d'inclure le codage de la valeur k).

En résumé, la définition générale de taille sert à établir des résultats négatifs plus puissants, et donc à prouver par exemple qu'une information particulière est la meilleure parmi un large ensemble d'autres informations. Il est important de trouver une définition "équitable" des tailles, puisque dans le cas contraire une borne inférieure sur les informations de même taille pourrait être très mauvaise, puisque cet ensemble contiendrait des informations de "puissance" très variées. Il semble non trivial de définir une notion de taille prenant en compte ces différents types d'information.

Nous allons à présent aborder des résultats récents ([Emek et al., 2009]) qui sont inscrits dans cette démarche de quantification générale de l'information et conséquemment d'obtention de résultats négatifs.

2.3.5 Application 3 : Etude d'une information combinatoire dynamique

La question de la définition générale de taille de l'information est discutée dans [Emek et al., 2009]. Les auteurs considèrent des informations combinatoires dynamiques, en imposant de donner exactement la même quantité d'information à chaque requête. La définition d'un algorithme interactif est donc la suivante.

Définition 2.21 ([Emek et al., 2009])**.** *On étend la définition 2.12 en utilisant les mêmes notations. Un algorithme online (déterministe) avec ajout d'information est une suite de paire de fonctions g_t, u_t, avec $g_t : X^t \times Y^t \to A_t$ et $u_t : X^* \to Y$.*

Etant donnée une instance $\sigma = (\sigma(1), \ldots, \sigma(n)) \in X^n$, l'action choisie par l'algorithme à l'étape t est $g_t(\sigma(1), \ldots, \sigma(t), u_1(\sigma), \ldots, u_t(\sigma))$.

Les auteurs choisissent un modèle dans lequel $|Y| = 2^b$ (avec b fixé). Ainsi, à chaque étape l'oracle peut donner b bits d'informations quelconques, et on retrouve bien une information paramétrable combinatoire dynamique.

Les auteurs fournissent dans ce cadre de travail des résultats positifs et négatifs pour deux problèmes classiques de l'algorithmique online : le problème du *metrical task system (MTS)* et du *k-server*. Deux des principaux résultats sont les suivants.

Théorème 2.22 ([Emek et al., 2009])**.** *Pour le problème du "n node uniform MTS", pour tout b tel que $1 \leq b \leq \Theta(log(n))$,*

$$\mathcal{LB}(\mathcal{A}, R_b) \in \Omega(\frac{log(n)}{b})$$

*avec $\mathcal{A}$ l'ensemble des algorithmes randomisés, et R_b l'ensemble des informations codables en b bits à chaque étape (*i.e. *avec $|Y| = 2^b$).*

Remarque 2.23. *La mesure de performance des algorithmes online randomisés étant l'espérance du ratio de compétitivité, le théorème 2.22 signifie donc que tout algorithme randomisé à en espérance un ratio dans* $\Omega(\frac{log(n)}{b})$.

Théorème 2.24 ([Emek et al., 2009])**.** *Pour le problème du "general n node MTS", pour tout b tel que* $1 \leq b \leq log(n)$*, il existe un algorithme déterministe* A_0 *de ratio* $\mathcal{O}(\frac{log(n)}{b})$.

Remarquons qu'un résultat positif dans ce cadre reste "classique", puisqu'il consiste à exhiber une information particulière. Par contre, notons que le Théorème 2.22 est un des rares résultats donnant une borne inférieure sur de larges ensembles d'informations en algorithmique online. De plus, cette borne est valide pour des algorithmes randomisés, et ce même lorsque tous les bits d'information sont donnés au début du calcul (*i.e.* en statique).

Ainsi, même si cette définition de la taille ne règle pas les problèmes mentionés en Section 2.3.4 (puisque certes on peut donner une information combinatoire quelconque, mais on est obligé de fournir la même quantité d'information à chaque étape), ce résultat original fournit la forme la plus "puissante" de borne inférieure, à savoir pour n'importe quel algorithme et n'importe quelle information combinatoire statique de taille fixée.

2.3.6 Bilan

Nous avons passé en revue quelques exemples dans lesquels l'utilisation d'un oracle (ou autrement dit l'ajout d'information) permet d'améliorer les bornes du online pur, ou d'obtenir une analyse plus fine. La définition d'une notion de "taille" est moins naturelle qu'en algorithme distribuée, et rend donc difficile l'étude des notions liées à la sensibilité à l'information.

Enfin, notons que bien d'autres solutions (que nous ne pourrons pas toutes mentionner ici) ont été envisagées pour se démarquer de l'analyse compétitive d'un algorithme online pur contre un adversaire offline. Certaines de ces solutions consistent à changer la mesure des algorithmes en contraignant l'optimal, en restreignant l'ensemble des instances considérées, en abandonnant la mesure de pire cas, en s'autorisant un buffer pour mettre en attente des requêtes. On pourra se référer par exemple à [Dorrigiv and López-Ortiz, 2005] où de nombreuses mesures sont présentées, [Koutsoupias and Papadimitriou, 2001] pour la notion de "comparative analysis", ou [Englert et al., 2008] pour un exemple de "buffer reordering", puisque ces considérations nous éloigneraient trop de notre problématique liée à l'ajout d'information.

Concernant les mesures liées à la quantité d'information disponible, on peut citer [Kirkpatrick, 2009] dans lequel les auteurs changent de mesure en restreignant la quantité d'information connue par l'adversaire. Ainsi, au lieu d'ajouter de l'information à l'algorithme et de le comparer à un adversaire online, on compare un algorithme online à un adversaire semi-online.

2.4 Utilisation de l'interactivité en complexité

2.4.1 Introduction

De tous les domaines abordés, la complexité est certainement celui où l'utilisation d'oracles est la plus classique. En effet, les deux démarches d'utilisation d'un résultat avec oracle (présentées en Section 2.1.1) semblent correspondre à des concepts très classiques en complexité. Etudier le nouveau problème consistant à retrouver séparément l'information donnée par l'oracle correspond à l'idée d'une réduction (de Karp, Turing [Garey and Johnson, 1979]). D'autre part, la démarche consistant à quantifier l'information apportée par l'oracle rappelle plutôt des définitions de classe du type NP, dont l'une des formes de définition est l'ensemble des langages reconnus par algorithme (déterministe) vérifiant une preuve de taille polynomiale.

Bien évidemment, le contexte est différent de celui de l'algorithmique distribuée, online, ou de l'optimisation offline, étant donné que les problèmes considérés dans le domaine de la complexité sont des problèmes de décision. On ne retrouve donc pas *a priori* directement une situation dans laquelle le but est d'ajouter de l'information pour améliorer une borne de performance. Les caractéristiques d'un protocole interactif sont en général plutôt définies *via* les notions de "completness" et "soundness", *i.e.* la capacité (typiquement définie par une probabilité) à accepter un mot lorsque celui-ci est effectivement dans le langage, et la capacité à le refuser lorsqu'il n'y appartient pas.

Nous présentons succinctement en Section 2.4.2 différents modèles de protocoles interactifs, puisque ces variantes s'éloignent du modèle restreint où l'interaction s'effectue (avec un seul oracle) au début du calcul que nous avons considéré dans les Sections 2.2, 2.3 et qui sera envisagé au Chapitre 3. La Section 2.4.3 aborde plus en détail un des modèles interactif (le modèle $\mathcal{PCP}$), car certains des résultats liés à cette classe ont un lien plus proche avec notre problématique.

Remarque 2.25. *On ne parle pas ici de la hiérarchie polynomiale, dans laquelle les classes disposent d'un oracle de plus en plus puissant, contrairement au cadre général de ce document où l'oracle est considéré comme "tout puissant".*

2.4.2 Présentation de modèles interactifs classiques

Cette partie se base entre autres sur le cours [Goubault-Larrecq,] et sur l'article [O'Donnell, 2005]. Depuis la définition interactive de $\mathcal{NP}$, d'autres protocoles interactifs ont été définis. Ces protocoles peuvent être regroupés sous le nom de "systèmes de preuve interactifs", à ne pas confondre toutefois avec une des classes interactives dénommée $\mathcal{IP}$ (pour "interactive proof"). De tels systèmes sont définis par la donnée de deux parties : un prouveur (l'oracle) et un vérifieur (randomisé ou non). Etant donnée une assertion, l'objectif du prouveur est de convaincre le vérifieur de sa validité, alors que l'objectif du vérifieur est de n'accepter que des assertions correctes. La notion de preuve est parfois considérée comme un objet "statique", *i.e.* une chaîne de caractère (comme dans $\mathcal{NP}$ ou $\mathcal{PCP}$), ou comme un processus impliquant un dialogue entre prouveur et vérifieur.

Il existe des travaux [Goldin and Wegner, 2008] sur l'interaction en complexité qui ne s'inscrivent pas dans ce type de modèle prouveur-vérifieur, mais dont l'objectif est de

définir un nouveau modèle de calcul (les PTM, pour "Persistent Turing Machines") qui correspondrait mieux aux "vraies" situations interactives de calcul. Un exemple de telle situation est donné dans [Eberbach et al., 2004] avec le "driving home problem", qui consiste à créer un pilote automatique capable de conduire la voiture entre deux points fournis au système en tant que paramètre d'entrée. Les auteurs concluent qu'il n'y a pas de fonction calculable qui prendrait en entrée une quantité *finie* d'information et qui donnerait la succession des commandes à effectuer pour conduire en sécurité (puisque par exemple il faudrait donner assez de détails en entrée pour pouvoir prédire les paramètres physiques extérieurs, la position des piétons, *etc.*). Or, ce problème est "calculable", en utilisant un mécanisme recevant en continu un flux vidéo représentant la route, et dirigeant en conséquence les roues et les freins. Ce calcul est donc *interactif* car les entrées et les sorties sont générées *pendant* le calcul (les entrées du présent dépendant des sorties passées). Ainsi, les auteurs distinguent le calcul interactif du calcul classique sur les points suivants :

- le calcul est vu comme un processus continu plutôt qu'une transformation finie d'entrée en sortie
- l'entrée est un flux (infini éventuellement) de "jetons" générés dynamiquement plutôt qu'une entrée de taille finie
- tous les "jetons" ne sont pas présents initialement (*i.e.* certains jetons dépendent de tous les jetons et sorties précédentes), alors que toute l'entrée est disponible au début dans le calcul classique

Le modèle décrit par les PTM correspond donc à cette vision du calcul interactif, et il est montré dans [Goldin et al., 2004] que ce modèle est "plus expressif" que les machines de Turing classiques. Ce type de travaux ne correspond donc pas à notre problématique dans laquelle la situation de calcul est "classique" (en leur sens). Nous allons donc plutôt présenter des modèles interactifs correspondant au contexte prouveur-vérifieur introduit précédemment.

Modèles avec preuve statique / Interaction au début du calcul

Nous allons à présent aborder des modèles interactifs classiques, dont le plus basique est le vérificateur de la classe $\mathcal{NP}$.

Définition 2.26 ($\mathcal{NP}$ Vérifieur)**.** *Un $\mathcal{NP}$ vérifieur V est une machine de Turing déterministe (non randomisée) qui pour toute entrée x de taille n, et toute preuve y (chaîne de caractères) de taille polynomiale en n, calcule un booléen $V(x, y)$ en temps polynomial en n.*

On définit alors $\mathcal{NP}$ comme suit :

Définition 2.27. *La classe $\mathcal{NP}$ est l'ensemble des langages L tels qu'il existe un $\mathcal{NP}$ vérifieur V et un polynôme p tels que*

- *pour tout $x \in L$, il existe y tel que $|y| \leq p(n)$, $V(x, y) = V$*
- *pour tout $x \notin L$, quel que soit y tel que $|y| \leq p(n)$, $V(x, y) = F$*

On remarque donc que la preuve est ici est un objet "statique", fournie dès le début du calcul.

Une extension [Arora and Safra, 1998] possible est l'introduction de l'aléa dans le vérifieur.

Définition 2.28 ($\mathcal{PCP}(R(n), Q(n))$ Vérifieur)**.** *Un $\mathcal{PCP}(R(n), Q(n))$ vérifieur V est une machine de Turing déterministe randomisée qui pour toute entrée x de taille n, et toute preuve y (chaîne de caractères) de taille polynomiale en n*

- *tire au hasard une chaîne τ de longueur $r = \mathcal{O}(R(n))$*
- *calcule $k = \mathcal{O}(Q(n))$ positions $p_1(x, \tau), \ldots, p_k(x, \tau)$ en temps polynomial en n*
- *lit les bits $p_1(x, \tau), \ldots, p_k(x, \tau)$ de la preuve y et calcule un booléen $f(p_1(x, \tau), \ldots, p_k(x, \tau)) = V(x, y, \tau)$ en temps polynomial en n*

On définit alors $\mathcal{PCP}(R(n), Q(n))$ comme suit :

Définition 2.29. *La classe $\mathcal{PCP}(R(n), Q(n))$ est l'ensemble des langages L tels qu'il existe un $\mathcal{PCP}(R(n), Q(n))$ vérifieur V et un polynôme p tels que*

- *pour tout $x \in L$, il existe y tel que $\Pr_\tau[V(x, y, \tau) = V] = 1$*
- *pour tout $x \notin L$, quel que soit y, $\Pr_\tau[V(x, y, \tau) = V] < \frac{1}{2}$*

Par définition on a donc $\mathcal{NP} = \mathcal{PCP}(0, poly(n))$, et $\mathcal{PCP}(R(n), Q(n)) \subset \mathcal{DTIME}(2^{2^{\mathcal{O}(R(n))}Q(n)} poly(n))$ puisque sur toutes les exécutions possibles le $\mathcal{PCP}(R(n), Q(n))$ vérifieur ne lit au plus que $2^{\mathcal{O}(R(n))}Q(n)$ bits de preuve. Rappelons que $\mathcal{DTIME}(t(n))$ désigne l'ensemble des langages reconnus par une machine de Turing déterministe en temps $t(n)$ [Garey and Johnson, 1979].

Nous verrons en Section 2.4.3 certains des principaux résultats sur ces classes, qui montrent en particulier que l'on peut capturer $\mathcal{NP}$ en utilisant un nombre constant de bits de preuve, et seulement $\mathcal{O}(log(n))$ bits d'aléa. Nous verrons également les conséquences des résultats de compromis entre $R(n)$, $Q(n)$ et la probabilité d'acceptation sur l'inapproximabilité de certains problèmes classiques.

Modèles avec processus de preuve / Dialogue entre les parties

Plutôt que d'utiliser une preuve "statique", on peut envisager des protocoles utilisant un "dialogue" entre les deux parties. L'objectif de cette section est de donner les définitions de certains protocoles de ce type, afin de comprendre quelles variations sont envisagées (nombre de tours de dialogue, tirages aléatoires publics/privés, prouveurs multiples), et quels sont certains des principaux résultats sur ces classes interactives.

La classe $\mathcal{IP}$ (pour "interactive proof"), introduite dans [Goldwasser et al., 1985], utilise un dialogue pour vérifier l'appartenance d'un mot via le protocole suivant. Dans ce type de modèle, un "tour d'interaction" signifie le calcul d'une question q_i en utilisant des bits d'aléa, et la réponse de l'oracle r_i. Ainsi, "x tours d'interaction" signifie que (pour $i \in \{1, \ldots, x\}$)

- chaque q_i dépend de tous les (q_j, r_j), pour $j < i$
- chaque r_i dépend de tous les (q_j, r_j), pour $j < i$, ainsi que de q_i

Tout "l'historique" de la discussion est donc disponible à chaque tour, **excepté** les bits d'aléa utilisés par le vérifieur, qui ne sont jamais communiqués au prouveur. Informellement, l'aléa sert à se protéger des prouveurs malveillants, et ne doit donc pas être révélé.

Définition 2.30. *Un système de preuve interactif (de type $\mathcal{IP}$) pour un langage L est un protocole à deux parties (un prouveur et un vérifieur) prenant une entrée x (fournie aux deux parties) tel que*

- *le vérifieur est une machine de Turing randomisée, s'exécutant en temps polynomial (en la longueur de l'entrée commune)*
- *pour toute entrée x*
  - *si $x \in L$, il existe une stratégie du prouveur tel que (après un certain nombre de tours d'interaction) le vérifieur accepte toujours l'entrée x*
  - *si $x \notin L$, alors quelle que soit la stratégie du prouveur le vérifieur accepte avec une probabilité inférieure à p (par exemple $\frac{1}{2}$)*

Informellement, l'ajout de l'aléa dans la définition précédente est essentiel, sans quoi le prouveur pourrait deviner toutes les questions du vérifieur, et donc donner directement une preuve "statique". On peut alors définir une hiérarchie de classes, en limitant le nombre de tours possibles.

Définition 2.31. *On note $\mathcal{IP}$ l'ensemble des langages ayant un système de preuve interactif. On note $\mathcal{IP}(r(.))$ l'ensemble des langages ayant un système de preuve interactif tel que pour toute entrée x, au plus $r(|x|)$ tours sont utilisés.*

Il existe un autre protocole très proche introduit dans [Babai, 1985], appelé "Arthur-Merlin" (Arthur étant le vérifieur et Merlin le prouveur) et noté $\mathcal{AM}$, dans lequel la seule différence avec le protocole $\mathcal{IP}$ est que le prouveur voit les bits d'aléas utilisés par le vérifieur (*i.e.* ces bits d'aléas sont ajoutés dans les questions posées au prouveur). On peut donc de même définir $\mathcal{AM}(r(.))$ de la même façon que pour $\mathcal{IP}$. On a donc par exemple par définition $\mathcal{NP} \subset \mathcal{IP}(1)$, puisque en un tour d'interaction dans $\mathcal{IP}$ on peut effectivement vérifier une preuve de taille polynomiale. On a également $\mathcal{AM}(r(.)) \subseteq \mathcal{IP}(r(.))$, puisqu'il suffit que le vérifieur inclue les tirages aléatoires dans les questions posées au prouveur.

Nous allons citer quelques résultats célèbres sur ces classes. Rappelons que $\mathcal{PSPACE}$ désigne l'ensemble des langages reconnus par une machine de Turing déterministe utilisant un espace au plus polynomial en la taille de l'instance. La théorème $\mathcal{IP} = \mathcal{PSPACE}$ prouvé par Shamir [Shamir, 1992] montre donc la puissance de la classe $\mathcal{IP}$. Il est également établi que pour toute **constante** k, $\mathcal{IP}(k) \subset \mathcal{AM}(k+1)$, ainsi que $\mathcal{AM}(k) \subseteq \mathcal{AM}(1)$ (résultat démontré par Babai dans [Babai, 1985]). Ce dernier résultat montre que "la hiérarchie Arthur-Merlin" s'effondre, puisque tout langage pouvant être décidé avec k tours entre Arthur et Merlin peut en fait être décidé en un seul tour. D'autres classes interactives ont été étudiées, comme par exemple MIP, dans laquelle plusieurs prouveurs "indépendants" interagissent avec un vérifieur. Même si ils sont très puissants, ces résultats ne nous concernent donc pas directement puisqu'ils mettent en jeu des variations du modèle d'interaction, qui sont plus complexes que celui considéré dans le reste du manuscrit (qui utilise simplement une preuve "statique", et sans aléa). Nous allons donc en Section 2.4.3 plutôt donner des détails sur le modèle $\mathcal{PCP}$.

2.4.3 Quelques résultats sur le modèle $\mathcal{PCP}$

Cette partie se base entre autres sur l'article de [Trevisan, 2004]. Un des résultats les plus importants concernant le modèle $\mathcal{PCP}$ est le suivant.

Théorème 2.32 (Théorème $\mathcal{PCP}$, [Arora, 1995, Dinur, 2007])**.**

$$\mathcal{PCP}(log(n), 1) = \mathcal{NP}$$

Ce théorème montre donc que l'utilisation de $\mathcal{O}(log(n))$ bits d'aléa permet de ne lire qu'un nombre constant de bits de preuve ! Remarquons que le sens $\mathcal{PCP}(log(n), 1) \subseteq \mathcal{NP}$ est trivial. En effet, pour tout mot, le nombre de bits de preuves examinés dans $\mathcal{PCP}(log(n), 1)$ est au plus $2^{\mathcal{O}(log(n))}$, ce qui constitue donc une preuve de taille polynomiale en n suffisant à décider (au sens de $\mathcal{NP}$) de l'appartenance d'un mot.

Nous allons à présent aborder un des liens importants entre le théorème $\mathcal{PCP}$ et l'inapproximabilité. Le problème $MAX-3SAT$ dénote le problème classique de maximisation du nombre de clauses satisfaites "simultanément" dans une formule $3-CNF$ (*i.e.* sous forme normale conjonctive).

Théorème 2.33 ([Arora, 1995]). *Théorème* $\mathcal{PCP} \Longrightarrow \exists \epsilon > 0$ *tel que approximer* $MAX-3SAT$ *avec un ratio* $(1+\epsilon)$ *est* $\mathcal{NP}hard$.

Nous allons plutôt prouver le lemme suivant, pour montrer l'importance des paramètres des classes $\mathcal{PCP}$.

Lemme 2.34. *Supposons que l'on sache approximer* $MAX-3SAT$ *avec un ratio* $(1+\epsilon)$ *en un temps* $t(\epsilon, x)$*, où* x *est le nombre de clauses de la formule. Alors, on peut décider* $\mathcal{PCP}(f(n), g(n))$ *en temps*

$$t\left(\frac{1}{(ag(n)-1)2^{ag(n)+1}-1}, \mathcal{O}(2^{b(f(n)+g(n))}g(n))\right) poly(n)$$

où $poly$ *est un polynôme et* a, b *des constantes.*

Preuve Cette preuve est tirée de [Arora, 1995], dans lequel elle y est directement écrite pour $f(n) = log(n)$ et $g(n) = 1$. Soit $L \in \mathcal{PCP}(f(n), g(n))$. Soient c et d tels que le nombre de bits aléatoires utilisés par le vérifieur (pour toute exécution) soit inférieur à c, et le nombre de bits de preuve lus soit inférieur à d. Soit R l'ensemble des chaînes binaires de longueur c. Pour chaque mot aléatoire $r \in R$ fixé, le vérifieur calcul au plus d positions (dans la preuve) $(y_1^r, \ldots, y_d^r)$, et on peut alors définir une fonction booléenne de d bits $f_r(x_1, \ldots, x_d)$ valant V *ssi* le vérifieur accepte lorsque $(y_1^r, \ldots, y_d^r) = (x_1, \ldots, x_d)$.

Pour chaque mot x fixé, on sait donc que si $x \in L$ alors il existe une preuve y satisfaisant toutes les $f_r, r \in R$, et que sinon pour tout y au moins la moitié des f_r sont fausses. On représente les f_r sous forme d'une conjonction de 2^d clauses $C_r^i, i \in \{1, \ldots, 2^d\}$ de tailles d (*i.e.* avec d littéraux). La formule $d-CNF$

$$\bigwedge_{r=1}^{2^c} \bigwedge_{i=1}^{2^d} C_r^i$$

est donc satisfiable *ssi* $x \in L$. Si $x \notin L$, tout assignement de valeur rend faux au moins une fraction de $\alpha = \frac{1}{2}\frac{1}{2^d}$ clauses (au moins une clause de fausse pour la moitié des f_r).

En appliquant les transformations standard pour réécrire chaque clause C_r^i de taille d sous la forme de $d-2$ clauses de taille 3, on obtient une formule sous forme $3-CNF$ ayant $2^c 2^d(d-2)$ clauses. Dans cette formule la fraction de clauses fausses devient $\alpha = \frac{1}{d-2}\frac{1}{2^{d+1}}$. Ainsi, le rapport en les valeurs (selon que $x \in L$ ou non) de la fonction objectif de cette instance de $MAX-3SAT$ est donc au moins de $\frac{1}{1-\alpha} = 1+\epsilon$

avec $\epsilon = \frac{1}{(d-1)2^{d+1}-1}$. □

Le Lemme 2.34 montre l'importance du compromis entre $f(n), g(n)$ et la probabilité d'acceptation. Ainsi, si on "capture" une grande classe X avec $\mathcal{PCP}(f(n), g(n))$, avec f et g "petites" (et une probabilité de $\frac{1}{2}$), et que X est supposée être "difficile" à décider, alors $t(\epsilon, x)$ devrait être grand. Remarquons que plus $f(n)$ et $g(n)$ sont petites, plus le fait que $t(\epsilon, x)$ soit grand est une "mauvaise" nouvelle. En effet, ϵ est décroissant en $g(n)$ et x est croissant en $g(n)$ et $f(n)$, et donc $t(\epsilon, x)$ devrait être petit lorsque la précision est grande et le nombre de clauses est petit.

Ce lemme implique directement le Théorème 2.33, puisque en considérant un $L \in \mathcal{NP} \subseteq \mathcal{PCP}(log(n), 1)$, la réduction proposée est polynomiale (et le nombre de clauses créées aussi), et approximer à $1 + \epsilon > 0$ le problème $MAX - 3SAT$ permet donc de décider L.

Ainsi, même une fois le théorème $\mathcal{PCP}$ établi, certains résultats ont été dédiés à "l'optimisation" des paramètres du vérifieur. Comme il est rappelé dans [Trevisan, 2004], la construction dans [Håstad, 2001] fournit par exemple un vérifieur $\mathcal{PCP}$ (légèrement différent de la définition 2.28) utilisant exactement 3 bits d'aléa (et ayant de plus un comportement très "simple"), et implique donc que approximer $MAX - 3SAT$ à $\frac{8}{7}$ est $\mathcal{NP}hard$.

L'inapproximabilité de $MAX - 3SAT$ grâce au théorème $\mathcal{PCP}$ a entraîné de nouveaux résultats d'inapproximabilité pour d'autres problèmes classiques (par exemple "vertex cover" et "independent set", voir [Trevisan, 2004]). De plus (voir [Arora, 1995] p. 90), la complétude de $MAX - 3SAT$ au sein de la classe $\mathcal{MAXSNP}$ pour la réduction appelée "approximation preserving reduction" a entraîné l'inapproximabilité à un facteur constant de tous les problèmes $\mathcal{MAXSNP}hard$.

Il faut enfin citer [Feige et al., 1996], qui est un autre lien célèbre entre les classes $\mathcal{PCP}$ est l'inapproximabilité. La construction dans cet article établit un lien direct avec entre un vérifieur $\mathcal{PCP}$ et le problème "max clique". Cette réduction s'applique sur la classe $\mathcal{PCP}_{c,s}(f(n), g(n))$, où c et s sont les probabilités d'acceptation et de refus du vérifieur (nous avons donc $c = 1$ et $s = \frac{1}{2}$ dans la définition 2.29). Informellement, pour tout mot x potentiellement dans L, avec $L \in \mathcal{PCP}_{c,s}(f(n), g(n))$, on crée un graphe (en temps $\mathcal{O}(2^{\mathcal{O}(f(n)+g(n))})$) G_x en simulant le vérifieur sur x avec tous les tirages aléatoires et toutes les réponses possibles. Si $x \in L$, alors G_x a un "independent set" de taille au moins $c2^{\alpha r(n)}$ (avec α constant). Autrement tout "independent set" a une taille bornée par $s2^{\alpha r(n)}$. Ainsi, une $\frac{c}{s}$-approximation pour le problème "independent set" permet de résoudre tout problème dans $\mathcal{PCP}_{c,s}(f(n), g(n))$ en temps polynomial en n et $2^{\mathcal{O}(f(n)+g(n))}$. On voit donc la encore l'intérêt de capturer de "grandes" classes de problèmes avec des classes $\mathcal{PCP}$ ayant des paramètres f et g "petits". En particulier, cette réduction combinée au théorème $\mathcal{PCP}$ (et à d'autres outils très techniques) a donné un résultat important sur l'inapproximabilité du problème "max clique".

2.4.4 Bilan

La définition d'un protocole interactif (et donc de la classe de problèmes que ce protocole reconnaît) sert entre autres à caractériser de nouvelles classes de problèmes (puis à comprendre les liens avec les autres classes, éventuellement non interactives), et

donc à mieux appréhender la puissance de différents modèles de calcul. Les variations entre les différents protocoles interactifs concernent par exemple le nombre d'oracles utilisés, la quantité d'aléa et de bits de preuves lus, le nombre de tours d'interaction entre prouveur(s) et vérifieur, et les probabilités d'acceptations des mots. Le modèle $\mathcal{PCP}$ semble plus proche de notre problématique puisque l'interaction y a lieu au début du calcul, et que les résultats liés à ce modèle sont également des résultats de "compromis" entre quantité d'information donnée par l'oracle, quantité d'aléa autorisée, et probabilité d'acceptation. Même si les paramètres de ce compromis sont assez différents de ceux considérés dans le reste de ce travail (quantité d'information, temps de calcul, performance), il se trouve que ces résultats ont des conséquences directes sur l'approximabilité de certains problèmes d'optimisation classiques. Ces résultats constituent donc un exemple important des conséquences possibles des résultats interactifs.

Chapitre 3

Utilisation d'un oracle en optimisation combinatoire, application à l'ordonnancement

3.1 Introduction, plan du chapitre

Nous avons précédemment abordé l'utilisation d'un oracle en algorithmique distribuée, online, et en complexité. Du point de vue des deux démarches présentées en Section 2.1.1 (*i.e.* quantification de l'information, ou réduction vers un nouveau problème), les travaux abordés sont plutôt centrés sur l'ajout d'information, et donc l'etude de compromis entre quantité d'information ajoutée, coût de l'algorithme interactif, et performance. L'objectif est maintenant de considérer le cas de l'optimisation offline (en particulier le problèmes d'ordonnancement) sous l'angle original de l'interaction, afin de voir en particulier que les deux démarches sont ici possibles.

Nous allons proposer en Section 3.2 un formalisme d'algorithme interactif pour les problèmes d'optimisation. La principale différence avec les situations précédentes est que dans ce contexte, toute l'instance est fournie à l'algorithme. Ainsi, l'oracle ne sert pas à dévoiler une partie inconnue de l'instance, mais plutôt à donner des informations sur la structure des solutions optimales. On pourra donc ici simuler l'oracle pour obtenir des algorithmes "normaux" de même performance, ce qui n'est pas *a priori* le cas en algorithmique distribuée et online. L'obtention d'un algorithme classique à partir d'un algorithme interactif sera abordée en Section 3.3. Nous étudierons en Section 3.4 les liens entre cette formulation interactive et les techniques de conception d'algorithmes "classiques", ou du moins utilisées dans certaines des contributions majeures en ordonnancement, ou citées dans les tutoriaux de [Shachnai and Tamir, 2007, Schuurman and Woeginger, 2000]. Bien que la notion d'algorithme interactif ne soit pas définie dans le but d'obtenir spécialement des PTAS (une PTAS étant un algorithme d'approximation dont on peut contrôler le ratio), nous allons voir que ces deux notions sont liées. En effet, l'utilisation d'une information paramétrable va en général mener à un schéma d'approximation, puisque l'on pourra contrôler le compromis entre la quantité d'information demandée (et donc le temps d'exécution de l'algorithme simulé) et le ratio d'approximation correspondant. Les deux méthodes pour simuler un algorithme intéractif incluent deux techniques classiques de

conception de PTAS présentées en Sections 3.4.1 et 3.4.2. Les Sections 3.4.3et 3.4.4 présentent des outils supplémentaires permettant un codage plus efficace des réponses de l'oracle. Des conclusions sur l'étendue du formalisme interactif sont présentées en Section 3.5.

Les résultats cités dans cette partie sont des résultats classiques, que nous avons réexprimés dans le formalisme interactif proposé. Les nouveaux résultats que avons obtenus grâce à ces techniques seront résumés en Section 4.3. Notons que, même si nous nous référons toujours à quelques problèmes seulement pour donner des exemples, les techniques mentionnées ci-dessus ont été appliquées à de nombreux cas. Afin de comprendre les diverses applications des algorithmes interactifs nous serons amené à présenter parfois en termes de méthodes interactives (ou d'oracle) des résultats n'étant pas formulés de cette façon à l'origine.

Enfin, nous supposons connues les définitions des problèmes classiques d'ordonnancement et d'empaquetage (tels le problème $P||C_{max}$ ou le problème du $Knapsack$), ainsi que les définitions liées au cadre de l'approximation ($PTAS$, $FPTAS$, *etc.*). Pour plus de détail, nous renvoyons le lecteur en annexe (voir Chapitre A).

3.2 Définition du formalisme avec oracle

3.2.1 Définition d'un algorithme interactif

Nous allons à présent définir la notion d'algorithme interactif pour le contexte de l'optimisation offline. Cette définition pourrait s'inspirer des nombreux modèles existants (notamment en complexité), cependant nous préférons donner une définition plus simple dans laquelle l'interaction a lieu au début du calcul, car elle suffira à décrire la majeure partie des exemples considérés. Nous reviendrons sur la justification de ce modèle en Section 3.3.3.

Définition 3.1 (Propriété)**.** *Soit* Π *un problème d'optimisation. Une propriété* P *est une fonction qui pour chaque instance* I *associe une fonction booléene* $P_I : R_I \to \mathbb{B} = \{V, F\}$*, telle que*

- R_I *est un ensemble fini*
- $P_I^{-1}\{V\} \neq \emptyset$

Définition 3.2 (Algorithme interactif)**.** *Soit* Π *un problème d'optimisation. Un algorithme interactif* A_P *pour le problème* Π *demandant la propriété* P *est un algorithme (au sens classique, déterministe) qui pour tout* I*, pour tout* $r_I^* \in P_I^{-1}\{V\}$*, fournit une solution* $A_P(I, r_I^*)$ *réalisable au problème* Π*. Pour alléger l'écriture un tel algorithme sera souvent simplement noté* A*.*

Définition 3.3 (Oracle)**.** *Pour toute propriété* P*, algorithme interactif* A_P *et instance* I*, un oracle fournit en temps et espace constant une réponse* $O(I) \in P_I^{-1}\{V\}$ *(choisie de façon pas nécessairement déterministe).*

Dans la définition 3.2, on considère donc que pour chaque instance I :

- l'algorithme A pose une question P_I
- l'oracle donne une réponse $O(I) \in P_I^{-1}\{V\}$

- le calcul de la solution $A(I, O(I))$ est effectué

Cette définition peut paraître surprenante au sens où il n'y a pas d'interaction tout au long du calcul, mais seulement au début. Un algorithme interactif est donc un algorithme classique ayant à disposition au début du calcul des informations supplémentaires. Dans de nombreux cas, ces informations rajouteront des contraintes sur le problème initial, limitant ainsi les choix sur les parties "critiques" du problème. Remarquons cependant que l'ajout de contraintes par le biais d'une mauvaise question peut parfois rendre le problème plus difficile ! Par exemple pour les problèmes d'ordonnancement de tâches malléables [Mounié et al., 2007] (dans lesquels on peut choisir le nombre de processeurs utilisés pour chaque tâche), demander le nombre de processeurs utilisés pour chaque tâche dans un optimal nous ramènerait à un problème sur des tâches rigides, qui ne sont pas clairement plus simples à traiter. Enfin, on remarquera qu'un algorithme non interactif (*i.e.* sans oracle) est simplement un algorithme demandant la propriété qui à tout I associe la fonction constante égale à V.

Le choix de R_I est laissé au concepteur de l'algorithme. Nous verrons en Section 3.3 qu'il existe deux méthodes principales pour obtenir un algorithme normal à partir d'un algorithme interactif : en calculant séparément un $r_I^* \in P_I^{-1}\{V\}$, ou en ré-exécutant A pour toute valeur $r \in R_I$. Ainsi, même si la définition de l'ensemble R_I n'est vraiment importante que dans le deuxième cas, nous avons choisi d'inclure R_I dans la définition d'un algorithme interactif. En effet, il nous semble que la notion intuitive (ou du moins correspondant à de nombreux travaux existants utilisant des "guess") d'un algorithme interactif correspond bien à un algorithme ayant deux paramètres I et $r \in R_I$, et ne fournissant le résultat escompté que pour des $r_I^* \in P_I^{-1}\{V\}$.

Remarque 3.4. *Soit A_P un algorithme interactif et I une instance. La notation $A_p(I, r)$ a donc un sens pour tout $r \in R_I$ (pas nécessairement dans $P_I^{-1}\{V\}$), et désigne donc l'exécution de A_p sur I et r, qui n'est certes pas garantie.*

Définition 3.5 (Taille de l'information de l'oracle)**.** *On définit la taille de l'information donnée par l'oracle pour une instance I comme $|O(I)| = \max_{r_I^* \in P_I^{-1}\{V\}} |r_I^*|$, où $|r_I^*|$ est le nombre de bits nécessaires pour coder r_I^* dans un codage fixé.*

La taille de l'information de l'oracle sera très importante puisque dans certains cas un algorithme classique est dérivé d'un algorithme interactif en essayant toutes les réponses possibles de l'oracle (voir Section 3.3.2).

3.2.2 Lien avec les définitions existantes

Les définitions précédentes ont été introduites dans ce manuscrit car elles permettent d'exprimer de façon naturelle plusieurs techniques de conception d'algorithmes. Nous n'avons *a priori* pas trouvé de telles définitions dans la littérature de la conception d'algorithmes d'approximation (en particulier dans la littérature concernant les PTAS , voir [Schuurman and Woeginger, 2000, Shachnai and Tamir, 2007]), et ces techniques sont souvent dénommées avec d'autres mots clefs que "interactivité" ou "oracle", et présentées dans un autre contexte que l'interactivité. Nous essaierons autant que possible de donner les appellations classiques des techniques présentées ici sous forme interactives.

On peut dès à présent présenter l'une des appellations les plus courantes : le *guess*, ou les *guessing techniques*. On peut traduire l'expression "we guess x" en disant qu'un algorithme A_P *devine* une valeur x (ou directement que l'on devine x) lorsque A_P est un algorithme interactif, et que $x \in P_I^{-1}\{V\}$. Très souvent, la propriété P mise en jeu est implicite. Nous verrons en effet dans les exemples 3.10, 3.11, 3.12 en Section 3.2.4 que définir P peut parfois paraître "artificiel". Après ces exemples nous ne donnerons en général pas explicitement la propriété P. Cette notion de propriété est donc plutôt définie dans un but d'unification que d'utilisation pratique.

3.2.3 Définition du temps d'exécution et du ratio d'approximation d'un algorithme interactif

On remarque que A_P n'est *a priori* pas déterministe puisque pour un I fixé il peut y avoir plusieurs $r_I^* \in R_I$ tels que $P(r_I^*) = V$. L'ensemble des solutions pouvant être construites pour un I fixé est donc $\{A_P(I, r_I^*), r_I^* \in P_I^{-1}\{V\}\}$. Nous allons donc redéfinir les notions de temps d'exécution sur une instance, de valeur d'une solution calculée sur une instance, et de ratio d'approximation. Deux définitions du temps d'exécution seront données, la justification de cette double définition apparaitra dans la remarque 3.16.

Définition 3.6 (Temps d'exécution)**.** *Soit I une instance d'un problème d'optimisation et A_P un algorithme interactif. Le temps d'exécution $t(A_P(I))$ de A_P sur I est défini par*

$$t(A_P(I)) = \max_{r_I^* \in P_I^{-1}\{V\}} (t_c(A_P(I, r_I^*)))$$

où $t_c()$ est la mesure classique du temps d'exécution d'un algorithme (non interactif déterministe).

Nous allons donner une deuxième définition du temps d'exécution, qui sera utile lorsque l'on utilisera A_p sans oracle, en essayant toutes les réponses possibles dans R_I (voir Section 3.3). Il faut alors que le temps soit garanti même pour les réponses "fausses".

Définition 3.7 (Temps d'exécution robuste)**.** *Soit I une instance d'un problème d'optimisation et A_P un algorithme interactif. Le temps d'exécution robuste $t_r(A_P(I))$ de A_P sur I est défini par*

$$t_r(A_P(I)) = \max_{r \in R_I}(t_c(A_P(I, r)))$$

où $t_c()$ est la mesure classique comme précédemment.

En pratique, on utilisera principalement la notion de temps d'exécution robuste. En effet, dans de nombreux cas le temps d'exécution d'un appel à $A_P(I, r)$ ne dépend pas fortement du fait que r satisfasse P_I ou non, contrairement à la valeur de la solution produite.

Définition 3.8 (Valeur d'un algorithme interactif sur une instance)**.** *Soit I une instance d'un problème d'optimisation et A_P un algorithme interactif. La valeur $v(A_P(I))$ de A_P sur I est définie par*

$$v(A_P(I)) = \max_{r_I^* \in P_I^{-1}\{V\}} (v(A_P(I, r_I^*)))$$

où $v(S)$ est la valeur de la solution S au sens de la fonction objectif du problème considéré.

Nous avons choisi pour ces définitions d'utiliser la fonction *max* car en pratique, lorsque l'on veut analyser le temps d'exécution ou la performance d'un algorithme interactif, on ne maîtrise pas le choix du $r_I^* \in P_I^{-1}\{V\}$. On commence donc l'analyse en considérant un $r_I^* \in P_I^{-1}\{V\}$ quelconque. Typiquement (pour un problème de minimisation), on montre pour une instance I que pour tout $r_I^* \in P_I^{-1}\{V\}$, on a $t(A_P(I, r_I^*)) \leq f(|I|)$ et $v(A_P(I, r_I^*)) \leq g(I)$, ce qui implique donc que $t(A_P(I)) \leq f(|I|)$ et $v(A_P(I)) \leq g(I)$.

Notons que dans la suite la notation v sera parfois abandonnée, et $A_P(I)$ désignera donc parfois $v(A_P(I))$. Maintenant qu'est définie la valeur d'un algorithme interactif sur une instance, la notion de ratio d'approximation peut être définie de façon classique (voir également en Annexe).

Définition 3.9 (Ratio d'approximation)**.** *Considérons un problème de minimisation. Un algorithme interactif A_P a un ratio d'approximation ρ (ou est une ρ-approximation) ssi pour toute instance I, $\frac{v(A_P(I))}{Opt(I)} \leq \rho$, où $Opt(I)$ désigne la valeur optimale associée à l'instance I.*

3.2.4 Exemples

Nous allons à présent donner quelques exemples d'algorithmes interactifs classiques (sur lesquels nous reviendrons) en ordonnancement en utilisant le formalisme introduit précédemment.

Exemple 3.10: Considérons le problème du *Knapsack* (dont la définition est rappelée en Annexe, Section A.2.1). Rappelons qu'une instance de ce problème est décrite par la donnée pour $i \in \{1, \ldots, n\}$ du prix p_i et de la taille s_i de l'objet i, et de la capacité C du sac. L'objectif est de sélectionner un sous-ensemble d'objets de prix maximal, sans excéder la capacité du sac. Soit $k \in N$. Nous allons décrire précisément l'algorithme interactif kMV (pour "k Most Valuable"), présenté dans [Shachnai and Tamir, 2007], qui demande à l'oracle les k objets les plus chers placés dans le sac à dos (ou tous les objets utilisés dans une solution optimale s'il existe un optimal utilisant moins de k objets), et qui termine ensuite avec l'algorithme glouton classique. La propriété P utilisée ici associe donc à chaque instance I la fonction $P_I(X)$ qui vaut V *ssi*

- il existe une solution optimale $Opt(I)$ telle que $|Opt(I)| \leq k$ et $Opt(I) = X$
- il existe une solution optimale $Opt(I) = \{i_1, \ldots, i_x\}$, avec $p_{i_l} \leq p_{i_{l+1}}$ pour tout l, telle que $|Opt(I)| > k$ et $\{i_1, \ldots, i_k\} = X$

Ensuite, étant donné un X^* tel que $P_I(X^*) = V$, kMV ajoute les objets de X^* dans le sac, puis choisit la meilleure solution entre ajouter l'objet le plus cher tenant dans le sac, et ajouter les objets restants par ordre décroissant de densité $\frac{p_i}{s_i}$.

Pour tout I, pour être sûr que R_I contienne un X satisfaisant P_I, on peut définir R_I comme l'ensemble de tous les sous- ensembles d'objets de cardinal au plus k. On peut également considérer que, en demandant à l'oracle de coder un r_I^* en donnant l'indice de tous les objets présents (et éventuellement un certain nombre de fois la valeur 0 si il existe un optimal avec moins de k objets), on obtient $|O(I)| \leq klog(n+1)$. On définit donc R_I comme l'ensemble des chaînes binaires de longueur $klog(n+1)$.

Nous verrons en Section 3.4.1 que kMV a un ratio d'approximation de $1 + \frac{1}{k+1}$, et qu'il peut être dérivé en une PTAS .

Exemple 3.11: Considérons le problème $P||C_{max}$ (voir la définition en Annexe, Section A.2.1), dont une instance est décrite par la donnée du nombre m de machines et de la taille de chaque tâche $p_i, i \in \{1, \ldots, n\}$. Rappelons que ce problème consiste à ordonnancer toutes les tâches, en minimisant la date de fin de la dernière tâche. Supposons que $p_i \geq p_{i+1}$, pour $i \in \{1, \ldots, n-1\}$. Soit $k \in \mathbb{N}$. On considère l'algorithme kBT (pour "k Biggest Tasks") donné dans [Graham, 1966] qui demande à l'oracle un ordonnancement optimal des k plus grosses tâches de l'instance. La propriété P utilisée ici associe donc à chaque instance I la fonction $P_I(x_1, \ldots, x_k)$ qui vaut V *ssi* il existe une solution optimale $Opt(I')$ (pour l'instance I' composée des m machines et des tâches $\{1, \ldots, k\}$) qui ordonnance la tâche i sur la machine x_i, pour $i \in \{1, \ldots, k\}$.

Ensuite, étant donné un $(x_1^*, \ldots, x_k^*)$ tel que $P_I(x_1^*, \ldots, x_k^*) = V$, kBT ordonnance les tâches restantes en utilisant l'algorithme classique de "List Scheduling" LS, c'est-à-dire en ordonnançant n'importe quelle tâche restant dès qu'un processeur est inactif.

Pour tout I, pour être sûr que R_I contienne un X satisfaisant P_I, on peut définir R_I comme l'ensemble de toutes les allocations possibles des k plus grosses tâches. On peut également considérer que, en demandant à l'oracle de coder un r_I^* en donnant pour chaque tâche l'indice de la machine où elle est ordonnancée. On obtient $|O(I)| \leq klog(m)$. On définit alors R_I comme l'ensemble des chaînes binaires de longueur $klog(m)$.

On remarque dans l'exemple précédent que l'information demandée n'est pas une "sous-partie" (*i.e.* les valeurs de certaines variables) d'une solution optimale de l'instance *initiale.*)

Exemple 3.12: Considérons le problème du Strip Packing (voir la définition détaillée en Annexe, Section A.2.2). Rappelons qu'une instance de ce problème est décrite par la donnée de la largeur $m \in \mathbb{N}$ d'une boîte, la largeur $w_i \leq m$ et la hauteur h_i du rectangle i pour $i \in \{1, \ldots, n\}$. L'objectif est de placer les rectangles dans la boîte en minimisant la hauteur atteinte. On considère un algorithme interactif qui demande la valeur de l'optimal. La propriété P utilisée ici associe donc à chaque instance I la fonction $P_I(x)$ qui vaut V *ssi* il existe une solution optimale $Opt(I)$ telle que $Opt(I) = x$. Cette fois ci, la définition de R_I nécessite un encadrement de la valeur de l'optimal, et on peut par exemple prendre $R_I = \{\lfloor \frac{\sum_{i=1}^n w_i h_i}{m} \rfloor, \ldots, \sum_{i=1}^n h_i\}$.

Demander la valeur v de l'optimal est une méthode très classique pour les problèmes d'empaquetage et d'ordonnancement. En effet, sans une telle limite sur "l'horizon" utilisable, de nombreux algorithmes créent une solution en partant du niveau 0 (ou du temps 0 en ordonnancement). Or, cette information permet entre autres de construire un empaquetage "depuis la hauteur v", et pas seulement en partant de la hauteur 0. Une autre intérêt évident est de pouvoir définir des partitions des tâches selon leur temps de calcul, comme par exemple définir les grandes tâches comme ayant un temps d'exécution plus grand que $\frac{v}{2}$. Remarquons que ce type de questions est fortement liée à la "dual approximation technique" [Hochbaum and Shmoys, 1988]. La différence est que ici l'algorithme n'a aucune garantie si la valeur optimale devinée est fausse, alors que dans la technique d'approximation duale l'algorithme est spécifié pour toute valeur devinée w, puisque soit l'on construit une solution de coût inférieur à ρw (avec ρ le

ratio visé pour l'algorithme final), soit on rejette w, signifiant que w est inférieur à la valeur optimale.

Remarquons que l'on va retrouver en optimisation (offline) la notion d'information paramétrable/non paramétrable utilisée dans le Chapitre 2. Dans l'algorithme kMV, on va contrôler la quantité d'information reçue grâce au paramètre k, ce qui n'est pas le cas pour l'exemple 3.12 où l'on demande la valeur de l'optimal. On pressent que la propriété demandée sera souvent du type "il existe une solution optimale telle que". Cependant, même si cela concerne un grand nombre de cas, il serait réducteur de considérer que les algorithmes interactifs demandent toujours simplement une sous-partie d'une solution optimale (voir Section 3.5 pour une discussion sur l'étendue du formalisme oracle).

Nous verrons que les algorithmes interactifs ont un intérêt méthodologique, au sens où ils mettent en valeur, à travers la propriété demandée, une difficulté du problème. Cependant, la principale conséquence est que les algorithmes interactifs permettent la construction d'algorithmes classiques (non interactifs), comme nous allons le voir dans la Section 3.3.

3.3 Obtention d'un algorithme classique

Nous allons présenter les deux techniques les plus courantes (simulation par énumération ou recherche séparée) pour obtenir un algorithme classique à partir d'un algorithme interactif.

Définition 3.13. *On dit que A^e simule par énumération A_P ssi pour tout I, A^e exécute l'algorithme suivant (pour un problème de minimisation) :*

- $best\_sol \leftarrow \emptyset$
- $best\_value \leftarrow +\infty$

for all $r \in R_I$ ***do***
- $current\_sol \leftarrow A_P(I, r)$
- ***if*** $v(current\_sol) < best\_value$ ***then***
  - $best\_sol \leftarrow current\_sol$
  - $best\_value \leftarrow v(current\_sol)$
- ***end if***

end for

return $best\_sol$

Définition 3.14. *On dit que A^s simule par recherche séparée A_P ssi pour tout I, A^s exécute :*

- *trouver* $r_I^* \in P_I^{-1}\{V\}$ *en exécutant* $H(I)$

return $A_P(I, r_I^*)$

où H est un algorithme fixé.

3.3.1 Remarques sur les différences entre ces deux techniques

La première technique consiste à essayer toutes les réponses possibles de l'oracle (on pourrait voir une analogie entre la méthode par énumération et le raisonnement par

tiers-exclu), et la deuxième à calculer d'abord séparément une des réponses possibles de l'oracle.

Nous avions introduit dans le Chapitre 2 deux démarches justifiant l'intérêt d'un résultat interactif (pouvant *a priori* être trivial si l'oracle aide "trop" l'algorithme) : l'étude séparée de la difficulté à retrouver l'information de l'oracle, et la quantification de l'information donnée par l'oracle. Ces deux démarches correspondent donc aux deux techniques de simulation. En effet, la technique par énumération est typique d'une démarche de quantification. Autrement dit, on ne s'intéresse pas à la difficulté à reconstruire l'information séparée (puisque l'on ne va pas le faire !), mais plutôt à l'existence d'un codage qui rendrait cette information courte. La technique par simulation est typique d'une démarche de réduction, puisque l'on montre que pour résoudre un problème A (*i.e.* fournir une performance fixée) il suffit de savoir résoudre un problème B (*i.e.* reconstruire l'information de l'oracle).

Informellement, on peut considérer que l'énumération sera bien choisie dans les cas où

- l'oracle donne peu de bits d'information (*i.e.* les ensembles R_I sont petits), ces bits étant très difficiles à calculer, comme par exemple les réponses aux questions "est ce qu'une instance de $3SAT$ est satisfiable ?", "quel est le nombre chromatique d'un graphe ?", "quelle est la valeur de l'optimal (voir Exemple 3.12) ?", "quels sont les k objets les plus chers à placer dans le sac à dos (voir Exemple 3.10) ?", *etc.*
- l'algorithme interactif a un temps d'exécution faible (puisque celui-ci est relancé pour chaque $r \in R_I$)

Cette technique permet donc en quelque sorte d'utiliser à faible coût des informations qui seraient par exemple "NP-complet" à retrouver. La recherche séparée sera quant à elle adaptée lorsque l'information peut être construite en temps raisonnable. Il faut donc bien différencier le fait qu'une information soit *courte* du fait qu'elle soit *facile à reconstruire*.

Remarquons que même si la recherche séparée paraît moins originale, puisqu'elle correspond à un simple appel à une sous-procédure, elle permet tout de même d'aborder les problèmes de façon différente, comme le montre la technique présentée en Section 3.4.2 dans laquelle on demande à l'oracle de résoudre le même problème, mais sur une instance de taille constante. Cela constitue donc une réduction d'un problème vers "lui-même" (en se limitant aux petites instances) !

A la différence de la recherche séparée, après l'exécution d'une simulation par énumération on ne connaît toujours pas *a priori* de $r_I^* \in P_I^{-1}\{V\}$. Ainsi, même lorsque la recherche séparée s'effectue par énumération (ce qui n'est pas exclu !) d'un ensemble X, celle-ci reste différente de la simulation par énumération. En effet, la recherche séparée ne pourra en général pas se contenter d'énumérer $X = R_I$, car savoir comment sélectionner un $r_I^* \in P_I^{-1}\{V\}$ peut s'avérer aussi difficile que le problème de base (comme cela est par exemple expliqué dans les remarques de la Section 3.4.1).

3.3.2 Caractéristiques des algorithmes obtenus

Nous allons à présent aborder la question de la performance (au sens de la fonction objectif du problème concerné) et du temps d'exécution de ces deux simulations.

Remarque 3.15. *Soit A_P un algorithme interactif (pour un problème de minimisation), et soient A^e et A^s les algorithmes de simulation de A_p. Pour toute instance I, on a $v(A^e(I)) \leq v(A_P(I))$ et $v(A^s(I)) \leq v(A_P(I))$.*

Les simulations préservent donc en particulier le ratio d'approximation. Pour ce qui est du temps d'exécution des simulations, nous allons donner la majoration évidente qui se dégage des définitions. On voit qu'il est nécessaire d'utiliser la définition de temps d'exécution *robuste* d'un algorithme interactif, étant donné que l'on va exécuter $A_p(I, r)$ pour des $r \in R_I | P_I(r) = F$.

Remarque 3.16. *Soit A_P un algorithme interactif (pour un problème de minimisation), et soit A^e et A^s les algorithmes de simulation de A_p. Pour toute instance I, on a $t(A^e(I)) \leq |R_I| t_r(A_P(I))$ et $t(A^s(I)) \leq t(H(I))t(A_P(I))$.*

Pour le cas de l'énumération, il est critique de majorer $|R_I|$ (ou $|O(I)|$). Deux visions sont possibles pour une telle majoration. Une première approche consiste à majorer "directement" $|R_I|$, par des arguments de dénombrement classiques. L'autre approche consiste à majorer $|O(I)|$, c'est-à-dire à se concentrer sur la définition d'un *codage* efficace. Ce deuxième point de vue consiste donc à :

- choisir un codage pour la réponse de l'oracle pour obtenir un majorant $b \in \mathbb{N}$ de $|O(I)|$ "petit"
- définir R_I comme étant l'ensemble des chaînes (binaires) de longueur b

Bien évidemment, pour certaines propriétés (par exemple demander un sous-ensemble d'un ensemble muni d'aucune loi) il semble inutile de se plonger dans l'étude d'un codage, puisque l'on est obligé de donner directement les éléments sélectionnés. Cependant cette vision axée sur le codage nous semble dans d'autres cas très fructueuse (voir par exemple une application en Section 3.4.4). En effet, cette formulation incite à étudier comment réduire le codage (c'est-à-dire définir un codage entraînant des pertes partielles d'information qui sont négligeables pour l'algorithme). Il en résulte des "allègements" des propriétés utilisées étant *a priori* moins faciles à caractériser directement.

Nous aborderons en Section 3.4.3 des techniques classiques de simplification de l'instance permettant entre autres un meilleur codage. Cette recherche de réponses courtes est souvent le point clef lors de la conception des algorithmes. D'un point de vue méthodologique, il peut être utile de fournir d'abord un résultat avec une réponse trop grande, et d'affiner ensuite en cherchant à déduire plus d'informations "soi-même" et donc en demandant moins à l'oracle.

D'autre part, il arrivera souvent que l'on puisse difficilement améliorer le facteur pessimiste $|R_I|$, mais qu'il existe en pratique une heuristique qui n'énumère que partiellement $|R_I|$ en évitant de considérer de nombreuses réponses "fausses" (voir Section 3.5).

3.3.3 Justification du modèle interactif choisi

Il serait facile de définir un modèle réellement interactif (comme en complexité par exemple) dans lequel l'algorithme pose plusieurs questions à l'oracle, chaque question dépendant de toutes les questions/réponses précédentes. La difficulté apparaîtrait en reliant les caractéristiques de l'algorithme interactif (temps, ratio d'approximation) à

celles de l'algorithme simulé. Le cas de la simulation par recherche séparée ne poserait pas de problème, puisqu'un algorithme réellement interactif, simulé par recherche séparée, correspond simplement à un algorithme effectuant plusieurs appels à une sous-procédure. Il suffirait donc d'une part de compter le nombre d'appels à l'oracle, et de l'autre de majorer le temps de simulation (*i.e.* le temps de cette procédure). L'analyse est donc simple par recherche séparée car pendant l'énumération, l'algorithme n'est jamais utilisé avec une réponse fausse. Le cas de la simulation par énumération serait plus délicat, puisque lors de la simulation une réponse fausse pourrait entraîner des questions inutiles et nécessitant d'autres réponses très longues. Autrement dit, il serait difficile de borner raisonnablement le temps de ces algorithmes. De plus, il arrive souvent en pratique que, même en concevant un algorithme sans se limiter *a priori* à une interaction réduite au début du calcul, on s'aperçoive qu'il est possible de reformuler l'algorithme pour que toutes les questions soient posées au début. Nous abordons l'idée d'une vraie interaction plutôt comme une piste possible, voir Section 3.5.5.

Nous verrons que ce formalisme suffit à décrire de nombreux algorithmes et techniques de conception d'algorithmes offline. Remarquons que cette interaction réduite à un ajout d'information est également courante dans les autres domaines abordés dans le Chapitre 2. Une des exceptions concerne l'algorithmique online et les informations dites "délivrées dynamiquement" (voir Section 2.3.4), c'est-à-dire fournies à l'arrivée de chaque nouvelle requête. Cependant, l'utilisation de cette définition plus riche est facilitée dans ce cadre puisque ces algorithmes online utilisant de telles informations ne sont pas destinés à être simulés sans oracle.

3.4 Lien avec les techniques classiques de conception de schémas d'approximation

3.4.1 Structuration de l'espace des solutions

Présentation de la méthode Nous allons donner la définition de la technique "structuring the output" telle qu'elle présentée dans [Schuurman and Woeginger, 2000]. Cette technique semble être très proche de la technique "Extend all possible solutions for small subsets" dans la classification de [Shachnai and Tamir, 2007], et de la technique des "outline schemes" introduite dans [Hall and Shmoys, 1989]. Cette technique est incluse dans la démarche de type quantification. L'idée est de partitionner l'ensemble des solutions réalisables en un grand nombre de petites *régions* sur lesquelles le problème est facile à approximer. Les trois étapes de cette technique sont :

- **Partitionner** : partitionner l'ensemble $\mathcal{S}$ des solutions réalisables d'une instance I en régions $\mathcal{S}^{(1)}$, ..., $\mathcal{S}^{(d)}$ telles que $\bigcup_{l=1}^{d} \mathcal{S}^{(l)} = \mathcal{S}$
- **Trouver des *représentants*** : pour chaque $\mathcal{S}^{(l)}$ trouver une solution (un représentant) $X^{(l)}$ étant une bonne approximation (à un ratio ρ) de $Opt^{(l)}$, la meilleure solution de $\mathcal{S}^{(l)}$
- **Sélectionner le meilleur** : sélectionner le meilleur des représentants comme solution pour l'instance I

Ainsi, étant donné qu'il y au moins une région l_0 contenant une solution optimale Opt, on obtiendra bien une ρ-approximation grâce au représentant de cette région.

Cette technique est présentée comme servant à concevoir des PTAS (même si l'on pourrait *a priori* l'appliquer pour construire des algorithmes d'approximation "quelconques"). Ainsi, plus le ratio désiré sera faible, plus le nombre de régions sera important. L'idée est que toutes les solutions d'une région vérifient une propriété commune, comme par exemple considérer pour un problème d'ordonnancement que la tâche 4 est ordonnancée sur la machine 2 à l'instant 10. Cette propriété fixe les valeurs de certaines variables, en espérant que l'optimisation sur les variables (non contraintes) restantes soit plus aisée.

Exemple détaillé d'application Nous allons à présent analyser en détail l'algorithme kMV de l'exemple 3.10 pour le problème du sac à dos. Cette PTAS a été introduite dans [Sahni, 1975], et est reprise dans [Shachnai and Tamir, 2007] pour illustrer la technique de "Extend all possible solutions for small subsets". Dans cet algorithme, une région est telle que toutes les solutions appartenant à celle-ci placent les mêmes k objets les plus chers.

Théorème 3.17 ([Sahni, 1975])**.** *Pour tout $k \in \mathbb{N}$, l'algorithme kMV de l'exemple 3.10 est une $(1 + \frac{1}{k+1})$-approximation.*

Preuve Une instance est composée de n objets, de taille s_i et profit p_i pour $i \in \{1, \ldots, n\}$, et de la capacité du sac C. Pour tout ensemble d'objet Y nous noterons $p(Y) = \sum_{i \in Y} p_i$ et $s(Y) = \sum_{i \in Y} s_i$.

Au lieu de directement considérer que kMV demande les k objets les plus chers d'une solution optimale (voir exemple 3.10 pour la défintion précise de la propriété demandée), nous allons supposer que kMV demande k objets *a priori* quelconques de l'optimal, pour mieux comprendre l'intérêt de demander les plus chers. En effet on pourrait à la base se demander ce que l'on obtiendrait en demandant les k objets les plus gros, ou les plus denses utilisés, *etc.*

Considérons une solution optimale Opt utilisant strictement plus de k objets (sinon kMV est optimal). Notons $Opt = \{b_1, \ldots, b_k, a_1, \ldots, a_x\}$, avec $\frac{p_{(a_i)}}{s_{(a_i)}} \geq \frac{p_{(a_{i+1})}}{s_{(a_{i+1})}}$ pour $i \in \{1, \ldots, x\}$. L'algorithme kMV construit donc la solution $S = \{b_1, \ldots, b_k\} \bigcup X$, où X est déterminé en plaçant à chaque pas l'objet le plus dense (ayant le plus grand $\frac{p_j}{s_j}$) possible. Soit $m = \min\{i \in \{1, \ldots, x\} | a_i \notin X\}$ (si m n'existe pas alors S est optimale). Soit $P_1^* = \{b_i, i \in \{1, \ldots, k\}\}, P_2^* = \{a_i, i \in \{1, \ldots, m-1\}\}$ et $\Delta = \sum_{X'} s_i$ ou $X' \subset X$ est l'ensemble des objets ajoutés par kMV avant a_m (*i.e.* l'ensemble des objets de X plus denses que a_m, a_m non compris).

On a

$$\begin{aligned} S &\geq p(P_1^*) + p(P_2^*) + \Delta \frac{p_{a_m}}{s_{a_m}} \\ Opt &\leq p(P_1^*) + p(P_2^*) + p_{a_m} + (C - s(P_1^*) - s(P_2^*) - s_{a_m}) \frac{p_{a_m}}{s_{a_m}} \\ \Longrightarrow S - Opt &\geq \frac{p_{a_m}}{s_{a_m}} (\Delta - s_{a_m} - C + s(P_1^*) + s(P_2^*) + s_{a_m}) \end{aligned}$$

Or, a_m n'étant pas mis dans le sac on sait que $\Delta + s(P_1^*) + s(P_2^*) + s_{a_m} > C$, impliquant $S > Opt - p_{a_m}$ et donc $\frac{S}{Opt} > 1 - \frac{p_{a_m}}{Opt}$. On voit donc avec cette analyse que l'on a intérêt à ce que Opt soit grand devant p_{a_m}. Ainsi, en demandant les k objets les plus chers on obtient $S > (k+1)p_{a_m} > (k+1)(Opt - S)$, et donc $Opt < (1 + \frac{1}{k+1})S$.

□

Corollaire 3.18. *Il existe une PTAS pour le problème du Knapsack qui pour tout $k \in \mathbb{N}$ fournit une $(1 + \frac{1}{k+1})$-approximation en un temps $\mathcal{O}(n^{k+1})$.*

Preuve En supposant négligé le coup initial de tri des objets selon leur densité, on obtient $t_r(kMV) \in \mathcal{O}(n)$. On remarque bien que le temps d'exécution de kMV ne dépend pas critiquement du fait que l'information fournie soit correcte. En simulant par énumération kMV, il faudra parcourir l'ensemble R_I, qui a été défini (voir Exemple 3.10) comme l'ensemble de tous les sous-ensembles d'objets de cardinal au plus k, ou comme l'ensemble des chaînes de longueur au plus $klog(n+1)$. Ainsi, on obtient un coût en $\mathcal{O}(n^k t_r(kMV)) = \mathcal{O}(n^{k+1})$. □

Remarques Notons tout d'abord que l'information demandée ici est *a priori* difficile à retrouver, même pour $k = 1$, au sens où décider en temps polynomial si un objet i est l'objet le plus cher d'un optimal permettrait de construire une solution exacte en temps polynomial. D'autre part, dans le cas de l'algorithme kMV, la propriété concerne simplement les valeurs dans un optimal d'un sous-ensemble des variables. Cependant, il ne faut pas se limiter à ce type de définition de régions. La définition de la technique d'un "outline scheme" introduite dans [Hall and Shmoys, 1989] montre d'ailleurs bien l'étendue possible pour la définition des régions. Un outline scheme est un étiquetage f qui à toute solution réalisable S associe une étiquette $f(S)$, censée représenter un "résumé" (incomplet en général) de S. Les auteurs définissent alors un "$1+\epsilon$-optimal outline scheme" comme un algorithme qui pour toute instance I et étiquette w construit une $(1+\epsilon)$-approximation de toute solution S telle que $f(S) = w$. Ainsi, on obtient $1+\epsilon$ d'une solution optimale $Opt(I)$ en fournissant $w_0 | Opt(I) \in f^{-1}(w_0)$. La définition d'un "outline scheme" semble donc bien correspondre à celles de "structuring the output" et de "extend all possible solutions for small subsets", et montre bien que f n'est pas restreinte à l'extraction de la valeur exacte de certaines variables.

Du point de vue des algorithmes interactifs, on remarque que cette méthode peut être vue comme un algorithme interactif (simulé par énumération) pour lequel l'information donnée par l'oracle est l'indice d'une région contenant une solution optimale. Remarquons que même si cela n'est pas le cas dans l'algorithme kMV, le "représentant" sélectionné pour chaque région peut ne pas appartenir à celle-ci. Le formalisme "structuring the output" nous montre au passage que l'on a pas seulement une ρ-approximation de l'optimal (global), mais également une ρ-approximation de tous les optimaux contraints à une région. Ce dernier point peut aussi être vu comme un inconvénient puisque la propriété demandée (approximer tous les optimum locaux) est plus forte que celle réellement nécessaire (garantir le ratio de l'algorithme uniquement pour des r_I^* satisfaisant P_I).

On note également que c'est le fait de considérer une information *paramétrable* qui permet dans l'exemple 3.10 de construire un schéma d'approximation. Cependant, ces techniques ont également un intérêt avec des informations non paramétrables, comme nous le verrons dans un des résultats que nous avons obtenu (voir Chapitre 11) dans lequel l'algorithme a un ratio fixé, et la taille de l'information donnée par l'oracle peut varier énormément selon les instances.

D'un point de vue plus pratique, nous reviendrons en Section 3.5 sur cet exemple en abordant la question de l'efficacité de l'énumération. L'idée étant que l'on peut utiliser des outils classiques (branch and bound,...) qui, même s'ils ne conduisent pas à de meilleures bornes sur le temps d'exécution, améliorent grandement les performances de la simulation en pratique.

Les questions liées aux résultats négatifs (a-t-on demandé la meilleure information parmi toutes celles de tailles "semblables" ?) seront abordées en Section 3.5.

Enfin, il nous faut rappeler que cette PTAS est présentée dans un but pédagogique, puisque de biens meilleurs algorithmes existent pour ce problème (en particulier grâce à des mélanges d'arrondis et de programmation dynamique [Lawler, 1979, Mastrolilli and Hutter, 2006]).

3.4.2 Extension d'une solution optimale d'un sous-ensemble

Présentation de la méthode Nous allons donner la définition de la technique "extending an optimal solution for a single subset" telle qu'elle présentée dans [Shachnai and Tamir, 2007]. Cette technique est incluse dans la démarche de type réduction. L'idée originale est d'exploiter la possibilité de résoudre le problème considéré *optimalement* sur une sous-instance de taille constante.

Les trois étapes de cette techniques sont :

- **Sélectionner un sous-ensemble** : sélectionner un sous-ensemble $I' \subset I$ (en général de taille constante) représentant la partie la plus "importante" de l'instance (typiquement en ordonnancement les plus grosses tâches)
- **Résoudre optimalement** : construire une solution $S^*(I')$ optimale pour I' (au pire par énumération)
- **Étendre** : construire à partir de $S^*(I')$ une solution $S(I)$

Cette technique est présentée comme servant à concevoir des PTAS (même si l'on pourrait *a priori* l'appliquer pour construire des algorithmes d'approximation "quelconques"). Ainsi, plus le ratio désiré sera faible, et plus le sous-ensemble d'éléments critiques sera grand.

Exemple détaillé d'application Nous allons à présent analyser en détail l'algorithme kBT de l'exemple 3.11 pour le problème $P||C_{max}$. Cet algorithme a été introduit dans [Graham, 1966], et est repris dans [Shachnai and Tamir, 2007] pour illustrer la technique de "Extending an optimal solution for a single subset".

Théorème 3.19 ([Graham, 1966]). *Pour tout $k \in \mathbb{N}$, l'algorithme kBT de l'exemple 3.11 est une $1 + \frac{1-\frac{1}{m}}{1+\lfloor k/m \rfloor}$-approximation (où m est le nombre de machines).*

Preuve Une instance est donc composée de m machines de même vitesse et de n tâches de tailles p_i. Supposons que $p_i \geq p_{i+1}$, pour $i \in \{1, \ldots, n-1\}$. Pour tout ensemble de tâches Y, nous noterons $W(Y) = \sum_{i \in Y} p_i$ le travail total de l'ensemble Y.

Au lieu de considérer directement que kBT demande à l'oracle un ordonnancement optimal des k plus grandes tâches (voir exemple 3.11 pour la définition précise de la propriété demandée), nous allons supposer que kBT demande un ordonnancement optimal d'un ensemble X de k tâches *a priori* quelconques, pour mieux comprendre l'intérêt de demander les plus grandes.

Rappelons que kBT ordonnance les tâches de $I \backslash X$ avec le principe de "List Scheduling" (noté LS), qui commence n'importe quelle tâche dès qu'une machine est inactive. Pour toute tâche i, notons C_i la date de fin de i. Soit x une tâche terminant en dernier, *i.e.* telle que C_x soit égal au makespan de $kBT(I)$. Soit s_x la date de début de x. Si $x \in X$, alors $C_x \leq Opt(X) \leq Opt(I)$, l'ordonnancement est donc optimal. Sinon, en appliquant l'analyse classique de LS, on sait que x n'a pas été ordonnancée avant s_x car toutes les machines étaient occupées, impliquant $W(I) \geq ms_x + p_x$. On en déduit

$$\begin{aligned} kBT(I) &= s_x + p_x \\ &\leq \frac{W(I)}{m} + p_x(1 - \frac{1}{m}) \\ &\leq Opt(I) + p_x(1 - \frac{1}{m}) \end{aligned}$$

Pour avoir un faible ratio, il faudrait donc que p_x soit petit devant $Opt(I)$. C'est ici que l'on constate qu'une solution possible pour avoir $\frac{p_x}{Opt(I)}$ petit est de choisir X comme étant les k plus grandes tâches. En effet, avec $k = am+b, (a,b) \in \mathbb{N}^2, b < m$, on constate que toute solution est obligée de placer au moins $a+1$ tâches de $X \bigcup \{p_{k+1}\}$ sur au moins une machine, et on obtient donc $Opt(I) \geq (a+1)p_{k+1} \geq (a+1)p_x$, et le ratio escompté. □

Remarque 3.20. *Dans la preuve précédente on a en fait pas besoin d'un ordonnancement optimal des k plus grandes tâches, mais uniquement d'une* $1 + \frac{1-\frac{1}{m}}{1+\lfloor k/m \rfloor}$*-approximation.*

Corollaire 3.21. *Il existe une PTAS pour* $Pm||C_{max}$ *qui pour tout* $k \in \mathbb{N}$ *fournit une* $1 + \frac{1-\frac{1}{m}}{1+\lfloor k/m \rfloor}$*-approximation en un temps* $\mathcal{O}(m^k + nlog(n))$.

Preuve On sait que le coût de LS, et donc de $t(kBT)$ est en $O(nlog(n))$. En effectuant une recherche séparée naïve par énumération, on obtient le résultat désiré. □

Remarques sur cet exemple Dans le cas étudié précédemment la propriété concerne les valeurs, dans un optimal pour une sous-instance, d'un sous-ensemble des variables. L'information demandée est paramétrable, et étant donné que la recherche séparée nécessite un temps croissant en la valeur du paramètre, on obtient donc un schéma d'approximation.

La formulation interactive permet là aussi "d'extraire" la difficulté du problème, et motive ici l'étude de l'ordonnancement optimal des km plus grandes tâches sur des machines identiques. Il est alors intéressant de constater que l'ordonnancement des m plus grosses tâches étant trivial, on peut obtenir une $\frac{3}{2}$-approximation de temps d'exécution semblable à LS. En fait, il est même bien connu que l'obtention d'une $\frac{4}{3}$-approximation des $2m$ plus grosses tâches est également simple à construire (en utilisant l'algorithme "Largest Processing Time" LPT, qui agit comme LS, excepté qu'il choisit la plus grande tâche restante), et on retrouve en quelque sorte la preuve "classique" du ratio de $\frac{4}{3}$ de LPT [Graham, 1969] dans laquelle on se ramène à l'étude des (au plus $2m$) tâches de tailles strictement plus grande que $\frac{Opt(I)}{3}$. Notons également que sur cet exemple il est possible d'améliorer le coût de simulation, avec une programmation dynamique classique (voir remarque 3.22).

Remarque 3.22. *Le coût de la recherche séparée de l'algorithme kBT peut être ramenée à $m2^{2k}$. Pour cela on définit $f(i, X)$ comme la valeur de l'ordonnancement optimal de l'ensemble de tâches X sur les machines $1, \ldots, i$. On a donc (avec $p(T) = \sum_{j \in T} p_j$)*

$$\begin{aligned} f(i, X) &= \min_{T \subseteq X} (\max(p(T), f(i-1, X \setminus T))) \\ f(1, X) &= p(X) \end{aligned}$$

Etant donné que l'on cherche $f(m, X_0)$ avec X_0 l'ensemble des k plus grandes tâches, on obtient donc l'information en temps $\mathcal{O}(m2^{2k})$ (et espace $\mathcal{O}(2^{2k})$).

Comme il l'a été suggéré dans la démonstration du théorème 3.19, on peut également se demander s'il existe une autre information de taille "semblable" qui conduirait à de meilleurs résultats. Ce type de questions sur les résultats négatifs sera abordé brièvement en Section 3.5.

Enfin, il nous faut également rappeler que cette PTAS est présentée dans un but pédagogique, puisque de meilleurs algorithmes existent pour ce problème [Hochbaum and Shmoys, 1987] (l'exemple étudié n'est pas une PTAS pour $P||C_{max}$, car la complexité pour un ratio de $1 + \epsilon$ est exponentielle en $\frac{m}{\epsilon}$).

Un exemple plus général d'algorithme interactif simulé par recherche séparée

Dans la méthode "extending an optimal solution for a single subset", ainsi que dans l'exemple précédent, il est suggéré que l'information demandée à l'oracle concerne la résolution optimale (ou presque) d'une "petite" sous-partie de l'instance. Cependant, il est important de voir qu'il ne faut pas se limiter à ce type d'informations. Nous allons donc donner un exemple utilisant une information de nature différente, et menant de plus à une FPTAS .

Il existe toute une littérature [Korte and Schrader, 1981a] concernant l'utilisation d'oracles pour l'optimisation pouvant être reliée aux algorithmes interactifs simulés par recherche séparée (selon notre terminologie). La motiviation initiale (d'après [Korte and Schrader, 1981a]) de ces travaux est l'étude des problèmes d'optimisation pour lesquels l'entrée du problème ne peut pas être encodée raisonablement. Par exemple lorsque les entrées comportent des fonctions non linéaires, des ensembles

de fonctions ou des "systèmes d'ensembles". Dans de tels cas il est non seulement difficile d'éstimer la longueur de l'entrée, mais également le temps nécessaire pour retrouver des informations basiques sur l'entrée (évaluation d'une fonction par exemple). Ainsi, de nombreux travaux contournent cette difficulté en utilisant la notion d'oracle pour mettre en "boîte noire" ce problème lié à l'entrée. La démarche consiste à d'abord fixer un oracle, en définissant ses capacités. Par exemple pour un problème de recherche de point fixe d'une fonction f, on considère un oracle r_0 qui pour tout x donne la valeur $f(x)$. Ensuite, un résultat positif consiste à construire un algorithme utilisant cet oracle (en comptabilisant le nombre d'appels à l'oracle), et un résultat négatif consiste à minorer le nombre d'appels nécessaires à cet oracle pour atteindre une performance fixée.

Les auteurs de [Korte and Schrader, 1981a] donnent un large aperçu de ce type de résultats à travers un formalisme interactif, où l'interaction est de plus une vraie discussion (*i.e.* n'a pas lieu uniquement au début). Même si cette démarche ressemble à celle menée ici au sens où l'oracle permet d'isoler la difficulté d'un problème, le contexte est différent puisque dans les problèmes qui nous intéressent il est simple d'encoder l'entrée du problème, et de retrouver les informations de base la concernant.

Cependant, certains de ces résultats sont reliés à notre problématique, car ils considèrent des oracles donnant plus qu'un simple accès à l'instance. En effet, si l'oracle considéré donne une information qui est triviale lorsque l'on a accès à l'instance, alors un résultat positif sera certes applicable dans notre formalisme, mais *a priori* inefficace puisque ne supposant que cet accès restreint à l'instance. Par exemple, les résultats dans [Hausmann et al., 1981] sur le problème du *Knapsack* utilisent un "feasibility oracle" qui pour tout ensemble d'objets X, dit simplement si X dépasse la capacité du sac ou non. Cet oracle est donc trivial. Ainsi, étudier le nombre d'appels à cet oracle est maladroit pour nous, puisque par exemple l'accès complet à l'instance permet d'écrire une programmation dynamique. Cependant, il existe des travaux où l'oracle effectue une tâche non triviale (même lorsque l'on a accès à l'instance), comme dans [Korte and Schrader, 1981b] où l'oracle $r_{\epsilon-dom}$, pour tous ensembles d'objets A et B, est capable de dire si A ϵ-domine B (au sens où il existe une "extension" de A de meilleur profit, à ϵ près, que toutes les extensions de B). Les auteurs prouvent le théorème suivant.

Théorème 3.23 ([Korte and Schrader, 1981b]). *Pour tout problème de maximisation sur un "independence system" (incluant le problème du Knapsack), il existe une FPTAS ssi l'oracle $r_{\epsilon-dom}$ peut être simulé en temps polynomial en n (le nombre d'objets) et $\frac{1}{\epsilon}$.*

Ainsi, ce théorème fournit un résultat positif lié à la simulation par recherche séparée, qui est intéressant même dans notre contexte d'optimisation offline où l'entrée est connue. On constate que l'information demandée n'est pas simplement la résolution optimale d'une sous-instance. Comme tous les résultats interactifs par simulation séparée, ce résultat peut être vu comme une réduction, puisque l'on montre que pour obtenir une FPTAS pour le problème de base il suffit de savoir simuler $r_{\epsilon-dom}$ en temps polynomial.

Il faut noter que ce résultat utilise une vraie interaction (pas seulement au début du calcul), ce qui n'est pas surprenant puisque comme nous l'avons remarqué précédem-

ment la notion de vraie interaction est naturelle lorsque la simulation se fait par recherche séparée.

3.4.3 Méthodes par simplification de l'instance et lemme structurel

Simplification de l'instance

Une autre technique courante pour l'élaboration de schémas d'approximations consiste à simplifier de façon "contrôlée" l'instance. Conformément à la définition de [Schuurman and Woeginger, 2000], les trois étapes principales consistent à :

- simplifier l'instance I en une instance I' selon la précision désirée ϵ (typiquement en éliminant ou en groupant les "objets insignifiants", en arrondissant les valeurs des entrées pour en obtenir un nombre constant, en groupant des objets entre eux, *etc.*) et borner l'écart entre $Opt(I')$ et $Opt(I)$
- fournir une solution S' pour I' (à un certain ratio près, ou éventuellement une solution exacte)
- traduire S' en une solution S pour I, et borner l'écart entre S' et S

Ces techniques sont appelées dans [Shachnai and Tamir, 2007, Schuurman and Woeginger, 2000] "applying enumeration to a compacted instance", ou "structuring the input".

Cependant, l'objectif n'étant pas de recenser les techniques de conception de schémas d'approximation, nous ne rentrerons pas dans les détails et les variantes de ces techniques de simplification. Ces techniques peuvent quasiment à elles seules aboutir à des schémas d'approximation (la passe "finale" de résolution par programmation dynamique ou linéaire n'étant pas forcément la plus difficile), comme par exemple dans [Fernandez de la Vega and Lueker, 1981, Kellerer, 1999]. Nous présentons ici ces techniques plutôt comme un ingrédient supplémentaire pouvant servir à définir des codages plus intelligents des réponses demandées à l'oracle, et donc pour communiquer plus efficacement avec celui-ci.

On peut néanmoins citer les méthodes d'arrondi les plus classiques afin de montrer en quoi elles permettent des codages efficaces. Considérons un problème de type ordonnancement ou empaquetage, décrit par la donnée d'un ensemble X de $m = |X|$ ressources (machines, boîtes) décrites par des paramètres $(y_i^1, \ldots, y_i^c), c \in \mathbb{N}, i \in \{1, \ldots, m\}$, et de n objets (tâches, rectangles) décrits par des paramètres $(x_j^1, \ldots, x_j^{c'}), c' \in \mathbb{N}, j \in \{1, \ldots, n\}$. Soit I une instance, $L \leq Opt(I)$ une borne inférieure (on considère un problème de minimisation), et $\epsilon > 0$. Les simplifications d'instance consistent typiquement à :

- supprimer les petits objets (tâches courtes, rectangles de petite aire, *etc.*) ou des ressources négligeables (machines trop lentes, boîtes trop petites *etc.*). Par exemple, supprimer les objets j tels que $x_j^a \leq \epsilon L$, ou supprimer les ressources i telles que $y_i^{a'} \leq f(\epsilon, X)$ pour a, a', f fixés.
- arrondir les paramètres des objets et ressources restants, typiquement afin d'obtenir un nombre constant (dépendant de ϵ) de "profils" d'objets ou de ressources. Par exemple, les arrondis dits arithmétiques ou géométriques consistent à arrondir les paramètres des objets sur des multiples de ϵL ou sur des puissances de ϵ (on peut aussi mélanger les deux [Mastrolilli and Hutter, 2006]),

ou de même à arrondir les paramètres des machines (vitesses, dimension des boîtes, *etc.*).

Ces simplifications sont en général définies pour que la construction de S' puisse se faire par énumération, programmation dynamique ou linéaire (souvent en utilisant [Lenstra Jr, 1983]), typiquement en temps $poly(n,m)^{f(1/\epsilon)}$ ou $poly(n,m)f(1/\epsilon)$.

On trouvera par exemple dans [Mastrolilli, 2003] une $EPTAS$ pour $P|r_j|L_{max}$ (voir définition en Annexe, page 200). Rappelons que ce problème classique consiste à ordonnancer n tâches, chaque tâche n'étant disponible qu'à partir de l'instant r_j et devant être complétée si possible avant une date d_j. Le but est de minimiser le retard (défini comme l'écart entre la date de fin d'une tâche et sa date d_j) maximum. Cette $EPTAS$ est obtenue grâce à des simplifications de l'instance menant à un nombre constant de *profils* (*i.e.* ayant les mêmes p_j, r_j, q_j) de tâches possibles, impliquant que le nombre x d'ensembles "candidats" sur une machine (c'est-à-dire un ensemble de tâches pouvant être allouées par un optimal sur une machine) est constant. On peut alors écrire un programme linéaire afin de déterminer pour tout $i \in \{1, \ldots, x\}$, le nombre de machines x_i traitant un ensemble candidat de type i. L'avantage de cette écriture est que le nombre de variables sera en $\mathcal{O}(x)$ (en l'occurrence $2x+1$), et l'application de [Lenstra Jr, 1983] pour la résolution de ce programme linéaire, n'entraînant un coût exponentiel uniquement en le nombre de variables, permettra une $EPTAS$. Ce type d'approche est même généralisé dans [Epstein and Sgall, 2004] pour des fonctions objectifs "quasi" quelconques.

Un autre résultat montrant bien la puissance des simplifications est dans [Fishkin et al., 2008], où une FPTAS (quand le nombre de machines m est fixé) est obtenue pour $R|c_{ij}|C_{max}$ (généralisation de $R|c_{ij}|C_{max}$ avec des coût supplémentaires) uniquement par des techniques de simplification (arrondis et fusions des "petites" tâches), associées à une programmation dynamique standard.

Du point de vue des algorithmes interactifs, et en particulier de ceux simulés par énumération, on peut voir ces simplifications comme un moyen de communiquer efficacement avec l'oracle. Ces techniques reviennent donc à définir I' afin de pouvoir coder de façon courte les informations demandées à l'oracle, et ce en bornant l'écart entre $Opt(I')$ et $Opt(I)$. Nous avons appliqué ce type de technique dans le Chapitre 11 pour le problème du MSP, dans lequel on arrondit la longueur de tous les "gros" rectangles afin de demander à l'oracle la longueur (arrondie) de certains d'entre eux, au lieu de demander directement leurs indices.

Cependant, il faudra bien distinguer les techniques de simplification qui *modifient* l'instance des autres techniques d'arrondis que nous avons nommées "guess approximation", (voir Section 3.4.4) qui consistent à demander approximativement les valeurs de certaines quantitées optimales, sans modifier l'instance.

Structural Lemma

Les lemmes structurels sont en quelque sorte une extension de la technique de simplification de l'instance. Les trois étapes principales sont donc les mêmes que ci-dessus. L'idée est de montrer qu'il existe une solution $\tilde{S}$ très "structurée" proche d'une solution optimale. La différence est que, pour structurer $\tilde{S}$ on ne se contente pas seulement de simplifier l'instance, mais on peut également ajouter des contraintes (autres que "n'utiliser qu'un sous-ensemble d'objets" !).

Un lemme structurel consiste donc typiquement à partir d'une instance I, d'un $\epsilon > 0$, et d'une solution optimale S_1^* pour I, et à montrer qu'à chaque simplification supplémentaire de I (ou ajout de contrainte), S_i^* ne se "dégrade" que de au plus ϵ en S_{i+1}^*. Notons qu'une fois obtenue une solution "assez" structurée $S_{i_0}^*$, la solution construite par l'algorithme (approchant $S_{i_0}^*$) ne respectera pas forcément les contraintes additionnelles. Bien que n'étant pas directement liée aux algorithmes interactifs, nous citons cette technique puisqu'elle s'inscrit finalement dans la démarche d'une communication efficace avec l'oracle, qui donnerait dans ce cas des informations concernant $S_{i_0}^*$.

On peut par exemple citer [Bansal et al., 2009a, Jansen and Thöle, 2008] pour des exemples concernant des problèmes de packing 2D. Ces lemmes structurels consistent entre autres à arrondir la hauteur des "hauts" rectangles sur des multiples d'un $\alpha > 0$, et à ajouter la contrainte de les placer dans la boîte uniquement à des hauteurs multiples de $\alpha > 0$. Ce type d'ajout de contraintes permet bien évidemment une énumération moins coûteuse de la position (du moins de la hauteur) de ces hauts rectangles.

Nous allons à présent aborder en Section 3.4.4 un dernier type de technique servant à raccourcir le codage des informations fournies par l'oracle, et donc à améliorer l'efficacité de l'énumération.

3.4.4 Guess approximation

Présentation de la technique

Il nous a semblé utile de distinguer ce que nous appelons "guess approximation" des techniques précédentes, du fait que l'on ne modifie ni l'instance ni l'ensemble des contraintes. La "guess approximation" consiste à demander à l'oracle des informations *approchées*. Ainsi, au lieu de demander à l'oracle r_I^* tel que $P_I(r_I^*) = V$, on demande $f(r_I^*)$ tel que $P_I(r_I^*) = V$, avec f un fonction de "contraction" vérifiant $|f(r)| \leq |r|$ pour tout r. Naturellement, cette technique est déjà contenue implicitement dans la définition 3.2 d'un algorithme interactif données en Section 3.2, et l'explicitation de la fonction f n'est en général pas nécessaire en pratique (voir les exemples ci-dessous). Cependant, il nous semblait important de nommer ce type de techniques afin de la présenter comme un ingrédient supplémentaire à la conception de questions élaborées.

Exemple 3.24: Un des exemples les plus courants (utilisé par exemple dans [Chekuri and Khanna, 2000, Jansen and Thöle, 2008]) de "guess approximation" consiste à demander (pour $\epsilon > 0$ fixé) $\tilde{O}$ tel que $(1-\epsilon)Opt \leq \tilde{O} \leq Opt$, au lieu de demander directement Opt.

Exemple 3.25: Pour le problème du *Multi Knapsack*. (voir la définition en Annexe, Section A.2.1), les auteurs de [Chekuri and Khanna, 2000] demandent à l'oracle, pour chaque ensemble d'objets U_i de même profit p_i utilisés dans un optimal, une valeur k_i telle que $k_i\alpha \leq p(U_i) = \sum_{x \in U_i} p_x \leq (k_i + 1)\alpha$, avec α fixé.

On constate qu'en pratique, expliciter la fonction f n'est pas forcément nécessaire. On remarque également que ces situations ne correspondent pas vraiment à un lemme structurel puisque l'on ne modifie pas l'instance et que l'on n'ajoute pas de contrainte.

Nous allons terminer la présentation de cette technique par un exemple détaillé d'application de la technique de "guess approximation". Encore une fois nous choisirons le problème du *Knapsack* (et plus précisément à l'algorithme interactif

de l'exemple 3.10) par soucis de simplicité, puisqu'il existe des approches plus efficaces par programmation dynamique couplée à des arrondis (par exemple des FPTAS dans [Mastrolilli and Hutter, 2006, Kellerer and Pferschy, 1998]).

Exemple 3.26: Soit $\{b_1, \ldots, b_k\}$ les k objets les plus chers dans un optimal pour une instance fixée du problème *Knapsack*. Soit $k\tilde{MV}_c$ l'algorithme qui

- découpe l'espace des prix $\{1, \ldots, p_{max}\}$ en tranches de largeur c, et demande le numéro a_i de la tranche contenant la valeur p_{b_i}, pour $i \in \{1, \ldots, k\}$ (autrement dit $a_i = \lfloor \frac{p_{b_i}}{c} \rfloor$).
- ajoute dans le sac l'ensemble G constitué pour chaque $i \in \{1, \ldots, k\}$ du plus petit (en taille) objet (restant) de la tranche a_i
- termine en ajoutant l'ensemble X déterminé avec l'algorithme glouton classique (qui ajoute le maximum entre l'objet le plus cher restant, et le choix des objets par densité décroissante)

Théorème 3.27. *Pour tout $k \in \mathbb{N}, \epsilon > 0$, l'algorithme $k\tilde{MV}_c$ est une $1 + \frac{1+(k+2)\epsilon}{k+1-(k+2)\epsilon}$-approximation, et peut être simulé en temps $\mathcal{O}(n(\frac{k}{\epsilon})^k)$ (alors que kMV est une $1+\frac{1}{k+1}$-approximation, simulée en temps $\mathcal{O}(n^{k+1})$).*

Preuve On adapte directement la preuve du théorème 3.17. Une instance est composée de n objets, de taille s_i et profit p_i pour $i \in \{1, \ldots, n\}$, et de la capacité du sac C. Pour tout ensemble d'objets Y nous noterons $p(Y) = \sum_{i \in Y} p_i$ et $s(Y) = \sum_{i \in Y} s_i$.

Considérons une solution optimale Opt utilisant strictement plus que k objets (sinon l'analyse est triviale). Notons $Opt = \{b_1, \ldots, b_k, a_1, \ldots, a_x\}$, avec $\frac{p_{(a_i)}}{s_{(a_i)}} \geq \frac{p_{(a_{i+1})}}{s_{(a_{i+1})}}$ pour $i \in \{1, \ldots, x\}$. L'algorithme $k\tilde{MV}_c$ construit donc la solution $S = G \bigcup X$, où G et X sont définis comme dans l'exemple 3.26. Soit $m = \min\{i \in \{1, \ldots, x\} | a_i \notin X\}$ (si m n'existe pas alors l'analyse est triviale). Soit $P_1^* = \{b_i, , i \in \{1, \ldots, k\}\}, P_2^* = \{a_i, i \in \{1, \ldots, m-1\}\}$ et $\Delta = \sum_{X'} s_i$ ou $X' \subset X$ est l'ensemble des objets de X ajoutés par kMV avant a_m (*i.e.* l'ensemble des objets de X plus denses que a_m, a_m non compris). En minorant simplement $p(G)$ par $p(P_1^*) - kc$, on obtient (de la même façon que pour la preuve du théorème 3.17) que $S - Opt \geq \frac{p_{a_m}}{s_{a_m}}(\Delta - s_{a_m} - C + s(P_1^*) + s(P_2^*) + s_{a_m}) - kc$. Or, a_m n'étant pas mis dans le sac, et les objets dans les tranches indiquées par l'oracle étant les plus petits (en taille) possible, on sait que $\Delta + s(P_1^*) + s(P_2^*) + s_{a_m} > \Delta + s(G) + s(P_2^*) + s_{a_m} > C$.

On obtient donc $S > Opt - p_{a_m} - kc$ d'une part et $S \geq p(G) + p_{a_m} \geq p(P_1^*) - kc + p_{a_m} \geq (k+1)p_{a_m} - kc$ d'autre part. Ainsi, on a $S > (k+1)(Opt - S - kc) - kc$, et donc $S > \frac{k+1}{k+2}Opt - kc$. On constate alors qu'avec $c = \frac{\epsilon}{k} p_{max}$, on obtient $S > Opt(\frac{k+1}{k+2} - \epsilon)$, et on obtient le ratio escompté.

Concernant le coût de simulation, on voit qu'avec une telle valeur le nombre de tranches est inférieur à $\lceil \frac{k}{\epsilon} \rceil$, d'où le coût annoncé. □

On constate donc sur cet exemple la différence entre la "guess approximation" et une question portant sur une instance simplifiée. En effet, si au lieu de considérer l'algorithme $k\tilde{MV}_c$ on modifiait en amont l'instance pour que les prix des objets soient arrondis au multiple (inférieur par exemple) de c, la majoration naïve conduirait à $Opt(I') \geq Opt(I) - |OPT(I)|c \geq Opt(I) - nc$, introduisant ainsi un facteur n au lieu

de k. Informellement, la "guess approximation" n'introduit donc de l'imprécision que sur la réponse de l'oracle, et non pas sur toute l'instance.

Application à un nouveau problème

Nous avons appliqué la technique de "guess approximation" dans [Bougeret et al., 2009a, Bougeret et al., 2009e] (voir le Chapitre 12) pour améliorer les résultats de [Bougeret et al., 2009b]. Ce résultat est résumé dans la Section 4.3.3, mais nous allons néanmoins en donner un aperçu ici. L'idée est de demander uniquement les j bits de poids fort d'un entier m^* (représentant un nombre de ressources allouées un algorithme) au lieu des $\lceil log(m^*) \rceil$ bits. Cette contraction permet d'avoir les valeurs exactes des "petits" m^* (ce qui s'avère critique dans notre algorithme pour ce problème où le ratio d'approximation est directement lié aux $\frac{m^*}{m}$, avec m le nombre décidé par l'algorithme), et des valeurs approchées à un ratio $1 + \frac{1}{2^{j-1}}$ pour les valeurs de m^* se codant sur plus de j bits.

Nous espérons que cette contraction "originale" (a notre connaissance, nous n'avons pas trouvé de formulations similaires), au sens où elle est définie simplement à partir du codage binaire, mènera à d'autres contractions similaires basées uniquement sur des réflexions sur le codage. Comme il l'est remarqué dans 4.3.3, cette contraction précise (j bits de poids forts) peut être reformulée (de façon moins directe) à l'aide de la technique de "guess approximation", et en utilisant des arrondis géométriques. On peut donc dire que ce type d'application de la technique de "guess approximation" permet d'exprimer simplement des arrondis géométriques.

3.5 Conclusion sur l'étendue et les limites du formalisme interactif

Le formalisme interactif permet d'exprimer plusieurs techniques classiques de conception d'algorithmes (et particulièrement de schémas d'approximation), et nous montre que le point commun entre toutes ces techniques est finalement un raisonnement en "deux temps", consistant d'abord à trouver quelle information supplémentaire demander à l'oracle, puis à étudier comment obtenir cette information. Cette séparation à tout d'abord un intérêt méthodologique puisqu'elle montre en quelque sorte d'où vient la difficulté du problème, et peut ainsi aider par exemple à caractériser des ensembles d'instances "faciles" (*i.e.* les instances pour lesquelles l'information demandée à l'oracle est facile à retrouver, ou peut être codée efficacement). D'autre part, les deux types de simulation d'algorithme interactif permettent d'exprimer deux techniques classiques de conception d'algorithmes (voir Section 3.4.1 et 3.4.2). Nous avons choisi de distinguer les notions de simplification de l'instance, de lemme structurel et de "guess approximation", même si les frontières sont parfois floues, afin de les présenter comme des outils supplémentaires pouvant être utilisés dans des algorithmes interactifs. Enfin, notons que même si les techniques abordées sont présentées dans le cadre de conception de schémas d'approximation, elles peuvent également servir à la conception d'algorithmes d'approximation "classiques", comme c'est le cas dans le résultat du Chapitre 11.

Nous allons à présent comparer plus en détail le formalisme interactif avec les deux techniques des Sections 3.4.1 and 3.4.2. Le but de ces comparaisons n'est pas

de cerner des cas "artificiels" pour lesquels les formulations classiques échouent. En effet, il serait peut-être possible d'étendre les définitions de ces techniques classiques suggérées dans [Schuurman and Woeginger, 2000, Shachnai and Tamir, 2007] afin de couvrir ces cas. Cependant, de telles extensions dénatureraient probablement ces techniques qui justement restent identifiables de part leur simplicité. L'objectif est donc plutôt de constater que le formalisme interactif permet de façon naturelle d'exprimer des cas qui ne correspondent pas à l'idée intuitive des techniques classiques présentées.

3.5.1 Comparaison entre simulation par énumération et techniques de type "structuring the output"

Rappelons tout d'abord que la méthode de "structuring the output" (ou des "outline scheme", ou "extending all possible solutions for small subsets") ne se limite pas à fixer les valeurs d'un sous-ensemble des variables. Il semble que ces trois techniques, formulées en terme d'algorithmes interactifs, consistent à demander des propriétés du type $P_I(w) = \exists Opt(I) | f(Opt(I)) = w$, pour $w \in R_I$, où f est une fonction associant à chaque solution réalisable S un "résumé" $f(S)$. On constate tout d'abord que ces propriétés se restreignent à des informations concernant un optimal du problème considéré, alors qu'il paraît tout à fait possible de poser des questions concernant un optimal d'un autre problème. On pourrait par exemple, pour des problèmes d'ordonnancement, poser des questions à l'oracle sur un remplissage optimal *en aire* d'un certain ensemble de machines sur une période de temps fixée. Une des conséquences est que ces techniques de type "outline scheme" suggèrent que l'ensemble des $\{f^{-1}(w), w \in R_I\}$ doit être une partition de l'ensemble des solutions réalisables, afin d'être sûr de capturer un optimum global, et que l'on doit fournir pour chaque élément de la partition un représentant approximant un optimum local. Or, il est possible qu'un algorithme interactif simulé par énumération ne "se base" pas sur une partition. On peut prendre l'exemple d'un problème d'ordonnancement sur des machines ayant des vitesses différentes, pour lequel on demanderait des informations pour ordonnancer la plus grande tâche sur une des x machines les plus rapides. Dans ce cas, on ne fournira lors de l'énumération un représentant uniquement pour les x régions correspondantes. Autrement dit, un représentant n'approximera pas uniquement un optimum local situé dans sa région. Remarquons également que certaines des régions de ces partitions peuvent être vides, comme par exemple avec $f(S)$ donnant la valeur de la fonction objectif de la solution S, où il est possible que $\{S | f(S) = w\}$ soit vide pour certains w (alors que dans la formulation interactive aucune spécification n'est donnée pour les "mauvais" w).

Ainsi, la formulation de type "structuring the output" peut être bien adaptée dans des cas simples, où l'idée d'avoir un représentant par région est intuitive. Cependant, dans des cas plus complexes (mais réellement utilisés, par exemple [Bougeret et al., 2009b]), on peut avoir une partition dans laquelle les représentants des régions n'appartiennent pas à celles-ci, les représentants peuvent couvrir plusieurs régions, ou certaines régions sont vides. Dans ce type de cas, il semble donc plus naturel d'utiliser le formalisme oracle.

3.5.2 Comparaison entre simulation par recherche séparée et techniques de type "extending an optimal solution for a single subset"

La technique de "extending an optimal solution for a single subset" suggère que l'on résout *optimalement* une instance de taille *constante*. Or, un algorithme interactif utilisant la simulation séparée peut par exemple ne demander à l'oracle qu'une approximation, (pour de plus une instance de taille quelconque), ou même une propriété n'étant pas une solution du problème de base. D'autre part, même lorsque l'on se restreint au cas de la résolution exacte d'une instance de taille constante, la technique de "extending an optimal solution for a single subset" suggère que l'on "étend" cette solution partielle, et donc que les variables déjà fixées ne seront pas modifiées. Cette contrainte n'est pas suggérée par la définition d'un algorithme interactif simulé par recherche séparée, dans laquelle on demande simplement la connaissance d'une telle solution.

3.5.3 Résultats négatifs en optimisation offline

Rappelons qu'une forme classique (*i.e.* sans oracle) de résultats négatifs en approximation offline est par exemple $\mathcal{LB}(\mathcal{A}, \emptyset) \geq \rho$ à moins que $\mathcal{P} = \mathcal{NP}$, où $\mathcal{A}$ est l'ensemble des algorithmes polynomiaux, et $\rho > 1$ représente un ratio d'approximation. Une extension naturelle de ces résultats consistant à étudier des bornes du type $\mathcal{LB}(\mathcal{A}, \{r \ t.q. \ |r| = x\})$ (comme en algorithmique distribué ou online) semble difficile, puisqu'il faudrait pour cela réussir à donner un nombre minimal de bits d'informations nécessaire pour que tout algorithme atteigne une certaine performance.

Nous avons donc considéré (voir Chapitre 12) des résultats du type $\mathcal{LB}(A_0, \mathcal{R})$, où A_0 est *un* algorithme fixé et $\mathcal{R}$ un ensemble d'informations dépendant de A_0. Plus précisément, l'algorithme interactif A_0 de [Bougeret et al., 2009b] demande des informations du type "quelle est le traitement (dans un optimal) des k objets parmi n vérifiant une condition particulière c?". Dans ce cas, un ensemble naturel $\mathcal{R}$ d'informations "équivalentes", et pour lesquelles A_0 est défini, est l'ensemble des informations du type "quel est le traitement dans un optimal des k objets (parmi n) vérifiant une propriété particulière c'" avec c' une propriété quelconque. En effet, pour tout $r \in \mathcal{R}$ la quantité d'information demandée à l'oracle est la même (linéaire en k). Nous avons donc construit une instance I pour laquelle pour tout sous-ensemble de k objets sélectionnés (et donc pour tout $r \in \mathcal{R}$), le ratio de A_0 est supérieur à une certaine valeur. Ce type de résultat peut donc servir en particulier à savoir si une information $r_0 \in \mathcal{R}$ est "la meilleure" parmi un large ensemble. Il serait donc intéressant de fournir ce type de résultats pour d'autres problèmes, ou de les généraliser à des ensembles d'algorithmes ou d'informations plus grands.

Remarquons enfin qu'il existe un autre type de résultat négatifs, valable uniquement dans le contexte où les problèmes considérés ont des entrées difficiles à coder raisonnablement (voir la fin de Section 3.4.2 pour plus de détails). Il existe dans [Hausmann et al., 1981] des résultats qui minorent le nombre d'appels (à un oracle fixé) nécessaires à tout algorithme dépassant une performance fixée. Cependant, rappelons que dans ce contexte l'oracle sert à fournir des informations sur l'instance,

qui n'est pas accessible à l'algorithme. Cette situation s'éloigne donc de notre contexte dans lequel l'instance est disponible.

3.5.4 Limite de l'interaction simulée par énumération ?

Nous avons comparé le formalisme interactif avec plusieurs techniques classiques de conception de schémas d'approximation. Or, une des techniques majeures de ce domaine n'a pas été spécifiquement abordée : la programmation dynamique. Associée à des simplifications d'instances, la programmation dynamique permet "souvent" (voir [Woeginger, 1999] pour l'étude de cette question) d'obtenir des FPTAS . Il est par exemple bien connu que, pour le problème du *Knapsack*, un arrondi des prix des objets associé à une programmation dynamique simple permet d'obtenir une FPTAS plus efficace que l'exemple 3.10. Il nous a semblé que le formalisme interactif ne permette pas une expression différente et simple de la technique de programmation dynamique. Cependant, cette question reste ouverte, ainsi que la recherche d'un exemple de FPTAS grâce à un algorithme interactif utilisant une information paramétrable, et simulé par énumération.

3.5.5 Pistes possibles

Le formalisme interactif suggère plusieurs ouvertures que nous allons aborder ici.

Une première remarque concerne la simulation par énumération, et ce que nous appelons "l'énumération intelligente". Ainsi, même si les bornes sur $|R_I|$ sont généralement pessimistes, il ne faut pas négliger en pratique l'importance de bonnes heuristiques pour énumérer R_I efficacement (par exemple avec des règles de "coupure"). Notre but n'est pas de dévier vers la littérature très large des heuristiques en ordonnancement, mais plutôt de rappeler qu'il faut associer aux algorithmes interactifs des techniques de conception d'heuristiques très classiques, comme par exemple celle suggérée ci-dessous. Considérons l'algorithme interactif que nous avons proposé dans le Chapitre 6 pour le problème $Q||C_{max}$. Dans cet algorithme on demande à l'oracle les k plus grosses tâches ordonnancées sur chaque machine dans un optimal. On pourrait donc appliquer une technique de type "branch and bound" qui, étant donné une allocation "partielle" de $k_i \leq k$ tâches sur la machine m_i pour tout $i \in \{1, \ldots, m\}$, vérifierait que cette allocation partielle est correcte en calculant une borne inférieure (en ordonnançant avec préemption par exemple) et en la comparant au meilleur résultat courant.

Une deuxième remarque concerne la définition de la notion d'algorithme interactif. Dans la définition proposée en Section 3.2, ainsi que dans la quasi-totalité des exemples abordés, l'interaction avec l'oracle a lieu au début du calcul. Autrement dit, on ne retrouve pas l'idée d'une "discussion" entre l'oracle et l'algorithme (comme par exemple dans certains classes de complexité), qui serait composée d'une alternance entre calcul d'une question par l'algorithme, et réponse par l'oracle. Cette "vraie" interaction pourrait permettre d'éviter de demander à l'oracle toute l'information au début dans les cas où l'algorithme se comporte "bien". En particulier, il existe des cas où en examinant la propriété réellement utilisée dans l'analyse, on s'aperçoit que l'on pourrait ne demander des informations à l'oracle que lorsque cela est vraiment nécessaire. Par exemple pour le problème $P||C_{max}$, au lieu de toujours demander un ordonnancement

optimal des k tâches les plus longues comme dans l'exemple 3.11, on pourrait (pour $j \leq k$) :

1. demander à l'oracle un ordonnancement optimal des j tâches les plus longues (avec j valant initialement 0)
2. terminer l'ordonnancement avec un algorithme classique (type LPT)
3. si la tâche déterminant le makespan est une des k plus grandes, incrémenter j et recommencer depuis l'étape 1

En effet, on voit dans la preuve du Théorème 3.19 que la seule chose importante est que p_x (la taille de la tâche déterminant le makespan) soit petite devant l'optimal, et que le fait que la tâche x ne soit pas une des k plus grandes assure cette propriété. On peut remarquer deux choses concernant cet algorithme et la nature de l'interaction. La première chose est que cet algorithme pourrait se réécrire avec le formalisme "faiblement" interactif utilisé dans ce manuscrit (voir Définition 3.2). En effet, on pourrait considérer l'algorithme qui demande (dès le début) j, et un ordonnancement optimal σ_j^* de X_j (les j plus grandes tâches), tels que : j est le minimum tel que (tout) ordonnancement optimal de X_j, "prolongé" par l'exécution de LPT, aboutira un ordonnancement où la tâche déterminant le makespan n'est pas une des k plus grande. Cependant, on constate que cette formulation est moins naturelle. Ainsi le formalisme "vraiment" interactif permettrait-il d'exprimer simplement ce type d'algorithmes. La deuxième remarque concerne la simulation de ce type d'algorithmes. En effet, si la simulation par recherche séparée semble facile à implémenter, la simulation par énumération pose la question de l'ordre dans lequel on simule les réponses (largeur contre profondeur). Pour $k = 2$ par exemple : faut-il d'abord énumérer toutes les machines pour $j = 1$, ou faut-il fixer une machine pour $j = 1$ et énumérer d'abord les allocations de la seconde plus grande tâche ? D'autre part, d'un point de vue théorique il semble que gagner un ordre de grandeur sur la complexité de la simulation par énumération nécessite une analyse difficile. Cependant, cette notion de "vraie" interaction semble une piste intéressante à étudier.

Enfin, on peut citer deux perspectives naturelles à long terme. La première concerne l'utilisation de l'aléa, évidemment directement inspirée du modèle $\mathcal{PCP}$. Informellement, l'idée serait donc de ne lire qu'une petite portion (choisie au hasard) de la preuve de l'oracle, et d'étudier l'espérance de l'algorithme interactif ainsi obtenu. Considérons par exemple le problème d'hybridation d'algorithmes abordé dans le Chapitre 12, consistant à partager m processeurs entre k algorithmes disponibles. Dans cet exemple l'algorithme demande à l'oracle le numéro des g algorithmes "les plus utilisés" dans l'optimal (notons X^* cet ensemble d'algorithme), ainsi que le nombre de processeurs qui leurs sont alloués. Une idée naturelle serait donc de plutôt choisir au hasard un sous ensemble X de g (ou même $g' = g + \epsilon$) algorithmes, pour lesquels on demande à l'oracle le nombre de processeurs utilisés. On réduirait donc la taille de l'information de $g(log(k) + log(m))$ à $glog(m)$, moyennant l'étude de l'espérance de la dégradation de performance de l'algorithme selon l'ensemble X.

Une autre perspective consisterait à continuer dans la direction de l'efficacité du codage, à travers l'étude de techniques de compression "sans perte" (qui amélioreraient en moyenne la longueur de l'information donnée par l'oracle), ou en appliquant la technique "guess approximation" (avec perte donc).

Chapitre 4

Bilan sur l'interaction quantifiée

4.1 Comparaison des résultats interactifs selon les domaines

A partir de l'analyse faite aux chapitres 2 et 3, on constate que les résultats interactifs (avec un oracle) peuvent être inscrits dans deux démarches : étudier séparément le problème consistant à retrouver l'information de l'oracle, ou plutôt quantifier la taille des réponses fournies par l'oracle. La majorité des exemples présentés en algorithmique distribuée, online et en complexité sont dans une approche de type quantification, alors que les deux méthodes naturelles de simulation d'algorithme interactif en optimisation offline permettent justement les deux démarches précédentes.

Cette différence peut s'expliquer en partie en observant le type d'informations demandées. Dans les domaines où par définition la totalité de l'instance n'est *pas accessible* à l'algorithme (en algorithmique distribuée ou online), de nombreux résultats utilisent des informations de type "découverte" (voir Section 2.3.4) qui révèlent simplement à l'algorithme une partie cachée de l'instance (excepté le résultat abordé en Section 2.3.5, qui est plus difficile à exploiter en pratique). En effet, l'utilisation d'une information de type découverte est souvent justifiée en remarquant que le modèle dans lequel rien n'est connu de l'algorithme est trop pessimiste en pratique. Ainsi, de tels résultats montrent que l'utilisation d'informations simples (qu'il serait potentiellement possible de fournir réellement aux algorithmes) suffit à améliorer les meilleures performances connues, ou même atteignables, dans le modèle pessimiste. Ainsi, l'étude du problème consistant à retrouver séparément ces informations n'a pas d'intérêt, puisque ce problème sera soit impossible à résoudre, soit trivial selon ce que l'on suppose connu.

Les conséquences des résultats positifs, consistant à obtenir une certaine performance avec un algorithme interactif, sont plus immédiates en optimisation offline puisque l'on peut toujours dériver un algorithme classique à partir d'un algorithme interactif, moyennant bien sûr du temps de calcul. Les résultats négatifs consistant à étudier $\mathcal{LB}(\mathcal{A}, \mathcal{R})$ sont souvent prouvés pour $\mathcal{A} = \mathcal{U}$ (l'ensemble de tous les algorithmes) dans les domaines où l'instance n'est pas connue entièrement de l'algorithme, l'adversaire ayant des degrés de liberté liés à ces parties inconnues. D'autre part, la définition d'une notion de taille permet d'obtenir des résultats négatifs sur de grands ensembles $\mathcal{R}$ (contenant toutes les informations de même taille par exemple), et conduit alors à des résultats du type "sensibilité à l'information".

Enfin, deux catégories principales de modèles interactifs se dégagent. Dans certains cas, la notion d'interactivité se limite à un ajout d'information initial, comme dans :

- les résultats d'algorithmique distribuée que nous avons étudiés
- les résultats d'algorithmique online utilisant des informations que nous avons appelées statiques
- le modèle $\mathcal{PCP}$ utilisé en complexité
- le modèle d'algorithme interactif pour l'optimisation offline que nous avons proposé dans le Chapitre 3 (ainsi que dans la majorité des résultats de ce chapitre)

A l'inverse, l'interactivité au sens d'un dialogue entre algorithme et oracle (dans lequel chaque question posée à l'oracle dépend de toutes les questions et réponses précédentes) est utilisée dans :

- les résultats d'algorithmique online utilisant des informations que nous avons appelées dynamiques
- certains algorithmes interactifs d'optimisation offline lorsqu'ils sont simulés par recherche séparée (*i.e.* lorsque l'on remplace simplement l'oracle par une sous procédure, pouvant donc être appelée plusieurs fois au cours du calcul)
- certains modèles utilisés en complexité, qui semblent s'éloigner de notre problématique

4.2 Conclusions générales sur l'interactivité

La problématique traitée est l'utilisation de l'interactivité pour la construction d'algorithmes ayant des garanties de performance (en particulier pour des problèmes d'ordonnancement). Nous avons montré dans le Chapitre 2 quels étaient les principaux résultats de ce type existant dans d'autres domaines que l'ordonnancement offline, en présentant systématiquement les résultats sous le point de vue interactif, et en cherchant à classifier les situations (information paramétrables/non paramétrables, bornes inférieures sur des ensembles d'algorithmes et d'informations différentes) pour dégager différentes démarches possibles. Nous avons alors abordé plus en détail dans le Chapitre 3 la conception d'algorithmes d'approximation en ordonnancement, sous cet angle original de l'interaction. Nous avons défini un formalisme pour la notion d'algorithme interactif (en optimisation offline), et montré quels étaient les liens entre ce formalisme et les techniques classiques de conception d'algorithme d'approximation (et particulièrement de PTAS).

Les cas simples "extrêmes" dans lesquels l'utilisation de l'interactivité ne semble pas particulièrement plus aisée que la formulation standard ont été distingués autant que possible. Par exemple en algorithmique distribuée ou online lorsque l'on étudie une information non paramétrable (voir Sections 2.2.2 et 2.3.2), ou pour la conception d'algorithmes d'approximation lorsque les informations demandées sont simples et correspondent bien aux définitions des techniques de conception de PTAS des Sections 3.4.1 et 3.4.2. En revanche, nous avons vu que la vision interactive permettait dans d'autres cas d'obtenir des résultats complètement nouveaux (par exemple lorsque la quantité d'information sert à mesurer la difficulté des problèmes), ou d'exprimer de façon simple plusieurs techniques classiques de conception d'algorithmes d'approximation, dont certaines extensions (la "guess approximation" par exemple) sont particulièrement faciles à caractériser en utilisant ce formalisme interactif.

Quel que soit le domaine, il ressort que l'utilisation de l'interactivité permet :

- d'aborder les problèmes différemment (par exemple en algorithmique distribuée et online lorsqu'il existe des bornes inférieures trop pessimistes, ou en ordonnancement lorsque la recherche d'un codage efficace de l'information de l'oracle permet de comprendre ce qui est négligeable dans une instance)
- d'isoler la difficulté (permettant par exemple en optimisation offline de caractériser simplement des classes d'instances faciles à approximer).

4.3 Présentation des nouveaux résultats obtenus

Nous allons résumer dans cette section les nouveaux résultats obtenus sur des problèmes d'ordonnancement. Un résumé en anglais sera donné au Chapitre 5. Nous signalerons parmi tous ces résultats ceux utilisant l'interactivité. Les détails des problèmes étudiés ainsi que des algorithmes proposés sont donnés dans les Chapitres 6 à 11. Ces articles n'utilisant pas toujours les termes définis dans le Chapitre 3, nous allons ici resituer ces contributions dans le cadre de travail du Chapitre 3.

4.3.1 Ordonnancement de tâches séquentielles sur des machines ayant des vitesses différentes

Le premier résultat présenté en Section 6 concerne le problème $Q||C_{max}$ (voir la définition en Annexe, ou dans le chapitre correspondant). Nous avons fourni pour ce problème un algorithme interactif ayant un ratio d'approximation de $1+\frac{1}{k+2}$ (en temps très rapide), en demandant l'indice des k (pour k fixé) plus grandes tâches ordonnancées sur chaque machine, dans un optimal fixé. L'information demandée est donc de type paramétrable, et la simulation de cet algorithme par énumération aboutit à une PTAS lorsque le nombre de machines est fixé. Ce schéma d'approximation a l'avantage d'être simple par rapport à d'autre schémas existants (qui sont par contre plus rapides pour des cas asymptotiques, mais parfois beaucoup plus coûteux en espace), et la démarche interactive nous montre bien d'où vient la difficulté du problème, suggérant ainsi d'autres points à étudier. Par exemple, déterminer (ne serait-ce que pour $k = 1$) des classes d'instances pour lesquelles cette information peut être construite rapidement serait une nouvelle façon de caractériser des instances "faciles" de ce problème.

4.3.2 Ordonnancement de tâches parallèles dans des environnement hiérarchiques

Les résultats présentés dans les Chapitres 7 à 11 concernent le problème d'ordonnancement de n tâches sur N clusters de machines (le cluster i ayant m_i machines), en minimisant le makespan. Les tâches sont parallèles et rigides, signifiant que la tâche j nécessite q_j machines (du même cluster) pendant p_j unités de temps. Ce problème est appelé "Multiple Strip Packing" et est noté MSP. Nous allons noter MSP^x_y les différentes variantes possibles de ce problème. La variable x indique si les tâches doivent être allouées sur des processeurs *contigus* ($x = c$) ou non ($x = nc$). Dans la version

contiguë, les tâches sont donc considérées comme des rectangles "indivisibles". La notation y indique si les clusters sont *réguliers* ($y = r$), signifiant que tous les m_i sont égaux, où si les clusters sont différents ($y = d$). Une présentation plus détaillée de ces variantes est disponible en Annexe, en Section A.2.2.

Du point de vue des résultats d'approximabilité sur ces problèmes, il est montré (avec une réduction depuis 2-partition) dans [Zhuk, 2006] que $MSP_r^{nc/c}$ est 2-inapproximable en temps polynomial à moins que $P = NP$, et ce même pour $N = 2$. Ainsi, toutes les variantes sont 2-inapproximables. D'autre part, deux types de résultats positifs ont été fournis. Il existe deux 3-approximation rapides (utilisables en pratique) pour deux variantes de MSP_d^{nc} (une fournie dans [Schwiegelshohn et al., 2008] dans laquelle les durées des tâches sont inconnues, et l'autre [Dutot et al., b] dans laquelle on optimise le makespan global sans dégrader les ordonnancements "initiaux"). D'autre part, une remarque formulée dans [Ye et al., 2009] montre que l'on peut atteindre un ratio de $2+2\epsilon$ pour $MSP_r^{nc/c}$ en appliquant simplement une PTAS pour $P||C_{max}$ pour répartir l'aire des tâches entre les clusters (on est donc à $1+\epsilon$ de la meilleure répartition d'aire possible), puis en appliquant la 2-approximation de [Steinberg, 1997] sur chaque cluster (aboutissant ainsi à un ratio de $2+2\epsilon$). Cette remarque peut être étendue (voir le Chapitre 10) au cas $MSP_d^{nc/c}$ avec l'hypothèse que $\max_j q_j \leq \min_i m_i$, en appliquant une PTAS pour $Q||C_{max}$. Cependant, ces résultats sont extrêmement coûteux (voir en page 79 le coût des PTAS pour $Q||C_{max}$), et donc difficilement utilisables en pratique. L'objectif est donc de résoudre "directement" ces problèmes, en cherchant bien sûr à améliorer les bornes précédentes.

Le résumé des tous les résultats que nous avons obtenus (y compris ceux n'utilisant pas d'oracle) est présenté dans la Figure 4.1.

Du point de vue de l'utilisation d'algorithmes interactifs, la $\frac{5}{2}$-approximation du Chapitre 11 pour le $MSP_d^{nc/c}$ (valable également pour une extension de $MSP_r^{nc/c}$ où les clusters ont des vitesses différentes) est basée sur un algorithme interactif. Signalons que la remarque précédente permettant l'obtention d'un ratio $2 + 2\epsilon$ en temps polynomial ne s'applique ici (comme il est expliqué au début du Chapitre 11). Cet algorithme demande pour chaque cluster l'indice de l'éventuelle unique tâche ordonnancée prenant plus de la moitié des processeurs pendant plus de la moitié du temps total. L'intuition de cette question à l'oracle est donnée dans le chapitre correspondant. Cet algorithme est ensuite amélioré par des techniques classiques de simplification de l'instance par arrondis (voir Section 3.4.3). On remarque que l'information utilisée dans ce résultat est *non* paramétrable, et que le ratio est fixé. Ce cas est intéressant au sens où $|O(I)|$ (la taille de l'information de l'oracle) peut varier beaucoup sur l'ensemble des instances de mêmes paramètres n, N et m_i. Ainsi, même si la majoration de $|O(I)|$ conduit, lorsque l'on simule par énumération, à un algorithme non polynomial dans le pire cas, il est facile de dégager de nombreuses classes d'instances "raisonnables" pour lesquelles l'algorithme sera utilisable en pratique.

4.3.3 Distribution de ressources dans un portfolio d'algorithmes

Les résultats présentés dans le Chapitre 12 concernent le problème d'allocation de ressources dans un portfolio (le "discrete Resource Sharing Scheduling Problem", noté

Variante	Ratio	Remarques	Source
$MSP_r^{nc/c}$	$2+\epsilon$	Par application de deux résultats en "boîtes noires", (nécessite de résoude $P\|\|C_{max}$ avec un ratio $1+\frac{\epsilon}{2}$)	[Ye et al., 2009]
MSP_r^{nc}	$\frac{5}{2}$	Rapide + Suggestion méthode pour $\frac{1}{1-\alpha}+\beta$, $0<\alpha<1$, $0<\beta<1$	Chapitre 8, [Bougeret et al., 2010b]
MSP_r^{nc}	$\frac{7}{3}$	Rapide, utilisation de la méthode suggérée dans le Chapitre 8	Chapitre 9
MSP_r^{nc}	2	**Nécessite $\max_j q_j \leq \frac{m}{2}$** Algorithme rapide, utilisation de la méthode suggérée dans le Chapitre 8	Chapitre 9
$MSP_r^{nc/c}$	2	Algorithme coûteux	Chapitre 7, [Bougeret et al., 2009d]
$MSP_r^{nc/c}$	AFPTAS	Constante additive en $\mathcal{O}(\frac{1}{\epsilon^2})$ et $\mathcal{O}(1)$ si N est grand	Chapitre 7, [Bougeret et al., 2009d]
MSP_d^{nc}	3	Rapide, algorithme non clairvoyant	[Schwiegelshohn et al., 2008]
MSP_d^{nc}	3	Rapide, avec contraintes supplémentaires de préallocation	[Dutot et al., b]
$MSP_d^{nc/c}$	$2+\epsilon$	**Nécessite $\max_j q_j \leq \min_i m_i$** En adaptant la remarque de [Ye et al., 2009], (nécessite de résoude $Q\|\|C_{max}$ avec un ratio $1+\frac{\epsilon}{2}$)	Chapitre 10, [Bougeret et al., 2010a]
$MSP_d^{nc/c}$	$\frac{5}{2}$	**Nécessite $\max_j q_j \leq \min_i m_i$** Rapide	Chapitre 10, [Bougeret et al., 2010a]
$MSP_d^{nc/c}$	$\frac{5}{2}$	L'information devinée nécessite une énumération non polynomiale dans le pire cas, s'applique avec des vitesses différentes pour $MSP_r^{nc/c}$	Chapitre 11

FIG. 4.1: Principaux résultats sur les problèmes de type MSP

$dRSSP$). Ce problème est issu de la communauté de l'hybridation d'algorithmes. La motivation de base de l'hybridation est la suivante : il arrive souvent pour un problème donné que de nombreux algorithmes existent, et que chacun d'entre eux ne soit efficace que pour une sous classe d'instances. L'idée est alors de trouver comment "combiner" ces algorithmes pour être efficace sur n'importe quelle instance. Dans le $dRSSP$, on dispose d'un ensemble de k algorithmes (un "portfolio" d'algorithmes), de n instances "représentatives" (*i.e.* un benchmark), et l'on connaît le coût de chacun des k algorithmes sur chacune des n instances du benchmark, et ce avec $i \in \{1, \ldots, m\}$ processeurs. Le but est de trouver comment distribuer les m ressources/processeurs (en donnant $S_i \in \mathbb{N}$ ressources à l'algorithme i) aux k algorithmes. Etant donné un partage des ressources défini, on résout chaque instance du benchmark en parallèle sur les k algorithmes, en supposant que tous les algorithmes s'arrêtent dès que l'un d'entre eux a trouvé la solution. Le but est de minimiser le temps total de résolution des n instances. Ce problème avait uniquement été étudié pour des partages fractionnaires ($S_i \in \mathbb{R}^+$) [Sayag et al., 2006], et nous avons donc étendu l'étude au cas discret. Le Chapitre 12 contient l'article [Bougeret et al., 2009a] qui généralise nos précédents résultats présentés dans [Bougeret et al., 2009b, Bougeret et al., 2009e].

Un algorithme rapide et simple (MA : Mean Allocation) consiste à partager équitablement les ressources entre les algorithmes, *i.e.* à choisir $\forall i, S_i = \lfloor \frac{m}{k} \rfloor$. Cet algorithme est une k approximation. Nous avons montré qu'en se basant simplement sur MA et en choisissant la "bonne" information à demander à l'oracle, on peut obtenir de biens meilleurs ratios. Plusieurs informations différentes sont considérées : étant donné $g \in \{1, \ldots, k-1\}$, l'algorithme demande à l'oracle le nombre de processeurs alloués à un certain ensemble E_g de g algorithmes dans une solution optimale. Selon l'ensemble E_g choisi, plusieurs compromis sont possibles, dont une $(k-g)$ approximation avec une information de longueur $glog(m)$, et une $\frac{k}{g+1}$ approximation avec une information de longueur $g(log(k)+log(m))$. Autrement dit, nous avons fourni un algorithme interactif, simulé par énumération, et analysé plusieurs informations paramétrables différentes.

Ces résultats ont ensuite été améliorés grâce à la technique dite de "guess approximation" présentée en Section 3.4.4. En effet, on remarque que la connaissance exacte du nombre de processeurs utilisés (dans un optimal) est critique lorsque ce nombre est petit (par exemple 1 processeur au lieu de x peut mener à un ratio de x), mais peu importante lorsque celui-ci est grand. Ainsi, on peut raffiner ce résultat en ne demandant que les j bits de poids fort (dans l'écriture en binaire) de ces quantités. Etant donné un nombre $S_i^* = a_i 2^{d_i} + b_i$, avec a_i codé sur j bits, et $b_i \leq 2^{d_i} - 1$, on ne demandera que les valeurs de a_i et d_i, et on considérera que $b_i = 0$. Ainsi, on passe d'un codage en $log(m)$ bits à un codage en $j + log(log(m) - j)$ bits, avec un ratio de

$$\frac{S_i^*}{a_i 2^{d_i}} \leq 1 + \frac{2^{d_i}}{a_i 2^{d_i}} \leq 1 + \frac{1}{2^{j-1}}$$

Le ratio étant bien sûr de 1 si S_i^* est codée sur moins de j bits. On remarque qu'une découpe géométrique de $\{1, \ldots, m\}$ de raison $r = 1 + \frac{1}{2^{j-1}}$ donne des résultats semblables. Pour avoir les mêmes propriétés, il faut donc demander exactement les valeurs inférieures entre 1 et 2^j, et demander pour les autres valeurs le numéro de la "tranche" dans laquelle elles sont contenues. Le nombre de tranches a_{max} vérifie donc $2^j r^{a_{max}} \geq m$, soit $a_{max} \geq \frac{log(m)-j}{log(1+\frac{1}{2^{j-1}})} \approx 2^{j-1}(log(m)-j)$. Ainsi, le nombre de cas à énumérer dans ces

deux formulations est comparable, même si la formulation directement sur le codage paraît plus simple.

Enfin, le dernier résultat obtenu est un résultat négatif, directement inspiré des notions du type "sensibilité à l'information" abordées dans le Chapitre 2. Nous avons envisagé un nouveau type de résultat négatif, visant à étudier s'il existe une information utile pour *un* algorithme fixé (ce résultat est donc moins général que ceux du distribué formulés sur l'ensemble de tous les algorithmes et l'ensemble des informations de taille fixée). Ainsi, pour le $dRSSP$ plusieurs ensembles E_g "naturels" semblent possibles : les g algorithmes ayant eu le plus (ou le moins ?) de processeurs dans une solution optimale, ou les g algorithmes les plus "utilisés" dans une solution optimale. La question de la sensibilité à l'information pourrait donc être ici de déterminer une borne inférieure sur le meilleur ratio possible, quelque soit la façon de choisir l'ensemble E_g. Nous avons démontré dans le Chapitre 12 que le meilleur ratio possible avec l'algorithme MA est de $\frac{k-g}{g+1}$, ce qui est proche du ratio de $\frac{k}{g+1}$ obtenu.

Deuxième partie

New results

Chapitre 5

Outline of the results

We summarize here the main results obtained on scheduling and packing problems. Chapter 6 is a result on a classical sequential scheduling problem (usually denoted $Q||C_{max}$), and serves as an introduction to our methods. In Section 5.1.1 we present results of Chapter 7 to 9 on multiple strip packing problems where strips have the same size. In section 5.1.2 we present results of Chapter 10 to 11 on multiple strip packing where strips have different sizes. Section 5.2 summarizes results of Chapter 12 on an algorithm portfolio management problem.

5.1 Multiple Strip Packing

In Chapter 7 to 11 we address several variants of Multiple Strip Packing problem (MSP). In these problems, the goal is to schedule n jobs on N clusters (cluster i owns m_i machines), while minimizing the makespan. Jobs are rigid, meaning that job j requires q_j machines (belonging to the same cluster) during p_j units of time. We denote all the variants of this problem by MSP_y^x.

The x parameter is set to c to denote the *contiguous* version where jobs must be allocated on contiguous indexes of processors (*i.e.* jobs are considered as rectangles), and is set to nc otherwise. Of course an algorithm for the non contiguous case is not valid for the contiguous one, as it does not respect the contiguous constraint. Notice also that a result for contiguous case may also not apply to non contiguous case, as the approximation ratios are not preserved. Indeed, an r-approximation of the contiguous optimal $Opt_c(I)$ is not necessarily an r-approximation of the non contiguous optimal $Opt_{nc}(I)$, as $Opt_{nc}(I) \leq Opt_c(I)$. However, in a lot of cases contiguous results do apply to the non contiguous case, as most of proofs are based on bounds of $Opt_{nc}(I)$ (typically area bounds). Finally, we will say that an algorithm applies for $MSP_y^{nc/c}$ when it applies to both contiguous and non contiguous versions. The y parameter is set to r to denote the *regular* version where all the clusters have the same number of machines (all the m_i are equal), and is set to d (for *different*) otherwise. Thus, four variants of the MSP possible are possible.

As shown in [Zhuk, 2006] using a gap reduction from the 2 partition problem, all these variants are 2-innaproximable in polynomial time unless $\mathcal{P} = \mathcal{NP}$, even for $N = 2$. The main previous positive results are the following :

- a 3-approximation in [Schwiegelshohn et al., 2008] for MSP_d^{nc} (where processing time of jobs is not known in advance)
- a $2+\epsilon$-approximation in [Ye et al., 2009] for $MSP_r^{nc/c}$ obtained simply using two classical results as "black boxes" (this requires solving $P||C_{max}$ with a ratio $1+\frac{\epsilon}{2}$)
- a $2+\epsilon$-approximation algorithm *for a special case of* $MSP_d^{nc/c}$ where every job fits in every cluster (*i.e.* where $\max_j q_j \leq \min_i m_i$), obtained in Chapter 10 using the same principle as in the previous case (this requires solving $Q||C_{max}$ with a ratio $1+\frac{\epsilon}{2}$)

Our algorithms achieve several ratios between 2 and 3, and most of them have a very low computational complexity. Let us now detail results according to the regular and different case.

5.1.1 Results for Multiple Strip Packing in the "regular" case

Our results on MSP_r^x are the following :

- in Chapter 7 we present a costly 2 approximation and an $AFPTAS$ for $MSP_r^{nc/c}$
- in Chapter 8 we present a fast $\frac{5}{2}$-approximation for the MSP_r^{nc} problem
- in Chapter 9 we improve the results of Chapter 8 on the MSP_r^{nc} problem by providing fast algorithms for with better ratios ($\frac{7}{3}$ and 2 for a special case).

All the results about MSP_r^x problems are presented in Figure 5.1.

Problem	Ratio	Remarks	Source
$MSP_r^{nc/c}$	$2+\epsilon$	Obtained by direct application of two classical existing results (need solving $P\|\|C_{max}$ with a ratio $1+\frac{\epsilon}{2}$)	[Ye et al., 2009]
MSP_r^{nc}	$\frac{5}{2}$	Fast algorithm, and "framework" suggestion for possibly getting $\frac{1}{1-\alpha}+\beta$ ratio, $0<\alpha<1$, $0<\beta<1$	Chapter 8, [Bougeret et al., 2010b]
MSP_r^{nc}	$\frac{7}{3}$	Fast algorithm that uses ideas suggested in Chapter 8	Chapter 9
MSP_r^{nc}	2	**Requires $\max_j q_j \leq \frac{m}{2}$** Fast algorithm that uses ideas suggested in Chapter 8	Chapter 9
$MSP_r^{nc/c}$	2	Costly algorithm	Chapter 7, [Bougeret et al., 2009d]
$MSP_r^{nc/c}$	AFPTAS	Additive constant in $\mathcal{O}(\frac{1}{\epsilon^2})$, and in $\mathcal{O}(1)$ for large values of N	Chapter 7, [Bougeret et al., 2009d]

FIG. 5.1: Main results on MSP_r^x problems

5.1.2 Results for Multiple Strip Packing in the "different" case

Our results on MSP_d^x are the following :

- in Chapter 10 we provide a $\frac{5}{2}$-approximation for the special case of $MSP_d^{nc/c}$ problem where $\max_j q_j \leq \min_i m_i$. This result is an improvement of the previous $\frac{5}{2}$-approximation for MSP_r^{nc} of Chapter 8
- in Chapter 11 we improve the results of Chapter 10 by providing a $\frac{5}{2}$-approximation for the (general) $MSP_d^{nc/c}$ problem (we recall that there is up to now no polynomial time algorithm with a ratio better than 3 for this problem)

All the results about MSP_d^x problems are presented in Figure 5.2.

Problem	Ratio	Remarks	Source
MSP_d^{nc}	3	Fast algorithm that handles non-clairvoyant jobs	[Schwiegelshohn et al., 2008]
MSP_d^{nc}	3	Fast algorithm that respects additional preallocation constraints	[Dutot et al., b]
$MSP_d^{nc/c}$	$2+\epsilon$	**Requires $\max_j q_j \leq \min_i m_i$** Obtained by adapting the remark of [Ye et al., 2009] (need solving $Q\|\|C_{max}$ with a ratio $1+\frac{\epsilon}{2}$)	Chapter 10, [Bougeret et al., 2010a]
$MSP_d^{nc/c}$	$\frac{5}{2}$	**Requires $\max_j q_j \leq \min_i m_i$** Fast algorithm	Chapter 10, [Bougeret et al., 2010a]
$MSP_d^{nc/c}$	$\frac{5}{2}$	Obtained using an oracle guess that needs a non polynomial enumeration in the worst case, applies for $MSP_r^{nc/c}$ when clusters have different speeds	Chapter 11

FIG. 5.2: Main results on MSP_d^x problems

The $\frac{5}{2}$-approximation of Chapter 11 is based on an interactive algorithm. This algorithm guesses the index of the potential (unique) "huge" job scheduled on each cluster. Using the formalism we introduced, this information is not "parametrizable", as we do not control the length of the answer. This leads to a fixed ratio algorithm, whose complexity heavily depends on the enumeration required to simulate the oracle. Even this enumeration leads to a non polynomial algorithm in the worst case, this result enables to easily describe large sets of "easy" instances where the bound on the enumeration cost can be tightened.

5.2 Resources allocation in a portfolio

The discrete Resource Sharing Scheduling Problem ($dRSSP$) addressed in Chapter 12 consists in allocating discrete resources to a set of heuristics to solve given instances in minimum time. In this problem we are given a finite set of n instances (a benchmark) and a finite set of k heuristics (a "portfolio of algorithms"). The time for solving any instance with any heuristic using any number of resources is also known.

The goal is to distribute m resources (typically processors) to the heuristics to minimize the time required for solving the n instances when running the k heuristics in parallel (using the chosen partition, and considering that and instance is solved as soon as one heuristic founds a solution). This problem had only be considered in [Sayag et al., 2006] for fractional distributions of resources, and we extended it to the discrete case.

In [Bougeret et al., 2009b] we provided a simple interactive algorithm, and considered the impact of different questions to ask to the oracle, leading to several approximation schemes. This results are improved in Chapter 12 using the "guess approximation technique" (as suggested in [Bougeret et al., 2009b]), that allows to communicate efficiently (meaning that the oracle answer is short) with the oracle. Moreover, we provide an original lower bound showing that on of the questions asked to the oracle is (almost) the best possible among a large set of natural questions.

Chapitre 6

Application to $Q||C_{max}$

Marin bougeret[‡], Pierre-François Dutot, and Denis Trystram

LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

[‡]This work is supported by DGA-CNRS

Abstract

In this chapter we consider the classical problem denoted by $Q||C_{max}$ of scheduling n independent jobs on m related machines. This problem is $\mathcal{NP}hard$ in the strong sense, and thus there is no $FPTAS$ for it unless $\mathcal{P} = \mathcal{NP}$. We present a $PTAS$ for $Q||C_{max}$ (where m is fixed) based on a (purely combinatorial) interactive algorithm. Due to the simplicity of the guess used, this result somehow emphasizes the difficulty of this problem.

6.1 Introduction, state of art

An instance $I = (J, M)$ of $Q||C_{max}$ is described by a set of job J and a set of machines M. We denote p_j the size of job j, and s_i the speed of machine i. Executing job j on machine i requires $\frac{p_j}{s_i}$ units of time. The optimal value is denoted C^*_{max}, and the value (*i.e.* makespan) of a schedule σ is denoted $C_{max}(\sigma)$.

A lot of approximation algorithms exist for this problem. Concerning "fast" algorithms, it is proved in [Gonzalez et al., 1977] that LPT (that schedules the biggest remaining job each time a machine is idle) is a 2-approximation. Several improvement have been provided, as the $\frac{3}{2}$-approximation in [Hochbaum and Shmoys, 1988] (that is used as a base of our interactive algorithm), a $1,4$-approximation in [Friesen and Langston, 1983], and a $1,382$-approximation in [Chen, 1991]. Concerning approximation schemes, the main results are summarized in Figure 6.1.

Source	Time/space for $1+\epsilon$ ratio	Remarks
[Hochbaum and Shmoys, 1988]	time $\mathcal{O}(mn^{\frac{10}{\epsilon^2}+3})$	using rounding
[Horowitz and Sahni, 1976]	time $\mathcal{O}((\frac{1}{\epsilon}n^2)^{m-1})$ space $\mathcal{O}((\frac{nm}{\epsilon})^m)$	apply to $R\|\|C_{max}$
[Lenstra et al., 1990]	time $\mathcal{O}((n+1)^{\frac{m}{\epsilon}}poly(n,m))$ space $\mathcal{O}(log(\frac{1}{\epsilon})poly(n,m))$	apply to $R\|\|C_{max}$ using LP
[Fishkin et al., 2008]	time $\mathcal{O}(n) + (\frac{log(m)}{\epsilon})^{\mathcal{O}(m^2)}$ space $\mathcal{O}((\frac{log(m)}{\epsilon})^{\mathcal{O}(m^2)})$	apply to $R\|c_{ij}\|C_{max}$
[Jansen, 2009]	time $\mathcal{O}(2^{\mathcal{O}(1/\epsilon^2 log(1/\epsilon)^3)}poly(n,m))$	using rounding and MILP
Chapter 6	time $\mathcal{O}(n^{\frac{m}{\epsilon}-2}log(np_{max})(mlog(m)+n)$ space $\mathcal{O}(poly(\frac{1}{\epsilon},n,m))$	combinatorial (no rounding or LP)

FIG. 6.1: Outline of the main approximation schemes ((MI)LP denoting (mixed integer) linear programming)

Thus, there exists for $Q||C_{max}$ some $FPTAS$ in [Horowitz and Sahni, 1976, Fishkin et al., 2008] when m is fixed (but with a large space complexity), and a recent $EPTAS$ in [Jansen, 2009].

In this chapter we present a $PTAS$ for $Q||C_{max}$ where m is fixed, that for any $\epsilon = \frac{1}{k}$ (with $k \in \{1, \ldots, 0, n\}$) achieve a $1+\epsilon$ ratio in time $\mathcal{O}(dt(A)n^{m(\frac{1}{\epsilon})-2})$, where $d = \mathcal{O}(log(np_{max}))$ and $t(A) = \mathcal{O}(mlog(m)+n)$ (being the execution time of the interactive algorithm). The space requirement of this algorithm is polynomial in the size of the input and in $\frac{1}{\epsilon}$ (with $\frac{1}{\epsilon} \leq n$).

Notice that, even if this algorithm does not improve the best known asymptotic dependencies in $\frac{1}{\epsilon}$, it may be in practice (for small values of $\frac{1}{\epsilon}$ and m) more efficient than the result of [Jansen, 2009] (because of the constant hidden in the exponent) or [Lenstra et al., 1990] (that requires solving $(n+1)^{\frac{m}{\epsilon}}$ linear programs), and use less space than [Horowitz and Sahni, 1976, Fishkin et al., 2008] where space requirement is

exponential in m.

This result is obtained by a (purely combinatorial, without rounding tricks *etc.*) interactive algorithm based on the $\frac{3}{2}$-approximation of [Hochbaum and Shmoys, 1988]. The property asked by the algorithm is "natural" (the algorithm guesses the $k = \frac{1}{\epsilon}$ biggest jobs executed on every machine in an optimal solution) and thus point out a difficulty of the problem.

6.2 Interactive algorithm for $Q||C_{max}$

6.2.1 Definition of the guess

Let $k \in \mathbb{N}$, $k \leq n$. The oracle provides a string $info$ such that : for any machine $i \in \{1, .., m\}$, $info$ contains the index of the k biggest jobs $\{o(i,1), ..., o(i,k)\}$ scheduled on machine i in an optimal solution. We consider that oracle use index $n+1$ if less than k jobs are scheduled on a machine. Without loss of generality, let us assume that $\forall i \in \{1, .., m\}, p_{o(i,1)} \geq p_{o(i,2)} ... \geq p_{o(i,k)}$ and let us denote by $min_i = p_{o(i,k)}$ the smallest job given by the oracle and each machine. We denote by I' the set of jobs that are not used in $info$. Finally, let $c_i^0 = \frac{\Sigma_{j=1}^{k} p_{o(i,j)}}{s_i}$ for any $i \in \{1, .., m\}$.

6.2.2 Definition of the interactive algorithm

As the algorithm A_{info} follows the well known dual approximation technique [Hochbaum and Shmoys, 1988], its specifications are the following :

- input of A_{info}
  - I the set of jobs and machines
  - $info$, given by the oracle
  - w, the current estimated value (of the optimal value) used in the dual approximation technique
- possible outputs of A_{info}
  - failure (A_{info} rejects w), implying $w < C^*_{max}$, or
  - a schedule σ such that $C_{max}(\sigma) \leq (1 + \frac{1}{k+2})w$

Thus, according to the dual approximation technique we obtain with A_{info} a $(1+\frac{1}{k+2})$-approximation in time $\mathcal{O}(log(np_{max})t(A_{info}))$ using a dichotomic search on w to find the smallest w that is not rejected.

Given a fixed value of w, let us define :

- the range of machine i : $r_i = s_i(w - c_i^0)$ (we assume $r_i \geq 0$, otherwise $w < C^*_{max}$)
- the corrected range of machine i : $r'_i = min(r_i, min_i)$
- the set $Aut_i = \{j, p_j \leq r'_i\}$ of authorized jobs for machine i

Thus, r_i is the size of the biggest job that machine i can execute without overlapping w. However, we know that there exists an optimal schedule such that any machine i only executes jobs j (of I') such that $p_j \leq min_i$. Thus, the corrected range of a machine is the size of the biggest job that is "authorized" to be scheduled on this machine. A_{info} will only schedule jobs from Aut_i on machine i.

A_{info} starts by scheduling on machine i jobs $\{o(i,1), ..., o(i,k)\}$. Recall that $I' = I \setminus \{o(i,j), i \in \{1, .., m\}, j \in \{1, .., k\}\}$ is the set of remaining jobs. We schedule I' using an adaptation of the $\frac{3}{2}$-approximation of [Hochbaum and Shmoys, 1988]. In the

original algorithm, machines are considered in the non decreasing order of their speed. Here, we order the set M of machines in non decreasing order of $r''_i = min(min_i, \frac{r_i}{2})$. Jobs of I' are scheduled by $Pack(I', M, m)$, defined in page 81.

Algorithm 1 pack(J,M,m)

if $\Sigma_{j\in J} p_j \leq \Sigma_{i\in S} r_i$ **then**
  $J_{sml} = \{j \in J | p_j \leq r''_m\} = Aut_m \cap \{j | p_j \leq \frac{r_m}{2}\}$
  $J_{new} = J - J_{sml}$
  $J_{fit} = J_{new} \cap Aut_m = Aut_m \cap \{j | p_j > \frac{r_m}{2}\}$
  if $J_{fit} \neq \emptyset$ **then**
    choose $\bar{j} | p_{\bar{j}} = \max_{j\in J_{fit}} p_j$
    schedule $\bar{j}$ on machine m
    $J_{new} = J_{new} - \bar{j}$
  end if
  if $J_{new} \neq \emptyset$ **then**
    $M_{new} = M - m$
    pack(J_{new},M_{new},$m-1$)
  end if
  while $J_{sml} \neq \emptyset$ **do**
    choose a machine i that finishes before w in the current schedule
    schedule j on i
    $J_{sml} = J_{sml} - \{j\}$
  end while
else
  return "reject w"
end if

6.2.3 Proof

Let us prove the following result.

Theroem 6.2. *If A_{info} reject w then $w < C^*_{max}$, otherwise A_{info} builds a schedule σ such that $C_{max}(\sigma) \leq (1 + \frac{1}{k+2})w$.*

The proof can be structured as in [Hochbaum and Shmoys, 1988].

Lemma 6.3. *If an instance (J, M) is feasible in w, then the instance (J_{new}, M_{new}) created by procedure pack(J, M, m) is also feasible in w.*

Proof If (J, M) is feasible in w, then (J_{new}, M) is also feasible. Moreover, we know that the only jobs of J_{new} scheduled in the considered optimal on machine m are in Aut_m, and thus in J_{fit}. Let $J_{new} = J_{fit} \cup \bar{J_{fit}}$. Machine m cannot schedule jobs of $\bar{J_{fit}}$. Thus, if $J_{fit} = \emptyset$, then we get directly that J_{new} is feasible in w using machines of M_{new}. Otherwise, we know that in the considered optimal (for I), there is at most one job of J_{fit} on machine m. Thus, as the algorithm schedules the biggest job of J_{fit}, we can prove that (J_{new}, M_{new}) is feasible using a classical swap argument. □

Lemma 6.4. *If $pack(J, M, m)$ rejects w, then $w < C^*_{max}$.*

Proof Let us suppose by contradiction that (J, M) is feasible in w. Then, according to Lemma 6.3, the instance created during each recursive call of *pack* is feasible in w. However, when the procedure fails we know that $\Sigma_{j\in J} p_j > \Sigma_{i\in S} r_i$, which is a contradiction. □

Lemma 6.5. *If $pack(J, M, m)$ does not reject w, then jobs of J are scheduled, and $C_{max} \leq (1 + \frac{1}{k+2})w$.*

Proof It is clear that all the jobs are scheduled, as the only possible failure could happen line 15 if all the machines had a load greater than w, which is impossible (according to line 1). Let us now prove that if we overlap w when scheduling (at line 16) a job $j \in J_{sml}$ on a machine i, then the time needed for processing this job is lower than $\frac{w}{k+2}$. We know that $p_j \leq r''_m \leq r''_i = min(min_i, \frac{r_i}{2})$. Two cases are possible according to the configuration of machine i just after adding jobs of $info$. If $c^0_i \geq \frac{kw}{k+2}$ (i is "saturated"), we use $p_j \leq \frac{r_i}{2} = \frac{s_i(w-c^0_i)}{2} \leq s_i \frac{w}{k+2}$, and thus $\frac{p_j}{s_i} \leq \frac{w}{k+2}$. Otherwise, $c^0_i < \frac{kw}{k+2}$, (i is "free"), and thus we get $k\frac{min_i}{s_i} \leq \frac{kw}{k+2}$ and $\frac{p_j}{s_i} \leq \frac{min_i}{s_i} \leq \frac{w}{k+2}$. □

Corollary 6.6. *For any $k \in \mathbb{N}, k \leq n$ there is a $1 + \frac{1}{k+2}$-approximation running in $\mathcal{O}(log(np_{max})n^{km}(mlog(m) + n))$.*

Proof We just run A_{info} while enumerating all the possible oracle answers. For any input I we have $|info| \leq kmlog(n + 1) = b$, and thus we enumerate the $\mathcal{O}(n^{km})$ strings of length at most b. Given that $C^*_{max} \in \{1, \ldots, np_{max}\}$, the dichotomic search on w requires $\mathcal{O}(log(np_{max}))$ repetitions. Finally, with w and $info$ fixed, sorting the machines according to r''_i costs $\mathcal{O}(mlog(m))$, and A_{info} can be implemented in $\mathcal{O}(n)$ following ideas of [Hochbaum and Shmoys, 1988]. □

6.2.4 Conclusion

The first comment is about the tuning of this result. The ϵ dependency could be improved using typically "guess approximation" techniques, where the information asked to the oracle is the processing time of the jobs (up to a $\delta(\epsilon)$ factor) rather than the exact indexes (as in the Example 3.26). Typically, we could first for each machine i ignore the set of jobs X_i that have processing time lower than $s_i\epsilon w$. If there are k' jobs of X_i among the k biggest jobs scheduled on machine i, then the oracle only give the $k - k'$ biggest jobs, and we complete by greedily adding jobs of X_i until reaching w. Thus, we only ask processing time of the (non negligible) $k - k'$ biggest jobs, and up to a multiplicative factor $(1 + \epsilon)$. The number of possible values for designating a job is reduced from n to $x = \frac{1}{\epsilon log(1+\epsilon)}$, as $\epsilon w(1 + \epsilon)^x = w$ For each value $l \in \{1, \ldots, x\}$ given by the oracle the algorithm schedules the biggest remaining job of size at most

$\epsilon w(1+\epsilon)^{l+1}$, and thus the ratio is only degraded by a factor ϵ. Thus, informally the complexity could be reduced to $\mathcal{O}(log(np_{max})(\frac{1}{\epsilon log(1+\epsilon)})^{km}(mlog(m)+n))$.

The second comment is more methodological. This results shows somehow that as it was expected the difficulty of this problem comes from the biggest jobs, and thus could motivate the study of that particular point. For example, even studying (for example dominance rules) this information for $k = 1$ would lead to a $\frac{4}{3}$ ratio with a simple algorithm, thus improving the (very technical) bound of $1,382$ of [Chen, 1991]. Finally, as suggested at the end of Chapter 3 it would be important for practical applications to enumerate "efficiently" the possible oracle answers, using for example branch and bound techniques that would delete partial allocation of $k_i \leq k$ (potential) biggest jobs on machine m_i that couldn't lead to solutions that fit in w even using preemption.

Bibliographie

[Chen, 1991] Chen, B. (1991). Tighter bound for MULTIFIT scheduling on uniform processors. *Discrete Applied Mathematics*, 31(3) :227–260.

[Fishkin et al., 2008] Fishkin, A., Jansen, K., and Mastrolilli, M. (2008). Grouping techniques for scheduling problems : simpler and faster. *Algorithmica*, 51(2) :183–199.

[Friesen and Langston, 1983] Friesen, D. K. and Langston, M. A. (1983). Bounds for multifit scheduling on uniform processors. *SIAM Journal on Computing*, 12(1) :60–70.

[Gonzalez et al., 1977] Gonzalez, T., Ibarra, O. H., and Sahni, S. (1977). Bounds for lpt schedules on uniform processors. *SIAM Journal on Computing*, 6(1) :155–166.

[Hochbaum and Shmoys, 1988] Hochbaum, D. and Shmoys, D. (1988). A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM Journal on Computing*, 17(3) :539–551.

[Horowitz and Sahni, 1976] Horowitz, E. and Sahni, S. (1976). Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM (JACM)*, 23(2) :317–327.

[Jansen, 2009] Jansen, K. (2009). An EPTAS for scheduling jobs on uniform processors : using an MILP relaxation with a constant number of integral variables. *Automata, Languages and Programming*, pages 562–573.

[Lenstra et al., 1990] Lenstra, J., Shmoys, D., and Tardos, E. (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1) :259–271.

Chapitre 7

Approximation Algorithms for Multiple Strip Packing [Bougeret et al., 2009d]

Marin bougeret*‡, Pierre-François Dutot*, Klaus Jansen†, Christina Otte† and Denis Trystram*

*LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

†Department of Computer Science, Christian-Albrechts-University, Kiel, Germany
kj, cot@informatik.uni-kiel.de

‡This work is supported by DGA-CNRS

†Work supported by EU project "AEOLUS : Algorithmic Principles for Building Efficient Overlay Computers", EU contract number 015964, and DFG project JA612/12-1, "Design and analysis of approximation algorithms for two- and threedimensional packing problems"

Abstract

In this paper we study the Multiple Strip Packing (MSP) problem, a generalization of the well-known Strip Packing problem. For a given set of rectangles, $r_1, \ldots, r_n$, with heights and widths ≤ 1, the goal is to find a non-overlapping orthogonal packing without rotations into $k \in \mathbb{N}$ strips $[0,1] \times [0,\infty)$, minimizing the maximum of the heights. We present an approximation algorithm with absolute ratio 2, which is the best possible, unless $\mathcal{P} = \mathcal{NP}$, and an improvement of the previous best result with ratio $2 + \epsilon$. Furthermore we present simple shelf-based algorithms with short running-time and an AFPTAS for MSP. Since MSP is strongly $\mathcal{NP}$-hard, an FPTAS is ruled out and an AFPTAS is also the best possible result in the sense of approximation theory.

7.1 Introduction

In this paper we study the Multiple Strip Packing (MSP) problem, a generalization of the well-known Strip Packing (SP) problem. For a given set of rectangles, $r_1, \ldots, r_n$, with heights and widths ≤ 1, the goal is to find a non-overlapping orthogonal packing without rotations into $k \in \mathbb{N}$ strips $[0,1] \times [0,\infty)$, minimizing the maximum of the heights. As much as Strip Packing, its generalization Multiple Strip Packing is not only of theoretical interest, but also has many applications to real-world problems as in computer grids, server consolidation and in cutting problems. In computer grids for example, MSP is related to the problem of finding a schedule for parallel tasks into different clusters of processors with minimum makespan [12]. Consider an instance $L = \{r_1, \ldots, , r_n\}$ of MSP. The value k always denotes the number of strips $S_1, \ldots, S_k$. For $i \in \{1, \ldots, k\}$ the value h_i denotes the height of a feasible packing in strip S_i. For an algorithm A for MSP let $A(L)$ be the output of the algorithm, in this case the maximum height of the packing generated, i.e. $\max_{i \in \{1,\ldots,k\}} h_i$. The optimal value is denoted with $OPT(L)$, in this case the minimal height that can be achieved. The quality of an approximation algorithm is measured by its performance ratio. For a minimization problem as MSP we say that A has *absolute ratio* α, if $\sup_L {}^{A(L)}/_{OPT(L)} \leq \alpha$, and *asymptotic ratio* α, if $\alpha \geq \limsup_{OPT(L)\to\infty} {}^{A(L)}/_{OPT(L)}$, respectively. A minimization problem admits an *(asymptotic) polynomial-time approximation scheme* ((A)PTAS), if there exists a family of polynomial-time approximation algorithms $\{A_\epsilon | \epsilon > 0\}$ of (asymptotic) $(1+\epsilon)$-approximations. We call an approximation scheme *fully polynomial* ((A)FPTAS), if the running-time of every algorithm A_ϵ is bounded by a polynomial in n and $\frac{1}{\epsilon}$. Zhuk showed in [16] that there is no approximation algorithm for MSP with absolute ratio less than 2. Since MSP can be reduced to 3-Partition, it is also strongly $\mathcal{NP}$-hard. Therefore a PTAS and an FPTAS are ruled out and an AFPTAS is asymptotically the best possible.

A related problem is 3D Strip Packing (3SP), which also is a generalization of Strip Packing. Here the goal is to find a packing of a given list of cuboids with side lengths bounded by one into a 3-dimensional strip $[0,1] \times [0,1] \times [0,\infty)$, minmizing the height of the packing. Multiple Strip Packing with k strips can be reduced to 3SP by introducing a cuboid with depth $1/k$ for each rectangle packing the strips next to each other.

Parallel Job Scheduling in Grids with identical machines is also a related problem. In the offline case we have m machines M_i with ℓ processors and jobs $j \in J$ with processing time p_j, and a size $size_j$. The jobs must be executed on parallel processors within one machine M_i, but not necessary on consecutive processors. The machines can be seen as strips with width l and the jobs as vertically scissile rectangles with width $size_j$ and height p_j. In Multiple Strip Packing we have just the additional constraint that a job must be scheduled on consecutive processors. Unfortunately this is the reason why approximation algorithms for Parallel Job Scheduling cannot be applied to MSP maintaining their ratio.

Known Results. Multiple Strip Packing was first considered by Zhuk [16], who showed that there is no approximation algorithm with abolute ratio better than

2, and later by Ye et. al. [15]. Both concentrated on the online case. Additonally an approximation algorithm for the offline case with ratio $2 + \epsilon$ was achieved in [15]. For Strip Packing Coffman et al. gave in [5] an overview about performance bounds for shelf-orientated algorithms as $NFDH$ (Next Fit Decreasing Height) and $FFDH$ (First Fit Decreasing Height). Those adopt an absolute ratio of 3, and 2.7, respectively. Schiermeyer [11] and Steinberg [13] presented independently an algorithm for SP with absolute ratio 2. A further important result is an AFPTAS for SP with additive constant $\mathcal{O}(1/\epsilon^2)$ of Kenyon and Rémila [9]. This constant was improved by Jansen and Solis-Oba, who presented in [8] an APTAS with additive constant 1. For 3SP Jansen and Solis-Oba obtained an algorithm with ratio $2 + \epsilon$ in [7] as an improvement of the formerly known result by Miyazawa and Wakabayashi [10], who presented an algorithm with asymptotic ratio at most 2.64. Bansal et al. presented in [2] an algorithm for $3SP$ with a ratio of $T_\infty \approx 1.69$, which is the best known result. Schwiegelshohn et al. [12] achieved ratio 3 for a version of Parallel Job Scheduling in Grids without release times, and ratio 5 with release times. Tchernykh et al. presented in [14] an algorithm with absolute ratio 10 for the case of machines with different numbers of processors and without release times. However, this algorithm cannot be applied directly to MSP because of the non-contiguity.

Our Results. In this paper we present an approximation algorithm with absolute ratio 2, which is an improvement of the former result of $2 + \epsilon$ by Ye et al. [15] and best possible, unless $\mathcal{P} = \mathcal{NP}$. We also introduce an AFPTAS for Mutiple Strip Packing, which is a generalization of the algorithm of Kenyon and Rémila [9]. Our algorithm achieves an additive constant of $\mathcal{O}(1)$, if the number of strips is sufficient large, otherwise an additive constant of $\mathcal{O}(1/\epsilon^2)$. Furthermore we show how to use the simple shelf-based heuristics $NFDH$ and $FFDH$ to obtain approximation algorithms for MSP with the same asymptotic ratio as for SP.

Organisation of the Paper. In the next section we introduce two shelf-based algorithms, using Next Fit and First Fit policies. In Section 7.3 we present a 2-approximation for MSP. Here we distinguish between different sizes for k. For $k = 1$ we use the 2-approximation of Steinberg [13] or Schiermeyer [11]. If $k = 2$ or bounded by a specified constant c we make use of a result by Bansal et al. [1] for Rectangle Packing with Area Maximization (RPA). For $k \geq c$ we use an approximation algorithm for 2D bin packing with asymptotic ratio 1.69 of Caprara [4]. In the last section we present an AFPTAS for MSP. Here we generalize the algorithm by Kenyon and Rémila [9]. Interestingly, the additive constant in our AFPTAS can be reduced from $\mathcal{O}(1/\epsilon^2)$ to $\mathcal{O}(1)$, if the number k of strips is large enough.

7.2 Shelf-based algorithms

In this section we modify the shelf-based heuristics NFDH and FFDH. [5]. A shelf is a row of items placed next to each other left-justified. The baseline of a shelf is either

the bottom of the bin or the extended upper edge of the tallest item packed in the shelf below. NFDH generates for a given list of rectangles $L = \{r_1, \ldots, r_n\}$ a packing into a strip with height at most $2OPT_{SP}(L) + h_{\max}$, FFDH produces a packing of height at most $1.7OPT_{SP}(L) + h_{\max}$, where $OPT_{SP}(L)$ is the optimum value of Strip Packing for the instance L and $h_{\max}$ is the height of the tallest item in L. Via this modification we obtain approximation algorithms for Multiple Strip Packing with the same asymptotic ratios. Furthermore, we present another algorithm, that computes for rectangles with widths bounded by $\epsilon < 1$ a packing of height at most $1/(1-\epsilon)OPT(L) + 2h_{\max}$.

Theroem 7.1. *Let A be one of the shelf-based Strip Packing algorithms NFDH or FFDH with asymptotic ratio $\alpha > 1$, that creates for an instance L a packing of height less than $\alpha Opt_{SP}(L) + h_{\max}$. For any $k \in \mathbb{N}$ there exists an algorithm A_k that packs a list of rectangles L into k strips with $A_k(L) \leq \alpha Opt(L) + h_{\max}$.*

Proof For any instance L of MSP we define the algorithm A_k as follows

1 Pack the sorted rectangles with A into one strip S. (In particular the rectangles are first sorted by non-increasing height.) Let $A(L)$ denote the height of S.

2 Cut out the first shelf and pack it into the first strip S_1.

3 Divide the residual strip S into k parts :

3.1 For each $\ell \in \{0, 1, \ldots, k\}$ draw a horizontal line across S at height $\ell(A(L) - h_{\max})/k$.

3.2 For $\ell \in \{0, 1, \ldots, k-1\}$ pack all items intersecting the ℓth line and all items between the ℓth and $(\ell+1)$th line into strip $S_{\ell+1}$.

We show now that for any instance L of MSP the output of A_k is less than $\alpha Opt(L) + h_{\max}$. Let $t \in \mathbb{N}$ be the number of shelves produced by A in Step 1 and H_j, $j \in \{1, \ldots, t\}$, the height of the jth shelf. Since there are no items intersecting the 0th line (see Fig 7.2), the height h_1 of the first strip S_1 is bounded by $h_{\max} + \frac{1}{k}\left(\sum_{j=1}^{t} H_j - h_{\max}\right)$ after the last step of the algorithm. For a strip S_i, $i \in \{2, \ldots, k\}$, containing the items between the $(i-1)$th and the ith line and the ones intersecting the $(i-1)$th line, we have $h_i \leq \frac{\sum_{j=1}^{t} H_j - h_{\max}}{k} + h_{\max} = \frac{A(L) - h_{\max}}{k} + h_{\max}$. We conclude

$$A_k(L) = \max_{i \in \{1,\ldots,k\}} h_i \leq \frac{A(L) - h_{\max}}{k} + h_{\max}$$
$$\leq \frac{\alpha Opt_{SP}(L) + h_{\max} - h_{\max}}{k} + h_{\max} = \frac{\alpha Opt_{SP}(L)}{k} + h_{\max}.$$

Since $1/k Opt_{SP}(L)$ is a lower bound for $Opt(L)$ the proof is complete. □
The running-time of the above algorithm is $\mathcal{O}(n \log n)$.

Corollary 7.3. *Let L be an instance of MSP. In a packing generated by the above algorithm A_k we have $\max_{i \in \{1,\ldots,k\}} |h_i - A_k(L)| \leq 2h_{\max}$, where h_i denotes the height of strip S_i.*

Another way to pack a set of rectangles with a modified version of the $NFDH$ heuristic into k strips is the following :

The packing generated by the above algorithm is very smooth, in the sense that the heights of the strips only differ by $h_{\max}$.

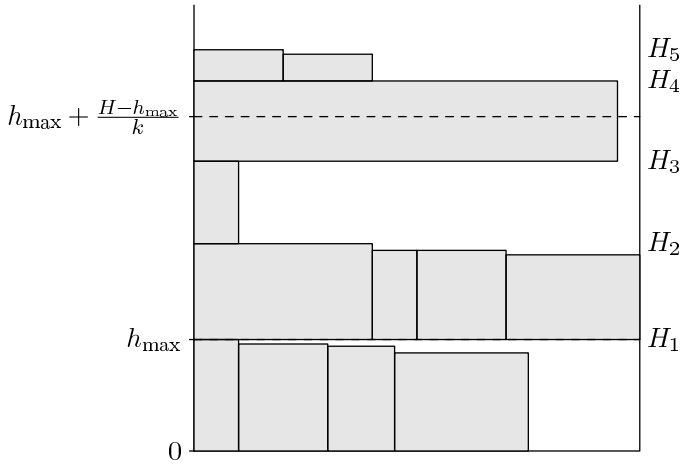

FIG. 7.2: Dividing strip S.

1 Sort the rectangles by non-increasing height.

2 For each $i \in \{1, \ldots, k\}$ pack one shelf according to the $NFDH$ heuristic into strip S_i, that means starting in the lower left corner pack the rectangles next to each other on the baseline of strip S_i, until the next rectangle does not fit. Draw a new baseline at the top edge of the tallest rectangle (that clearly is the first one).

3 Take the strip S^- with the current lowest height h^- and pack one shelf according to the $NFDH$ heuristic on top of the shelves.

4 Repeat Step 3 until all rectangles are packed.

Lemma 7.4. *For a set of rectangles $L = \{r_1, \ldots, r_n\}$ Algorithm 7.2 with output $A(L)$ generates a packing into k strips, so that* $\max_{i \in \{1,\ldots,k\}} |A(L) - h_i| \leq h_{\max}$.

This leads to a further result about rectangles with bounded width. Coffman et al. showed in [5] that $FFDH$ applied to an instance L of rectangles with widths bounded by $1/m$ for some integer m generates a packing into a strip of height at most $(1 + \frac{1}{m})OPT_{SP}(L) + h_{\max}$. Our result for packing into k strips is the following :

Theroem 7.5. *For a set of rectangles $L = \{r_1, \ldots, r_n\}$ with widths bounded by $\epsilon > 0$ we obtain by the Algorithm 7.2 with output $A(L)$ a packing into k strips with height less than $\frac{1}{1-\epsilon}OPT(L) + 2h_{\max}$.*

For $\epsilon = \frac{1}{m}$ this is equal to $A(L) \leq \left(1 + \frac{1}{m-1}\right) Opt(L) + 2h_{max}$.

7.3 A two-approximation for MSP

In this section we construct a polynomial-time approximation algorithm for MSP with absolute ratio 2. Since there is no approximation algorithm for MSP with ratio smaller than 2 (unless P=NP), this is the best possible result. To handle different sizes of k we use, besides the well-known algorithms of Steinberg [13] or Schiermeyer [11], a

result of Bansal et al. [1] for Rectangle Packing with Area Maximization (RPA) and a of Caprara [4].

7.3.1 One or two strips

The case $k = 1$ is trivial, because we can use the algorithm of Steinberg [13] or Schiermeyer [11] with absolute performance bound 2.

Theroem 7.6 (Steinberg [13]). *Let $L = \{r_1, \ldots, r_n\}$ be a set of rectangles with heights h_i and widths w_i and Q be a rectangle with width u and height v. Let $h := \max_{i\in\{1,\ldots,n\}} h_i$ and $w := \max_{i\in\{1,\ldots,k\}} w_i$. If the following inequalities hold,*

$$w \leq u, \quad h \leq v, \quad 2SIZE(L) \leq uv - (2w - u)_+(2h - v)_+ \tag{7.1}$$

then it is possible to pack L into the rectangle Q.(As usual, $x_+ = \max(x, 0)$.)

Therefore let us first consider the case for $k = 2$. Here we use the PTAS found by Bansal et al. [1] for RPA. In RPA we are given a set of rectangles $L = \{r_1, \ldots, r_n\}$ with widths w_i and heights h_i and a bin of unit size. The goal is to find a feasible packing of a subset $L' \subset L$ of the rectangles and to maximize the area of the rectangles in L'.

---

1 Guess the height of an optimal solution for MSP and denote it with v.

2 Scale the heights of the rectangles in L by $1/v$ so that the corresponding packing fits into one bin of height and width one.

3 The set of resulting rectangles L_v is now considered as an instance of RPA with $Opt_{RPA}(L) = SIZE(L_v)$, where $SIZE(L_v)$ is the total area of all rectangles in L_v. Apply the algorithm in [1] with accuracy $\varepsilon = 1/2$ and find a packing of a subset $L'_v \subset L_v$ with total area at least $(1-\varepsilon)SIZE(L_v)$. By rescaling the rectangles of L'_v get a packing for the first strip with height at most v.

4 Since $SIZE(L_v) \leq 2$ the remaining items in $L_v \backslash L'_v$ have total area $SIZE(L_v \backslash L'_v) \leq \varepsilon SIZE(L_v) \leq 1$. Therefore we can pack them with Steinberg's algorithm into a strip of height at most 2. Rescaling gives us a second strip of height at most $2v$.

The running-time of the algorithm is polynomial in n :
In the first step we can assume that the heights of the rectangles are rational, so by multiplying with a common denominator they become integer values. Then the optimum height v of MSP is also integer and equals a sum of heights of the rectangles in L, so we have $h_{\max} \leq v \leq nh_{\max}$. Thus Binary Search takes at most $\log(nh_{\max})$ iterations to find the value v. Step 3 is also polynomial, since we apply the algorithm in [1] for a fixed accuracy $\epsilon = 1/2$.

7.3.2 A bounded number of strips

In the case of a constant number of strips we can use an extended version of the PTAS for RPA in [1] called $kRPA$. Another helpful tool is the next lemma. The proof

can be obtained applying Steinberg's algorithm for $h, w, u = 1$ and $v = k/2$ in equation 7.1.

Lemma 7.7. *If L is an instance of 2DBP with total area $SIZE(L) \leq k/4$ and $k \geq 3$, then there exists a packing of L into k bins.*

---

1 Guess an optimal height for MSP and denote it with v.

2 Scale the heights of the rectangles in L by $1/v$ so that the corresponding packing fits into k bins of height and width one.

3 The set of resulting rectangles L_v is now considered as an instance of RPA with $Opt_{RPA}(L) = SIZE(L_v)$. Apply $kRPA$ to k bins of unit size and find for an accuracy $\varepsilon \leq 1/4$ a packing for a subset $L'_v \subset L_v$ with total area $(1-\varepsilon)SIZE(L_v)$. By rescaling the rectangles of L'_v we get k bins of height v.

4 For the total area of the remaining rectangles in $L_v \backslash L'_v$ we have $SIZE(L_v \backslash L'_v) = \varepsilon SIZE(L_v) \leq k/4$. Pack those rectangles according to Lemma 7.7 into k bins and rescale the rectangles. This results again in k bins of height at most v.

5 Stack every two bins on top of each other and get a solution with k bins of height at most $2v$.

7.3.3 A large number of strips

Caprara presented in [4] a shelf algorithm for $2DBP$ that produces a solution whose asymptotic ratio can be made arbitrarily close to $T_\infty = 1.69....$ Clearly if the number of strips is large enough ($\approx 10^4$) applying this algorithm we get a two-approximation for MSP stacking every two bins on each other. Alternatively, we can use the recently published two-approximation for 2DBP by Jansen et al. [6] to achieve this result. Along with the previous sections we have the following :

Theroem 7.8. *For any $k \in \mathbb{N}$ there is a polynomial-time algorithm for MSP with absolute ratio two.*

7.4 An AFPTAS for MSP

In this section we present an AFPTAS for MSP. The algorithm is a generalization of an AFPTAS found by Kenyon and Rémila [9] for Strip Packing. For an instance L of Strip Packing and an accuracy $\epsilon > 0$ their algorithm generates a packing with height $(1+\epsilon)OPT_{SP}(L) + \mathcal{O}(1/\epsilon^2)h_{\max}$. Our algorithm achieves the same ratio for Multiple Strip Packing. For instances with k sufficient large, namely $k \in \Omega(1/\epsilon^3)$, our algorithm adopts an improved additive constant of $\mathcal{O}(1)$. More precisely for an accuracy ϵ and $k \geq \lceil 128/\epsilon^3 \rceil$ we get an approximation ratio of $(1+\epsilon)Opt(L) + 6h_{\max}$.

7.4.1 The regular case

As in Section 7.2 we divide a packing into one strip into k parts of nearly the same height and distribute them to k strips.

Theroem 7.9 (Kenyon & Rémila [9]). *For a list $L = \{r_1, \ldots, r_n\}$ of rectangles and an accuracy $\epsilon > 0$ the algorithm A_ϵ^{KR} in [9] generates a packing into one strip with height at most $(1+\epsilon)OPT_{SP}(L) + (4(\frac{2+\epsilon}{\epsilon})^2 + 1)h_{\max}$.*

Our result is the following :

Theroem 7.10. *For a list $L = \{r_1, \ldots, r_n\}$ of rectangles with widths and heights ≤ 1 and an accuracy $\epsilon > 0$ there exits an algorithm A_ϵ that generates a packing into k strips, so that $A_\epsilon(L) \leq (1+\epsilon)OPT(L) + (2(\frac{2+\epsilon}{\epsilon})^2 + 2)h_{\max}$.*

7.4.2 Instances with a large number of strips

In this section we consider the case $k \geq \lceil 128/\epsilon^3 \rceil$. In this case it is possible to improve the additive constant to $\mathcal{O}(1)h_{\max}$ by balancing the configurations.

Rounding. We choose $\epsilon' = \epsilon/4$ (w.l.o.g. $1/\epsilon'$ integral) and divide the list of rectangles L into a list of narrow rectangles $L_{narrow} := \{r_i \in L | w(r_i) \leq \epsilon'\}$ and a list of wide rectangles $L_{wide} := \{r_i \in L | w(r_i) > \epsilon'\}$. Then we round L_{wide} to an instance L_{sup} with only $M := (1/\epsilon')^2$ different widths. For the rounding step we put the wide rectangles sorted by non-increasing widths left-aligned on a stack. Let $STACK(L)$ denote the total area of the plane covered by this stack and let H denote its height. Moreover, for arbitrary lists L'', L' we define a relation $\leq_g$, so that $L'' \leq_g L'$, if and only if $STACK(L'') \subseteq STACK(L')$. We draw $M-1$ horizontal lines through $STACK(L)$ with distance H/M starting at the bottom. Therefore we get M so-called *threshold* rectangles. A rectangle is a threshold rectangle if it either with its interior or with its lower edge intersects a line at height iH/M, $i \in \{1, \ldots, M-1\}$. For $i \in \{1, \ldots, M-1\}$ we round up the width of each rectangle between the lines iH/M and $(i+1)H/M$ to the width of the ith threshold rectangle. The widths of the rectangles below the first line are rounded up to the width of the undermost rectangle in the stack. So we get at most M groups of different widths (see Fig 7.11). Furthermore, we get a list L_{sup} of rectangles with widths larger than ϵ' and only M different widths, in particular we have $L_{wide} \leq_g L_{sup}$.

Fractional Packing. Our first objective is to create a fractional packing for the wide rectangles into k strips. To do this we introduce *configurations*. A configuration is a non-empty multiset of widths, which sum up to less than one. Denote with q the number of different configurations C_j with height x_j. Let α_{ij} be the number of occurence of width w_i in configuration C_j and let β_i be the total height of all rectangles of width

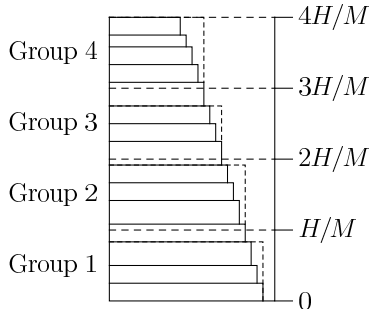

FIG. 7.11: Rounding the rectangles in L_{sup}.

w_i. Based on the solution of the following Linear Program

$$\begin{aligned} &\min \frac{\sum_{j=1}^{q} x_j}{k} \\ \text{s.t. } &\sum_{j=1}^{q} \alpha_{ij} x_j \geq \beta_i \text{ for all } i \in \{1,\ldots,M\} \\ &x_j \geq 0 \text{ for all } j \in \{1,\ldots,q\}, \end{aligned} \qquad (\text{LP}(L_{sup}))$$

by distributing the configurations to k strips we get the requested fractional packing for the rectangles in L_{sup}. Note that $\text{rank}(\alpha_{ij})_{ij} \leq M$ and hence a basic solution x of $LP(L_{sup})$ has at most M nonzero entries. In the next section we show how to transform a fractional packing into a feasible packing for L_{sup}. Later the rectangles in L_{narrow} are packed into the idle space in a Greedy manner. For a list L of rectangles let $LIN(L)$ denote the height of an optimum fractional packing for L. Let $h_0 := LIN(L_{sup})$ and note that $h_0 \leq OPT(L)$.

Lemma 7.12. *Let $x = (x_1,\ldots,x_q)$ be a solution of $LP(L_{sup})$ with at most $m \leq M$ nonzero entries $x_1,\ldots,x_m$. For $k \geq \lceil 128/\epsilon^3 \rceil$ we get a fractional packing into k strips with height at most $(1+\epsilon')h_0$ and at most $m' \leq 2M$ different configurations.*

Proof First we fractionally pack the rectangles into the configurations. Imagine each configuration C_j as a bin with height x_j and width c_j and divide it into α_{ij} columns of widths w_i and height x_j. Pack the rectangles in L_{sup} of width w_i in a Greedy manner fractionally into the columns of width w_i until exactly height x_j, starting with $j = 1$. In this way each column contains a sequence of rectangles, which completely fits inside the column, and possibly the top part of a rectangle, that started in a previous column, and the bottom part of a rectangle, that is too tall to fit into this column. Since $\sum_{j=1}^{m} \alpha_{ij} x_j \geq \beta_i$, there will be maybe more than enough space for the rectangles of width w_i in the configurations. In this case we distribute the rectangles among the columns and delete the additional space. So we split a configuration C_j into two parts, one of the old type where the columns of width w_i are completely filled and one without columns of width w_i. This case may happen only M times. So we have in total $m' = m + M \leq 2M$ configurations $C_1,\ldots C_{m'}$ with nonzero heights $x_1,\ldots,x_{m'}$. Notice that there exist configurations with height larger or equal h_0, since if not we conclude $\sum_{j=1}^{m'} x_j < m'h_0 \leq 2Mh_0 \overset{\epsilon'=\epsilon/4}{=} \frac{32h_0}{\epsilon^2} < kh_0$, which is a contradiction. Consider

a configuration C_j, $j \in \{1, \ldots, m'\}$. If $x_j \geq h_0$ we allocate $\lfloor {}^{x_j}/_{h_0} \rfloor$ empty strips with height h_0 for C_j. If then ${}^{x_j}/_{h_0} - \lfloor {}^{x_j}/_{h_0} \rfloor \leq \epsilon' h_0$, we assign to C_j additional space with height $({}^{x_j}/_{h_0} - \lfloor {}^{x_j}/_{h_0} \rfloor)$ in a strip, that has already height h_0. If ${}^{x_j}/_{h_0} - \lfloor {}^{x_j}/_{h_0} \rfloor > \epsilon' h_0$, we divide $({}^{x_j}/_{h_0} - \lfloor {}^{x_j}/_{h_0} \rfloor)$ into at most ${}^1/_{\epsilon'}$ stripes with height less or equal $\epsilon' h_0$. So assign to C_j additional space of height $\epsilon' h_0$ in no more than ${}^1/_{\epsilon'}$ strips, which are already occupied until height h_0. In the same way as the remaining stripes we handle configurations of height less than h_0. Since there are at most $2M$ configurations with nonzero height, we get at most ${}^{2M}/_{\epsilon'} = {}^2/_{\epsilon'^3} \leq {}^{2 \cdot 4^3}/_{\epsilon^3} \leq k$ additional assignments of height $\epsilon' h_0$, which can be distributed to k strips. Thus by this assignment policy, where the configurations are balanced, each strip has allocated area of height at most $(1 + \epsilon')h_0$ for at most 2 different configurations. □

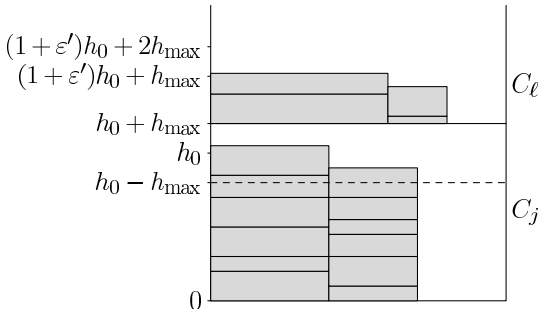

FIG. 7.13: S_i with C_j and C_ℓ.

Integral Packing. The next Lemma shows how to get from a fractional packing to a feasible integral packing. A proof is given in the full paper.

Lemma 7.14. *Let $x = (x_1, \ldots, x_q)$ be a solution of $LP(L_{sup})$ with at most $m' \leq 2M$ nonzero entries $x_1, \ldots, x_{m'}$. For $k \geq \lceil {}^{128}/_{\epsilon^3} \rceil$ we can convert x to a feasible packing for the wide rectangles with height at most $(1 + \epsilon')h_0 + 2h_{\max}$ and at most 2 different configurations per strip.*

Since we can guarantee that there are at most 2 different configurations per strip, the additive constant will be improved, while the running-time is still polynomial in n and ${}^1/_{\epsilon}$.

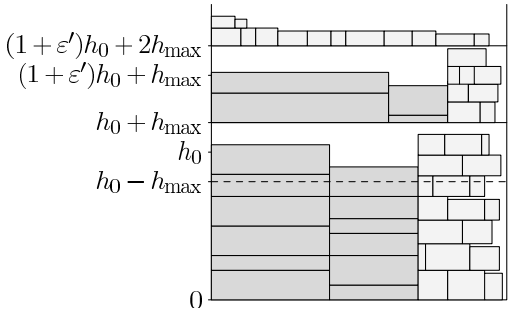

FIG. 7.15: S_i after packing the narrow rectangles.

Our last step is to pack the narrow rectangles. We use a modified version of the NFDH algorithm : For strip S_i as above we pack narrow rectangles with NFDH into the empty space next to the configurations until the total height is at most $(1+\epsilon')h_0+2h_{\max}$. After that we repeat the process for strip S_{i+1}. When all strips are filled in this way, we draw a horizontal line at height $(1+\epsilon')h_0+2h_{\max}$ in each strip and pack the remaining narrow rectangles with Algorithm 7.2 on top (see Fig 7.13 and 7.15). Thus we can ensure by Lemma 7.4 that the maximum difference of the heights of two arbitrary strips is at most $h_{\max}$ (see Fig 7.15). Let h_{final} denote the height of the packing after packing the narrow rectangles.

Lemma 7.16. *Let $k \geq \lceil 128/\epsilon^3 \rceil$. If $h_{final} \geq (1+\epsilon')h_0 + 2h_{\max}$, then we have $h_{final} \leq \frac{SIZE(L_{sup} \cup L_{narrow})}{k(1-\epsilon')} + 6h_{\max} + \epsilon' h_0$.*

For details we refer to the full paper. The next lemma is shown in [9] for the Linear Program corresponding to Strip Packing, but obviously also holds for our linear program $LP(L_{sup})$.

Lemma 7.17. *[9] For the rounded instance L_{sup} and L_{wide} the inequalities $LIN(L_{sup}) \leq LIN(L_{wide})\left(1+\frac{1}{M\epsilon'}\right)$ and $SIZE(L_{sup}) \leq SIZE(L_{wide})\left(1+\frac{1}{M\epsilon'}\right)$ hold.*

The entire algorithm is now defined as follows :

---

1 Set $\epsilon' := \epsilon/4$ and $M := (1/\epsilon')^2$.

2 Partition L into L_{wide} and L_{narrow}.

3 Construct L_{sup}, so that $L_{wide} \leq_g L_{sup}$ and there are only M different widths in L_{sup}.

4 Solve the linear program $LP(L)$.

5 Construct a feasible solution for L_{sup} by balancing the configurations.

6 Use modified $NFDH$ to pack the rectangles in L_{narrow} into the remaining space and on top of the strips.

Theroem 7.18. *If $k \geq \lceil 128/\epsilon^3 \rceil$ the Algorithm 7.4.2 generates for an instance L of MSP a packing of height at most $(1+\epsilon)Opt(L) + \mathcal{O}(1)h_{\max}$.*

Bibliographie

[1] N. Bansal, A. Caprara, K. Jansen, L. Prädel, and M. Sviridenko. How to maximize the total area of rectangle packed into a rectangle. *In Preparation*, 2009.

[2] N. Bansal, X. Han, K. Iwama, M. Sviridenko, and G. Zhang. Harmonic algorithm for 3-dimensional strip packing problem. In *Proceedings of the eighteenth ACM-SIAM symposium on Discrete algorithm (SODA)*, pages 1197–1206, 2007.

[3] M. Bougeret, P-F. Dutot, K. Jansen, C. Otte, and D. Trystram. Approximation algorithm for multiple strip packing. In *Proceedings of the 7th Workshop on Approximation and Online Algorithms (WAOA)*, 2009.

[4] A. Caprara. Packing 2-dimensional bins in harmony. In *IEEE 43th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.

[5] EG Coffman Jr, MR Garey, DS Johnson, and RE Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9 :808, 1980.

[6] K. Jansen, L. Prädel, and U. M. Schwarz. Two for one : Tight approximation of 2d bin packing. *Submitted to : Algorithms and Data Structures Symposium (WADS 2009)*, 2009.

[7] K. Jansen and R. Solis-Oba. An asymptotic approximation algorithm for 3d-strip packing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA)*, page 152. ACM, 2006.

[8] K. Jansen and R. Solis-Oba. New approximability results for 2-dimensional packing problems. *Lecture Notes in Computer Science*, 4708 :103, 2007.

[9] C. Kenyon and E. Rémila. A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, pages 645–656, 2000.

[10] F. Miyazawa and Y. Wakabayashi. Cube packing. *LATIN 2000 : Theoretical Informatics*, pages 58–67, 2000.

[11] I. Schiermeyer. Reverse-fit : A 2-optimal algorithm for packing rectangles. *Lecture Notes in Computer Science*, pages 290–290, 1994.

[12] U. Schwiegelshohn, A. Tchernykh, and R. Yahyapour. Online scheduling in grids. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10, 2008.

[13] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26 :401, 1997.

[14] A. Tchernykh, J. Ramírez, A. Avetisyan, N. Kuzjurin, D. Grushin, and S. Zhuk. Two level job-scheduling strategies for a computational grid. *Parallel Processing and Applied Mathematics*, pages 774–781, 2006.

[15] D. Ye, X. Han, and G. Zhang. On-Line Multiple-Strip Packing. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications (COCOA)*, page 165. Springer, 2009.

[16] SN Zhuk. Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1) :73–85, 2006.

Chapitre 8

Approximating the non-contiguous Multiple Organization Packing Problem [Bougeret et al., 2010b]

Marin bougeret*‡, Pierre-François Dutot*, Klaus Jansen†, Christina Otte† and Denis Trystram*

*LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

†Department of Computer Science, Christian-Albrechts-University, Kiel, Germany
kj, cot@informatik.uni-kiel.de

‡This work is supported by DGA-CNRS

†Work supported by EU project "AEOLUS : Algorithmic Principles for Building Efficient Overlay Computers", EU contract number 015964, and DFG project JA612/12-1, "Design and analysis of approximation algorithms for two- and threedimensional packing problems"

Abstract

We present in this paper a 5/2-approximation algorithm for scheduling rigid jobs on multi-organizations. For a given set of n jobs, the goal is to construct a schedule for N organizations (composed each of m identical processors) minimizing the maximum completion time (makespan). This algorithm runs in $O(n(N+\log(n))\log(np_{max}))$, where p_{max} is the maximum processing time of the jobs. It improves the best existing low cost approximation algorithms. Moreover, the proposed analysis can be extended to a more generic approach which suggests different job partitions that could lead to low cost approximation algorithms of ratio better than 5/2.

8.1 Problem statement

In this paper we consider the problem of scheduling rigid jobs on Multi-organizations. An organization is a set of m identical available processors. A job j must be executed on q_j processors (sometimes called the *degree of parallelism*) during p_j units of time. The q_j processors must be allocated on the same organization. The makespan of the schedule is defined as the maximum finishing time over all the jobs. Given a set of n jobs, the goal is to find a non-overlapping schedule of all the jobs on N organizations while minimizing the makespan.

This problem is closely related to strip packing problems. Indeed, if we add the constraint of using contiguous processors, then scheduling a job j on q_j contiguous processors during p_j units of time is equivalent to packing a rectangle of width q_j and height p_j.

Related works. Strip packing, rigid jobs scheduling and Multi-organizations scheduling problems are all strongly $\mathcal{NP}$-hard, and Zhuk [14] showed that there is no polynomial time approximation algorithm with absolute ratio better than 2 for strip packing.

For Strip Packing problem, Coffman et al. gave in [3] an overview about performance bounds for shelf-oriented algorithms as $NFDH$ (Next Fit Decreasing Height) and $FFDH$ (First Fit Decreasing Height). These algorithms have a approximation ratio of 3 and 2.7, respectively. Schiermeyer [10] and Steinberg [12] presented independently an algorithm for Strip Packing with absolute ratio 2. A further important result for the Strip Packing problem is an AFPTAS with additive constant $\mathcal{O}(1/\epsilon^2)$ of Kenyon and Rémila [9]. This constant was improved by Jansen and Solis-Oba, who presented in [8] an APTAS with additive constant 1. Concerning the multi-strip packing problem, there is a $2+\epsilon$ approximation in [13] whose algorithmic cost is doubly exponential in $\frac{1}{\epsilon}$. In [1] we gave a 2 approximation with a large algorithmic cost and an AFPTAS for this problem.

Let us now review the related work about rigid job scheduling. For one organization, the famous List Algorithm for scheduling with resource constraints of Garey and Graham [6] can be applied (when there is only one resource to share) to schedule rigid jobs, and is then a 2 approximation. The rigid job scheduling problem on multi-organization has been studied with an on-line setting in [11]. The authors achieved a ratio of 3 without release times (and 5 with release times). Notice that these results do not require the knowledge of the processing times of the jobs. Moreover, the organizations may have a different number of processors. The rigid job scheduling problem on multi-organizations has been extended in [5] for the case where the jobs are submitted to local queues on each cluster with the extra constraint that the initial local schedules must not be worsened. The authors provide a 3-approximation.

Generally, the results about rigid job scheduling cannot be adapted to the more constrained contiguous version. To the best of our best knowledge, there is still no (reasonable) α such that for any instance I, $Optc(I) \leq \alpha Optnc(I)$ (where $Optc$ denotes the contiguous optimal value and $Optnc$ the non-contiguous one). The authors of [4] show that $\alpha > 1$ by constructing a (rather) simple instance with 8 jobs and 4 machines.

Our contribution. In this paper, we present a $\frac{5}{2}$ approximation algorithm for the rigid job scheduling problem on multi-organizations that runs in $O(n(N+\log(n))\log(np_{max}))$, where p_{max} is the maximum processing time of the jobs. Moreover, we suggest how the approach used for the $5/2$-algorithm could extended to get approximation algorithms with better ratio and a low algorithmic cost.

Organization of the Paper. The preliminaries for the $5/2$-approximation are in Section 8.2. In Section 8.3.1 to 8.3.4 we describe how to construct a preallocation of the "big" jobs that fits in the targeted makespan. In Section 8.4 we show how to turn this preallocation into a compact schedule, and in Section 8.5 we analyze the complexity of the algorithm. The discussions on the approach are in Section 8.6.

8.2 Principle and definitions

Let us now give some definitions that are used throughout the proofs and the description of the algorithm. We first extend the previous p_j and q_j notations to $Q(X)$ and $P(X)$ where X is a set of jobs. We also define the surface (sometimes also called the area) of a set of jobs as $S(X) = \Sigma_{j\in X} q_j p_j$. A *layer* is a set of jobs which are scheduled sequentially on the same organization. The length of a layer Lay is $P(Lay)$, the sum of the processing time of all the jobs in Lay. A *shelf* is a set of jobs which are scheduled on the same organization, and which start at the same time. Given a shelf sh, the value $Q(sh)$ is called the *height* of sh. What we call a *bin* can be seen as a reservation of a certain number of processors (generally m) during a certain amount of time. The algorithm will add some jobs to bins, and given a bin b, we denote by $Q(b)$ the value $\Sigma_{\{j\in b\}} q_j$. Given a sequence of bins seq, we denote by $Q(seq)$ the value $\Sigma_{b\in seq} Q(b)$. These notations are extended in the same way for P and S. In the whole paper, we consider that the sets of jobs used as parameters in the algorithms are modified after the calls.

Let us sketch how $5/2$ algorithm is constructed. Let OPT denote the value of an optimal solution. We target a $\frac{5}{2}$ ratio by both ensuring that, for each organization at least half of the processors are used at any time before the starting time of the last job, and that the small jobs (whose processing time is lower than $OPT/2$ and height lower than $m/2$) are scheduled at the end. Thus, if the makespan of the final schedule is due to a small job, it is lower than the processing time of the small job plus the starting time of this job, implying a makespan lower than $OPT/2 + 2OPT = 5OPT/2$. As the optimal value is not known, we use the well known dual approximation technique [7]. Let w denote the current guess of OPT. The schedule is built in three steps. In the first one we compute a preallocation π_0 of the "big" ($p_j > w/2$ or $q_j > m/2$) jobs. Then we apply a list algorithm which turns π_0 into a "compact" schedule π_1 (see Section 8.4). Finally, the final schedule π is constructed by adding to π_1 the small remaining jobs using again a list algorithm (see also Section 8.4).

Let us define the following sets :

- let $L_H = \{j | q_j > m/2\}$ be the set of high jobs
- let $L_{XL} = \{j | p_j > 3w/4\}$ be the set of extra long jobs
- let $L_L = \{j | 3w/4 \geq p_j > w/2\}$ be the set of long jobs
- let $L_B = (L_{XL} \bigcup L_L) \bigcap L_H$ be the set of huge jobs
- let $I' = L_H \bigcup L_{XL} \bigcup L_L$

We will prove that either we schedule I with a resulting makespan lower than $5w/2$, or $w < OPT$. Notice that for the sake of simplicity we did not add the "reject" instructions in the algorithm. Thus we consider in all the proof that $w \geq OPT$, and it is implicit that if one of the claimed properties is wrong during the execution, the considered w should be rejected. Notice that we only consider the w values such that $Q(L_{XL} \bigcup L_L) \leq Nm$ and $P(L_H) \leq Nw$.

We start by providing in Section 8.3 the three phase algorithm *Build_Prealloc* that builds the preallocation π_0 of the jobs of I'. We will denote by π_0^i the set of preallocated jobs in organization O_i. In phase 1 we preallocate the high jobs. In phase 2 and phase 3 we preallocate the long and extra long jobs by first packing shelves of jobs into bins, and then putting these bins into organizations. An example of a preallocation is depicted Figure 8.2.

8.3 Construction of the preallocation

8.3.1 Phase 1

Let N_1 be the number of organizations used in phase 1. In phase 1, the jobs of L_H are packed in N_1 organizations. The $Create\_Layer(X, l)$ procedure creates a layer Lay of length at most l, using a Best Fit (according to the processing times) policy (BFP). Thus, $Create\_Layer(X, l)$ add at each step the longest job that fits. Thus, phase 1 calls for each organization (until L_H is empty) $Create\_Layer(L_H, 5w/2)$.

Let us introduce some notations. Let Lay_i denote the set of jobs scheduled in the layer created in organization O_i. Let L_{XL}^1 and L_L^1 denote the remaining jobs of L_{XL} and L_L after phase 1. Thus, for the moment we have $\pi_0^i = Lay_i$ for all $i \leq N_1$.

Lemma 8.1 (phase 1). *If $\exists i_0 < N_1$ such that $P(\pi_0^{i_0}) \leq 2w$ then it is straightforward to pack all the jobs of I'. Otherwise, we get $\forall i \in \{1, \ldots, N_1 - 1\}$, $S(\pi_0^i) > wm$ and $N_1 \leq \lceil N/2 \rceil$.*

Proof First let us notice that phase 1 ends, as $P(L_H) \leq Nw$ and $P(\pi_0^i) > w$ for every organization where we do not run out of jobs to schedule. We first suppose that $\exists i_0 < N_1$ such that $P(\pi_0^{i_0}) \leq 2w$. In this case we just have to prove that it is straightforward to preallocate $L_{XL} \bigcup L_L$. We proceed by contradiction by supposing that we never ran out of jobs of $L_{XL} \bigcup L_L$. When the algorithm creates a layer for a organization i, we know due to the BFP order that it will pack at least two jobs of L_B, if L_B is not empty. The hypothesis implies that during the execution of phase 1, $L_H \setminus L_B$ was empty before L_B. Thus, for $i < N_1$, there is at least two jobs of L_B in π_0^i, meaning that $\forall i$ with $1 \leq i < N_1, Q((L_{XL} \bigcup L_L) \bigcap \pi_0^i) > m$.

Concerning the $N - N_1$ other organizations, we can create shelves of jobs of $L_L \bigcup L_{XL}$ using a best fit according to the height (BFH), implying that each shelf has a height of at least $2m/3$ according to Lemma 8.3. Packing two shelves in each organization, we get $\forall i > N_1, Q((L_{XL} \bigcup L_L) \bigcap \pi_0^i) > 4m/3 > m$.

Finally, let us check what is scheduled in organization N_1. If two jobs of L_B are scheduled in this organization, then $Q((L_{XL} \bigcup L_L) \bigcap \pi_0^{N_1}) > m$. If one job of L_B is scheduled, then we create one shelf of jobs of $L_{XL} \bigcup L_L$, and $Q((L_{XL} \bigcup L_L) \bigcap \pi_0^{N_1}) >$

$m/2 + 2m/3$. If no huge job is scheduled in organization N_1, we pack as before two shelves of jobs of $L_{XL} \bigcup L_L$. Thus, if in every case we have $Q((L_{XL} \bigcup L_L) \bigcap \pi_0^{N_1}) > m$. Thus, we get $Q((L_{XL} \bigcup L_L)) > Nm$, which is impossible.

Let us prove the second part of the lemma. First notice that for any $i < N_1, S(\pi_0^i) > 2w^m/2 = mw$. Moreover, we have $2(N_1 - 1)w < \Sigma_{i=1}^{N_1} P(\pi_0^i) = P(L_H) \leq Nw$, implying $N_1 \leq \lceil N/2 \rceil$. □

Thus, we now assume until the end of the proof that we are in the second case of Lemma 8.1 where $\forall i \in \{1, \ldots, N_1 - 1\}$, $S(\pi_0^i) > mw$ and $N_1 \leq \lceil N/2 \rceil$.

8.3.2 Phase 2

In phase 2 the jobs of $L_{XL}^1 \bigcup L_L^1$ are scheduled in organization N_1 by creating shelves according to what is already scheduled in organization N_1. We denote by L_{XL}^2 and L_L^2 the remaining jobs of L_{XL}^1 and L_L^1 after phase 2. Let us first define two procedures used for phase 2 and phase 3.

The procedure $Pack\_Shelf(X, b, f)$ creates a shelf sh using the Best Fit (according to the height) policy (BFH), and packs it into bin b. The f parameter represents the available height of b (meaning that b corresponds to f free processors during a certain amount of time), implying of course that $Q(sh) \leq f$. Thus $Pack\_Shelf(X, b, f)$ adds at each step the highest possible job of X that fits. We assume that the length of the bin is larger than p_j, for all $j \in X$.

The procedure $GreedyPack(X, seq)$ creates for each empty bin $b \in seq$ one shelf of jobs of X using $Pack\_Shelf(X, b, m)$. This procedure returns the last bin in which a shelf has been created. Let us now come back to the description of phase 2.

Depending on the set of jobs already scheduled in O_{N_1}, the $Create\_Padding()$ procedure creates n_{bin_L} empty bins of length $3w/4$ and $n_{bin_{XL}}$ empty bins of length w, which are added in organization O_{N_1}. Let us define for each case how many bins of each type are created by $Create\_Padding()$:

- If $P(Lay_{N_1}) \in]3w/2, 7w/4]$ then set $(n_{bin_L}, n_{bin_{XL}})$ to $(1, 0)$
- If $P(Lay_{N_1}) \in]w, 3w/2]$ then set $(n_{bin_L}, n_{bin_{XL}})$ to $(0, 1)$
- If $P(Lay_{N_1}) \in]3w/4, w]$ then
  if $Q(L_L^1) \geq 5/4$ then set $(n_{bin_L}, n_{bin_{XL}})$ to $(2, 0)$
  else set $(n_{bin_L}, n_{bin_{XL}})$ to $(0, 1)$
- If $P(Lay_{N_1}) \in]w/2, 3w/4]$ then set $(n_{bin_L}, n_{bin_{XL}})$ to $(1, 1)$
- If $P(Lay_{N_1}) \in [0, w/2]$ then set $(n_{bin_L}, n_{bin_{XL}})$ to $(0, 2)$

Let pad_L be a sequence of n_{bin_L} bins of length $3w/4$ and pad_{XL} be a sequence of $n_{bin_{XL}}$ bins of length w. $Create\_Padding()$ returns (pad_L, pad_{XL}). All in all, phase 2 can be described by the following procedure calls :

- Let $(pad_L, pad_{XL}) = Create\_Padding()$
- $GreedyPack(L_{XL}^1, pad_{XL})$
- $GreedyPack(L_L^1, pad_L)$
- $GreedyPack(L_L^1, pad_{XL})$

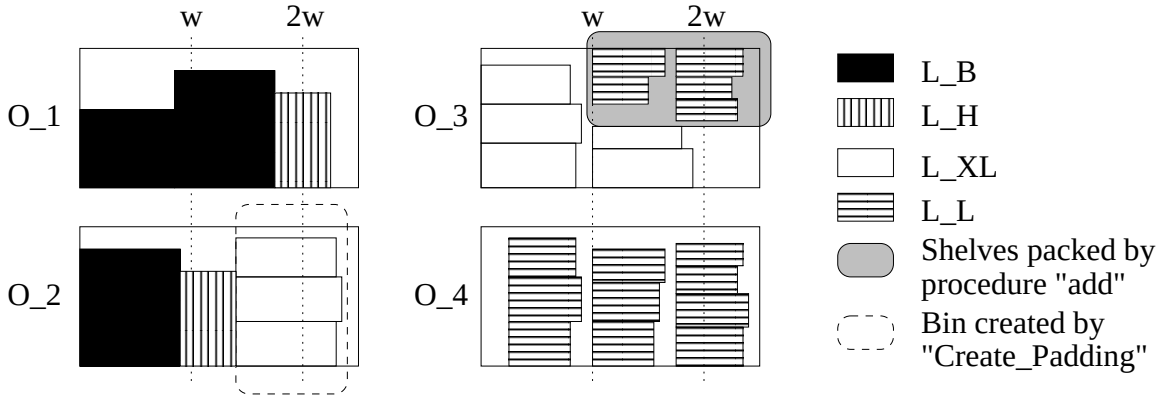

FIG. 8.2: An example of pre-allocation

8.3.3 Phase 3

In phase 3 we first schedule the jobs of L^2_{XL} using the $N_2 = N - N_1$ remaining organizations. Then, we schedule the jobs of L^2_L using also this N_2 organizations. Finally, the possibly remaining jobs of L^2_L are added to the last bin used for the extra long jobs. Therefore, let us define the $add(X, b)$ procedure. The $add(X, b)$ procedure packs one or two "small" shelves of jobs of X in the bin b (starting from the top of the bin for the sake of clarity). Notice that, as b will be the last bin used for extra long jobs, the available height (for the jobs of X) in b will be generally lower than m. Here is the description of $add(X, b)$:

- If the left side of b is at time w then let $l = 2$ else let $l = 1$
- Repeat l times the call $Pack\_Shelf(X, b, m - Q(b))$ and pack the created shelves in b.

An example of a call to the *add* procedure is given in Figure 8.2 for the case where $l = 2$.

We now define two sequences of bins seq_{XL} and seq_L, such that every bin of seq_{XL} (resp. seq_L) will (possibly) contains one shelf of jobs of L^2_{XL} (resp. L^2_L). Notice that a free organization can be seen as two bins of length w (and height m), three bins of length $3w/4$, or one bin of length w and two bins of length $3w/4$. Thus, seq_{XL} is composed of $2(N - N_1)$ bins $\left(b_1, \ldots, b_{2(N-N_1)}\right)$ of length w, considering that we created two bins in each of the organizations $\{O_{N_1+1}, \ldots O_N\}$, starting from O_{N_1+1}. This implies that for all $i \geq 1$, bins b_{2i-1} and b_{2i} are in O_{N_1+i}. The sequence seq_L is composed of $3(N - N_1)$ bins $\left(b'_1, \ldots, b'_{3(N-N_1)}\right)$ of length $3w/4$, considering that we created three bins in each of the organizations $\{O_{N_1+1}, \ldots O_N\}$, from O_N to O_{N_1+1}. This implies that for all $i \geq 1$, bins b'_{3i-2}, b'_{3i-1} and b'_{3i} are in O_{N-i+1}. Notice that these two sequences are not ordered in the same way.

All in all, phase 3 can be described by the following procedure calls :

- Let $last = GreedyPack(L^2_{XL}, seq_{XL})$
- $GreedyPack(L^2_L, seq_L)$
- $add(L^2_L, last)$

Let start the analysis of phase 3 with a remark about $Pack\_Shelf(X, b, f)$.

Lemma 8.3. *Let Sh denote the shelf created by $Pack\_Shelf(X, b, f)$. If we know that the k highest jobs of X fit in f, then $Q(Sh) > \frac{k}{k+1}f$.*

Proof Let x be the cardinal of X. Let us assume that $q_i \geq q_{i+1}$ for $1 \leq i < x$. Let $i_0 \geq k+1$ be the first index such that job i_0 is not in Sh. Let $a = \Sigma_{i=1}^{i_0-1} q_i$. We have $Q(Sh) \geq a \geq (i_0 - 1)q_{i_0} > (i_0 - 1)(f - a)$ leading to $a > \frac{i_0-1}{i_0}f \geq \frac{k}{k+1}f$. □

Lemma 8.4 (phase 3)**.** *If there remains an unscheduled job after phase 3, then $S(L_{XL}^2 \bigcup L_L^2) > (N_2 + 1/8)mw$.*

Proof Let us first suppose that $L_{XL}^2 \neq \emptyset$. Let $a_{XL} = 2(N - N_1)$ be the number of bins in seq_{XL}. After having filled the first $a_{XL} - 1$ bins (using a width of at least $2/3$ according to Lemma 8.3), the width of remaining jobs of L_{XL}^2 is strictly larger than m. Thus we get $Q(L_{XL}^2) > 2m/3(a_{XL} - 1) + m = 4m/3N_2 + m/3$ and $S(L_{XL}^2) > (N_2 + 1/4)mw$.

We now suppose that $L_{XL}^2 = \emptyset$. In every organization that contains two bins of jobs of L_{XL}^2, the total scheduled area is strictly larger than $2 \times 2m/3 \times 3w/4 = wm$. In every organization that contains three bins of jobs of L_L^2, the total scheduled area is strictly larger than $3 \times 2m/3 \times w/2 = wm$. We have to consider two cases according to the position of the last bin *last* (the left side of *last* may be located at time 0 or w). Let i_0 be the index of the organization that contains *last*.

In the first case where the left side of *last* is at time 0, two bins (of length $3w/4$ and height m) were created after the bin *last* in organization O_{i_0}. Then, if the remaining jobs of L_L^2 do not fit in *last*, the total area of the jobs scheduled in organization O_{i_0} is strictly larger than $(2^{2m}/_3 + m)w/2 > 7wm/6$. Then we just sum the area packed over all the organizations, and get the desired result.

In the second case where the left side of *last* is at time w (as depicted in Figure 8.2), the only room in organization O_{i_0} to schedule jobs of L_L^2 is in *last*. In organization O_{i_0}, the area of (extra long) jobs contained in the first bin is strictly larger than $wm/2$. The *add* procedure will create two shelves (one next to the other) of jobs of L_L^2 in *last*.

Let *last'* and L'_L be the set of jobs in *last* and L_L^2 respectively, just before the call of the *add* procedure. If $Q(last') > m/2$, and as the remaining jobs of L_L^2 don't fit in *last*, we have that $Q(L'_L) > m - Q(last')'$. This implies $S(last' \bigcup L'_L) > 3w/4Q(last') + w/2(m - Q(last)') > 5wm/8$. If $Q(last') \leq m/2$ (see Figure 8.2) then *add* creates a first shelf of jobs of L_L^2 of height at least $(m-Q(last'))/2$, and then tries to pack the remaining jobs in the second shelf. Thus in this case, $S(last' \bigcup L'_L) > 3w/4Q(last') + w/2\left(\frac{(m-Q(last'))}{2} + m - Q(last')\right) > 3mw/4$.

□

8.3.4 Main algorithm

In this section we recall the overall algorithm that builds the preallocation, and we provide the main proof of the preallocation. Notice that we drop the L_L^i and L_{XL}^i notations for writing the algorithm as we consider that the sets of jobs (used as parameters in the procedures) are modified after the calls.

Build_Prealloc(I')

Phase 1	[1] Let $i = 0$
	[2] Let $i = i + 1$, and let $Lay_i = Create\_Layer(L_H, {}^{5w}/_2)$
	Pack Lay_i in organization S_i from time 0
	[3] Repeat step 3 until L_H is empty
Phase 2	[4] Let $(pad_L, pad_{XL}) = Create\_Padding()$
	[5] Let $last = GreedyPack(L_{XL}, pad_{XL})$
	[6] Call $GreedyPack(L_L, pad_L)$
	[7] Call $GreedyPack(L_L, pad_{XL})$
Phase 3	[8] Let seq_{XL} and seq_L be defined as described in Section 8.3.3
	[9] Let $last_2 = GreedyPack(L_{XL}, seq_{XL})$
	[10] If $last_2$ is not null, set $last$ to $last_2$
	[11] Call $GreedyPack(L_L, seq_L)$
	[12] Call $add(L_L, last)$

Theroem 8.5. *Build_Prealloc(I') creates a preallocation π_0 of makespan lower than ${}^{5w}/_2$.*

Proof Remind that L^1_{XL} and L^1_L denote the remaining jobs of L_{XL} and L_L after Phase 1. The makespan of the preallocation is by construction lower than ${}^{5w}/_2$. We know that according to Lemma 8.1 phase 1 terminates and the area scheduled in the first $N_1 - 1$ organizations is greater than $(N_1 - 1)wm$. We proceed by contradiction by supposing that $L^1_{XL} \bigcup L^1_L$ is not empty after Phase 2 and Phase 3, and showing that $S(I') > Nmw$. We proceed by case analysis according to what is scheduled in O_{N_1}.

If $P(Lay_{N_1}) > \frac{7}{4}w$, then $S(Lay_{N_1}) > \frac{7}{8}mw$ and *CreatePadding* doesn't create any bin. If L^1_{XL} and L^1_L are not completely scheduled by phase 3, then according to Lemma 8.4 we get $S(L^1_{XL} \bigcup L^1_L) > (N_2 + \frac{1}{8})mw$. Thus in this case we have $S(Lay_{N_1} \bigcup L^1_{XL} \bigcup L^1_L) > (N_2 + 1)mw$, implying $S(I') > Nmw$.

If $\frac{7}{4}w \geq P(Lay_{N_1}) > \frac{3}{2}w$, then $S(Lay_{N_1}) > \frac{3}{4}mw$ and *CreatePadding* creates one bin of length $\frac{3}{4}w$. Recall that the jobs of L^1_L are first scheduled in pad_L. If $Q(pad_L)$ is larger than $\frac{m}{2}$, then $S(Lay_{N_1} \bigcup pad_L) > \frac{3}{4}mw + \frac{1}{4}mw = mw$. Thus, the total area packed in the first N_1 is strictly larger than N_1wm. Then, according to Lemma 8.4, $L^2_{XL} \bigcup L^2_L$ must fit in the N_2 remaining organizations. If $Q(pad_L) \leq \frac{m}{2}$, then the N_2 remaining organizations are available for L^1_{XL}. Thus, if $L^1_{XL} \neq \emptyset$ at the end, then $S(L^1_{XL}) > (N_2 + \frac{1}{4})mw$, and $S(Lay_{N_1} \bigcup L^1_{XL}) > (N_2 + 1)mw$.

If $\frac{3}{2}w \geq P(Lay_{N_1}) > w$, then $S(Lay_{N_1}) > \frac{1}{2}mw$ and *CreatePadding* creates one bin of length w. If $Q(pad_{XL})$ is larger than $\frac{2m}{3}$ then $S(Lay_{N_1} \bigcup pad^1_{XL}) > mw$ and we conclude with Lemma 8.4. Otherwise, the N_2 remaining organizations are available for L^1_L. Moreover, remind that in this case the only bin in pad_{XL} will be used for jobs of L_L during the call of *add*. Then, if L^1_L does not fit, we have $Q(L^1_L \bigcup L^1_{XL}) > (2N_2+1)m$ and $S(Lay_{N_1} \bigcup L^1_L \bigcup L^1_{XL}) > \frac{mw}{2} + N_2wm + \frac{wm}{2} = (N_2 + 1)mw$.

If $w \geq P(Lay_{N_1}) > \frac{3}{4}w$, then $S(Lay_{N_1}) > \frac{3}{8}mw$ and two cases are possible according to the value of $Q(L^1_L)$. If $Q(L^1_L) \geq \frac{5m}{4}$, *CreatePadding* creates two bins of length $\frac{3}{4}w$. Then, $S(Lay_{N_1} \bigcup pad_L) > (\frac{3}{8} + \frac{5}{8})mw$ and we conclude with Lemma 8.4. Otherwise, if $Q(L^1_L) < \frac{5m}{4}$, *CreatePadding* creates one bin pad_{XL} of length w. If $Q(pad_{XL})$ (after

the call line 5) is larger than $\frac{2m}{3}$ then $S(Lay_{N_1} \bigcup pad^1_{XL}) > \frac{7}{8}mw$ and we conclude with Lemma 8.4. Otherwise, jobs of L^1_{XL} are all scheduled in pad_{XL}. As $N_2 \geq 1$, at least three bins are available for L^1_L, which is sufficient given that $Q(L^1_L) < \frac{5m}{4}$.

If $\frac{3}{4}w \geq P(Lay_{N_1}) > \frac{1}{2}w$, then $S(Lay_{N_1}) > \frac{1}{4}mw$ and $CreatePadding$ creates one bin of length w and one bin of length $\frac{3}{4}w$. If extra long jobs are not scheduled at the end of the algorithm, then $S(Lay_{N_1} \bigcup pad_{XL}) > \frac{3}{4}mw$. Since L^2_{XL} do not fit into N_2 free organizations, we have also $S(L^2_{XL}) > (N_2 + \frac{1}{4})mw$. Thus we conclude that the extra long jobs are successfully scheduled. Let us suppose now that the long jobs are not completely scheduled. If $Q(pad_{XL}) \geq \frac{5m}{9}$ then $S(Lay_{N_1} \bigcup pad_{XL} \bigcup pad_L) > (\frac{1}{4} + \frac{5}{12} + \frac{1}{3})mw = mw$. Otherwise, let L'_L denote the set of remaining jobs of L^1_L just before the call to *add*. The area scheduled in the N_2 last organizations is larger than the one scheduled in the optimal. If L'_L does not fit in pad_{XL} during the call to *add*, then $Q(L^1_{XL} + L'_L) > m$ and $S(Lay_{N_1} \bigcup pad_L \bigcup pad_{XL} \bigcup L'_L) > (\frac{1}{4} + \frac{1}{3} + \frac{1}{2})mw > mw$.

If $\frac{1}{2}w > P(Lay_{N_1})$, $CreatePadding$ creates two bins of length w. If $N_1 > 1$, then $S(\bigcup_{i=1}^{N_1} Lay_i) > S(\bigcup_{i=1}^{N_1-2} Lay_i) + \frac{5}{4}mw > (N_1 - 1)mw + \frac{1}{4}mw$ because the first job of Lay_{N_1} does not fit in the previous organization. Thus, if $Q(L^1_{XL}) > m$ then we have $S(\bigcup_{i=1}^{N_1} Lay_i \bigcup pad_{XL}) > N_1 mw$ and we conclude with Lemma 8.4. Otherwise, we have an empty bin in the sequence pad_{XL} and N_2 free organizations available for L^1_L. Let L'_L be L^1_L before the call to *add*. If *add* does not schedule L'_L in *last* then $Q(L^1_{XL}) + Q(L'_L) > m$ and $S(\bigcup_{i=1}^{N_1} Lay_i \bigcup L^1_{XL} \bigcup L'_L) > (N1 - \frac{3}{4} + \frac{1}{3} + \frac{1}{2})mw > N_1 mw$. If $N_1 \leq 1$ then we have two bins of length w in each of the N organizations, which is of course sufficient to pack $L^1_{XL} \bigcup L^1_L$.

□

8.4 From the preallocation to the final schedule

From now on, we suppose that the preallocation π_0 is built. For each organization O_i, π_0 indicates first a (possibly empty) sequence of high jobs $j^i_1, \ldots, j^i_{x_i}$ that have to be scheduled sequentially from time 0. Then, π_0 contains an ordered sequence of shelves $Sh^i_1, \ldots, Sh^i_{x'_i}$. Moreover, the makespan of π_0 is by construction less than $5w/2$.

Définition 8.6. *Let $u_i(t)$ be the utilization of organization O_i at time t, i.e. $u_i(t)$ is the sum of all the q_j for any job j which scheduled on organization i at time t. A schedule is $1/2$ compact if and only if for every organization O_i there exists a time t_i such that for all $t \leq t_i, u_i(t) \geq m/2$ and u_i restricted to $t > t_i$ is not increasing.*

Let us now describe the algorithm LS_{π_0} which turns π_0 into a $1/2$ compact schedule π_1 of I'. We first define the procedure $Add\_Asap(X, O_i)$ which scans organization O_i from time 0, and for every time t starts any possible job(s) in X that fit(s) at time t. The LS_{π_0} works as follows : for every organization O_i, pack first sequentially the high jobs j^i_x for $1 \leq x \leq x_i$ and then call $Add\_Asap(Sh_x, O_i)$ for $1 \leq x \leq x'_i$.

Lemma 8.7. *The makespan of π_1 is lower than the one of π_0, and π_1 is $1/2$ compact.*

Proof Let σ_i be a schedule in a (single) organization O_i (of makespan C_i), let X be a set of jobs and let σ'_i be the schedule (of makespan C'_i) produced by $Add\_Asap(X, O_i)$. If σ_i is $1/2$ compact and if forall $j \in X, q_j \leq m/2$, then σ'_i is $1/2$ compact. The proof is straightforward by induction on the cardinality of X. Moreover, if $\sum_X q_j \leq m$, then $C'_i \leq C_i + max_X p_j$ because in the worst case all the jobs of X only start at time C_i. Using these two properties, we prove the lemma for every organization O_i by induction on the number of call(s) to $Add\_Asap(Sh_x, O_i)$. □

Remark 8.8. *Notice that in Lemma 8.7 we do not take care of the particular structure which occurs when add creates two shelves of jobs of L_L as depicted Figure 8.2. However, it is easy to see that the proof can be adapted.*

Now that π_1 is built, we add the small remaining jobs $(I \setminus I')$ using a list algorithm that scans all the organizations from time 0 and schedules as soon as possible any non scheduled job. Let π denote the obtained schedule.

Theroem 8.9. *The makespan of π is lower than $5w/2$.*

Proof The proof is by induction on the cardinal of $I \setminus I'$. At the beginning, π_1 is $1/2$ compact, as proved in Lemma 8.7. Each time a job j is scheduled by the list algorithm, the obtained packing remains $1/2$ compact because $q_j \leq \frac{m}{2}$. Thus it is clear that π is $1/2$ compact.

Let us assume that the makespan of π is due to a job $j \in I \setminus I'$ that starts at time s. As π is $1/2$ compact, this implies that when scheduling job j we had $t_i \geq s$ for any organization i. Thus, we have $S(I) > \sum_{i=1}^{N} \frac{t_i}{2} \geq N\frac{s}{2}$, implying that $s < 2w$, and thus that the makespan of π is lower than $5w/2$. □

8.5 Complexity

Phase 1 can be implemented in $O(Nn+n\log(n))$. Indeed, we first sort the high jobs in non increasing order of their processing times. Then, each layer can be created in $O(n)$. Phase 2 and phase 3 can also be implemented in $O(Nn+n\log(n))$ by sorting the long (and extra long) jobs in non increasing order of their required processors. Thus π_0 is constructed in $O(Nn + n\log(n))$.

The LS_{π_0} algorithm can be implemented in $O(n\log(n))$. Instead of scanning time by time and organization by organization, this algorithm can be implemented by maintaining a list that contains the set of "currently" scheduled jobs. The list contains 3-tuples (j, t, i) indicating that job j (scheduled on organization i) finishes at time t. Thus, instead of scanning every time from 0 it is sufficient to maintain sorted this list according to the t values (in non decreasing order), and to only consider at every step the first element of the list. Then, it takes $O(\log(n))$ to find a job j_0 in the appropriate shelf that fits at time t, because a shelf can be created as a sorted array. It also takes $O(\log(n))$ to insert the new event corresponding to the end of j_0 in the list.

The last step, which turns π_1 into the final schedule can also be implemented in $O(n\log(n))$ using a similar global list of events. Notice that for any organization O_i,

there exists a t_i such that before t_i the utilization is an arbitrary function strictly larger than $m/2$, and after t_i a non increasing after. Scheduling a small job before t_i would require additional data structure to handle the complex shape. Thus we do not schedule any small job before t_i as it is not necessary for achieving the $5/2$ ratio. Therefore, we only add those events that happen after t_i when initializing the global list for this step. To summarize, for this step we only need to sort the small jobs in non increasing order of their required number of processors, and then apply the same global list algorithm.

The binary search on w to find the smallest w which is not rejected can be done in $O(\log(np_{max}))$ as all the processing times can be assumed to be integers. Thus the overall complexity of the $5w/2$ approximation is in $O(\log(np_{max})n(N + \log(n)))$.

8.6 Toward better approximation ratios

In this paper we provided a low cost $5/2$-approximation algorithm using a new approach. We discuss in this section how the proposed approach can be used for reaching better approximation bounds. The approach can be summarized in the two following main steps. The first one consists in constructing a $1/2$ compact schedule π_1 of the big jobs I' by creating a pre-allocation π_0 and "compressing" it. Then, the remaining small jobs $(I \setminus I')$ are added to π_1 in a second step using the classical list scheduling algorithm LS.

We would like to recall the arguments that make our second step easy to analyze, and see what could be some other promising partitions. In our partition, the second step guaranties a makespan lower than $5w/2$ because :

- adding a job j with $q_j \leq m/2$ to a $1/2$ compact schedule with LS produces another $1/2$ compact schedule,
- if the makespan is due to a small job j_0 that starts at time s_0, then $s_0 \leq 2w$ (since the schedule is $1/2$ compact), leading to a makespan lower than $s_0 + p_{j_0} \leq 5w/2$.

Let us now propose other partitions that could be considered. We could define $I' = \{j | q_j > \alpha m_i \text{ or } p_j > \beta w\}$ with appropriate values $0 < \alpha < 1$ and $0 < \beta < 1$. Then, the previous steps become :

1. construct a pre-allocation π_0 of I' (for instance based on shelves and layers) and make sure that, when compressed using LS_{π_0}, the obtained schedule π_1 is $1 - \alpha$ compact (meaning that for every organization, the utilization is greater than $1 - \alpha$, and then it is non-increasing),
2. add the small remaining jobs $(I \setminus I')$ using LS.

Thus, the makespan of jobs added in the second step would be bounded by $b = (\frac{1}{1-\alpha} + \beta)w$, implying that the makespan of the pre-allocation should also be bounded by b.

For example, we can target a $7/3$ ratio by only studying how to pre-allocate $I' = \{j | q_j > \frac{m_i}{2} \text{ or } p_j > \frac{w}{3}\}$, or a ratio 2 by studying how to pre-allocate $I' = \{j | q_j > \frac{m_i}{3} \text{ or } p_j > \frac{w}{2}\}$. Obviously, if the preallocation is built using again shelves and layers, the difficulty will probably arise when merging the different types of jobs (high, extra long or long ones for example), and will may be only need to handle more particular cases.

Let us remark that this technique will not be easy to apply with (contiguous) rectangles, since the property of $1/2$ compactness becomes hard to guarantee.

Bibliographie

[1] M. Bougeret, P-F. Dutot, K. Jansen, C. Otte, and D. Trystram. Approximation algorithm for multiple strip packing. In *Proceedings of the 7th Workshop on Approximation and Online Algorithms (WAOA)*, 2009.

[2] M. Bougeret, P-F. Dutot, K. Jansen, C. Otte, and D. Trystram. Approximating the non-contiguous multiple organization packing problem. In *Proceedings of the 6th IFIP International Conference on Theoretical Computer Science (TCS)*, 2010.

[3] EG Coffman Jr, MR Garey, DS Johnson, and RE Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9 :808, 1980.

[4] P-F. Dutot, G. Mounié, and D. Trystram. Scheduling Parallel Tasks : Approximation Algorithms. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis, JY-T. Leung (Eds.)*.

[5] P-F. Dutot, F. Pascual, Rzadca K., and Trystram D. Approximation algorithms for the multi-organization scheduling problem. *Submitted to : IEEE Transactions on Parallel and Distributed Systems (TPDS)*.

[6] MR Garey and R.L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2) :187–200, 1975.

[7] D. Hochbaum and D. Shmoys. A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM Journal on Computing*, 17(3) :539–551, 1988.

[8] K. Jansen and R. Solis-Oba. New approximability results for 2-dimensional packing problems. *Lecture Notes in Computer Science*, 4708 :103, 2007.

[9] C. Kenyon and E. Rémila. A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, pages 645–656, 2000.

[10] I. Schiermeyer. Reverse-fit : A 2-optimal algorithm for packing rectangles. *Lecture Notes in Computer Science*, pages 290–290, 1994.

[11] U. Schwiegelshohn, A. Tchernykh, and R. Yahyapour. Online scheduling in grids. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10, 2008.

[12] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26 :401, 1997.

[13] D. Ye, X. Han, and G. Zhang. On-Line Multiple-Strip Packing. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications (COCOA)*, page 165. Springer, 2009.

[14] SN Zhuk. Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1) :73–85, 2006.

Chapitre 9

Improvements of previous ratios for the non-contiguous case

Marin bougeret[‡], Pierre-François Dutot, and Denis Trystram

LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

[‡]This work is supported by DGA-CNRS

Abstract

In this chapter we improve the results of Chapter 8 on the non contiguous regular Multiple Strip Packing problem (denoted by MSP_r^{nc} in the classification of Chapter 5). Using the principles introduced in Chapter 8 for the $\frac{5}{2}$-approximation algorithm, we derive a $\frac{7}{3}$ ratio for MSP_r^{nc}, and a 2 (optimal) ratio for a special case of MSP_r^{nc} where the width of any rectangle is lower than half of $\frac{1}{2}$. Both algorithms run in $O(log_2(nh_{max})N(n + log(n)))$, where h_{max} is the maxium height of the rectangles.

9.1 Introduction

In this chapter we study the non contiguous regular Multiple Strip Packing (MSP_r^{nc}) problem, a generalization of the well-known Strip Packing problem. For a given set of rectangles $L = \{r_1, \ldots, r_n\}$ having arbitrary heights, and widths lower or equal than 1, the goal is to find a non-overlapping orthogonal packing without rotations into $N \in \mathbb{N}$ strips $[0,1] \times [0,\infty)$, minimizing the maximum height of the strips. We are interested in the non-contiguous version of this problem where a rectangle can be split into several rectangles of same height, that must be allocated in the same strip and at the same level. The main application of this non-contiguous version is the makespan minimization when scheduling parallel jobs on a multi-cluster environment, where the width of the strips corresponds to the number of processors of clusters, the width of a rectangle (job) corresponds to the number of processors required by the job, and the length of a rectangle corresponds to the processing time. In this scheduling context, there is no need to have a constraint that forces to use contiguous indexes of processors.

Related Work. As shown in [Zhuk, 2006] using a gap reduction from the 2 partition problem, this problem is 2-inapproximable in polynomial time unless $\mathcal{P} = \mathcal{NP}$, even for $N = 2$.

The main previous positive results are mainly a (fast) 3-approximation algorithm for MSP_d^{nc} in [Schwiegelshohn et al., 2008], and an $2 + 2\epsilon$-approximation algorithm for $MSP_r^{nc/c}$ obtained in [Ye et al., 2009] using simply as "black boxes" a $PTAS$ for classical sequential job scheduling problem $(P||C_{max})$ with precision ϵ, and the strip packing algorithm of [Steinberg, 1997] on each strip.

In our previous works, we provided in [Bougeret et al., 2009] (see Chapter 7) a costly 2-approximation and an AFPTAS for the $MSP_r^{nc/c}$, and a $5/2$-approximation (running in $O(log_2(nh_{max})N(n + log(n))))$ for MSP_r^{nc} in Chapter 8 (see Chapter 8).

Contributions. In this paper we improve our previous ratios following the ideas suggested in Chapter 8. We obtain a $7/3$-approximation for the MSP_r^{nc}, and a 2-approximation for a restriction of the MSP_r^{nc} where all the rectangles have a width lower than $1/2$. Both algorithms run in $O(log_2(nh_{max})N(n+log(n)))$, where h_{max} is the maximum height of the rectangles.

As this restriction of MSP_r^{nc} is still 2-inapproximable (see Section 9.4) and the proposed algorithm is very fast, this result is somehow the best possible.

Main notations. Given a rectangle r_j, let w_j and h_j be respectively the width and the height of r_j, and let $s(r_j) = w_j h_j$ be the surface of r_j. We extend these notations to $W(X)$, $H(X)$ and $S(X)$ with X a set of rectangles.

9.2 General principles

9.2.1 Preliminaries

A *layer* is a set of rectangles packed one on top of the other in the same strip (as depicted Figure 9.1). The height of a layer Lay is $H(Lay)$, the sum of the height of all

the rectangles in Lay. A *shelf* is a set of rectangles that are packed in the same strip, such as the bottom level of all the rectangles is the same. Even if it is not relevant for the non-contiguous case, we consider for the sake of simplicity that in a shelf, the right side of any rectangle (except the right most one) is adjacent to the left side of the next rectangle in the shelf. Given a shelf sh (sh denotes the set of rectangles in the shelf), the value $W(sh)$ is called the *width* of sh. Packing a shelf at level l means that all the rectangles of the shelf have their bottom at level l. A *bin* is a rectangular area that can be seen as reserved space in a particular strip for packing rectangles. As a bin always has width 1, we define a bin by giving its height h_b, its bottom level l_b and the index i_b of the strip it belongs to. Packing a shelf sh in a bin b means that sh is packed in strip S_{i_b} at level l_b. Moreover we always guarantee that the height of any rectangle of sh is lower than h_b.

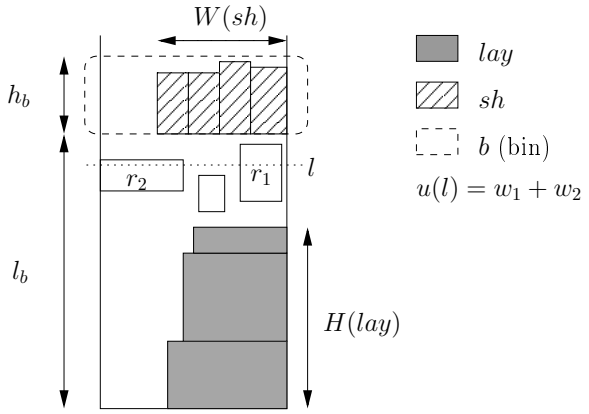

FIG. 9.1: Example of a layer, a shelf, a bin and of the utilization function. sh is packed in b.

The *utilization* $u_i^\pi(l)$ of a packing π in strip S_i at level l (sometimes simply denoted by $u(l)$ or $u_i(l)$) is the sum of the width of all the rectangles packed in S_i that cut the horizontal line-level l (see Figure 9.1). Of course we have $0 \leq u_i^\pi(l) \leq 1$ for any l and i.

Let us now describe three useful procedures. The $CreateLayer(X, h)$ procedure creates a layer Lay (using rectangles of X) of height at most h, using a Best Fit (according to the height) policy (BFH). Thus, $CreateLayer(X, h)$ adds at each step the highest rectangle that fits. Of course, the layer produced by the procedure is such that $H(Lay) \leq h$. Moreover, notice that we will always pack the layers in the strips with **narrowest rectangles on the top**. The $CreateShelf(X, w)$ creates a shelf sh (using rectangles of X) of width at most w, using the Best Fit (according to the width) policy (BFW). Thus, $CreateShelf(X, w)$ adds at each step the widest rectangle that fits. Of course, the shelf produced by the procedure is such that $W(sh) \leq w$. Throughout the paper, we consider that the sets of jobs used as parameters in the algorithms are modified after the calls. Finally, the procedure $GreedyPack(X, seq)$ (where seq is an ordered sequence of bins) creates for each empty bin $b \in seq$ a shelf of rectangles of X using $CreateShelf(X, 1)$ and packs it into b. This procedure returns the last bin in which a shelf has been created, or *null* if no shelf is created. Notice that we will always use sequence of bins that have the same height h_b, and such that $max_{r_x \in X} h_x \leq h_b$.

Let us now state a standard lemma about the efficiency of the "best fit" policies.

Lemma 9.2. *Let Sh denote the shelf created by $CreateShelf(X, w)$. If the k widest rectangles of X are added to sh, then $W(Sh) > \frac{k}{k+1}w$.*

Proof Let x be the cardinality of X. Let us assume that $w_i \geq w_{i+1}$ for $1 \leq i < x$. Let $i_0 \geq k+1$ be the first index such that r_{i_0} is not in Sh. Let $a = \Sigma_{i=1}^{i_0-1} w_i$. We have $W(Sh) \geq a \geq (i_0 - 1)w_{i_0} > (i_0 - 1)(w - a)$ leading to $a > \frac{i_0-1}{i_0}w \geq \frac{k}{k+1}w$. □

Finally, let us recall two useful results that are used to claim that a set of rectangles (of height at most v) *select* of total surface at most $\frac{7v}{6}$ can be packed in one strip with a height at most $\frac{7v}{3}$.

Theroem 9.3 ([Steinberg, 1997]). *Let $L = \{r_1, \ldots, r_n\}$ be a set of rectangles. Let $w_{max} = max_j w_j$ and $h_{max} = max_j h_j$. If $w_{max} \leq u, h_{max} \leq v$ and*

$$2S(L) \leq uv - max(2w_{max} - u, 0)max(2h_{max} - v, 0)$$

then it is possible to pack L (in time $\mathcal{O}(nlog^2(n)/log(log(n)))$ in a rectangular box of width u and height v.

Notice also that in our particular case of non-contiguous packing, we can simply use the Widest First algorithm (that runs in $\mathcal{O}(nlog(n))$) that scans strip from level 0 and packs for every level the widest possible remaining rectangle. Indeed, let us recall the following simple lemma (proved in Chapter 10)

Lemma 9.4. *Let X be a set of rectangles such that $S(X) \leq \alpha v$, with $\alpha \geq 1$. Let v such that for all $j \in X$, $h_j \leq v$. Then, the Widest first algorithm packs X in a strip with a height lower than $2\alpha v$.*

9.2.2 Discarding technique applied to non-contiguous multiple strip packing

Introduction

Discarding techniques are common for solving packing/scheduling problem. As mentioned before, the idea is to define properly a set of "small" items (rectangles here), and to prove that adding these small items only at the end of the algorithm will not degrade the approximation ratio. Thus, the effort can be focused on the remaining "large" items. In this section we use an adaptation of this general technique to the context of non-contiguous multiple strip packing. As usual, the set of big rectangles $I'(\alpha, \beta) \subset I$ depends on parameters (α and β here) that we chose, and the larger the set $I'(\alpha, \beta)$ we can handle, the better the approximation ratio will be (as the remaining small rectangles become really negligible).

Definition

In order to partition rectangles according to their height, we need to use the well-known dual approximation technique [Hochbaum and Shmoys, 1988], and we denote

by v the guess of the optimal value. Given an instance I, let $L_{WD} = \{w_j > \alpha\}$ be the set of wide rectangles, $L_H = \{r_j > \beta v\}$ be the set of high rectangles, and $I' = L_{WD} \cup L_H$ be the set of big rectangles, with $0 < \alpha < 1$ and $0 < \beta < 1$. Let $r(\alpha, \beta) = (\frac{1}{1-\alpha} + \beta)$ be the approximation ratio we target (the origin of this formula will be explained in Lemma 9.10). We also need the following definition.

Définition 9.5. *A packing is x-compact (see Figure 9.6) if and only if for every strip S_i there exists a level l_i such that for all $l \leq l_i, u_i(l) > x$ and u_i restricted to $l > l_i$ is non-increasing.*

Let us now describe the three main steps of our approach. Notice that what we call a preallocation is a "normal" packing (*i.e.* that define the bottom level of each rectangle, which is sufficient) that is based on simple structures like shelves and layers.

a) construct a preallocation π_0 of I' that fits in $r(\alpha, \beta)v$
b) turn π_0 into a $(1-\alpha)$-compact packing π_1 (of height lower than $height(\pi_0)$) using the list algorithm LS_{π_0} (see Lemma 9.8)
c) add the small remaining rectangles $(I \setminus I')$ using LS (see Lemma 9.10)

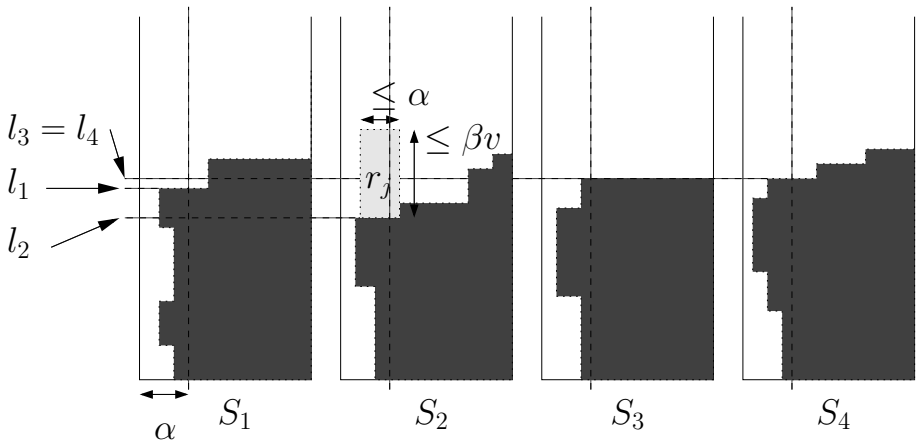

FIG. 9.6: Example showing why step c) is simple : adding as soon as possible a small rectangle r_j (having $h_j \leq \beta v$ and $w_j \leq \alpha$) to a $(1-\alpha)$ packing cannot exceed $v(\frac{1}{1-\alpha}+\beta)$. The l_i values are defined according to Definition 9.5.

Step a) is the most difficult one. Thus, Sections 9.3 and 9.4 are entirely devoted to the construction of π_0 (for (α, β) equal to $(\frac{1}{2}, \frac{1}{3})$ and $(\frac{1}{3}, \frac{1}{2})$, respectively). Of course, building the preallocation becomes harder when α and β are small, as the number of rectangle of I' increases and $r(\alpha, \beta)$ decreases. Roughly speaking, the simple shapes of rectangles of I' allows us to construct π_0 with a simple structure. We will denote by π_0^i the set of rectangles packed by π_0 in S_i.

Details

We now prove that applying steps b) and c) leads to a $r(\alpha, \beta)$ ratio. We start by studying how to compact the preallocation. Given π_0, we have to repack $I' = L_{WD} \cup L_H$ in a $(1-\alpha)$-compact packing.

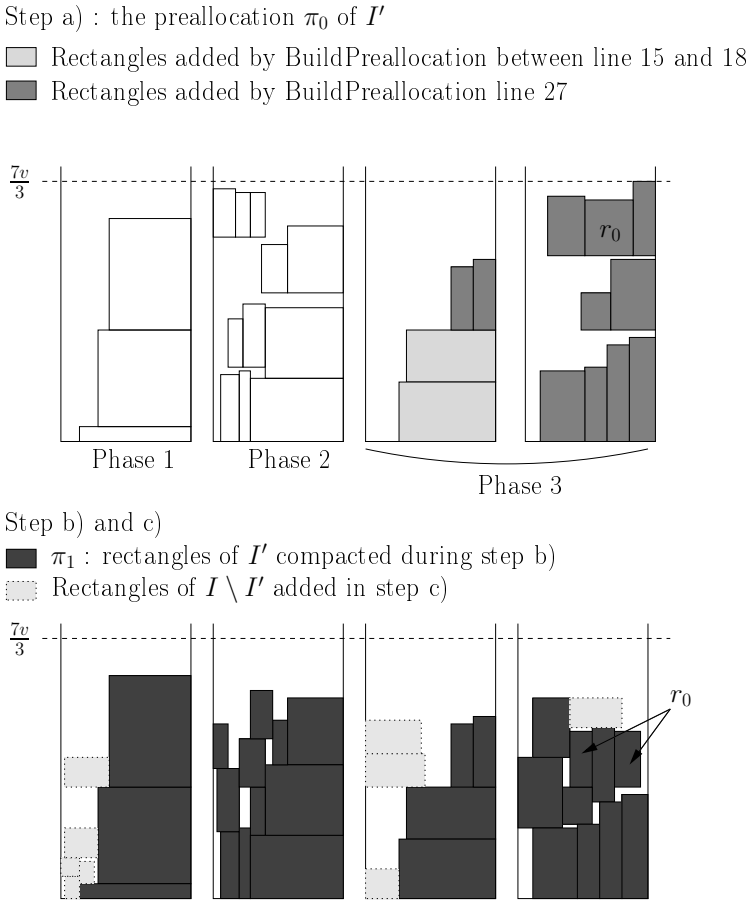

FIG. 9.7: Example of the overall algorithm with $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{3}$. Notice that during phase b), r_0 is packed in a non continuous way.

The packing of L_{WD} is straightforward, as we only pack rectangles of L_{WD} as their were preallocated. Remind that we only use an α equal to $\frac{1}{2}$ or $\frac{1}{3}$, and as the preallocation of L_{WD} is simply based on layers starting at level 0. More precisely, with $\alpha = \frac{1}{2}$ (when wide rectangles have a width strictly larger than half of the strip) rectangles of L_{WD} are preallocated in layers (with the narrowest rectangles on the top) from level 0, which is $\frac{1}{2}$-compact. The case where $\alpha = \frac{1}{3}$, used in Section 9.4 for the restriction of MSP_r^{nc} where $w_j \leq \frac{1}{2}$ for any j, is also simple. In this case our preallocation packs one or two layers (again with the narrowest rectangles on the top) of rectangles of L_{WD} in the same strip in "parallel", which is $\frac{2}{3}$-compact.

We now study in Lemma 9.8 how to repack $I' \setminus L_{WD} \subset L_H$.

Lemma 9.8 (Step b)). *Let π_0 be the preallocation of I' constructed in Step a). Let $\pi'_0 = \pi_0 \cap L_{WD}$ denote π_0 when keeping only rectangles that of L_{WD}. As described below, we already have a $(1-\alpha)$-compact packing $\widehat{\pi_1}$ of rectangles of L_{WD} such that for any strip S_i and level l we have*

$$u_i^{\widehat{\pi_1}}(l) = u_i^{\pi'_0}(l)$$

Then, we can complete $\widehat{\pi_1}$ into a $(1-\alpha)$-compact packing π_1 of I', such that the height of π_1 is lower or equal to the height of π_0.

Proof Let us define the LS_{π_0} algorithm that adds rectangles of $I' \setminus L_{WD}$. Let us consider a single strip S_i. Let π_0^i denote π_0 restricted to S_i, and $\widehat{\pi_1^i}$ denote $\widehat{\pi_1}$ restricted to S_i. Let $X = \{r_1, \ldots, r_p\}$ be the set of preallocated rectangles of $I' \setminus L_{WD}$ that we have to add to S_i. We assume that $lvl(j) \leq lvl(j+1)$, where $lvl(j)$ is the bottom level of r_j in π_0.

For our considered strip S_i, the LS_{π_0} algorithm executes $AddAsap(r_j, \widehat{\pi_1^i})$, for $1 \leq j \leq p$, where $AddAsap(r, \widehat{\pi_1^i})$ adds rectangle r to $\widehat{\pi_1}$ (in S_i) at the smallest possible level. Notice first that adding with $AddAsap$ a rectangle r_j with $w_j \leq \alpha$ to a $(1-\alpha)$-compact packing creates another $(1-\alpha)$-compact packing. Thus it is clear that π_1 is $(1-\alpha)$-compact.

For any $1 \leq j \leq p$, let $(\widehat{\pi_1^i}, j)$ denote the packing in S_i just before adding r_j with $AddAsap$, and let $(\pi_0^{i'}, j)$ denote the packing $\pi_0^i \cap (L_{WD} \cup \{r_1, \ldots, r_{j-1}\})$. Let us prove by induction on $j \in \{1, \ldots, p\}$ that $u^{(\widehat{\pi_1^i}, j)}(l) \leq u^{(\pi_0^{i'}, j)}(l)$, for any $l \geq lvl(j)$. The hypothesis of the Lemma gives the property for $j = 1$. Let us suppose that the property is true for j, and prove it for $j+1$. Let $l \geq lvl(j+1)$. The induction property for rank j implies that r_j is added by $AddAsap$ at a level lower or equal to $lvl(j)$. Thus, if r_j intersects l in $(\widehat{\pi_1^i}, j+1)$, then it also occurs in $(\pi_0^{i'}, j+1)$. Thus in this case we have

$$\begin{aligned} u^{(\widehat{\pi_1^i}, j+1)}(l) &= u^{(\widehat{\pi_1^i}, j)}(l) + w_j \\ &\leq u^{(\pi_0^{i'}, j)}(l) + w_j \\ &= u^{(\pi_0^{i'}, j+1)}(l) \end{aligned}$$

If r_j does not intersect l in $(\widehat{\pi_1^i}, j)$, then clearly $u^{(\widehat{\pi_1^i}, j+1)}(l) = u^{(\widehat{\pi_1^i}, j)}(l) \leq u^{(\pi_0^{i'}, j)}(l) \leq u^{(\pi_0^{i'}, j+1)}(l)$

Thus we proved that for any $1 \leq j \leq p$ we have $u^{(\widehat{\pi_1^i}, j)}(l) \leq u^{(\pi_0^{i'}, j)}(l)$ for any $l \geq lvl(j)$, implying that every r_j is added by $AddAsap$ at a level lower or equal to $lvl(j)$. Thus, the height of π_1 is lower or equal to the height of π_0

□

Remark 9.9. *Notice that in this proof we only need (before adding rectangles of $I' \setminus L_{WD}$) that $u_i^{\widehat{\pi_1}}(l) \leq u_i^{\pi_0'}(l)$, for any $l \geq l_i^{min}$, where l_i^{min} is the smallest bottom level of a rectangle of $I' \setminus L_{WD}$ preallocated in π_0 in strip S_i.*

We now prove in Lemma 9.10 that after adding rectangles in step c), the height of the packing do not exceed $r(\alpha, \beta)v = (\frac{1}{1-\alpha} + \beta)v$. This explains why the height of the pre-allocation should also be bounded by $r(\alpha, \beta)v$.

Lemma 9.10 (Step c)). *Let π_1 be a $(1-\alpha)$-compact packing of I'. Adding to π_1 rectangles of $I \setminus I'$ with a List Scheduling algorithm (LS) leads to a packing π having height lower than $max(height(\pi_1), v(\frac{1}{1-\alpha} + \beta))$.*

Proof The LS algorithm scans all the strips from level 0, and at any level adds any rectangle of $I \setminus I'$ that fits. Notice that the final packing π is $(1-\alpha)$-compact, since we add rectangles r_j with $w_j \leq \alpha$.

Let us assume that the height of π is due to a rectangle $r_j \in I \setminus I'$ that starts at level s. This implies that when packing r_j we had $l_i \geq s$ for any strip i (with l_i defined as in Definition 9.5). We recall that we have according to this definition $u_i(l) > 1 - \alpha$ for any $l \leq l_i$. Thus, we have $S(I) > \sum_{i=1}^{N} l_i(1-\alpha) \geq N(1-\alpha)s$, implying that $s < v\frac{1}{1-\alpha}$, and thus that of height of π is lower or equal to $s + max_{j \in I \setminus I'} h_j \leq v(\frac{1}{1-\alpha} + \beta)$. □

Thus, we target in Section 9.3 a $\frac{7}{3}$ ratio by choosing $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{3}$, and a 2 ratio (for a particular case) in Section 9.4 by choosing $\alpha = \frac{1}{3}$ and $\beta = \frac{1}{2}$.

9.3 A $\frac{7}{3}$-approximation

9.3.1 Definition of the considered partition

Conforming to the idea presented in Section 9.2.2, we define several special sets of rectangles. We do not only define the set of high rectangles as $\{r_j | h_j > v/3\}$ as for example we have to treat differently "extra high" rectangles having height larger than $2v/3$ and "medium" rectangles having height between $v/3$ and $v/2$.

- let $L_{WD} = \{r_j | w_j > 1/2\}$ be the set of wide rectangles
- let $L_{XH} = \{r_j | h_j > 2v/3\}$ be the set of extra high rectangles
- let $L_H = \{r_j | 2v/3 \geq h_j > v/2\}$ be the set of high rectangles
- let $L_M = \{r_j | v/2 \geq h_j > v/3\}$ be the set of medium rectangles
- let $L_B = L_{WD} \cap (L_{XH} \cup L_H \cup L_M)$ be the set of huge rectangles
- let $I' = L_{WD} \cup L_{XH} \cup L_H \cup L_M$
- let $\bar{L_{XH}} = L_{XH} \setminus L_{WD}$, and let $\bar{L_H}$ and $\bar{L_M}$ be defined in the same way

According to our framework, it is sufficient to provide a $\frac{1}{2}$-compact preallocation for rectangles of I'. Conforming to the dual approximation technique, we will prove that either we pack I with a resulting height lower than $7v/3$, or $v < Opt$. Notice that for the sake of simplicity we did not add the "reject" instructions in the algorithm. Thus we consider in all the proof that $v \geq Opt$, and it is implicit that if one of the claimed properties is wrong during the execution, the considered v should be rejected.

9.3.2 Counting the width of packed rectangles

We start by giving a bound on the total width of extra high, high and medium rectangles.

Lemma 9.11. *If $v \geq Opt$, then*

- $W(L_{XH}) + W(L_H) \leq N$
- $2W(L_{XH}) + W(L_H) + W(L_M) \leq 2N$.

Proof Let us suppose that I can be packed in v. Let us consider an arbitrary packing of height at most v in a strip S_i. What we call "abscissa l", with $l \in [0, 1]$, is the infinite vertical slice of the strip located at l, considering that the left part of the strip is at abscissa 0 and the right part at abscissa 1. If we were using scheduling vocabulary, abscissa l would correspond to the "activity" of processor l, with $l \in \{1, \ldots, m\}$ (strip would have m processors instead of having a width 1).

Let x_{XH} be the number of rectangles packed in S_i that cut abscissa l. Again, in scheduling vocabulary x_{XH} would be the number of jobs of L_{XH} that are processed by processor l. We define x_H and x_M in the same way. Let h be the total height of rectangles packed in S_i at abscissa l. By our assumption we have $h \leq v$. Moreover, we have $h > x_{XH}\frac{2v}{3} + x_H\frac{v}{2} + x_M\frac{v}{3}$. This implies $4x_{XH} + 3x_H + 2x_M < 6$, and since the left member of the inequality is an integer we even get $4x_{XH} + 3x_H + 2x_M \leq 5$. From this we deduce $2x_{XH} + x_H + x_M \leq \frac{5}{2}$ and $x_{XH} + x_H \leq \frac{5}{3}$. Again, as the left member of each inequality is an integer, we get $2x_{XH} + x_H + x_M \leq 2$, and $x_{XH} + x_H \leq 1$. Then, by summing over all the abscissas and strips we get the desired result. □

We will sometimes use the bounds of Lemma 9.11 to prove that I' must be packed by counting the total width of extra high, high and medium rectangles packed by the algorithm. Given a packing π_i (of one strip), let us define functions f^i such that $f^1(\pi_i) = W(\pi_i \cap L_{XH})$, $f^2(\pi_i) = W(\pi_i \cap L_H)$ and $f^3(\pi_i) = W(\pi_i \cap L_M)$. We can now define the notion of dominating packing.

Définition 9.12. *A packing in one strip (or simply a set) π_i is dominating iff $2f^1(\pi_i) + f^2(\pi_i) + f^3(\pi_i) > 2$. A packing $\pi = (\pi_1, \ldots, \pi_x)$ of x strips is dominating iff all the π_i are dominating.*

Remark 9.13. *This notion of domination must not be confused with area domination. We will say a packing π_i is "area-dominating" iff $S(\pi_i) > v$.*

We now state Lemma 9.14 that shows why dominating packing are interesting.

Lemma 9.14. *Let π be a dominating packing of I' in x strips. Then $x < N$.*

Proof We have $2N \geq 2W(L_{XH}) + W(L_H) + W(L_M) = 2\sum_{i=1}^{x} W(L_{XH} \cap \pi_i) + \sum_{i=1}^{x} W(L_H \cap \pi_i) + \sum_{i=1}^{x} W(L_m \cap \pi_i) = \sum_{i=1}^{x}(2f^1(\pi_i) + f^2(\pi_i) + f^3(\pi_i)) > 2x$. □

Finally, let us show how to create a dominating packing in a free strip.

Lemma 9.15. *If we do not run out of rectangles, it is always possible to create a dominating packing in a free strip, using rectangles of $I' \setminus L_{WD}$.*

Proof Let us consider a strip i. If $\bar{L_{XH}}$ is such that $W(\bar{L_{XH}}) > 1$, then we just create two shelves sh_1 and sh_2 (such that $W(sh_1) + W(sh_2) > 1$) using rectangles of $\bar{L_{XH}}$, that we pack at level 0 and v. This packing is dominating.

Otherwise (see Figure 9.16), we pack $\bar{L_{XH}}$ in one shelf sh_{XH} at level 0. Then, we create a shelf sh_1 by adding greedily (in any order) remaining rectangles of $I' \setminus L_{WD}$ at level 0, until a rectangle r_1 does not fit. We pack r_1 at level v. Notice that we have $W(sh_1) + w_1 > 1 - W(\bar{L_{XH}})$. As $r_1 \notin \bar{L_{XH}}$ we know that the top of r_1 is at level at most $\frac{5v}{3}$.

Then, we create a shelf sh_2 by adding greedily (in any order) remaining rectangles of $I' \setminus L_{WD}$ (that belong to $\bar{L_H} \cup \bar{L_M}$) at level $\frac{5v}{3}$, until a rectangle r_2 does not fit. Finally, r_2 is packed at level v (r_1 and r_2 both fit at level v as we consider rectangles of $I' \setminus L_{WD}$). Notice that we have $W(sh_2) + w_2 > 1$.

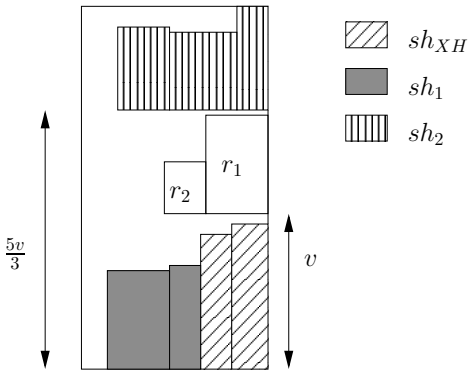

FIG. 9.16: Example of a dominating packing built in Lemma 9.15.

The packing is dominating since

$$\begin{aligned} 2f^1(\pi_i) + f^2(\pi_i) + f^3(\pi_i) &\geq 2W(\bar{L_{XH}}) + W(sh_1) + w_1 \\ &\quad + W(sh_2) + w_2 \\ &> 2 + W(\bar{L_{XH}}) \end{aligned}$$

□

9.3.3 Description of the algorithm

We now describe the algorithm that builds the preallocation π_0 of the rectangles of I'. Remind that π_0^i denotes the set of rectangles packed in S_i. The description of the *BuildPreallocation* is in Algorithm 2, Page 126, and uses several Lemmas that are detailed later. Notice that there is one special case in phase 3 where we pack all the rectangles of I, and not only the one in I'.

To prove that Algorithm 2 packs I' (or even sometimes I), we will either use area-domination (*i.e.* area argument) or domination (*i.e.* counting argument). Thus, the key ideas of this algorithm are the following.

- A packing of a strip must be area-dominating or dominating.
- Packing created in phase 1 are dominating, and sometimes area-dominating.
- Packing created in phase 2 are dominating and always area-dominating.
- The condition $|L_{WD} \cap L_H| < 2$ in phase 3 is important, as only using highest first order (when calling the procedure *CreateLayer*) on L_{WD} could produce packing that are neither area-dominating nor dominating (if for example the only remaining rectangles of $L_{WD} \cap L_H$ have width $\frac{1}{2} + \epsilon$ and height $\frac{2v}{3} - 2\epsilon$, only three such rectangles could be packed in a strip). Thus the special case where a lot of rectangles of $L_{WD} \cap L_H$ remain after phase 2 is handled easily as it implies (due to our "while condition" in phase 2) that $W(\bar{L_{XH}}) \leq 1$ and $W(\bar{L_H} \cup \bar{L_M}) \leq \frac{3}{2}$.

Let us now prove the feasibility of the different phases of Algorithm 2. Notice that it is obvious that phase 1 stops, and that phase 1 uses at most $\lfloor \frac{N}{2} \rfloor$ strips as $|L_{XH} \cap L_{WD}| \leq N$.

Algorithm 2 BuildPreallocation

1: $i \leftarrow 0$
2: ——————————- phase 1 ——————————-
3: **while** $|L_{WD} \cap L_{XH}| \geq 2$ **do**
4: $i \leftarrow i + 1$
5: $Lay_i = CreateLayer(L_{WD}, 7v/3)$
6: Pack Lay_i in S_i with the narrowest rectangles on the top
7: **end while**
8: ——————————- phase 2 ——————————-
9: **while** $(|L_{WD} \cap L_H| \geq 2)$ and $(W(\bar{L_{XH}}) > 1$ or $W(\bar{L_H} \cup \bar{L_M}) > \frac{3}{2})$ **do**
10: $i \leftarrow i + 1$
11: Create a packing π_i in S_i such that π_i is dominating (see Definition 9.12) and $S(\pi_i) > v$ (See Lemma 9.17)
12: **end while**
13: ——————————- phase 3 ——————————-
14: $few\_high \leftarrow (|L_{WD} \cap L_H| < 2)$
15: **while** L_{WD} is not empty **do**
16: $i \leftarrow i + 1$
17: $Lay_i = CreateLayer(L_{WD}, 7v/3)$
18: Pack Lay_i in S_i with the narrowest rectangles on the top
19: **end while**
20: **if** $few\_high = true$ **then**
21: **if** $(S(\pi_x) \geq v$ for any $x \leq i - 1)$ **then** //in this case we even pack I
22: **for** $l = i + 1$ to N **do**
23: pack in S_l an area of rectangles of I greater than v (if there is enough remaining rectangles) using Lemma 9.19
24: **end for**
25: pack in S_i all the remaining rectangles of I and the rectangles already contained in S_i using Steinberg's (or Widest First) algorithm (see Lemma 9.21).
26: **else**
27: pack all the remaining rectangles of I' using Lemma 9.22
28: **end if**
29: **else** $//W(\bar{L_{XH}}) \leq 1$ and $W(\bar{L_H} \cup \bar{L_M}) \leq \frac{3}{2}$
30: pack all the remaining rectangles of I' using Lemma 9.24
31: **end if**

9.3.4 Feasibility of phase 2

Lemma 9.17 (Feasibility of Line 11). *Let us suppose that* ($|L_{WD} \cap L_H| \geq 2$) *and* (($W(\bar{L_{XH}}) > 1$ *or* ($W(\bar{L_H} \cup \bar{L_M}) > \frac{3}{2}$)). *Then, it is possible to create a packing* π_i *such that* π_i *is dominating and* $S(\pi_i) > v$.

Proof Let r_1 and r_2 in $L_{WD} \cap L_H$, with $w_1 \geq w_2$. We pack r_1 and r_2 right justified, with r_1 at level 0 and r_2 at level h_1. Notice that $v < h_1 + h_2 \leq \frac{4v}{3}$.

Let us proceed by case analysis, and first suppose that $W(\bar{L_{XH}}) > 1$. In this case we create a shelf sh_1 using $CreateShelf(\bar{L_{XH}}, 1)$, and we pack it at level $\frac{4v}{3}$. Then, we try to pack a rectangle $r_x \in \bar{L_{XH}} \backslash sh_1$ at level 0. Notice that $W(sh_1) \geq 2w_x$ as $CreateShelf$ uses the Widest First order. If r_x fits then $S(\pi_i) > (h_1 + h_2)\frac{1}{2} + \frac{2v}{3}(W(sh_1) + w_x) > \frac{v}{2} + \frac{2v}{3} > v$. Moreover, $2f^1(\pi_i) + f^2(\pi_i) + f^3(\pi_i) \geq 2f^1(\pi_i) > 2$. If r_x does not fit, then we get $W(sh_1) \geq max(2w_x, 1 - w_x)$. Then, we get

$$\begin{aligned}
S(\pi_i) &> h_1(1-w_x) + h_2\frac{1}{2} + \frac{2v}{3}W(sh_1) \\
&> h_1(1-w_x) + (v-h_1)\frac{1}{2} + \\
&\quad + \frac{2v}{3}max(2w_x, 1-w_x) \\
&= h_1(\frac{1}{2} - w_x) + \frac{v}{2} + \frac{2v}{3}max(2w_x, 1-w_x) \\
&> \frac{3v}{4} + max(\frac{5w_xv}{6}, \frac{2v}{3} - \frac{7w_xv}{6}) \\
\overset{w_x=\frac{1}{3}}{>} &\ \frac{3v}{4} + \frac{5v}{18} > v
\end{aligned}$$

Moreover, $2f^1(\pi_i) + f^2(\pi_i) + f^3(\pi_i) \geq 2W(sh_1) + w_1 + w_2 > 2W(sh_1) + 1$, and as $W(sh_1) > \frac{2}{3}$ we get $2W(sh_1) + 1 > 2$.

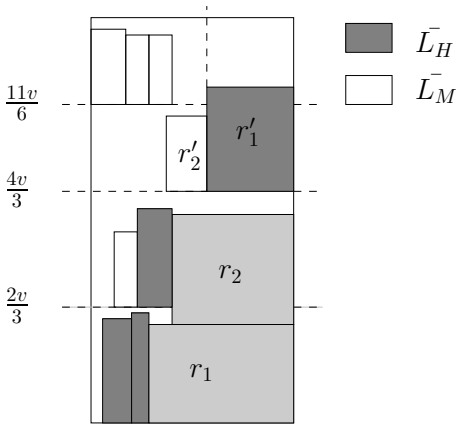

FIG. 9.18: Example of packing built in phase 2 when $W(\bar{L_H} \cup \bar{L_M}) > \frac{3}{2}$.

Let us now suppose that $W(\bar{L_H} \cup \bar{L_M}) > \frac{3}{2}$ (as depicted Figure 9.18). In this case we create at level 0 (using rectangles of $\bar{L_H} \cup \bar{L_M}$) a shelf sh_1, using a **highest first**

order. Let r'_1 be the first rectangle that does not fit. We pack r'_1 right justified at level $\frac{4v}{3}$. Then we create at level $\frac{2v}{3}$ a second shelf sh_2 using again a **highest first** order. Let r'_2 be the first rectangle that does not fit. We pack r'_2 right justified at level $\frac{4v}{3}$. At this stage the packing is already dominating as $f^2(\pi_i) + f^3(\pi_i) > 2$.

We now prove that we can get an area dominating packing. Notice first that if we pack all rectangles $\bar{L_H} \cup \bar{L_M}$, then $S(\pi_i) > \frac{v}{2} + \frac{3}{2}\frac{v}{3} = v$.

If $r'_2 \in L_H$, then $S(\pi_i) > 2\frac{v}{2} = v$. If $r'_2 \in L_M$ and $r'_1 \in L_H$, then all the remaining rectangles are in $\bar{L_M}$. Thus, we create sh_3 using $CreateShelf(\bar{L_M}, 1 - w'_1)$, and we pack sh_3 left justified at level $\frac{11v}{6}$. Notice that we pack sh_3 left justified as we cannot stack r_1, r_2, r'_1 and sh_3. Moreover, as we suppose that we do not pack all rectangles of $\bar{L_H} \cup \bar{L_M}$, we get $W(sh_3) > \frac{1-w'_1}{2} > \frac{1}{4}$. In this case we get

$$\begin{aligned} S(\pi_i) &> (w_1 + W(sh_1) + w'_1)\frac{v}{2} + w_2\frac{v}{2} \\ &\quad +(W(sh_2) + w'_2)\frac{v}{3} + W(sh_3)\frac{v}{3} \\ &> \frac{v}{2} + w_2\frac{v}{2} + (1 - w_2)\frac{v}{3} + \frac{v}{12} \end{aligned}$$

which leads for $w_2 = \frac{1}{2}$ to v. If r'_2 and r'_1 are in L_M, we create sh_3 using $CreateShelf(\bar{L_M}, 1)$, and we pack sh_3 right justified at level $\frac{11v}{6}$. Thus, if all the rectangles are not packed, we have $W(sh_3) > \frac{2}{3}$. Then, we get $S(\pi_i) > (w_1 + w_2)\frac{v}{2} + (1 - w_1 + 1 - w_2)\frac{v}{3} + \frac{2}{3}\frac{v}{3}$ which leads for $w_1 = w_2 = \frac{1}{2}$ to $S(\pi_i) > v$.

□

9.3.5 Feasibility of phase 3

We start by proving Lemma 9.19 that shows how to pack a area larger than v in a free strip.

Lemma 9.19 (Feasibility of Line 23)**.** *It is always possible (if we do not run out of rectangles) to pack an area of rectangles of $I \setminus L_{WD}$ greater than v, in an empty strip S_l and with a height at most $\frac{7v}{3}$.*

Proof Let *select* be an empty set. We add to *select* some rectangles (in non increasing order of their surface) until $S(select) \geq v$ or $I \setminus L_{WD}$ is empty. Let $select = \{r_1, \ldots, r_p\}$, with $S(r_j) \geq S(r_{j+1})$. If $S(select) \leq \frac{7}{6}v$, we know according to Steinberg algorithm (or simply using the Widest First algorithm, see Section 9.2.1) that *select* can be packed with a height at most $2^7/_6 v$.

Let us now suppose that $S(select) > \frac{7}{6}v$ (see Figure 9.20). This implies that the last rectangle r_p added to *select* has a surface strictly larger $\frac{v}{6}$ (otherwise the algorithm would have stopped before), and thus $S(r_j) > \frac{v}{6}$ for all j. Moreover, we get $p \leq 6$, $h_j > \frac{v}{3}$ (as $r_j \notin L_{WD}$), and $w_j > \frac{1}{6}$ for all $j \in \{1, \ldots, p\}$.

Notice that for $p \leq 4$ the lemma is straightforward as two shelves are sufficient to pack *select*. Moreover, if four rectangles of *select* fit in one shelf, then the lemma is also proved as there is at most two remaining rectangles that fit in a second shelf. Thus, we

consider now that $p \geq 5$, and we re-sort rectangles according to their width, implying now $w_j \geq w_{j+1}$.

If $w_1 + w_2 + w_3 \leq 1$ then $w_4 + w_5 + w_6$ is also lower than 1 and two shelves are sufficient.

Let us first consider the case where $w_1 + w_2 + w_3 > 1$ and $w_3 + w_4 + w_5 \leq 1$. In this case it is possible to pack $select \setminus r_6$ in two shelves sh_1 and sh_2, with $sh_1 = \{r_1, r_2\}$ and $sh_2 = \{r_3, r_4, r_5\}$. Then, we pack sh_1 at level 0 and the rectangles of sh_2 top right justified, such that the highest rectangles are on the right side (see Figure 9.18). Then, we pack r_6 right justified at level $l_6 := \min\{l | w_6$ processors are idle in $S_l\} = min(h_1, h_2)$. Let r_{j_0} be the shortest (with the smaller h_j) rectangle of sh_2. If r_6 intersects sh_2, it implies that r_6 intersects r_{j_0}.Thus, with $\gamma = S(select \setminus \{r_6\})$ we have :

$$\begin{aligned}
\gamma &\geq l_6(1 - w_6) \\
&\quad +(\frac{7}{3}v - l_6 - h_6)(W(sh_2) - (1 - w_6) + (1 - w_6)) \\
&> l_6(1 - w_6) \\
&\quad +(\frac{4}{3}v - l_6)(1 - w_6) + \frac{v}{3}(W(sh_2) - (1 - w_6)) \\
&> \frac{4v}{3}(1 - w_6) + \frac{v(4w_6 - 1)}{3} \qquad \text{as } W(sh_2) > 3w_6 \\
&= v
\end{aligned}$$

which is a contradiction. Thus in this case r_6 must fit.

Let us now consider the case where $w_1 + w_2 + w_3 > 1$ and $w_3 + w_4 + w_5 > 1$. Notice that we have then :

1. $w_j > \frac{1}{3}$ for $1 \leq j \leq 3$
2. $w_3 + w_4 > \frac{2}{3}$
3. $h_{min} \leq \frac{3v}{5}$ with $h_{min} = Min_{j \in \{1,\ldots,5\}}$
4. $\sum_{j=1}^{4} h_j < 3v$

The first inequality is true as for $1 \leq j \leq 3$, $3w_j \geq w_3 + w_4 + w_5 > 1$. If the second one were false we would have $2w_4 \leq w_3 + w_4 \leq \frac{2}{3}$ and thus $w_3 + w_4 + w_5 \leq w_3 + 2w_4 \leq 1$. The third one is true since $v > S(\cup_{j=1}^{5} r_j) \geq \sum_{j=1}^{5} w_j h_{min} > \frac{5}{3} h_{min}$. The last one is true since

$$\begin{aligned}
v &> S(\cup_{j=1}^{4} r_j) \\
&\geq (h_1 + h_2 + h_3)w_3 + h_4 w_4 \\
&= (h_1 + h_2 + h_3 - h_4)w_3 + h_4 w_3 + h_4 w_4 \\
&> (h_1 + h_2 + h_3 - h_4)\frac{1}{3} + \frac{2}{3}h_4 \\
&= \frac{1}{3}(h_1 + h_2 + h_3 + h_4)
\end{aligned}$$

Let us start supposing that $p = 5$. In this case we create two layers Lay_i using $CreateLayer(\{r_1, \ldots, r_5\}, \frac{7v}{3})$, that are packed at level 0. Then we repack this layers such that the narrowest rectangles are on the top, implying that the utilization u

function of the strip is decreasing. According to inequality 4 we get $\sum_{j=1}^{5} h_j < 4v$. As $h_{min} \leq \frac{3v}{5}$, we get $H(Lay_1) \geq \frac{7v}{3} - \frac{3v}{5} > \frac{5v}{3}$ and thus $H(Lay_2) \leq 4v - \frac{5v}{3} \leq \frac{7v}{3}$.

It remains now to handle the case where $p = 6$. Notice that if $w_6 > \frac{1}{3}$ then $\sum_{j=1}^{6} h_j < 4v$ (as $v > S(\cup_{j=1}^{5} r_j) > \frac{1}{3}\sum_{j=1}^{5} h_j$) and as before the two layers are sufficient. Thus, we consider that $w_6 \leq \frac{1}{3}$. For $p = 6$ we have $h_1 + h_2 + h_3 \leq 2v$, since $v > S(\cup_{j=1}^{5} r_j) > \frac{1}{3}(h_1 + h_2 + h_3) + S(r_4) + S(r_5) > \frac{1}{3}(h_1 + h_2 + h_3) + 2\frac{v}{6}$, implying $min_{1 \leq j \leq 3} h_i \leq \frac{2v}{3}$ (as $h_1 + h_2 + h_3 \leq 2v$).

If $w_6 \leq \frac{1}{4}$, then we use the same algorithm as for $p = 5$, and then we add r_6 at level $\frac{4v}{3}$. Rectangle r_6 must fit, otherwise $u(\frac{4v}{3}) > \frac{3}{4}$ and thus $S(\cup_{j=1}^{5} r_j) > u(\frac{4v}{3})\frac{4v}{3} > v$. Thus, we consider now that $\frac{1}{3} \geq w_6 > \frac{1}{4}$.

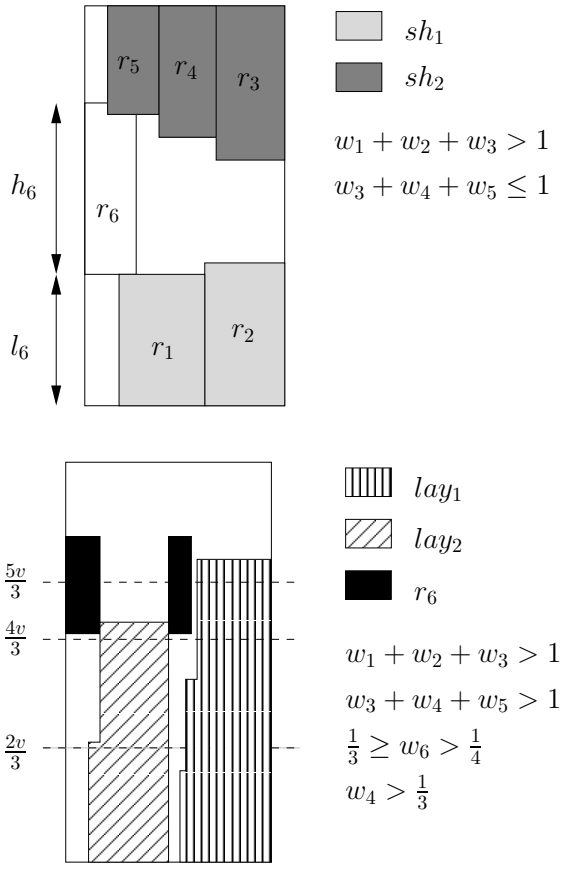

FIG. 9.20: Example of two possible packing built in Lemma 9.19.

If $w_4 > \frac{1}{3}$ (see Figure 9.20), we create (at level 0) a layer lay_1 using $CreateLayer(\{r_1, \ldots, r_4\}, \frac{7v}{3})$ and a layer lay_2 containing all the remaining rectangle except r_6. We repack this layers such that the narrowest rectangles are on the top, implying that the utilization u function of the strip is decreasing. Then, we add r_6 at level $\frac{4v}{3}$. If $\{r_1, \ldots, r_4\}$ all fit in lay_1 then it is clear that r_6 fits in $\frac{7v}{3}$ (as $lay_2 = \{r_5\}$).

Otherwise, let us first see why lay_2 fits :

$$\begin{aligned} H(lay_2) &\leq \sum_{j=1}^{5} h_j - H(lay_1) \\ &\leq 4v - H(lay_1) && \text{according to 4)} \\ &< 4v - \frac{5v}{3} && \text{as } min_{1\leq j\leq 3} h_i \leq \frac{2v}{3} \end{aligned}$$

If r_6 does not fit, we have $u(\frac{4v}{3}) > \frac{2}{3}$, and thus

$$\begin{aligned} S(\cup_{j=1}^{5} r_j) &> u(\frac{4v}{3})\frac{4v}{3} + (min_{1\leq j\leq 4} w_j)(H(lay_1) - \frac{4v}{3}) \\ &> \frac{8v}{9} + \frac{1}{3}\frac{v}{3} \\ &= v \end{aligned}$$

Finally, if $w_4 \leq \frac{1}{3}$ (implying $\sum_{j=4}^{6} w_j \leq 1$), we create one shelf at level 0 with $\{r_4, r_5, r_6\}$, and two layers Lay_1 and Lay_2 using $CreateLayer(\{r_1, r_2, r_3\}, \frac{4v}{3})$, that we pack at level v. Thus, given that $H(Lay_1) > \frac{1}{2}\frac{4v}{3}$, we have $H(Lay_2) \leq h_1 + h_2 + h_3 - H(Lay_1) \leq 2v - H(Lay_1) \leq \frac{4v}{3}$ and thus all the rectangles fit. □

It remains now to study how to finish packing all the rectangles in cases described Lines 25, 27 and 30. Remind that in case described Line 25 we pack all the rectangles of I, and not only I'.

Lemma 9.21 (First end of Algorithm 2)**.** *In the case Line 25, it is possible to pack all the remaining rectangles of I.*

Proof Let i_0 be the value of i Line 19 when all wide rectangles are packed. We have by definition here $S(\pi_x) \geq v$, for any $x \leq i_0 - 1$. Let X be the set of remaining rectangles and $X' = \pi_{i_0}$, at the beginning of Line 25. We will prove that $S(X \cup X') \leq v$, and thus Steinberg algorithm (or Widest First) packs $X \cup X'$ with a height lower than $2v$. If $S(X \cup X') > v$, we never ran out of rectangles when packing S_l, $i_0 + 1 \leq l \leq N$ and according to Lemma 9.19 we have $S(\pi_l) > v$ for $i_0+1 \leq l \leq N$, leading to $S(I) > Nv$. □

Lemma 9.22 (Second end of Algorithm 2)**.** *In the case Line 27, it is possible to pack all the remaining rectangles of I'.*

Proof Let i_0 be the value of i Line 19. We know that there exists $x \leq i_0 - 1$ such that $S(\pi_x) < v$. It means that we cannot use the same area argument as in Lemma 9.21. Thus, we will rather use counting arguments (see Section 9.3.2).

Let i_2 (resp. i_3) be the value of the index of the first strip used in phase 2 (resp. 3). Let us first prove that π_l is dominating (see Definition 9.12), for $1 \leq l \leq i_0 - 1$. All the strips packed in phase 1 are dominating since $CreateLayer$ packs at least two rectangles

of $L_{WD} \cap L_{XH}$ in each strip. According to Lemma 9.17, all the strips packed in phase 2 are also dominating. Thus, we now that all the π_l are dominating for $1 \leq l \leq i_3 - 1$.

We now prove that the layers created at the beginning of phase 3 are dominating, i.e. π_l is dominating for $i_3 \leq l \leq i_0 - 1$.

Remark 9.23. *We never ran out of rectangles of L_B when creating Lay_l,for $l \in [|i_3, i_0 - 1|]$ (meaning that L_B was not empty when starting creating Lay_{i_0}.)*

Proof Let x be the smaller index such that $S(\pi_x) < v$. We now that S_x can only be a strip packed during phase 1 or phase 3. This implies that $H(Lay_x) < 2v$, and thus the algorithm ran out of rectangles of $L_{WD} \setminus L_B$ when creating Lay_x. Then, there are only rectangles of L_B in all the layers created after Lay_x. □

Thus, when starting phase 3 we now that

- $|L_{WD} \cap L_{XH}| < 2$, as phase 1 finished
- $|L_{WD} \cap L_H| < 2$, as by assumption of this lemma we have $few\_high = true$
- we did not run ouf of rectangles of L_B when creating Lay_l,for $l \in [|i_3, i_0 - 1|]$ in Phase 3

Under this conditions, we will now prove that π_l is dominating for $i_3 \leq l \leq i_0 - 1$. Let (a_{XH}, a_H, a_M) be the number of rectangles of L_{XH}, L_H and L_M added to a layer Lay_l, with $i_3 \leq l \leq i_0 - 1$. We have $(a_{XH}, a_H, a_M) \in \{(1,1,1), (0,1,3), (0,0,4)\}$, implying that any Lay_l for $l \in [|i_3, i_0 - 1|]$ is dominating.

Thus, π_l is dominating for $1 \leq l \leq i_0 - 1$. Then, according to Lemma 9.15, we can create (if we don't run out of rectangles) dominating packing in empty strips S_l, $i_0 + 1 \leq l \leq N$. Finally, let X be the remaining rectangles after filling these strips, and let X' be the rectangles of π_{i_0} at Line 19. It remains to pack $X \cup X'$ in S_{i_0}. We prove by case analysis (according to rectangles of $L_{WD} \cap L_B$ packed in π_{i_0}) that if the remaining rectangles do not fit in S_{i_0}, then I' is partitioned in N sets of rectangles $(\pi_1, \ldots, \pi_{i_0-1}, X \cup X', \pi_{i_0+1}, \ldots, \pi_N)$ that are dominating, which is impossible according to Lemma 9.14. We only have to consider cases where π_{i_0} is not dominating. Let (b_{XH}, b_H, b_M) be the number of rectangles of $L_{WD} \cap L_{XH}$, $L_{WD} \cap L_H$ and $L_{WD} \cap L_M$ contained in π_{i_0}. As before we now that b_{XH} and b_H are strictly lower than 2. Moreover, as π_0 must be non dominating, we must have $2b_{XH} + b_H + b_M \leq 4$. Thus, the only possible cases are $(b_{XH}, b_H, b_M) \in \{(1,1,0), (1,0,1), (1,0,0), (0,1,j), (0,0,j')\}$, for $0 \leq j \leq 2, 0 \leq j' \leq 3$. For each possible value of (b_{XH}, b_H, b_M), we have to consider different cases according to what kind of rectangles remain in X. For the sake of brevity, we only analyze here two different cases. Proofs for the other cases can be directly adapted.

Let start with $(b_{XH}, b_H, b_M) = (1,1,0)$. Let $X' = \{r_1, r_2\}$ with $r_1 \in L_{WD} \cap L_{XH}$ and $r_2 \in L_{WD} \cap L_H$. We start by repacking r_1 and r_2 from level 0, right justified, such that the narrowest rectangle is on the top. In this case we do not pack rectangles of X on the top of the one of X', since $H(X')$ could be equal to $v + \frac{2v}{3}$, and we could have $X \cap L_{XH} \neq \emptyset$. However, if all the rectangles of X do not fit at level $lvl(r_1)$ (which is the level where r_1 is packed), then we get $W(X) > 1 - w_1$. Thus, we have $\sum_{x=1}^{3} f^x(X \cup X') > 2w_1 + w_2 + (1 - w_1) > 2$. Let us now consider a case where we pack rectangles on top of the one of X'. For example with $(b_{XH}, b_H, b_M) = (0,1,1)$. Let $X' = \{r_1, r_2\}$ with $r_1 \in L_{WD} \cap L_H$ and $r_2 \in L_{WD} \cap L_M$. We also start by repacking r_1 and r_2 from level 0, right justified, such that the narrowest rectangle is on the top.

We have $h_1 + h_2 < \frac{4v}{3}$. If all the rectangles of X do not fit at level $\frac{4v}{3}$, then we get $W(X) > 1$, and $\sum_{x=1}^{3} f^x(X \cup X') > w_1 + w_2 + 1 > 2$.

The other cases can be treated following the same arguments, and we conclude that we can pack $X \cup X'$ in S_{i_0}. □

It remains now to analyze the last possible end of Algorithm 2.

Lemma 9.24 (Third end of Algorithm 2)**.** *In the case Line 30, it is possible to pack all the remaining rectangles of I'.*

Proof Let i_3 be the value of the index of the first strip packed in Phase 3. As we packed two rectangles of $L_{XH} \cap L_{WD}$ or $L_H \cap L_{WD}$ in each of the $i_3 - 1$ first strips, and as few_high is false, we have $2(i_3 - 1) + 2 \leq |L_{WD} \cap (L_{XH} \cup L_H)| \leq N$, implying $i_3 \leq \lfloor \frac{N}{2} \rfloor$.

In this lemma, we had (at the beginning of phase 3) $L_{WD} \cap L_H \geq 2$, implying $W(\bar{L_{XH}}) \leq 1$ and $W(\bar{L_H} \cup \bar{L_M}) \leq \frac{3}{2}$ Thus, we need at most one free strip to pack $\bar{L_{XH}} \cup \bar{L_H} \cup \bar{L_M}$. Let x be the number of strips packed with rectangles of L_{WD} in phase 3. We need $i_3 + x - 1 < N$.

If $x \geq 2$, we have

$$\begin{aligned} H(L_{WD}) &> v(i_3 - 1) + \sum_{j=i_3}^{i_3+x-3} H(Lay_j) \\ &\quad + H(Lay_{i_3+x-2}) + H(Lay_{i_3+x-1}) \\ &> v(i_3 - 1) + v(x - 2) + 2v \end{aligned}$$

Thus, as $H(L_{WD}) \leq Nv$, and we get $i_3 + x - 1 < N$. If $x \leq 1$, then $x + i_3 \leq \lfloor \frac{N}{2} \rfloor + 1 < N + 1$.

□

Thus, we proved that the three possible ends of *BuildPreallocation* are feasible. According to the main steps defined in Section 9.2.2, the *BuildPreallocation* algorithm that preallocates I' is sufficient to get a $^7/_3$-approximation. Indeed, we simply add rectangles of $I \setminus I'$ using list algorithms defined in Section 9.2.2

9.3.6 Complexity

Let us now bound the computational complexity of the algorithm. Remark first that Phase 1 and Phase 2 of *BuildPreallocation* can be implemented in $\mathcal{O}(Nn + nlog(n))$. Indeed, before Phase 1 we sort the wide rectangles according to their height. Thus, all the calls to *CreateLayer* in phase 1 can be implemented in $\mathcal{O}(n)$. Then, before Phase 2 we resort the remaining rectangles according to their width. Thus, all the calls to $CreateShelf$ in phase 2 can be implemented in $\mathcal{O}(n)$ (remark that there is only a constant number of rectangles that are packed in Phase 2 without using $CreateShelf$).

Phase 3 runs in $\mathcal{O}(N(n + nlog(n)))$, using Widest First rather than Steinberg algorithm. Indeed, this bound is clearly true for algorithms described in Lemma 9.22 and 9.24. Applying algorithm described in Lemma 9.19 on strips $\{S_a, \ldots, S_{a+x}\}$ requires

$\mathcal{O}((x+1)n)$ for creating sets *select* for each strip (by resorting the remaining rectangles according to their area before filling this $x+1$ strips), plus $\mathcal{O}(\sum_{i=a}^{a+x} n_i log(n_i)) = \mathcal{O}(nlog(n))$ for applying Widest First on each strip (where n_i is the number of rectangles of *select* in S_i). Thus π_0 is constructed in $\mathcal{O}(n(N+log(n)))$.

Due to the simple structure of preallocation, the LS_{π_0} algorithm can be implemented in $\mathcal{O}(nlog(n))$. Instead of scanning level by level and strip by strip, this algorithm can be implemented by maintaining a list that contains the set of "currently" packed rectangles. The list contains 3-tuples (j,l,i) indicating that the top of rectangle r_j (packed on strip S_i) is at level l. Thus, instead of scanning every level from 0 it is sufficient to maintain sorted this list according to the l values (in non decreasing order), and to only consider at every step the first element of the list. Then, it takes $\mathcal{O}(log(n))$ to find a rectangle r_{j_0} in the appropriate shelf that fits at level l, because a shelf can be created as a sorted array. It also takes $\mathcal{O}(log(n))$ to insert the new event corresponding to the end of r_{j_0} in the list.

The last step, which turns π_1 into the final packing can also be implemented in $\mathcal{O}(nlog(n))$ using a similar global list of events. Notice that for any strip S_i, there exists a l_i such that bellow l_i the utilization is an arbitrary function strictly larger than $1/2$, and after l_i a non increasing after. Packing a small rectangle before l_i would require additional data structure to handle the complex shape. Thus we do not pack any small rectangle before l_i as it is not necessary for achieving the $7/3$ ratio. Therefore, we only add those events that happen after l_i when initializing the global list for this step. To summarize, for this step we only need to sort the small rectangles in non increasing order of their required number of processors, and then apply the same global list algorithm.

The binary search on v to find the smallest v which is not rejected can be done in $\mathcal{O}(log_2(nh_{max}))$. Thus the overall complexity of the $7/3$-approximation is in $\mathcal{O}(log_2(nh_{max})N(n+log(n)))$.

9.4 A 2-approximation for a special case

In this section we propose a 2-approximation for a special case of MSP_r^{nc}. We follow the ideas presented in Section 9.2, and thus we re-use the notion of **layer**, **shelf** and **bin**, the procedures named **CreateLayer**, **CreateShelf** and **GreedyPack**. We will also use the dual approximation technique [Hochbaum and Shmoys, 1988], and we denote by v the guess of the optimal value.

The construction of the preallocation π_0 of I' is presented from Section 9.4.3 to 9.4.5. The final steps to turn π_0 into a $\frac{2}{3}$-compact packing π_1 and to turn π_1 into the final packing π are quickly described in Section 9.4.6, as they follow the steps presented in Section 9.2.2.

9.4.1 Definition of the considered problem

The $2+\epsilon$-approximation in [Ye et al., 2009] and the 2-approximation we recently proposed in [Bougeret et al., 2009] are very costly, and thus hard to use for "large" instances. As for the $\frac{5}{2}$-approximation in Chapter 8 and the $\frac{7}{3}$-approximation presented bellow, we aim at constructing low cost algorithms that could be used in a practical context. Thus, we are looking for a (reasonable) restriction of the MSP_r^{nc} that would

help to tight the bounds, and we consider that all the rectangles have width lower or equal to 1/2.

Lemma 9.25. *The MSP_r^{nc} where every rectangle has width lower (or equal) to $\frac{1}{2}$ has no polynomial algorithm with a ratio strictly better than 2, unless $P = NP$.*

Proof As in [Zhuk, 2006] for the general version, we construct a gap reduction from the 2-partition problem. Let $\{x_1, \ldots, x_n\} \subset \mathbb{N}^n$ and a such that $\sum_{i=1}^{n} = 2a$. Without loss of generality, let us assume that for any i, $x_i < a$. In order to also have only items with size at least two, we define $x'_i = 2x_i$ for any i, and $a' = 2a$. We construct the following instance I_{MSP} of "restricted" MSP_r^{nc}. We chose $N = 2$ strips, each strip having size $2a' - 1$. The set of rectangle is $\{r_1, \ldots, r_n, r_{n+1}, r_{n+2}\}$, with $w_i = x'_i$ for $1 \leq i \leq n$, $w_{n+1} = w_{n+2} = a' - 1$, and $h_i = 1$ for $1 \leq i \leq n + 2$. We have $w_i \leq \frac{2a'-1}{2}$ for any i, as all the x_i are strictly lower than a. Notice than any solution to I_{MSP} that packs r_{n+1} and r_{n+2} is the same strip have a height of at least 2, as the available width of size 1 in that strip cannot be used by any rectangle.

Obviously, if there is a 2-partition, then $Opt(I_{MSP}) = 1$. Otherwise, as r_{n+1} and r_{n+2} cannot be packed together, we have $Opt(I_{MSP}) = 2$ □

9.4.2 Definition of the considered partition

Recall that all rectangles have $w_j \leq \frac{1}{2}$. Let us define the following sets :

- let $L_{WD} = \{r_j | w_j > 1/3\}$ be the set of wide rectangles
- let $L_{XH} = \{r_j | h_j > 2v/3\}$ be the set of extra high rectangles
- let $L_H = \{r_j | 2v/3 \geq h_j > v/2\}$ be the set of high rectangles
- let $L_B = (L_{XH} \cup L_H) \cap L_{WD}$ be the set of huge rectangles, and $b = Card(L_B)$.
- let $I' = L_{WD} \cup L_{XH} \cup L_H$

As in the previous $\frac{5}{2}$-approximation, we do not mention explicitly the "reject" instruction, and we assume that $v \geq Opt$. Notice than we only consider the values v such that

- $W(L_{XH} \cup L_H) \leq N$
- $H(L_{WD}) \leq 2Nv$

We now provide a two phases algorithm that builds the preallocation π_0 of the rectangles of I'. Let π_0^i denote the set of rectangles packed in S_i. Phase 1 (see Section 9.4.3) preallocates rectangles of L_{WD}, and phase 2 (see Section 9.4.5) preallocates rectangles of $L_H \cup L_{XH}$.

9.4.3 Phase 1

Description of phase 1

Phase 1 packs the rectangles of L_{WD} by calling for each strip (until L_{WD} is empty) two times $CreateLayer(L_{WD}, 2v)$. Let us denote by Lay_{2i-1} and Lay_{2i} the layers created in strip S_i. Let us say that Lay_{2i-1} is packed left justified, and Lay_{2i} is packed right justified. Moreover, each layer is repacked in non increasing order of the widths, such that the narrowest rectangles are packed on the top.

Let N_1 denote the number of strips used in phase 1, and let i_1 denote the index of the last created layer (Lay_{i_1} is of course in S_{N_1}). Let L_H^1 and L_{XH}^1 denote the set of remaining rectangles after phase 1 of L_H and L_{XH}, respectively. Thus, for the moment we have $\pi_0^i = Lay_{2i} \cup Lay_{2i-1}$ for all $i \leq N_1$.

Analysis of phase 1

Lemma 9.26. *If $\exists i_0 < i_1$ such that $H(Lay_{i_0}) \leq \frac{3v}{2}$ then it is straightforward to preallocate I'.*

Proof Let $i_0 < i_1$ such that $H(Lay_{i_0}) \leq \frac{3v}{2}$. This implies that we ran out of rectangles of $L_{Wd} \setminus (L_H \cup L_{XH})$ while creating layer i_0. Thus, because of the BFH order there are at least two rectangles of L_B in every layer Lay_i, for $1 \leq i < i_1$, implying that the width of high and extra high rectangles packed in each of these layers is strictly larger than 2/3. Thus, $W(\pi_0^i \cap (L_H \cup L_{XH})) > 4/3 > 1$ for $1 \leq i < N_1$. Thus, the total width of remaining high and extra high rectangles is lower than $N - (N_1 - 1)$.

Let us prove that we can pack all the remaining rectangles of I' (which are included in $(L_H \cup L_{XH})$) in the remaining strips. For each $i \in [|N_1 + 1, N|]$ we create two shelves in S_i (one at level 0 and one at level v). If there are still some unpacked rectangles, then all the shelves are "full", that is the width of each shelf is larger than 2/3 (as all the width of any rectangle of $L_H \cup L_{XH}$ is lower than 1/3). Thus, we have $W(\pi_0^i \cap (L_H \cup L_{XH})) > 4/3 > 1$ (for $N_1 + 1 \leq i \leq N$). This implies that the total width of remaining rectangles of $L_H \cup L_{XH}$ (including those in strip S_{N_1}) is now lower than 1. Thus, we can pack all of them in one shelf in S_{N_1}. □

From now we assume that $H(Lay_i) > \frac{3w}{2}$ for all $i < i_1$. This implies that $S(\pi_0^i) > v$ for $i < N_1$. Moreover, we have $2Nv \geq H(L_{Wd}) \geq \sum_{i=1}^{N_1-1} H(\pi_0^i \cap L_{Wd}) > (N_1 - 1)2(3v/2)$, implying $N_1 < \frac{2}{3}N + 1$.

Lemma 9.27. *If there is a rectangle of L_B in lay_{i_1-1}, then it is straightforward to preallocate I'.*

Proof We first consider the case where there are two layers in strip N_1 Let us count the cumulative width of high and extra high rectangles that already packed. In the first $N_1 - 1$ strips, we packed at least two rectangles of B in each layer, implying $\sum_{1 \leq i \leq N_1 - 1} W(\pi_0^i \cap (L_{XH} \cup L_H)) \geq \frac{4}{3}(N_1 - 1)$. In strip N_1, we have $W(\pi_0^{N_1} \cap (L_{XH} \cup L_H)) > 1/3$ by hypothesis. Let $N_2 = N - N_1$. As in Lemma 9.26, we create two shelves (using a widest first policy) of rectangles of $L_{XH} \cup L_H$ in each strip S_{N_1+i}, for $1 \leq i \leq N_2$. If we don't run out of rectangles, the remaining width of high and extra high rectangles after packing the first $2N_2 - 1$ shelves is strictly larger than 1. Given that each shelf has width at least $\frac{3}{4}$ (see Lemma 9.2), we have $N \geq W(L_{XH} \cup L_H) > \frac{4}{3}(N_1 - 1) + \frac{1}{3} + \frac{3}{4}(2N_2 - 1) + 1 = -\frac{N_1}{6} + \frac{3N}{2} - \frac{3}{4}$, leading to $N_1 \geq 3N - \frac{9}{2}$. As $N_1 < \frac{2}{3}N + 1$, we conclude that $3N - \frac{9}{2} < \frac{2N}{3} + 1$, which is a contradiction for $N \geq 3$.

If $N = 2$, we have $W(L_{XH} \cup L_H) \leq 2$ and $H(L_{WD}) \leq 4v$. Given that $H(\pi_0^1 \cap L_{WD}) > 3v$, we get $H(lay_3) \leq v$ and $H(Lay_4) = 0$ which is a contradiction because we supposed that there were two layers in strip N_1.

The case where there is only one layer in strip N_1 can be treated using the same arguments for $N \geq 3$. If $N = 2$, then we have (as before) $H(lay_3) \leq v$. Moreover, $W(\pi_0^1 \cap (L_{XH} \cup L_H)) > \frac{4}{3} > 1$ because there are at least two rectangles of L_B in Lay_1 and one rectangle of L_B in Lay_2. Thus, there is enough space in strip S_2 to pack one shelf of $L_{XH} \cup L_H$, which is sufficient. □

Naturally we consider from now on that the area packed in the first $N_1 - 1$ is strictly more than $(N_1 - 1)v$, and that there is no huge rectangle in the last two layers created by phase 1. It remains now to pack $L_H^1 \cup L_{XH}^1$. Notice that $(L_H^1 \cup L_{XH}^1) \cap L_{WD} = \emptyset$.

9.4.4 Packing techniques for high and extra high rectangles

Preliminaries

Let $N_2 = N - N_1$ denote the number of free strips after phase 1. Roughly speaking, phase 2 packs shelves of high of extra high rectangles in each of the N_2 last strips (using the $Pack\_Shelf$, $GreedyPack$ and Add_2 procedures), and merges some high or extra high rectangles with the ones packed in strip N_1 (using the $Merge$ procedure).

In this section we present a technique to fill α empty strips with high or extra high rectangles. In the Section 9.4.5, we use this technique for $\alpha = N_2$ (using strips S_{N_1+1} ... S_N) and an additional merging algorithm (that fills efficiently strip S_{N_1}) to pack $L_H^1 \cup L_{XH}^1$.

We now define the two sequences of bins seq_{XH} and seq_H that will be used by $GreedyPack$. Every bin of seq_{XH} (resp. seq_H) will (possibly) contain one shelf of rectangles of L_{XH} (resp. L_H). Notice that in a free strip it is possible to pack two bins of height v (and width 1), three bins of height $2v/3$, or one bin of size v and one bin of size $2v/3$. Thus, seq_{XH} is composed of 2α bins $(b_1, \ldots, b_{2\alpha})$ of height v, considering that we created two bins of height one in each of the strips $S_1, \ldots S_\alpha$. More precisely, for all i we locate b_{2i-1} and b_{2i} in S_i, with b_{2i-x} at level $v - x$ for $x \in \{0, 1\}$. The sequence seq_H is composed of 3α bins $(b'_1, \ldots, b'_{3\alpha})$ of height $2v/3$, considering that we created three bins in each of the strips $S_\alpha, \ldots S_1$. It means that for all $i \geq 1$, bins b'_{3i-2}, b'_{3i-1} and b'_{3i} are located in $S_{\alpha-i+1}$, with b'_{3i-x} at level $\frac{2xv}{3}$ for $x \in \{0, 2\}$. Thus, given $\widehat{L_{XH}} \subset L_{XH} \setminus L_{WD}$ and $\widehat{L_H} \subset L_H \setminus L_{WD}$, we use the previous sequences of bins as follows :

- $last$ = Greedy_Pack($\widehat{L_{XH}}, seq_{XH}$)
- Greedy_Pack($\widehat{L_H}, seq_H$)

Finally, let us define the $Add_2(X, S_{i_{last}})$ procedure that packs the remaining rectangles of $X \subset L_H \setminus L_{Wd}$ in $S_{i_{last}}$. Two cases are possible according to what is preallocated in $S_{i_{last}}$.

In the first case $S_{i_{last}}$ contains a first "full" shelf (whose surface is at least $v/2$) of rectangles of L_{XH} at level 0, and a shelf sh of rectangles of L_{XH} packed at level v, right justified. In this case, Add_2 creates a shelf sh_1 using $CreateShelf(X, 1 - W(sh))$ and preallocate sh_1 at level v, left justified.

In the second case $S_{i_{last}}$ (see Figure 9.28) contains only a shelf of rectangles of L_{XH} packed at level 0, right justified. In this case, Add_2 first moves some rectangles from sh to a new shelf $\widehat{sh}$ until $W(sh) \leq 2/3$. Then, Add_2 packs (right justified) the widest of these two shelves (denoted by sh_A) at level 0, and the other one (denoted by sh_B) at

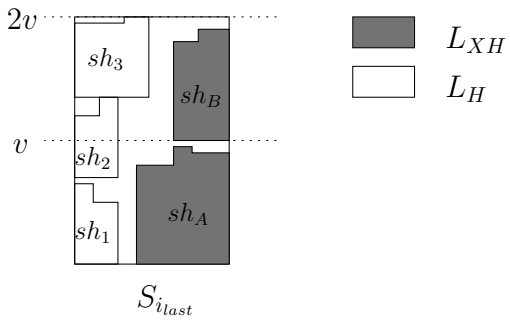

FIG. 9.28: Example the add_2 procedure.

level v. Then, Add_2 creates two shelves sh_1 and sh_2 using $CreateShelf(X, 1-W(sh_A))$ and one shelf sh_3 using $CreateShelf(X, 1 - W(sh_B))$. Then, sh_i is packed at level $\frac{2v(i-1)}{3}$, left justified. Notice that if we packed sh_B at level 0 (instead of sh_A) then sh_2 could intersect sh_B. Note also that stacking the shelves sh_1, sh_2, sh_3 does not exceed $2v$.

Filling α empty strips with high and extra high rectangles

The next lemma shows how to fill α free strips.

Lemma 9.29. *Let $\widehat{L_{XH}} \subset L_{XH} \setminus L_{WD}$ and $\widehat{L_H} \subset L_H \setminus L_{WD}$. Suppose that we execute the following calls :*

1. *last = Greedy_Pack($\widehat{L_{XH}}, seq_{XH}$)*
2. *Greedy_Pack($\widehat{L_H}, seq_H$)*
3. *$Add_2(\widehat{L_H}, S_{i_{last}})$ where $S_{i_{last}}$ denotes the strip containing bin last.*

Then we get the following properties :

- *If $\widehat{L_{XH}} \neq \emptyset$ after 1, then $S(\widehat{L_{XH}}) > (\alpha + \frac{1}{6})v$*
- *Otherwise, if $\widehat{L_H} \neq \emptyset$ after 3, then $S(\widehat{L_{XH}} \cup \widehat{L_H}) > \alpha v$.*

Remark 9.30. *Notice that before the call to Add_2, $S_{i_{last}}$ is the only strip that maybe already contains high and extra high rectangles. Moreover, this can happen only if the last shelf of extra high rectangles is at level 0, (because otherwise the places corresponding to $b'_{3i_{last}-2}$ and $b'_{3i_{last}-2}$ are not completely free, and thus GreedyPack do not pack any high rectangles in these bins).*

Remark 9.31. *Let X such that $X \cap L_{Wd} = \emptyset$ and let sh denote a shelf created by $Pack\_Shelf(X, 1)$, supposing that we didn't run out of rectangle while creating the shelf. Then, $W(sh) > 3/4$ according to Lemma 9.2 as at least three rectangles fit. Moreover, if $X \subset L_{XH}$ then $S(sh) > v/2$, and if $X \subset L_H$ then $S(sh) > 3v/8$.*

To prove Lemma 9.29 we first need to show the following one.

Lemma 9.32. *Let $X \subset L_H \setminus L_{WD}$ and S_i be a strip packed as expected for $Add_2(X, S_i)$. Let π_0^i denote the rectangles packed in S_i before the call $Add_2(X, S_i)$. If $X \neq \emptyset$ after calling the procedure, then $S(\pi_0^i \cup X) > v$.*

Proof Remember that two cases are possible according to what is already packed in S_i before the call. Let us first suppose that there is one full shelf (of area strictly larger than $v/2$) of extra high rectangles (at level 0) and another shelf sh of extra high rectangles at level v. Then, $X \neq \emptyset$ after the call implies that $W(X) > 1 - W(sh)$, and we have $S(\pi_0^i \cup X) > \frac{v}{2} + W(sh)\frac{2v}{3} + W(X)\frac{v}{2} > v$.

Let us now suppose that S_i contains only one shelf sh of L_{XH} at level 0. Let $sh_A, sh_B, sh_1, sh_2, sh_3$ be defined as described in the Add_2 procedure, Section 9.4.4. As $W(sh_A) \leq \frac{2}{3}$ and $X \cap L_{WD} = \emptyset$, sh_1 and sh_2 contain at least one rectangle, implying that $W(sh_1)$ and $W(sh_2)$ are strictly larger than $\frac{1-W(sh_A)}{2}$ according to Lemma 9.2. Moreover, $X \neq \emptyset$ after the call implies that after creating sh_1 and sh_2 the total width of remaining rectangles of X was strictly larger than $1-W(sh_B)$. Putting this together, we get $S(\pi_0^i \cup X) > (1 - W(sh_A))\frac{v}{2} + (1 - W(sh_B))\frac{v}{2} + (W(sh_A) + W(sh_B))\frac{2v}{3} > v$. □

Let us now prove Lemma 9.29.

Proof [Proof of lemma 9.29] Let us first suppose that $\widehat{L_{XH}} \neq \emptyset$ after step 1. It implies that after creating the first $2\alpha - 1$ shelves of width at least $3/4$ (according to Remark 9.31), the total remaining width of rectangles of $\widehat{L_{XH}}$ was strictly larger than 1. Thus, $S(\widehat{L_{XH}}) > \frac{3}{4}(2\alpha - 1)\frac{2v}{3} + \frac{2v}{3} = (\alpha + \frac{1}{6})v$.

Let us now suppose that $\widehat{L_{XH}} = \emptyset$ and $\widehat{L_H} \neq \emptyset$ after step 3. Let sh denote the shelf of rectangles of $\widehat{L_{XH}}$ contained in bin $last$. Remind that i_{last} is the index of the strip containing $last$. For all $i \in [|1, i_{last} - 1|]$, $S(\pi_0^i \cap (\widehat{L_{XH}} \cup \widehat{L_H})) > 2\frac{v}{2} = v$. For all $i \in [|i_{last} + 1, \alpha|]$, $S(\pi_0^i \cap (\widehat{L_{XH}} \cup \widehat{L_H})) > 3\frac{3v}{8} > v$. According to Lemma 9.32, $\widehat{L_H} \neq \emptyset$ implies $S(\pi_0^{i_{last}} \cap \widehat{L_H}) > v$. Then the lemma follows. □

9.4.5 Phase 2

In phase 1 we preallocated L_{WD} in strips $S_1, \ldots, S_{N_1}$. Recall that each layer created in phase 1 is sorted It remains to preallocate $L^1_{XH} \cup L^1_H$ in $S_{N_1}, \ldots, S_N$.

Lemma 9.33. *It is possible to preallocate $L^1_{XH} \cup L^1_H$ in $S_{N_1}, \ldots, S_N$ with a resulting height lower than $2v$.*

Lemma 9.33 is proved by case distinction according to what is preallocated by phase 1 in S_{N_1}. In each case we will define an appropriate **Merge** procedure. In almost all of these cases we prove that if the considered calls are not sufficient to pack $L^1_{XH} \cup L^1_H$, then $S(I') > Nv$. Let us introduce the notation $av(l)$ to denote the available width at level l (in a given strip).

Cases where two layers are preallocated in S_{N_1}

Recall that i_1 is the index of the last created layer in phase 1. We assume here that $i_1 = 2N_1$. Let $p_1 = H(Lay_{i_1})$, $p_2 = H(Lay_{i_1-1})$ and $k = Card(Lay_{i_1})$. Without loss of generality, let us denote by $\{r_1, \ldots, r_k\}$ the rectangles of Lay_{i_1}, with $r_j, 1 \leq j \leq k$ sorted in non increasing order of their heights (so $p_1 = \sum_{1 \leq j \leq k} h_j$). Remember that we assume that no rectangle of L_B in the last two layers of phase 1. We proceed by case analysis according to the value of p_1.

If $p_1 > \frac{4v}{3}$.

The algorithm makes the following calls :

1. $last = \text{Greedy\_Pack}(L^1_{XH}, seq_{XH})$
2. $\text{Greedy\_Pack}(L^1_H, seq_H)$
3. $Add_2(L^1_H, S_{i_{last}})$

Let us prove that $p_1 + p_2 > 3v$. We have $p_2 > 2 - h_k$ as r_k was not included in Lay_{i_1-1}, and $p_1 \geq max(kh_k, \frac{4v}{3})$. Thus $p_1 + p_2 \geq max(2 + (k-1)h_k, 2 - h_k + \frac{4v}{3})$. This last term is minimized for $h_k = \frac{4v}{3k}$, leading to $p_1 + p_2 \geq 2 + \frac{4v}{3}(1 - \frac{1}{k})$ which is greater than $3v$ for $k \geq 4$.

Moreover, $k > 2$ as $p > \frac{4v}{3}$ and no rectangles of L_B are packed in strip S_{N_1} according to Lemma 9.27. Thus, it remains to consider the case where $k = 3$. Given that there is at least 4 rectangles in Lay_{i_1-1} (as no rectangles of L_B are packed in strip N_1) we have $p_2 > 4h_1$, and as $h_1 > \frac{4}{3k}$, we get $p_1 + p_2 > \frac{16}{3k} + \frac{4}{3} > \frac{28}{9} > 3$.

Thus in this case it is not necessary to preallocate anything else in S_{N_1} as we have $p_1 + p_2 > 3v$ implying $S(\pi_0^{N_1}) > v$. Then, we conclude that no rectangle of $L^1_{XH} \cup L^1_H$ remains after step 3 using Lemma 9.29.

If $\frac{4v}{3} \geq p > v$.

Let us first define the $Merge(X)$ procedure for this case. In this case (see Figure 9.34 Page 140), $merge(X)$ creates one shelf sh using $CreateShelf(X, x)$, where $x = av(\frac{4v}{3})$, and packs it at level $\frac{4v}{3}$, right justified.

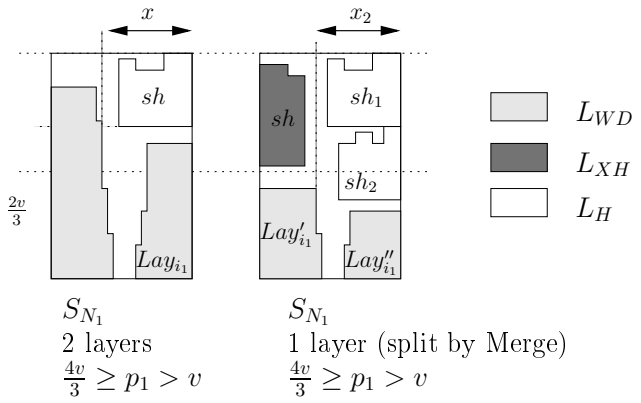

FIG. 9.34: Example of $Merge$ for two different cases.

The algorithm makes the following calls :

1. Merge(L_H^1)
2. $last$ = Greedy_Pack(L_{XH}^1, seq_{XH})
3. Greedy_Pack(L_H^1, seq_H)
4. $Add(L_H^1, S_{i_{last}})$

Let us consider a first case where L_H^1 is not empty after step 1. According to Lemma 9.2, $W(sh) > \frac{x}{2}$. After step 1, we have $S(\pi_0^{N_1}) > \frac{1}{3}(p_1) + S(Lay_{i_1-1}) + S(sh)$. Moreover, $S(Lay_{i_1-1}) + S(sh) \geq (1-x)\frac{4v}{3} + (p_2 - \frac{4v}{3})\frac{1}{3} + \frac{xv}{4}$, (see Figure 9.34 Page 140), which is decreasing in x. Thus, the lower bound is reached for $x = \frac{2}{3}$, and in this case $S(\pi_0^{N_1}) > \frac{p_1+p_2}{3} + W(sh)\frac{v}{2} > \frac{5v}{6} + \frac{1}{3}\frac{v}{2} = v$. Then, according to Lemma 9.29, no rectangle remains after step 4.

We now suppose that L_H^1 is empty after step 1. Before 1, $S(\pi_0^{N_1}) > \frac{p_1+p_2}{3} > (\frac{3}{2}+1)\frac{v}{3} = \frac{5v}{6}$. If $L_{XH}^1 \neq \emptyset$ after step 2, then according to Lemma 9.29 $S(L_{XH}^1) > (N_2 + \frac{1}{6})v$ implying $S(\pi_0^{N_1}) + S(L_{XH}) > (N_2+1)v$. Thus in this case no rectangle remains after step 2.

If $v \geq p > \frac{2v}{3}$.

We first analyze the case where $N_1 > 1$. Let us consider the same algorithm as in the previous case (with in particular the same definition of x). Let us first suppose that $L_H^1 \neq \emptyset$ after step 1. We will bound the surface of rectangle precallocated in $\pi_0^{N_1}$ after step 2 as before, except that we also take into account the $N_1 - 1$ first strips. Thus, after step 2 we have $S(\bigcup_{i=1}^{N_1} \pi_0^i) > (N_1-2)v + (H(Lay_{i_1-3}) + H(Lay_{i_1-2}) + p_1)\frac{1}{3} + S(Lay_{i_1-1}) + S(sh)$. As before, $S(Lay_{i_1-1}) + S(sh)$ is decreasing in x, and we replace x by $\frac{2}{3}$. Moreover, notice that $H(Lay_{i_1-3}) + H(Lay_{i_1-2}) + p_1 > 4v$ as there is at least two rectangles in lay_{i_1}, and none of these rectangles has been included in Lay_{i_1-3} or Lay_{i_1-2}. Finally, we get $S(\bigcup_{i=1}^{N_1} \pi_0^i) > (N_1-2)v + \frac{4v}{3} + \frac{p_2}{3} + \frac{1}{3}\frac{v}{2}$. Using $p_2 \geq \frac{3v}{2}$, we get $S(\bigcup_{i=1}^{N_1} \pi_0^i) > N_1 v$ The case where L_H^1 is empty after step 1 can be adapted in the same way.

Let us now consider the case with $N_1 = 1$. In this case it is sufficient to create two shelves of rectangles of $L_H^1 \cup L_{XH}^1$ in each S_i for $2 \leq i \leq N$. If we don't run out of rectangles, the total width of high and extra high rectangles packed in each of these strips is strictly larger than $\frac{3}{2}$. Thus, the total width λ of remaining rectangles of $L_H^1 \cup L_{XH}^1$ is at most $N - \frac{3}{2}(N-1)$ which is negative for $N \geq 3$. If $N = 2$, $\lambda \leq \frac{1}{2}$, and we can finish packing $L_H^1 \cup L_{XH}^1$ using one bin of width $\frac{1}{2}$ and height v whose bottom is located at level v.

If $\frac{2v}{3} \geq p > 0$.

Let us first define the $Merge(X)$ procedure for this case. If $X \subset L_{XH}$, $merge(X)$ creates one shelf sh using $CreateShelf(X, x)$, where $x = av(v)$, and packs it at level v, right justified.

If $X \subset L_H$, two sub-cases are possible according to what is packed in strip S_{N_1}. If no shelf of L_{XH} is preallocated in S_{N_1}, $merge(X)$ creates two shelves sh_1 and sh_2 using $CreateShelf(X, x_1)$ and $CreateShelf(X, x_2)$ respectively, where $x_i = av(\frac{(6-2i)v}{3})$. Notice that if sh_2 is not empty then $W(sh_1) + W(sh_2) > x_1$ as all the rectangles of sh_2 were not included in sh_1. Then, sh_i is packed at level $\frac{(6-2iv)}{3}$, right justified.

If there is a shelf sh of rectangles of L_{XH} preallocated in S_{N_1} (right justified, at level v), $merge(X)$ creates one shelf sh_1 using $CreateShelf(X, x)$ where $x = av(v)$ (x takes into account the wide rectangles added in phase 1 and the extra high ones in sh), and packs it at level v.

The algorithm makes the following calls :

1. $last = Greedy\_Pack(L^1_{XH}, seq_{XH})$
2. $Merge(L^1_{XH})$
3. $Merge(L^1_H)$
4. $Greedy\_Pack(L^1_H, seq_H)$
5. $Add(L^1_H, S_{i_{last}})$

Let us prove that L^1_{XH} is empty after step 2 by contradiction. If L^1_{XH} is not empty after step 2, then we claim that $S(I') > Nv$. Let $\bar{L^1_{XH}}$ and $\bar{\pi_0}^{N_1}$ be respectively L^1_{XH} and $\pi_0^{N_1}$ just before step 2. Just before step 2, the area packed in each of the N_2 last strips is strictly larger than v as $Greedy\_Pack(L^1_{XH}, seq^1_{XH})$ packs two shelves of area strictly larger than $\frac{v}{2}$ in each strip. If L^1_{XH} is not empty after step 2 then $W(\bar{L^1_{XH}}) > x$ (with x as defined just before for the $Merge$ procedure) and $S(\bar{\pi_0}^{N_1}) + S(\bar{L^1_{XH}}) > p_1\frac{1}{3} + S(Lay_{i_1-1}) + S(\bar{L^1_{XH}})$. As before, $S(Lay_{i_1-1}) + S(\bar{L^1_{XH}}) > v(1-x) + (p_2 - v)\frac{1}{3} + \frac{2v}{3}x$ is decreasing in x and the minimum is reached for $x = \frac{2}{3}$. Replacing x by this value and using the fact that $p_1 + p_2 > 2v$, we get $S(\bar{\pi_0}^{N_1}) + S(\bar{L^1_{XH}}) > \frac{2v}{3} + \frac{4v}{9} > v$, implying $S(I') > Nv$.

Thus L^1_{XH} is empty after step 2. There is now two cases according to what is packed in S_{N_1}. Let us start with the first case where no extra high rectangle packed in S_{N_1} (*i.e.* L_{XH} is empty after step 1). Remind that in this case two shelves (sh_1 and sh_2) of rectangles of L^1_H are created, and $W(sh_1) + W(sh_2) > x_1$. We will prove that L_H is empty after step 5 by contradiction. We first show that after step 3 we have $S(\pi_0^{N_1}) > v$, and then we will conclude using Lemma 9.29.

After phase 3 we get $S(\pi_0^{N_1}) > p_1\frac{1}{3} + S(Lay_{i_1-1}) + S(sh_1) + S(sh_2)$. As before, $S(Lay_{i_1-1}) + S(sh_1) + S(sh_2) > \frac{4v}{3}(1 - x_1) + (p_2 - \frac{4v}{3})\frac{1}{3} + x_1\frac{v}{2}$ is decreasing in x_1 and the minimum is reached for $x_1 = \frac{2}{3}$. Replacing x_1 we get $S(\pi_0^{N_1}) > (p_1 + p_2 - \frac{4v}{3})\frac{1}{3} + \frac{4v}{9} + \frac{v}{3} > v$. Then, we prove that step 4 must pack the remaining rectangles using Lemma 9.29. Thus in this case all the rectangles are packed.

In the second case some rectangles of L^1_{XH} are packed in S_{N_1} during step 2. Thus, the area packed (with rectangles of L^1_{XH}) in each of the last N_2 strips using $GreedyPack$ is strictly larger than v. We prove by contradiction that L^1_H must be packed at step 3. We use the same argument as before : the case which minimizes the our lower bound on $S(pi_0^{N_1}) + S(L^1_H)$ occurs when all the rectangles of Lay_{i_1-1} have width $\frac{1}{3}$. Thus, if L^1_H is not packed at step 3 we get $S(pi_0^{N_1}) + S(L^1_H) > v$, leading as usual to $S(I') > Nv$.

Cases where one layer is preallocated in S_{N_1}

We consider now cases where $i_1 = 2N_1 - 1$. Let $p_1 = H(Lay_{i_1})$, $p_2 = H(Lay_{i_1-1})$, $p_3 = H(Lay_{i_1-2})$ and $k = Card(Lay_{i_1})$. Without loss of generality, let us denote by $\{r_1, \ldots, r_k\}$ the set Lay_{i_1} (so $p_1 = \sum_{1 \leq j \leq k} h_j$), with r_j sorted in non increasing order of their height. We proceed by case analysis according to the value of p_1. Remember that we assume that no rectangle of L_B in the last two layers of phase 1.

We start treating all these cases with one layer by stating some useful properties in Lemma 9.35.

Lemma 9.35. *For the cases with one layer in S_{N_1}, we have*

1. *if $N_1 = 1$ then it is straightforward to preallocate $L^1_H \cup L^1_{XH}$*
2. *for all the cases where $p_1 > 1$, then $N_1 < N$*
3. *if $p_1 > \lambda$ with $\lambda > \frac{v}{2}$, then $p_1 + p_2 + p_3 > 3v + \lambda + max(\frac{v}{5}, v - 2\frac{\lambda}{k})$.*

Notice that property 3 states that we can improve the naive bound $p_1+p_2+p_3 > 3v+\lambda$.

Proof We start with property 1. If $N_1 = 1$ we can pack two shelves of rectangles of $L^1_H \cup L^1_{XH}$ in each strip $S_i, i \geq 2$, implying a that the width packed rectangles of $L^1_H \cup L^1_{XH}$ is at least $\frac{3}{2}$ in each strip. Therefore, the remaining width of rectangles of $L^1_H \cup L^1_{XH}$ will be at most $\frac{1}{2}$, and all these rectangles must fit in one shelf in S_1.

Property 2 is true as $2N \geq H(L_{WD}) > 3(N_1 - 1) + p_1$, leading to $N_1 < \frac{2N-1}{3} + 1$.

We now prove property 3. First notice that $\lambda > \frac{v}{2}$ implies $k \geq 2$. Moreover, p_2 and p_3 are strictly greater than $2v - h_k$ as r_k did not fit in Lay_{i_1-2} and Lay_{i_1-1}. On one side, we have $p_1 + p_2 + p_3 > b_1$ with $b_1 = \lambda + 2(2v - h_k)$, which is large for small values of h_k. On the other side (for large values of h_k), we have $p_1 + p_2 + p_3 > b_2$ with $b_2 = \lambda + (2v - h_k) + 4h_k$ as there is at least four rectangles in Lay_{i_1-1}, and $p_1 + p_2 + p_3 > b_3$ with $b_3 = kh_k + 2(2v - h_k)$ as $p_1 \geq kh_k$. Thus, $p_1 + p_2 + p_3$ is larger than $max(b_1, b_2)$ and $max(b_1, b_3)$. Then, we just prove that $max(b_1, b_2) \geq \frac{v}{5}$ and $max(b_1, b_3) \geq v - 2\frac{\lambda}{k}$ □

We now study the different cases, assuming that $N_1 > 1$.

If $p_1 > \frac{5v}{3}$.

Let us first define the $Merge(X)$ procedure for this case. If $X \subset L_{XH}$, $Merge$ creates two shelves sh_1 and sh_2 using $CreateShelf(X, x_1)$ and $CreateShelf(X, x_2)$ respectively, where $x_i = av((2 - i)v)$. Notice that if sh_2 is not empty then $W(sh_1) + W(sh_2) > x_1$, as rectangles of sh_2 were not included in sh_1. Then sh_i is packed at level $(2 - i)v$, right justified. If $X \subset L_H$, $Merge$ creates two shelves sh_1 and sh_2 using $CreateShelf(X, x_1)$ and $CreateShelf(X, x_2)$ respectively, where $x_i = av((i - 1)v)$ (we take into account wide rectangles preallocated in phase 1 and maybe extra high rectangles). Then sh_i is packed at level $(i - 1)v$, right justified.

The algorithm makes the following calls :

1. $Merge(L^1_{XH})$
2. $last = Greedy\_Pack(L^1_{XH}, seq_{XH})$
3. $Greedy\_Pack(L^1_H, seq_H)$
4. $Add_2(L^1_H, S_{i_{last}})$
5. $Merge(L^1_H)$

Let first suppose that two shelves of extra high rectangles are created in step 1. Recall that in this case $W(sh_1) + W(sh_2) > x_1$. Thus, after phase 1 we get $S(\pi_0^{N_1}) > (1 - x_1)v + (p_1 - v)\frac{1}{3} + \frac{2v}{3}x_1$, which is a decreasing function of x_1. With $x_1 = \frac{2}{3}$ we

get $S(\pi_0^{N_1}) > \frac{5v}{9} + \frac{4v}{9} = v$. Therefore we conclude that step 2, 3 and 4 must pack the remaining rectangles using Lemma 9.29.

We now suppose that step 1 created only one shelf, implying that L_{XH}^1 is empty after step 1. Let $\bar{L_H}^1$ denote the remaining rectangles of L_H^1 just before step 5. If $\bar{L_H}^1$ is not empty after step 5 the end, we get $S(\pi_0^i) > v$ for any $i \in [|N_1+1, N|]$. Moreover, $S(\pi_0^{N_1} \cup \bar{L_H}^1) > (1-x_2)v + (p_1 - v)\frac{1}{3} + S(sh_1) + x_2\frac{v}{2}$ which is a decreasing function of x_2. With $x_2 = \frac{2}{3}$ we get $S(\pi_0^{N_1} \cup \bar{L_H}^1) > \frac{5v}{9} + \frac{v}{8} + \frac{v}{3} > v$, leading to $S(I') > v$.

If $\frac{5v}{3} \geq p_1 > \frac{4v}{3}$.

In this case, we first repack Lay_{i_1} by calling two times $CreateLayer(Lay_{i_1}, v)$. Let Lay'_{i_1} and Lay''_{i_1} be respectively the first and the second new layers. We repack these layers at level 0, with Lay'_{i_1} left justified and Lay''_{i_1} right justified, with the narrowest rectangles on the top. As there no huge rectangles remaining, we have $H(Lay'_{i_1}) > \frac{2v}{3}$, implying $H(Lay''_{i_1}) \leq v$.

In this case, the $Merge(X)$ procedure creates a shelf sh using $CreateShelf(X, x)$ where $x = av(v)$, and packs it at level v, right justified. Notice that when $X \subset L_H$, x can be lower than 1 as there may be some extra high rectangles at level v.

The algorithm makes the following calls :

1. $Merge(L_{XH}^1)$
2. $last = Greedy\_Pack(L_{XH}^1, seq_{XH})$
3. $Greedy\_Pack(L_H^1, seq_H)$
4. $Add_2(L_H^1, S_{i_{last}})$
5. $Merge(L_H^1)$

According to Lemma 9.35, we get $p_1 + p_2 + p_3 > 3v + \frac{4v}{3} + \frac{v}{6}$. If L_{XH}^1 is not empty after step 1, then $S(\pi_0^{N_1} \cup \pi_0^{N_1-1}) > (p_1+p_2+p_3)\frac{1}{v} + S(sh) > v + \frac{4v}{9} + \frac{v}{18} + \frac{v}{2} > 2v$. Then step 2, 3 and 4 must pack all the remaining rectangles according to Lemma 9.29.

If L_{XH}^1 is empty after step 1, then step 3 packs at least a surface v in each of the last N_2 strips. Then, if L_H^1 is not empty after step 4 we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1} \cup L_H^1 \cup L_{XH}^1) > v + \frac{4v}{9} + \frac{v}{18} + \frac{v}{2} > 2v$.

If $\frac{4v}{3} \geq p_1 > v$.

In this case we also repack the wide rectangles, but not in every sub-cases. Let us describe the $Merge(X)$ procedure in this case (see Figure 9.34 Page 140 for an example of $Merge(X)$). If $X \subset L_{XH}$ (and $X \neq \emptyset$), $Merge(X)$ repacks Lay_{i_1} by calling two times $CreateLayer(Lay_{i_1}, v)$. Let Lay'_{i_1} and Lay''_{i_1} be respectively the first and the second new layers. $Merge$ repacks these layers at level 0, with Lay'_{i_1} left justified and Lay''_{i_1} right justified, with the narrowest rectangles on the top. As there no huge rectangles remaining, we have $H(Lay'_{i_1}) > \frac{2v}{3}$, implying $H(Lay''_{i_1}) \leq \frac{2v}{3}$. Then, $Merge(X)$ creates a shelf sh using $CreateShelf(X, 1)$ and packs it at level v, **left** justified.

If $X \subset L_H$, two cases are possible as Lay_{i_1} had maybe been already repacked by a previous call to $Merge$. If Lay_{i_1} is already repacked, $Merge(X)$ creates two shelves sh_1 and sh_2 using $CreateShelf(X, x_1)$ and $CreateShelf(X, x_2)$ respectively, where $x_1 = av(\frac{4v}{3})$ and $x_2 = Min(x_1, av(\frac{2v}{3}))$. Then sh_i is packed right justified at level $\frac{6-2iv}{3}$. If Lay_{i_1} is not already repacked (then S_{N_1} does not contain any extra high rectangle),

$Merge(X)$ creates three shelves shi (for $1 \leq i \leq 3$) using $CreateShelf(X, x_i)$, where $x_i = av(\frac{(6-2i)v}{3})$. Then, shi is packed at level $\frac{(6-2i)v}{3}$, right justified. Notice that if sh_3 is not empty then $W(sh_2) + W(sh_3) > x_2$ as rectangles of sh_3 were not included in sh_2.

The algorithm makes the following calls :

1. $last = Greedy\_Pack(L^1_{XH}, seq_{XH})$
2. $Merge(L^1_{XH})$
3. $Merge(L^1_H)$
4. $Greedy\_Pack(L^1_H, seq_H)$
5. $Add_2(L^1_H, S_{i_{last}})$

According to Lemma 9.35, we have $p_1 + p_2 + p_3 > 3v + v + (v - \frac{2}{k}) > 4v + \frac{v}{3}$ as $k > 3$ in this case (remember that we assume that there is no huge rectangles in the last two layers of phase 1).

Let us suppose first that L^1_{XH} is not empty after step 2. Then after step 1 each of the last N_2 strips is filled with an area strictly larger than v. Let $\bar{L_{XH}}^1$ denote L^1_{XH} after step 1 and $\bar{\pi_0}^{N_1}$ denote the set packed in S_{N_1} before step 2. We have $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup \bar{L_{XH}}^1) > \frac{4v}{3} + \frac{v}{9} + \frac{2v}{3} > 2v$, leading to $S(I') > Nv$.

If L^1_{XH} is empty after step 2, two cases are possible according to what is packed in S_{N_1}. If no extra high rectangle is packed in S_{N_1}, then Lay_{i_1} is not repacked by $Merge$. After step 3, if L^1_H is not empty we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1}) > S(sh_1) + (W(sh_2) + W(sh_3))\frac{v}{2} + (p_1 + p_2 + p_3)\frac{1}{3} > \frac{3v}{8} + x_2\frac{v}{2} + (1 - x_2)\frac{2v}{3} + (p_1 - \frac{2v}{3} + p_2 + p_3)\frac{1}{3}$ which is decreasing in x_2. For $x_2 = \frac{2}{3}$ we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1}) > 2v$, and according to Lemma 9.29 step 4 and 5 must pack all the remaining rectangles.

If a shelf sh of extra high rectangles is packed in S_{N_1}, then Lay_{i_1} is repacked by $Merge$. Before step 3 the area packed in the last N_2 strips is already strictly larger than v. Let us analyze what happen if L^1_H is not packed after step 3. Let $\bar{\pi_0}^{N_1}$ denote the set of packed rectangles before step 1. We proceed by case analysis according to $W(sh)$. If $W(sh) \leq \frac{1}{3}$ then x_2 is the available width at level $\frac{2v}{3}$, and we get $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup sh \cup L^1_H) > (1 - x_2)\frac{2v}{3} + (p_1 - \frac{2v}{3} + p_2 + p_3)\frac{1}{3} + x_2\frac{v}{2} + \frac{1-W(sh)}{2}\frac{v}{2} + W(sh)\frac{2v}{3}$ which is a decreasing function of x_2. For $x_2 = \frac{2}{3}$ we get $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup sh \cup L^1_H) > \frac{13v}{9} + \frac{v}{3} + \frac{1-W(sh)}{2}\frac{v}{2} + W(sh)\frac{2v}{3}$, which is an increasing function of $W(sh)$. Replacing $W(sh)$ by 0 we get $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup sh \cup L^1_H) > 2v$. If $W(sh) > \frac{1}{3}$ then we only need to use that $W(L^1_H) > 1 - W(sh)$. Thus we have $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup sh \cup L^1_H) > \frac{13v}{9} + W(sh)\frac{2v}{3} + (1 - W(sh))\frac{v}{2}$, leading for $W(sh) = \frac{1}{3}$ to $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup sh \cup L^1_H) > 2v$.

If $v \geq p_1 > \frac{2v}{3}$.

Let us first describe the $Merge(X)$ in this case. If $X \subset L^1_{XH}$, $Merge$ creates two shelves sh_1 and sh_2 using $CreateShelf(X, 1)$ and $CreateShelf(X, x_2)$ where $x_2 = av(0)$. Then, sh_i is packed right justified at level $(2 - i)v$. If $X \subset L^1_H$, $Merge$ creates two shelves sh_1 and sh_2 using $CreateShelf(X, x_1)$ and $CreateShelf(X, x_2)$ where $x_i = av(\frac{v(2i-1)}{3})$. Then, sh_i is packed at level $\frac{v(2i-1)}{3}$, right justified.

The algorithm makes the following calls :

1. $Merge(L^1_{XH})$
2. $last = Greedy\_Pack(L^1_{XH}, seq_{XH})$

3. $Greedy\_Pack(L_H^1, seq_H)$
4. $Add_2(L_H^1, S_{i_{last}})$
5. $Merge(L_H^1)$

According to Lemma 9.35, we have in this case $p_1 + p_2 + p_3 > 4v$. If step 1 creates two shelves of extra high rectangles, then (after step 1) $W(\pi_0^{N_1} \cap L_{XH}^1) > 1$ and we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1}) > \frac{4v}{3} + \frac{2v}{3} > 2v$. Thus step 2, 3 and 4 must pack the remaining rectangles according to Lemma 9.29. Otherwise, L_{XH}^1 is completely packed (in one shelf) in S_{N_1}. Let us assume that L_H^1 is not packed after step 5. The surface packed in the last N_2 strips is strictly larger than v. Let $\bar{L_H}^1$ be L_H^1 just before step 5. If $\bar{L_H}^1$ is not packed by step 5 we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1} \cup \bar{L_H}^1 \cup L_{XH}^1) > (1-x_1)\frac{v}{3} + (p_1 - \frac{v}{3} + p_2 + p_3)\frac{1}{3} + \frac{x_1}{2}\frac{v}{2} + \frac{v}{2}$, which is decreasing in x_1. Thus, for $x_1 = \frac{2}{3}$ we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1} \cup \bar{L_H}^1 \cup L_{XH}^1) > \frac{4v}{3} + \frac{v}{6} + \frac{v}{2} = 2v$.

If $\frac{2v}{3} \geq p_1$.

If $k \geq 2$, then we get $p_1 + p_2 + p_3 \geq 4v$, as none of the rectangles of Lay_{i_1} fit in the previous layers. Thus, the analysis is the same as in the previous case. We now assume that $k = 1$, meaning that there is only one wide rectangle r_1 of height h_1 and width w_1.

Let us describe the $Merge(X)$ procedure in this case. If $X \subset L_{XH}$, $Merge(X)$ creates two shelves sh_1 and sh_2, using $CreateShelf(X, x_1)$ and $CreateShelf(X, 1)$ respectively, where $x_1 = av(0)$. Then sh_i is packed at level $(i-1)v$, right justified.

If $X \subset L_H$, $Merge(X)$ creates three shelves sh_i for $1 \leq i \leq 3$ using $CreateShelf(X, x_i)$, where $x_i = av(\frac{v(6-2i)}{3})$. Then sh_i is packed at level $\frac{v(6-2i)}{3}$, with sh_1 and sh_2 left justified, and sh_3 right justified.

The algorithm makes the following calls :

1. $last = Greedy\_Pack(L_{XH}^1, seq_{XH})$
2. $Merge(L_{XH}^1)$
3. $Merge(L_H^1)$
4. $Greedy\_Pack(L_H^1, seq_H)$
5. $Add_2(L_H^1, S_{i_{last}})$

Let us first bound $S(Lay_{i_1-2} \cup Lay_{i_1-1} \cup r_1)$. As r_1 did not fit in Lay_{i_1-2} and Lay_{i_1-1}, p_2 and p_3 are greater than $2v - h_1$. Moreover, as no rectangle of L_B is in Lay_{i_1-1} we get $p_2 > 4h_1$. Thus $S(Lay_{i_1-2} \cup Lay_{i_1-1} \cup r_1) > max(\frac{2(2v-h_1)}{3} + w_1h_1, \frac{2v-h_1}{3} + \frac{4h_1}{3} + w_1h_1)$. Notice that the left part of the max is decreasing in h_1 as $w_1 \leq \frac{1}{2}$. Thus, replacing h_1 by $\frac{2v}{5}$ we get $S(Lay_{i_1-2} \cup Lay_{i_1-1} \cup r_1) > \frac{16v}{15} + \frac{2w_1v}{5}$.

Let us suppose first that L_{XH}^1 is not empty after step 2. Then after step 1 each of the last N_2 strips is filled with an area strictly larger than v. Let $\bar{L_{XH}}^1$ denote L_{XH}^1 after step 1 and $\bar{\pi_0}^{N_1}$ denote the set packed in S_{N_1} before step 2. We have $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup \bar{L_{XH}}^1) > \frac{16v}{15} + \frac{2w_1v}{5} + \frac{1-w_1}{2}\frac{2v}{3} + \frac{2v}{3}$ which is increasing in w_1. For $w_1 = \frac{1}{3}$ we get $S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup \bar{L_{XH}}^1) > 2v$, leading to $S(I') > Nv$.

If L_{XH}^1 is empty after step 2, two cases are possible according to what is packed in S_{N_1}. Let us first assume that no extra high rectangle is packed in S_{N_1}. After step 3, if L_H^1 is not empty we get $S(\pi_0^{N_1} \cup \pi_0^{N_1-1}) > \frac{16v}{15} + \frac{2w_1v}{5} + \frac{6v}{8} + \frac{1-w_1}{2}\frac{v}{2}$ which is greater than

$2v$ for $w_1 = \frac{1}{3}$. Then, step 4 and 5 must pack all the remaining rectangles according to Lemma 9.29.

We now assume that some extra high rectangles are packed in S_{N_1}. In this case, the area packed before step 3 in the last N_2 strips is already strictly larger than v. Let $\bar{\pi_0}^{N_1}$ denote the set of packed rectangles before step 1. We proceed by contradiction by supposing that L_H^1 is not packed after step 3.

If only one shelf sh_1 of extra high rectangles is packed in S_{N_1}, we get

$$\begin{aligned} S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup L_H^1 \cup sh_1) &> \frac{16v}{15} + \frac{2w_1v}{5} + \frac{3v}{8} + (1 - W(sh_1))\frac{v}{2} + W(sh_1)\frac{2v}{3} \\ &> \frac{16v}{15} + \frac{2w_1v}{5} + \frac{7v}{8} \\ &> 2v \end{aligned}$$

If two shelves sh_1 and sh_2 of extra high rectangles are packed in S_{N_1}, two cases are possible according to $W(sh_2)$. If $W(sh_2) > \frac{2}{3}$, we get

$$\begin{aligned} S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup L_H^1 \cup sh_2 \cup sh_1) &> \frac{16v}{15} + \frac{2w_1v}{5} + (1 - W(sh_2))\frac{v}{2} + W(sh_2)\frac{2v}{3} \\ &\quad + \frac{1 - w_1}{2}\frac{2v}{3} \\ &> \frac{16v}{15} + \frac{2w_1v}{5} + \frac{v}{6} + \frac{4v}{9} + (1 - w_1)\frac{v}{3} \\ &> 2v \end{aligned}$$

If $W(sh_2) \leq \frac{2}{3}$, we get

$$\begin{aligned} S(\bar{\pi_0}^{N_1} \cup \pi_0^{N_1-1} \cup L_H^1 \cup sh_2 \cup sh_1) &> \frac{16v}{15} + \frac{2w_1v}{5} + \frac{3}{2}(1 - W(sh_2))\frac{v}{2} + W(sh_2)\frac{2v}{3} \\ &\quad + \frac{1 - w_1}{2}\frac{2v}{3} \\ &> \frac{16v}{15} + \frac{2w_1v}{5} + \frac{v}{4} + \frac{4v}{9} + (1 - w_1)\frac{v}{3} \\ &> 2v \end{aligned}$$

9.4.6 Complexity

According to the main steps defined in Section 9.2.2, the algorithm that preallocates I' is sufficient to get a 2-approximation. Indeed, we simply add rectangles of $I \setminus I'$ using list algorithms defined in Section 9.2.2.

We can bound the computational complexity of the algorithm using the same arguments as for the $5/2$-approximation (see Section 9.3.6).

Roughly speaking, Phase 1 runs in $\mathcal{O}(nlog(n) + Nn)$ as for each strip creating a layer can be done in $\mathcal{O}(n)$ (by initially sorting wide rectangles according to their heights). Phase 2 also runs in $\mathcal{O}(nlog(n) + Nn)$ as for each strip creating a constant number of shelves (generally two or three) can be done in $\mathcal{O}(n)$ (by initially sorting high and extra high rectangles according to their widths). Thus, taking into account the repetitions due to the binary search on v, this algorithm can be implemented in $O(log_2(nh_{max})n(N + log(n)))$.

Bibliographie

[Bougeret et al., 2009] Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., and Trystram, D. (2009). Approximation algorithm for multiple strip packing. In *Proceedings of the 7th Workshop on Approximation and Online Algorithms (WAOA)*.

[Hochbaum and Shmoys, 1988] Hochbaum, D. and Shmoys, D. (1988). A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM Journal on Computing*, 17(3) :539–551.

[Schwiegelshohn et al., 2008] Schwiegelshohn, U., Tchernykh, A., and Yahyapour, R. (2008). Online scheduling in grids. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10.

[Steinberg, 1997] Steinberg, A. (1997). A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26 :401.

[Ye et al., 2009] Ye, D., Han, X., and Zhang, G. (2009). On-Line Multiple-Strip Packing. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications (COCOA)*, page 165. Springer.

[Zhuk, 2006] Zhuk, S. (2006). Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1) :73–85.

Chapitre 10

A fast 5/2-approximation algorithm for hierarchical scheduling [Bougeret et al., 2010a]

Marin bougeret*‡, Pierre-François Dutot*, Klaus Jansen†, Christina Otte† and Denis Trystram*

*LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

†Department of Computer Science, Christian-Albrechts-University, Kiel, Germany
kj, cot@informatik.uni-kiel.de

‡This work is supported by DGA-CNRS

†Work supported by EU project "AEOLUS : Algorithmic Principles for Building Efficient Overlay Computers", EU contract number 015964, and DFG project JA612/12-1, "Design and analysis of approximation algorithms for two- and threedimensional packing problems"

Abstract

We present in this article a new approximation algorithm for scheduling a set of n independent rigid (meaning requiring a fixed number of processors) jobs on hierarchical parallel computing platform. A hierarchical parallel platform is a collection of k parallel machines of different sizes (number of processors). The jobs are submitted to a central queue and each job must be allocated to one of the k parallel machines (and then scheduled on some processors of this machine), targeting the minimization of the maximum completion time (makespan). We assume that no job require more resources than available on the smallest machine.

This problem is hard and it has been previously shown that there is no polynomial approximation algorithm with a ratio lower than 2 unless $P = NP$. The proposed scheduling algorithm achieves a $\frac{5}{2}$ ratio and runs in $O(log(np_{max})knlog(n))$, where p_{max} is the maximum processing time of the jobs. Our results also apply for the Multi Strip Packing problem where the jobs (rectangles) must be allocated on contiguous processors.

10.1 Introduction

Context and motivation. The evolution of the technology over the last past years leads to the emergence of new types of parallel and distributed platforms. Many of these new computing systems are hierarchical in nature. Such computing systems may be composed of several parallel machines, each having a certain number of processors, or composed of several clusters, each having a certain number of computers [4]. Thus, we make no distinctions between machines/clusters on one side, and between processors/computers on the other side.

In order to fully exploit the large number of available resources for reaching the best performances, we need to revisit the classical resource management algorithms taking into account new features induced by the hierarchical hardware. In such platforms, the users submit their jobs in a centralized queue. The resource manager selects the jobs and allocates them to a machine, and then the jobs are scheduled locally. We assume in this work that the machines have enough processors so as each job fits entirely in any machine. Notice that jobs are not executed on processors that belong to different machines.

In the common context of an execution by successive batches, we target on the minimization of the makespan (defined as the maximum completion time over all the processors) which is the most popular objective function [3]. We assume that the processing times of the jobs are available at the beginning (the algorithm is *clairvoyant* [3]).

Related works. Let us qualify by *regular*, the case where the machines have the same number of processors. The *irregular case* corresponds to the situation where the machines have different sizes.

There exist a lot of related works for the regular case, as it is closely related to the multiple strip packing problem. Indeed, the only difference is that in multiple strip packing (MSP), the jobs are considered as rectangles and thus, they must be allocated on "consecutive" processors. This problem (and the non-contiguous version) are strongly $\mathcal{NP}$-hard, and Zhuk [10] (and later Schwiegelshohn et al. [7]) showed that there is no approximation algorithm with absolute ratio better than 2 unless $P = NP$, even for 2 machines. The proof is a gap reduction from 2-partition problem. The main related positive results for MSP are a $2+\epsilon$-approximation in [9] whose cost is doubly exponential in $\frac{1}{\epsilon}$, a (costly) 2-approximation and an AFPTAS in [1]. Some variants have been investigated (like in [6]) where additional constraints come from pre-allocation.

Concerning the hierarchical scheduling problem in the irregular case, one of the most relevant work is the result of Schwiegelshohn, Tchernykh and Yahyapour [7]. The authors address a version of this problem where a job may not fit everywhere, meaning that a job may requires more processors than the total number of processors of some machines. They provide a 3-approximation for the off-line case, and a 5-approximation when the jobs are submitted over time (on-line). Their algorithms handle non-clairvoyant jobs (meaning that the processing time of a job is only known at the end of its execution), but do not easily apply for MSP. They also emphasize that the performance of the classical Garey and Graham's list scheduling algorithm is significantly worsened in hierarchical environments. Finally, (as we will see in remark 10.11), it is possible to derive a better approximation ratio using the remark in [9] that leads

to a $2+\epsilon$ ratio (in time polynomial in $\frac{1}{\epsilon}$) for the regular case. However, the cost of the induced algorithm is considerably too large, for instance taking $\epsilon = \frac{1}{2}$ leads to a cost in $\Omega(n^{60})$!

Contribution. Our main contribution is a new $\frac{5}{2}$-approximation (almost) greedy algorithm (called $FAHS$ for fast approximation hierarchical scheduler) for scheduling a set of (clairvoyant) rigid jobs on parallel machines that have different number of processors, and where every job fits everywhere. The algorithm runs in $O(log(np_{max})knlog(n))$. This result also applies for MSP.

10.2 Preliminaries

10.2.1 Problem statement

The problem studied in this paper is to schedule n jobs on a parallel system composed of k parallel machines. Machine M_i (for $1 \leq i \leq k$) has m_i processors. We assume that $m_i \leq m_{i+1}$ for any i. A job J_j is characterized by the number of processing resources needed for its completion, denoted by $q_j \in \mathbb{N}^*$ (sometimes called the *degree of parallelism*) and its corresponding processing time $p_j \in \mathbb{N}^*$. The q_j processors must be allocated on the same machine. We assume that every jobs fits everywhere, meaning that $max_j q_j \leq m_1$. We denote by $w_j = p_j q_j$ the *work* of job J_j. The objective is to minimize the maximum completion time over all the machines (called the *makespan* and denoted by C_{max}). Given a fixed instance, we denote by C^* the optimal makespan for this instance.

10.2.2 Other notations

Given a schedule, we denote by $u_i(t)$ the utilization of M_i at time t, which is the sum of the required number of processors of all the jobs that are running on M_i at time t. We say that job J_{j_1} is higher than job J_{j_2} if $q_{j_1} \geq q_{j_2}$. We extend the p_j, q_j, w_j previous notations to $P(X)$, $Q(X)$ and $W(X)$ where X is a set of jobs and the capital letter denotes the sum of the corresponding quantity of all the jobs in the set (for instance $P(X) = \Sigma_{J_j \in X}\, p_j$).

We call *shelf* a set of jobs that are scheduled on the same machine and finish at the same time. Scheduling a shelf at time t means that all the jobs of the shelf finish at time t. This reverse definition of shelf (where usually jobs start simultaneously) is used because some machines are filled from right to left. Using v as a guess for the optimal makespan C^*, let $L = \{J_j | p_j > \frac{v}{2}\}$ denote the set of long jobs, and let $H_i = \{J_j | q_j > \frac{m_i}{2}\}$ denote the set of high jobs for M_i, for $1 \leq i \leq k$.

10.2.3 Useful results

We present briefly in this section some classical results that we will use later. Let us recall first the well known Highest First (HF) list algorithm that, given a set of jobs and one machine, starts from time 0 and schedules at any time the highest job which fits. Secondly, let us state the following result that was established by Steinberg [8].

Notice that even if this result is stated in term of rectangle packing, it remains valid for parallel jobs (without the contiguous constraint).

Theroem 10.1 ([8]). *Let $L = \{r_1, \ldots, r_n\}$ be a set of rectangles. Let w_j, h_j and s_j denote the width, the height and the surface of r_j. Let $S(L) = \Sigma_{j=1}^{n} s_j$, $w_{max} = max_j w_j$ and $h_{max} = max_j h_j$. If*

$$w_{max} \leq u, h_{max} \leq v \text{ and } 2S(L) \leq uv - max(2w_{max} - u, 0)max(2h_{max} - v, 0)$$

then it is possible to pack L (in time $O(nlog^2(n)/log(log(n)))$ in a rectangular box of width u and height v.

We are now ready to describe the $FAHS$ algorithm.

10.3 Algorithm

As we target a 5/2 ratio, it is convenient to consider that we have in each machine an empty available rectangular area of height m_i and length $5C^*/2$. Moreover, we want to particularly take care of the jobs that have a processing time greater than $C^*/2$, since these jobs are harder to schedule. As C^* is unknown, we use the classical dual approximation technique [5], and we denote by v the current guess of C^*. According to this technique, our goal is to provide the $FAHS$ (Fast Approximation Hierarchical Scheduler) algorithm that either schedules all the jobs with a makespan lower than $5v/2$ or rejects v, implying that $v < C^*$. For the sake of simplicity, we do not mention in the algorithm the "reject" instruction. Thus, we consider throughout the paper that $v \geq C^*$, and it is implicit that if one of the claimed properties is wrong during the execution, the considered v should be rejected.

$FAHS$ is described in detail in Algorithm 3. The principle is to fill the machines from the smallest (M_1) to the largest one. As long as it is possible, we aim at scheduling on each M_i a set of jobs whose total work is greater than $m_i v$. The algorithm is designed such that the only reason for failing (meaning that it does not reach $m_i v$) is the lack of jobs. Thus, if a failure occurs, there will be only a few amount of jobs that will be straightforward to schedule.

In order to schedule a work of (at least) $m_i v$, we first intent to schedule a set X of jobs of H_i such that $P(X)$ is greater than $2v$ (phase 1). If it fails, we schedule as many jobs of H_i as possible (again in phase 1), with a strict limit $P(X) < 3v/2$ to have enough space to pack a shelf of long jobs between $3v/2$ and $5v/2$. Then, we select (in phase 2) some jobs of $I \backslash H_i$, targeting a total work of at least $m_i v$.

These selected jobs are then scheduled in phase 3. If the work goal $m_i v$ is reached we schedule the considered jobs in M_i (using the *add* algorithm in Algorithm 4 in case 1 or HF in case 2), otherwise (case 3) there is by construction only a few amount of unscheduled jobs, and they are scheduled using the *pack* procedure described in Algorithm 5.

Remark 10.2 (Notation). *For any set X considered in the algorithm or in the proof, we use the $\tilde{X}$ notation to denote the set X during the execution of the algorithm ($\tilde{X} \subset X$). We consider that a job is removed from the instance when it is scheduled.*

Algorithm 3 Fast Approximation Hierarchical Scheduler ($FAHS$)

for all i, $\tilde{H}_i \leftarrow H_i$ and $\tilde{L} \leftarrow L$
for $i = 1$ to k **do**
——————————- phase 1 ——————————-
if $\tilde{H}_i \cap \tilde{L} \neq \emptyset$ **then**
• $J_{j_i} \leftarrow$ highest job of $\tilde{H}_i \cap \tilde{L}$
• schedule J_{j_i} at time 0 and remove it from $\tilde{H}_i \cap \tilde{L}$
else
• $J_{j_i} \leftarrow$ "dummy" job (with $p_{j_i} = q_{j_i} = 0$)
end if
if $p_{j_i} + P(\tilde{H}_i \setminus \tilde{L}) \geq 2v$ **then**
• $target \leftarrow 2v$
else
• $target \leftarrow v$
end if
• schedule sequentially (in any order) jobs of $\tilde{H}_i \setminus \tilde{L}$ (and remove them from $\tilde{H}_i$) until one of them ends later than $target$ or until $(\tilde{H}_i \setminus \tilde{L} = \emptyset)$
• $High \leftarrow$ jobs scheduled on M_i
• reschedule jobs of $High$ in non-increasing order of height
if $target = v$ **then**
——————————- phase 2 ——————————-
• $Select \leftarrow \emptyset$
• add to $Select$ some jobs of $\tilde{I} \setminus \tilde{H}_i$ (and remove them from $\tilde{I}$) in non increasing order of their work, until $(W(High) + W(Select) \geq m_i v)$ or until $(\tilde{I} \setminus \tilde{H}_i = \emptyset)$
——————————- phase 3 ——————————-
if $W(High) + W(Select) > \frac{5}{4} m_i v$ **then** //**case 1**
• schedule $Select$ using $add(i, Select)$
else if $\frac{5}{4} m_i v \geq W(High) + W(Select) \geq m_i v$ **then** //**case 2**
• reschedule jobs $High \cup Select$ on M_i using HF algorithm
else //**case 3**
if i=k **then**
• reschedule $High \cup Select \cup \tilde{H}_i$ on M_i using HF algorithm (and remove jobs from $\tilde{H}_i$)
else
• reschedule $High \cup Select$ on M_i using HF algorithm
• schedule $\tilde{H}_i$ on $\{M_{i+1}, \ldots, M_k\}$ using $pack(i, \tilde{H}_i)$ (and remove jobs from $\tilde{H}_i$)
end if
• break //all the jobs are scheduled
end if
end if
end for

Algorithm 4 *add*$(i, Select)$

- create a shelf Sh with the highest jobs of $Select$ and schedule Sh at time $\frac{5v}{2}$
- reschedule Sh such that the shortest jobs are on the top (as depicted figure 10.5)

if one job $J_1 \in Select$ remains **then**

- $t_1 \leftarrow$ smallest t such that $u_{M_i}(t) + q_1 \leq m_i$
- schedule J_1 at time t_1

end if

if two jobs $\{J_1, J_2\} \subset Select$ remain **then**

- create a shelf Sh' using J_1 and J_2 and schedule Sh' at time $\frac{3v}{2}$

end if

Algorithm 5 *pack*$(i, \tilde{H}_i)$

if $\tilde{H}_i \cap L = \emptyset$ **then**

- schedule jobs of $\tilde{H}_i$ sequentially on M_{i+1} from 0 (we know that $i < k$)

else

- $x \leftarrow i$

while $\tilde{H}_i \cap L \neq \emptyset$ **do**

- $x \leftarrow x + 1$
- $Sh \leftarrow \emptyset$

while $((Q(Sh) \leq m_x)$ and $(\tilde{H}_i \cap L \neq \emptyset))$ **do**

- add to Sh a job J_j from $\tilde{H}_i \cap L$

end while

- schedule all the jobs of $Sh \setminus \{J_j\}$ at time 0

if $Q(Sh) > m_x$ **then**

- schedule J_j at time v

else

- schedule J_j at time 0

end if

end while

- schedule all the remaining jobs (which all belong to $\tilde{H}_i \setminus L$) sequentially on M_k from v

end if

Let us now quickly analyze the running time of $FAHS$.

Remark 10.3. *$FAHS$ runs in $O(log(np_{max})knlog(n))$.*

Proof Phase 1 and phase 2 run in $O(nlog(n))$ as the jobs are sorted according to their height at the end of phase 1 and according to their work before starting the phase 2. Phase 3 runs in $O(nlog(n))$ because of the HF algorithm. Thus, for a fixed value of v, the cost of $FAHS$ is in $O(knlog(n))$, and the dichotomic search on v can be done in $log(np_{max})$ since np_{max} is an upper bound of the optimal. □

10.4 Analysis of the algorithm

Notice first that the case where $target = 2v$ is clear as in this case the schedule has a makespan lower than $\frac{5v}{2}$, and the scheduled work is greater than $2v\frac{m_i}{2} = m_iv$. Thus, we will focus on the second part of $FAHS$, where $target = v$. We will first prove in lemma 10.4, 10.6 and 10.8 that (if $v \geq C^*$, as usual) it is possible to schedule the considered set of jobs with a makespan lower than $\frac{5v}{2}$ in the three cases of phase 3. Then, the final proof of $FAHS$ is derived in theorem 10.10.

Lemma 10.4 (Feasibility of case 1)**.** *Any call to add$(i, Select)$ produces a valid schedule of makespan lower than $\frac{5v}{2}$.*

Proof Notice first that calling the *add* procedures implies that $target = v$. We assume that the sets $High$ and $Select$ have been constructed in phase 1 and 2, implying $W(High) + W(Select) > \frac{5}{4}m_iv$. Let $p = |Select|$ and $Select = \{J'_1, \ldots, J'_p\}$, with $w'_{j+1} \leq w'_j$ for all j, as stated in Algorithm 3. Let us start with the following properties :

1) $P(High) \leq \frac{3}{2}v$
2) for all $J'_j \in Select$, $w'_j > \frac{m_iv}{4}$, implying $J'_j \in L$ and $0 \leq p \leq 4$
3) $\forall X \subset Select, W(High) + W(Select \setminus X) < vm_i$

The first property is true because $target = v$, and the jobs of $High$ scheduled in phase 1 have a processing time lower than $\frac{v}{2}$. The second point comes from the fact that $W(Select \setminus \{J'_p\}) < m_iv$ and $W(Select) > \frac{5}{4}m_iv$. This implies that $w'_p > \frac{m_iv}{4}$ and property 2. The third point is true by definition of phase 2.

We start by creating a shelf Sh with the higher jobs of $Select$ (as depicted figure 10.5). We schedule Sh at time $\frac{5v}{2}$ (remind that it means that all the jobs of Sh finish at time $\frac{5v}{2}$), which is possible according to property 1. Let α denote the number of jobs in this shelf. As $Select \cap H_i = \emptyset$, we know that $\alpha \geq 2$, implying that cases where $p \leq 2$ are straightforward.

We now study the cases where $p = 3$. Let J_1 be the remaining job (we use the same notations as in Algorithm 4). Let us now analyze how the *add* procedure schedules J_1. Let us suppose by contradiction that J_1 intersects Sh. Let $t'_1 = \frac{5v}{2} - t_1 - p_1$. We get

$$\begin{aligned} W(High \cup Sh) &> t_1(m_i - q_1) + t'_1(m_i - q_1) + (Q(Sh) - (m_i - q_1))\frac{v}{2} \\ &> t_1(m_i - q_1) + (\frac{3v}{2} - t_1)(m_i - q_1) + (3q_1 - m_i)\frac{v}{2} \\ &= m_iv \end{aligned}$$

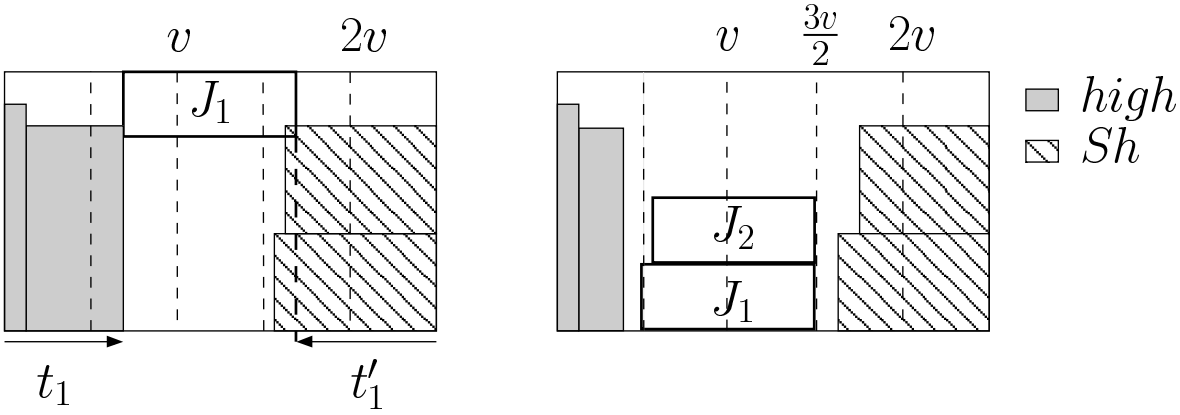

FIG. 10.5: Example for the *add* procedure. Case where $p = 3$ is depicted on the left side, and case where $p = 4$ is depicted on the right side.

The inequality $Q(Sh) \geq 2q_1$ used above is true since *add* schedules the highest jobs in the shelf. The inequality $W(High \cup Sh) > m_i v$ is a contradiction according to property 3, implying that J_1 must fit in the "staircase".

Let us now study the case where $p = 4$. Let $Select' = Select \setminus \{J'_4\}$. We have $W(Select') > 3\frac{m_i v}{4}$ (according to property 2), implying $W(High) + 3\frac{m_i v}{4} < W(High \cup Select') \leq v m_i$. Thus, we get $P(High)\frac{m_i}{2} < W(High) < \frac{m_i v}{4}$, leading to $P(High) < \frac{v}{2}$. Thus, the second shelf sh' scheduled at time $\frac{3v}{2}$ does not intersect jobs of $High$. It ends the proof of feasibility of case 1.

□

Concerning the feasibility of case 2, we prove a slightly more general statement which can be interpreted as a particular case of Steinberg's theorem [8].

Lemma 10.6 (Feasibility of case 2)**.** *Let us consider an arbitrary machine M with m processors. Let v such that for all $j \in X$, $p_j \leq v$. Let X be a set of jobs such that $W(X) \leq \alpha m v$, with $\alpha \geq 1$. Then, the HF algorithm schedule X on M with a makespan lower than $2\alpha v$.*

Proof Let C denote the makespan of the schedule, and let J_{j_0} be a job that finishes at time C. If $q_{j_0} > \frac{m}{2}$, then all the jobs J_j of X have $q_j > \frac{m}{2}$, implying $\alpha m v \geq W(X) > \frac{C}{2} m$. Let us suppose now that $q_{j_0} \leq \frac{m}{2}$. Let s denote the starting time of J_{j_0}. As there are strictly less than q_{j_0} available processors between time 0 and s, we have $W(X) > p_{j_0} q_{j_0} + s(m - q_{j_0})$, implying $s < \frac{W(X)}{m - q_{j_0}} - \frac{p_{j_0} q_{j_0}}{m - q_{j_0}}$. Thus,

$$\begin{aligned} C &\leq s + p_{j_0} \\ &\leq \frac{W(X)}{m - q_{j_0}} + p_{j_0} \frac{m - 2q_{j_0}}{m - q_{j_0}} \\ &\leq v \frac{(\alpha + 1)m - 2q_{j_0}}{m - q_{j_0}} \end{aligned}$$

As the last expression is an increasing function of q_{j_0} and $q_{j_0} \leq \frac{m}{2}$ we get $C \leq 2\alpha v$. □

Remark 10.7. *Notice that Lemma 10.6 does not apply for strip packing. However this lemma can be simply adapted using Steinberg algorithm [8], as explained in Section 10.5*

We now study the feasibility of case 3.

Lemma 10.8 (Feasibility of case 3)**.** *The pack procedure schedules all the remaining jobs.*

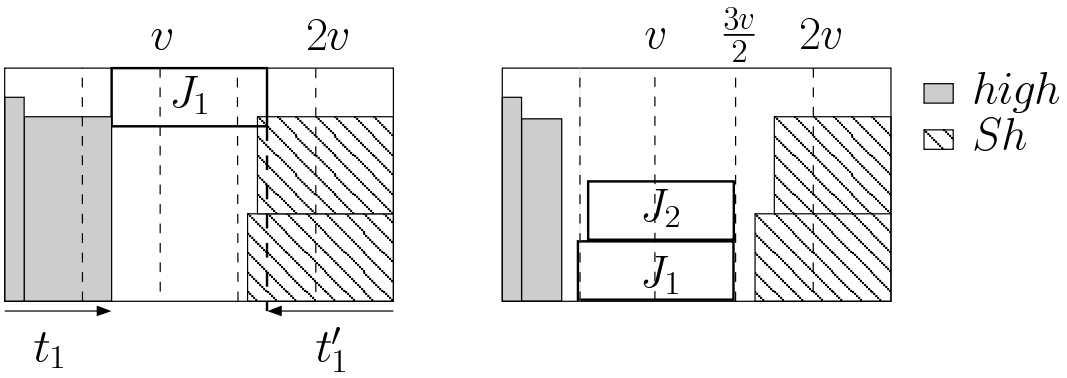

FIG. 10.9: Example for the *pack* procedure. Case 3 occurs for $i = 3$. $\tilde{H}_i^1$ is defined as in Lemma 10.8.

Proof Let i be the index of the current machine when case 3 occurs for the first time. Let $\tilde{H}_i^0$ be $\tilde{H}_i$ at line 7 of phase 1 (after setting the value of J_{j_i}) and let $\tilde{H}_i^1$ be $\tilde{H}_i$ at the beginning of phase 3. Notice that $W(High \cup Select) < m_i v$ implies that we collected all the jobs of $\tilde{I} \setminus \tilde{H}_i^1$, and thus the set of jobs that have not been scheduled on $\{M_1, \ldots, M_{i-1}\}$ is $High \cup Select \cup \tilde{H}_i^1$. If $i = k$, then $W(High \cup Select \cup \tilde{H}_i^1) \leq v m_i$ (otherwise, v is rejected), and HF is sufficient according to lemma 10.6. We consider now that $i < k$. Jobs of $High \cup Select$ are scheduled (on M_i) with a makespan lower than $\frac{5v}{2}$ using again lemma 10.6. We now prove that $pack(i, \tilde{H}_i^1)$ successfully schedules $\tilde{H}_i^1$. If $\tilde{H}_i^1 \cap L = \emptyset$, it implies that $\tilde{H}_i^0 \setminus \tilde{L} = \tilde{H}_i^0$, and thus $P(\tilde{H}_i^1) \leq P(\tilde{H}_i^0) \leq 2v - p_{j_i} \leq 2v$ because phase 3 only occurs if $target = v$. Thus, we can schedule $\tilde{H}_i^1$ sequentially on M_{i+1}.

If $\tilde{H}_i^1 \cap L \neq \emptyset$, then there is one job of $H_i \cap L$ scheduled on $M_1, \ldots, M_i$. Moreover, $P(\tilde{H}_i^1 \setminus L) \leq P(\tilde{H}_i^0 \setminus L) \leq \frac{3v}{2}$ as there was not enough jobs to fill M_i up to $2v$. We will prove that $pack(i, \tilde{H}_i^1)$ can schedule $\tilde{H}_i^1 \cap L$ on machines $\{M_{i+1}, \ldots, M_k\}$ such that no job (of $\tilde{H}_i^1 \cap L$) is scheduled after time v on M_k. Then, it will be obvious that $\tilde{H}_i^1 \setminus L$ can be added on M_k between v and $\frac{5v}{2}$.

Let us prove it by contradiction by assuming that there is a job of $\tilde{H}_i^1 \cap L$ scheduled on machine M_k after time v (meaning that we stared a second shelf of jobs of $\tilde{H}_i^1 \cap L$ on M_k).

For any $t \in [|1, \ldots, k|]$, let S_t denote the schedule constructed by $FAHS$ on machine M_t, and S_t^* be the schedule (for a fixed optimal solution) on machine M_t. Let $X = \cup_{t=1}^{i} S_t$ and $X^* = \cup_{t=1}^{i} S_i^*$. We have $Q(X \cap H_i \cap L) \geq Q(X^* \cap H_i \cap L)$ as in phase 1 we scheduled on machines $M_1 \ldots M_i$ the i highest jobs of $H_i \cap L$ (called the J_{j_t} in

the algorithm), and the optimal cannot schedule more than i jobs of $H_i \cap L$ on these machines. Moreover, we have $Q(S_t \cap H_i \cap L) > m_t$ for every t in $[|i+1, k|]$ as we started a second shelf on each of these machines. Thus, it means that $Q(\tilde{H}_i^1 \cap L) > \Sigma_{t=i+1}^{k} m_t$ which is impossible as the optimal solution had a total height of jobs of $H_i \bigcap L$ greater than $Q(\tilde{H}_i^1 \cap L)$ to schedule on $M_{i+1}, \ldots, M_k$. □

We can now complete the main proof.

Theroem 10.10. *If $v \geq C^*$, $FAHS$ schedules all the jobs with a makespan lower than $\frac{5v}{2}$.*

Proof If case 3 of phase 3 occurs, then we know according to lemma 10.8 that all the remaining jobs are scheduled in $\frac{5v}{2}$. Otherwise, we know that for every $i \in [|1, \ldots, k|]$, the $FAHS$ algorithm had a *target* equal to $2v$, or executed case 1 or case 2. This implies by construction that the area scheduled on M_i is greater than (and in this case exactly equal to) $m_i v$. Thus, there can not be unscheduled jobs at the end. □

10.5 Concluding remarks

In this paper, we presented a new algorithm for scheduling a set of clairvoyant rigid jobs in a hierarchical parallel platform, assuming that every job fits in any machine. A sophisticated analysis proved that it is a $\frac{5}{2}$-approximation, improving the best existing algorithm. There exists for this problem a lower bound equal to 2.

We now explain how to adapt the algorithms and the proofs to the MSP problem where jobs, called rectangles, must be allocated on contiguous processors. The only adaptation needed is in Lemma 10.6, where HF can be directly replaced by Steinberg's algorithm (see Section 10.2.3). Phase 1, *pack* and *add* build contiguous schedule (and thus a valid schedule for rectangles), and proofs of Lemma 10.4 and 10.8 still hold for the contiguous case. Thus, we get a $\frac{5}{2}-$approximation algorithm for MSP that runs $O(log(np_{max})knlog^2(n)/log(log(n))$.

We describe now how to achieve a $2+\epsilon$ ratio for this problem at the price of a much higher algorithm complexity. This result is an extension of the result in [9] that was established for the regular case.

Remark 10.11. *There is a $2 + 2\epsilon$-approximation that runs in $O(nlog^2(n)/log(log(n))f(k, n, \epsilon))$ where $f(k, n, \epsilon)$ is the complexity of any $1+\epsilon$ algorithm for the $Q||C_{max}$ problem with k machines and n jobs (for example $f(\epsilon) = O(kn^{\frac{10}{\epsilon^2}})$ using [5]).*

Proof We follow here the same idea as in [9]. Let I denote the instance of the irregular hierarchical problem (where every job fits everywhere). We define an instance I' of the $Q||C_{max}$ problem as follows. For each job J_J of I we associate a job J'_j of I' of processing time $p'_j = w_j$. For each machine M_i of I' we associate a machine M'_i of I' whose speed is m_i. Then, we apply a $PTAS$ (and get a schedule $S' \leq (1+\epsilon)Opt(I')$) on I'. The schedule gives a partition of the jobs of I among the different machines. Let

X'_i denote the set of jobs scheduled on M'_i in S', and let X_i denote the corresponding set of jobs (meaning the jobs of same index) in I. Let c'_i denote the completion time on M'_i. We create the schedule S of I as follows. According to the Steinberg's theorem (see Section 10.2.3), we know that for each i the set X_i can be scheduled in a "box" (created on machine M_i) of size $m_i \times 2c'_i$ as $c'_i = \frac{W(X'_i)}{m_i} = \frac{W(X_i)}{m_i}$. Thus, the makespan of S is lower than $2max_i c'_i \leq 2(1+\epsilon)Opt(I') \leq 2(1+\epsilon)Opt(I)$. □

A natural (but hard) question is to fill the gap between the new approximation ratio $\frac{5}{2}$ and the lower bound with a low cost algorithm.

Bibliographie

[1] M. Bougeret, P-F. Dutot, K. Jansen, C. Otte, and D. Trystram. Approximation algorithm for multiple strip packing. In *Proceedings of the 7th Workshop on Approximation and Online Algorithms (WAOA)*, 2009.

[2] M. Bougeret, P-F. Dutot, K. Jansen, C. Otte, and D. Trystram. A fast 5/2-approximation for hierarchical scheduling. In *Proceedings of the 16th International European Conference on Parallel and Distributed Computing (EUROPAR)*, 2010.

[3] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–34. Springer, 1997.

[4] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3) :200, 2001.

[5] D. Hochbaum and D. Shmoys. A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM J. Comput.*, 17(3) :539–551, 1988.

[6] F. Pascual, K. Rzadca, and D. Trystram. Cooperation in multi-organization scheduling. In *Proceedings of the 13th International European Conference on Parallel and Distributed Computing (EUROPAR)*, 2007.

[7] U. Schwiegelshohn, A. Tchernykh, and R. Yahyapour. Online scheduling in grids. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, pages 1–10, 2008.

[8] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26 :401, 1997.

[9] D. Ye, X. Han, and G. Zhang. On-Line Multiple-Strip Packing. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications (COCOA)*, page 165. Springer, 2009.

[10] SN Zhuk. Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1) :73–85, 2006.

Chapitre 11

An extention of the 5/2-approximation algorithm using oracle

Marin bougeret[‡], Pierre-François Dutot, and Denis Trystram

LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

[‡]This work is supported by DGA-CNRS

Abstract

In this chapter we consider the Multiple Cluster Scheduling Problem (MCSP). The MCSP corresponds to the $MSP_d^{nc/c}$ in the classification of Chapter 5* (a cluster is equivalent to a strip), where additionally cluster have different speeds. We provide a $\frac{5}{2}$-approximation algorithm (using an oracle guess) for $MSP_d^{nc/c}$, improving thus the result of Chapter 10 that requires the assumption that all the jobs fit on all the clusters. Moreover, this result also hold for clusters having same size but different speeds (*i.e.* for $MSP_r^{nc/c}$ where clusters have different speeds). Notice that our algorithm even apply for "contiguous scheduling", where jobs must be allocated on contiguous indexes of processors (*i.e.* jobs are rectangles).

*We did not change the MCSP notation throughout this chapter in order to faithfully reproduce the corresponding report research

11.1 Introduction

In the grid computing paradigm, several clusters share their computing resources in order to distribute the workload. Each cluster is a set of identical processors connected by a local interconnection network. Jobs are submitted in successive packets called batches. The objective is to minimize the time when all the jobs of a batch are completed, then, the next batch of jobs can be processed. Many such computational grid systems are available all over the world, and the efficient management of the resources is a crucial problem.

Let us now introduce the Multiple Cluster Scheduling Problem (MCSP) more formally. We are given n parallel jobs $J = \{J_1, \ldots, J_n\}$ and N clusters $Cl_1, \ldots, Cl_N$. Each job J_j is described by a processing time p_j and a width q_j (the number of required processors). The area of a job J_j is $q_j p_j$, consequently the total area of a set of jobs X is defined as $A(X) := \sum_{J_j \in X} p_j q_j$. In the same way we define $Q(X) := \sum_{J_j \in X} q_j$ and $P(X) := \sum_{J_j \in X} p_j$. A cluster Cl_ℓ has m_ℓ identical processors, each of them running with speed s_ℓ. A job J_j is only allowed to be scheduled within one cluster, its processing time in cluster Cl_ℓ is $p_j^\ell := \frac{p_j}{s_\ell}$ if $q_j \leq m_\ell$ else $p_j^\ell = \infty$. We assume the clusters to be sorted by non-decreasing order of their number of processors (or machines), i.e. $m_1 \leq m_2 \leq \ldots \leq m_N$. Furthermore we assume $\min_\ell s_\ell = 1$ and define $p_{\max} := \frac{\max_j p_j}{\min_\ell s_\ell} = \max_j p_j$. The objective is to find a non-preemptive schedule of the jobs into the clusters minimizing the makespan, i.e. the latest finishing time of a job.

MCSP is closely related to Multiple Strip Packing (MSP) problem where a cluster can be seen as a strip and a job as a rectangle. However, there is an additional constraint in MSP, since packing a rectangle corresponds to schedule a job in MCSP using consecutive addresses of processors (in other words, the allocation must be contiguous). Thus, results for MCSP do not necessarily apply to MSP as the schedule may be not contiguous. Obviously, a solution for MSP is a (feasible) solution for MCSP. However, approximation ratios are not preserved because the optimal value for MSP is an upper bound of the optimal value for MCSP.

Related work

In the case if $N = 1$ the problem is identical to scheduling n parallel jobs on m identical machines. Here the contiguous case corresponds directly to strip packing. For the case that the number of machines is polynomially bounded in the number of jobs a $(1.5 + \epsilon)$-approximation for the contiguous case and a $(1 + \epsilon)$-approximation for the non-contiguous case where given in [Jansen and Thöle, 2008]. For strip packing Coffman *et al.* gave in [Coffman Jr et al., 1980] an overview about performance bounds for shelf-orientated algorithms as $NFDH$ (Next Fit Decreasing Height) and $FFDH$ (First Fit Decreasing Height), that have an absolute ratio of 3, and 2.7, respectively. Schiermeyer [Schiermeyer, 1994] and Steinberg [Steinberg, 1997] presented independently an algorithm for strip packing with absolute ratio 2. This result was recently improved by Harren *et al.*, in [Harren et al., 2010] they presented an algorithm with absolute ratio $5/3+\epsilon$. A further important result is an AFPTAS for strip packing with additive constant $\mathcal{O}(1/\epsilon^2 h_{\max})$ of Kenyon and Rémila [Kenyon and Rémila, 2000], where $h_{\max}$ denotes the height of the tallest rectangle (i.e. the length of the longest job). This constant was improved by Jansen and Solis-Oba, who presented in [Jansen and Solis-Oba, 2007] an APTAS with additive constant $h_{\max}$.

For MCSP with clusters of identical sizes and speeds, i.e. $s_\ell = 1$ and $m_\ell = m$ for all $\ell \in \{1, \ldots, N\}$, Zhuk [Zhuk, 2006] showed that MSP has no polynomial time approximation algorithm (unless $P = NP$) with abolute ratio better than 2. The remark of [Ye et al., 2009] that consists in applying a $PTAS$ to balance the area of the jobs among the clusters, provide a $2+2\epsilon$-appoximation algorithm whose complexity is in $O(f(\epsilon)g)$, where f is the complexity of a $PTAS$ for the classical $P||C_{max}$ problem with precision ϵ, and g the complexity of Steinberg's algorithm [Steinberg, 1997]. For the non-contiguous case, we proposed recently a low cost $5/2$-approximation in [Bougeret et al., 2010b].

For MCSP with clusters of different sizes but identical speeds, Schwiegelshohn *et al.* [Schwiegelshohn et al., 2008] achieved ratio 3 for a version of parallel job scheduling in grids without release times, and ratio 5 with release times. We recently get a fast $5/2$-approximation in [Bougeret et al., 2010a] that only apply when all the jobs fit in all the clusters, i.e. when $\max_j q_j \leq \min_\ell m_\ell$. As explained in [Bougeret et al., 2010a], the previous remark to get a $2+2\epsilon$ ratio can be extended (using a $PTAS$ for $Q||C_{max}$) for MCSP with clusters of different sizes but identical speeds *only* under this hypothesis $\max_j q_j \leq \min_\ell m_\ell$. For the general problem where a job may not fit into a cluster, we would have to use a $PTAS$ for the problem of scheduling jobs with *inclusive processing set restrictions* where machines have different speeds. However there is up to now (see the survey [Leung and Li, 2008]) only $PTAS$ for scheduling nested jobs when the machines have the same speed [Li and Wang, 2010] (or $FPTAS$ for the $Rm||C_{max}$ problem [Horowitz and Sahni, 1976]). Thus, there is up to now no polynomial algorithm with ratio better than 3 for the MCSP problem where clusters have the same speed.

To the best of our knowledge, there are no specific results for MCSP with clusters of same sizes and different speeds. Notice however that, again, the remark of [Ye et al., 2009] applies for this problem, using also a $PTAS$ for $Q||C_{max}$.

Our results

We present in Section 11.2 a pure "combinatorial" (without linear programming) algorithm of ratio $5/2$ that applies for MCSP when all the processors have the same speed, but the m_ℓ may differ. This improves the previous 3-approximation algorithms cited before (which moreover only applies for non-contiguous scheduling). That algorithm can be adapted to MCSP with clusters of the same size but with different speed values (see the appendix). The algorithm needs an oracle guess which will require (when enumerating all the possible answers of the oracle) to enumerate $O(n^N)$ possibilities in the worst case. From the methodological point of view, such an combinatorial algorithm with oracle may emphasize what is critical in the problem and provide insight for the considered problem. From the point of view of practical applications, even if the previous complexity is only polynomial for fixed N, this algorithm is faster than using approximation schemes for $Rm||C_{max}$ with $\epsilon = \frac{1}{4}$. Moreover, this running time can be improved (see Section 11.2.4) using classical rounding techniques. Since we assign each jobs to processors of consecutive addresses, all these results also apply for MSP (*i.e* for contiguous version).

11.2 A 5/2-Approximation for MCSP where clusters have the same speed

11.2.1 Main Ideas and Algorithm

In this section we study the problem of scheduling rigid jobs on clusters that have different numbers of processors, supposing that all clusters run at the same speed. We provide a 5/2-approximation that both applies for job scheduling and multiple strip packing.

The main idea of the algorithm is to schedule a set π_ℓ in each cluster Cl_ℓ, starting from Cl_1, such that $\sum_{\ell=1}^{\ell_0} A(\pi_\ell) \geq \sum_{\ell=1}^{\ell_0} A(\pi_\ell^*)$ for any ℓ_0, where π_ℓ^* is the set scheduled in Cl_ℓ in a fixed optimal solution. As the optimal value is not known, we use the classical dual approximation technique [Hochbaum and Shmoys, 1988] and denote by $T \in [p_{\max}, np_{\max}]$ the current value of the guess (of the non-contiguous optimal). Since the clusters may have different numbers of processors we define $Fit^\ell := \{J_j | q_j \leq m_\ell\}$, the set of jobs that fit in Cl_ℓ. Moreover we define $Lg := \{J_j | p_j \geq \frac{T}{2}\}$ the set of long jobs and $Wd^\ell := \{J_j | m_\ell \geq q_j \geq \frac{m_\ell}{2}\}$ the set jobs that are wide in Cl_ℓ. A way to guarantee the area domination is to select for each ℓ a set of jobs X such that $A(X) \geq m_\ell T$, and to schedule it below $\frac{5T}{2}$.
If $m_\ell T \leq A(X) \leq \frac{5}{4} m_\ell T$, we already know according to Steinberg's theorem [Steinberg, 1997] that X can be scheduled below $\frac{5T}{2}$ in polynomial time. For the cases where $A(X) > \frac{5}{4} m_\ell T$, we have to proceed differently. By cleverly choosing the set X as a union of a subset $wide \subset Wd^\ell$ of the wide rectangles and a subset $select \subset (Fit^\ell \setminus Wd^\ell)$ we make sure that we have only a very small number of critical jobs to handle in this case,and that X can be scheduled in $\frac{5T}{2}$. For example, with $X = \{J_1, J_2, J_3, J_4\}$, with $q_1 = q_2 = q_3 = \frac{m_\ell}{2} + \epsilon$, $p_1 = \frac{T}{2} + \epsilon$, $p_2 = p_3 = \frac{T}{2}$, $p_4 = T$ and $q_4 = \frac{m_\ell}{2}$, we have $A(X') < T$ for any $X' \subsetneq X$ and X cannot be scheduled in $\frac{5T}{2}$.

It appears that the only restriction we need for defining X is to have $P(X \cap Wd^\ell \leq \frac{3T}{2})$. Thus, it is possible that for a given ℓ_0 we have $A(X) < m_{\ell_0} T$ with $Fit^{\ell_0} \neq \emptyset, Fit^{\ell_0} \subset Wd^{\ell_0}$. In this case, to ensure the area domination we also need an area domination for the wide jobs, that is $\sum_{\ell=1}^{\ell_0} A(\pi_\ell \cap Wd^{\ell_0}) \geq \sum_{\ell=1}^{\ell_0} A(\pi_\ell^* \cap Wd^{\ell_0})$.

This area domination for the wide jobs could be guaranteed by scheduling for each cluster the widest possible job, until reaching T. However, by using only a widest first policy we could overlap $\frac{3T}{2}$ because of "big" jobs of $Wd^{\ell_0} \cap Lg$. Thus, by guessing for each cluster the (potential) unique big job scheduled in this cluster in the optimal, we can use the widest first policy with jobs of $Wd^{\ell_0} \setminus Lg$ and avoid the previous problem.

So briefly described our algorithm works as follows. We first enumerate the unique big job for each cluster. Then, for each cluster (starting with Cl_1), we select during phase 1 some wide jobs with widest first policy from $(Wd^\ell \setminus Lg)$ and add them to $wide$ until we have a total of length $P(wide)$ at least T and at most $\frac{3T}{2}$. The only way to schedule the wide jobs is one after another. So we sort the jobs in non-increasing order of their widths and schedule them bottom-left justified starting with the widest.
In phase 2 we add jobs with largest area from $(Fit^\ell \setminus Wd^\ell)$ to $select$ as long as $A(wide \cup select) < Tm_\ell$. As mentioned before in phase 3 we reschedule $wide \cup select$ with Steinberg if possible or use the fact that we have selected only few critical jobs.

Algorithm 6

guess $J_{j_\ell^*} \in Lg \cap Wd^\ell \cap \pi_l^*$ for all $\ell \in \{1, \ldots, N\}$ and remove them from the initial set of jobs
for $\ell = 1$ to N **do**
——————————- phase 1 ——————————-
$wide \leftarrow \emptyset$
add $J_{j_\ell^*}$ to $wide$
while $((P(wide) < T)$ and $(Wd^\ell \setminus Lg \neq \emptyset))$ **do**
$J_{j_0} \leftarrow$ widest job of $Wd^\ell \setminus Lg$
add J_{j_0} to $wide$
end while
Reschedule jobs of $wide$ sequentially in non-increasing order of their width starting with the widest bottom-left justified (see Figure 11.2).
——————————- phase 2 ——————————-
$select \leftarrow \emptyset$
while $((A(wide) + A(select) < m_\ell T)$ and $(Fit^\ell \setminus Wd^\ell \neq \emptyset))$ **do**
$J_{j_0} \leftarrow$ job of $Fit^\ell \setminus Wd^\ell$ with largest area
add J_{j_0} to $select$
end while
——————————- phase 3 ——————————-
if $A(wide) + A(select) \leq \frac{5}{4} m_\ell T$ **then**
reschedule $wide \cup select$ using Steinberg [Steinberg, 1997] algorithm
else
schedule $select$ using lemma 11.1
end if
end for
if there is an unscheduled job **then**
reject T
end if

11.2.2 Analysis

Given that we use the dual approximation technique, we have to prove that either Algorithm 6 produces a schedule of makespan lower than $\frac{5T}{2}$, or that $T < Opt$ (in this case we say that T is rejected), where Opt denotes the non-contiguous optimal value . For the sake of simplicity, we do not mention everywhere the "reject" instruction in the algorithm. Thus we assume throughout the section that $T \geq Opt$, and it is implicit that if during execution one of the claimed properties is wrong then T should be rejected.

We start by proving that the set of selected jobs assigned by our algorithm to a cluster Cl_ℓ can always be scheduled in $\frac{5T}{2}$.

Lemma 11.1. *Let $\ell \in \{1, \ldots, N\}$, $p \in \mathbb{N}$ and let wide and select $= \{J_1, \ldots, J_p\}$ be the set of jobs selected for Cl_ℓ in phase 1 and 2. There exists a feasible schedule of $wide \cup select$ into Cl_ℓ with a makespan lower than $\frac{5T}{2}$.*

Proof Since it is always possible to schedule $wide \cup select$ with the algorithm of Steinberg [Steinberg, 1997] into the designated space if $A(wide) \cup A(select) \leq \frac{5T}{4}$, we

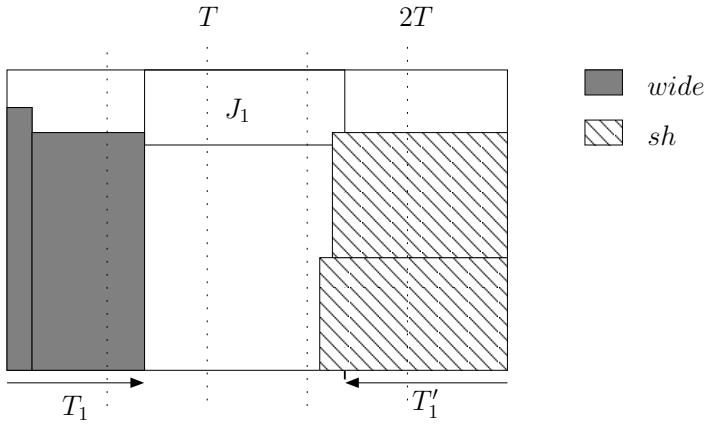

FIG. 11.2: Example of schedule built in Lemma 11.1

only consider cases with $A(wide) \cup A(select) > \frac{5T}{4}$ in phase 3. If $A(wide) \geq m_\ell T$, the algorithm skips phase 2 and consequently $A(select) = \emptyset$ and since $P(wide) \leq {}^{3T}/_{2}$ by construction, all jobs are scheduled below $\frac{5T}{2}$.
Let us assume $A(wide) < m_\ell T$. Since $A(wide) + A(select) > \frac{5}{4} m_\ell T$ the last job J_p added to *select* has total area strictly larger than $\frac{m_\ell T}{4}$ (otherwise the algorithm would have stopped before). This implies $A(J_j) > \frac{m_\ell T}{4}$ for all $j \in \{1, \ldots, p\}$ and thus $p \leq 4$. Since $J_j \notin Wd^\ell$ we furthermore conclude $q_j > \frac{m_\ell}{4}$ and $p_j > \frac{T}{2}$. We add now the jobs in *select* in the following way (see Figure 11.2) :

- Sort the jobs in *select* by decreasing width.
- Starting at time $\frac{5T}{2}$ schedule as many jobs of *select* as possible in the reverse direction using widest first policy bottom-right justified. Let *sh* denote this set of jobs, and let α denote the number of jobs in *sh*.
- If $\alpha < p$, schedule J_j $(\alpha < j \leq p)$ top justified into Cl_ℓ as soon as possible (i.e. at time $t_j := \min\{t | q_j$ consecutive processors are idle in $Cl_\ell\}$).

Since $Wd^\ell \cap select = \emptyset$, we have $\alpha \geq 2$. Since $P(wide) \leq \frac{3T}{2}$ the schedule is feasible for $p \leq 2$. Consequently we only study two other cases, namely $p = 3$ or $p = 4$.
Let $p = 3$ and let $J_1 \in select \backslash sh$. Assume that the previous algorithm fails when scheduling J_1, implying that J_1 scheduled at time t_1 intersects *sh*. With $t'_1 := \frac{5T}{2} - t_1 - p_1$ we conclude

$$\begin{aligned} A(wide \cup sh) &> t_1(m_\ell - q_1) + t'_1(m_\ell - q_1) + (Q(sh) - (m_\ell - q_1))\frac{T}{2} \\ &\overset{Q(sh) \geq 2q_1}{>} t_1(m_\ell - q_1) + (\frac{3T}{2} - t_1)(m_\ell - q_1) + (3q_1 - m_\ell)\frac{T}{2} \geq m_\ell T, \end{aligned}$$

which is a contradiction, since we have $\forall X \subset select : A(wide) + A(select \backslash X) < m_\ell T$.
Now let $p = 4$. Without loss of generality we assume that there are jobs $J_1, J_2 \in select \backslash sh$ with $p_1 \geq p_2$. Notice that since the algorithm selected 4 jobs of area strictly larger than $\frac{m_\ell T}{4}$ in phase 2 we have $A(wide) < \frac{m_\ell T}{4}$ and thus $P(wide) \leq \frac{T}{2}$. Thus we have empty space of widths one between level $\frac{T}{2}$ and $\frac{3T}{2}$ where we can directly schedule J_1 and J_2 at time $\frac{T}{2}$. □

Now we prove that the area of "wide" jobs we scheduled by the algorithm is larger

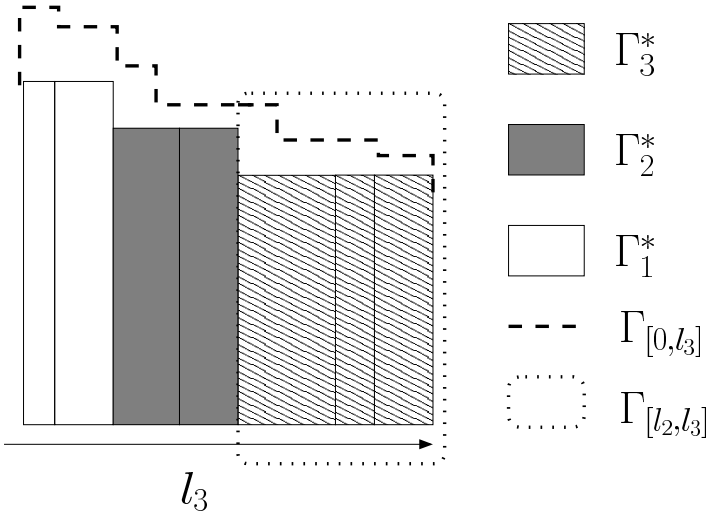

FIG. 11.4: Example of stacks used in Lemma 11.3

than the one in the optimal. Recall that for all ℓ we denote by π_ℓ the set of jobs scheduled in Cl_ℓ by the algorithm, and π_ℓ^* the set of jobs scheduled in Cl_ℓ in a fixed optimal solution.

Lemma 11.3. *For $\ell \in \{1, \ldots, N\}$ let $\Pi = \bigcup_{t=1}^{\ell} \pi_t$ be the set of jobs scheduled by the algorithm after finishing the ℓth iteration and $\Pi^* = \bigcup_{t=1}^{\ell} \pi_t^*$ the corresponding optimal set of jobs. Then $A(\Pi \cap Wd^\ell) \geq A(\Pi^* \cap Wd^\ell)$.*

Proof Roughly speaking, this area domination for wide jobs is true since for each cluster Cl_ℓ, we schedule (without counting the guessed jobs that are common to our schedule and the optimal) a length of at least $T - p_{j_\ell^*}$ of the widest possible jobs, and the available length for schedule wide jobs in the optimal is at most $T - p_{j_\ell^*}$. We now start the formal proof.

Assume that $Wd_\ell \neq \emptyset$ after iteration ℓ, otherwise the claim follows directly.
We first consider jobs of $B^\ell = Wd^\ell \cap Lg$. We have $\Pi \cap B^\ell = \Pi^* \cap B^\ell$. Indeed, the only jobs of B^ℓ that we scheduled are the guessed one, as no job of B^ℓ can be scheduled in phase one, two or three in clusters $Cl_1 \ldots Cl_\ell$. In addition, the only jobs of B^ℓ scheduled in $\pi_1^* \ldots \pi_\ell^*$ are also the guessed one, as it is not possible to schedule more than one job of B^ℓ in any cluster $Cl_1 \ldots Cl_\ell$. Thus we only consider now jobs of $Wd^\ell \setminus Lg$. Let $\Gamma = (\Pi \cap Wd^\ell) \setminus Lg$ and $\Gamma^* = (\Pi^* \cap Wd^\ell) \setminus Lg$. W.l.o.g., we assume that there are $a \leq |\Gamma^*|$ different widths $q_1 \geq \ldots \geq q_a$ in Γ^*. Let $\Gamma_j^* \subset \Gamma^*$ be the subset of jobs of width q_j and $n_j := |\Gamma_j^*|$. Let $l_j := \sum_{x=1}^{j} P(\Gamma_x^*)$. We consider the shapes of the rows built by stacking the jobs in Γ and Γ^*, respectively, next to each other sorted by non-increasing width (see Figure 11.4). We introduce a partial order over stacks of rectangles denoted with "$\leq$". Given two stacks Γ_1 and Γ_2, we say $\Gamma_1 \leq \Gamma_2$ if the shape representing Γ_1 is contained in the one representing Γ_2. For levels l, l' let $\Gamma_{[l,l']}$ denote the row of Γ between l and l'. Remark that $\Gamma^*_{[l_{j-1},l_j]}$ corresponds exactly to Γ_j^*. We show by induction over the number of different widths in Γ^* that $\Gamma^* \leq \Gamma$. Suppose that $\Gamma^*_{[0,l_{j-1}]} \leq \Gamma_{[0,l_{j-1}]}$ and let us prove that $\Gamma^*_{[0,l_j]} \leq \Gamma_{[0,l_j]}$. If all the jobs of $\Gamma^*_{[l_{j-1},l_j]}$ are scheduled by the algorithm we get the desired result. Indeed, no job of Γ_j^*

is contained in $\Gamma_{[0,l_{j-1}]}$ otherwise there would be a job J_x and a level $0 \leq l' \leq l_{j-1}$ with $\Gamma^*_{[l',l'+p_x]} > \Gamma_{[0,l'+p_x]}$, as all jobs of $\Gamma^*_{[0,l_{j-1}]}$ are strictly wider than q_j. Thus, $\Gamma^*_{[l_{j-1},l_j]}$ is included in $\Gamma_{[l_{j-1},l_j]}$ and we conclude using the induction hypothesis.
Assume now that there is a job $J_{x_0} \in \Gamma^*_j \backslash \Gamma$. The total processing time of jobs that are wider than J_{x_0} scheduled in the optimal (into clusters $\{Cl_1, \ldots, Cl_\ell\}$) is l_j. Let l' denote the total processing time of jobs wider than J_{x_0} that the algorithm packed. We prove that $l' \geq l_j$.
Let N' be the number of clusters where J_{x_0} fits (J_{x_0} fits in clusters $Cl_{\ell-N'+1} \ldots, Cl_\ell$). If J_{x_0} is not scheduled in any of these N' clusters, it means that the algorithm scheduled other jobs, that are wider than J_{x_0}. Thus, for any $t \in \{\ell - N' + 1, \ldots, \ell\}$, the total processing time of jobs wider than J_{x_0} scheduled by the algorithm on Cl_t is larger than $T - p_{j^*_t}$, which is also an upper bound for the total processing time of schedulable jobs of width larger than q_{x_0} in the optimum. Consequently, we get $l' \geq \Sigma^{\ell}_{t=i-N'+1}(T - p_{j^*_t}) \geq l_j$, which implies $\Gamma[l_{j-1}, l_j] \geq \Gamma^*_{[l_{j-1},l_j]}$. □

We can now prove that all the jobs are scheduled in the end.

Lemma 11.5. *All the jobs are scheduled when the algorithm stops.*

Proof We prove by induction on ℓ that for all $i \in \{1, \ldots, N\}$ we have $A(\bigcup_{t=1}^{\ell} \pi_t) \geq A(\bigcup_{t=1}^{\ell} \pi^*_t)$ after finishing scheduling Cl_ℓ. Let $\Pi = \bigcup_{t=1}^{\ell} \pi_t$ and $\Pi^* = \bigcup_{t=1}^{\ell} \pi^*_t$. Two cases are possible according to what happens in phase 3. If $A(wide) + A(select) \geq m_\ell T$, then we conclude directly. Let us assume that $A(wide) + A(select) < m_\ell T$. This implies that $Fit^\ell \setminus Wd^\ell = \emptyset$ when scheduling Cl_ℓ. Thus, if we write $A(\Pi) = A(\Pi \bigcap Wd^\ell) + A(\Pi \bigcap (I \setminus Wd^\ell))$ we get the desired result as $A(\Pi \bigcap Wd^\ell) \geq A(\Pi^* \bigcap Wd^\ell)$ (according to lemma 11.3) and $\Pi^* \bigcap (I \setminus Wd^\ell) \subset Fit^\ell \setminus Wd^\ell = \Pi \bigcap (I \setminus Wd^\ell)$. □

11.2.3 Complexity

Algorithm 6 needs an oracle that provides for every cluster the index of the big (meaning wide and long) job scheduled on this cluster (if such a job is scheduled in the optimum). Thus, the cost of the enumeration is in $O(\prod_{\ell=1}^{N} x_\ell)$, where $x_\ell = |Wd^\ell \cap Lg| + 1$ (we need to add one to encode the possibility where no job of $Wd^\ell \cap Lg$ is scheduled on Cl_ℓ). The problem is that the rough upper bound on this cost (n^N) is almost tight for instances where there are n jobs of width and processing time 1, $N-1$ clusters of size $2-\epsilon$ and 1 very large cluster (let us say of size n). In this case there are indeed n possible big jobs for the first $N-1$ clusters. The overall complexity for the algorithm is in $O(N \frac{n \log^2 n}{\log(\log(n))} \log(np_{max}) n^N)$ (the $\frac{n \log^2 n}{\log(\log(n))}$ factor is the complexity of Steinberg's algorithm, and the $\log(np_{max})$ factor is the running-time of the binary search).

We propose in the appendix an improvement of Algorithm 6 using a classical input rounding to replace the n^N factor by c^N, with c constant.

11.2.4 Improvement using rounding

The idea is that we could still guarantuee the "area domination for wide jobs" (see Lemma 11.3) by only guessing the processing time of of the (potential) big job scheduled on each cluster, and schedule the widest job that has this processing time. Thus, this new guess could become smaller if the number of different processing times of long jobs is small. We will prove the following theorem.

Theroem 11.6. *There is a $\frac{5}{2}(1+\epsilon)$-approximation for MCSP where clusters have the same speed that runs in $O(N\frac{n\log^2 n}{\log(\log(n))}\log(np_{max})(\frac{1}{2\epsilon}+1)^N)$.*

Let us first define the rounding.

Lemma 11.7. *Let I be the original instance, T a guess of $Opt(I)$ and $\epsilon > 0$. We can construct I'_T such that*

- *there are at most $\frac{1}{2\epsilon}+1$ different processing times for all the jobs of Lg' (where $Lg' = \{J_j | p_j > T/2\} \cap I'_T$)*
- *if $T \geq OPT(I)$ then $Opt(I'_T) \leq T(1+\epsilon)$*

Proof We generate I'_T by rounding up the processing time p_j of every long job $J_j \in Lg$ to a value $p'_j := \frac{T}{2} + (a_j+1)\epsilon T$ with $\frac{T}{2} + a_j\epsilon T \leq p_j \leq p'_j$. Of course there are at most $\frac{1}{2\epsilon}+1$ different processing times in Lg'. Since the jobs in Lg are executed in parallel in the optimal solution (since $\frac{T}{2} \geq \frac{OPT(I)}{2}$), replacing those jobs by the ones in Lg' increases the makespan by at most ϵT. Thus $Opt(I'_T) \leq T(1+\epsilon)$. □

Let $T' = T(1+\epsilon)$. Let us now describe the Algorithm 7, that given an instance I'_T (as defined in Lemma 11.7) either schedules all the jobs with a makespan lower than $\frac{5}{2}T'$, or rejects T' implying that $T' < OPT(I'_T)$ (and thus $T < OPT(I)$). We consider of course that $Wd^\ell = \{J_j \in I'_T | q_j > \frac{m_\ell}{2}\}$.

Algorithm 7

for all $\ell \in \{1,\ldots,N\}$, guess $p_{j\ell^*}$ the processing time of the (potential) job of $Lg' \cap Wd^\ell \cap \pi^*_l$
for $\ell = 1$ to N **do**
  $J_{x_\ell} \leftarrow$ widest (that have the biggest q_j) job of of processing time $p_{j^*_\ell}$
  if $q_{x_\ell} > \frac{m_\ell}{2}$ **then**
    schedule J_{x_ℓ} on Cl_ℓ //otherwise we say that J_{x_ℓ} is **discarded** by Cl_ℓ
  end if
end for
run Algorithm 6 replacing T by T' (and replacing of course $J_{j^*_l}$ by J_{x_ℓ})

We only have to prove the equivalent of Lemma 11.3.

Lemma 11.8. *Let $\ell \in \{1,\ldots,N\}$, let $\Pi = \bigcup_{t=1}^{\ell}\pi_t$ after finishing scheduling Cl_ℓ, and let $\Pi^* = \bigcup_{t=1}^{\ell}\pi^*_t$. Then we have $A(\Pi \cap Wd^\ell) \geq A(\Pi^* \cap Wd^\ell)$.*

Proof Let $\ell \in \{1, \ldots, N\}$ and $Wd^\ell \neq \emptyset$ after iteration ℓ of the algorithm. Otherwise the claim follows directly.

We first consider jobs of $B^\ell = Wd^\ell \cap Lg'$ and $B^\ell_x = B^\ell \cap \{J_j | p_j = \frac{T}{2} + x\epsilon T\}$. Let $\Gamma_x = \Pi \cap B^\ell_x$ and $\Gamma^*_x = \Pi^* \cap B^\ell_x$. We will prove that $A(\Pi \cap B^\ell) \geq A(\Pi^* \cap B^\ell)$ by proving that for every x, $A(\Gamma_x) \geq A(\Gamma^*_x)$. Let x be fixed. We proceed as in Lemma 11.3 by stacking the jobs of Γ^*_x and Γ_x. Let us assume that there are $a \leq |\Gamma^*_x|$ different widths $q_1 \geq \ldots \geq q_a$ in Γ^*_x. Let $\Gamma^*_{x,j} \subset \Gamma^*_x$ be the subset of jobs of width q_j and $n_j := |\Gamma^*_{x,j}|$. Let $l_j := \sum_{t=1}^{j} P(\Gamma^*_{x,t})$. For levels l, l' let $\Gamma_{[l,l']}$ denote the jobs scheduled in the stack of Γ between l and l'. We show by induction over the number of different widths in Γ^*_x that $\Gamma^*_x \leq \Gamma_x$, where $\leq$ denotes the same partial order as in Section 11.3.

Suppose that $\Gamma^*_{x\ [0,l_{j-1}]} \leq \Gamma_{x\ [0,l_{j-1}]}$ and let us prove that $\Gamma^*_{x\ [0,l_j]} \leq \Gamma_{x[0,l_j]}$. If all the jobs of $\Gamma^*_{x,j}$ are scheduled by the algorithm we get the desired result.

Assume now that there is a job $J_{x_0} \in \Gamma^*_{x,j} \backslash \Gamma$ (we have $q_{x_0} = q_j$). Let $X_{x_0} = \{J_s | q_s \geq q_{x_0}\}$. Due to the induction hypothesis, we only need to prove that $\Gamma_{x\ [l_{j-1},l_j]} \geq \Gamma^*_{x\ [l_{j-1},l_j]}$, and thus we will only prove that $|\Gamma_x \cap X_{x_0}| \geq |\Gamma^*_x \cap X_{x_0}|$.

We have $|\Gamma^*_x \cap X_{x_0}| = \sum_{t=1}^{j} n_t$, implying that there are at least $\sum_{t=1}^{j} n_t$ clusters where J_{x_0} fits. Moreover, $q_{x_0} > \frac{m_\ell}{2}$ implies that $q_{x_0} > \frac{m_{\ell'}}{2}$ for all $\ell' < \ell$. Thus, J_{x_0} has not been discarded by any cluster between 1 and ℓ. Thus, J_{x_0} has not been scheduled in any of the $\sum_{t=1}^{j} n_t$ clusters where it fits because the algorithm scheduled wider job instead, which proves that $|\Gamma_x \cap X_{x_0}| \geq \sum_{t=1}^{j} n_t = |\Gamma^*_x \cap X_{x_0}|$.

The proof of $A(\Pi \cap Wd^\ell \setminus Lg') \geq A(\Pi^* \cap Wd^\ell \setminus Lg')$ is exactly the same as in Lemma 11.3 as for any ℓ', the processing time of the jobs $J_{x_{\ell'}}$ is either the same as the potential big job scheduled by the optimal in $Cl_{i'}$, or zero if $J_{x_{\ell'}}$ is discarded. □

Thus, Algorithm 7 is a $\frac{5}{2}(1 + \epsilon)$-approximation, and runs in $O(N \frac{n \log^2 n}{\log(\log(n))} \log(np_{max})(\frac{1}{2\epsilon} + 1)^N)$.

11.2.5 A 5/2-approximation for clusters of same size but different speed

It is possible to adapt Algorithm 6 to get a 5/2-approximation for the case where each cluster Cl_ℓ has m identical processors of speed s_ℓ. Again this result also applies for the contiguous case.

We also proceed by dual approximation, and denote by T the current guess. Moreover, let $Fit^\ell = \{J_j | \frac{p_j}{s_\ell} \leq T\}$ be the set of jobs that fit in Cl_ℓ, $Lg^\ell = \{J_j | T \geq \frac{p_j}{s_\ell} > \frac{T}{2}\}$ and $Wd = \{J_j | q_j > \frac{m}{2}\}$.

The idea is to use exactly Algorithm 6 replacing of course Lg by Lg^ℓ and Wd^ℓ by Wd, and scheduling the clusters from the slowest (Cl_1) to the fastest one (Cl_N). Consequently the guess for each cluster ℓ is now the potential job in $Lg^\ell \cap Wd \cap \pi^*_l$ where π^*_l is the set of jobs scheduled on Cl_ℓ in the optimal solution.

The proof of the feasibility of phase 3 is exactly the same as in Lemma 11.3. The only adaptation needed is to prove the following lemma.

Lemma 11.9. *For any $x \in \{1, \ldots, N\}$, let $\Pi_x = \bigcup_{t=1}^{x} \pi_t$ be the set of jobs scheduled*

by the algorithm after finishing the xth iteration and $\Pi_x^* = \cup_{t=1}^{x} \pi_t^*$ *the corresponding optimal set of jobs.*

Then we have, for any $\ell \in \{1, \ldots, N\}$, $(\Pi_\ell \cap Wd) \geq (\Pi_\ell^* \cap Wd)$, *where* $\geq$ *denotes the same partial order for rows of jobs as in Section 11.3 for stacks of rectangles.*

Proof Let us prove the desired result by induction on ℓ. Let us suppose that $(\Pi_{\ell-1} \cap Wd) \geq (\Pi_{\ell-1}^* \cap Wd)$ (the proof for $\ell = 1$ can be done using the same ideas).

Let $\Gamma = \Pi_l \cap Wd$ and $\Gamma^* = \Pi_l^* \cap Wd$. As in Lemma 6 we prove that $\Gamma^* \leq \Gamma$ by induction on the different numbers of widths of jobs in Γ^*. We use the same notations as in Lemma 11.3. We suppose that $\Gamma^*_{[0,l_{j-1}]} \leq \Gamma_{[0,l_{j-1}]}$ and we prove that $\Gamma^*_{[0,l_j]} \leq \Gamma_{[0,l_j]}$ by showing that $\Gamma^*_{[l_{j-1},l_j]} \leq \Gamma_{[l_{j-1},l_j]}$. Let us only consider the case where there is J_{x_0} in $\Gamma_j^* \backslash \Gamma$. This implies $J_{x_0} \notin Lg^\ell$, otherwise J_{x_0} would belong to Lg^t for $1 \leq t \leq \ell$, and thus would be a guessed job as the optimal scheduled it in one of the first ℓ clusters. Let $X_\alpha = \{J_s \in J | q_s \geq \alpha\}$ be the set of jobs wider than α. As in Lemma 11.3 we prove that $\Gamma^*_{[l_{j-1},l_j]} \leq \Gamma_{[l_{j-1},l_j]}$ by showing that $l' \geq l_j$, where $l' = P(X_{q_{x_0}} \cap \Gamma)$ and l_j defined as in Lemma 11.3 (recall that the definition of l_j implies that $l_j = P(X_{q_{x_0}} \cap \Gamma^*)$).

The hypothesis $(\Pi_{\ell-1} \cap Wd) \geq (\Pi_{\ell-1}^* \cap Wd)$ implies that for any α, $P(X_\alpha \cap \Gamma \cap \Pi_{l-1}) \geq P(X_\alpha \cap \Gamma^* \cap \Pi_{l-1}^*)$, thus we use it with $\alpha = q_{x_0}$. Moreover, as J_{x_0} is not scheduled by the algorithm on Cl_ℓ whereas $J_{x_0} \notin Lg^\ell$, it implies that we scheduled wider jobs than J_{x_0} on Cl_ℓ. Thus, we get also $P(X_{q_{x_0}} \cap \Gamma \cap \pi_l) \geq P(X_{q_{x_0}} \cap \Gamma^* \cap \pi_l^*)$, leading to $l' \geq l_j$ and to $\Gamma^*_{[l_{j-1},l_j]} \leq \Gamma_{[l_{j-1},l_j]}$. □

Bibliographie

[Bougeret et al., 2010a] Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., and Trystram, D. (2010a). A fast 5/2-approximation for hierarchical scheduling. In *Proceedings of the 16th International European Conference on Parallel and Distributed Computing (EUROPAR)*.

[Bougeret et al., 2010b] Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., and Trystram, D. (2010b). Approximating the non-contiguous multiple organization packing problem. In *Proceedings of the 6th IFIP International Conference on Theoretical Computer Science (TCS)*.

[Coffman Jr et al., 1980] Coffman Jr, E., Garey, M., Johnson, D., and Tarjan, R. (1980). Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9 :808.

[Harren et al., 2010] Harren, R., Jansen, K., Prädel, L., and Van Stee, R. (2010). A $5/3 + \epsilon$ approximation for strip packing. submitted.

[Hochbaum and Shmoys, 1988] Hochbaum, D. and Shmoys, D. (1988). A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM Journal on Computing*, 17(3) :539–551.

[Horowitz and Sahni, 1976] Horowitz, E. and Sahni, S. (1976). Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM (JACM)*, 23(2) :317–327.

[Jansen and Solis-Oba, 2007] Jansen, K. and Solis-Oba, R. (2007). New approximability results for 2-dimensional packing problems. *Lecture Notes in Computer Science*, 4708 :103.

[Jansen and Thöle, 2008] Jansen, K. and Thöle, R. (2008). Approximation algorithms for scheduling parallel jobs : Breaking the approximation ratio of 2. In *International Colloquium on Automata, Languages and Programming*, pages 234–245.

[Kenyon and Rémila, 2000] Kenyon, C. and Rémila, E. (2000). A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, pages 645–656.

[Leung and Li, 2008] Leung, J. and Li, C. (2008). Scheduling with processing set restrictions : A survey. *International Journal of Production Economics*, 116(2) :251–262.

[Li and Wang, 2010] Li, C. and Wang, X. (2010). Scheduling parallel machines with inclusive processing set restrictions and job release times. *European Journal of Operational Research (EJOR)*, 200(3) :702–710.

[Schiermeyer, 1994] Schiermeyer, I. (1994). Reverse-fit : A 2-optimal algorithm for packing rectangles. *Lecture Notes in Computer Science*, pages 290–290.

[Schwiegelshohn et al., 2008] Schwiegelshohn, U., Tchernykh, A., and Yahyapour, R. (2008). Online scheduling in grids. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10.

[Steinberg, 1997] Steinberg, A. (1997). A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26 :401.

[Ye et al., 2009] Ye, D., Han, X., and Zhang, G. (2009). On-Line Multiple-Strip Packing. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications (COCOA)*, page 165. Springer.

[Zhuk, 2006] Zhuk, S. (2006). Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1) :73–85.

Chapitre 12

Approximating the Discrete Resource Sharing Scheduling Problem [Bougeret et al., 2009a]

Marin bougeret[*‡], Pierre-François Dutot[*], Alfredo Goldman[†], Yanik Ngoko[*§] and Denis Trystram[*]

[*]LIG, Grenoble University, France
bougeret,dutot,trystram@imag.fr

[‡]This work is supported by DGA-CNRS

[†]Alfredo Goldman has been partially supported by the MINALOGIC project SCEPTRE funded by Region Rhone-Alpes in France

[§]Yanik Ngoko was sponsored by the international exchange program EGIDE

Abstract

The goal of this work is to study the portfolio problem which consists in finding a good combination of multiple heuristics given a set of a problem instances to solve. We are interested in a parallel context where the resources are assumed to be discrete and homogeneous, and where it is not possible to allocate a given resource (processor) to more than one heuristic. The objective is to minimize the average completion time over the whole set of instances. We extend in this paper some existing analysis on the problem. More precisely, we provide a new complexity result for the restricted version of the problem, then, we generalize previous approximation schemes. In particular, they are improved using a guess approximation technique. Experimental results are also provided using a benchmark of instances on SAT solvers.

12.1 Introduction

12.1.1 Description of the Portfolio problem

We are interested in this work in solving hard computational problems like the satisfiability problem SAT [12]. It is well-established that a single algorithm cannot solve efficiently all the instances of such problems. In most cases, the algorithms are characterized by the great variability of their execution time depending on the considered instances. Thus, a good effective solution is to consider several heuristics and combine them in such a way to improve the mean execution time when solving a large set of instances. In this paper, we are interested in designing adequate combination schemes.

The suggested solution is based on the portfolio problem, introduced in the field of finance many years ago [15]. This problem can be informally recalled as follows : given a set of opportunities, an amount of possible investments on the set of opportunities and the payoff obtained when investing an amount on each opportunity, what is the best amount of investment to make on each opportunity in order to maximize the sum of the payoffs ? Using the vocabulary of Computer Science, we assume that there exists a benchmark composed of a finite set of instances and some heuristics which solve these instances. The expected execution times of heuristics on all the instances is known. The objective is to determine the best possible resource allocation for the heuristics in order to minimize the mean execution time of the set of instances. The execution time of an instance given a resource allocation is taken here as the shortest execution time of a heuristic when executing simultaneously all the heuristics on this instance.

This formulation of the problem as a portfolio is motivated by the fact that we may not know which is the best suited heuristic to solve an instance before actually solving it. The interest of this sharing model is that in practice if the benchmark of instances is representative over the set to be solved, we can expect a better mean execution time than using only one heuristic.

12.1.2 Related works

There exist many studies focusing on the construction of automated heuristic selection process. For a given problem, the approaches usually proceed first by identifying the set of features which characterize its instances. A matching is then built between types of instances and heuristics in order to determine an efficient heuristic for any instance.

An et al. [2], for example, introduce a generic framework for heuristic selection in a parallel context and apply it to several classical problems including sorting and parallel reductions. Weerawarana et al. [22] suggest a model for the heuristics selection in the case of the resolution of partial differential equations. The SALSA project (Self Adapting Large-scale Solver Architecture) uses statistical techniques for solver selection [9]. Bhowmick et al. [3] study the construction of a selection process for solving linear systems. The construction of automated selection process requires the identification of a representative set of features. This can be very difficult depending on the targeted problems [9].

There exist other alternative works based on heuristic portfolio that can be used in these cases. A portfolio of heuristics is a collection of different algorithms (or algorithms

with different parameters) running in an interleaved scheme. In [13, 14], the authors have demonstrated the interest to use heuristic portfolio on randomized heuristics. The concurrent use of heuristics for solving an instance has also been suggested in [8, 21] with the concept of asynchronous team. Streeter et al. [20] studied how to interleave the execution of various heuristics in order to reduce the execution time of a set of instances.

Sayag et al. [16] have also studied a related problem, namely the time switching problem. This problem considers a finite set of instances and assumes a finite set of interruptible heuristics. To solve instances, the execution of the various heuristics are interleaved in a fixed pattern of time intervals. The execution ends as soon as one heuristic solves the current instance. As previously, the goal in the task switching problem is to find a schedule which minimizes the mean execution time on the set of instances. This approach is interesting in a single resource problem and has also been studied by Streeter et al. [20]. Sayag et al. [16] proved that to each resource sharing schedule corresponds a time switching with a lower execution time, which means that if there is no overhead on context switching, it is always better to use time slicing on the heuristics on the whole resources instead of applying resource sharing. Even if the time switching approach produces theoretically schedules with constant approximation ratio, it assumes that the heuristics can be interrupted at any moment. However, all the interrupted states have to be stored, leading to a prohibitive memory cost on multiple resources.

Let us notice also that in several cases, giving more resources to a heuristic does not have a positive impact on its execution. This is especially true when using heuristics that are hard to parallelize, like those involving a large number of irregular memory accesses and communications [23]. That is why we focus on the discrete version of the resource sharing scheduling problem (denoted by RSSP) instead of time switching as introduced in [16].

To the best of our knowledge, there are mainly two papers about the RSSP that are closely related to our work. Sayag et al.[16] address the continuous linear RSSP, where the output is a fraction of the available resources allocated to each heuristic, and the cost function is linear, meaning that the cost for solving an instance with only one resource is x times the cost with x resources. In our previous work [5] a discrete version of the linear restricted RSSP is studied, denoted by lr-dRSSP. The discrete setting means that the output is an integer representing the number of resources allocated to each heuristic, and the restriction assumption means that every solution must allocate at least one processor to every heuristic. The restricted version was proposed given that the l-dRSSP (without restriction) has no constant ratio polynomial approximation algorithm (unless $P = NP$) [5].

12.1.3 Contributions and organization of the paper

In this work, we study the r-dRSSP problem. More precisely, given any heuristic h_i, any instance I_j and any number of allocated resources in $\{1, \ldots, m\}$, we only assume that $C(h_i, I_j, s)$ (the cost for solving I_j with h_i deployed on s resources) verifies $1 \leq \frac{C(h_i,I_j,s)}{C(h_i,I_j,\lambda s)} \leq \lambda, \forall \lambda \geq 1$. This reasonable assumption means that we handle any instance where the speedup is non “super linear”.

A preliminary version for the lr-dRSSP problem was published in the APDCM workshop [5]. The r-dRSSP problem is defined in Section 12.2.1. We provide in Section 12.2.2 a proof for the complexity of lr-dRSSP (which was still open in [5]). In Section 12.2.3, a first greedy algorithm is provided to give an insight on the problem. Oracle-based algorithms are then introduced in Section 12.3.1. We extend in 12.3.2 and 12.3.3 the oracle based approximation schemes proposed in [5] for lr-dRSSP to r-dRSSP. We give in 12.3.4 a kind of tightness result by proving that the question asked to the oracle is almost the most "efficient" possible one. Then, we improve in Section 12.3.5 these approximation schemes by applying the guess approximation technique (introduced and applied to lr-dRSSP in [6]). Finally, in Section 12.4 we run experiments (using instances extracted from actual execution times of heuristics solving the SAT problem) to evaluate these algorithms.

12.2 Discrete Resource Sharing Scheduling Problem

12.2.1 Definition of the problem

restricted discrete Resource Sharing Scheduling Problem (r-dRSSP)

Instance : A finite set of instances $I = \{I_1, \ldots, I_n\}$, a finite set of heuristics $H = \{h_1, \ldots, h_k\}$, a set of m identical resources, a cost $C(h_i, I_j, s) \in R^+$ for each $I_j \in I$, $h_i \in H$ and $s \in \{1, \ldots, m\}$, such that $1 \leq \frac{C(h_i,I_j,s)}{C(h_i,I_j,\lambda s)} \leq \lambda, \forall \lambda \geq 1$

Output : An allocation of the m resources $S = (S_1, \ldots, S_k)$ such that $S_i \in \mathbb{N}$ and $\sum_{i=1}^{k} S_i \leq m$ that minimizes $\sum_{j=1}^{n} \min_{1 \leq i \leq k} \{C(h_i, I_j, S_i) | S_i > 0\}$

The idea in this formulation is to find an efficient partition of resources to deploy the set of heuristics on the homogeneous resources. The cost function ($\min_{1 \leq i \leq k} \{C(h_i, I_j, S_i)\}$) introduced by Sayag et al. [16] and used here, considers that for each instance, all the different heuristics are executed with the defined share and then stop their execution when at least one heuristic finds a solution.

12.2.2 Complexity

The NP completeness proof of the l-dRSSP is provided in [5], using a reduction from the vertex cover problem. However, this proof cannot be directly adapted to lr-dRSSP. Indeed, the gap between the optimal and non-optimal cases comes from the fact that it is impossible to give one resource to every "important" heuristic when there is no vertex cover, whereas the new constraint in lr-dRSSP forces any solution to allocate at least one resource to every heuristic. Thus, we add in the new reduction (still from the vertex cover) some artificial instances that amortize this new constraint.

Theroem 12.1. *lr-dRSSP is NP hard.*

Proof First, let remark that it is straightforward to verify that the problem is in NP. Let us denote by T the threshold value of the lr-dRSSP decision problem. Given any solution S and any subset $X \subset I$, we denote by $f(S)_{|X} = \sum_{I_j \in X} \min_{1 \leq i \leq k} \{C(h_i, I_j, S_i) | S_i > 0\}$ the restricted cost of S on X. Being given

a graph $G = (V, E)$, $V = \{v_1, \ldots, v_k\}, k = |V|$, $|E| = n$ in which we are looking for a vertex cover $V^c \subseteq V$ of size x, we construct an instance of the lr-dRSSP problem as follows. We define the set of heuristics as $H = \{h_1, \ldots, h_k\}$ (to each h_i corresponds the vertex $v_i \in V$). The number of resources is $m = k + x$. Given that any solution of lr-dRSSP must allocate at least one resource to each heuristic, the decision only concerns the x extra resources. We choose $I = I' \cup I''$, with $I' = \{I_1, \ldots, I_n\}$ (to each $I_j \in I'$ corresponds an edge $(v_{j_1}, v_{j_2}) \in E$), and $I'' = \{I_{n+1}, \ldots, I_{n+k}\}$ (I'' forces the solutions to balance the x "extra" resources, which means not giving more than 2 resources to anybody). According to the linear cost assumption, it is sufficient to define only the $C(h_i, I_j, m)$ values. The cost function is :

$$C(h_i, I_j, m) = \begin{cases} \alpha & \text{if } j \in \{1, \ldots, n\} \text{ and } v_i = v_{j_1} \text{ or } v_i = v_{j_2} \\ Z & \text{if } j \in \{n+1, \ldots, n+k\} \text{ and } n + i = j \\ \beta & \text{otherwise} \end{cases}$$

The constant $\alpha > 0$ is chosen arbitrarily, and $\beta = mT + 1$ so that if a heuristic computes an instance with this cost (even with all the resources), the corresponding solution cannot be under the threshold value T. The values of T and Z later will be fixed later. The basic idea of this reduction is the same as for the l-dRSSP problem : if there is a vertex cover of size x, then there is a good naive solution S^1 which gives one "extra" resource to each of the x corresponding heuristics. Otherwise, we know that any potentially good solution S must balance the x extra resources (I'' forces any solution to well-balance the x), and then it is easy to see that S^1 is better than S on I'.

More precisely, if there is a vertex cover $V^c = \{v_{i_1}, \ldots, v_{i_x}\}$ of size x, we define $S^1_i = 2$ if $v_i \in V^c$, and 1 otherwise. We have $\sum_{i=1}^{k} S^1_i = k + x$, and $Opt \leq f(S^1) = f(S^1)_{|I'} + f(S^1)_{|I''} = nm\frac{\alpha}{2} + Zm(k - x + \frac{x}{2})$. Thus, we define the threshold value $T = nm\frac{\alpha}{2} + Zm(k - x + \frac{x}{2})$. This value still depends on Z, which will be defined after.

Otherwise, let us consider a solution S, and let $a = card\{S_i = 1\} = k - x + j, j \in \{0, \ldots, x-1\}$. Because of I'', a should not be too large. We proceed by cases according to the value of j.

If $j > 0$, $f(S) \geq f(S)_{|I''} = mZ(a + \Sigma_{S_i \neq 1} \frac{1}{S_i})$. Given that the function $t(x) = \frac{1}{x}$ is convex, we know that $\Sigma_{S_i \neq 1} t(S_i) \geq (x - j)t(\frac{\Sigma_{S_i \neq 1} S_i}{x-j}) = (x - j)t(\frac{2x-j}{x-j})$, which implies $f(S) \geq mZ(k - x + j + \frac{(x-j)^2}{2x-j})$. Hence,

$$f(S) - T \geq m(Z(j + \frac{(x-j)^2}{2x-j} - \frac{x}{2}) - \frac{n\alpha}{2})$$

$$\begin{aligned} f(S) - T &\geq m(Z(\frac{xj}{2(2x-j)}) - \frac{n\alpha}{2}) \\ &\geq m(\frac{Z}{4} - \frac{n\alpha}{2}) \text{ because } (j \geq 1) \end{aligned}$$

Thus, we define $Z = 2n\alpha + 1$, and we get $f(S) - T > 0$. So in this first case, the considered solution S is strictly over the threshold value T. In the other case ($j = 0$), $f(S) = f(S)_{|I'} + f(S)_{|I''} \geq (n-1)m\frac{\alpha}{2} + m\alpha + f(S^1)_{|I''}$. Then, $f(S) - T \geq \frac{\alpha}{2} > 0$. In both cases, if there is no vertex cover of size x, the cost of any solution S is strictly greater than T which implies $Opt > T$.

□

Notice that in Theorem 12.1 we proved the NP hardness using a linear cost function, which means that for all i, j and for every number of allocated resources $s \in \{1, \ldots, m\}$, $C(h_i, I_j, s) = C(h_i, I_j, m)\frac{m}{s}$. This theorem implies of course that the r-dRSSP is also NP hard, through the "natural" reduction which consists in writing explicitly the m values for each heuristic and each instance. However, one could argue that it is sufficient in the linear case to only give the $C(h_i, I_j, m)$ values, implying that the input could be described using only $O(nk \log(Max_{(y1,y2)}(C(y1, y2, m))) + \log(m))$ bits. Then, the previous reduction from lr-dRSSP to r-DRSSP would not be polynomial in $\log(m)$, but only in m. However, this $O(m)$ dependencies is not important because Theorem 12.1 even proves that lr-dRSSP is unary NP hard.

12.2.3 A first greedy algorithm : mean-allocation (MA)

To solve r-dRSSP, let us now analyze a greedy algorithm which will serve as a basis for more sophisticated approximations presented in the next section. We consider the algorithm **mean-allocation (MA)**, which consists in allocating $\lfloor \frac{m}{k} \rfloor$ processors to each heuristic.

Let us now introduce some new definitions and notations, given a fixed valid solution S (not necessarily produced by MA), and a fixed optimal solution S^*.

Définition 12.2. *Let $\sigma(j) = argmin_{1 \leq i \leq k} C(h_i, I_j, S_i)$ be the index of the heuristic which finds the solution first for the instance j in S (ties are broken arbitrarily). Let define in the same way $\sigma^*(j)$ as the index of the used heuristic for the instance j in S^*.*

Définition 12.3. *Let $T(I_j) = C(h_{\sigma(j)}, I_j, S_{\sigma(j)})$ be the processing time of instance j in S. Let define in the same way $T^*(I_j)$ as the processing time of instance j in S^*.*

Proposition 12.4. ***MA*** *is a k-approximation for r-dRSSP.*

Proof Let a and b in $\mathbb{N}$ such that $m = ak + b, b < k$. Notice that $m \geq k$, otherwise there are no feasible solutions where each heuristic has at least one processor. Hence, a is greater or equal to 1.

For any instance $j \in \{1, .., n\}$, we have $T(I_j) = C(h_{\sigma(j)}, I_j, S_{\sigma(j)}) \leq C(h_{\sigma^*(j)}, I_j, S_{\sigma^*(j)})$ by definition of $T(I_j)$. In the worst case, $S_{\sigma^*(j)} \leq S^*_{\sigma^*(j)}$, implying $C(h_{\sigma^*(j)}, I_j, S_{\sigma^*(j)}) \leq \frac{S^*_{\sigma^*(j)}}{S_{\sigma^*(j)}} C(h_{\sigma^*(j)}, I_j, S^*_{\sigma^*(j)}) \leq \frac{m-(k-1)}{S_{\sigma^*(j)}} T^*(I_j)$ because in the worst case, the considered optimal solution allocates the maximum possible number of resources to heuristic $\sigma^*(j)$. Finally,

$$\frac{m-(k-1)}{S_{\sigma^*(j)}} T^*(I_j) = \frac{ak+b-(k-1)}{a} T^*(I_j) \leq kT^*(I_j).$$

□

We will now study how this algorithm can be improved thanks to the use of an oracle.

12.3 Approximation schemes based on oracle

12.3.1 Introduction

In this section, we provide approximation schemes for r-dRSSP using the oracle formalism : we first assume the existence of a reliable oracle that can provide some extra information for each instance, which allows to derive good approximation ratios, and finally we will enumerate the set of the possible oracle answers to get a "classical" algorithm (without oracle). The concept of adding some quantifiable information to study what improvements can be derived exists in other fields than optimization. Let us briefly review these oracle techniques.

In the distributed context [10], a problem is called *information sensitive* if a few bits of information enable to decrease drastically the execution time. The information sensitiveness is used to classify problems by focusing on lower bounds on the size of advice necessary to reach a fixed performance, or giving explicitly oracle information and studying the improvement (like in [10] and [11]).

In the on-line context, this quantifiable oracle information could be recognized in the notion of "look-ahead". The look-ahead could be defined as a slice of the future which is revealed to the on-line algorithm. Thus, it is possible to prove lower and upper bounds depending on the size of this slice [1, 24].

In the context of optimization, some polynomial time approximation schemes have been designed thanks to the guessing technique [18, 17]. This technique can be decomposed in two steps : proving an approximation ratio while assuming that a little part of the optimal solution is known, and finally enumerating all the possibilities for this part of the optimal solution.
In this section, we show that by choosing "correctly" the asked information, it is possible to derive very good approximation ratio, even with MA as a basis. Moreover, we will not only apply the guessing technique, but have a methodological approach by proving that the question asked to the oracle is somehow "the best" possible. At last, we improve the obtained approximation scheme by using the guess approximation technique [6]. More precisely, we provide (for any $g \in \{1, \ldots, k-1\}$ and an execution time in $O(kn)$) a $(k-g)$-approximation with an information of size* $g\log(m)$, and a $\frac{k}{g+1}$-approximation with an information of size $g(\log(k)+\log(m))$. Then, we prove that no information of "the same type" (coupled with MA) could lead to a better ratio than $\frac{k-g}{g+1}$. Finally, aiming at reducing the size of the oracle answer, we provide a ρ-approximation with an information of size $g(\log(k)+j_1+\log(\log(m)))$, where ρ is defined as :

$$\rho = \frac{k + \frac{g}{2^{j_1-1}}}{g+1}.$$

12.3.2 Choosing an arbitrary subset of heuristics

As a first step, we choose arbitrarily g heuristics (denoted by $\{h_{i_1}, \ldots, h_{i_g}\}$ and called "the guessed heuristics") among the k available heuristics. In the first guess $\tilde{G}_1$,

*As the encoding of the instance is fixed, all the information sizes are given exactly, without using the O notation.

the oracle provides a part of an optimal solution : the oracle gives the number of processors allocated to these g heuristics in an optimal solution of r-dRSSP.

Définition 12.5 (Guess 1). *Let $\tilde{G_1} = (S^*_{i_1}, \ldots, S^*_{i_g})$, for a fixed subset $(i_1, \ldots, i_g)$ of g heuristics and a fixed optimal solution S^*.*

Notice that this guess can be encoded using $|\tilde{G_1}| = g\log(m)$ bits. We first introduce some notations : let $k' = k - g$ be the number of remaining heuristics, $s = \Sigma_{l=1}^{g} S^*_{i_l}$ the number of processors used in the guess, and $m' = m - s$ the number of remaining processors. We also define $(a', b') \in \mathbb{N}^2$ such that $m' = a'k' + b', b' < k'$.

Let us consider a first algorithm MA^G which, given any guess $G = [(i_1, \ldots, i_g)(X_1, \ldots, X_g)], X_i \geq 1$, allocates X_l processors to heuristic $h_{i_l}, l \in \{1, \ldots, g\}$, and applies MA on the k' other heuristics with the m' remaining processors. This algorithm used with $G_1 = [(i_1, \ldots, i_g)(\tilde{G_1})]$ leads to the following ratio.

Proposition 12.6. *MA^{G_1} is a $(k-g)$-approximation for r-dRSSP, for any $g \in \{0 \ldots k-1\}$.*

Proof First, remark that MA^{G_1} produces a valid solution because we know that $a' \geq 1$ (there is at least one processor per heuristic in the considered optimal solution). Then, for any instance j treated by a guessed heuristic in the considered optimal solution ($\sigma^*(j) \in \{1, \ldots, g\}$), MA^{G_1} is at least as good as the optimal. For the other instances, the analysis is the same as for MA, and leads to the desired ratio.

□

Corollary 12.7. *There is an $(k-g)$-approximation for r-dRSSP which runs in $O(m^g * kn)$, for any $g \in \{0 \ldots k-1\}$.*

Proof We simply enumerate all the possible answers of the oracle, and choose the best solution, incurring a cost in $O(2^{|\tilde{G_1}|} * kn)$. □

In the following, instead of choosing an arbitrary subset of g heuristics, we will look for what could be the "best" properties to ask for.

12.3.3 Choosing a convenient subset of heuristics

In this section we define a new guess, which is larger than $\tilde{G_1}$, but leads to a better approximation ratio. As shown in [5], the "difficult" instances seem to be the ones where the optimal solution uses only a few heuristics (meaning that the major part of the computation time is only due to these few heuristics). For example, the worst cases of MA and MA^{G_1} occur when the optimal uses only one heuristic and allocates almost all the resources to it. Hence, we are interested in the most useful heuristics and we introduce the following definition. For any heuristic $h_i, i \in \{1, .., k\}$, let $T^*(h_i) = \Sigma_{j/\sigma^*(j)=i} T^*(I_j)$ be the "useful" computation time of heuristic i in the solution S^*. We define the second guess as follows.

Définition 12.8 (Guess 2). *Let $G_2 = [(i_1^*, \ldots, i_g^*), (S_{i_1^*}^*, ..., S_{i_g^*}^*)]$, be the number of processors allocated to the g most efficient heuristics (which means $T^*(h_{i_1^*}) \geq .. \geq T^*(h_{i_g^*})$ and $T^*(h_{i_g^*}) \geq T^*(h_i), \forall i \notin \{i_1^*, .., i_g^*\}$) in a fixed optimal solution S^*.*

Notice that this guess can be encoded using $|G_2| = g(\log(k)+\log(m))$ bits to indicate which subset of g heuristics must be chosen, and the allocation of the heuristics. Thanks to this larger guess, we derive the following better ratio.

Proposition 12.9. *MA^{G_2} is a $\frac{k}{g+1}$-approximation for r-dRSSP.*

Proof For the sake of clarity, we can assume without loss of generality that the heuristics indicated by the oracle are the g first one, meaning that $(i_1^*, \ldots, i_g^*) = (1, \ldots, g)$. The proof with an arbitrary cost function is structured as the one in [5] with linear cost assumption. Let $X^+ = \{i \in \{g+1, \ldots, k\} | S_i > S_i^*\}$ be the set of "non-guessed" heuristics for which MA^{G_2} allocated more than the optimal number of resources. We define in the same way $X^- = \{i \in \{g+1, \ldots, k\} | S_i \leq S_i^*\}$.

For all the instances j such that $\sigma^*(j)$ is in $\{1, \ldots, g\}$ or X^+, we have $T(I_j) \leq T^*(I_j)$ as MA^{G_2} allocates at least the number of resources used in the optimal for these instances. For j such that $\sigma^*(j)$ is in X^-, we have $T(I_j) \leq \frac{S_i^*}{S_i} T^*(I_j)$ since the cost function is not super-linear. Thus, summing over all heuristics we get :

$$\begin{aligned} T_{MA^{G_2}} &\leq \sum_{i=1}^{g} T^*(h_i) + \sum_{i \in X^+} T^*(h_i) + \sum_{i \in X^-} \frac{S_i^*}{S_i} T^*(h_i) \\ &= \sum_{i=1}^{k} T^*(h_i) + \sum_{i \in X^-} \left(\frac{S_i^*}{S_i} - 1 \right) T^*(h_i) \end{aligned}$$

$$T_{MA^{G_2}} \leq Opt + M \underbrace{\left(\frac{\sum_{i \in X^-} S_i^*}{a'} - card(X^-) \right)}_{\lambda}, \text{ with } M = \max_{i \in \{g+1, \ldots, k\}} (T^*(h_i))$$

Then, we claim that $\lambda \leq k' - 1$. Let $j = card\{X^-\}$. We can assume $j \geq 1$, otherwise MA^{G_2} is optimal. We use the same notations ($m' = a'k' + b'$ with $a' \geq 1$, $b' < k'$) as in the previous proof. The worst case occurs when the optimal solution allocates only one resource to each heuristic in X^-, leading to $\sum_{i \in X^-} S_i^* = m' - \sum_{i \in X^+} S_i^* \leq m' - (k' - j)$. Thus, $\lambda \leq \frac{m' - k' + j(1-a')}{a'} \leq \frac{a'k' + b' - (k'-1) - a'}{a'} \leq k' - 1$.

Moreover, $Opt = \Sigma_{i=1}^{g} T^*(h_i) + \Sigma_{i=g+1}^{k} T^*(h_i) \geq g T^*(h_g) + M \geq (g+1)M$. Finally, the ratio for MA^{G_2} is $r \leq 1 + \frac{k'-1}{g+1} = \frac{k}{g+1}$.

□

Corollary 12.10. *There is an $\frac{k}{g+1}$-approximation for r-dRSSP which runs in $O((km)^g * kn)$, for any $g \in \{0 \ldots k-1\}$.*

12.3.4 Tightness : comparison with the best subset of heuristics

In the previous subsection, we investigated a "convenient" property to decide which subset of heuristics should be asked to the oracle. By asking the allocation of a particular subset of g heuristics as proposed in Definition 12.8 (rather than an arbitrary subset of size g), we improved the $(k-g)$-approximation ratio to a $\frac{k}{g+1}$ ratio. This drastic improvement leads to a natural question : is there another property that would lead to really better approximation ratio than $\frac{k}{g+1}$? As we will explain in Proposition 12.11, we investigated this question and found that no property leads to a better ratio than $\frac{k-g}{g+1}$.

Proposition 12.11 (Tightness). *There is no selection property (i.e. a criteria to select for which subset of g heuristics we ask the optimal allocation to the oracle) that leads to a better ratio than* $\frac{k-g}{g+1}$*, even for a linear cost function.*

Proof To prove this statement, we construct an instance such that whatever the selected subset of g heuristics $(i_1, \ldots, i_g)$, the corresponding guess $G = [(i_1, \ldots, i_g)(S^*_{i_1}, \ldots, S^*_{i_g})]$ is such that $MA^G \geq \frac{k-g}{g+1} Opt$.

Let us define now this particular instance X, for fixed size of guess g (we use the same notation $k' = k - g$). We consider k heuristics, $m = (g+2)k' - 1$ resources, and a set $I = I' \cup I''$ of k instances, with $I' = \{I_1, \ldots, I_{g+1}\}$ and $I'' = \{I_{g+2}, \ldots, I_k\}$. Given that we are under the linear cost assumption, it is sufficient to only define the $C(h_i, I_j, m)$ values. The cost function is :

$$C(h_i, I_j, m) = \begin{cases} Z & \text{if } j \in \{1, \ldots, g+1\} \text{ and } i = j \\ \epsilon & \text{if } j \in \{g+2, \ldots, k\} \text{ and } i = j \\ \beta & \text{otherwise.} \end{cases}$$

We choose $\beta = mZ + 1$ so that for any solution S the cost for solving I_j only depends on S_j (because $\forall S, \sigma(j) = j$).

We claim that the optimal solution S^* for X_c allocates k' resources to heuristic $h_i, i \in \{1, \ldots, g+1\}$, and 1 resource $h_i, i \in \{g+2, \ldots, k\}$. Notice that the total number of allocated resources is equal to m. The cost of S^* is $f(S^*) = m((g+1)\frac{Z}{k'} + (k'-1)\epsilon)$. We will now prove that this solution is optimal with a similar argument as in the NP hardness proof : any "good" solution must allocate as many resources as possible (ie $(g+1)k'$) to solve the first $g+1$ instances, and moreover these resources must be evenly distributed to the first $g+1$ heuristics. More precisely, consider a solution $S = \{S_1, \ldots, S_k\}$ such that $\exists i_0 \in \{1, \ldots, g+1\} / S_{i_0} = k' - j, j \geq 1$. Then, $f(S) \geq f(S)_{|I'} = m(\frac{Z}{S_{i_0}} + \Sigma_{i \in \{1,\ldots,g+1\}, i \neq i_0} \frac{Z}{S_i})$. Using the same argument of convexity, we get $f(S) \geq mZ(\frac{1}{S_{i_0}} + g\frac{g}{gk'+j})$ Finally, $f(S) - f(S^*) \geq m(Z(\frac{1}{k'-j} + \frac{g^2}{gk'+j} - \frac{g+1}{k'}) - (k'-1)\epsilon) = m(Z\frac{j^2(g+1)}{k'(gk'+j)(k'-j)} - (k'-1)\epsilon) > 0$ for Z arbitrary large. Hence, S^* is an optimal solution for X.

Let us now analyze the cost of MA^G for any subset $\{i_1, \ldots, i_g\}$. Let $x = Card(\{i_1, \ldots, i_g\} \cap \{g+2, \ldots, k\})$. Notice that $0 \leq x \leq min(g, k'-1)$ The total number of resources in the guess is $s = (g-x)k' + x$, and $m' = (x+1)k' + k' - 1 - x$, implying $m' < (x+2)k'$. Then, the cost of MA^G (under the linear cost assumption) is $T_{MA^G} \geq m((g-x)\frac{Z}{k'} + (x+1)\frac{Z}{\alpha})$, with $\alpha \leq (x+1)$. Thus, $T_{MA^G} \geq m((g-x)\frac{Z}{k'} + Z) \geq mZ$,

and $\frac{T_{MA^G}}{Opt} \geq \frac{Z}{\frac{(g+1)Z}{k'}+(k'-1)\epsilon} \underset{Z\to\infty}{\longrightarrow} \frac{k'}{g+1}$.

□

Notice that the previous instance could be used to prove that the $\frac{k}{g+1}$ ratio of MA^{G_2} is tight. Indeed, with Z large enough we will have $G_2 = [(1, 2, \ldots, g-1, g), (k', \ldots, k')]$ and $\frac{T_{MA^{G_2}}}{Opt} \geq \frac{g\frac{Z}{k'}+Z}{(g+1)\frac{Z}{k'}+(k'-1)\epsilon} \underset{Z\to\infty}{\longrightarrow} \frac{k}{g+1}$.

12.3.5 Improvements with guess approximation

The idea of the guess approximation technique (introduced in [6]) is to find how to "contract" the answer of the oracle, without neither loosing too much information nor worsening the approximation ratio. We now look for what could be contracted here. In the case of guess 2, the oracle provides two types of information : a set of indices of heuristics (which verify a particular property) and a set of numbers of resources. The first type of information seems to be hard to approximate. Indeed, the information here is intrinsically binary, that is a given heuristic satisfies or not a given property. Thus, we are more interested in approximating the second part of the oracle information : the number of processors allocated to particular heuristics (in an optimal solution). Using the same notation as in Definition 12.8, let $\bar{G_2} = [(i_1^*, \ldots, i_g^*), (\bar{S_{i_1^*}^*}, ..., \bar{S_{i_g^*}^*})]$ be the contracted answer.

In order to define "correctly" the $\bar{S_{i_l}^*}$, let us consider the two following points. First, notice that it could be wrong to allocate more processors than the optimal for the guessed heuristics (*i.e.* $\sum_{l=1}^{g} \bar{S_{i_l}^*} > \sum_{l=1}^{g} S_{i_l}^*$), because there could be not enough remaining resources to allocate to the "non-guessed" heuristics, leading to an infeasible solution. Thus, the contraction can be chosen such that $\sum_{l=1}^{g} \bar{S_{i_l}^*} \leq \sum_{l=1}^{g} S_{i_l}^*$ (we will even choose $\bar{S_{i_l}^*} \leq S_{i_l}^*, \forall l$). Secondly, using only x resources instead of $x + \epsilon$ could be very bad if x is small.

Taking into account these two remarks, let us define the $\bar{S_{i_l}^*}$ as follows. We consider a mantissa-exponent representation for the $S_{i_l}^*$, with a fixed size† $j_1 \in \{1, .., \lceil \log(m) \rceil\}$ of mantissa. Thus, $S_{i_l}^* = a_{i_l} 2^{e_{i_l}} + b_{i_l}$, where a_{i_l} is encoded on j_1 bits, $0 \leq e_{i_l} \leq \lceil \log(m) \rceil - j_1$, and $b_{i_l} \leq 2^{e_{i_l}} - 1$. Then, we define $\bar{S_{i_l}^*} = a_{i_l} 2^{e_{i_l}}$. Notice that the length of the approximated guess becomes $|\bar{G_2}| = \Sigma_{l=1}^{g}(\log(i_l) + |a_{i_l}| + |e_{i_l}|) \leq g(\log(k) + j_1 + \log(\log(m)))$. We now study the approximation ratio derived with the same algorithm as previously.

Proposition 12.12. *$MA^{\bar{G_2}}$ is a $\frac{k+\frac{g}{2^{j_1}-1}}{g+1}$-approximation for r-dRSSP.*

Proof Again, let us assume without loss of generality that $(i_1^*, \ldots, i_g^*) = (1, \ldots, g)$. We first prove that $\forall i \in \{1, \ldots, g\}, S_i^*/\bar{S_i^*} \leq \beta$, with $\beta = 1 + \frac{1}{2^{j_1}-1}$. If $S_i^* \leq 2^{j_1} - 1$, we have $\bar{S_i^*} = S_i^*$ because S_i^* can be written with j_1 bits. Otherwise, $S_i^*/\bar{S_i^*} = \frac{a_i 2^{e_i} + b_i}{a_i 2^{e_i}} \leq 1 + \frac{1}{a_i} \leq 1 + \frac{1}{2^{j_1}-1} = \beta$.

Then, the proof can be directly adapted from Proposition 12.9 (we use the same

†In all the paper, log denote the $\log_2$ function

notation X^+ and X^-) :

$$\begin{aligned} T_{MA^{\bar{G}_2}} &\leq \beta \sum_{i=1}^{g} T^*(h_i) + \sum_{i \in X^+} T^*(h_i) + \sum_{i \in X^-} \frac{S_i^*}{S_i} T^*(h_i) \\ &= \beta \sum_{i=1}^{k} T^*(h_i) + \sum_{i \in X^+} (1-\beta) T^*(h_i) + \sum_{i \in X^-} (\frac{S_i^*}{S_i} - \beta) T^*(h_i) \\ &\leq \beta Opt + M \underbrace{(\frac{\sum_{i \in X^-} S_i^*}{a'} - \beta card(X^-))}_{\lambda}, \text{ with } M = max_{i \in \{g+1,\ldots,k\}}(T^*(h_i)) \end{aligned}$$

For the same reason as in Proposition 12.9, we have $\lambda \leq k' - \beta$. Then, the ratio of $M\bar{A}^{G_2}$ is $r \leq \beta + \frac{k'-\beta}{g+1} = \frac{k + \frac{g}{2^{j_1}-1}}{g+1}$. □

Corollary 12.13. *There is an* $\frac{k + \frac{g}{2^{j_1}-1}}{g+1}$*-approximation for r-dRSSP which runs in* $O((k2^{j_1}\log(m))^g * kn)$*, for any* $g \in \{0 \ldots k-1\}$ *and* $j_1 \in \{1, .., \lceil \log(m) \rceil\}$.

Notice that the MA^{G_1} algorithm could also be improved with the guess approximation technique. We would get a $\beta(k-g)$ ratio, using an information of size $j_1 + \log(\log(m))$. In our previous work, we also considered another naive algorithm (which simply redistributes a well chosen fraction of the resources given to the guessed heuristics to the others), however it does not bring any significant improvement over the algorithms presented above.

12.4 Experiments

Some preliminary experiments have been reported in [5] (comparing execution times for MA^{G_1} and MA^{G_2} using the linear cost assumption). Thus we will focus in the following experiments on two axes : studying the impact of the guess approximation on the execution times and returned values, and investigating an example of the non linear case. The complete description (including source codes) of the experiments is available on-line [7].

12.4.1 Description of the protocol

Inputs of r-dRSSP used for the experiments

We applied our algorithms on the satisfiability problem (SAT). The SAT problem consists in determining whether a formula of a logical proposition given as a conjunction of clauses is satisfied for at least one interpretation. Since this hard problem is very well known, there exist many heuristics that have been proposed to provide solutions for it.

Let us describe here how we constructed our inputs for the experiments. We used a SAT database (SatEx[‡]) which gives for a set of 23 heuristics (SAT solvers) and a

[‡] `http ://www.lri.fr/~simon/satex/satex.php3`

benchmark of 1303 instances for SAT the CPU execution times (on a single machine) of each heuristics on the 1303 instances of the benchmark. Thus we did not actually run these heuristics, but we used these CPU times to have a realistic matrix cost. The 1303 instances of the SatEx database are issued from many domains where the SAT problem are encountered. Some of them are : logistic planning, formal verification of microprocessors and scheduling. These instances are also issued from many challenging benchmarks for SAT which are used in one of the most popular annual SAT competition [§].

The SatEx database contains 23 heuristics issued from three main SAT solvers family [19]. These are :

- The DLL family with the heuristics : $asat$, $csat$, $eqsatz$, $nsat$, $sat-grasp$, $posit$, $relsat$, $sato$, $sato-3.2.1$, $satz$, $satz-213$, $satz-215$, $zchaff$;
- The DP family with : $calcres$, dr, $zres$;
- The randomized DLL family with : $ntab$, $ntab-back$, $ntab-back2$, $relsat-200$.

Other heuristics are $heerhugo$, $modoc$, $modoc-2.0$

We define thanks to these $1303 * 23$ values all the $C(h_i, I_j, m)$, for all i and j. In the linear case, these values are sufficient to entirely define the cost matrix. For the experiments concerning the non-linear case, we have chosen a logarithmic speedup, meaning that we defined $C(h_i, I_j, x) = \frac{C(h_i,I_j,1)}{\log(x)}$. All the considered inputs have the 1303 instances. For the inputs of r-dRSSP with k heuristics, $k < 23$, we randomly chose a subset of heuristics of size k (using an uniform distribution).

Description of the measures

There are two types of results : the returned values (the computed cost for solving the considered input) and the execution times. The only difficulty for measuring the returned values concerns MA^{G_1} and $MA^{\bar{G}_1}$. Indeed, recall that these algorithms use an arbitrary subset of g heuristics. Thus, the returned values could depend critically on the chosen subset. To avoid this problem we considered the mean cost of these algorithms over 30 experiments, while choosing randomly (using an uniform distribution) a subset of g heuristics for each experiment and keeping unchanged all the other parameters. The figures show the standard deviations, which were small in all the experiments.

Concerning the execution times, these experiments were done on a AMD Opteron 246 CPU with 2 cores and 2 GB of memory. For MA^{G_1} and $MA^{\bar{G}_1}$, we considered their mean execution times over the 30 experiments (the standard deviation was negligible).

12.4.2 Experiment 1

The goal of the first experience is to study the impact of the guess approximation technique on the returned values. Thus, the comparison are made between the cost of MA^{G_1}, MA^{G_2} and the one of $MA^{\bar{G}_1}$, $MA^{\bar{G}_2}$ for different values of g. The input of this experiment is constructed with the 23 heuristics, and $m = 100$ machines. The j_1 parameter used in the $MA^{\bar{G}_i}$ algorithm must be between 1 and 6. We decided to only show the results for two representative values of j_1, that is $j_1 = 1$ and $j_1 = 4$. In this case we assumed the linear cost assumption.

[§] `http ://www.satcompetition.org`

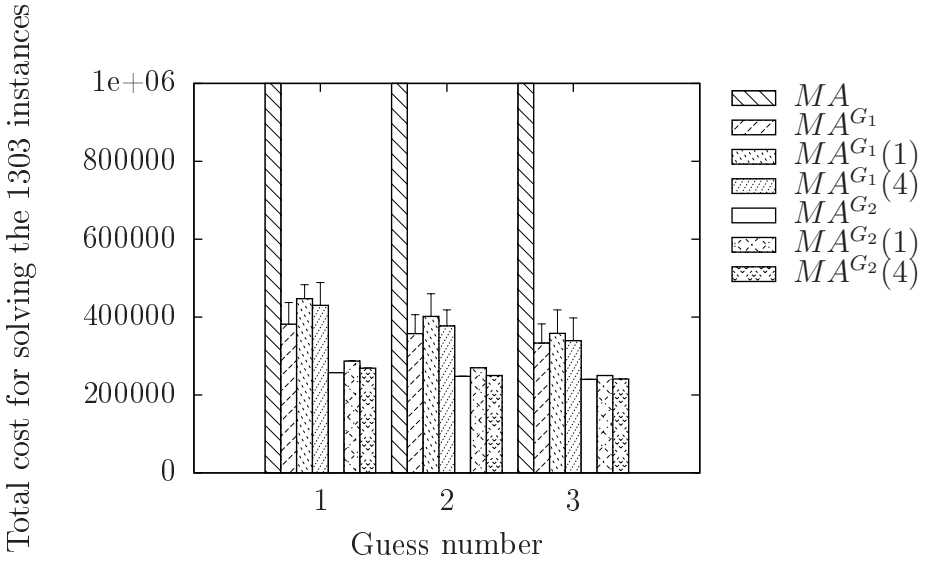

FIG. 12.14: Discrete Resource Sharing Cost with 23 heuristics and 100 resources

Figure 12.14 depicts the portfolio cost obtained in this experiment. $MA^{G_i}(x)$ in this picture corresponds to the algorithm $MA^{\bar{G}_i}$ with $j_1 = x$. As one can observe here, the new heuristics proposed are better than MA (which presented an absolute cost of 4253266, or 9.50 times worse than $MA^{G1}(1)$). Moreover, the guess approximation technique (applied in Section 12.3.5) does not degrade too much the quality of the solutions. Indeed, the ratio between the portfolio cost computed by $MA^{\bar{G}_1}(1)$ and MA^{G_1} for $g = 1$ is equal to 1.17. For MA^{G_2}, this ratio is equal to 1.11.

12.4.3 Experiment 2

In this experiment we are interested in comparisons with the optimal values and computation time. We compare the cost (and the execution time) of MA^{G_1}, MA^{G_2}, $MA^{\bar{G}_1}$ and $MA^{\bar{G}_2}$ with the optimal one for different values of g. Notice that the optimal solution is computed by simple enumeration, leading to a cost in $O(m^k)$. Thereby, the input of this experiment is constructed with only 6 (randomly chosen) heuristics¶, and $m = 50$ machines. We decided to only show the results for two representative values of j_1, that is $j_1 = 1$ and $j_1 = 3$. In this case we also assumed the linear cost assumption.

Figure 12.15 depicts the portfolio cost obtained in this experiment. One can observe again that it is interesting to restrict the subset of allocation for MA^{G_1} and MA^{G_2}. For instance, when guessing 4 heuristics, the ratio between the portfolio cost of $MA^{\bar{G}_2}(1)$ and the optimal algorithm is equal to 1.3, which is actually lower than the theoretical ratio equal to $\frac{6+4/2}{5} = 1.6$. Notice also that when guessing 4 heuristics with $j_1 = 3$, $MA^{\bar{G}_2}(3)$ finds the optimal portfolio cost. We also illustrate through this experiment the benefit of $MA^{\bar{G}_i}$ algorithms in considering their executions times. In Figure 12.17, we present the execution times of $MA^{\bar{G}_i}$, MA^{G_i} and the optimal algorithm. The $MA^{\bar{G}_i}$ algorithms in general provide a lower execution time than the other ones. For example,

¶For the sake of reproducibility, the chosen heuristics were : $csat$, $ntab - back2$, $modoc$, dr, $ntab$, $zchaff$

when guessing $g = 4$ heuristics, $MA^{\bar{G}_1}(1)$ is 44 times faster than $MA^{\bar{G}_1}(3)$, which is also 5.3 times faster than $MA^{\bar{G}_1}$. Similarly, $MA^{\bar{G}_2}(1)$ is 201 times faster than $MA^{\bar{G}_2}(3)$, which is also 1.5 times faster than $MA^{\bar{G}_2}$.

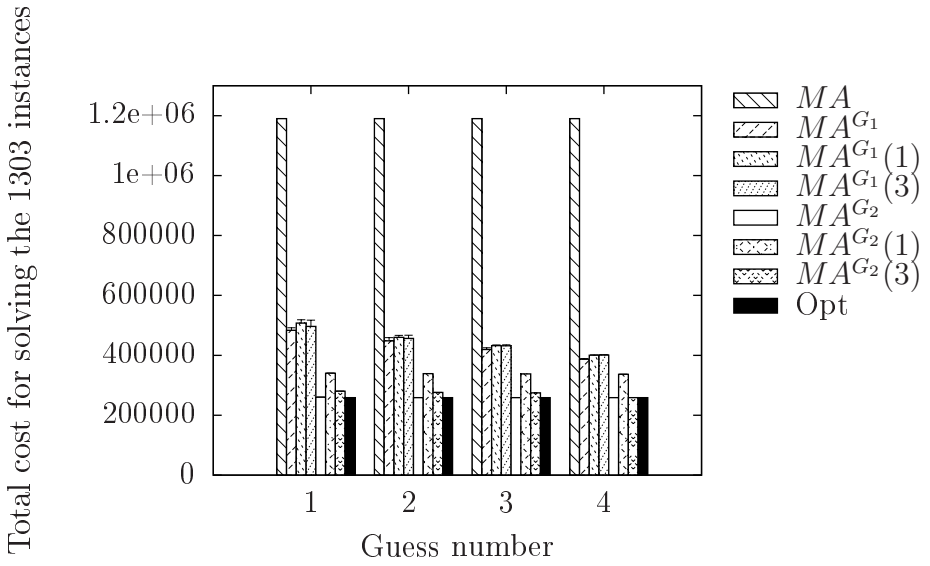

FIG. 12.15: Discrete Resource Sharing Cost with 6 heuristics and 50 resources (Linear case)

12.4.4 Experiment 3

This experiment is exactly the same as Experiment 2, except that we assume here the logarithmic cost function.

Figure 12.15 depicts the portfolio cost obtained in this experiment. This Figure shows that the relative behavior of the different algorithms does not change from the linear to the logarithmic case. Indeed, the $MA^{\bar{G}_i}$ algorithms compute a solution which is slightly worse than the MA^{G_i} ones. For instance the ratio between the portfolio cost of $MA^{\bar{G}_1}$ and MA^{G_1} for $g = 4$ and $j_1 = 1$ is equal to 1.02. For $MA^{\bar{G}_2}$, this ratio is equal to 1.03. The comparisons with the optimal solution are also tighter : the ratio between the portfolio cost of $MA^{\bar{G}_2}(1)$ (for $g = 4$) and the optimal algorithm is equal to 1.08. These ratios are lower than in the linear case. Indeed, if the cost function is "close to" a constant one, it seems reasonable that the gap between the solutions decreases.

12.5 Conclusion

In this work we extended our previous results [5] on the linear restricted discrete resource sharing scheduling problem. The main contributions are the complexity proof of lr-dRSSP, the extension (and the improvement using a guess approximation technique [6]) of previous approximation schemes [5] to the non-linear case, the tightness

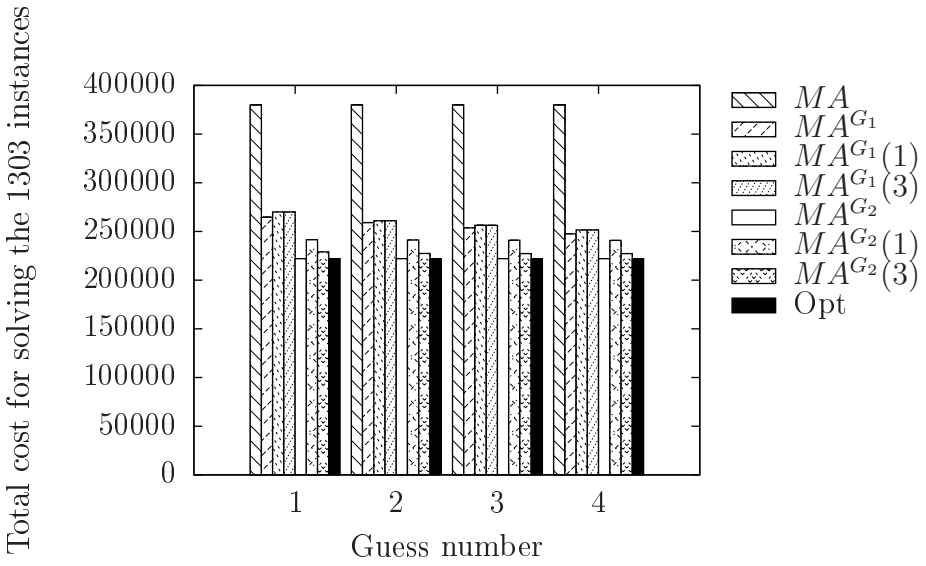

FIG. 12.16: Discrete Resource Sharing Cost with 6 heuristics and 50 resources (Non-linear case)

result that shows that the question asked to the oracle was well chosen, and the experiments for these new algorithms. There are many perspectives for continuing this work, namely the proposal of a model and new heuristics for the case of heterogeneous resources and the study of a mixed problem with both resource sharing and time switching.

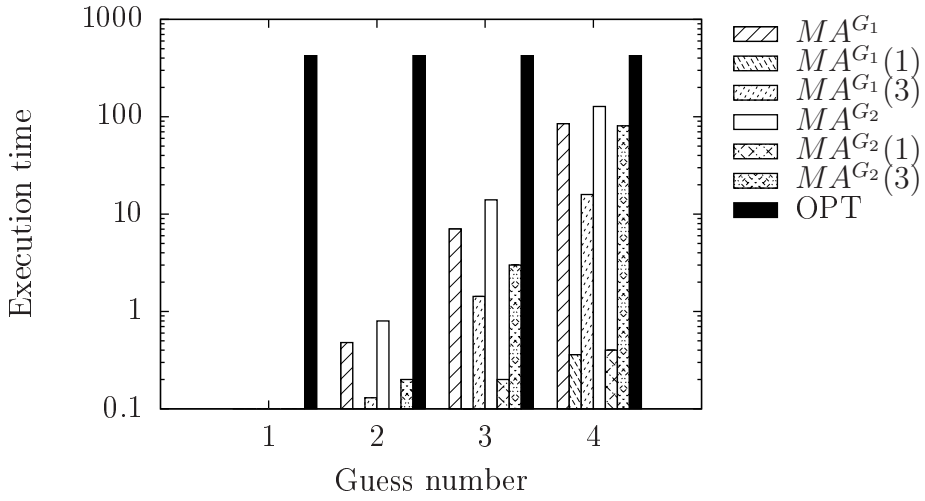

FIG. 12.17: Execution time with 6 heuristics and 50 resources (Linear case)

Bibliographie

[1] Susanne Albers. On the influence of lookahead in competitive paging algorithms. *ALGORITHMICA*, 18 :283–305, 1997.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL : An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208, 2003.

[3] S. Bhowmick, V.Eijkhout, Y.Freund, E. Fuentes, and D. Keyes. Application of machine learning to the selection of sparse linear solvers. `www.tacc.utexas.edu~eijkhout/Articles/2006-bhowmick.pdf`, 2006.

[4] M. Bougeret, P-F. Dutot, A. Goldman, Y. Ngoko, and D. Trystram. Approximating the discrete resource sharing scheduling problem. *International Journal of Foundations of Computer Science (IJFCS) (extension of the APDCM paper)*, 2009.

[5] M. Bougeret, P-F. Dutot, A. Goldman, Y. Ngoko, and D. Trystram. Combining multiple heuristics on discrete resources. In *11th Workshop on Advances in Parallel and Distributed Computational Models APDCM, (IPDPS)*, 2009.

[6] M. Bougeret, P-F. Dutot, and D. Trystram. The guess approximation technique and its application to the discrete resource sharing scheduling problem. In *Models and Algorithms for Planning and Scheduling Problems (MAPSP)*, 2009.

[7] Marin Bougeret, Pierre-François Dutot, Alfredo Goldman, Yanik Ngoko, and Denis Trystram. Experiments on approximating the discrete resource sharing scheduling problem. `http ://moais.imag.fr/membres/yanik.ngoko/index.php ?view=experiment`, 2009.

[8] V.A. Cicirello. *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. PhD thesis, Carnegie Mellon University, 2003.

[9] J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, GE Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, et al. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3) :223–238, 2006.

[10] Pierre Fraigniaud, Cyril Gavoille, David Ilcinkas, and Andrzej Pelc. Distributed computing with advice : Information sensitivity of graph coloring. In *International Colloquium on Automata, Languages and Programming (ICALP)*, number 34, 2007.

[11] Pierre Fraigniaud, David Ilcinkas, and Andrzej Pelc. Oracle size : a new measure of difficulty for communication tasks. In *25th ACM Symposium on Principles Of Distributed Computing (PODC)*, Denver, Colorado, USA, 2006.

[12] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness. A Series of Books in the Mathematical Sciences*. WH Freeman and Company, San Francisco, Calif, 1979.

[13] C.P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1) :43–62, 2001.

[14] B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296) :51, 1997.

[15] H.M. Markowitz. The early history of portfolio theory : 1600-1960. *Financial Analysts Journal*, pages 5–16, 1999.

[16] T. Sayag, S. Fine, and Y. Mansour. Combining multiple heuristics. In Durand and Thomas, editors, *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *LNCS*, pages 242–253. Springer, 2006.

[17] P. Schuurman and G. Woeginger. Approximation schemes - a tutorial. In *Lectures on Scheduling*, 2000.

[18] H. Shachnai and T. Tamir. Polynomial time approximation schemes - a survey. *Handbook of Approximation Algorithms and Metaheuristics, Chapman & Hall, Boca Raton*, 2007.

[19] L. Simon and P. Chatalic. SATEx : a web-based framework for SAT experimentation. *Electronic Notes in Discrete Mathematics*, 9 :129–149, 2001.

[20] M. Streeter, D. Golovin, and S.F. Smith. Combining multiple heuristics online. In *Proceedings of the national conference on artificial intelligence*, volume 22, page 1197. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2007.

[21] S. Talukdar, L. Baerentzen, A. Gove, and P. De Souza. Asynchronous teams : Cooperation schemes for autonomous agents. *Journal of Heuristics*, 4(4) :295–321, 1998.

[22] S. Weerawarana, E.N. Houstis, J.R. Rice, A. Joshi, and C.E. Houstis. PYTHIA : a knowledge-based system to select scientific algorithms. *ACM Transactions on Mathematical Software*, 22(4) :447–468, 1996.

[23] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 14th international conference on Supercomputing*, pages 66–77. ACM New York, NY, USA, 2000.

[24] Feifeng Zheng, Yinfeng Xu, and E. Zhang. How much can lookahead help in online single machine scheduling. *Information Processing Letters*, 106, 2008.

Annexe A

Annexe

Cette annexe regroupe des définitions de base liées à la théorie de l'approximation des problèmes d'optimisation, ainsi que les définitions de certains problèmes classiques d'ordonnancement abordés.

A.1 Notations et définitions générales

On suppose connue la définition d'un problème d'optimisation (voir [Garey and Johnson, 1979]). Dans ce manuscrit, on notera autant que possible :

- Π un problème d'optimisation (offline monocritère)
- I une instance du problème considéré
- $|I|$ la longueur de l'instance (dans un codage "raisonnable")
- $S(I)$ (ou S) une solution solution réalisable de I
- $Opt(I)$ une solution optimale de I
- $v(S)$ la valeur de S (parfois simplement notée S)
- $A(I)$ la solution produite par l'algorithme A pour I
- $t(A(I))$ le temps d'exécution (assimilé au nombre d'opérations élémentaires) de A sur I

Rappelons également des définitions classiques liées à la théorie de l'approximation (voir par exemple [Hochbaum, 1997]).

Définition A.1. *Un algorithme A a un ratio d'approximation asymptotique r ssi il existe une constante $c > 0$ telle que pour toute instance I d'un problème Π on a*

- $\frac{v(A_P(I))}{Opt(I)} \leq r + c$ *si* Π *est un problème de minimisation*
- $\frac{v(A_P(I))}{Opt(I)} \geq r - c$ *si* Π *est un problème de maximisation*

Lorsque $c = 0$, on dit que A a un ratio d'approximation (absolu) r (ou que A est une r-approximation).

Définition A.2. *Un schéma d'approximation (resp. asymptotique) est une famille d'algorithmes $\mathcal{A}$ telle que pour tout $\epsilon > 0$, il existe $A_\epsilon \in \mathcal{A}$ de ratio (resp. asymptotique) $1 + \epsilon$.*

On dit qu'un schéma d'approximation est (en notant n la taille de l'instance) :

- *une PTAS (ou APTAS si le schéma est asymptotique) lorque A_ϵ est polynomial en n*

- *une EPTAS (ou AEPTAS si le schéma est asymptotique) lorque* A_ϵ *est en* $\mathcal{O}(f(\epsilon)n^c)$ *avec* f *quelconque et* c *constante indépendante de* ϵ
- *une FPTAS (ou AFPTAS si le schéma est asymptotique) lorque* A_ϵ *est polynomial en* n *et* $\frac{1}{\epsilon}$

Remarquons que ces appellations dénotent plutôt traditionnellement des classes de problèmes, *PTAS* désignant par exemple l'ensemble des problèmes admettant une *PTAS* au sens ci-dessus.

A.2 Définitions de problèmes classiques en ordonnancement

A.2.1 Problèmes avec des tâches séquentielles

Nous utiliserons la notation à trois champs (par exemple $P||C_{max}$) introduite dans [Graham et al., 1979] pour décrire de nombreux problèmes d'ordonnancement. Pour tous ces problèmes n désignera le nombre de tâches, m le nombre de machines, et p_j la taille de la tâche j. Rappelons rapidement certains paramètres de cette notation à trois champs. Le premier champ (P, Q, ou R) désigne le type de machines disponibles. Ordonnancer une tâche j sur une machine i nécessite un temps

- p_j dans l'environnement P (toutes les machines ont une vitesse de 1)
- $\frac{p_j}{s_i}$ dans l'environnement Q (la machine i a une vitesse s_i)
- p_{ij} dans l'environnement R

Le champ du milieu décrit les contraintes supplémentaires, comme par exemple r_j pour signifier qu'une tâche n'est disponible qu'à partir de l'instant r_j, ou d_j pour signifier que la tâche doit être complétée avant l'instant d_j. Enfin, le dernier champ décrit la fonction objectif, principalement C_{max} signifiant minimiser la date de fin de la dernière tâche, ou L_{max} signifiant minimiser le retard maximal (lorsque les tâches ont des d_j associés).

Nous dénoterons *Knapsack* le problème du $0-1$ knapsack, dans lequel étant donné n objets de prix $p_i \in \mathbb{N}$ et taille $s_i \in \mathbb{N}$, $i \in \{1, \ldots, n\}$, et un sac de taille C, l'objectif est :

$$\begin{array}{rc} \max & \sum_{i=1}^{n} p_i x_i \\ t.q. & \sum_{i=1}^{n} s_i x_i \leq C \\ & x_i \in \{0,1\} \end{array}$$

Nous noterons *Multi Knapsack* l'extension de ce problème avec k sac à dos, *a priori* de capacitées différentes.

Du point de vue des algorithmes classiques pour ces problèmes, on peut noter deux algorithmes gloutons bien connus. Pour les problèmes d'ordonnancement, les algorithmes dits de liste ("List Scheduling", noté LS) [Graham, 1966] sont des algorithmes qui ordonnancent une tâche (quelconque) dès qu'une machine devient libre. N'importe quel algorithme de liste est une $2-\frac{1}{m}$-approximation pour $P||C_{max}$, et la variante notée *LPT* (pour Longest Processing Time), qui choisit la tâche à ordonnancer en prenant

la plus longue restante, atteint un ratio de $\frac{4}{3} - \frac{1}{3m}$. Pour le problème *Knapsack*, l'algorithme glouton le plus standard est celui qui trie par ordre décroissant les objets selon les $\frac{p_j}{s_j}$, et qui place à chaque pas le premier (dans l'ordre précédent) objet possible (ou qui place uniquement l'objet le plus cher si cette solution est meilleure). Cet algorithme est une 2-approximation (en prenant le maximum entre la solution décrite précédemment et l'objet le plus cher).

A.2.2 Problèmes avec des tâches parallèles

Définitions

Nous allons ici définir plusieurs variantes de problèmes d'ordonnancement de tâches parallèles, c'est-à-dire utilisant plusieurs processeurs. Parmi les différents modèles de tâches parallèles (rigides, moldables, malléables, *etc.*), nous considérerons seulement le modèle des tâches rigides dans lequel une tâche j doit s'exécuter simultanément sur q_j processeurs pendant p_j unités de temps, p_j et q_j étant fixés dans l'instance. Nous nous intéresserons au problème noté MSP (pour "Multiple Strip Packing") d'ordonnancement de n tâches rigides sur N clusters (le cluster i possédant m_i processeurs), visant à minimiser le makespan. Bien évidemment, une tâche ne peut être exécutée en même temps sur deux clusters différents, *i.e.* les q_j processeurs doivent appartenir au même cluster.

Une première variante est possible selon la "taille" des clusters. Nous noterons donc MSP_r (r désignant *régulier*) le cas particulier où tous les m_i sont égaux, et MSP_d (d désignant *différents*) le cas général. Une deuxième variante vient d'une éventuelle contrainte sur l'allocation des tâches. On parlera de la variante *contiguë* (ou continue) d'un problème lorsque les q_j processeurs doivent être alloués sur des processeurs d'indices contigus. Cette contrainte n'a en général pas de sens en ordonnancement "pur" (utiliser pour $q_j = 3$ les processeurs 1, 3, 7 ou 1, 2, 3 est équivalent), mais à un sens lorsque ces problèmes concernent la disposition de *rectangles* de hauteur q_j et longueur p_j (appelés problèmes de *strip packing*). Les variantes contiguës et non contiguës du MSP seront respectivement notées MSP^{nc} et MSP^{c}. Ainsi, quatre variantes de MSP sont possibles.

Remarques

On peut formuler quelques remarques sur l'application d'un résultat en non contigu au cas contigu (et réciproquement). Un algorithme pour le cas non contigu ne s'applique pas *a priori* au cas contigu, puisque évidemment les contraintes de contiguïté ne sont pas forcément respectées. Dans l'autre sens, même si les algorithmes s'appliquent, les ratios d'approximation ne sont pas nécessairement conservés. En effet, une r-approximation de l'optimal contigu $Opt_c(I)$ n'est pas forcément une r-approximation de l'optimal non contigu $Opt(I)$, puisque $Opt(I) \leq Opt_c(I)$. Ainsi, le cas non contigu et le cas contigus sont *a priori* complètement séparés. Cependant, trois remarques sont nécessaires pour modérer cette conclusion.

Tout d'abord, les résultats en contigu s'appliquent en général au non contigu, puisque dans les preuves on compare généralement l'algorithme à $Opt(I)$ (via par exemple des arguments d'aire) plutôt qu'à $Opt_c(I)$. De plus, l'écart entre $Opt(I)$ et $Opt_c(I)$ est difficile à caractériser. On peut trouver des contre-exemples [Dutot, 2004]

pour un cluster avec $m = 4$ processeurs pour lesquels $Opt(I) < Opt_c(I)$, où démontrer que $Opt(I) = Opt_c(I)$ pour 3 processeurs, mais le cas général semble peu connu. Enfin, on peut généralement dire que le cas contigu est le plus difficile. Comme évoqué précédemment, il est difficile de se comparer à $Opt_c(I)$ au lieu de $Opt(I)$, et on construit donc souvent une solution contiguë approximant l'optimal non contigu. De plus, on peut remarquer que les techniques d'analyse simples (de type List Scheduling) échouent en contigu. En effet, l'argument classique consistant à dire que "ne pas pouvoir ordonnancer une tâche j à un instant t implique qu'au moins $m - q_j$ processeurs sont actifs (et donc que l'utilisation du cluster est bonne)" n'est valable qu'en non contigu. D'ailleurs, l'algorithme de liste de Graham [Garey and Graham, 1975] est une 2-approximation pour un cluster en non contigu, alors que des techniques plus sophistiquées ont dû être développées [Steinberg, 1997, Schiermeyer, 1994] pour atteindre un ratio de 2 en contigu.

Bibliographie

[Albers, 1997a] Albers, S. (1997a). Better bounds for online scheduling. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, page 139. ACM.

[Albers, 1997b] Albers, S. (1997b). On the influence of lookahead in competitive paging algorithms. *ALGORITHMICA*, 18 :283–305.

[Alves and Clímaco, 2007] Alves, M. and Clímaco, J. (2007). A review of interactive methods for multiobjective integer and mixed-integer programming. *European Journal of Operational Research (EJOR)*, 180(1) :99–115.

[An et al., 2003] An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., and Rauchwerger, L. (2003). STAPL : An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208.

[Anderson, 2008] Anderson, D. (2008). http ://boinc.berkeley.edu/trac/wiki/BossaIntro.

[Anderson et al., 2000] Anderson, D., Anderson, E., Lesh, N., Marks, J., Mirtich, B., Ratajczak, D., and Ryall, K. (2000). Human-guided simple search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 209–216. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999.

[Angelelli et al., 2004] Angelelli, E., Nagy, A., Speranza, M., and Tuza, Z. (2004). The on-line multiprocessor scheduling problem with known sum of the tasks. *Journal of Scheduling*, 7(6) :421–428.

[Angelelli et al., 2003] Angelelli, E., Speranza, M., and Tuza, Z. (2003). Semi-Online Scheduling on Two Parallel Processors with an Upper Bound on the Items. *Algorithmica*, 37(4) :243–262.

[Arora, 1995] Arora, S. (1995). Probabilistic checking of proofs and hardness of approximation problems. *University of California at Berkeley, Berkeley, CA*.

[Arora and Safra, 1998] Arora, S. and Safra, S. (1998). Probabilistic checking of proofs : A new characterization of NP. *Journal of the ACM (JACM)*, 45(1) :122.

[Awerbuch et al., 1990] Awerbuch, B., Goldreich, O., Vainish, R., and Peleg, D. (1990). A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2) :238–256.

[Babai, 1985] Babai, L. (1985). Trading group theory for randomness. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, page 429. ACM.

[Bansal et al., 2009a] Bansal, N., Caprara, A., Jansen, K., Pr
"adel, L., and Sviridenko, M. (2009a). A structural lemma in 2-dimensional packing, and its implications on approximability. *Algorithms and Computation*, pages 77–86.

[Bansal et al., 2009b] Bansal, N., Caprara, A., Jansen, K., Prädel, L., and Sviridenko, M. (2009b). How to maximize the total area of rectangle packed into a rectangle. *In Preparation.*

[Bansal et al., 2007] Bansal, N., Han, X., Iwama, K., Sviridenko, M., and Zhang, G. (2007). Harmonic algorithm for 3-dimensional strip packing problem. In *Proceedings of the eighteenth ACM-SIAM symposium on Discrete algorithm (SODA)*, pages 1197–1206.

[Bhowmick et al., 2006] Bhowmick, S., V.Eijkhout, Y.Freund, Fuentes, E., and Keyes, D. (2006). Application of machine learning to the selection of sparse linear solvers. `www.tacc.utexas.edu~eijkhout/Articles/2006-bhowmick.pdf`.

[Borodin and El-Yaniv, 1998] Borodin, A. and El-Yaniv, R. (1998). *Online computation and competitive analysis.* Cambridge University Press Cambridge.

[Bougeret et al., 2009a] Bougeret, M., Dutot, P.-F., Goldman, A., Ngoko, Y., and Trystram, D. (2009a). Approximating the discrete resource sharing scheduling problem. *International Journal of Foundations of Computer Science (IJFCS) (extension of the APDCM paper).*

[Bougeret et al., 2009b] Bougeret, M., Dutot, P.-F., Goldman, A., Ngoko, Y., and Trystram, D. (2009b). Combining multiple heuristics on discrete resources. In *11th Workshop on Advances in Parallel and Distributed Computational Models APDCM, (IPDPS).*

[Bougeret et al., 2009c] Bougeret, M., Dutot, P.-F., Goldman, A., Ngoko, Y., and Trystram, D. (2009c). Experiments on approximating the discrete resource sharing scheduling problem. `http ://moais.imag.fr/membres/yanik.ngoko/index.php ?view=experiment`.

[Bougeret et al., 2009d] Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., and Trystram, D. (2009d). Approximation algorithm for multiple strip packing. In *Proceedings of the 7th Workshop on Approximation and Online Algorithms (WAOA).*

[Bougeret et al., 2010a] Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., and Trystram, D. (2010a). A fast 5/2-approximation for hierarchical scheduling. In *Proceedings of the 16th International European Conference on Parallel and Distributed Computing (EUROPAR).*

[Bougeret et al., 2010b] Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., and Trystram, D. (2010b). Approximating the non-contiguous multiple organization packing problem. In *Proceedings of the 6th IFIP International Conference on Theoretical Computer Science (TCS).*

[Bougeret et al., 2009e] Bougeret, M., Dutot, P.-F., and Trystram, D. (2009e). The guess approximation technique and its application to the discrete resource sharing scheduling problem. In *Models and Algorithms for Planning and Scheduling Problems (MAPSP).*

[Caprara, 2002] Caprara, A. (2002). Packing 2-dimensional bins in harmony. In *IEEE 43th Annual IEEE Symposium on Foundations of Computer Science (FOCS).*

[Chandra and Toueg, 1996] Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2) :225–267.

[Chekuri and Khanna, 2000] Chekuri, C. and Khanna, S. (2000). A PTAS for the multiple knapsack problem. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 213–222. Society for Industrial and Applied Mathematics.

[Chen, 1991] Chen, B. (1991). Tighter bound for MULTIFIT scheduling on uniform processors. *Discrete Applied Mathematics*, 31(3) :227–260.

[Cicirello, 2003] Cicirello, V. (2003). *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. PhD thesis, Carnegie Mellon University.

[Clementi et al., 2001] Clementi, A., Monti, A., and Silvestri, R. (2001). Selective families, superimposed codes, and broadcasting on unknown radio networks. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 709–718. Society for Industrial and Applied Mathematics.

[Coffman Jr et al., 1980] Coffman Jr, E., Garey, M., Johnson, D., and Tarjan, R. (1980). Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9 :808.

[Cole and Vishkin, 1986] Cole, R. and Vishkin, U. (1986). Deterministic coin tossing and accelerating cascades : micro and macro techniques for designing parallel algorithms. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 206–219. ACM.

[Coleman and Mao, 2002] Coleman, B. and Mao, W. (2002). Lookahead scheduling in a real-time context. In *Proceedings of the Sixth International Conference on Computer Science and Informatics, Durham, NC, USA*, pages 205–209.

[David and Borodin, 1990] David, S. and Borodin, A. (1990). A new measure for the study of online algorithms. *Manuscript*.

[Dinur, 2007] Dinur, I. (2007). The PCP theorem by gap amplification. *Journal of the ACM (JACM)*, 54(3) :12.

[Dongarra et al., 2006] Dongarra, J., Bosilca, G., Chen, Z., Eijkhout, V., Fagg, G., Fuentes, E., Langou, J., Luszczek, P., Pjesivac-Grbovic, J., Seymour, K., et al. (2006). Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3) :223–238.

[Dorrigiv and López-Ortiz, 2005] Dorrigiv, R. and López-Ortiz, A. (2005). A survey of performance measures for on-line algorithms. *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 36(3) :67–81.

[Dutot, 2004] Dutot, P. (2004). *Algorithmes d'ordonnancement pour les nouveaux supports d'exécution*. PhD thesis.

[Dutot et al., a] Dutot, P.-F., Mounié, G., and Trystram, D. Scheduling Parallel Tasks : Approximation Algorithms. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis, JY-T. Leung (Eds.)*.

[Dutot et al., b] Dutot, P.-F., Pascual, F., K., R., and D., T. Approximation algorithms for the multi-organization scheduling problem. *Submitted to : IEEE Transactions on Parallel and Distributed Systems (TPDS)*.

[Eberbach et al., 2004] Eberbach, E., Goldin, D., and Wegner, P. (2004). Turing's ideas and models of computation. *Alan Turing : Life and Legacy of a Great Thinker, ed. Christof Teuscher, Springer*.

[Emek et al., 2009] Emek, Y., Fraigniaud, P., Korman, A., and Rosén, A. (2009). Online computation with advice. *Automata, Languages and Programming*, pages 427–438.

[Englert et al., 2008] Englert, M., Ozmen, D., and Westermann, M. (2008). The power of reordering for online minimum makespan scheduling. In *IEEE 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*.

[Epstein and Sgall, 2004] Epstein, L. and Sgall, J. (2004). Approximation schemes for scheduling on uniformly related and identical parallel machines. *Algorithmica*, 39(1) :43–57.

[Feige et al., 1996] Feige, U., Goldwasser, S., Lovász, L., Safra, S., and Szegedy, M. (1996). Interactive proofs and the hardness of approximating cliques. *Journal of the ACM (JACM)*, 43(2) :268–292.

[Feitelson et al., 1997] Feitelson, D., Rudolph, L., Schwiegelshohn, U., Sevcik, K., and Wong, P. (1997). Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–34. Springer.

[Fernandez de la Vega and Lueker, 1981] Fernandez de la Vega, W. and Lueker, G. (1981). Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1(4) :349–355.

[Fich and Ruppert, 2003] Fich, F. and Ruppert, E. (2003). Hundreds of impossibility results for distributed computing. *Distributed computing*, 16(2) :121–163.

[Fishkin et al., 2008] Fishkin, A., Jansen, K., and Mastrolilli, M. (2008). Grouping techniques for scheduling problems : simpler and faster. *Algorithmica*, 51(2) :183–199.

[Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid : Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3) :200.

[Fraigniaud et al., 2007] Fraigniaud, P., Gavoille, C., Ilcinkas, D., and Pelc, A. (2007). Distributed computing with advice : Information sensitivity of graph coloring. In *International Colloquium on Automata, Languages and Programming (ICALP)*, number 34.

[Fraigniaud et al., 2006] Fraigniaud, P., Ilcinkas, D., and Pelc, A. (2006). Oracle size : a new measure of difficulty for communication tasks. In *25th ACM Symposium on Principles Of Distributed Computing (PODC)*, Denver, Colorado, USA.

[Friesen and Langston, 1983] Friesen, D. K. and Langston, M. A. (1983). Bounds for multifit scheduling on uniform processors. *SIAM Journal on Computing*, 12(1) :60–70.

[Fusco and Pelc, 2008] Fusco, E. and Pelc, A. (2008). Trade-offs between the size of advice and broadcasting time in trees. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 77–84. ACM.

[Garey and Graham, 1975] Garey, M. and Graham, R. (1975). Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2) :187–200.

[Garey and Johnson, 1979] Garey, M. and Johnson, D. (1979). *Computers and intractability. A guide to the theory of NP-completeness. A Series of Books in the Mathematical Sciences.* WH Freeman and Company, San Francisco, Calif.

[Gavoille et al., 2001] Gavoille, C., Peleg, D., Pérennes, S., and Raz, R. (2001). Distance labeling in graphs. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, page 219. Society for Industrial and Applied Mathematics.

[Ghosh, 2007] Ghosh, S. (2007). *Distributed systems : an algorithmic approach.* CRC press.

[Goldin et al., 2004] Goldin, D., Smolka, S., Attie, P., and Sonderegger, E. (2004). Turing machines, transition systems, and interaction. *Information and Computation*, 194(2) :101–128.

[Goldin and Wegner, 2008] Goldin, D. and Wegner, P. (2008). The interactive nature of computing : Refuting the strong Church–Turing thesis. *Minds and Machines*, 18(1) :17–38.

[Goldman et al., 1997] Goldman, S., Parwatikar, J., and Suri, S. (1997). On-line scheduling with hard deadlines. *Algorithms and Data Structures*, pages 258–271.

[Goldwasser et al., 1985] Goldwasser, S., Micali, S., and Rackoff, C. (1985). The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, page 304. ACM.

[Gomes and Selman, 2001] Gomes, C. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1) :43–62.

[Gonzalez et al., 1977] Gonzalez, T., Ibarra, O. H., and Sahni, S. (1977). Bounds for lpt schedules on uniform processors. *SIAM Journal on Computing*, 6(1) :155–166.

[Goubault-Larrecq,] Goubault-Larrecq, J. Classes de complexité randomisées. Polycopié de cours.

[Graham, 1966] Graham, R. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9) :1563–1581.

[Graham, 1969] Graham, R. (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17 :263–269.

[Graham et al., 1979] Graham, R., Lawler, E., Lenstra, J., and Rinnooy Kan, A. (1979). Optimization and approximation in deterministic sequencing and scheduling : a survey. *Discrete optimization : proceedings*, page 287.

[Hall and Shmoys, 1989] Hall, L. A. and Shmoys, D. B. (1989). Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 134–139, Washington, DC, USA. IEEE Computer Society.

[Harren et al., 2010] Harren, R., Jansen, K., Prädel, L., and Van Stee, R. (2010). A $5/3 + \epsilon$ approximation for strip packing. submitted.

[Håstad, 2001] Håstad, J. (2001). Some optimal inapproximability results. *Journal of the ACM (JACM)*, 48(4) :798–859.

[Hausmann et al., 1981] Hausmann, D., Kannan, R., and Korte, B. (1981). Exponential lower bounds on a class of knapsack algorithms. *Mathematics of Operations Research*, pages 225–232.

[Hochbaum, 1997] Hochbaum, D. (1997). Approximation algorithms for NP-hard problems. *ACM SIGACT News*, 28(2) :40–52.

[Hochbaum and Shmoys, 1987] Hochbaum, D. and Shmoys, D. (1987). Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1) :144–162.

[Hochbaum and Shmoys, 1988] Hochbaum, D. and Shmoys, D. (1988). A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM Journal on Computing*, 17(3) :539–551.

[Horowitz and Sahni, 1976] Horowitz, E. and Sahni, S. (1976). Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM (JACM)*, 23(2) :317–327.

[Huberman et al., 1997] Huberman, B., Lukose, R., and Hogg, T. (1997). An economics approach to hard computational problems. *Science*, 275(5296) :51.

[Jansen, 2009] Jansen, K. (2009). An EPTAS for scheduling jobs on uniform processors : using an MILP relaxation with a constant number of integral variables. *Automata, Languages and Programming*, pages 562–573.

[Jansen et al., 2009] Jansen, K., Prädel, L., and Schwarz, U. M. (2009). Two for one : Tight approximation of 2d bin packing. *Submitted to : Algorithms and Data Structures Symposium (WADS 2009)*.

[Jansen and Solis-Oba, 2006] Jansen, K. and Solis-Oba, R. (2006). An asymptotic approximation algorithm for 3d-strip packing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA)*, page 152. ACM.

[Jansen and Solis-Oba, 2007] Jansen, K. and Solis-Oba, R. (2007). New approximability results for 2-dimensional packing problems. *Lecture Notes in Computer Science*, 4708 :103.

[Jansen and Thöle, 2008] Jansen, K. and Thöle, R. (2008). Approximation algorithms for scheduling parallel jobs : Breaking the approximation ratio of 2. In *International Colloquium on Automata, Languages and Programming*, pages 234–245.

[Kellerer, 1999] Kellerer, H. (1999). A polynomial time approximation scheme for the multiple knapsack problem. *Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques*, pages 51–62.

[Kellerer and Pferschy, 1998] Kellerer, H. and Pferschy, U. (1998). A new fully polynomial approximation scheme for the knapsack problem. *Approximation Algorithms for Combinatiorial Optimization*, pages 123–134.

[Kenyon and Rémila, 2000] Kenyon, C. and Rémila, E. (2000). A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, pages 645–656.

[Kirkpatrick, 2009] Kirkpatrick, D. (2009). Hyperbolic dovetailing. *Algorithms-ESA*, pages 516–527.

[Klau et al., 2002] Klau, G., Lesh, N., Marks, J., and Mitzenmacher, M. (2002). Human-guided tabu search. In *Proceedings of the national conference on artificial interlligence*, pages 41–47. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999.

[Klau et al., 2010] Klau, G., Lesh, N., Marks, J., and Mitzenmacher, M. (2010). Human-guided search. *Journal of Heuristics*, 16(3) :289–310.

[Korte and Schrader, 1981a] Korte, B. and Schrader, R. (1981a). A survey on oracle techniques. *Mathematical Foundations of Computer Science 1981*, pages 61–77.

[Korte and Schrader, 1981b] Korte, B. and Schrader, R. (1981b). ON THE EXISTENCE OF FAST APPROXIMATION SCHEMES1. In *Nonlinear programming 4 : proceedings of the Nonlinear Programming Symposium 4*, page 415. Academic Press.

[Koutsoupias and Papadimitriou, 2001] Koutsoupias, E. and Papadimitriou, C. (2001). Beyond competitive analysis. *SIAM Journal on Computing*, 30(1) :300–317.

[Kowalski and Pelc, 2007] Kowalski, D. and Pelc, A. (2007). Optimal deterministic broadcasting in known topology radio networks. *Distributed Computing*, 19(3) :185–195.

[Lawler, 1979] Lawler, E. L. (1979). Fast approximation algorithms for knapsack problems. *Mathematics of Operation Research*, 4(4) :339–356.

[Lenstra et al., 1990] Lenstra, J., Shmoys, D., and Tardos, E. (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1) :259–271.

[Lenstra Jr, 1983] Lenstra Jr, H. (1983). Integer programming with a fixed number of variables. *Mathematics of operations research*, pages 538–548.

[Lesh et al., 2005] Lesh, N., Marks, J., McMahon, A., and Mitzenmacher, M. (2005). New heuristic and interactive approaches to 2D rectangular strip packing. *Journal of Experimental Algorithmics (JEA)*, 10 :1–2.

[Leung and Li, 2008] Leung, J. and Li, C. (2008). Scheduling with processing set restrictions : A survey. *International Journal of Production Economics*, 116(2) :251–262.

[Li and Wang, 2010] Li, C. and Wang, X. (2010). Scheduling parallel machines with inclusive processing set restrictions and job release times. *European Journal of Operational Research (EJOR)*, 200(3) :702–710.

[Marcotte and Soland, 1986] Marcotte, O. and Soland, R. M. (1986). An interactive branch-and-bound algorithm for multiple criteria optimization. *MANAGEMENT SCIENCE*.

[Markowitz, 1999] Markowitz, H. (1999). The early history of portfolio theory : 1600-1960. *Financial Analysts Journal*, pages 5–16.

[Mastrolilli, 2003] Mastrolilli, M. (2003). Efficient approximation schemes for scheduling problems with release dates and delivery times. *Journal of Scheduling*, 6(6) :521–531.

[Mastrolilli and Hutter, 2006] Mastrolilli, M. and Hutter, M. (2006). Hybrid rounding techniques for knapsack problems. *Discrete Applied Mathematics*, 154(4) :640–649.

[Miyazawa and Wakabayashi, 2000] Miyazawa, F. and Wakabayashi, Y. (2000). Cube packing. *LATIN 2000 : Theoretical Informatics*, pages 58–67.

[Mounié et al., 2007] Mounié, G., Rapine, C., and Trystram, D. (2007). A 3/2-dual approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM Journal on Computing*, 37(2) :401–412.

[O'Donnell, 2005] O'Donnell, R. (2005). A history of the PCP theorem. *Course notes on the PCP Theorem and Hardness of Approximation, Autumn.*

[Pascual et al., 2007] Pascual, F., Rzadca, K., and Trystram, D. (2007). Cooperation in multi-organization scheduling. In *Proceedings of the 13th International European Conference on Parallel and Distributed Computing (EUROPAR).*

[Peleg, 1999] Peleg, D. (1999). Proximity-preserving labeling schemes and their applications. In *Graph-Theoretic Concepts in Computer Science*, pages 30–41. Springer.

[Peleg, 2000a] Peleg, D. (2000a). *Distributed computing : a locality-sensitive approach.* Society for Industrial Mathematics.

[Peleg, 2000b] Peleg, D. (2000b). Informative labeling schemes for graphs. *Mathematical Foundations of Computer Science 2000*, pages 579–588.

[Peleg, 2000c] Peleg, D. (2000c). Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33 :167–176.

[Sahni, 1975] Sahni, S. (1975). Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 22(1) :115–124.

[Sayag et al., 2006] Sayag, T., Fine, S., and Mansour, Y. (2006). Combining multiple heuristics. In Durand and Thomas, editors, *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *LNCS*, pages 242–253. Springer.

[Schiermeyer, 1994] Schiermeyer, I. (1994). Reverse-fit : A 2-optimal algorithm for packing rectangles. *Lecture Notes in Computer Science*, pages 290–290.

[Schuurman and Woeginger, 2000] Schuurman, P. and Woeginger, G. (2000). Approximation schemes - a tutorial. In *Lectures on Scheduling.*

[Schwiegelshohn et al., 2008] Schwiegelshohn, U., Tchernykh, A., and Yahyapour, R. (2008). Online scheduling in grids. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10.

[Shachnai and Tamir, 2007] Shachnai, H. and Tamir, T. (2007). Polynomial time approximation schemes - a survey. *Handbook of Approximation Algorithms and Metaheuristics, Chapman & Hall, Boca Raton.*

[Shamir, 1992] Shamir, A. (1992). Ip= pspace. *Journal of the ACM (JACM)*, 39(4) :869–877.

[Simon and Chatalic, 2001] Simon, L. and Chatalic, P. (2001). SATEx : a web-based framework for SAT experimentation. *Electronic Notes in Discrete Mathematics*, 9 :129–149.

[Sleator and Tarjan, 1985] Sleator, D. and Tarjan, R. (1985). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2) :208.

[Steinberg, 1997] Steinberg, A. (1997). A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26 :401.

[Streeter et al., 2007] Streeter, M., Golovin, D., and Smith, S. (2007). Combining multiple heuristics online. In *Proceedings of the national conference on artificial intelligence*, volume 22, page 1197. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999.

[Talukdar et al., 1998] Talukdar, S., Baerentzen, L., Gove, A., and De Souza, P. (1998). Asynchronous teams : Cooperation schemes for autonomous agents. *Journal of Heuristics*, 4(4) :295–321.

[Tchernykh et al., 2006] Tchernykh, A., Ramírez, J., Avetisyan, A., Kuzjurin, N., Grushin, D., and Zhuk, S. (2006). Two level job-scheduling strategies for a computational grid. *Parallel Processing and Applied Mathematics*, pages 774–781.

[Trevisan, 2004] Trevisan, L. (2004). Inapproximability of combinatorial optimization problems. In *Proceedings of the Electronic Colloquium on Computational Complexity (ECCC)*. Citeseer.

[Von Ahn, 2007] Von Ahn, L. (2007). Human computation. In *Proceedings of the 4th international conference on Knowledge capture*, page 6. ACM.

[Von Ahn et al., 2006] Von Ahn, L., Liu, R., and Blum, M. (2006). Peekaboom : a game for locating objects in images. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, page 64. ACM.

[Weerawarana et al., 1996] Weerawarana, S., Houstis, E., Rice, J., Joshi, A., and Houstis, C. (1996). PYTHIA : a knowledge-based system to select scientific algorithms. *ACM Transactions on Mathematical Software*, 22(4) :447–468.

[Woeginger, 1999] Woeginger, G. (1999). When does a dynamic programming formulation guarantee the existence of an FPTAS ? In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 820–829. Society for Industrial and Applied Mathematics.

[Ye et al., 2009] Ye, D., Han, X., and Zhang, G. (2009). On-Line Multiple-Strip Packing. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications (COCOA)*, page 165. Springer.

[Yu and Rauchwerger, 2000] Yu, H. and Rauchwerger, L. (2000). Adaptive reduction parallelization techniques. In *Proceedings of the 14th international conference on Supercomputing*, pages 66–77. ACM New York, NY, USA.

[Zheng et al., 2008] Zheng, F., Xu, Y., and Zhang, E. (2008). How much can lookahead help in online single machine scheduling. *Information Processing Letters*, 106.

[Zhuk, 2006] Zhuk, S. (2006). Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications*, 16(1) :73–85.