Ordonnancement sur machines parallèles: minimiser la somme des coûts.

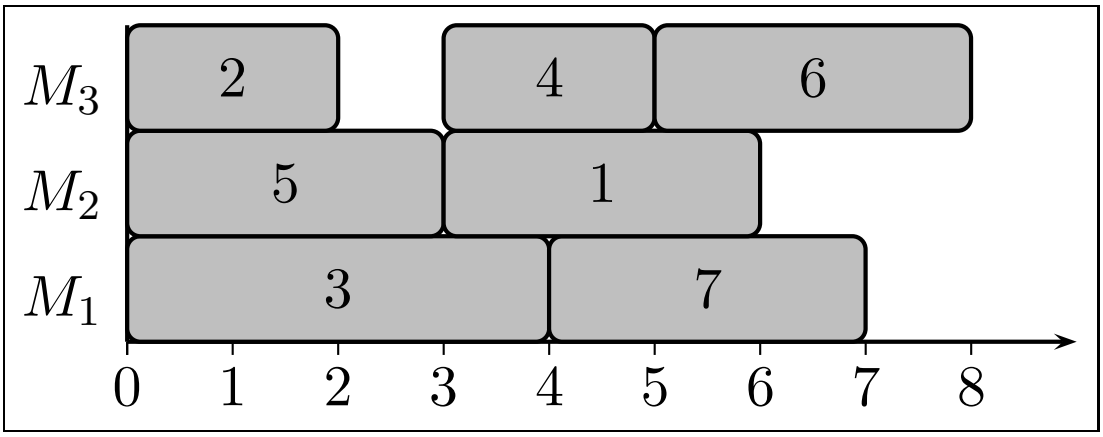
David Savourey

Université de Technologie Compiègne

Par David Savourey

Ordonnancement sur machines parallèles : minimiser la somme des coûts

Thèse présentée
pour l'obtention du grade
de Docteur de l'UTC.



Soutenue le : 5 décembre 2006
Spécialité : Technologie de l'information et des systèmes

Ordonnancement sur machines parallèles : minimiser la somme des coûts

Thèse soutenue le 5 décembre 2006 devant le jury composé de :

M.	Chengbin Chu	(Rapporteur)
M.	Mohamed Haouari	(Rapporteur)
M.	Jacques Carlier	(Président, Examinateur)
M.	Francis Sourd	(Examinateur)
M.	Antoine Jouglet	(Directeur de thèse)
M.	Philippe Baptiste	(Directeur de thèse)

Remerciements

Je souhaite en premier lieu remercier M. Antoine Jouglet, maître de conférences à l'Université de Technologie de Compiègne. Antoine m'a fait confiance dans cette aventure ; il m'a guidé, il m'a formé avec patience et attention. Si cette expérience a été si enrichissante, sur le plan scientifique comme sur le plan humain, c'est en grande partie grâce à lui.

J'adresse également mes remerciements à M. Philippe Baptiste, chargé de recherches au CNRS et professeur associé à l'École Polytechnique, qui a accepté de co-encadrer cette thèse. Il a su orienter mes travaux et « booster » ma motivation ; la recherche semble toujours simple et passionnante à ses côtés !

J'ai été très honoré que MM. Chengbin Chu et Mohamed Haouari, respectivement professeur à l'Université de Technologie de Troyes et professeur à l'École Polytechnique de Tunisie, rapportent mes travaux de thèse. Je tiens à les en remercier.

Merci aussi à M. Francis Sourd, chargé de recherches CNRS à Paris 6, d'avoir accepté d'être examinateur de cette thèse. Il m'a également donné de précieux conseils.

Je suis fier que M. Jacques Carlier, professeur à l'Université de Technologie de Compiègne,ait accepté de présider mon jury de soutenance. Je tiens à le remercier pour les années que j'ai passées à ses côtés au sein du laboratoire HeuDiaSyC.

L'équipe d'optimisation de Compiègne est également composée de M. Aziz Moukrim, professeur des Universités, et de MM. Jean-Paul Boufflet et Dritan Nace, maîtres de conférences. Je souhaite les remercier chaleureusement.

La sallé café du département d'informatique, haut lieu de convivialité à l'UTC, m'a permis de rencontrer bon nombre de collègues que je souhaite remercier ici pour ces moments de détente passés en leur compagnie.

Relire une thèse pour y débusquer les fautes diverses et variées, les phrases trop lourdes est un travail fastidieux. Heureusement, Chiffonnette et François ont accepté cette tâche. Qu'ils soient remerciés pour leur minutie. Je veux également remercier Julien, maître LaTeX, dont les nombreux conseils m'ont aidé à définir le style graphique de ce mémoire.

Je voudrais adresser deux remerciements particuliers. Je suis passé pendant cette thèse par une étape difficile, où je me sentais « perdu ». Discuter

avec Laurent Bordes et Christophe Ambroise, ou bien simplement les voir faire leur métier avec passion et sans amertume, m'a permis de redonner du sens à ce travail de thèse.

J'ai rencontré de nombreuses personnes sans qui ces années passées à Compiègne n'auraient pas été aussi agréables. Je pense en particulier à Julien, Pierre-Alexandre, Delphine et Antoine, Gaëlle et François, Laurence et Laurent, Anne et Sergio, Véronique et Fabrice, Geneviève et Jean-Paul, Marc, Célia et Yoann, Yacine, Linh, Cuc et Huy, Mylène, Yves, Astride, David, Céline, Joseph et Hermann.

Je voudrais également remercier mes parents pour m'avoir soutenu dans cette aventure et pour toute la confiance qu'ils m'ont apportée.

Enfin, je souhaite conclure en remerciant Marlyse, qui a fait bien plus que m'accompagner tout au long de ce challenge. Son optimisme déconcertant à mon égard m'a porté quand j'en ai eu besoin.

Table des matières

Table des figures

Liste des tableaux

Résumé des contributions

Cette thèse porte sur l'étude de quatre problèmes d'ordonnancement classiques sur machines parallèles, où les jobs à exécuter ont des dates de disponibilité, des durées, des poids et des dates échues différents. Les critères que nous cherchons à minimiser sont des critères totaux : la somme des dates de fin, la somme des dates de fin pondérée, le retard total ou le retard total pondéré. Ces quatre problèmes se notent respectivement $Pm|r_i|\sum C_i$, $Pm|r_i|\sum w_i C_i$, $Pm|r_i|\sum T_i$ et $Pm|r_i|\sum w_i T_i$.

La première contribution de cette thèse porte sur de nouvelles règles de dominance, qui se divisent en trois catégories. Nous présentons tout d'abord une classe dominante parmi les ordonnancements actifs. Cette règle s'appuie sur des dominances locales. La notion de localité sur machines parallèles est mise en place à l'aide des fronts d'adjacence. Après avoir défini cette classe, nous montrons qu'elle est bien dominante pour les quatre problèmes qui nous intéressent. Nous présentons ensuite trois nouvelles règles de dominance qui s'appliquent à des ordonnancements partiels. Ces trois règles portent soit sur une machine, soit sur m machines ou soit sur k parmi m machines. L'idée commune à ces trois règles est la suivante : comparer deux « configurations » composées des mêmes jobs et déterminer si l'une est nécessairement plus intéressante que l'autre. Par « nécessairement plus intéressante », nous entendons qu'il sera toujours préférable de choisir une configuration plutôt que l'autre, quel que soit l'ordonnancement des jobs non encore ordonnancés à la suite de ces configurations. Nous proposons enfin une règle de dominance qui porte sur les jobs non ordonnancés. Étant donnés un ordonnancement partiel et un ensemble de jobs non ordonnancés, cette règle établit des conditions suffisantes pour qu'un job non ordonnancé en domine un autre par rapport à un ordonnancement partiel. Pour cela, on montre qu'il est inutile d'ordonnancer le job dominé à la suite de l'ordonnancement partiel dans la mesure où l'échange entre les deux jobs serait alors améliorant, et ce quelle que soit la position du job dominant. Ces travaux ont été présentés lors d'une conférence internationale [62] et lors d'une conférence francophone [63].

La deuxième contribution de cette thèse concerne l'élaboration de bornes inférieures. Nous proposons plusieurs bornes inférieures qui peuvent être classées en quatre catégories. Tout d'abord, nous proposons une borne inférieure obtenue en relâchant les dates de disponibilité en deux groupes de valeurs égales. Nous montrons comment ce nouveau problème peut être résolu polynomialement pour le critère de la somme des dates de fin, afin d'obtenir une borne inférieure valide. Cette borne dépend d'un paramètre qui permet de définir les deux groupes de jobs réunis par leurs dates de disponibilité. Pour la deuxième catégorie de bornes inférieures, les jobs sont découpés en morceaux de taille unitaire qui ont un poids adapté à ce découpage. Le problème ainsi formé devient un problème de flot maximal à coût minimal, que l'on sait résoudre de manière optimale. Dans cette modélisation, la ligne de temps est divisée en plusieurs intervalles que l'on peut choisir. Nous proposons trois méthodes différentes pour définir ces intervalles de temps. La

troisième catégorie de bornes inférieure utilise la notion de date de fin minimale. La i^{e} date de fin minimale est par définition égale à la date de fin minimale du job qui termine en i^{e} position, parmi l'ensemble des ordonnancements réalisables. Nous montrons tout d'abord comment obtenir des bornes inférieures pour les quatre critères à partir d'un ensemble de dates de fin minimales. Nous proposons ensuite plusieurs méthodes permettant d'obtenir des dates de fin minimales. Nous proposons une méthode simple s'appuyant la règle de McNaughton [50]. Nous utilisons également un théoreme de Horn permettant de résoudre le problème $Pm|pmtn, r_i|C_{max}$ [41]. À partir de ce dernier, nous présentons un *PLNE* dont la solution optimale correspond à la i^{e} date de fin minimale dans le cas préemptif. Nous présentons également deux manières de relâcher ce *PLNE* pour obtenir de nouvelles dates de fin minimales plus rapidement. La première méthode consiste simplement à relâcher la contrainte d'intégrité. La seconde méthode est plus complexe, elle transforme les données du problème de sorte que sa résolution devienne facile. Comme ce n'est pas toujours la même méthode qui obtient la plus petite i^{e} date de fin minimale, nous proposons de tirer parti de ces méthodes « rapides » afin d'obtenir une borne inférieure efficace. La dernière catégorie de bornes inférieures s'appuie sur une formulation indexée sur le temps. Nous proposons, à partir de cette formulation, plusieurs relaxations qui aboutissent à différentes bornes inférieures. La première d'entre elles relâche la contrainte d'intégrité afin d'obtenir un *PL* qui doit être résolu exactement. Les autres relaxations sont des relaxations lagrangiennes, tantôt sur la contrainte de ressource, tantôt sur la contrainte d'occurrence. Une série de résultats numériques permet de comparer l'ensemble de ces bornes inférieures. Ces résultats sont présentés critère par critère. Cette contribution a été présentée lors d'une conférence internationale [61] et lors d'une conférence francophone [60]. Elle a également été acceptée dans une revue internationale [12].

La résolution exacte des quatre problèmes d'ordonnancement est la troisième contribution de cette thèse. Nous proposons une méthode arborescente de type *Branch&Bound* pour résoudre ces problèmes. Nous définissons tout d'abord une classe dominante d'ordonnancements, dits quasi-actifs. Nous montrons que ceux-ci peuvent être représentés par des séquences. Cela nous permet d'utiliser les séquences comme schéma de branchement dans le *Branch&Bound*. Nous montrons comment les règles de dominance que nous avons proposées peuvent être intégrées dans cette méthode exacte. Cette dernière utilise également les bornes inférieures que nous avons présentées. Un algorithme glouton est exécuté en début de méthode pour initialiser la valeur de la borne supérieure. Ce dernier intègre deux règles de dominance afin d'améliorer la solution retournée. Nous présentons plusieurs résultats numériques relatifs à cette méthode exacte. Une première série de résultats montre l'efficacité des règles de dominance au sein de la méthode exacte. Ces résultats nous permettent de déterminer les méthodes de filtrage efficaces et celles qui le sont moins. Elle nous permet de motiver les choix effectués dans l'algorithme utilisé pour la résolution exacte. Avec la configuration que nous avons jugée la plus intéressante, une série de résultats préliminaires de la

méthode exacte est également présentée. Ces résultats sont présentés critère par critère. Cette contribution a été soumise afin d'être présentée dans une conférence francophone.

Introduction

Sommaire

1.1 Recherche Opérationnelle, Ordonnancement

La recherche opérationnelle est une discipline relativement récente, elle apparaît principalement dans les années 1940, motivée par des applications militaires. Elle s'inscrit aujourd'hui pleinement dans le cadre de la compétitivité économique. Elle vise à la rationalisation de systèmes complexes, bien souvent par l'optimisation d'un certain critère. Une partie des problèmes traités par la recherche opérationnelle sont dits combinatoires. Face à un grand nombre de solutions possibles et pour un critère donné, on cherche la meilleure solution ou une bonne solution, suivant les besoins à satisfaire. L'ordonnancement fait partie de cette catégorie de problèmes que l'on regroupe sous le nom d'optimisation combinatoire. Il vise à organiser, à agencer « au mieux » des tâches (ou jobs) à exécuter au cours du temps sur un certain nombre de ressources [8]. Par exemple, la réalisation de microcontrôleur nécessite un grand nombre d'opérations. Ces dernières ne peuvent s'effectuer dans n'importe quel ordre. De plus, certaines tâches demandent plus de temps que d'autre. Chercher à réaliser un microcontrôleur en un temps minimum apparaît donc comme un problème d'ordonnancement. On trouve de nombreux domaines d'applications : la productique, l'informatique, le secteur tertiaire, ...

Les problèmes d'ordonnancement étant très variés, ils sont classés en plusieurs familles. On trouve de nombreux ouvrages présentant l'ordonnancement et ses différents problèmes [8, 17, 21, 28, 32, 39, 47]. Il existe éga-

lement un site web qui recense les résultats de complexité des problèmes d'ordonnancement [18]. Si l'on dispose d'une seule machine pour exécuter tous les jobs, on parle de problème à une machine. À l'inverse, on peut disposer de plusieurs machines. Ce sont les problèmes à machines parallèles. Les machines peuvent être de différentes natures : elles peuvent être interchangeables, on parle de machines parallèles identiques ; elle peuvent être correllées entre elles, un seul paramètre de vitesse les faisant différer ; elles peuvent être indépendantes les unes des autres. Cela signifie que l'exécution d'un job peut être plus longue que celle d'un autre job sur une machine, tandis que la situation est inversée sur une autre machine. Lorsqu'une tâche est composée de plusieurs opérations nécessitant des ressources différentes, on parle de problèmes d'atelier. On peut citer trois grands problèmes d'atelier : on dispose de n jobs qui se décomposent en m opérations qui doivent s'exécuter sur m machines distinctes. Lorsque l'ordre des opérations est fixé et est le même pour tous les jobs, on est face à un problème de *Flow-Shop*. Si l'ordre est fixé mais varie d'un job à l'autre, c'est un problème de *Job-Shop*. Si cet ordre peut être quelconque, on parle d'*Open-Shop*.

Par ailleurs, plusieurs types de contraintes portent sur la nature des jobs à exécuter. S'il est possible d'interrompre l'exécution d'un job et de la reprendre à une date ultérieure, on dit que le problème est préemptif. Des contraintes de précédence peuvent également régir l'ordre d'exécution des jobs. Les jobs peuvent disposer d'une date de disponibilité avant laquelle leur exécution ne peut commencer. On peut également citer les problèmes particuliers où les jobs sont de durées unitaires, ceux où ils sont de durées égales et le cas général où les durées sont différentes.

La définition des ressources et des jobs permet de fixer les contraintes. Ces dernières suffisent à définir les problèmes de faisabilité , *i.e.* existe-t-il une solution qui vérifie toutes les contraintes ? Pour un problème d'optimisation, il faut se donner une fonction objectif : le critère que l'on va chercher à optimiser. Le problème consiste alors à trouver la solution faisable qui optimise le critère. Le critère le plus classique en ordonnancemnent est sans doute la date de fin de projet ou *makespan*. On peut aussi chercher à minimiser le retard maximal parmi les jobs, ou encore l'avance-retard maximal. Il existe également de nombreux critères qui s'expriment comme une somme : la somme des dates de fin, la somme des retards, le nombre de jobs en retard. Ces critères existent aussi en version pondérée, on affecte alors un poids à chaque job.

Toutes ces possibilités permettent de définir un grand nombre de problèmes d'ordonnancement différents. La notation $(\alpha|\beta|\gamma)$, introduite par Graham *et al.* [36], est couramment utilisée pour identifier un problème d'ordonnancement. Le champ α renseigne les informations relatives aux ressources , *i.e.* le type et le nombre de machines disponibles. Le champ β indique les caractéristiques des jobs qui doivent être ordonnancés. Enfin, le champ γ correspond au critère que l'on cherche à optimiser. Par exemple, le problème $(1|pmtn|\sum U_i)$ est un problème à une machine, où les jobs ont des dates échues, où la préemption des jobs est autorisée, et où l'on cherche à minimiser le nombre de jobs en retard.

1.2 Survol des méthodes d'optimisation combinatoire

Nous présentons ici, de manière succinte, les grands principes des méthodes d'optimisation qui sont utilisées dans cette thèse. Un ouvrage présente en détail les grandes méthodes d'optimisation combinatoire [22].

1.2.1 Programmation Linéaire

La Programmation Linéaire (*PL*) [29, 37, 65] regroupe l'ensemble des problèmes d'optimisation où la fonction objectif et les contraintes sont linéaires :

$$\min_x f(x) \quad \begin{cases} g(x) \leq b \\ h(x) = c \\ x \in \mathbb{R}^n, \end{cases}$$

avec f, g , h linéaires.

L'ensemble des solutions admissibles, aussi nommé polytope, se définit comme suit :

$$\mathcal{P} = \{x \in \mathbb{R}^n, g(x) \leq b \text{ et } h(x) = c\}.$$

Le polytope $\mathcal{P}$ peut être de trois natures différentes :

Il peut être vide si les contraintes sont incompatibles, par exemple $x \leq 3$ et $x > 5$ en dimension 1. Il n'y a alors pas de solution.

Il peut être non-borné. Par exemple, si l'on considère les contraintes $x_1 - x_2 \geq 2$ et $x_1 + x_2 \leq 1$ en dimension 2, on obtient le polytope représenté sur la Figure 1.1. Il y a alors deux cas de figure. Soit il est borné dans le sens de la fonction objectif et il existe une solution optimale, soit il n'est pas borné dans le sens de la fonction objectif et il n'y a alors pas de solution optimale, puisque l'on peut faire diminuer (pour un problème de minimisation) arbitrairement la valeur de $f(x)$.

Enfin, le polytope des solutions admissibles peut être borné et non vide. Un exemple en dimension 2 est présenté sur la Figure 1.2, avec les contraintes $2x_1 + x_2 \leq 2$, $x_1 - x_2 \geq -1$ et $x_2 \geq -2$. L'ensemble des solutions optimales peut alors être réduit à un singleton (sommet du polytope), ou bien être infini (arête du polytope).

On sait que l'ensemble des sommets du polytope forme un sous-ensemble dominant parmi les solutions admissibles. En d'autres termes, il existe au moins une solution optimale qui est un sommet du polytope.

L'algorithme du simplexe [30], proposé par Dantzig en 1947, permet de résoudre le problème de la *PL*. Si ce dernier est assez efficace la plupart du temps, il peut néanmoins demander un temps exponentiel en fonction de la donnée pour converger. L'idée du simplexe consiste à se déplacer d'un sommet du polytope à l'autre en suivant les arêtes, jusqu'à atteindre la meilleure solution.

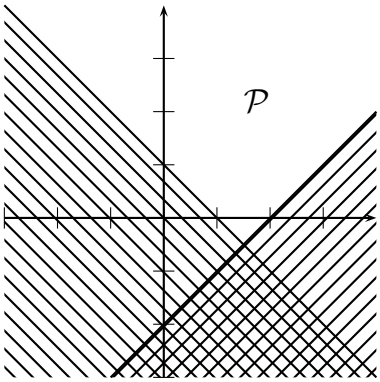


Figure 1.1 – Exemple de polytope non-borné.

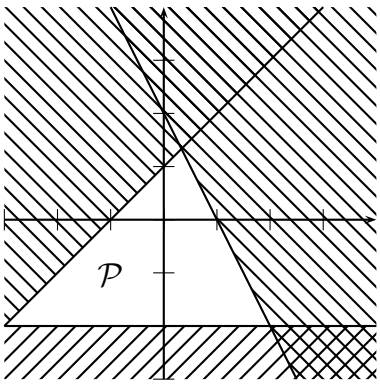


Figure 1.2 – Exemple de polytope borné et non vide.

Khachiyan, en 1979, a proposé la méthode de l'ellipsoïde [46], inefficace en pratique, mais qui a permis de démontrer que le problème de la *PL* était polynomial.

Il est également possible d'utiliser la méthode projective, introduite par Karmarkar [44], qui est un algorithme à la fois polynomial et efficace en pratique.

1.2.2 Programmation Linéaire en Nombres Entiers

Lorsque les variables doivent prendre des valeurs entières, on entre dans le champ de la Programmation Linéaire en Nombres Entiers (*PLNE*) (voir par exemple [1, 33, 51]). Si la *PL* est polynomiale, la *PLNE* est quant à elle difficile à résoudre. Il existe principalement deux familles de méthodes pour résoudre un *PLNE* : les méthodes arborescentes et les méthodes de coupe. Parmi les méthodes arborescentes, on peut citer la méthode de *Branch&Bound* (présentée dans la Section 1.2.4), la méthode *Branch&Cut* ou

encore la méthode de *Branch&Price&Cut*. Les deux dernières méthodes ne sont pas abordées dans cette introduction. Nous présentons maintenant quelques généralités sur les méthodes de coupe.

Notons (I) le *PLNE* que l'on cherche à résoudre, et (C) le *PL* obtenu à partir de (I) en relâchant la contrainte d'intégrité, *i.e.* $x \in \mathbb{N}^n$ devient $x \in \mathbb{R}^n$. Les méthodes de coupe fonctionnent de manière itérative. Si $\mathcal{P}$ représente le polytope des solutions admissibles de (C), on note $\hat{\mathcal{P}}$ l'ensemble des solutions entières incluses dans $\mathcal{P}$. L'ensemble $\hat{\mathcal{P}}$ représente l'ensemble des solutions admissibles du problème (I). Lorsqu'on applique une méthode de coupe, on résout successivement des *PL*, jusqu'à ce que la solution optimale soit entière. On résout tout d'abord (C). Ensuite, on va ajouter des contraintes à (C) pour « rogner » le polytope $\mathcal{P}$ dans la zone de la solution optimale. Un exemple de coupe est présenté sur la Figure 1.3.

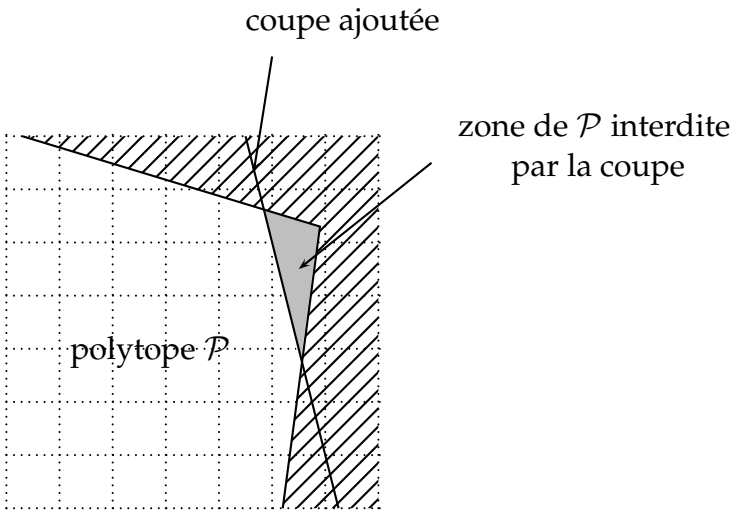


Figure 1.3 – Exemple de coupe.

Les contraintes ajoutées ont la propriété de ne jamais exclure des élements de $\hat{\mathcal{P}}$. Ainsi, dès que (C) et ses contraintes supplémentaires ont une solution optimale entière, cette solution est aussi la solution optimale du *PLNE* (I). La difficulté principale de cette méthode réside dans le choix des coupes. Cela peut en effet grandement influencer le temps de convergence de l'algorithme. On peut par exemple citer la méthode de coupes de Gomory [34].

1.2.3 Relaxation lagrangienne

La relaxation lagrangienne est une technique fréquemment utilisée en recherche opérationnelle [35]. Elle permet bien souvent de calculer des évaluations par défaut (ou bornes inférieures) de la solution optimale. L'idée consiste à relâcher une partie des contraintes (souvent celles qui rendent le problème difficile) et à les injecter dans la fonction objectif sous forme d'une combinaison linéaire de pénalités. Les coefficients de cette combinaison sont appelés multiplicateurs de Lagrange.

Soit le problème d'optimisation suivant :

$$\begin{min_x f(x) \\ \begin{cases} g_1(x) \leq b_1 \\ g_2(x) \leq b_2 \\ x \in D(x). \end{cases}$$

On suppose ici que les contraintes qui rendent le problème difficile sont $g_1(x) \leq b_1$. On appelle Lagrangien du problème la fonction :

$$L(x, \lambda) = f(x) + \langle \lambda, g_1(x) - b_1 \rangle .$$

On se propose alors de résoudre le problème suivant :

$$\begin{min_x L(x, \lambda) \\ \{ \ g_2(x) \leq b_2.$$

On note $h(\lambda) = \min_{x / g_2(x) \leq b_2} L(x, \lambda)$. On sait alors que

$$\forall \lambda > 0, \forall x \text{ tel que } g_2(x) \leq b_2, \ \ h(\lambda) \leq f(x).$$

Lorsqu'on effectue une relaxation lagrangienne dans le but d'obtenir une borne inférieure de la valeur optimale, on va bien souvent chercher le vecteur de multiplicateurs λ qui maximise h, sachant que cette valeur sera toujours inférieure à l'optimum du problème initial. On peut utiliser pour cela une méthode de sous-gradient [35]. Cette dernière permet de mettre à jour les multiplicateurs de Lagrange. En effet, la direction indiquée par un sous-gradient est nécessairement une direction de descente si l'on ne se trouve pas sur un optimum local. Par contre, rien ne nous indique *a priori* la taille du pas à effectuer pour descendre. Plus précisément, voici comment les multiplicateurs de Lagrange sont mis à jour dans une méthode de gradient :

$$\lambda^{(k+1)} = \lambda^{(k)} + \alpha^{(k)} \gamma(\lambda^{(k)}),$$

où $\lambda^{(i)}$ indique le vecteur de multiplicateurs à la i^{e} itération. Le vecteur $\gamma(\lambda^{(k)})$ est un sous-gradient obtenu au point $\lambda^{(k)}$. Enfin, le scalaire $\alpha^{(k)}$ permet de règler la taille du pas que l'on va effectuer. Ce paramètre peut être fixé par de nombreuses méthodes différentes, que nous ne présenterons pas ici. Néanmoins, on peut noter qu'on fait en général tendre $\alpha^{(k)}$ vers 0 quand k augmente, afin de garantir la convergence de l'algorithme. La façon de fixer la valeur de $\alpha(k)$ est déterminante pour l'efficacité de la méthode.

1.2.4 Méthode de *Branch&Bound*

La méthode de *Branch&Bound* appartient à la classe des méthodes arborescentes généralement utilisées pour résoudre un *PLNE* [51]. On l'utilise lorsque l'espace des solutions réalisables est très grand. L'idée générale de cette méthode consiste à trouver un moyen d'énumérer l'ensemble des solutions x, en évaluant la quantité $f(x)$ afin de retenir la meilleure. Notons $S(P)$ l'ensemble des solutions réalisables associées au problème (P).

Le problème initial (P) est séparé en plusieurs sous-problèmes (P_i), de sorte que la résolution de ces sous-problèmes entraîne la résolution de (P). Il faut pour cela que l'union des solutions visitées dans chacun des sous-problèmes soit égale à l'ensemble des solutions réalisables du problème (P), afin de garantir que l'énumération a bien été exhaustive. Autrement dit, la condition $\bigcup_i S(P_i) = S(P)$ doit être vérifiée.

Bien évidemment, un sous-problème peut à son tour être décomposé en une famille de sous-problèmes. D'une manière générale, un problème sera décomposé dès lors qu'il n'est pas facile à résoudre. Cette cascade de décompositions forme ce que l'on nomme l' « arbre de recherche », dont un exemple est représenté sur la Figure 1.4.

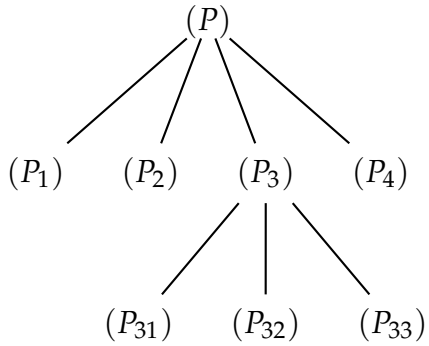


Figure 1.4 – Exemple d'arbre de recherche.

Dans l'arbre de recherche, un sommet ϕ représente un sous-problème (P_ϕ). Évaluer un sommet ϕ consiste à déterminer la valeur optimale $F^*(P_\phi)$ associée à ce sous-problème. Si le problème est facile, on sait calculer directement la valeur de cet optimum. Sinon, la valeur optimale est obtenue en séparant le problème et en retenant la meilleure valeur parmi les *optima* des sous-problèmes. Si le problème (P_ϕ) se décompose en k sous-problèmes $(P_\phi^1), (P_\phi^2), \ldots, (P_\phi^k)$, $F^*(\phi)$ correspond à la meilleure valeur parmi $F^*(P_\phi^1), F^*(P_\phi^2), \ldots, F^*(P_\phi^k)$.

Plusieurs stratégies d'exploration de l'arbre de recherche sont possibles. Si l'on choisit une exploration en profondeur d'abord, on évaluera en priorité les sommets ouverts situés « en bas » de l'arbre de recherche. À l'inverse, il peut être intéressant d'évaluer en premier les sommets les plus hauts, on adopte alors une stratégie en largeur d'abord. Il est également possible d'évaluer en priorité les sommets ouverts les plus prometteurs.

Il est également important de choisir une heuristique qui décide l'ordre dans lequel les sous-problèmes sont évalués. En effet, on préferera visiter le plus tôt possible les bonnes solutions, afin d'avoir une bonne base de comparaison pour éviter l'évaluation de sommets « inintéressants ».

Il est classique d'intégrer dans un *Branch&Bound* des bornes supérieures et des bornes inférieures. Une borne supérieure est une évaluation par excès de l'optimum, tandis qu'une borne inférieure est une évaluation par défaut. L'utilisation de bornes de bonne qualité est cruciale pour l'efficacité de la

méthode. En général, on les utilise de la manière suivante. Avant d'évaluer un sommet S associé à un sous-problème (SP), on calcule une borne inférieure $LB(SP)$ de l'optimum du problème (SP). Si la valeur de cette borne inférieure LB est supérieure ou égale à la valeur d'une borne supérieure du problème initial $UB(P)$, *i.e.* $LB(SP) \geq UB(P)$, l'évaluation du sommet S est inutile, elle n'a donc pas lieu. Dès lors qu'un sommet a été évalué, on met à jour la valeur $UB(P)$ si l'optimum de ce sommet lui est inférieur.

1.2.5 Algorithmes gloutons

Un algorithme glouton [29] est une méthode où l'on construit incrémentalement une solution à un problème d'optimisation. À chaque étape, on prend la décision qui est localement optimale. Si la solution ainsi construite est optimale, l'algorithme glouton est une méthode exacte. Dans le cas contraire, il tient lieu d'heuristique.

1.3 Présentation des problèmes étudiés

Nous nous intéressons dans cette thèse à quatre problèmes sur machines parallèles. Il s'agit d'ordonnancer n jobs, définis par des dates de disponibilité r_i et des durées p_i. Suivant le critère que l'on cherche à minimiser, on affecte également aux jobs des dates échues d_i et/ou des poids w_i. La préemption n'est pas autorisée sur ces jobs. Les quatre problèmes diffèrent par le critère que l'on souhaite minimiser, qui sont :

- la somme des dates de fin $\sum C_i$;
- la somme pondérée des dates de fin $\sum w_i C_i$;
- le retard total $\sum T_i$;
- le retard total pondéré $\sum w_i T_i$,

où C_i et $T_i = \max(0, C_i - d_i)$ représentent respectivement la date de fin et le retard du job i. Ces quatre problèmes se notent $Pm|r_i|\sum C_i$, $Pm|r_i|\sum w_i C_i$, $Pm|r_i|\sum T_i$ et $Pm|r_i|\sum w_i T_i$. Le retard total pondéré est le critère le plus général. On peut hiérarchiser ces quatre critères comme l'indique la Figure 1.5.

Ce sont quatre problèmes NP-Difficiles au sens fort [33]. La résolution exacte de ces problèmes n'a pas, à notre connaissance, été beaucoup étudiée jusqu'à maintenant, bien qu'ils constituent une modélisation idéalisée de problèmes réels. On peut toutefois citer les travaux de Yalaoui et Chu qui proposent une méthode exacte pour le problème $Pm|r_i|\sum C_i$ [68], ainsi que ceux de Nessah *et al.* pour les problèmes $Pm|r_i|\sum C_i$ et $Pm|r_i, sds|\sum C_i$ [53].

Le Chapitre 2 présente une série de résultats théoriques de dominance. Une règle de dominance est une contrainte qui peut être ajoutée au problème étudié sans changer la valeur de l'optimum. Autrement dit, il existe au moins une solution optimale qui vérifie cette contrainte. Ces résultats sont d'un grand interêt puisqu'ils vont permettre de réduire l'espace de recherche lors d'une résolution exacte ou approchée. Nous rappelons tout d'abord la caractérisation des ordonnancements actifs dans la Section 2.2.1,

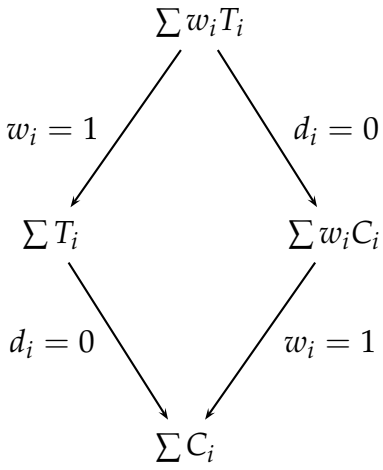


Figure 1.5 – Hiérarchie des critères.

qui forment une classe dominante pour les critères étudiés. Nous proposons ensuite quelques raffinements visant à établir un sous-ensemble dominant au sein des ordonnancements actifs : les ordonnancements LOWS-actifs (Section 2.2.3). Dans la Section 2.3, nous proposons trois règles de dominance qui portent sur des ordonnancements partiels. D'une manière générale, un ordonnancement partiel P_1 est dominé s'il existe un autre ordonnancement partiel P_2 tel que tout ordonnancement débutant par P_1 peut être amélioré en remplaçant P_1 par P_2. La première d'entre elles (RD_1) permet de comparer deux ordonnancements partiels sur une machine composés des mêmes jobs (Section 2.3.1). L'idée consiste à chercher sous quelles conditions il est nécessairement avantageux de remplacer l'un par l'autre. La deuxième règle de dominance (RD_m) permet de comparer deux ordonnancements partiels sur m machines (Section 2.3.2). Une fois de plus, il faut qu'ils soient composés des mêmes jobs. Nous établissons des conditions permettant d'affirmer qu'un ordonnancement partiel est dominé. Enfin, la troisième règle de dominance RD_k, présentée dans la Section 2.3.3, généralise les deux précédentes. Elles permet une fois encore de comparer deux ordonnancements partiels sur m machines, mais qui ne sont pas nécessairement composés des mêmes jobs. Il faut néanmoins qu'il existe un sous-ensemble de machines pour chacun des ordonnancements partiels, composés des mêmes jobs, de sorte qu'un sous-ensemble soit dominé par l'autre. Lorsque les sous-ensembles se réduisent à une machine, on retombe sur la règle RD_1. S'ils portent sur l'ensemble des machines, on retombe sur RD_m. Nous présentons ensuite une règle de dominance qui porte sur des jobs non ordonnancés. Si l'on dispose d'un ordonnancement partiel P et d'un ensemble de jobs non ordonnancés NS, on établit des conditions suffisantes pour qu'un job j de NS domine un job k de NS par rapport à P. Le job j domine le job k par rapport à P si tout ordonnancement O commençant par P et où k est ordonnancé au plus tôt à la suite de P est dominé par un ordonnancement O' obtenu à partir de O en échangeant les positions des jobs j et k. Enfin, une discussion est menée

dans la Section 2.5 afin de comparer les règles de dominance qu'ils ont proposées pour la somme des dates de fin avec celles que nous avons présentées.

L'étude de bornes inférieures constitue l'objet du Chapitre 3. L'utilisation de bornes inférieures de bonne qualité est cruciale dans une méthode arborescente de type *Branch&Bound*. Elles permettent un élagage efficace dans l'arbre de recherche. Nous présentons tout d'abord deux bornes inférieures, pour la somme des dates de fin, obtenues en relâchant les dates de disponibilité (Section 3.2) de manières différentes. Si l'ensemble des dates de disponibilité est relâché à 0, on sait résoudre exactement le problème (Section 3.2.1). Sinon, on peut chercher à diminuer les dates de disponibilité pour qu'elles forment des sous-ensembles de valeurs égales (Section 3.2.2). Nous montrons qu'il suffit alors de résoudre séparement plusieurs sous-problèmes faciles pour obtenir une borne inférieure. Nous étudions dans la Section 3.3 des bornes inférieures obtenues en modélisant une relaxation du problème sous la forme d'un problème de flot maximal à coût minimal. La Section 3.4 porte sur l'élaboration de bornes inférieures s'appuyant sur la notion de date de fin minimale d'un job exécuté dans une position donnée. Nous entendons par i^e date de fin minimale une valeur qui sera nécessairement inférieure à la date du job qui termine en i^e position, et ce quelque soit l'ordonnancement considéré. Nous utilisons en particulier un résultat de Horn qui nous permet de proposer plusieurs bornes. Nous présentons dans la Section 3.5 plusieurs bornes inférieures dérivées d'une formulation indexée sur le temps. Ces bornes sont obtenues soit en relâchant la contrainte d'intégrité, soit en relâchant la contrainte de ressource ou soit en relâchant la contrainte d'occurrence des jobs. Plusieurs de ces bornes sont des relaxations lagrangiennes. Ce chapitre se termine par une série de résultats numériques permettant de comparer toutes ces bornes inférieures (Section 3.6).

Le Chapitre 4 présente la méthode exacte que nous proposons pour résoudre ces quatre problèmes. Nous présentons tout d'abord la modélisation adoptée (Section 4.2). Une description détaillée de l'utilisation des différentes règles de dominance est fournie dans la Section 4.3. Nous présentons dans la Section 4.4 l'heuristique utilisée dans la méthode exacte pour initialiser la valeur de la borne supérieure. L'utilisation des bornes inférieures fait l'objet de la Section 4.5. Nous présentons dans la Section 4.6 le calcul d'une borne supérieure du *makespan* de tout ordonnancement. Cette valeur est utile pour certaines règles de dominance et certaines bornes inférieures. L'algorithme de *Branch&Bound* fait l'objet de la Section 4.7. Enfin, plusieurs études numériques montrent des résultats préliminaires de cette méthode exacte, ainsi que l'efficacité des règles de dominance au sein de cette méthode (Section 4.8).

Enfin, le Chapitre 5 conclut cette étude et propose des perspectives de recherche.

Règles de dominance

Sommaire

Nous présentons dans ce chapitre différentes règles de dominance. La première catégorie de règles de dominance consiste à rejeter les ordonnancements réalisables mais trivialement améliorables (jobs non calés à gauche, temps d'inactivité pouvant être utilisés). Nous raffinons ensuite ce sous-ensemble, en généralisant les travaux de Jouglet *et al.* [42, 43] sur une machine. Nous abordons alors une autre famille de règles de dominance, où l'on cherche à savoir s'il existe un ordonnancement partiel composé des mêmes jobs qu'un ordonnancement partiel donné mais de coût inférieur et/ou qui termine plus tôt. Nous présentons également une règle de dominance portant sur les jobs non encore ordonnancés. Enfin, nous comparons, lorsque cela est possible, les règles de dominance proposées par Yalaoui et Chu pour $\sum Ci$ avec les règles présentées dans ce chapitre.

2.1 État de l'art

Plusieurs auteurs ont proposé des règles de dominance pour les problèmes que nous étudions ou pour des problèmes proches de ces derniers. En particulier, le cas à une machine a été abondamment traité dans la littérature. Par exemple, Chu a proposé des règles de dominance pour les problèmes $1|r_i|\sum Ci$ [24] et $1|r_i|\sum T_i$ [25]. Ces travaux ont été améliorés pour les problèmes $1|r_i|\sum C_i$ et $1|r_i|\sum T_i$, et généralisés pour les problèmes $1|r_i|\sum w_iC_i$ et $1|r_i|\sum w_iT_i$ par Jouglet [43]. Akturk et Ozdemir [2] ont proposé une règle de dominance pour le problème $1||\sum w_iT_i$. Enfin, Bianco et Ricciardelli [15], Belouadah *et al.* [13] et Jouglet [42] ont présenté des règles de dominance pour le problème $1|r_i|\sum w_iC_i$. Lorsque $m > 1$, la littérature est moins abondante. Néanmoins, on peut citer Yalaoui et Chu qui ont proposé plusieurs règles de dominance pour le problème $Pm|r_i|\sum C_i$ [68]. Celles-ci sont présentées plus en détail en fin de chapitre. Concernant le problème $Rm||\sum w_iTi$, Liaw *et al.* ont également caractérisé à travers quatre propriétés des ordonnancements optimaux [49].

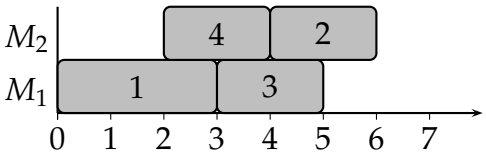
2.2 Dominances locales

2.2.1 Ordonnancements actifs

Rappelons tout d'abord que nous disposons d'un ensemble $N = \{1, 2, \ldots, n\}$ de jobs à ordonnancer sur m machines.

Définition 2.1. *[17] Un ordonnancement est semi-actif si aucune tâche ne peut être exécutée plus tôt sans changer l'ordre d'exécution sur une des machines ni violer de contraintes.*

Un exemple d'ordonnancement semi-actif est représenté sur la Figure 2.1.



job i	1	2	3	4
r_i	0	0	1	2
p_i	3	2	2	2

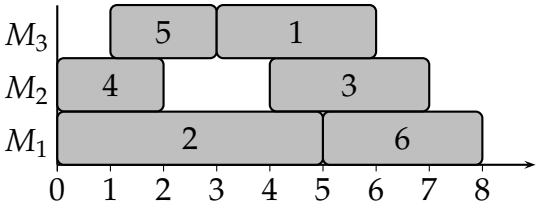
Figure 2.1 – Exemple d'ordonnancement semi-actif.

Nous avons défini la notion d'ordonnancement « quasi-actif », qui nous sera utile dans le Chapitre 4.

Définition 2.2. *Un ordonnancement est quasi-actif s'il est semi-actif et si l'exécution d'un job ne peut pas être avancée en ordonnançant ce job à la fin d'une autre machine.*

Définition 2.3. *[17] Un ordonnancement actif est un ordonnancement où l'on ne peut pas avancer l'exécution d'un job sans retarder celle d'un autre job.*

Un exemple d'ordonnancement actif est proposé sur la Figure 2.2. En revanche, l'ordonnancement semi-actif représenté sur la Figure 2.1 n'est pas actif.



job i	1	2	3	4	5	6
r_i	0	0	4	0	1	2
p_i	3	5	3	2	2	3

Figure 2.2 – Exemple d'ordonnancement actif .

Nous proposons ici quelques raffinements qui permettent de contruire un sous-ensemble dominant parmi les ordonnancements actifs. Ces travaux s'appuient sur et généralisent les travaux de Chu et Portmann [27], puis ceux de Jouglet [42, 43].

Parmi les ordonnancements semi-actifs, les ordonnancements actifs sont ceux où aucune période d'inactivité ne peut être comblée (partiellement ou non) par un job ordonnancé après le début de la période d'inactivité.

On rappelle qu'un critère est dit régulier [8, 28] lorsque le coût d'une solution est une fonction croissante des dates de fin des jobs.

Proposition 2.1. *L'ensemble des ordonnancements actifs est dominant pour les problèmes $Pm|r_i|\sum F_i$, où F est un critère régulier.*

Démonstration. Supposons qu'il n'existe aucun ordonnancement optimal qui soit actif. Considérons un ordonnancement optimal O^*. Il est facile de constater que l'on peut construire un ordonnancement semi-actif O' à partir de O^*, en calant à gauche les jobs qui ne le sont pas. Cette opération ne peut pas faire croître le coût, donc $F(O') \leq F(O^*)$. De la même manière, on peut tenter de combler les périodes d'inactivité en y insérant un job exécuté plus tard dans O' pour former un ordonnancement O''. Cette opération ne peut pas dégrader le coût, donc $F(O'') \leq F(O')$. Ces deux opérations – décalage à gauche et insertion à gauche – peuvent être répétées tant qu'elles modifient l'ordonnancement. Une fois cette étape terminée, l'ordonnancement obtenu O_2 est actif. Comme le coût des jobs n'a pas pu augmenter pendant le passage de O^* à O_2, on a $F(O_2) \leq F(O^*)$. L'ordonnancement O_2 est donc optimal, ce qui contredit l'hypothèse de départ. □

2.2.2 Ordonnancements LO-actifs

Pour présenter les ordonnancements LO-actifs (LO pour *Locally Optimal*), nous devons introduire la notion d'adjacence entre deux jobs. Cette notion est intuitive lorsque $m = 1$ (une machine), mais elle mérite d'être détaillée lorsque $m > 1$. Sur une machine, un job i est adjacent à un job j dans un ordonnancement O si j précède ou succède à i dans O. Pour le cas à machines parallèles, nous allons définir ce que nous nommons « front d'adjacence » (figure 2.3). Soient $m(i)$ la machine sur laquelle s'exécute le job i, et Δ_i la date de disponibilité de la machine $m(i)$ avant l'exécution du job i.

Définition 2.4. *Soient O un ordonnancement et i un job de O. Soit l'ensembe de jobs $\psi(i) = \{j/\ \Delta_j < \Delta_i$ ou $(\Delta_j = \Delta_i$ et $m(j) < m(i))\}$. Le front d'adjacence de i, noté $Adj(i)$ est composé des derniers jobs ordonnancés de $\psi(i)$ sur chacune des machines. Plus formellement, on a $Adj(i) = \{j \in \psi(i)/\forall k \in \psi(i)\,(m(j) = m(k)) \Rightarrow (start_j > start_k)\}$.*

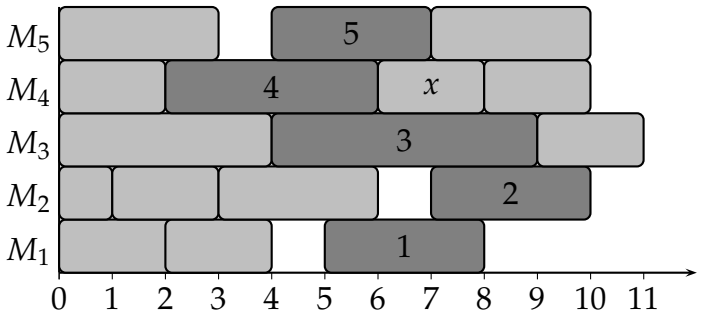


Figure 2.3 – Les jobs $\{1, 2, 3, 4, 5\}$ forment le front d'adjacence du job x.

La notion d'adjacence étant désormais introduite, nous pouvons définir la notion d'optimalité locale. Nous dirons qu'un ordonnancement O est LO-actif si aucun échange entre un job i et un job $j \in Adj(i)$ ne permet de diminuer le coût sans retarder l'exécution des jobs n'appartenant pas à $\psi(i)$.

Notons $C(i,t)$ la date de fin du job i lorsqu'il est ordonnancé au plus à partir de la date t, *i.e.* $C(i,t) = \max(r_i, t) + p_i$ et $F(i,t)$ le coût de ce job dans ce même cas. On a $F(i,t) = w_i * \max\big(0, C(i,t) - d_i\big)$.

Définition 2.5. *Soit O un ordonnancement actif. Cet ordonnancement est LO-actif si :*

1. *Pour tout job i dans O et pour tout job j de $Adj(i)$ tel que $m(j) \neq m(i)$, une des conditions suivantes est vérifiée :*
   - *(a)* $F(i,\Delta_i) + F(j,\Delta_j) \leq F(i,\Delta_j) + F(j,\Delta_i)$;
   - *(b)* $\min\big(C(i,\Delta_i), C(j,\Delta_j)\big) < \min\big(C(i,\Delta_j), C(j,\Delta_i)\big)$;
   - *(c)* $\max\big(C(i,\Delta_i), C(j,\Delta_j)\big) < \max\big(C(i,\Delta_j), C(j,\Delta_i)\big)$.

2. *Pour tout job i dans O et pour le job j de $Adj(i)$ tel que $m(j) = m(i)$, une des conditions suivantes est vérifiée :*

   (a) $F(i, \Delta_i) + F(j, \Delta_j) \leq F(i, \Delta_j) + F(j, C(i, \Delta_j))$;

   (b) $C\big(i, C(j, \Delta_j)\big) < C\big(j, C(i, \Delta_j)\big)$.

Étudions tout d'abord le cas où les jobs i et j ne sont pas ordonnancés sur la même machine. Lorsque la condition $(1a)$ est vérifiée, on remarque que le fait d'échanger les positions des jobs i et j augmente le coût de ces derniers. Lorsque la condition $(1b)$ (resp. $(1c)$) est vérifiée, l'échange des jobs i et j retarde la machine qui finit au plus tôt (resp. au plus tard). Si les jobs i et j sont ordonnancés sur la même machine, la condition $(2a)$ assure que l'échange des deux jobs augmente la somme de leurs coûts. Enfin, si la condition $(2b)$ est vérifiée, on sait que l'échange de i et j retarde la machine sur laquelle ils sont ordonnancés.

Proposition 2.2. *Tout ordonnancement actif optimal pour le retard total pondéré est LO-actif.*

Démonstration. Supposons qu'il existe un ordonnancement actif optimal qui ne vérifie aucune des propriétés pour un couple de jobs i et j. Il est clair dans ce cas qu'on peut construire un autre ordonnancement actif en échangeant les jobs i et j. On obtient alors un nouvel ordonnancement qui est strictement améliorant, puisque le coût de deux jobs a diminué strictement tandis le coût des autres jobs n'a pas augmenté. L'ordonnancement initial n'est donc pas optimal. □

2.2.3 Ordonnancements LOWS-actifs

Il est encore possible de construire un sous-ensemble dominant au sein des ordonnancements LO-actifs. En effet, nous pouvons exhiber une catégorie d'ordonnancements LO-actifs dominés. En fait, le cas de figure que nous cherchons à éliminer est celui où l'échange entre un job i et un job j ne fait pas varier le coût mais retarde une des machines. Lorsqu'aucun couple de job n'est dans cette configuration, on dira que l'ordonnancement est LOWS-actif (LOWS pour *Locally Optimal Well Sorted*) :

Définition 2.6. *Soit O un ordonnancement LO-actif. On dira que O est LOWS-actif lorsqu'au moins une des conditions suivantes est vérifiée :*

1. *Pour tout job i de O et pour tout job j de $Adj(i)$ tel que $m(j) \neq m(i)$, une des conditions suivantes est vérifiée :*

   (a) $F(i, \Delta_i) + F(j, \Delta_j) < F(i, \Delta_j) + F(j, \Delta_i)$;

   (b) $\min\big(C(i, \Delta_i), C(j, \Delta_j)\big) < \min\big(C(i, \Delta_j), C(j, \Delta_i)\big)$;

(c) $\max\left(C(i,\Delta_i), C(j,\Delta_j)\right) < \max\left(C(i,\Delta_j), C(j,\Delta_i)\right)$;

(d) $F(i,\Delta_i) + F(j,\Delta_j) = F(i,\Delta_j) + F(j,\Delta_i)$ *et*
$\min\left(C(i,\Delta_i), C(j,\Delta_j)\right) = \min\left(C(i,\Delta_j), C(j,\Delta_i)\right)$ *et*
$\max\left(C(i,\Delta_i), C(j,\Delta_j)\right) = \max\left(C(i,\Delta_j), C(j,\Delta_i)\right)$.

2. *Pour tout job i de O et pour le job j de $Adj(i)$ tel que $m(j) = m(i)$, une des conditions suivantes est vérifiée :*

(a) $F(i,\Delta_i) + F(j,\Delta_j) < F(i,\Delta_j) + F(j,C(i,\Delta_j))$;

(b) $C\big(i,C(j,\Delta_j)\big) < C\big(j,C(i,\Delta_j)\big)$;

(c) $F(i,\Delta_i) + F(j,\Delta_j) = F(i,\Delta_j) + F(j,C(i,\Delta_j))$ *et*
$C\big(i,C(j,\Delta_j)\big) = C\big(j,C(i,\Delta_j)\big)$.

Pour qu'un ordonnancement soit LOWS-actif, il faut que l'échange des jobs i et j dégrade strictement un des trois critères (ou deux si les jobs sont ordonnancés sur la même machine), ou bien qu'il n'ait aucun effet. En effet, quand un ordonnancement est LOWS-actif, la permutation des jobs i et j augmente strictement le coût, ou la (les) date(s) de fin de la (des) machine(s), ou bien ne change rien.

Proposition 2.3. *La classe des ordonnancements LOWS-actifs est dominante pour $Pm|r_i|\sum F_i$, où F_i est un critère régulier.*

Démonstration. Supposons qu'aucun ordonnancement optimal ne soit LOWS-actif. Soit O un ordonnancement optimal. La Proposition 2.2 nous permet d'affirmer que O est LO-actif. Comme O n'est pas LOWS-actif, nous savons donc qu'il existe deux jobs i et j qui ne vérifient pas les propriétés d'un ordonnancement LOWS-actif.

Si $m(i) \neq m(j)$, alors on sait que :
- $F(i,\Delta_i) + F(j,\Delta_j) \geq F(i,\Delta_j) + F(j,C(i,\Delta_j))$;
- $\min\left(C(i,\Delta_i), C(j,\Delta_j)\right) \geq \min\left(C(i,\Delta_j), C(j,\Delta_i)\right)$;
- $\max\left(C(i,\Delta_i), C(j,\Delta_j)\right) \geq \max\left(C(i,\Delta_j), C(j,\Delta_i)\right)$.

De plus, l'une des trois inégalités doit être vérifiée au sens strict, sans quoi il serait LOWS-actif. En permutant les jobs i et j, on peut constater qu'on obtient un ordonnancement qui vérifie nécessairement les conditions d'un ordonnancement LOWS-actif et qui aura un coût inférieur ou égal à celui de O. Cet ordonnancement est donc optimal et LOWS-actif. La démonstration est similaire dans le cas où $m(i) = m(j)$. □

Sur la Figure 2.4, on peut voir une représentation schématique de l'imbrication des ensembles d'ordonnancements dominants.

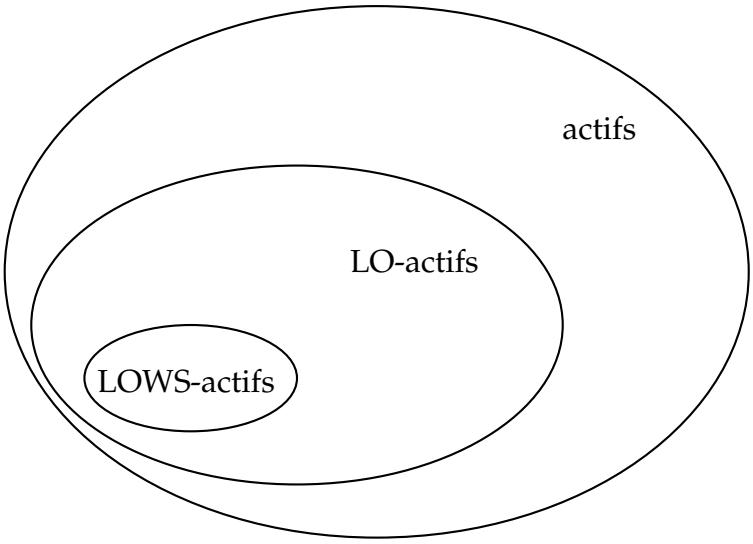


Figure 2.4 – Classes d'ordonnancements dominants.

2.3 Ordonnancements partiels

Nous allons désormais étudier des règles de dominance qui portent sur des ordonnancements partiels , *i.e.* des sous-ensembles de jobs ordonnancés. Nous chercherons à savoir si, pour un même sous-ensemble de jobs ordonnancés, certains ordonnancements partiels sont dominés par d'autres. Un ordonnancement partiel P_1 domine un autre ordonnancement partiel P_2 lorsque le remplacement de l'un par l'autre ne peut être qu'avantageux en terme de coût , *i.e.* tout ordonnancement commençant par P_2 peut être amélioré en remplaçant P_2 par P_1. Ces travaux, menés pour les problèmes à machines parallèles, sont des généralisations des travaux de Jouglet [42, 43] sur une machine.

Nous allons tout d'abord introduire les notions d' « ordonnancement partiel au moins aussi bon que » et de « meilleur ordonnancement partiel que ». Ces définitions sont valables aussi bien pour le cas à une machine que pour le cas à machines parallèles.

Définition 2.7. *Soient P_1 et P_2 deux ordonnancements partiels composés des mêmes jobs. Nous dirons que P_1 est au moins aussi bon que P_2, si pour tout ordonnancement commençant P_2, on peut remplacer P_2 par P_1 sans augmenter le coût de l'ordonnancement.*

Définition 2.8. *Soient P_1 et P_2 deux ordonnancements partiels composés des mêmes jobs. Nous dirons que P_1 est meilleur que P_2, si au moins une des conditions suivantes est vérifiée :*

1. *P_1 est au moins aussi bon que P_2 et P_2 n'est pas au moins aussi bon que P_1 ;*
2. *P_1 est au moins aussi bon que P_2 et P_1 est lexicographiquement plus petit que P_2.*

Définition 2.9. *Un ordonnancement partiel P_2 est dominé s'il existe un ordonnancement partiel P_1 tel que P_1 est meilleur que P_2.*

Nous pouvons constater qu'à partir de la relation « être au moins aussi bon », nous pouvons définir la relation « être meilleur que ». Nous allons désormais décliner ces dominances dans trois cas : sur une machine, sur m machines, et enfin sur k parmi m machines (cas général).

Soit P un ordonnancement partiel. Nous noterons désormais $O = (P|Q)$ un ordonnancement qui commence par l'ordonnancement partiel P.

Soit $F_O(P)$ le coût associé aux jobs appartenant à l'ordonnancement partiel P dans l'ordonnancement O.

2.3.1 RD_1 : avec une machine

Proposition 2.4. *Soit S un sous-ensemble de jobs de $N = \{1, 2, \ldots, n\}$, et NS l'ensemble des jobs non ordonnancés*, i.e. *$NS = N \setminus S$. On note r_{min} la plus petite date de disponibilité parmi les jobs non ordonnancés*, i.e. *$r_{min} = \min_{i \in NS} r_i$. Soit P_1 et P_2 deux ordonnancements partiels sur une machine composés des jobs de S. P_1 est au moins aussi bon que P_2 si au moins l'une des conditions suivantes est vérifiée :*

1. *$C_{max}(P_1) \leq C_{max}(P_2)$ et $F(P_1) \leq F(P_2)$;*
2. *$C_{max}(P_1) > C_{max}(P_2)$ et $F(P_1) + \Delta \sum_{i \in NS} w_i \leq F(P_2)$,*

avec $\Delta = \max\left(C_{max}(P_1), r_{min}\right) - \max\left(C_{max}(P_2), r_{min}\right)$.

Démonstration. Nous devons montrer ici que lorsque l'une des conditions précédentes est vérifiée, on peut construire à partir de tout ordonnancement $O_2 = (P_2|Q)$ un ordonnancement $O_1 = (P_1|Q)$, avec $F(O_1) \leq F(O_2)$.

Supposons que $C_{max}(P_1) \leq C_{max}(P_2)$ et $F(P_1) \leq F(P_2)$. Soit $O_2 = (P_2|Q)$ un ordonnancement commençant par P_2. Comme $C_{max}(P_1) \leq C_{max}(P_2)$, on peut remplacer l'ordonnancement partiel P_2 par l'ordonnancement partiel P_1 dans O_2 sans retarder aucun job de Q. On note O_1 l'ordonnancement ainsi formé. On a $F_{O_1}(Q) \leq F_{O_2}(Q)$. Par ailleurs, on a $F(P_1) \leq F(P_2)$. Finalement, on trouve que $F(O_1) = F(P_1) + F_{O_1}(Q) \leq F(P_2) + F_{O_2}(Q) = F(O_2)$.

Supposons désormais que $C_{max}(P_1) > C_{max}(P_2)$ et $F(P_1) + \Delta \times \sum_{i \in NS} w_i \leq F(P_2)$, avec $\Delta = \max\left(C_{max}(P_1), r_{min}\right) - \max\left(C_{max}(P_2), r_{min}\right)$. Δ représente le décalage maximum des jobs de Q lorsque l'on remplace P_2 par P_1 et que $C_{max}(P_1) > C_{max}(P_2)$. Soit $O_2 = (\sigma_2|Q)$ un ordonnancement commençant par l'ordonnancement partiel P_2. Construisons l'ordonnancement $O_1 = (P_1|Q)$ et comparons les coûts de O_1 et de O_2. Comme $C_{max}(P_1) > C_{max}(P_2)$, les jobs de Q dans l'ordonnancement O_1 risquent d'être retardés par rapport à leurs exécutions dans O_2. Plus

précisément, le décalage induit par le remplacement de P_2 par P_1 est majoré par $\Delta = \max\left(C_{max}(P_1), r_{min}\right) - \max\left(C_{max}(P_2), r_{min}\right)$. La somme des coûts des jobs de Q augmente donc au maximum de $\Delta * \sum_{i \in NS} w_i$ entre O_2 et O_1, donc $F_{O_1}(Q) \leq F_{O_2}(Q) + \Delta \sum_{i \in NS} w_i$. Par ailleurs, on a $F(O_2) = F(P_2) + F_{O_2}(Q)$ et $F(O_1) = F(P_1) + F_{O_1}(Q)$. Or on a supposé que $F(P_1) + \Delta \times \sum_{i \in NS} w_i \leq F(P_2)$. On a donc $F(O_1) = F(P_1) + F_{O_1}(Q) \leq F(P_1) + F_{O_2}(Q) + \Delta \times \sum_{i \in NS} w_i \leq F(P_2) + F_{O_2}(Q) \leq F(O_2)$. Finalement, on a montré que $F(O_1) \leq F(O_2)$.

□

La Figure 2.5 illustre une situation de dominance entre deux ordonnancements partiels sur une machine. Si l'on cherche à minimiser la somme des dates de fin, l'ordonnancement partiel P_1 coûte 10 tandis que P_2 coûte 13.

job i	1	2	3
r_i	2	0	0
p_i	3	2	1

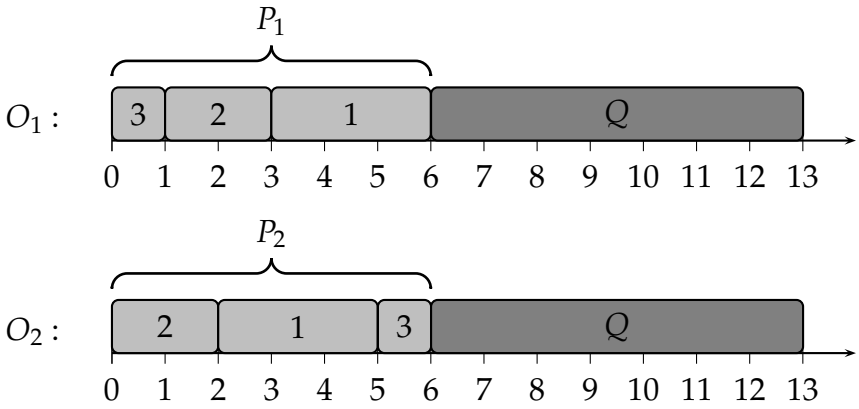


Figure 2.5 – Exemple de dominance sur une machine pour le problème $1|r_i|\sum C_i$.

Définition 2.10. *Soit P un ordonnancement partiel sur m machines. Nous notons $M_i(P)$ l'ordonnancement partiel correspondant aux jobs de l'ordonnancement partiel P exécutés sur la machine i. Nous dirons que $M_i(P)$ est un ordonnancement partiel machine.*

Un exemple est d'ordonnancement partiel machine est donné sur la Figure 2.6.

Proposition 2.5 (RD_1)**.** *Soit P un ordonnancement partiel sur m machines. Soit NS l'ensemble des jobs n'appartenant pas à cet ordonnancement partiel. Soit $M_i(P)$ un ordonnancement partiel machine de P. S'il existe un ordonnancement partiel*

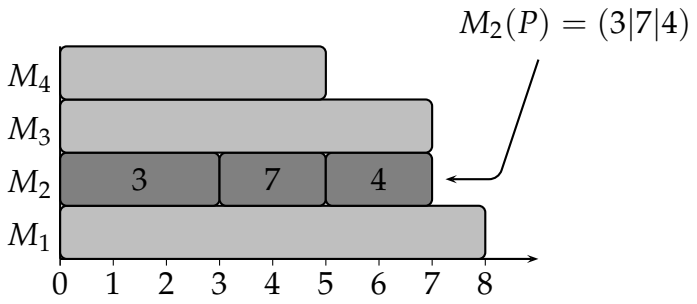


Figure 2.6 – Exemple d'ordonnancement partiel machine.

(sur une machine) P_1 meilleur que $M_i(P)$, alors l'ordonnancement partiel P sur m machines est dominé.

Démonstration. Supposons l'existence de P, de $M_i(P)$ et de P_1. Il est alors aisé de constater que tout ordonnancement sur m machines débutant par l'ordonnancement partiel P est dominé par un ordonnancement où l'ordonnancement partiel machine $M_i(P)$ a été remplacé par P_1. □

2.3.2 RD_m : avec m machines

Nous cherchons désormais à caractériser une situation de dominance entre deux ordonnancements partiels impliquant l'ensemble des machines.

Nous allons définir les conditions sous lesquelles un ordonnancement partiel sur m machines peut être au moins aussi bon qu'un autre ordonnancement partiel. Il nous faut tout d'abord introduire un lemme qui nous permettra de démontrer la proposition qui suivra.

Lemme 2.1. *Soient $a_1, a_2, \ldots, a_m$ et $b_1, b_2, \ldots, b_m$ deux séries de m entiers positifs. Si l'on autorise la renumérotation de ces deux séries, la quantité* $\max_i \big(\max(0, a_i - b_i)\big)$ *est minimale pour $a_1 \leq a_2 \leq \cdots \leq a_m$ et $b_1 \leq b_2 \leq \cdots \leq b_m$.*

Démonstration. Nous allons montrer que cette propriété est vraie lorsque les séries respectent le même ordre. Le cas $a_1 \leq a_2 \leq \cdots \leq a_m$ et $b_1 \leq b_2 \leq \cdots \leq b_m$ étant un cas particulier où l'ordre des valeurs suit l'ordre des indices, nous aurons démontré le lemme.

Supposons que les séries (a_i) et (b_i) ne respectent pas le même ordre. Cela implique qu'il existe i et j tels que $a_i \leq a_j$ et $b_i > b_j$. Les cas symétriques obtenus en échangeant les rôles de i et de j, ou de (a_i) et ceux de (b_i) se démontrent de la même manière. Dans ce cas, l'échange des indices de a_i et de a_j ne peut pas faire augmenter la valeur $\max_i \big(\max(0, a_i - b_i)\big)$. En effet, on a $a_i - b_i < a_j - b_j$, $a_j - b_j > a_i - b_j$ et $a_j - b_j > a_j - b_i$. Par contre elle

peut éventuellement la faire diminuer. Par une procédure itérative et en un nombre fini d'étapes, on peut ainsi renuméroter les séries jusqu'à ce qu'elles respectent le même ordre. De plus, comme aucune de ces étapes ne peut faire augmenter la valeur de $\max_i \big(\max(0, a_i - b_i)\big)$, les séries qui respectent le même ordre minimise cette dernière.

□

Ce lemme nous est utile, puisqu'il nous permet de calculer la valeur maximale du décalage que subissent les jobs de Q entre les ordonnancements $O_1 = (P_1|Q)$ et $O_2 = (P_2|Q)$. En effet, il suffit de définir la série (a_i) à l'aide des dates de fin des machines de P_1 et la série (b_i) à l'aide des dates de fin des machines de P_2, *i.e.* $a_i = C_{max}(M_i(P_1))$ et $b_i = C_{max}(M_i(P_2))$. La renumérotation des machines est possible, puisque les machines sont identiques. Par ailleurs, la quantité $\max_i(\max(0, a_i - b_i))$ correspond effectivement au décalage maximal que subit un job de Q entre les ordonnancements O_1 et O_2.

Nous supposerons désormais que les machines de tous les ordonnancements partiels que nous considérons sont indicées selon leurs dates de fin croissantes.

Proposition 2.6 (RD_m)**.** *Soit S un sous-ensemble de jobs ordonnancés de $N = \{1, 2, \ldots, n\}$, et NS l'ensemble des jobs non ordonnancés. On note r_{min} la plus petite date de disponibilité parmi les jobs non ordonnancés*, i.e. $r_{min} = \min_{i \in NS} r_i$. *Soient P_1 et P_2 deux ordonnancements partiels composés des jobs de S sur m machines. P_1 est au moins aussi bon que P_2 si au moins l'une des conditions suivantes est vérifiée :*

1. $\forall i \in [1, m], C_{max}\big(M_i(P_1)\big) \leq C_{max}\big(M_i(P_2)\big)$ *et* $F(P_1) \leq F(P_2)$;
2. $\exists i \in [1, m], C_{max}\big(M_i(P_1)\big) > C_{max}\big(M_i(P_2)\big)$ *et* $F(P_1) + \Delta \times \sum_{j \in NS} w_j \leq F(P_2)$,

avec $\Delta = \max_{i \in [1,m]} \Big(C_{max}\big(M_i(P_1)\big) - \max\Big(r_{min}, C_{max}\big(M_i(P_2)\big)\Big)\Big)$.

Démonstration. Soit $O_2 = (P_2|Q)$ un ordonnancement qui commence par l'ordonnancement partiel P_2. Supposons qu'il existe un ordonnancement partiel P_1 tel que P_1 soit au moins aussi bon que P_2.

Supposons que P_1 et P_2 vérifient la première condition , *i.e.* $\forall i \in [1, m]$ $C_{max}\big(M_i(P_1)\big) \leq C_{max}\big(M_i(P_2)\big)$ et $F(P_1) \leq F(P_2)$. Dans ce cas, on peut construire l'ordonnancement $O_1 = (P_1|Q)$ et l'on sait qu'aucun job de Q ne commence plus tard dans O_1 que dans O_2 puisque $\forall i \in [1, m]$ $C_{max}\big(M_i(P_1)\big) \leq C_{max}\big(M_i(P_2)\big)$. Comme, par ailleurs on a $F(P_1) \leq F(P_2)$, on peut en conclure que $F(O_1) \leq F(O_2)$.

Supposons maintenant que P_1 et P_2 vérifient la seconde condition , *i.e.* $\exists i \in [1, m] C_{max}\big(M_i(P_1)\big) > C_{max}\big(M_i(P_2)\big)$ et $F(P_1) + \Delta \times \sum_{j \in NS} w_j \leq$

$F(P_2)$, avec $\Delta = \max_{i\in[1,m]}\left(C_{max}(M_i(P_1)) - \max\left(r_{min}, C_{max}(M_i(P_2))\right)\right)$. Soit $O_2 = (P_2|Q)$ un ordonnancement qui commence par l'ordonnancement partiel P_2. Examinons l'ordonnancement O_1, construit en remplaçant P_2 par P_1 dans l'ordonnancement O_2. On sait que $\exists i \in [1,m]\ C_{max}(M_i(P_1)) > C_{max}(M_i(P_2))$, c'est-à-dire que le remplacement de l'ordonnancement partiel P_2 par l'ordonnancement partiel P_1 dans l'ordonnancement peut décaler vers la droite l'exécution des jobs de Q. Nous allons maintenant majorer ce décalage. Lorsque les machines sont indicées selon leurs dates de fin croissantes, on sait que la quantité $\max_{i\in[1,m]}\left(C_{max}(M_i(P_1)) - C_{max}(M_i(P_2))\right)$ est une borne supérieure du décalage que subissent les jobs de Q, si l'on remplace P_2 par P_1. Il est possible de raffiner ce résultat en considérant les dates de disponibilité des jobs de Q. En effet, si un job k de Q a une date de disponibilité plus grande que $C_{max}(M_i(P_2))$, le décalage qu'il subit ne pourra pas excéder $C_{max}(M_i(P_1)) - r_k$. Dans le cas particulier où tous les jobs de Q ont une date de disponibilité plus grande que $C_{max}(M_i(P_2))$, le décalage sur la machine i est majoré par $C_{max}(M_i(P_1)) - r_{min}$. D'une manière générale, on sait que la quantité $\max_{i\in[1,m]}\left(C_{max}(M_i(P_1)) - \max\left(r_{min}, C_{max}(M_i(P_2))\right)\right)$ majore le décalage que subiront les jobs de Q. En tenant compte des poids, on peut affirmer que $\left(\sum_{j\in NS} w_j\right)\max_{i\in[1,m]}\left(C_{max}(M_i(P_1)) - \max\left(r_{min}, C_{max}(M_i(P_2))\right)\right)$ est une borne supérieure de l'augmentation du coût porté par les jobs de Q. Finalement, comme la condition $F(P_1) + \Delta \times \sum_{j\in NS} w_j \leq F(P_2)$ est vérifiée, on peut constater que $F(O_1) \leq F(P_1) + F(Q) + \Delta \times \sum_{j\in NS} w_j$, donc que $F(O_1)$ est inférieur à $F(O_2) = F(P_2) + F(Q)$.

□

Nous pouvons voir sur la Figure 2.7 un exemple sur deux machines, où l'ordonnancement partiel P_1 (jobs en gris clair dans l'ordonnancement O_1) est meilleur que l'ordonnancement partiel P_2 (jobs en gris clair dans l'ordonnancement O_2) pour la somme pondérée des dates de fin, lorsque w_5 est suffisamment grand. Avec les données indiquées, P_2 a un coût égal à 117 tandis que P_1 ne coûte que 91.

On peut également remarquer que la règle RD_1 devient un cas particulier de la règle de dominance RD_m, qui la généralise dans le cas où $m > 1$.

2.3.3 RD_k : avec k machines

La règle que nous allons décrire dans cette section généralise les deux précédentes, à savoir RD_1 et RD_m. Nous cherchons encore une fois à caractériser

job i	1	2	3	4	5
r_i	0	0	0	3	2
p_i	3	3	2	4	5
w_i	1	1	1	1	10

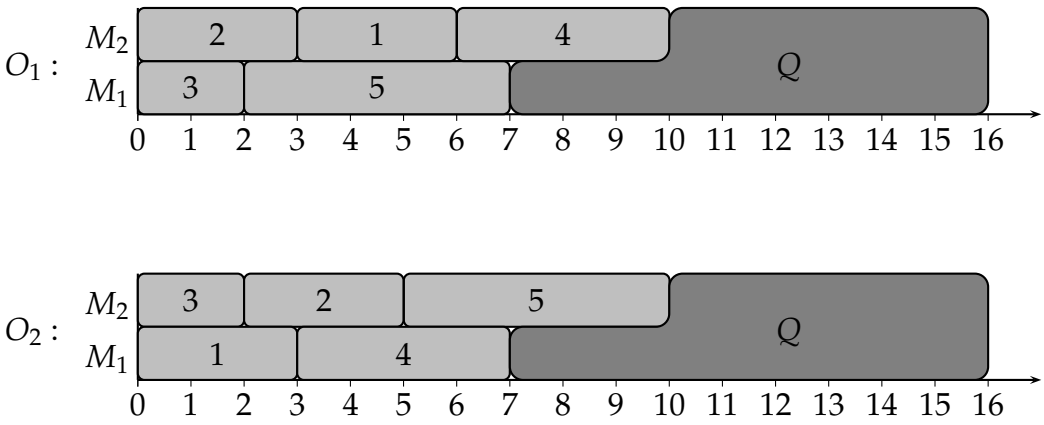


Figure 2.7 – Exemple de dominance selon la règle RD_m.

une situation de dominance sur m machines. Alors que la règle de dominance RD_1 compare les machines une par une et que RD_m compare l'ensemble des m machines, nous cherchons une situation de dominance portant sur un sous-ensemble de k machines.

Définition 2.11. *Soient P un ordonnancement partiel sur m machines et η un sous-ensemble de $[1, m]$. On note $M(P, \eta)$ l'ordonnancement partiel sur $card(\eta)$ machines formé des jobs exécutés dans P sur les machines $\{M_i(P), i \in \eta\}$.*

La Figure 2.8 montre un exemple de sous-ordonnancement partiel avec $\eta = \{1, 4, 5\}$ (machines foncées).

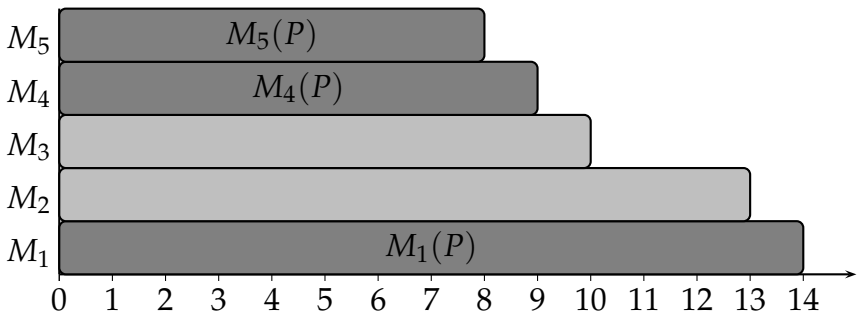


Figure 2.8 – Exemple de sous-ordonnancement partiel.

Proposition 2.7 (RD_k)**.** *Soient P un ordonnancement partiel sur m machines et NS l'ensemble des jobs non ordonnancés dans la séquence P. L'ordonnancement partiel P est dominé s'il existe un sous-ensemble η de $[1,m]$ et un ordonnancement partiel P_k sur $card(\eta)$ machines tels que :*

1. $\forall i \in \eta, C_{max}\big(M_i(P_k)\big) \leq C_{max}\Big(M_i\big(M(P,\eta)\big)\Big)$ *et* $F(P_k) \leq F\big(M(P,\eta)\big)$;
2. $\exists i \in \eta, C_{max}\big(M_i(P_k)\big) > C_{max}\Big(M_i\big(M(P,\eta)\big)\Big)$ *et* $F(P_k) + \Delta \times \sum_{j \in NS} w_j \leq F\big(M(P,\eta)\big)$,

avec $\Delta = \max_{i \in \eta} C_{max}\big(M_i(P_k)\big) - \max\left(r_{min}, C_{max}\Big(M_i\big(M(P,\eta)\big)\Big)\right)$.

Démonstration. Considérons un ordonnancement $O = (P|Q)$ sur m machines. Puisqu'il existe un ordonnancement partiel P_k qui est meilleur que le sous-ordonnancement partiel $r(P,\eta)$ extrait de P, on peut remplacer dans l'ordonnancement O le sous-ordonnancement partiel $M(P,\eta)$ par P_k pour former un ordonnancement O', qui sera meilleur que l'ordonnancement O. □

La Figure 2.9 illustre une possible situation de dominance, avec $k = 3$ et $m = 5$. Les jobs en gris foncé représentent les deux sous-ordonnancements partiels que l'on compare. Si l'on veut minimiser la somme des dates de fin, $M(P,\eta)$ coûte 44 tandis que P_k coûte 43.

Nous pouvons constater que cette règle généralise les règles de dominance RD_1 et RD_m, qui peuvent désormais être vues comme des cas particuliers de RD_k (avec $k = 1$ et $k = m$).

2.4 Jobs non ordonnancés

Nous présentons ici une règle de dominance portant sur les jobs non ordonnancés. Nous nous plaçons dans la situation suivante. Nous disposons d'un ordonnancement partiel P et d'un ensemble de jobs non ordonnancés NS. Soient deux jobs j et k de NS. On suppose que le job k est ordonnancé au plus tôt après l'ordonnancement P. On sait que le job j sera également ordonnancé dans tout ordonnancement complet. Nous cherchons alors les conditions sous lesquelles il est préférable d'échanger les jobs j et k, quelle que soit la position de j. On dit alors que le job j domine le job k pour l'ordonnancement partiel P. Nous supposons en outre que le critère étudié est le retard total pondéré, qui correspond au cas le plus général. Par ailleurs, nous supposons que les dates de disponibilité des jobs de NS ont été ajustées, *i.e.* les dates de disponibilité inférieures à la plus petite date δ à laquelle une machine est disponible après l'ordonnancement partiel P sont désormais égales à δ. Enfin, nous nous plaçons dans le cas où $w_j \geq w_k$. Cette étude, s'inspire des travaux de Jouglet sur une machine [42] et les généralise pour le cas où l'on dispose de plusieurs machines parallèles.

job i	1	2	3	5	7	8	10	13
r_i	2	0	0	1	6	0	0	5
p_i	2	2	4	5	4	1	3	3

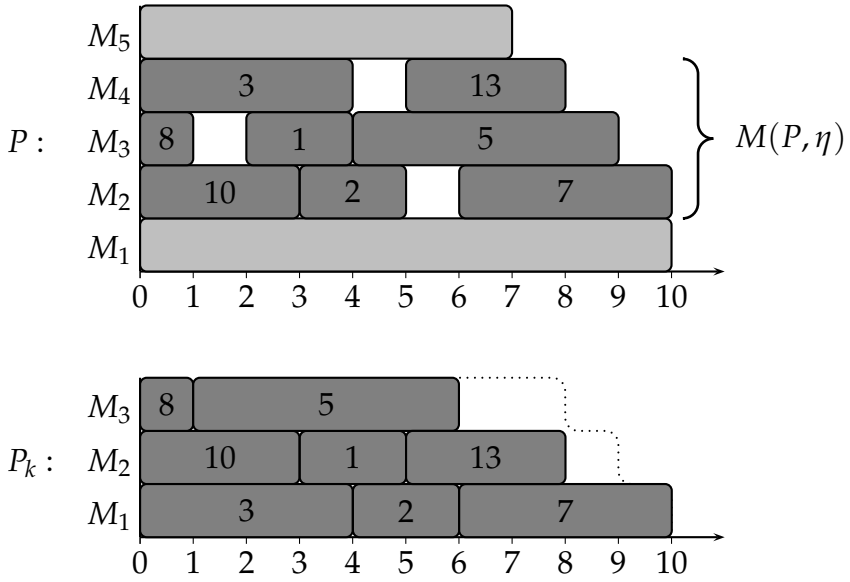


Figure 2.9 – Exemple de dominance selon la règle RD_k.

2.4.1 Principe général

Nous décrivons ici de manière plus formelle la méthode que nous proposons pour mettre en évidence des situations de dominance entre deux jobs par rapport un ordonnancement partiel donné.

Soient P un ordonnancement partiel et NS l'ensemble des jobs non ordonnancés dans P. Soient j et k deux jobs de NS. Soit O un ordonnancement complet commençant par P, *i.e.* $O = (P|Q)$. Supposons que le job k est ordonnancé le plus tôt possible à la suite de P. On sait également que le job j est ordonnancé quelquepart dans Q. La Figure 2.10 illustre la situation que nous venons de décrire.

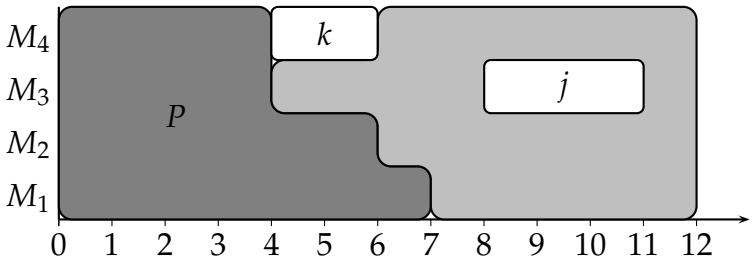


Figure 2.10 – Situation initiale.

On s'intéresse désormais à l'ordonnancement O' formé à partir de l'or-

donnancement O en échangeant les positions des jobs j et k. On sait déjà que O' peut s'écrire sous la forme $O' = (P|Q')$. La Figure 2.11 montre l'ordonnancement O' obtenu à partir de l'ordonnancement O présenté sur la Figure 2.10.

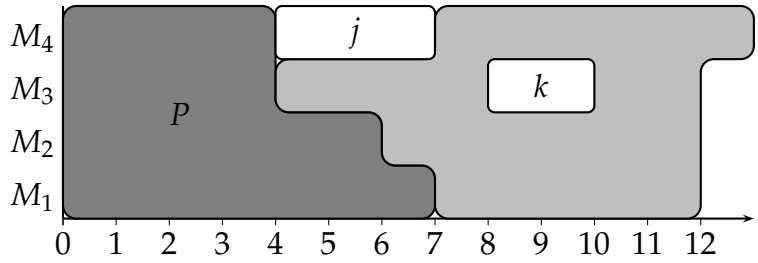


Figure 2.11 – Situation après échange des jobs j et k.

Il nous importe de comparer les coûts des ordonnancements O et O'. Plus précisément, nous cherchons à calculer une borne inférieure ϕ de $F(O) - F(O')$. Si ϕ est positive, on peut conclure que O' domine O, donc que le job j domine le job k par rapport à l'ordonnancement partiel P.

Nous allons maintenant présenter les grandes étapes du calcul de ϕ. Il nous faut envisager toutes les situations possibles quant à la position du job j. En particulier, on peut distinguer deux grands cas :

1. les jobs j et k sont ordonnancés sur la même machine dans O ;
2. les josb j et k sont ordonnancés sur deux machines différentes dans O.

Ces deux cas seront traités dans la suite de manière séparée.

2.4.1.1 Jobs j et k sur la même machine

Nous supposons ici que les jobs j et k sont ordonnancés sur la même machine. Soit A l'ensemble des jobs ordonnancés sur cette machine, entre le job k et le job j. Nous notons B l'ensemble des jobs ordonnancés sur cette même machine après le job j. Enfin, nous notons C l'ensemble des jobs n'appartenant pas à P et qui ne sont pas exécutés sur cette machine. Nous n'avons aucune connaissance *a priori* sur le contenu des ensembles A, B et C. Soient t la date de début d'exécution du job j dans O et δ la plus petite date de fin des machines de P. Soit H une borne supérieure du *makespan* de tout ordonnancement actif. Cette situation est représentée sur la Figure 2.12.

Soit $\phi_1(t)$ une borne inférieure de $F(O) - F(O')$, lorsque j et k sont exécutés sur la même machine et que le job j est exécuté à la date t. On sait que :

$$\begin{aligned} F(O) - F(O') \;=\; & F_O(P) + F_O(k) + F_O(j) + F_O(A) + F_O(B) + F_O(C) \\ & - F_{O'}(P) - F_{O'}(k) - F_{O'}(j) - F_{O'}(A) - F_{O'}(B) - F_{O'}(C). \end{aligned}$$

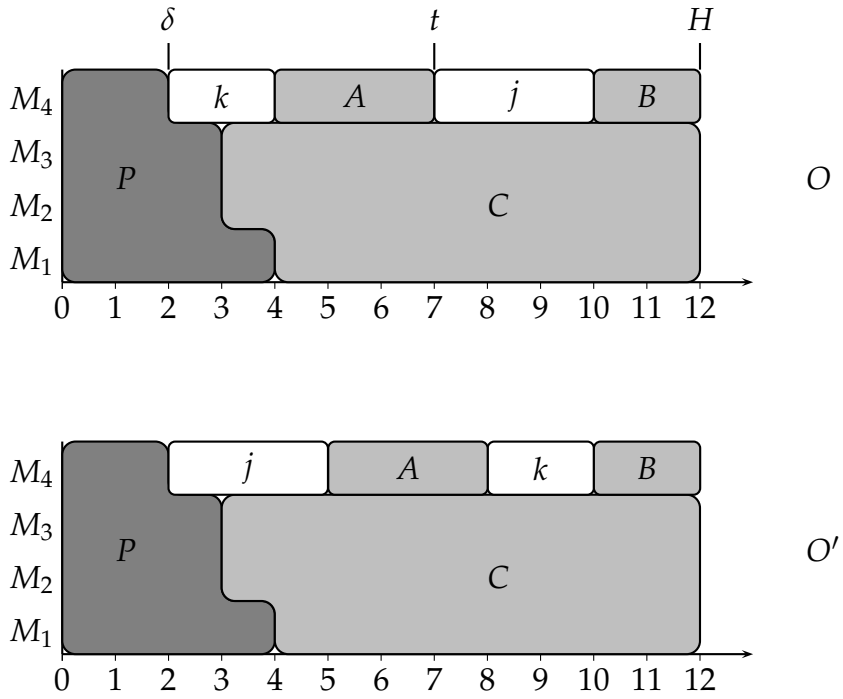


Figure 2.12 – Ordonnancements O et O' lorsque j et k sont sur la même machine.

Or les parties P et C sont identiques dans les ordonnancements O et O', donc

$$\begin{aligned} F(O) - F(O') &= F_O(k) + F_O(j) - F_{O'}(k) - F_{O'}(j) \\ &\quad + F_O(A) + F_O(B) - F_{O'}(A) - F_{O'}(B). \end{aligned}$$

Nous allons maintenant séparer la quantité $F(O) - F(O')$ en deux parties : la différence des coûts portés par les jobs j et k entre O et O' et la différence des coûts portés par les jobs de A et de B entre O' et O. Une borne inférieure de la première sera notée $jk_1(t)$, *i.e.*

$$jk_1(t) \leq F_O(k) + F_O(j) - F_{O'}(k) - F_{O'}(j);$$

nous notons $ab_1(t)$ une borne supérieure de la seconde, *i.e.*

$$ab_1(t) \geq F_{O'}(A) + F_{O'}(B) - F_O(A) - F_O(B).$$

Cela nous permet de formuler ϕ_1 de la manière suivante :

$$\phi_1(t) = jk_1(t) - ab_1(t).$$

Afin de déterminer ϕ_1, il convient d'étudier ses variations. Nous allons pour cela étudier les variations des fonctions jk_1 et ab_1. Nous choisissons la fonction jk_1 de la manière suivante :

$$\begin{aligned} jk_1(t) &= w_k \max(0, r_k + p_k - d_k) + w_j \max(0, t + p_j - d_j) \\ &\quad -w_k \max(0, t + \tau + p_k - d_k) - w_j \max(0, r_j + p_j - d_j), \end{aligned}$$

où τ est une borne supérieure de la différence entre la date de début de j dans O et celle de k dans O'. Si l'on pose $a = d_j - p_j$ et $b = d_k - p_k - \tau$, il faut envisager deux cas pour connaître les variations de la fonction jk_1 :

si $a > b$ **:** Avant b, la fonction est constante. Entre b et a, elle est décroissante puisque le job k est en retard dans O' tandis que j ne l'est pas dans O. Enfin, après a, la fonction est croissante puisque k dans O' et j dans O sont en retard et que $w_j \geq w_k$. La Figure 2.13 illustre ce comportement. En outre, le minimum est atteint lorsque $t = a$.

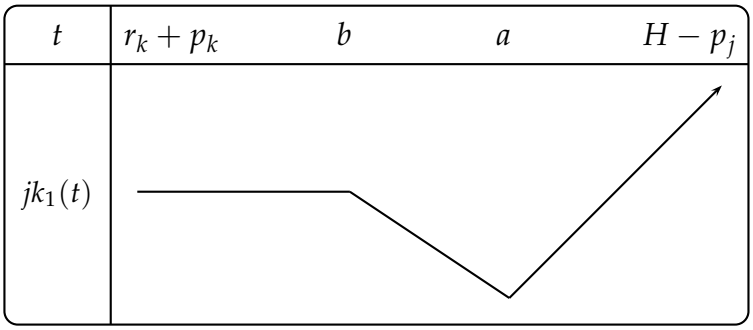


Figure 2.13 – Comportement de la fonction jk_1 lorsque $a > b$.

si $a \leq b$ **:** Dans ce cas, la fonction est constante jusqu'à a, puis croissante à partir de a, à cause du retard du job j dans O qui augmente. À partir de b, le job k dans O' est également en retard, mais la fonction reste croissante puisque $w_j \geq w_k$. La Figure 2.14 illustre cette situation. En outre, le minimum est atteint lorsque $t = a$.

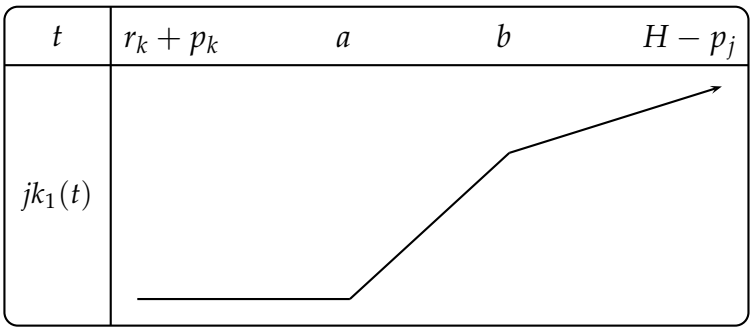


Figure 2.14 – Comportement de la fonction jk_1 lorsque $a \leq b$.

Nous pouvons constater que dans les deux cas, la fonction jk_1 atteint un minimum lorsque $t = a$. On sait par ailleurs que $t \geq r_k + p_k$. Si $a < r_k + p_k$, alors le minimum est atteint en $r_k + p_k$. Notons $a' = \max(d_j - p_j, r_k + p_k)$. D'une manière générale, on sait que la fonction jk_1 est minimale en a'. Nous noterons également $b' = \max(r_k + p_k, d_k - p_k - \tau)$.

Intéressons-nous désormais à la fonction ab_1. Cette dernière est, par définition, une borne supérieure de la différence du coût porté par les jobs de A et de B entre les ordonnancements O' et O. On peut remarquer que lorsqu'un job i est décalé vers la droite de x unités de temps, la différence de coût induite ne peut pas excéder xw_i. Partant de ce constat, nous allons chercher à calculer cette borne supérieure en affectant certains poids à A et d'autres à B. Nous introduisons les variables suivantes :

γ_a **:** borne supérieure du décalage à droite des jobs de A entre O' et O ;

γ_b **:** borne supérieure du décalage à droite des jobs de B entre O' et O.

On peut alors exprimer ab_1 sous la forme :

$$ab_1(t) = \omega_A(t)\gamma_A + \omega_B(t)\gamma_B.$$

Les fonctions ω_A et ω_B vont s'exprimer sous la forme d'une somme de poids de jobs de NS. Il nous importe ici que le choix des valeurs de $\omega_A(t)$ et de $\omega_B(t)$ soit tel que $ab_1(t)$ soit une borne supérieure de la différence des coûts de A et B entre O' et O. Nous pouvons d'abord affirmer que la partie A n'excédera pas $l_A(t) = t - p_k - r_k$ unités de temps, puisqu'elle ne pourra pas commencer avant $r_k + p_k$ et ne pourra pas terminer après t. On peut également savoir que $l_B(t) = H - t - p_j$ est une borne supérieure de la longueur de la partie B.

Afin de trouver des valeurs convenables pour $\omega_A(t)$ et $\omega_B(t)$, nous utilisons la technique de *splitting* de Belouadah *et al.* [13]. Nous définissons l'ensemble NS' de la manière suivante : chaque job i de NS est découpé en p_i morceaux $J_{il}, l \in [1, p_i]$ de durée unitaire de poids w_i/p_i que l'on place dans l'ensemble NS'. Ces morceaux sont ensuite réindicés selon l'ordre des poids décroissants. Nous appellerons $e_1, e_2, \ldots, e_{\sum_{i \in NS} p_i}$ ces morceaux. Il y a ensuite deux cas à envisager :

si $\gamma_A > \gamma_B$ **:** Les valeurs $\omega_A(t) = \sum_{u=1}^{l_A(t)} e_u$ et $\omega_B(t) = \sum_{u=1}^{l_B(t)} e_{u+l_A(t)}$ sont alors valides pour que $ab_1(t)$ soit une borne supérieure. La fonction $-ab_1$ est alors décroissante en fonction de t (voir Figure 2.15).

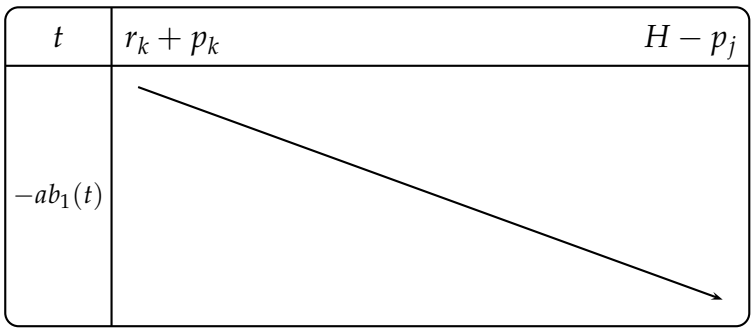


Figure 2.15 – Comportement de la fonction $-ab_1$ lorsque $\gamma_A \geq \gamma_B$.

si $\gamma_A \leq \gamma_B$ **:** Les valeurs $\omega_B(t) = \sum_{u=1}^{l_B(t)} e_u$ et $\omega_A(t) = \sum_{u=1}^{l_A(t)} e_{u+l_B(t)}$ sont alors valides pour que $ab_1(t)$ soit une borne supérieure. La Figure 2.16 illustre le comportement de la fonction $-ab_1$ dans ce cas.

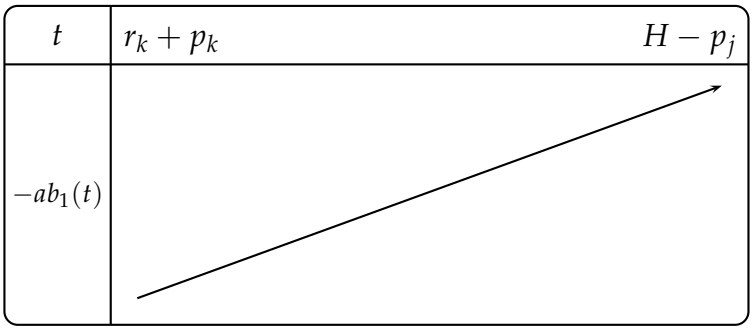


Figure 2.16 – Comportement de la fonction $-ab_1$ lorsque $\gamma_A < \gamma_B$.

2.4.1.2 Jobs j et k sur deux machines différentes

Nous allons désormais étudier le cas où les jobs j et k sont ordonnancés sur deux machines différentes. La méthode présentée pour déterminer si le job k domine le job j par rapport à l'ordonnancement partiel P est similaire à celle présentée dans le cas où les jobs sont ordonnancés sur la même machine, mais les rôles des parties A et B sont différents.

Nous notons A l'ensemble des jobs ordonnancés après le job k et sur la même machine. Notons C l'ensemble des jobs ordonnancés sur la même machine que le job j, avant celui-ci et après l'ordonnancement partiel P. Notons B l'ensemble des jobs ordonnancés après le job j sur la même machine. Notons enfin D l'ensemble des jobs n'appartenant pas à P et qui ne sont pas exécutés sur les machines des jobs k et j. Une fois encore, nous n'avons aucune connaissance *a priori* sur les ensembles A, B, C et D. De plus, nous notons t la date de début d'exécution du job j dans O et δ la plus petite date de fin des machines de P. Nous notons de plus t_{min} le maximum entre la deuxième plus petite date de fin des machines de P et la date de disponibilité du job j. Une illustration de cette situation est représentée sur la Figure 2.17.

Soit $\phi_2(t)$ une borne inférieure de $F(O) - F(O')$, lorsque j et k sont exécutés sur deux machines différentes et que le job j est exécuté à la date t. En outre, on sait que

$$\begin{aligned} F(O) - F(O') &= F_O(P) + F_O(k) + F_O(j) \\ &\quad + F_O(A) + F_O(B) + F_O(C) + F_O(D) \\ &\quad - F_{O'}(P) - F_{O'}(k) - F_{O'}(j) \\ &\quad - F_{O'}(A) - F_{O'}(B) - F_{O'}(C) - F_{O'}(D). \end{aligned}$$

Or les parties P, C et D sont identiques dans les ordonnancements O et O', donc

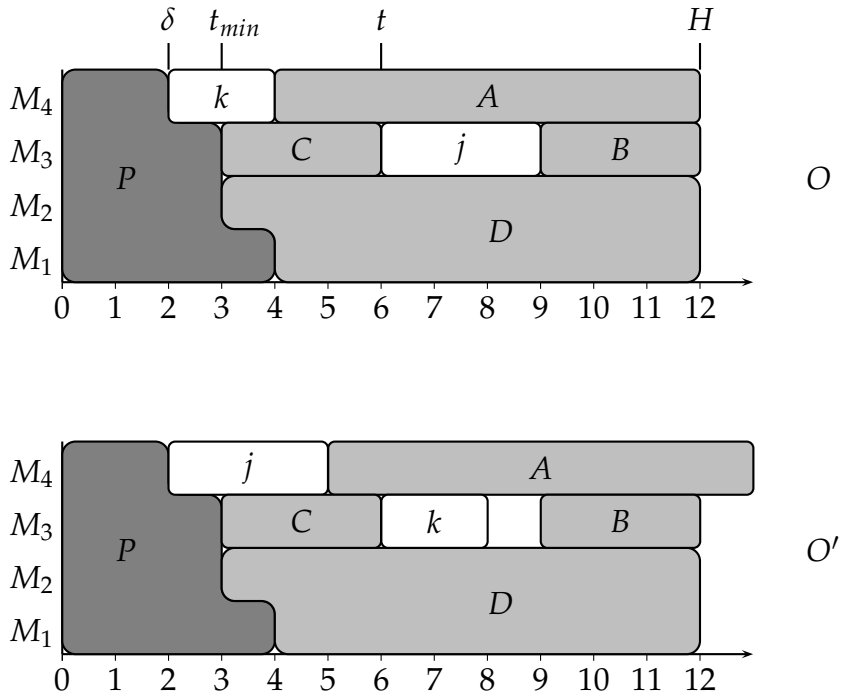


Figure 2.17 – Ordonnancements O et O' lorsque j et k sont sur deux machines différentes.

$$\begin{aligned} F(O) - F(O') &= F_O(k) + F_O(j) - F_{O'}(k) - F_{O'}(j) \\ &\quad + F_O(A) + F_O(B) - F_{O'}(A) - F_{O'}(B). \end{aligned}$$

Ici, la quantité $F(O) - F(O')$ est décomposée en trois parties : la différence des coûts portés par les jobs j et k entre O et O' ; la différence des coûts portés par les jobs de A entre O' et O et la différence des coûts portés par les jobs de B entre O' et O. Une borne inférieure de la première sera notée $jk_2(t)$, *i.e.*

$$jk_2(t) \leq F_O(k) + F_O(j) - F_{O'}(k) - F_{O'}(j);$$

nous notons $a_2(t)$ une borne supérieure de la seconde, *i.e.*

$$a_2(t) \geq F_{O'}(A) - F_O(A);$$

et nous notons $b_2(t)$ une borne supérieure de la troisième, *i.e.*

$$b_2(t) \geq F_{O'}(B) - F_O(B).$$

Enfin, la somme des fonctions a_2 et b_2 est notée

$$ab_2(t) = a_2(t) + b_2(t).$$

Cela nous permet de formuler ϕ_2 de la manière suivante :

$$\phi_2(t) = jk_2(t) - ab_2(t).$$

Nous cherchons le minimum de la fonction ϕ_2. Pour ce faire nous allons étudier les variations des fonctions jk_2 et ab_2. Nous choisissons la fonction jk_2 de la manière suivante :

$$\begin{aligned} jk_2(t) &= w_k \max(0, r_k + p_k - d_k) + w_j \max(0, t + p_j - d_j) \\ &\quad - w_k \max(0, t + \tau + p_k - d_k) - w_j \max(0, r_j + p_j - d_j), \end{aligned}$$

où τ est une borne supérieure de la différence entre la date de début de j dans O et celle de k dans O'. Si l'on pose $a = d_j - p_j$ et $b = d_k - p_k - \tau$, il faut envisager deux cas pour connaître les variations de la fonction jk_2 :

si $a > b$ **:** Dans ce cas, la fonction jk_2 est nulle jusqu'à $t = b$, décroissante entre $t = b$ et $t = a$ puis croissante à partir de $t = a$. La Figure 2.18 illustre le comportement de la fonction jk_2. En outre, le minimum est atteint lorsque $t = a$.

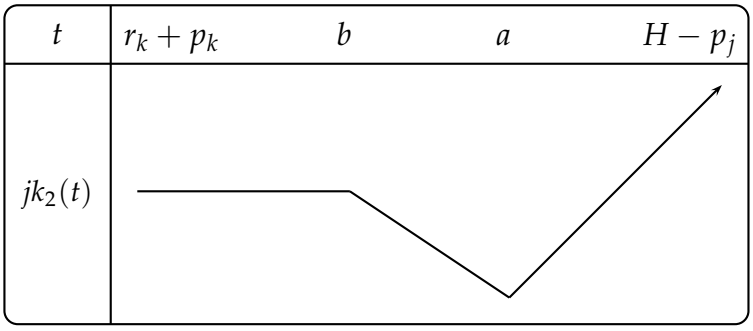


Figure 2.18 – Comportement de la fonction jk_2 lorsque $a > b$.

si $a \leq b$ **:** Dans ce cas, la fonction jk_2 est nulle jusqu'à $t = a$ et croissante après. La Figure 2.19 illustre le comportement de la fonction jk_2. En outre, un minimum est atteint lorsque $t = a$.

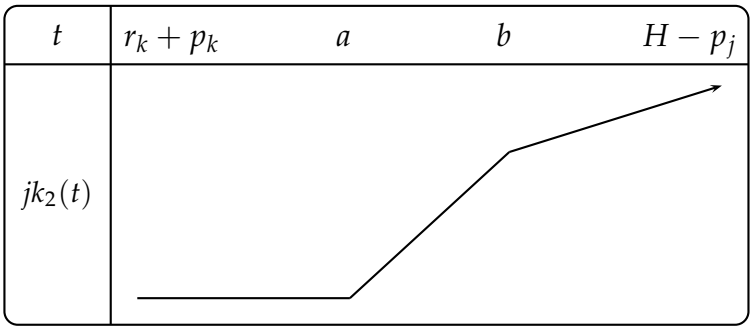


Figure 2.19 – Comportement de la fonction jk_2 lorsque $a \leq b$.

Nous pouvons constater que dans les deux cas, la fonction jk_2 atteint un minimum lorsque $t = a$. Si $t_{min} > a$, alors la fonction est trivialement croissante et minimale en t_{min}. Nous pouvons donc dire que dans tous les cas la fonction jk_2 est minimale en $a' = \max(t_{min}, a)$. Nous définissons de même $b' = \max(t_{min}, b)$.

Afin d'étudier la fonction ab_2 , nous allons procéder de la même manière que dans le cas où les jobs j et k sont ordonnancés sur la même machine : pour chacune des parties A et B, nous affectons une somme de poids ω à cette partie, que nous multiplions ensuite par une majoration du décalage à droite γ de la partie concernée, de sorte que le produit obtenu $\omega\gamma$ soit une borne supérieure de la différence du coût porté par les jobs de la partie concernée entre les ordonnancements O' et O.

Nous notons γ_A le décalage maximal vers la droite que subissent les jobs de A entre O et O'. La longueur maximale de A est notée l_A. Enfin, la somme des poids affectée à la partie S est notée $\omega_A(t)$. Finalement, la quantité a_2 peut s'écrire :

$$a_2(t) = \omega_A(t)\gamma_A.$$

Intéressons-nous à la fonction b_2. Cette dernière est, par définition, une borne supérieure de la différence du coût porté par les jobs de B entre les ordonnancements O' et O. Nous notons γ_B une borne supérieure du décalage à droite des jobs de B entre O' et O. La fonction b_2 s'écrit de la manière suivante :

$$b_2(t) = \omega_B(t)\gamma_B.$$

Nous pouvons d'abord affirmer que la partie A n'excède pas $l_A = H - p_k - r_k$ unités de temps, puisqu'elle ne peut pas commencer avant $r_k + p_k$ et ne peut pas terminer après H. On peut également voir que $l_B(t) = H - t - p_j$ est une borne supérieure de la longueur de la partie B.

Afin de trouver des valeurs convenables pour $\omega_A(t)$ et $\omega_B(t)$, nous utilisons la technique de *splitting* de Belouadah *et al.* [13]. Nous définissons l'ensemble NS' de la manière suivante : chaque job i de NS est découpé en p_i morceaux $J_{il}, l \in [1, p_i]$ de durée unitaire de poids w_i/p_i que l'on place dans l'ensemble NS'. Ces morceaux sont ensuite réindicés selon l'ordre des poids décroissants. Nous appellerons $e_1, e_2, \ldots, e_{\sum_{i \in NS} p_i}$ ces morceaux. Il y a ensuite deux cas à envisager :

si $\gamma_A > \gamma_B$: Les valeurs $\omega_A(t) = \sum_{u=1}^{l_A} e_u$ et $\omega_B(t) = \sum_{u=1}^{l_B(t)} e_{u+l_A}$ sont alors valides pour que $a_2(t) + b_2(t)$ soit une borne supérieure. En fait, lorsque $\gamma_A \geq \gamma_B$, on peut constater que la fonction ω_A est constante par rapport à t. Les variations de ab_2 sont donc fixées par celles de la fonction b_2. Lorsque t augmente, $l_B(t)$ diminue, donc $\omega_B(t)$ diminue également. La fonction ab_2 est donc décroissante quand t augmente. Finalement, la fonction $-ab_2$ est croissante en fonction de t (voir Figure 2.20).

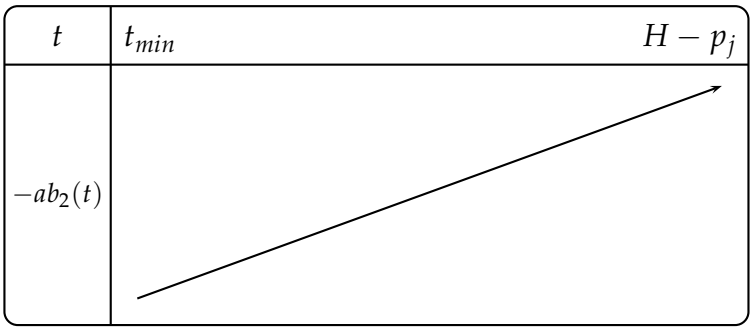


Figure 2.20 – Comportement de la fonction $-ab_2$ lorsque $\gamma_A > \gamma_B$.

si $\gamma_A \leq \gamma_B$ **:** Les valeurs $\omega_B(t) = \sum_{u=1}^{l_B(t)} e_u$ et $\omega_A(t) = \sum_{u=1}^{l_A} e_{u+l_B(t)}$ sont alors valides pour que $ab_2(t)$ soit une borne supérieure. La fonction a_2 est alors croissante lorsque t augmente. En effet, plus t est grand, plus les élements de NS' affectés à A ont une grande valeur, puisque l'on va en affecter de moins en moins à B. Par ailleurs, la fonction b_2 est naturellement décroissante. Cependant, on peut montrer que la fonction ab_2 sera globalement décroissante. Il suffit pour cela de calculer $ab_2(t) - ab_2(t+1)$. On a

$$\begin{aligned} ab_2(t) - ab_2(t+1) &= \omega_A(t)\gamma_A + \omega_B(t)\gamma_B \\ &\quad -\omega_A(t+1)\gamma_A - \omega_B(t+1)\gamma_B \\ &= \sum_{u=1}^{l_B(t)} e_u - \sum_{u=1}^{l_B(t+1)} e_u \\ &\quad + \sum_{u=1}^{l_A} e_{u+l_B(t)} - \sum_{u=1}^{l_A} e_{u+l_B(t+1)} \\ &= e_{l_B(t)}(\gamma_B - \gamma_A) + e_{l_B(t)+l_A(t)}\gamma_B. \end{aligned}$$

Or, on sait que $\gamma_B \geq \gamma_A$, que $\gamma_B > 0$ et que $\forall u \in [1, \sum p_i], e_u > 0$. On peut donc conclure que

$$ab_2(t) - ab_2(t+1) > 0.$$

La fonction ab_2 est décroissante quand t augmente. Ainsi, la fonction $-ab_2$ est croissante entre t_{min} et $H - p_j$, ce qui est représenté sur la Figure 2.21.

2.4.2 Les différents cas pour τ, γ_A et γ_B

Nous allons maintenant présenter la façon dont sont obtenues les valeurs de τ, de γ_A et de γ_B. Nous séparons encore deux grands cas : lorsque les jobs j et k sont ordonnancés sur la même machine et lorsqu'ils sont ordonnancés sur deux machines différentes. Nous présentons à chaque fois la valeur de ϕ_1 ou de ϕ_2 correspondant à une borne inférieure de $F(O) - F(O')$.

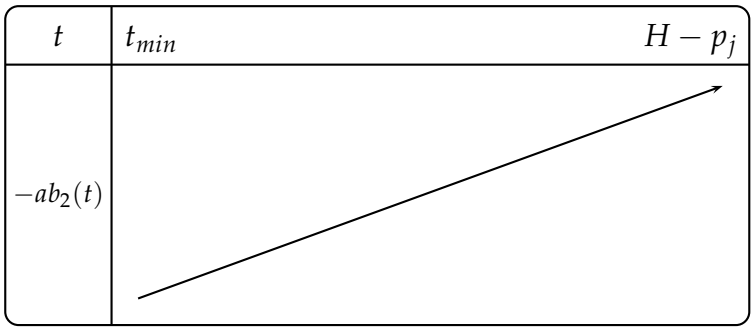


Figure 2.21 – Comportement de la fonction $-ab_2$ lorsque $\gamma_A < \gamma_B$.

Pour chacun des deux cas, nous présentons un arbre de décision, fonction des données initiales (les dates de disponibilités de j et k, les durées de j et k, ...), qui permet de déterminer le calcul de ϕ à effectuer.

2.4.2.1 Jobs j et k sur la même machine

Il y a ici dix cas à considérer suivant les données initiales. La Figure 2.22 présente l'arbre menant aux dix cas. Nous allons désormais traiter chacun de ces cas séparement :

Cas 1 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i \leq r_k + p_k$, les jobs de A sont ordonnancés $r_k + p_k - \max(r_j + p_j, \max_{i \in NS} r_i)$ unités de temps plus tôt dans O' que dans O. Le job k dans O' commence donc $r_k + p_k - \max(r_j + p_j, \max_{i \in NS}$ unités de temps plus tôt que le job j dans O, donc $\tau = \max(r_j + p_j, \max_{i \in NS} r_i) - r_k - p_k$. Enfin, la partie B commence dans O à la date $r_k + p_k + p_A + p_j$, où p_A indique la durée de la partie A. Dans O', B commence à $\max(\max_{i \in NS} r_i, r_j + p_j) + p_A + p_k$. Finalement, puisque l'on sait que $\max(\max_{i \in NS} r_i, r_j + p_j) > r_k + p_j$, on peut conclure que $\gamma_B = \max(\max_{i \in NS} r_i, r_j + p_j) - r_k - p_j$. On sait enfin que $a \leq b$.
La fonction jk_1 est donc croissante entre $r_k + p_k$ et $H - p_j$ (voir Figure 2.13), la fonction $-ab_1$ également (Figure 2.16). La fonction ϕ_1 est alors globalement croissante, donc elle atteint son minimum en $t = r_k + p_k$. Finalement, on a :

$$\phi_1 = jk_1(r_k + p_k) - ab_1(r_k + p_k).$$

Cas 2 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i \leq r_k + p_k$, les jobs de A sont ordonnancés $r_k + p_k - \max(r_j + p_j, \max_{i \in NS} r_i)$ unités de temps plus tôt dans O' que dans O. Le job k dans O' commence donc $r_k + p_k - \max(r_j + p_j, \max_{i \in NS}$ unités de temps plus tôt que le job j dans O, donc $\tau = \max(r_j + p_j, \max_{i \in NS} r_i) - r_k - p_k$. Enfin, la partie B commence dans O à la date $r_k + p_k + p_A + p_j$, où p_A indique la durée de la partie A. Dans O', B commence à $\max(\max_{i \in NS} r_i, r_j + p_j) + p_A + p_k$. Finalement, puisque l'on sait que $\max(\max_{i \in NS} r_i, r_j + p_j) > r_k + p_j$, on peut conclure que $\gamma_B = \max(\max_{i \in NS} r_i, r_j + p_j) - r_k - p_j$. On sait enfin que $a > b$.

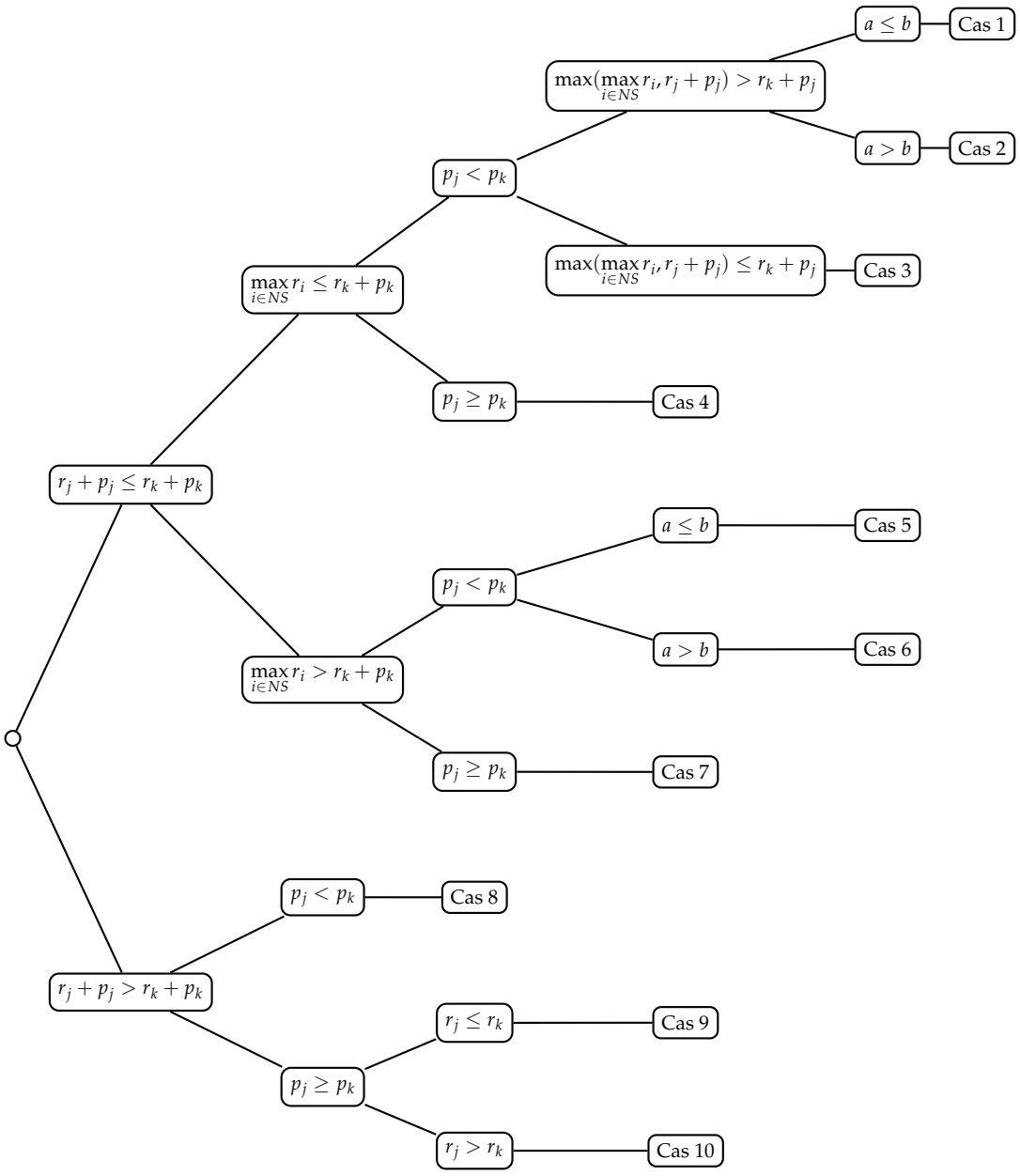


Figure 2.22 – Arbre de décision lorsque j et k sont ordonnancés sur la même machine.

La fonction jk_1 est donc constante entre $r_k + p_k$ et b, décroissante entre b et a puis croissante entre a et $H - p_j$ (voir Figure 2.14). La fonction $-ab_1$ est croissante entre $r_k + p_k$ et $H - p_j$ (Figure 2.16). La fonction ϕ_1 est donc croissante sur les intervalles $[r_k + p_k, b]$ et $[a, H - p_j]$. Le minimum se trouve donc soit en $r_k + p_k$, soit dans l'intervalle $]b, a]$. Comme on sait que la fonction jk_1 est décroissante sur cet intervalle et que la fonction $-ab_1$ est elle croissante, on peut affirmer que la quantité $jk_1(a') - ab_1(b')$ est inférieure à toute valeur de $\phi_1(t)$ sur cet intervalle. Finalement, on a :

$$\phi_1 = \min\Big(jk_1(r_k + p_k) - ab_1(r_k + p_k), jk_1(a') - ab_1(b')\Big).$$

Cas 3 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i \leq r_k + p_k$, les jobs de A sont ordonnancés $r_k + p_k - \max(r_j + p_j, \max_{i \in NS} r_i)$ unités de temps plus tôt dans O' que dans O. Le job k dans O' commence donc $r_k + p_k - \max(r_j + p_j, \max_{i \in NS}$ unités de temps plus tôt que le job j dans O, donc $\tau = \max(r_j + p_j, \max_{i \in NS} r_i) - r_k - p_k$. Enfin, la partie B commence dans O à la date $r_k + p_k + p_A + p_j$, où p_A indique la durée de la partie A. Dans O', B commence à $\max(\max_{i \in NS} r_i, r_j + p_j) + p_A + p_k$. Finalement, puisque l'on sait que $\max(\max_{i \in NS} r_i, r_j + p_j) \geq r_k + p_j$, on peut conclure que $\gamma_B = 0$.
La fonction $-ab_1$ est donc nulle, puisque $\gamma_A = 0$ et $\gamma_B = 0$. La fonction ϕ_1 est alors égale à la fonction jk_1, qui atteint son minimum en $t = a'$ (voir Figure 2.13 et 2.14). Finalement, on a :

$$\phi_1 = jk_1(a').$$

Cas 4 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i \leq r_k + p_k$, les jobs de A sont ordonnancés $r_k + p_k - \max(r_j + p_j, \max_{i \in NS} r_i)$ unités de temps plus tôt dans O' que dans O. Le job k dans O' commence donc $r_k + p_k - \max(r_j + p_j, \max_{i \in NS}$ unités de temps plus tôt que le job j dans O, donc $\tau = \max(r_j + p_j, \max_{i \in NS} r_i) - r_k - p_k$. Enfin, comme $p_j \geq p_k$ et comme la partie A ne subit pas de décalage vers la droite, la partie ne peut pas non plus subir de décalage vers la droite entre O et O', donc $\gamma_B = 0$.
La fonction $-ab_1$ est donc nulle, puisque $\gamma_A = 0$ et $\gamma_B = 0$. La fonction ϕ_1 est alors égale à la fonction jk_1, qui atteint son minimum en $t = a'$ (voir Figure 2.13 et 2.14). Finalement, on a :

$$\phi_1 = jk_1(a').$$

Cas 5 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i > r_k + p_k$, on ne sait pas si les jobs de A peuvent être décalés vers la gauche dans O'. Il faut considérer que le job k dans O' commence à la date t, donc $\tau = 0$. On sait également que $p_j < p_k$ donc les jobs de B subissent un décalage maximum vers la droite de $p_k - p_j$, d'où $\gamma_B = p_k - p_j$. Enfin, on a $a \leq b$.

La fonction jk_1 est donc croissante entre $r_k + p_k$ et $H - p_j$ (voir Figure 2.13), la fonction $-ab_1$ également (Figure 2.16). La fonction ϕ_1 est alors globalement croissante, donc elle atteint son minimum en $t = r_k + p_k$. Finalement, on a :

$$\phi_1 = jk_1(r_k + p_k) - ab_1(r_k + p_k).$$

Cas 6 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i > r_k + p_k$, on ne sait pas si les jobs de A peuvent être décalés vers la gauche dans O'. Il faut considérer que le job k dans O' commence à la date t, donc $\tau = 0$. On sait également que $p_j < p_k$ donc les jobs de B subissent un décalage maximum vers la droite de $p_k - p_j$, d'où $\gamma_B = p_k - p_j$. Enfin, on a $a > b$.
La fonction jk_1 est donc constante entre $r_k + p_k$ et b, décroissante entre b et a puis croissante entre a et $H - p_j$ (voir Figure 2.14). La fonction $-ab_1$ est croissante entre $r_k + p_k$ et $H - p_j$ (Figure 2.16). La fonction ϕ_1 est donc croissante sur les intervalles $[r_k + p_k, b]$ et $[a, H - p_j]$. Le minimum se trouve donc soit en $r_k + p_k$, soit dans l'intervalle $]b, a]$. Comme on sait que la fonction jk_1 est décroissante sur cet intervalle et que la fonction $-ab_1$ est croissante, on peut affirmer que la quantité $jk_1(a') - ab_1(b')$ est inférieure à toute valeur de $\phi_1(t)$ sur cet intervalle. Finalement, on a :

$$\phi_1 = \min\Big(jk_1(r_k + p_k) - ab_1(r_k + p_k), jk_1(a') - ab_1(b')\Big).$$

Cas 7 : Nous avons $r_j + p_j \leq r_k + p_k$, donc $\gamma_A = 0$. Par ailleurs, comme $\max_{i \in NS} r_i > r_k + p_k$, on ne sait pas si les jobs de A peuvent être décalés vers la gauche dans O'. Il faut considérer que le job k dans O' commence à la date t, donc $\tau = 0$. On sait également que $p_j \geq p_k$, donc $\gamma_B = 0$.
La fonction $-ab_1$ est donc nulle, puisque $\gamma_A = 0$ et $\gamma_B = 0$. La fonction ϕ_1 est alors égale à la fonction jk_1, qui atteint son minimum en $t = a'$ (voir Figure 2.13 et 2.14). Finalement, on a :

$$\phi_1 = jk_1(a').$$

Cas 8 : Nous avons $r_j + p_j > r_k + p_k$, donc les jobs de A sont décalés à droite entre O et O'. On a $\gamma_A = r_j + p_j - r_k - p_k$. Le décalage entre la date de début de j dans O et celle de k dans O' sera égal à γ_A, donc $\tau = r_j + p_j - r_k - p_k$. On sait aussi que la partie B commence à la date $t + p_j$ dans O et qu'elle ne commence pas après la date $t + \tau + p_k$ dans O'. Le décalage vers la droite n'excède pas $t + \tau + p_k - t - p_j = r_j - r_k$. Comme $p_j \leq p_k$, on sait que $r_j > r_k$, donc que $\gamma_B = r_j - r_k$.
La fonction jk_1 est minimale en $t = a'$ (voir Figure 2.13 et 2.14). Par ailleurs, comme $\gamma_A \leq \gamma_B$, la fonction $-ab_1$ est croissante entre $r_k + p_k$ et $H - p_j$ (voir Figure 2.16), donc minimale en $t = r_k + p_k$. On peut donc affirmer que la quantité $jk_1(a') - ab_1(r_k + p_k)$ est inférieure à toute valeur de $\phi_1(t)$ entre $r_k + p_k$ et $H - p_j$. On pose donc :

$$\phi_1 = jk_1(a') - ab_1(r_k + p_k).$$

Cas 9 : Nous avons $r_j + p_j > r_k + p_k$, donc les jobs de A sont décalés à droite entre O et O'. On a $\gamma_A = r_j + p_j - r_k - p_k$. Le décalage entre la date de début de j dans O et celle de k dans O' sera égal à γ_A, donc $\tau = r_j + p_j - r_k - p_k$. On sait aussi que la partie B commence à la date $t + p_j$ dans O et qu'elle ne commencera pas après la date $t + \tau + p_k$ dans O'. Le décalage vers la droite n'excédera pas $t + \tau + p_k - t - p_j = r_j - r_k$. Comme $r_j \leq r_k$, on sait que $\gamma_B = 0$.
La fonction jk_1 est minimale en $t = a'$ (voir Figure 2.13 et 2.14). Par ailleurs, comme $\gamma_A > \gamma_B$, la fonction $-ab_1$ est décroissante entre $r_k + p_k$ et $H - p_j$ (voir Figure 2.15), donc minimale en $t = H - p_j$. On peut donc affirmer que la quantité $jk_1(a') - ab_1(H - p_j)$ est inférieure à toute valeur de $\phi_1(t)$ entre $r_k + p_k$ et $H - p_j$. On pose donc :

$$\phi_1 = jk_1(a') - ab_1(H - p_j).$$

Cas 10 : Nous avons $r_j + p_j > r_k + p_k$, donc les jobs de A sont décalés à droite entre O et O'. On a $\gamma_A = r_j + p_j - r_k - p_k$. Le décalage entre la date de début de j dans O et celle de k dans O' est égal à γ_A, donc $\tau = r_j + p_j - r_k - p_k$. On sait aussi que la partie B commence à la date $t + p_j$ dans O et qu'elle ne commence pas après la date $t + \tau + p_k$ dans O'. Le décalage vers la droite n'excéde pas $t + \tau + p_k - t - p_j = r_j - r_k$. Comme $r_j > r_k$, on sait que $\gamma_B = r_j - r_k$.
La fonction jk_1 est minimale en $t = a'$ (voir Figure 2.13 et 2.14). Par ailleurs, comme $\gamma_A > \gamma_B$, la fonction $-ab_1$ est décroissante entre $r_k + p_k$ et $H - p_j$ (voir Figure 2.15), donc minimale en $t = H - p_j$. On peut donc affirmer que la quantité $jk_1(a') - ab_1(H - p_j)$ est inférieure à toute valeur de $\phi_1(t)$ entre $r_k + p_k$ et $H - p_j$. On pose donc :

$$\phi_1 = jk_1(a') - ab_1(H - p_j).$$

2.4.2.2 Jobs j et k sur deux machines différentes

Comme dans le cas où les jobs j et k sont ordonnancés sur la même machine, nous présentons ici un arbre sur la Figure 2.23 qui indique les six cas à différencier lorsque j et k ne sont pas ordonnancés sur la même machine. Ces cas sont maintenant présentés en détail. Cependant, la valeur de γ_A sera commune à tous les cas. En effet, on a $\gamma_A = \max(0, r_j + p_j - r_k - p_k)$.

Cas 1 : On a $p_j \geq p_k$ et $t_{min} \geq r_k$, donc les jobs de B ne sont pas décalés à droite dans O' par rapport à O, donc $\gamma_B = 0$. De plus, on a $\tau = 0$ puisque $t_{min} \geq r_k$.
Lorsque $\gamma_A > 0$, la fonction $-ab_2$ est constante, puisque la fonction b_2 est nulle et que la fonction a_2 ne dépend pas de t (voir Figure 2.20). En conséquence, le minimum de la fonction ϕ_2 sera celui de la fonction jk_2. Si $\gamma_A = 0$, la fonction $-ab_2$ est nulle et le minimum de la fonction ϕ_2 coïncide encore avec celui de la fonction jk_2. Or, le minimum de la fonction jk_2 est atteint lorsque $t = a'$. Finalement, on a :

$$\phi_2 = jk_2(a') - ab_2(a').$$

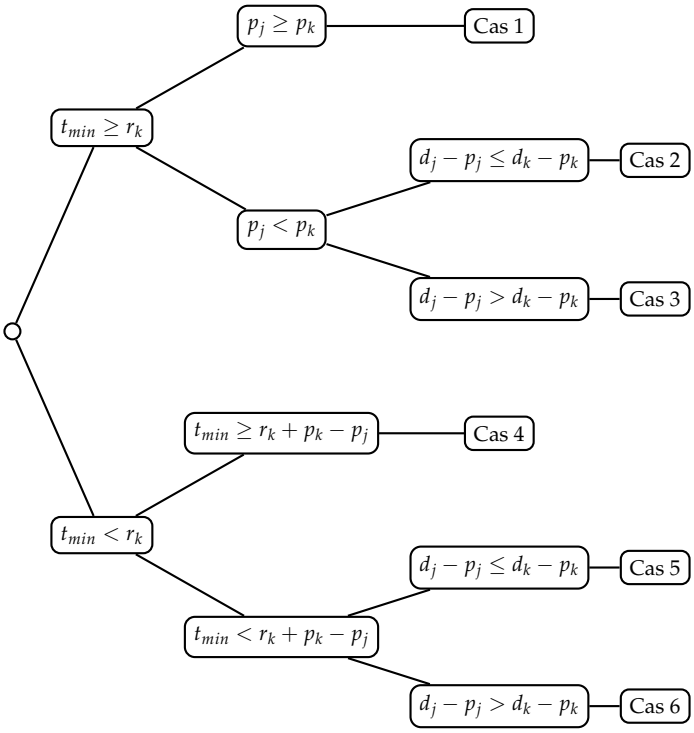


Figure 2.23 – Arbre de décision lorsque j et k sont ordonnancés sur deux machines différentes.

Cas 2 : On a $p_j < p_k$, donc les jobs de B subissent un décalage maximal vers la droite de $p_k - p_j$ unités de temps. Donc $\gamma_B = p_k - p_j$. Par ailleurs, comme $t_{min} \geq r_k$, on a $\tau = 0$. On sait de plus que $a \leq b$.

Comme $a \leq b$, la fonction jk_2 est croissante entre t_{min} et $H - p_j$. Par ailleurs, la fonction $-ab_2$ est également croissante sur cet intervalle. Finalement, le minimum est atteint lorsque $t = t_{min}$, donc :

$$\phi_2 = jk_2(t_{min}) - ab_2(t_{min}).$$

Cas 3 : On a $p_j < p_k$, donc les jobs de B subissent un décalage maximal vers la droite de $p_k - p_j$ unités de temps. Donc $\gamma_B = p_k - p_j$. Par ailleurs, comme $t_{min} \geq r_k$, on a $\tau = 0$. On sait de plus que $a > b$.

Comme $a > b$, la fonction jk_2 est constante entre t_{min} et b, décroissante entre b et a, puis croissante entre a et $H - p_j$ (voir Figure 2.18). Par ailleurs, la fonction $-ab_2$ est croissante sur l'intervalle $[t_{min}, H - p_j]$. Le minimum se trouve donc soit en t_{min}, soit dans l'intervalle $]b, a]$. Comme on sait que la fonction jk_2 est décroissante sur cet intervalle et que la fonction $-ab_2$ est croissante, on peut affirmer que la quantité $jk_2(a') - ab_2(b')$ est inférieure à toute valeur de $\phi_2(t)$ sur cet intervalle. Finalement, on a :

$$\phi_2 = \min\Big(jk_2(t_{min}) - ab_2(t_{min}), jk_2(a') - ab_2(b')\Big).$$

Cas 4 : On sait que $t_{min} \geq r_k + p_k - p_j$, donc que les jobs de B ne subissent pas de décalage vers la droite entre O et O'. Donc $\gamma_B = 0$. On a ici $t_{min} < r_k$, donc il se peut que le job k dans O' soit décalé par rapport au job j dans O. Nous avons $\tau = r_k - t_{min}$.

Lorsque $\gamma_A > 0$, la fonction $-ab_2$ est constante, puisque la fonction b_2 est nulle et que la fonction a_2 ne dépend pas de t (voir Figure 2.20). En conséquence, le minimum de la fonction ϕ_2 sera celui de la fonction jk_2. Si $\gamma_A = 0$, la fonction $-ab_2$ est nulle et le minimum de la fonction ϕ_2 coïncide encore avec celui de la fonction jk_2. Or, le minimum de la fonction jk_2 est atteint lorsque $t = a'$. Finalement, on a :

$$\phi_2 = jk_2(a') - ab_2(a').$$

Cas 5 : On sait que $t_{min} < r_k + p_k - p_j$, donc que les jobs de B subissent un décalage maximal vers la droite entre O et O' de $r_k + p_k - p_j - t_{min}$ unités de temps. Donc $\gamma_B = r_k + p_k - p_j - t_{min}$. On a ici $t_{min} < r_k$, donc il se peut que le job k dans O' soit décalé par rapport au job j dans O. Nous avons $\tau = r_k - t_{min}$. On sait également que $a \leq b$.

Comme $a \leq b$, la fonction jk_2 est croissante entre t_{min} et $H - p_j$. Par ailleurs, la fonction $-ab_2$ est également croissante sur cet intervalle. Finalement, le minimum est atteint lorsque $t = t_{min}$:

$$\phi_2 = jk_2(t_{min}) - ab_2(t_{min}).$$

Cas 6 : On sait que $t_{min} < r_k + p_k - p_j$, donc que les jobs de B subissent un décalage maximal vers la droite entre O et O' de $r_k + p_k - p_j - t_{min}$ unités de temps. Donc $\gamma_B = r_k + p_k - p_j - t_{min}$. On a ici $t_{min} < r_k$, donc il se peut que le job k dans O' soit décalé par rapport au job j dans O. Nous avons $\tau = r_k - t_{min}$. On sait également que $a > b$.

Comme $a > b$, la fonction jk_2 est constante entre t_{min} et b, décroissante entre b et a, puis croissante entre a et $H - p_j$ (voir Figure 2.18). Par ailleurs, la fonction $-ab_2$ est croissante sur l'intervalle $[t_{min}, H - p_j]$. Le minimum se trouve donc soit en t_{min}, soit dans l'intervalle $]b, a]$. Comme on sait que la fonction jk_2 est décroissante sur cet intervalle et que la fonction $-ab_2$ est elle croissante, on peut affirmer que la quantité $jk_2(a') - ab_2(b')$ est inférieure à toute valeur de $\phi_2(t)$ sur cet intervalle. Finalement, on a :

$$\phi_2 = \min\Big(jk_2(t_{min}) - ab_2(t_{min}), jk_2(a') - ab_2(b')\Big).$$

2.4.3 Règle de dominance

Nous sommes désormais en mesure d'énoncer la règle de dominance portant les jobs non ordonnancés.

Proposition 2.8. *Soient P un ordonnancement partiel sur m machines et NS un ensemble de jobs non ordonnancés. Soient j et k deux jobs appartenant à l'ensemble NS. Soit $\phi = \min(\phi_1, \phi_2)$. Le job j domine le job k par rapport à P si $\phi > 0$.*

Démonstration. Soit O un ordonnancement commençant par l'ordonnancement partiel P et où le job k est ordonnancé le plus tôt possible après P. Soit O' l'ordonnancement obtenu en permutant les positions des jobs j et k dans O. Comme $\phi > 0$, on sait que $\phi_1 > 0$ et $\phi_2 > 0$, cela signifie que $F(O) - F(O') > 0$, puisque ϕ_1 en est une borne inférieure quand j et k sont sur la même machine et ϕ_2 quand j et k sont sur deux machines différentes. Par conséquent, l'ordonnancement O coûte plus cher que l'ordonnancement O'. L'ordonnancement O ne peut donc pas être optimal, c'est pourquoi le job j domine le job k par rapport à l'ordonnancement partiel P. □

2.5 Discussion autour des règles de dominance proposées par Yalaoui et Chu

Traitant le problème $Pm|r_i|\sum C_i$ par un *Branch&Bound*, Yalaoui et Chu ont également proposé plusieurs règles de dominance. Nous énonçons tout d'abord ces résultats. Nous comparons ensuite avec les règles présentées dans ce chapitre.

Avant de rappeler les règles de dominance de Yalaoui et Chu, nous introduisons plusieurs notations dans le Tableau 2.1. Les démonstrations ne figurent pas dans cette partie.

notation	définition
$R_i(t)$	date de début d'exécution au plus tôt du job i à partir de la date t
$E_i(t)$	date de fin au plus tôt du job i s'il est ordonnancé à partir de la date t
$J(P)$	ensemble des jobs de l'ordonnancement partiel P
$J_j(P)$	ensemble des jobs ordonnancés sur la machine j dans P
$C_i(P)$	date de fin du job i dans P
$\varphi_j(P)$	date de fin du dernier job ordonnancé sur la machine j
$\mu(P)$	indice de la machine disponible au plus tôt après P
$(P\|i)$	ordonnancement partiel formé où le job i est ordonnancé au plus tôt à la suite de P
$\Omega(P,i)$	ordonnancement de plus faible coût parmi les ordonnancements commençant par $(P\|i)$

Tableau 2.1 – Notations

La plupart des règles de dominance utilisent les deux observations suivantes. On suppose qu'on dispose de n jobs à ordonnancer sur m machines.

Observation 2.1. *[68] Tout ordonnancement partiel P avec strictement plus de $n-m+1$ jobs sur une machine est dominé.*

Cette observation repose sur le fait qu'il existe toujours un ordonnancement optimal où chaque machine exécute au moins un job. À partir de ce constat, on peut dresser la seconde observation suivante.

Observation 2.2. *[68] Étant donné un ordonnancement partiel P, le nombre de jobs ajoutés sur la machine i dans tout ordonnancement $\Omega(P,i)$ est inférieur à $U_i(P) = \min(n-m+1-card(J_i(P)), n-card(J(P)))$. On note $U(P) = \max_{i\in[1,m]} U_i(P)$.*

Ces deux observations permettent de majorer le coût induit par des décalages à droite. Pour la règle de dominance portant sur les jobs non ordonnancés, nous utilisons deux variables γ_A et γ_B (voir Section 2.4) qui jouent le même rôle. Ces dernières s'expriment sous forme d'une somme de poids, qui

correspond pour $Pm|r_i|\sum C_i$ à un nombre de jobs. Néanmoins, l'évaluation par excès n'est pas réalisée de la même manière. Nous utilisons une borne supérieure du temps de fin d'exécution des machines pour borner cette somme de poids, tandis que Yalaoui et Chu utilisent les observations présentées précédemment pour borner le nombre de jobs.

Proprieté 2.1. *[68] Soient P un ordonnancement partiel et i un job non ordonnancé dans P. S'il existe un job j ordonnancé dans P sur la machine μ, tel que $E_i(\Delta_j) \leq E_j(\Delta_j)$ et $E_i(\Delta_j) - E_j(\Delta_j) \leq (p_i - p_j)U_\mu(P)$, alors tout ordonnancement $\Omega(P,i)$ est dominé.*

La règle de dominance 2.1 ne porte pas exactement sur les jobs non ordonnancés, mais on peut néanmoins la considérer en tant que telle. Il suffit de supposer que les jobs situés sur la machine μ après le job j ne sont pas ordonnancés. La situation est alors celle où les jobs de la séquence A ne sont pas retardés, mais où les jobs de B peuvent l'être. Cette proprieté correspond aux cas 1, 3, 4, 5 et 7 quand i et j sont sur la même machine. Le fait de décomposer en plusieurs cas nous permet d'imposer des conditions moins fortes que celle de cette proprieté.

Proprieté 2.2. *[68] Soient P un ordonnancement partiel et i un job non ordonnancé dans P. S'il existe un job j ordonnancé dans P sur une machine $g \neq \mu$, tel que $\Delta_j \leq \varphi_\mu$ et $R_i(\varphi_\mu) < r_j$, alors tout ordonnancement $\Omega(P,i)$ est dominé.*

La proprieté 2.2 permet d'éviter les temps d'inactivité qui peuvent être réductibles à coup sûr. En effet, elle indique qu'en présence d'un job j ordonnancé après la date φ_μ alors que la machine était disponible avant et d'un job i qui peut être ordonnancé plus tôt à la place de j, l'ordonnancement est dominé. Il est clair en effet qu'on pourrait alors permuter les jobs i et j ainsi que leurs successeurs pour former un meilleur ordonnancement.

Proprieté 2.3. *[68] Soient P un ordonnancement partiel et i un job non ordonnancé dans P. S'il existe un autre job non ordonnancé j tel que $E_i(\varphi_\mu) \geq E_j(\varphi_\mu)$ et $E_i(\varphi_\mu) - E_j(\varphi_\mu) \geq (p_i - p_j)(U(P) - 1)$, alors tout ordonnancement $\Omega(P,i)$ est dominé.*

La proprieté 2.3 est similaire à la proprieté 2.1, sauf que les jobs sont ici considérés comme non ordonnancés. Dans cette situation, les jobs i et j peuvent ne pas ordonnancés sur la même machine. Dans ce cas, cette règle couvre l'ensemble des cas où $\gamma_A = 0$ (voir Section 2.4).

Proprieté 2.4. *[68] Soient P un ordonnancement partiel et i un job non ordonnancé dans P. S'il existe un autre job non ordonnancé j tel que*

$$\left(E_j(\varphi_\mu) - R_i(\varphi_\mu)\right)\left(U_\mu(P) - 1\right) \leq \min\left(R_j(\psi) - R_j(\phi_\mu), \min_{k \notin J(P)}(p_k)\right),$$

où ψ représente la seconde plus petite date de fin des machines de P, alors tout ordonnancement $\Omega(P,i)$ est dominé.

La proprieté 2.4 établit des conditions suffisantes pour que l'insertion future d'un job j non ordonnancé entre l'ordonnancement partiel P et le job i soit nécessairement améliorante. Elle n'est pas comparable avec les règles de dominance que nous proposons, qui s'appuient sur des échanges de jobs uniquement.

Proprieté 2.5. *[68] Soient P un ordonnancement partiel et i un job non ordonnancé dans P. S'il existe un autre job non ordonnancé j tel que $E_i(\varphi_\mu) \leq E_j(\varphi_\mu)$ et $(p_i - p_j) \leq \left(E_i(\varphi_\mu) - E_j(\varphi_\mu)\right)(U_\mu(P) - 1)$, alors tout ordonnancement $\Omega(P, i)$ est dominé.*

La proprieté 2.5 est complémentaire de la proprieté 2.3. Elle traite les cas où l'échange du job i par le job j retarde les jobs situés après le job i. Si i et j sont ordonnancés sur la même machine, cette proprieté englobe les cas 8, 9 et 10. Quand i et j sont sur deux machines distinctes, elle traite les cas où $\gamma_A > 0$ (voir Section 2.4).

Proprieté 2.6. *[68] Soient P un ordonnancement partiel et i un job non ordonnancé dans P. S'il existe un autre ordonnancement P' tel que*

$$\begin{aligned} J(K) &= J(K) \cup \{i\} \text{ et} \\ F(P') &\leq F(P|i) \text{ et} \\ (n - card(J(P))) \times \delta &\leq F(P|i) - F(P'), \end{aligned}$$

où $\delta = \max_{k\in[1,m]}(\varphi'_k - \varphi_k)$ avec φ'_k et φ_k sont respectivement les plus k^e petites dates de fin des ordonnancements partiel $(P|i)$ et P', alors tout ordonnancement $\Omega(P, i)$ est dominé.

Enfin, la proprieté 2.6 permet de comparer deux ordonnancements partiels composés des mêmes jobs. Elle est équivalente à la règle RD_m, appliquée au critère $\sum C_i$, que nous avons présentée dans ce chapitre.

Bornes inférieures

Sommaire

Ce chapitre est dédié à l'étude de bornes inférieures pour les quatre problèmes d'ordonnancement qui nous intéressent ici. L'utilisation de bornes inférieures de bonne qualité est cruciale dans une méthode arborescente de type *Branch&Bound*. Elles permettent un élagage efficace dans

l'arbre de recherche. Les grandes techniques utilisées dans ce chapitre ont été présentées plus en détail dans la Section 1.2. Après un bref état de l'art dans la Section 3.1, nous présentons deux bornes inférieures obtenues en relâchant les dates de disponibilité (Section 3.2). La Section 3.3 présente une borne inférieure obtenue en modélisant une relaxation de notre problème sous forme de flot maximal à coût minimal. Plusieurs bornes inférieures, obtenues en calculant des dates de fin minimales des jobs terminant en i^{e} position, sont proposées dans la Section 3.4. Nous présentons enfin plusieurs bornes inférieures dérivées d'une formulation indexée sur le temps dans la Section 3.5. Finalement, une série de résultats numériques, nous permettant de comparer toutes ces bornes inférieures, est fournie dans la Section 3.6.

Tout au long de ce chapitre, des exemples permettent d'illustrer les différentes techniques présentées. L'ensemble des exemples est construit à partir d'une même instance de cinq jobs à exécuter sur deux machines. Les caractéristiques des jobs sont présentées dans le Tableau 3.1.

job i	1	2	3	4	5
r_i	0	1	3	4	5
p_i	6	4	3	1	4
d_i	7	5	8	7	12
w_i	1	4	1	2	3

Tableau 3.1 – Instance utilisée pour les exemples.

3.1 État de l'art

Le problème $Pm|r_i|\sum C_i$ a été étudié par Yalaoui et Chu dans [68]. Les auteurs proposent deux bornes inférieures. Nous décrirons en détail ces deux bornes dans les Sections 3.2 et 3.4.3. La première consiste à relâcher toutes les dates de disponibilité à $r_{min} = \min_{i\in[1,n]} r_i$, puis à résoudre le problème $Pm||\sum C_i$, qui est polynomial. La seconde borne inférieure s'appuie sur la technique de *Job Splitting* : la préemption, ainsi que l'exécution simultanée de plusieurs parties d'un même job sont autorisées. Le problème ainsi relâché peut être résolu exactement en temps polynomial. La valeur obtenue est une borne inférieure valide du problème initial. Nessah *et al.* ont également proposé une borne inférieure pour le problème $Pm|r_i|\sum C_i$ [54]. Ils s'appuient pour cela sur une amélioration présentée par Della Croce et T'kindt pour le problème à une machine [31]. Webster a traité le problème sans dates de disponibilité et avec poids $Pm||\sum w_i C_i$, pour lequel il a proposé deux bornes inférieures [66]. Le problème d'avance-retard, noté $Pm|r_i|\sum \alpha_i E_i + \beta_i T_i$ a également été traité par Kedad-Sidhoum *et al.* dans [45]. Les auteurs y proposent une formulation indexée sur le temps qu'ils relaxent de différentes manières afin d'obtenir des bornes inférieures du problème. Le cas particulier où $m = 1$ (problème à une machine) a aussi fait l'objet de nombreuses études. Dans

[42, 43], Jouglet *et al.* proposent des bornes inférieures pour le retard total et pour le retard total pondéré. Enfin, Rivreau a présenté une borne inférieure, basée sur une relaxation lagrangienne d'une formulation indexée sur le temps, pour un ensemble de problèmes à une machine $1|r_i|F_i$ où la fonction de coût respecte une certaine contrainte [59].

3.2 Relaxation des dates de disponibilité

Les deux bornes inférieures que nous présentons ici sont dédiées au problème de minimisation de la somme des dates de fin. Cependant, on peut en déduire des bornes inférieures pour le retard total (voir Section 3.2.3).

3.2.1 Relaxer toutes les dates de disponibilité à la plus petite

Cette borne inférieure a été présentée par Yalaoui et Chu dans [68] pour le critère $\sum C_i$. Toutes les dates de disponibilité sont relâchées à la plus petite d'entre elles. Le problème se ramène à $Pm||\sum C_j$ et peut être résolu en temps polynomial [40]. Tous les jobs sont ordonnancés selon la règle *Shortest Processing Time* (le plus court d'abord) sur la machine disponible au plus tôt. Cette borne inférieure peut être calculée en $O(n \log n)$. Nous notons cette borne inférieure $lb_{no-release}$.

Sur l'instance exemple, la solution optimale pour la somme des dates de fin est représentée sur la Figure 3.1. Toutes les dates de disponibilité valent 0 et la valeur de la borne est $lb_{no-release} = 27$.

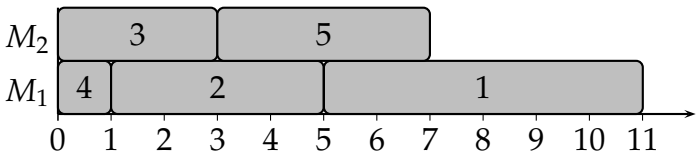


Figure 3.1 – Solution optimale obtenue lorsque les dates de disponibilité sont relâchées à la plus petite d'entre elles.

3.2.2 Décomposition en deux sous-problèmes

Dans cette section, nous présentons une borne inférieure pour laquelle les jobs sont classés en sous-ensembles en fonction de leurs dates de disponibilité. L'idée consiste à séparer les jobs en plusieurs sous-ensembles en diminuant les dates de disponibilité. Au sein d'un sous-ensemble, les jobs ont la même date de disponibilité. Ensuite, chaque sous-ensemble est ordonnancé optimalement avec la règle *Shortest Processing Time*. La somme des

coûts de chacun des sous-ensembles est une borne inférieure valide du problème initial. Dans la suite, nous présenterons uniquement le cas à deux sous-ensembles. Cependant, rien n'empêcherait d'utiliser un nombre plus important de sous-ensembles.

Soit t une date. On définit les deux sous-ensembles N_0 et N_t par $N_0 = \{i \in N / r_i < t\}$ et $N_t = \{i \in N / r_i \geq t\}$. Nous avons désormais deux problèmes à résoudre :

1. Ordonnancer de manière optimale les jobs de N_0 à partir de la date 0 ;
2. Ordonnancer de manière optimale les jobs de N_t à partir de la date t.

Ces deux sous-problèmes peuvent être résolus en temps polynomial, et la somme des deux valeurs optimales est notée $\Gamma(t)$. Il est possible de raffiner ce résultat, en constatant qu'on peut ordonnancer les jobs de N_0 à partir du profil formé des m plus petites dates de disponibilité. En effet, dans tout ordonnancement réalisable, aucun job ne peut commencer avant les m plus petites de disponibilité. La valeur $\Gamma(t)$ est une borne inférieure du problème initial. Par ailleurs, on peut remarquer que n'importe quelle valeur de t mène à une borne inférieure valide. Nous avons donc intérêt à choisir la valeur de t qui maximise $\Gamma(t)$. Il faut calculer $\Gamma(t)$ pour $t \in [0, \max_i(r_i)]$. Dans les faits, il est suffisant de tester uniquement les dates qui correspondent à des dates de disponibilité pour atteindre le maximum de la fonction Γ. Nous noterons cette borne inférieure $lb_{no-release-subsets}$.

Cette borne peut être calculée en $O(n^2)$ dans la mesure où n valeurs de t (au maximum) sont testées. Une fois que les jobs ont été triés selon les durées croissantes ($O(n \log n)$), chacune de ces bornes peut être calculée en $O(n)$. En effet, chaque problème est résolu de manière optimale en ordonnançant les jobs selon les durées croissantes.

L'Algorithme 3.1 détaille le calcul de la borne inférieure $lb_{no-release-subsets}$. La première étape consiste à trier les dates de disponibilité et à placer les m plus petites dans un tableau. Ces dernières servent de base pour ordonnancer la partie « de gauche ». Ensuite, pour chaque valeur de i, on prend la valeur r_i comme « pivot », *i.e.* les jobs qui ont une date de disponibilité inférieure strictement à r_i seront ordonnancés « à gauche ». On parcourt alors l'ensemble des jobs selon les durées croissantes, et on les ordonnance itérativement d'un côté ou de l'autre. La valeur obtenue est alors une borne inférieure valide, et on retient la plus grande, qui sera la valeur finale de cette borne inférieure. Cette borne se calcule en $O(n^2)$, puisque l'on effectue n itérations qui coûte $O(n)$ et deux tris en $O(n \log(n))$.

Sur le Tableau 3.2, on peut voir les valeurs de la fonction Γ sur l'instance d'exemple. La valeur maximale $\Gamma(t)$ est atteinte lorsque $t = 4$, on a alors $lb_{no-release-subsets} = 29$. Les deux ordonnancements des sous-problèmes sont représentés sur la Figure 3.2.

3.2.3 Extension au retard total

Les bornes inférieures que nous venons de voir sont valides pour la somme des dates de fin. Cependant, on peut les généraliser au retard total en sous-

Algorithme 3.1 : Calcul de la borne inférieure $lb_{no-release-subsets}$

Données : n jobs indicés tels que $p_1 \leq p_2 \leq \dots p_n$

Résultat : La valeur de la borne inférieure LB

$\Gamma \leftarrow 0$;
Trier les dates de disponibilité et placer les m plus petites dans un tableau tab;
pour $i = 1$ *à* n **faire**
  $\Gamma(t) \leftarrow 0$;
  $t \leftarrow r_i$;
  pour $j = 1$ *à* m **faire**
    $gauche[j] \leftarrow tab[j]$;
    $droite[j] \leftarrow t$;
  pour $k = 1$ *à* n **faire**
    si $r_k < t$ **alors**
      $u \leftarrow 1$;
      pour $l = 2$ *à* m **faire**
        si $gauche[l] < gauche[u]$ **alors** $u \leftarrow l$;
      $gauche[u] \leftarrow gauche[u] + p_k$;
      $\Gamma(t) \leftarrow \Gamma(t) + gauche[u]$;
    sinon
      $u \leftarrow 1$;
      pour $l = 2$ *à* m **faire**
        si $droite[l] < droite[u]$ **alors** $u \leftarrow l$;
      $droite[u] \leftarrow droite[u] + p_k$;
      $\Gamma(t) \leftarrow \Gamma(t) + droite[u]$;
  si $\Gamma(t) > \Gamma$ **alors** $\Gamma \leftarrow \Gamma(t)$;
retourner Γ

t	0	1	3	4	5
$\Gamma(t)$	27	26	28	**29**	27

Tableau 3.2 – Valeurs de Γ.

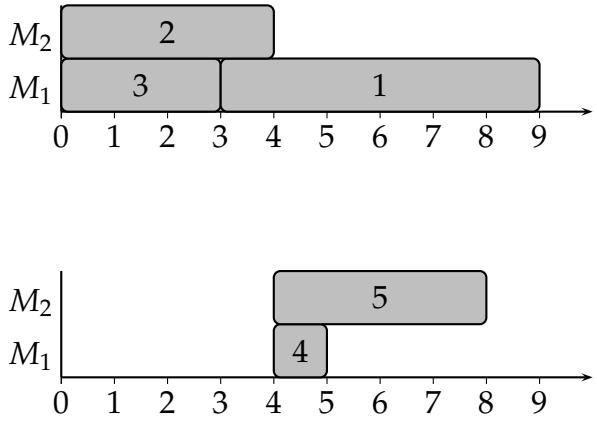


Figure 3.2 – Meilleure solution atteinte pour $t = 4$.

trayant la quantité $\sum_{i=1}^{n} d_i$ à la valeur obtenue pour la somme des dates de fin [2]. En effet, pour tout ordonnancement (donc également s'il est optimal), on a $\sum_{i=1}^{n} \max(0, C_i - d_i) \geq \max(0, \sum_{i=1}^{n} C_i - d_i)$. Par exemple, si la borne inférieure pour la somme des dates de fin vaut V sur notre exemple, on sait que $V' = \max(0, V - \sum_{i=1}^{n} d_i) = \max(0, V - 39)$ est une borne inférieure valide pour le retard total.

3.3 Flot

Cette borne inférieure est valide pour les quatre critères étudiés ($\sum C_i$, $\sum w_i C_i$, $\sum T_i$ et $w_i T_i$). Elle s'appuie sur une relaxation du problème initial qui peut être résolue grâce à une modélisation en problème de flot maximal à coût minimal. Dans la Section 3.3.1, nous présentons cette borne inférieure dans le cas du retard total pondéré qui est le critère le plus général. Dans la Section 3.3.1, une amélioration dans le cas de la somme des dates de fin (pondérée ou non) est présentée.

3.3.1 Le retard total (pondéré)

Notons π notre problème initial, à savoir $Pm|r_i|w_i T_i$. Nous allons construire un problème noté π' de la façon suivante : chaque job i du problème π est découpé (voir par exemple [13]) en p_i morceaux J_{ik}, $k \in 1, \ldots, p_i$ de durée $p_{ik} = 1$. On associe à chaque morceau J_{ik} un poids $\frac{w_i}{p_i}$ et une date échue d_i. Ce découpage correspond à la notion de *splitting* introduite par Belouadah, Posner et Potts [13].

Proposition 3.1. *La solution optimale du problème π' est inférieure à celle du problème π.*

Démonstration. Soit S un ordonnancement optimal du problème initial π. On construit la solution correspondante S' pour le problème π' en remplaçant chaque job i du problème π par les morceaux J_{ik}, $k \in 1, \ldots, p_i$ dans le problème π'. L'ordonnancement S coûte $WT(S) = \sum_{i=1}^{n} w_i \max\left(0, C_i(S) - d_i\right)$, où $C_i(S)$ représente la date de fin du job i dans l'ordonnancement S. L'ordonnancement S' coûte donc $WT(S') = \sum_{i=1}^{n} w_i/p_i \sum_{k=1}^{p_i} \max\left(0, C_i(S) - k + 1 - d_i\right)$. Par ailleurs, on a $\max\left(0, C_i(S) - k + 1 - d_i\right) \leq \max\left(0, C_i(S) - d_i\right)$ pour $k \geq 1$. Donc $WT(S') \leq \sum_{i=1}^{n} w_i/p_i \sum_{k=1}^{p_i} \max\left(0, C_i(S) - d_i\right) = WT(S)$. En conclusion, la valeur d'une solution optimale du problème π' est inférieure à celle d'une solution optimale du problème π. □

Une valeur optimale ou une borne inférieure du problème π' peut être calculée en considérant le problème de flot suivant. On considère un graphe orienté et connexe $G(X, U)$, où X est un ensemble de sommets et U un ensemble d'arcs. L'ensemble X est composé d'une source S, d'un puits T, de n sommets associés aux n jobs de N et de u sommets associés à des intervalles de temps consécutifs $I_1, I_2, \ldots, I_u$ qui forment une partition de $[r_{min}, H]$ où H est une borne supérieure de la date de fin de projet d'un ordonnancement préemptif. Les intervalles de temps sont obtenus à partir de l'ensemble de dates $t_1 = r_{min}, t_2, \ldots, t_{u+1} = H$, en posant $I_k = [t_k, t_{k+1}[$ (voir Figure 3.3). On note l_k la longueur de l'intervalle I_k, *i.e.* $l_k = t_{k+1} - t_k$.

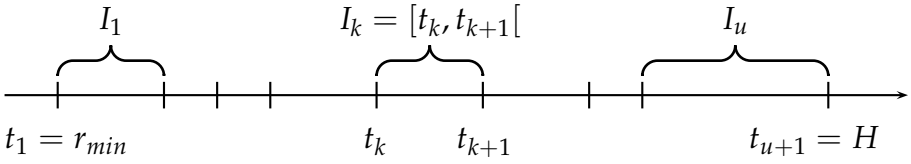


Figure 3.3 – Intervalles I_1, I_2, …, I_u.

S est relié à chaque sommet i par un arc de capacité p_i de coût 0. Chaque sommet I_k est relié au puits T par un arc de capacité ml_k et de coût 0. Enfin, pour chaque job i et chaque intervalle I_k tels que $r_i \geq t_k$, le sommet i est relié au sommet I_k par un arc de capacité l_k et de coût c_{ik} (voir Figure 3.4). Ce réseau est représenté sur la Figure 3.3. Chaque arc est étiqueté par une paire (x, y) où x représente la capacité de l'arc et y le coût unitaire associé à cet arc.

Il est aisé de constater que le flot maximal est égal à $\sum p_i$. On peut établir le résultat suivant :

Proposition 3.2. *Si* $\forall k$, $l_k = 1$ *et pour chaque coût on a* $c_{ik} = \frac{w_i}{p_i} \max(0, t_k + 1 - d_i)$, *alors le problème de flot maximal à coût minimal sur ce réseau est équivalent au problème* π'. *La valeur obtenue est donc une borne inférieure du problème initial* π.

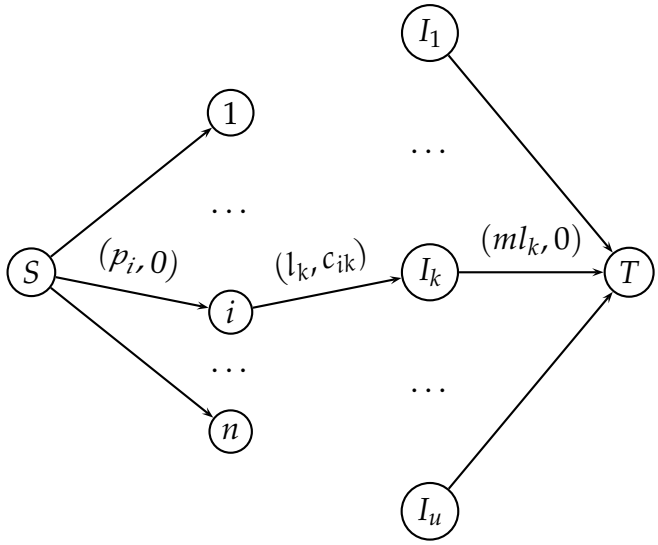


Figure 3.4 – Modélisation sous forme de flot.

On peut néanmoins remarquer que le nombre d'intervalles $P = H - r_{min}$ peut être très important. Afin d'obtenir une borne inférieure calculable en temps polynomial, on peut définir ces intervalles à partir des dates de disponibilité et des dates échues $\{r_i\} \cup \{d_i\}$. Les valeurs c_{ik} sont choisies de sorte que le coût du flot optimal soit toujours plus faible que le coût de l'ordonnancement optimal du problème π'. Plus précisément, on constate que $c_{ik} = \frac{w_i}{p_i} \max(0, t_k + 1 - d_i)$ respecte cette condition dans la mesure où chaque morceau du job i placé sur l'intervalle k a une date de fin au moins égale à $t_k + 1$. Comme le nombre de sommets est en $O(n)$ et que le nombre d'arcs est en $O(n^2)$, cette borne inférieure peut être calculée en $O(n^3 \log^2(n))$ [56]. Nous appellerons désormais cette borne lb_{flow}.

La discrétisation du temps décrite ci-dessus peut mener à des résultats de faible qualité dans certaines situations. Il est facile de construire une instance pour laquelle cette discrétisation comporte des intervalles très larges. Afin de limiter cette faiblesse, on peut décider d'ajouter un certain nombre de dates de coupure (n par exemple). Pour insérer une telle date, on choisit l'intervalle de plus grande largeur et on le « coupe » en deux parties égales. On réitère cette procédure autant de fois qu'il y a de dates de coupure à insérer. Comme le nombre d'intervalles reste linéaire si le nombre de coupures est lui-même linéaire, cette borne inférieure, notée $lb_{flow'}$, a aussi une complexité de $O(n^3 \log^2(n))$.

L'Algorithme 3.2 permet d'améliorer la discrétisation temporelle utilisée dans la borne inférieure lb_{flow}. Les dates de coupure initiales sont placées dans le tableau $coup$. Pour introduire une nouvelle date de coupure, on parcourt le tableau $coup$ et on retient l'intervalle entre deux coupures de plus grande largeur. Ce dernier est alors coupé en deux parties égales par l'insertion d'une nouvelle date de coupure d. Ce procédé est répété n fois. Cette amélioration s'effectue en $O(n^2)$.

Algorithme 3.2 : Amélioration polynomiale de la discrétisation temporelle pour la borne de flot $lb_{flow'}$

Données : Un tableau de date de coupure $coup$ trié de manière croissante

Résultat : Le tableau $coup$ auquel on a ajouté de n nouvelles dates de coupure

pour $k = 1$ *à* n **faire**
- $i \leftarrow 1$;
- $i_{max} \leftarrow 0$;
- $l_{max} \leftarrow 0$;
- **tant que** $coup[i+1]$ *existe* **faire**
  - **si** $coup[i+1] - coup[i] > l_{max}$ **alors**
    - $i_{max} \leftarrow i$;
    - $l_{max} \leftarrow coup[i+1] - coup[i]$;
- $d \leftarrow \frac{coup[i_{max}] + coup[i_{max}+1]}{2}$;
- Insérer d dans le tableau $coup$ entre $coup[i_{max}]$ et $coup[i_{max}+1]$;

retourner $coup$

Enfin, il peut être intéressant de borner la longueur des intervalles, même si on perd alors la linéarité du nombre d'intervalles. Par exemple, on peut partir de la discrétisation initiale $(\{r_i\} \cup \{d_i\})$ et partager en deux parties égales tout intervalle de longueur supérieure à un paramètre τ, tant qu'il en existe. Cette borne est pseudo-polynomiale. Elle est notée $lb_{flow''}(\tau)$.

L'Algorithme 3.3 décrit cette amélioration de la discrétisation temporelle. Elle dépend d'un paramètre τ, qui indique la largeur maximale autorisée pour la discrétisation temporelle. Cet algorithme fonctionne de la manière suivante : tant qu'il existe un intervalle de largeur supérieure à τ, on introduit une nouvelle date de coupure en son milieu. Cette procédure est pseudo-polynomiale, puisque le nombre d'itérations est fonction de l'horizon, qui est lui-même fonction des données.

3.3.2 La somme pondérée des dates de fin

Nous proposons ici une amélioration de la valeur obtenue dans le cas de la minimisation de la somme des dates de fin (pondérée ou non). Cette dernière s'appuie sur une idée présentée par Belouadah, Posner et Potts [13] pour le problème $1|r_i|\sum w_i C_i$. L'idée consiste à ajouter une certaine quantité qui peut être vue comme le coût engendré par la séparation des jobs en morceaux de longueur unitaire dans π'. Nous énonçons maintenant une proposition, qui peut être vue comme un cas particulier des résultats de [13] :

Algorithme 3.3 : Amélioration pseudo-polynomiale de la discrétisation temporelle pour la borne de flot $lb_{flow''}(\tau)$

Données : Un tableau de date de coupure $coup$ trié de manière croissante et un entier τ

Résultat : Le tableau $coup$ auquel on a éventuellement ajouté de nouvelles dates de coupure

$i \leftarrow 1$;
tant que $coup[i+1]$ *existe* **faire**
  si $coup[i+1] - coup[i] > \tau$ **alors**
    $d \leftarrow \dfrac{coup[i] + coup[i+1]}{2}$;
    Insérer d dans le tableau $coup$ entre $coup[i]$ et $coup[i+1]$;
  sinon
    $i \leftarrow i+1$;
retourner $coup$

Proposition 3.3. *La valeur de la solution optimale du problème π', à laquelle on ajoute la quantité $\sum_{i=1}^{n} w_i(p_i - 1)/2$, est inférieure ou égale à la valeur d'une solution optimale du problème π .*

Démonstration. Soit S un ordonnancement optimal du problème initial π. On construit la solution S' correspondante au problème π', dans laquelle chaque job i est remplacé par les morceaux J_{ik} lui correspondant dans le problème π'. Le coût de l'ordonnancement S est $WC(S) = \sum_{i=1}^{n} w_i C_i(S)$ où $C_i(S)$ est la date de fin du job i dans l'ordonnanceent S. Le coût de l'ordonnancement S' est alors $WC(S') = \sum_{i=1}^{n} w_i/p_i \sum_{k=1}^{p_i} C_i(S) - k + 1 = \sum_{i=1}^{n} w_i C_i(S) + \sum_{i=1}^{n} w_i/p_i \sum_{k=1}^{p_i} 1 - k = WC(S) - \sum_{i=1}^{n} w_i(p_i - 1)/2$. On peut donc conclure que la valeur d'une solution optimale du problème π', à laquelle on ajoute la quantité $\sum_{i=1}^{n} w_i(p_i - 1)/2$, est toujours inférieure à la valeur d'une solution optimale du problème π. □

Sur l'instance d'exemple, le flot optimal entre les sommets N et les sommets $I_i, i \in \{1, \ldots, u\}$ est représenté dans le Tableau 3.3. Le coût du flot optimal est de 23,333.... Après ajout de la quantité $\sum_{i=1}^{n} w_i(p_i - 1)/2$, ici égale à 14, on obtient une borne inférieure égale à 38.

job i	I_1	I_2	I_3	I_4	I_5
1	1	2	0	0	3
2	0	2	1	0	1
3	0	0	1	1	1
4	0	0	0	1	0
5	0	0	0	0	4

Tableau 3.3 – Flot optimal sur l'instance d'exemple.

3.4 Calculs des dates de fin minimales

On note $\underline{C}[i]$, $i \in N$ la date de fin minimale du job qui termine en i^e position, parmi l'ensemble des ordonnancements réalisables. À partir de cette notion, nous proposons plusieurs bornes inférieures. Dans la Section 3.4.1, nous montrons comment des bornes inférieures peuvent être calculées à partir des valeurs $\underline{C}[i]$ pour chaque critère. Dans la Section 3.4.2, nous présentons quelques manières simples de calculer des valeurs de $\underline{C}$. Dans la Section 3.4.3, une borne inférieure basée sur le *Job Splitting* est présentée [68]. Nous proposons ensuite plusieurs bornes inférieures utilisant un résultat de Horn [41] dans la Section 3.4.4. Enfin, une borne inférieure utilisant les bornes inférieures polynomiales est présentée dans la Section 3.4.5.

3.4.1 Comment obtenir des bornes inférieures

On note $\lambda[i]$, $i \in N$ toute valeur qui est inférieure à $\underline{C}[i]$. Soit S un ordonnancement optimal d'une instance du problème considéré, $[i](S)$ le job qui termine en i^e position dans S et $C_{[i]}(S)$ sa date de fin. On peut déjà remarquer qu'on a, par définition, $\forall i \in N, \lambda[i] \leq \underline{C}[i] \leq C_{[i]}(S)$.

Nous allons désormais détailler la façon de calculer des bornes inférieures à partir de valeurs λ, et ce, critère par critère.

3.4.1.1 $Pm|r_i|\sum C_i$

On a $\forall S, \sum \lambda[i] \leq \sum \underline{C}[i] \leq \sum C_{[i]}(S)$. Ainsi, $\sum \lambda[i]$ est une borne inférieure du problème $Pm|r_i|\sum C_i$.

3.4.1.2 $Pm|r_i|\sum w_i C_i$

Soient $w'_1 \leq w'_2 \leq \ldots \leq w'_n$ les poids des jobs, triés par ordre croissant. En s'appuyant sur le fait que $w_1C_1 + w_2C_2 \geq w_2C_1 + w_1C_2$ si $w_1 \leq w_2$ et $C_1 \leq C_2$, on peut montrer que $\sum w'_i C_{[i]}(S) \leq \sum w_{[i](S)} C_{[i]}(S)$. Finalement, on a $\sum w'_i \lambda[i] \leq \sum w'_i C_{[i]}(S) \leq \sum w_{[i](S)} C_{[i]}(S)$. Ainsi, on sait que $\sum w'_i \lambda[i]$ est une borne inférieure du problème.

3.4.1.3 $Pm|r_i|\sum T_i$

Soient $d'_1 \leq d'_2 \leq \ldots \leq d'_n$ les dates échues des jobs triées par valeurs croissantes. En s'appuyant sur le fait que $\max(0, C_1 - d_1) + \max(0, C_2 - d_2) \geq \max(0, C_1 - d_2) + \max(0, C_2 - d_1)$ si $C_1 \leq C_2$ et $d_1 \geq d_2$, on peut montrer que $\sum \max(0, C_{[i]}(S) - d'_i) \leq \sum \max(0, C_{[i]}(S) - d_{[i](S)})$. Par conséquent, on peut affirmer que $\sum \max(0, \lambda[i] - d'_i) \leq \sum \max(0, C_{[i]}(S) - d_{[i](S)})$. Cela montre que $\sum \max(0, \lambda[i] - d'_i)$ est une borne inférieure de ce problème.

3.4.1.4 $Pm|r_i|\sum w_i T_i$

Soient $d'_1 \leq d'_2 \leq \ldots \leq d'_n$ les dates échues des jobs triées par valeurs croissantes, et $w'_1 \leq w'_2 \leq \ldots \leq w'_n$ les poids des jobs, triés par valeurs croissantes. On a $\sum w_i T_i = \sum w_i \max(0, C_i - d_i) = \sum w_i \max(d_i, C_i) - \sum w_i d_i \geq \sum w_i C_i - \sum w_i d_i$. Or $\sum w_i C_i \geq \sum w'_i C_{[i]}(S)$ et $\sum w_i d_i \geq \sum w'_i d'_i$. On obtient donc $\sum w_i T_i \geq \sum w'_i C_{[i]}(S) - \sum w'_i d'_i$. Finalement, la quantité $\sum w'_i \lambda[i] - \sum w'_i d'_i$ est une borne inférieure pour le problème du retard total pondéré.

3.4.2 Bornes inférieures simples

Nous décrivons ici deux manières de calculer des bornes inférieures λ des valeurs $\underline{C}$.

Tout d'abord, nous supposons sans perte de généralité que les jobs ont été indexés selon les sommes de leur date de disponibilité et de leur durée croissantes $(r_i + p_i)$. On peut maintenant remarquer que le job qui termine en i^{e} position dans un ordonnancement réalisable quelconque ne peut terminer avant la date $r_i + p_i$. On en déduit que la valeur $r_i + p_i$ est une borne inférieure de $\underline{C}[i]$. Cette borne peut être calculée en $O\big(n \log(n)\big)$, puisqu'il suffit d'effectuer un tri de n valeurs. Nous noterons désormais $\lambda^{r+p}[i] = r_i + p_i$, $i \in N$ ces valeurs et $lb_{[\,]r+p}$ la borne inférieure qui leur est associée.

Supposons maintenant que les dates de disponibilité sont relâchées à la plus petite d'entre elles, notée r, et que la préemption est autorisée. Le problème $P|pmtn|C_{max}$ peut être résolu polynomialement en $O(n)$ par l'algorithme de McNaughton [50], avec

$$C^*_{max} = r + \max\left(\left\lceil \frac{1}{m}\sum_{i=1}^{n} p_i \right\rceil, \max_{i \in N} p_i\right).$$

Afin de calculer des valeurs de $\lambda[i]$,on peut donc chercher le sous-ensemble de i jobs qui mène à une date de fin de projet minimale. Cette recherche est très simple puisqu'il suffit de conserver les i jobs qui ont les plus petites durées. Si l'on suppose que l'on a trié les jobs selon leurs durées croissantes, la valeur $r + \max\left(\left\lceil \frac{1}{m}\sum_{i=1}^{n} p_i \right\rceil, \max_{i \in N} p_i\right)$ est une borne inférieure de $\underline{C}[i]$. Nous notons désormais ces valeurs $\lambda^{spt}[i] = r + \max\left(\left\lceil \frac{1}{m}\sum_{i=1}^{n} p_i \right\rceil, \max_{i \in N} p_i\right)$, $i \in N$ et $lb_{[\,]spt}$ la borne inférieure qui leur est associée. Lors du calcul de cette borne, le tri des jobs selon les durées croissantes est l'opération la plus coûteuse, sa complexité est donc $O\big(n \log(n)\big)$.

Sur l'instance d'exemple, les valeurs de λ^{r+p} et de λ^{spt} sont indiquées sur le Tableau 3.4. Sur cette instance, les bornes inférieures $lb_{[\,]r+p}$ et $lb_{[\,]spt}$ calculées pour la somme des dates de fin valent respectivement 30 et 29.

job i	1	2	3	4	5
$\lambda^{r+p}[i]$	6	5	6	5	9
$\lambda^{spt}[i]$	1	3	5	7	11

Tableau 3.4 – Valeurs de λ^{r+p} et de λ^{spt}.

3.4.3 Utilisation du *Job Splitting*

La technique du *Job Splitting* consiste à autoriser la préemption ainsi que l'exécution simultanée de plusieurs parties d'un même job sur des machines différentes [68]. Un ordonnancement optimal de ce problème relâché peut être construit en ordonnançant les jobs selon la règle *Shortest Remaining Processing Time* étendue : à la date t, le job k qui a la plus petite durée restante est choisi. On ordonnance des morceaux unitaires de ce job k le plus tôt possible sur une ou plusieurs machines, tant que le job n'est pas terminé et qu'aucune date de disponibilité n'a été atteinte. La i$^{\text{e}}$ plus petite date de fin au sein de cet ordonnancement optimal est une borne inférieure de $\underline{C}[i]$ [68]. Nous notons ces valeurs $\lambda^{split}[i]$, $i \in N$ et $lb_{[\,]split}$ la borne inférieure qui leur est associée. Cette dernière peut se calculer en $O(n \log n)$ opérations à l'aide d'un tas.

Sur la Figure 3.5, un ordonnancement optimal sur l'instance d'exemple est représenté, ainsi que les valeurs λ^{split} obtenues. Pour le critère de la somme des dates de fin, cette borne inférieure vaut $lb_{[\,]split} = 29$.

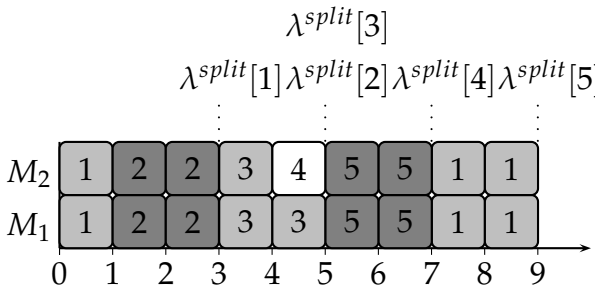


Figure 3.5 – Une solution optimale lorsque le *Job Splitting* est autorisé.

3.4.4 Utilisation du théorème de Horn

Nous présentons dans cette section plusieurs bornes inférieures qui utilisent un résultat de Horn [41]. Ce théorème permet de résoudre en temps polynomial le problème $Pm|r_i, pmtn|C_{max}$ et peut être reformulé comme suit :

Théorème 3.1 ([41]). *Il existe un ordonnancement préemptif où tous les jobs terminent avant la date C si et seulement si :*

$$\begin{aligned} \forall j \in N, \sum_{k=1}^{n} r(k,j) &\leq m(C - r_j) \\ \forall j \in N, r_j + p_j &\leq C, \end{aligned}$$

où $r(k,j) = \max\left(0, \min(p_k, r_k + p_k - r_j)\right)$.

La valeur $r(k,j)$ représente la quantité minimale de temps du job k qui doit nécessairement être exécutée après la date r_j.

Nous allons désormais utiliser ce théorème afin d'obtenir des bornes inférieures de $\underline{C}$ en relâchant la contrainte de non-préemption de notre problème. Pour chaque $i \in N$, l'idée consiste à essayer de trouver le sous-ensemble de i jobs qui minimise le *makespan*. La valeur ainsi obtenue est une borne inférieure de $\underline{C}[i]$. En effet, la date de fin minimale du job qui termine en i^{e} position dans le cas préemptif est naturellement plus faible que la date de fin minimale du job qui termine en i^{e} position dans le cas non-préemptif. Nous notons $\lambda^{mip}[i]$ ces valeurs, et $lb_{[\,]mip}$ la borne inférieure qui leur est associée.

Afin de calculer de manière exacte les valeurs $\lambda^{mip}[i]$, *i.e.* le *makespan* minimal lorsque i jobs sont ordonnancés, nous proposons un *PLNE* qui se base sur le théorème de Horn. Il nous permet de déterminer si un job k fait partie de l'ensemble des i jobs à conserver ($X_k = 1$) ou bien s'il doit être rejeté ($X_k = 0$).

$$\min \lambda^{mip}[i] \tag{3.1}$$

$$\forall j \in N,\ r_j X_j + \frac{1}{m} \sum_{k=1}^{n} r(k,j) X_k \leq \lambda^{mip}[i] \tag{3.2}$$

$$\forall j \in N,\ (r_j + p_j) X_j \leq \lambda^{mip}[i] \tag{3.3}$$

$$\sum_{k=1}^{n} X_k = i \tag{3.4}$$

$$\forall j \in N,\ X_j \in \{0;1\}. \tag{3.5}$$

La Contrainte 3.2 assure que pour chaque job j retenu ($X_j = 1$), le temps machine disponible entre r_j et $\lambda^{mip}[i]$ est au moins égal à la somme des durées minimales des jobs qui doivent nécessairement y être exécutés. La Contrainte 3.3 garantit que chaque job retenu dispose du temps minimal nécessaire à son exécution. Enfin, la Contrainte 3.4 spécifie que i jobs doivent être retenus.

i	1	2	3	4	5
λ^{mip}	5	5	6	8	10
sous-ensembles	$\{4\}$	$\{2,4\}$	$\{1,2,4\}$	$\{1,2,3,4\}$	$\{1,2,3,4,5\}$

Tableau 3.5 – Sous-ensembles « optimaux » de jobs et les valeurs de λ^{mip}.

Dans le cas particulier montré en exemple dans le Tableau 3.5, les sous-ensembles « optimaux » sont incrémentaux, c'est-à-dire englobant le sous-ensemble de rang inférieur. Cette propriété n'est pas nécessairement vérifiée.

Nous présentons désormais une manière d'améliorer le calcul de $\lambda^{mip}[i]$. Cette dernière est calculée à l'aide du *PLNE* qui doit être résolu de manière exacte. Nous verrons plus loin une borne inférieure $lb_{[\,]resort}$, qui est une relaxation de la borne $lb_{[\,]mip}$. On sait donc que $\lambda^{resort}[i] \leq \lambda^{mip}[i]$. Si on connaît par ailleurs une borne supérieure $\underline{C}[i]$ de $\lambda^{mip}[i]$, on peut tester si $\lambda^{resort}[i] = \underline{C}[i]$, avant de lancer la résolution du *PLNE*. Si tel est le cas, on connaît alors la valeur de $\lambda^{mip}[i]$ sans avoir à résoudre le *PLNE*. Dans le cas contraire, on peut ajouter la contrainte $\lambda^{resort}[i] \leq \lambda^{mip}[i] \leq \underline{C}[i]$ au *PLNE*. Cette méthode est décrite par l'Algorithme 3.5. Le calcul d'une borne supérieure de $\underline{C}[i]$ est aisé, puisque tout ordonnancement réalisable peut fournir des valeurs de $\underline{C}[i]$. Néanmoins, afin que ces valeurs soient aussi faibles que possible, nous proposons la méthode suivante, décrite dans l'Algorithme 3.4. L'idée consiste à calculer $\underline{C}[i]$ en ajoutant un job j à l'ensemble des jobs utilisés pour calculer $\underline{C}[i-1]$. Le job j est toujours celui qui mène à la valeur de $\underline{C}[i]$. Autrement dit, notre calcul serait optimal si l'on ajoutait une contrainte obligeant les sous-ensembles « optimaux » à être incrémentaux. Cette procédure requiert un temps de calcul en $O(n \log n)$.

Algorithme 3.4 : Calcul d'une borne supérieure des $\underline{C}[i]$

Données : n jobs

Résultat : Un tableau UB de taille n

Initialiser le tableau *pris* à 0 dans toutes les cases;

pour $i = 1$ *à* n **faire**

  $UB[i] \leftarrow \infty$; $k^* \leftarrow -1$;

  pour chaque *job k tel que* $pris[k] = 0$ **faire**

    Calculer le *makespan* optimal C_k du problème $Pm|r_i, pmtn|C_{max}$ avec les jobs $\{j \in [1,n],\ pris[j] = 1\} \cup \{k\}$;

    si $UB[i] > C_k$ **alors**

      $UB[i] \leftarrow C_k$; $k^* \leftarrow k$;

  $pris[k^*] \leftarrow 1$;

Algorithme 3.5 : Calcul amélioré de $\lambda^{mip}[i]$

Données : n jobs

Résultat : $\lambda^{mip}[i]$

Calculer $lb_{[\,]resort}[i]$ et $UB[i]$;
si $lb_{[\,]resort}[i] = UB[i]$ **alors**
  $\lambda^{mip}[i] = UB[i]$;
sinon
  Résoudre le *PLNE* avec la contrainte supplémentaire
  $lb_{[\,]resort} \leq \lambda^{mip}[i] \leq UB[i]$;
retourner $\lambda^{mip}[i]$

3.4.4.1 Relâcher la contrainte d'intégrité

Afin de réduire le temps de calcul nécessaire pour obtenir des valeurs λ^{mip}, nous proposons une relaxation continue des variables X_j dans la précédente formulation. Nous notons $\lambda^{lp}[i]$ les valeurs obtenues de cette façon, et $lb_{[\,]lp}$ la borne inférieure qui leur est associée.

3.4.4.2 Trier les coûts

Nous proposons ici une méthode qui nous permet d'obtenir des bornes inférieures de $\underline{C}[i]$ en temps polynomial.

Rappelons qu'à chaque job j est associé un ensemble de valeurs $r(k,j),\ k \in N$ qui correspondent à la quantité minimale du job k qui doit s'exécuter après la date r_j. Pour j fixé, notons $r'(k,j),\ k \in N$ ces mêmes valeurs triées de manière croissante , *i.e.* $r'(1,j) \leq r'(2,j) \leq \ldots \leq r'(n,j)$. On a alors $\forall j \in N, \sum_{k=1}^{i} r'(k,j) \leq \sum_{k=1}^{n} r(k,j)X_k$ puisque dans le meilleur des cas, les i jobs qui sont conservés ($X_k = 1$) sont ceux auxquels sont associées les i plus petites valeurs $r(k,j)$. Ainsi, on obtient $\forall j \in N, r_j X_j + \frac{1}{m}\sum_{k=1}^{i} r'(k,j) \leq r_j X_j + \frac{1}{m}\sum_{k=1}^{n} r(k,j)X_k \leq \lambda^{mip}[i]$.

Supposons désormais que le job j fasse partie de l'ensemble des i jobs tels que $X_k = 1$. On note $\theta_j = r_j + \frac{1}{m}\sum_{k=1}^{i} r'(k,j)$. On a $\theta_j \leq r_j + \frac{1}{m}\sum_{k=1}^{n} r(k,j)X_k \leq \lambda^{mip}[i]$. On note $\theta'_j, j \in N$ ces mêmes valeurs triées dans l'ordre croissant , *i.e.* $\theta'_1 \leq \theta'_2 \leq \ldots \leq \theta'_n$. Or, on sait que i jobs doivent être conservés. Donc $\lambda^{mip}[i]$ doit être supérieur ou égal à i valeurs de θ_j au minimum. On peut donc affirmer que $\theta'_i \leq \lambda^{mip}[i]$.

De plus, la contrainte 3.3 impose que pour chaque job j conservé, *i.e.* $X_j = 1$, on ait $\lambda^{mip}[i] \geq r_j + p_j$. Dans la mesure où i jobs doivent être conservés, on peut dire que la i^{e} plus petite valeur parmi $\{r_j + p_j,\ j \in N\}$ est une borne inférieure de $\lambda^{mip}[i]$. En réutilisant les notations introduites dans la Section 3.4.2, on peut écrire que $\lambda^{r+p}[i] \leq \lambda^{mip}[i]$.

Finalement, la quantité $\max(\theta'_i, \lambda^{r+p}[i])$ est une borne inférieure de $\lambda^{mip}[i]$, donc une borne inférieure de $\underline{C}[i]$. Nous notons $\lambda^{resort}[i] =$

$\max(\theta'_i, \lambda^{r+p}[i])$, $i \in N$ ces valeurs, et $lb_{[\,]resort}$ la borne inférieure qui leur est associée.

Les valeurs $r'(k,j)$ peuvent être calculées en $O(n^2 \log n))$ puisque n tris de n valeurs doivent être effectués. À partir de là, les valeurs de $\lambda^{resort}[i]$ sont calculées incrémentalement : pour chaque i, les n valeurs de θ sont calculées en $O(n)$ en considérant les valeurs de θ de l'itération précédente et sont triées en $O(n^2 \log n)$. Finalement, la borne inférieure $lb_{[\,]resort}$ peut être calculée en $O(n^2 \log n)$.

L'Algorithme 3.6 présente le calcul des valeurs $\lambda^{resort}[i]$. Il faut tout d'abord, pour chaque valeur de j, trier les valeurs $r(k,j)$ par ordre croissant, pour que $r(1,j) \leq r(2,j) \leq \cdots \leq r(n,j)$. Les valeurs $\lambda^{resort}[i]$ vont ensuite être calculées de la façon suivante. Lorsque i est fixé, on calcule pour chaque valeur de j la valeur $\theta_j = \left\lceil r_j + \frac{1}{m} \sum_{u=1}^{i} r(u,j) \right\rceil$. La valeur $s(i,j)$ correspond à $\sum_{u=1}^{i} r(u,j)$. Ces dernières sont calculées incrémentalement. Une fois tous les θ_j calculés, on les trie par ordre croissant. La valeur $\lambda^{resort}[i]$ est alors égale à $\max(\lambda^{r+p}[i], \theta_i)$. Cette opération est repétée pour chaque valeur de i. Il y a deux étapes dans cet algorithme : l'initialisation et les calculs. Pendant l'initialisation, on effectue n tris de n valeurs, cette étape coûte donc $O(n^2 \log n)$. Pour chaque valeur de i, l'étape de calcul coûte $O(n \log n)$, puisque l'opération la plus coûteuse est le tri des valeurs θ_j. Finalement, cet algorithme est en $O(n^2 \log n)$.

Algorithme 3.6 : Calcul de la borne inférieure $lb_{[\,]resort}$

Données : n jobs et une matrice r contenant les valeurs $r(k,j)$

Résultat : Un tableau $Resort$ de taille n où $Resort[i]$ est égal à $\lambda^{resort}[i]$

pour $j = 1$ *à* n **faire**
  Trier les $r(k,j)$ pour que $r(1,j) \leq r(2,j) \leq \cdots \leq r(n,j)$;
Initialiser une matrice s de dimension n^2 à 0;
pour $i = 1$ *à* n **faire**
  pour $j = 1$ *à* n **faire**
    $s(i,j) \leftarrow s(i,j) + r(i,j)$;
    $\theta_j \leftarrow \left\lceil r_j + \frac{1}{m} s(i,j) \right\rceil$;
  Trier les θ_j pour que $\theta_1 \leq \theta_2 \leq \cdots \leq \theta_n$;
  $Resort[i] = \max(\lambda^{r+p}[i], \theta_i)$;

Sur le Tableau 3.6 apparaissent les différentes valeurs nécessaires pour calculer $\lambda^{resort}[4]$ sur l'instance d'exemple. On sait que 4 jobs doivent être ordonnancés, on regarde donc la valeur θ'_4 qui vaut 7. Par ailleurs, on a $\lambda^{r+p}[4] = 6$ donc $\lambda^{resort}[4] = \max(7,6) = 7$. La valeur globale de la borne inférieure $lb_{[\,]resort}$ sur cette instance est égale à 34 pour le critère de la somme des dates de fin.

$$r(k,j) = \begin{pmatrix} 6 & 5 & 3 & 2 & 1 \\ 4 & 4 & 2 & 1 & 0 \\ 3 & 3 & 3 & 2 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix} \quad r'(k,j) = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 3 & 3 & 2 & 1 & 0 \\ 4 & 4 & 3 & 2 & 1 \\ 4 & 4 & 3 & 2 & 1 \\ 6 & 5 & 4 & 4 & 4 \end{pmatrix}$$

job j	1	2	3	4	5
θ_j	6	7	7.5	7	6

Tableau 3.6 – Calculs de la valeur de $\lambda^{resort}[4]$.

3.4.5 Une combinaison de bornes polynomiales

Dans les sections précedentes, nous avons décrit plusieurs manières de calculer des valeurs λ en temps polynomial. Certaines méthodes en dominent d'autres, par exemple λ^{resort} domine λ^{r+p}, tandis que certaines bornes sont incomparables entre elles. Nous proposons ici de combiner les méthodes non dominées entre elles, afin de constituer une borne inférieure aussi performante que possible sans pour autant perdre sa polynomialité. Chaque méthode calcule un ensemble de valeurs λ_i, mais ce n'est pas toujours la même méthode qui obtient la plus forte valeur lorsque i varie. On souhaite donc retenir la meilleure des valeurs pour chaque valeur de i. On définit donc pour chaque $i \in N$ la valeur $\lambda^{combo}[i]$ de la manière suivante :

$$\lambda^{combo}[i] = \max(\lambda^{split}[i], \lambda^{resort}[i]).$$

Nous notons désormais $lb_{[\,]combo}$ la borne inférieure associée à ces valeurs. La complexité de cette borne est de $O(n^2 \log n)$, qui est la complexité de λ^{resort}. Le Tableau 3.7 montre les valeurs de la borne inférieure $lb_{[\,]combo}$ sur l'instance d'exemple.

k	1	2	3	4	5
$\lambda^{resort}[k]$	5	5	6	7	10
$\lambda^{split}[k]$	3	5	5	7	9
$\lambda^{combo}[k]$	5	5	6	7	10

Tableau 3.7 – Valeurs de λ^{combo}.

3.5 Formulation indexée sur le temps

Les différentes bornes inférieures que nous présentons ici sont des relaxations d'une formulation linéaire indexée sur le temps. Dans cette formulations, les variables sont :

$$x_{jt} = \begin{cases} 1 \text{ si le job } j \text{ commence à la date } t\,; \\ 0 \text{ sinon.} \end{cases}$$

Soit H une borne supérieure du *makespan* de tout ordonnancement actif. On sait par ailleurs que le problème de maximisation du *makespan* parmi les ordonnancements actifs est NP-Difficile [4], nous en calculerons donc une borne supérieure (voir Section 4.6). À l'aide de ces variables, nous introduisons la formulation suivante :

$$\min_{j,t} \sum_{j=1}^{n} \sum_{t=0}^{H} x_{jt} f_j(t + p_j) \tag{3.6}$$

$$\forall j \in N, \forall t < r_j, \quad x_{jt} = 0 \tag{3.7}$$

$$\forall t \in [0, H], \quad \sum_{j=1}^{n} \sum_{t'=t-p_j+1}^{t} x_{jt'} \leq m \tag{3.8}$$

$$\forall j \in N, \quad \sum_{t=0}^{H} x_{jt} = 1 \tag{3.9}$$

$$\forall t \in [0, H], \forall j \in [1, n], \quad x_{jt} \in \{0, 1\} \tag{3.10}$$

La Contrainte 3.7 garantit qu'un job ne commence pas son exécution avant sa date de disponibilité. Il ne peut y avoir plus de m jobs qui s'exécutent à une même date grâce à la Contrainte 3.8. Enfin, chaque job commence son exécution une fois et une seule (Contrainte 3.9).

Parmi les bornes inférieures basées sur une formulation indexée sur le temps, plusieurs sont des relaxations lagrangiennes (voir Section 1.2.3). L'Algorithme 3.7 présente le schéma itératif utilisé. Il permet de mettre à jour les coefficients lagrangiens. Nous utilisons pour cela une méthode de sous-gradient. Cela nous permet de choisir une direction de descente. Une fois la direction choisie, la principale difficulté dans ce type de méthode réside dans le choix du pas. L'algorithme commence par initialiser tous ces coefficients lagrangiens à 1. Le paramètre ρ, initialisé à 2, permet de calibrer la taille du pas. Sa valeur diminue au fur et à mesure d'une exécution de l'algorithme. Plus précisement, s'il n'y a pas d'amélioration depuis n itérations, ρ est multiplié par 0.95. L'algorithme se termine lorsque ρ est jugé suffisamment petit (ici inférieur à 0.0005).

3.5.1 Relaxation continue

Pour obtenir une borne inférieure de la solution optimale de ce *PLNE*, on peut tout d'abord relâcher la contrainte d'intégrité (Contrainte 3.10) et ré-

Algorithme 3.7 : Schéma itératif de mise à jour des coefficients de Lagrange

Données : n jobs et un vecteur de Lagrange λ, une borne supérieure UB de la solution V

Résultat : Une valeur V

$V \leftarrow 0$;
$\rho \leftarrow 2$;
$iter \leftarrow 0$;
$best\_iter \leftarrow 0$;
Mettre tous les coefficients λ_i à 1;
tant que $\rho > 0.0005$ **faire**
    $iter \leftarrow iter + 1$;
    Calculer $V(\lambda)$;
    si $V(\lambda) > V$ **alors**
        $V \leftarrow V(\lambda)$;
        $best\_iter \leftarrow iter$;
    si $best\_iter > iter + n$ **alors**
        $\rho \leftarrow \rho * 0.95$;
        $best\_iter \leftarrow iter$;
    $\lambda \leftarrow \lambda + \rho\gamma(\lambda)\frac{UB - V(\lambda)}{\|\gamma(\lambda)\|^2}$;
retourner V

soudre le *PL* ainsi formé. Cette contrainte devient :

$$\forall t \in [0, H], \forall j \in N, \;\; x_{jt} \in [0, 1].$$

Nous notons cette borne inférieure lb_{t-lp}. Bien sûr, dans le cas particulier où la solution obtenue en résolvant le *PL* est entière, elle est également optimale pour le *PLNE*.

3.5.2 Relaxation lagrangienne de la contrainte de ressource

Une autre manière d'obtenir une borne inférieure de ce problème consiste à réaliser une relaxation lagrangienne sur la contrainte de ressource 3.8. Cette contrainte garantit qu'aucun job n'en chevauche un autre. Cette dernière est donc retirée et placée dans la fonction objectif. On note $\{\lambda_t, t \in [1, H]\}$ les multiplicateurs de Lagrange qui lui sont associés. On obtient la formulation suivante :

$$\begin{aligned}
\min_{j,t} \sum_{j=1}^{n} \sum_{t=0}^{H} x_{jt} f_j(t + p_j) + \sum_{t=0}^{H} \lambda_t \left(\sum_{j=1}^{n} \sum_{t'=t-p_j+1}^{t} x_{jt'} - m \right) \\
\forall j \in N, \forall t < r_j, \;\; x_{jt} = 0 \\
\forall j \in N, \;\; \sum_{t=0}^{H} x_{jt} = 1 \\
\forall t \in [0, H], \forall j \in N, \;\; x_{jt} \in \{0, 1\}.
\end{aligned}$$

Nous introduisons désormais la fonction δ :

$$\delta(t_1, t_2, j) = \begin{cases} \lambda_{t_2} & \text{si } t_1 \in [t_2 - p_j + 1, t_2[\,; \\ 0 & \text{sinon.} \end{cases}$$

À l'aide de la fonction δ, on peut maintenant reformuler la fonction objectif de la manière suivante :

$$\sum_{j=1}^{n} \sum_{t=0}^{H} x_{jt} \left(f_j(t + p_j) + \sum_{t'=0}^{H} \delta(t, t', j) \right) - m \sum_{t=0}^{H} \lambda_t.$$

La formulation de cette relaxation lagrangienne devient :

$$\begin{aligned}
\min_{j,t} \sum_{j=1}^{n} \sum_{t=0}^{H} x_{jt} \left(f_j(t + p_j) + \sum_{t'=0}^{H} \delta(t, t', j) \right) - m \sum_{t=0}^{H} \lambda_t \\
\forall j \in N, \forall t < r_j, \;\; x_{jt} = 0 \\
\forall j \in N, \;\; \sum_{t=0}^{H} x_{jt} = 1 \\
\forall t \in [0, H], \forall j \in N, \;\; x_{jt} \in \{0, 1\}.
\end{aligned}$$

Pour des raisons pratiques, on introduit également les variables suivantes :

$$\begin{aligned} A &= m\sum_{t=0}^{H}\lambda_t; \\ \forall j \in N, \forall t \in [0,H],\ \alpha_{jt} &= w_j\left(f_j(t+p_j)+\sum_{t'=0}^{H}\delta(t,t',j)\right). \end{aligned}$$

On peut maintenant remarquer que A n'est pas fonction des variables x_{jt} et qu'il est par conséquent un terme constant dans la fonction objectif. On peut retirer A de la fonction objectif sans changer la solution optimale.

Par ailleurs, les variables x_{jt} ne sont plus liées que par le paramètre t. Afin de trouver une solution optimale, on peut donc minimiser indépendamment $\sum \alpha_{jt}x_{jt}$ pour chaque valeur de j. De plus, lorsque j est fixé, seule une variable x_{jt} doit être non nulle d'après la contrainte $\forall j \in N, \sum x_{jt} = 1$. Donc minimiser $\sum_{t=0}^{H}\alpha_{jt}x_{jt}$ revient à choisir $t = t^*$ tel que $\alpha_{jt^*} = \min_{t\in[r_j,H]}\alpha_{jt}$.

Une fois la valeur optimale obtenue, on utilise alors une technique de sous-gradient (voir par exemple [35]) pour mettre à jour les multiplicateurs de Lagrange et obtenir des valeurs successives de bornes inférieures. Nous notons $lb_{t-lag-mach}$ cette borne inférieure.

L'Algorithme 3.8 décrit le calcul d'une valeur LB lorsque la contrainte de ressource a été relâchée. Pour chaque valeur de j, on cherche la valeur de t qui va minimiser $\alpha(j,t), t \in [r_j, H]$. La somme de ces valeurs, à laquelle on retranche la quantité $m\sum_{t=0}^{H}\lambda_t$, définit la valeur LB. Cet algorithme est en $O(nH)$, où H représente l'horizon.

Algorithme 3.8 : Calcul d'une solution optimale de la relaxation lagrangienne de la contrainte de ressource

Données : Un ensemble de n jobs et H coefficients lagrangiens λ_i
Résultat : Une valeur LB

Calculer $\delta(t_1,t_2,j)$ pour $(t_1,t_2,j) \in [0,H]^2 \times [1,n]$;
Calculer $\alpha(j,t)$ pour $(j,t) \in [1,n] \times [0,H]$;
$LB \leftarrow 0$;
pour $j = 1$ *à* n **faire**
  $min_j \leftarrow +\infty$;
  pour $t = r_j$ *à* H **faire**
    si $\alpha(j,t) < min_j$ **alors** $min_j \leftarrow \alpha(j,t)$;
  $LB \leftarrow LB + \min_j$;
$LB \leftarrow LB - m\sum_{t=0}^{H}\lambda_t$;
retourner LB

Proposition 3.4. *Les bornes inférieures lb_{t-lp} et $lb_{t-lag-mach}$ sont égales.*

Démonstration. On peut remarquer que la solution optimale de la relaxation lagrangienne est obtenue en choisissant pour chaque valeur de j la valeur t^* de t telle que $\alpha_{jt^*} = \min_{t\in[r_j,H]} \alpha_{jt}$. La contrainte d'intégrité est donc inutile au sein de cette formulation. Par ailleurs, on sait que le saut de dualité pour une relaxation lagrangienne d'un *PL* est nul. On peut donc affirmer que les bornes inférieures lb_{t-lp} sont $lb_{t-lag-mach}$ sont égales. □

3.5.3 Relaxation lagrangienne sur le nombre d'occurrences

3.5.3.1 Résolution dans le cas basique

Nous choisissons dans cette section de relâcher la contrainte qui garantit qu'un job s'exécute une fois et une seule. On utilise ici encore une relaxation lagrangienne pour obtenir la formulation suivante :

$$\min_{j,t} \sum_{j=1}^{n}\sum_{t=0}^{H} x_{jt} f_j(t+p_j) + \sum_{j=1}^{n} \lambda_j (1 - \sum_{t=0}^{H} x_{jt}) \tag{3.11}$$

$$\forall j \in N, \forall t < r_j, \;\; x_{jt} = 0 \tag{3.12}$$

$$\forall t \in [0,H], \;\; \sum_{j=1}^{n} \sum_{t'=t-p_j+1}^{t} x_{jt'} \leq m \tag{3.13}$$

$$\forall t \in [0,H], \;\; \forall j \in N, x_{jt} \in \mathbb{N}. \tag{3.14}$$

Pour résoudre ce problème, il faut trouver l'ordonnancement optimal sur m machines où les jobs peuvent ne pas être exécutés, être exécuté une fois ou exécutés plusieurs fois. Le coût d'une solution réalisable est composé de deux parties : la somme des coûts des jobs $\sum_{j=1}^{n}\sum_{t=0}^{H} x_{jt}\big(f_j(t+p_j) - \lambda_j\big)$ et une quantité constante $\sum_{j=1}^{n} \lambda_j$. Dans la mesure où il est autorisé de n'exécuter aucun job, on sait que la somme des coûts des jobs minimale doit être négative ou nulle. Par ailleurs, on peut remarquer qu'il existe une solution optimale où toutes les machines accueillent le même ordonnancement. Dans le cas contraire, il serait possible de construire un ordonnancement meilleur que l'optimum en reproduisant m fois la machine de coût minimal au sein de l'ordonnancement supposé optimal. Cette remarque nous amène à chercher l'ordonnancement de coût minimal sur une machine. On note $\Lambda(t)$ le coût minimal induit par l'ordonnancement de jobs entre les dates t et H sur une machine. On sait que $\Lambda(H) = 0$. On utilise alors le programme dynamique suivant pour calculer les valeurs de Λ :

$$\Lambda(t) = \min\Big(\Lambda(t+1), \min_{j/r_j \geq t} \big(f_j(t+p_j) - \lambda_j + \Lambda(t+p_j)\big)\Big).$$

Le coût minimal sur m machines est alors égal à $\lambda^* = m\Lambda(0) + \sum_{j=1}^{n} \lambda_j$. Le calcul d'un $\Lambda(t)$ se fait en $O(n)$, cette borne se calcule donc en $O(nH)$. Nous notons $lb_{t-lag-occ}$ cette borne inférieure.

L'Algorithme 3.9 détaille le calcul d'une valeur de la borne inférieure $lb_{t-lag-occ}$. C'est un programme dynamique. On calcule à chaque itération la valeur $\Lambda(t)$ qui correspond au coût minimal entre H et t, lorsque la contrainte d'occurrence a été relâchée. Cet algorithme est en $O(nH)$.

Algorithme 3.9 : Calcul d'une solution de la relaxation lagrangienne de la contrainte d'occurrence

Données : Un ensemble de n jobs et n coefficients lagrangiens λ_i;

Résultat : Une valeur V^*

$\Lambda(H) \leftarrow 0$;

pour $t = H - 1$ *à* 0 **faire**

$$\Lambda(t) \leftarrow \min\left(\Lambda(t+1), \min_{j/r_j \geq t} f_j(t+p_j) - \lambda_j + \Lambda(t+p_j)\right);$$

retourner $m\Lambda(0) + \sum_{j=1}^{n} \lambda_j$

Proposition 3.5. *Les bornes inférieures lb_{t-lp} et $lb_{t-lag-occ}$ sont égales.*

Démonstration. On peut remarquer que la contrainte d'intégrité est inutile au sein de cette formulation relâchée. Par ailleurs, on sait que le saut de dualité pour une relaxation lagrangienne d'un *PL* est nul. On peut donc affirmer que les bornes inférieures lb_{t-lp} sont $lb_{t-lag-occ}$ sont égales. □

3.5.3.2 Interdire les répétitions consécutives

Afin d'améliorer la borne inférieure présentée dans la section précédente, on peut ajouter la contrainte qui interdit un job de s'éxecuter plusieurs fois de suite de manière consécutive sur une machine. Nous restons bien dans le cas d'une relaxation du problème initial, puisque cette contrainte était implicitement établie par la contrainte d'occurrence. Cette notion est un cas particulier de la notion de $P(\lambda)$-chemins définie par Rivreau [59]. Nous devons désormais introduire quelques fonctions afin de résoudre ce problème :

- $\Lambda_1(t)$: le coût minimal d'un ordonnancement sans répétition entre les dates t et H ;
- $J(t)$: le job qui est exécuté en première position dans l'ordonnancement qui a permis de fixer la valeur de $\Lambda_1(t)$;
- $\Lambda_2(t)$: le coût minimal d'un ordonnancement sans répétition entre les dates t et H, qui ne commence pas par le job $J(t)$.

On peut alors calculer la valeur optimale de notre problème en utilisant le schéma suivant [59] :

Il nous faut tout d'abord introduire les notations suivantes :

$$\bar{\Lambda}(j,t) = \begin{cases} \Lambda_1(t) & \text{si } j \neq J(t); \\ \Lambda_2(t) & \text{sinon.} \end{cases}$$

$$\bar{J}(j,t) = \arg\min_{j/r_j \geq t} \Big(f_j(t+p_j) - \lambda_j + \bar{\Lambda}(j,t+p_j) \Big).$$

Les différents valeurs de Λ_1 s'obtiennent de la façon suivante :

$$\Lambda_1(t) = \min \Big(\Lambda_1(t+1), \min_{j/r_j \geq t} \big(f_j(t+p_j) - \lambda_j + \bar{\Lambda}(j,t+p_j) \big) \Big).$$

On peut désormais calculer $J(t)$ comme suit :

$$J(t) = \begin{cases} J(t+1) & \text{si } \Lambda_1(t) = \Lambda_1(t+1); \\ \bar{J}(t) & \text{sinon.} \end{cases}$$

Enfin, on obtient $\Lambda_2(t)$ de la manière suivante :

$$\Lambda_2(t) = \min \left(\bar{\Lambda}(J(t), t+1), \min_{\substack{j/r_j \geq t \\ \text{et } j \neq J(t)}} \big(f_j(t+p_j) - \lambda_j + \bar{\Lambda}(j,t+p_j) \big) \right).$$

À l'aide de ce schéma récursif, on peut calculer la valeur de $\Lambda_1(0)$, qui correspond à la valeur optimale de notre problème. À t fixé, tous les calculs sont linéaires, *i.e.* on peut calculer $\Lambda_1(t)$, $\Lambda_2(t)$ et $J(t)$ en $O(n)$. Finalement, le temps de calcul de cette borne est donc de $O(nH)$. Nous notons $lb_{t-lag-ltd-occ}$ cette borne inférieure.

L'Algorithme 3.10 présente le calcul d'une valeur LB de la relaxation de la contrainte d'occurrence où la répétition consécutive est interdite. C'est également un programme dynamique.

3.6 Résultats numériques

Une série de résultats numériques est présentée dans cette section. Elle nous permet de comparer l'efficacité de chacune de ces bornes inférieures. Cette étude est composée de cinq parties : une pour chaque critère et une pour des considérations générales. Tous ces calculs ont été effectués avec un *Pentium-M* 1,6 Ghz opérant sous *MS-Windows XP*. Nous avons utilisé des schémas de génération de la littérature pour les instances utilisées (voir Section 4.8.1).

Les tests ont été effectués sur des instances de 20, 50 et 100 jobs. Pour chaque taille d'instance, cinq nombres de machines sont possibles : 2, 3, 4, 5 ou 10. Pour chaque critère et pour chaque instance, nous avons calculé toutes les bornes inférieures, exceptée la borne lb_{t-lp} qui n'a pas été lancée sur les instance à 100 jobs en raison du temps de calcul nécessaire trop important. Nous avons ensuite calculé pour chaque instance la distance entre chacune des bornes inférieures et la meilleure d'entre elles. Sur chacun des tableaux présentés dans cette section, nous rapportons la distance moyenne « dist » à la meilleure des bornes (en pourcentage), groupée par taille d'instance. Nous fournissons aussi le temps de calcul moyen « cpu » (en secondes).

Algorithme 3.10 : Calcul d'une solution de la relaxation lagrangienne de la contrainte d'occurrence sans répétition

Données : Un ensemble de n jobs et n coefficients lagrangiens λ_i;
Résultat : Une valeur LB

$\Lambda_1(H) \leftarrow 0\,;\, \Lambda_2(H) \leftarrow 0\,;\, J(H) \leftarrow -1;$
pour $t = H-1$ *à* 0 **faire**
  $\Lambda_1(t) \leftarrow \Lambda_1(t+1)\,;\, J(t) \leftarrow J(t+1);$
  pour $j = 1$ *à* n **faire**
    si $r_j \geq t$ *et* $t + p_j \leq H$ **alors**
      $f_{jt} \leftarrow f_j(t+p_j) - \lambda_j;$
      si $J(t+p_j) \neq j$ **alors** $f_{jt} \leftarrow f_{jt} + \Lambda_1(t+p_j);$
      sinon $f_{jt} \leftarrow f_{jt} + \Lambda_2(t+p_j);$
      si $f_{jt} < \Lambda_1(t)$ **alors** $\Lambda_1(t) \leftarrow f_{jt}\,;\, J(t) \leftarrow j;$
  si $J(t) = J(t+1)$ **alors** $\Lambda_2(t) \leftarrow \Lambda_2(t+1);$
  sinon $\Lambda_2(t) \leftarrow \Lambda(t+1);$
  pour $j = 1$ *à* n **faire**
    si $j \neq J(t)$ *et* $t \geq r_j$ *et* $t + p_j \leq H$ **alors**
      $f_{jt} \leftarrow \Lambda_1(j,t) - \lambda_j;$
      si $J(t+p_j) \neq j$ **alors** $f_{jt} \leftarrow f_{jt} + \Lambda_1(t+p_j);$
      sinon $f_{jt} \leftarrow f_{jt} + \Lambda_2(t+p_j);$
      si $f_{jt} < \Lambda_2(t)$ **alors** $\Lambda(t) \leftarrow f_{jt};$
retourner $m\Lambda(0) + \sum_{j=1}^{n} \lambda_j$

Des test préliminaires ont été effectués afin d'évaluer les valeurs les plus favorables pour le paramètre de discrétisation de la borne basée sur un flot $lb_{flow''}()$. Il ressort à l'issue de ces tests que la valeur $\tau = 15$ semble être raisonnable.

3.6.1 $Pm|r_i|\sum C_i$

Les résultats relatifs à la somme des dates de fin sont présentés dans le Tableau 3.8. On peut y voir que les bornes obtenues à partir de la formulation indexée sur le temps fournissent les meilleurs résultats. Cependant, le temps de calcul nécessaire est assez important. Les bornes de flot où la discrétisation est améliorée $lb_{flow'}$ et $lb_{flow''}()$ sont également assez efficaces et demandent des temps de calcul beaucoup plus faibles. Parmi les bornes calculant des dates de fin minimales, on peut voir que la borne $lb_{[\,]mip}$ domine les autres, ce qui était attendu théoriquement. Finalement, c'est la borne $lb_{[\,]combo}$ qui semble réaliser le compromis le plus intéressant entre performance et rapidité.

	n=20		**n=50**		**n=100**	
lower bound	**dist**	**cpu**	**dist**	**cpu**	**dist**	**cpu**
$lb_{no-release}$	37,4	0,0	38,7	0,0	39,4	0,0
$lb_{no-release-subsets}$	15,1	0,0	18,5	0,0	19,5	0,0
lb_{flow}	14,3	0,0	11,7	0,1	10,2	0,8
$lb_{flow'}$	3,6	0,0	1,8	0,4	0,3	5,1
$lb_{flow''}(15)$	3,7	0,0	1,9	0,5	0,4	5,4
$lb_{[\,]r+p}$	10,4	0,0	13,4	0,0	14,1	0,0
$lb_{[\,]spt}$	46,1	0,0	44,2	0,0	42,6	0,0
$lb_{[\,]split}$	18,0	0,0	9,1	0,0	4,3	0,0
$lb_{[\,]mip}$	4,3	0,1	2,9	0,3	1,2	1,8
$lb_{[\,]lp}$	5,0	0,1	4,2	0,3	3,1	1,8
$lb_{[\,]resort}$	5,3	0,0	5,4	0,0	5,1	0,0
$lb_{[\,]combo}$	4,8	0,0	3,7	0,0	1,9	0,0
lb_{t-lp}	0,0	2,8	6,0	73,7	-	-
$lb_{t-lag-mach}$	0,1	50,2	0,0	775,3	-	-
$lb_{t-lag-occ}$	0,1	2,7	0,0	41,3	-	-
$lb_{t-lag-ltd-occ}$	0,0	5,4	0,0	82,9	-	-

Tableau 3.8 – Résultats pour la somme des dates de fin.

3.6.2 $Pm|r_i|\sum w_iC_i$

Le talbeau 3.9 montre les résultats obtenus par les bornes inférieures sur les instances de la somme pondérée des dates de fin. Les bornes inférieures basées sur la formulation indexée sur le temps fournissent les meilleurs résultats. Une fois de plus, cette qualité s'accompagne d'un temps de calcul assez élevé. L'ensemble des bornes basées sur les dates de fin minimales semble assez loin de la meilleure des bornes en moyenne. Les bornes de flot semblent pour ce critère être celles qui fournissent les résultats les plus intéressants si l'on considère la qualité et la rapidité.

	n=20		**n=50**		**n=100**	
lower bound	**dist**	**cpu**	**dist**	**cpu**	**dist**	**cpu**
lb_{flow}	12,5	0,00	9,7	0,08	8,0	0,95
$lb_{flow'}$	3,4	0,02	1,8	0,50	0,3	6,61
$lb_{flow''}(15)$	3,6	0,02	1,9	0,51	0,4	6,98
$lb_{[]r+p}$	28,4	0,00	32,9	0,00	35,1	0,00
$lb_{[]spt}$	62,0	0,00	64,7	0,00	64,1	0,00
$lb_{[]split}$	38,6	0,00	33,4	0,00	30,4	0,00
$lb_{[]mip}$	25,1	0,05	26,9	0,31	26,9	1,98
$lb_{[]lp}$	25,7	0,06	28,4	0,33	29,1	1,80
$lb_{[]resort}$	26,1	0,00	29,4	0,00	31,0	0,00
$lb_{[]combo}$	25,6	0,00	27,6	0,00	27,7	0,00
lb_{t-lp}	0,0	2,40	0,0	49,63	-	-
$lb_{t-lag-mach}$	0,0	51,29	0,0	808,47	-	-
$lb_{t-lag-occ}$	0,0	2,83	0,0	43,10	-	-
$lb_{t-lag-ltd-occ}$	0,0	5,62	0,0	85,55	-	-

Tableau 3.9 – Résultats pour la somme des dates de fin pondérée.

3.6.3 $Pm|r_i|\sum T_i$

Les résultats pour le retard total sont fournis dans le Tableau 3.10. On peut voir que les différentes relaxations de la formulation indexée sur le temps fournissent en moyenne des valeurs bien plus élevées que les autres bornes. Une fois encore, le temps de calcul nécessaire est important pour cette borne. Sur les instances de 100 jobs, comme elles ne sont pas calculées, nous pouvons comparer les autres bornes inférieures. Ce sont les bornes inférieures basées sur les flots qui semblent réaliser le compromis le plus intéressant entre la qualité de résultats et la rapidité de calcul.

	n=20		n=50		n=100	
lower bound	**dist**	**cpu**	**dist**	**cpu**	**dist**	**cpu**
$lb_{no-release}$	51,3	0,0	48,2	0,0	40,0	0,0
$lb_{no-release-subsets}$	50,6	0,0	47,5	0,0	39,2	0,0
lb_{flow}	58,8	0,0	50,5	0,2	27,7	1,6
$lb_{flow'}$	48,1	0,0	32,7	0,5	3,5	6,1
$lb_{flow''}(15)$	48,5	0,0	33,3	0,5	4,6	5,8
$lb_{[]r+p}$	68,5	0,0	69,3	0,0	64,5	0,0
$lb_{[]spt}$	51,5	0,0	42,4	0,0	29,7	0,0
$lb_{[]split}$	49,7	0,0	39,3	0,0	25,4	0,0
$lb_{[]mip}$	47,2	0,1	36,8	0,5	20,8	3,7
$lb_{[]lp}$	47,8	0,1	39,2	0,3	26,7	2,0
$lb_{[]resort}$	49,6	0,0	41,2	0,0	30,2	0,0
$lb_{[]combo}$	48,7	0,0	38,7	0,0	24,9	0,0
lb_{t-lp}	1,6	1,8	0,7	44,2	-	-
$lb_{t-lag-mach}$	4,1	22,0	1,8	326,6	-	-
$lb_{t-lag-occ}$	6,2	1,0	5,1	14,5	-	-
$lb_{t-lag-ltd-occ}$	3,3	1,7	2,5	25,1	-	-

Tableau 3.10 – Résultats pour le retard total.

3.6.4 $Pm|r_i|\sum w_iT_i$

Le Tableau 3.11 montre les résultats pour le critère du retard total pondéré. Le comportement des bornes inférieures semble assez proche de celui constaté pour le retard total. En particulier, les relaxations de la formulation indexée sur le temps fournissent les meilleurs résultats, avec un écart important par rapport aux autres bornes inférieures. Les bornes qui s'appuient sur les dates de fin minimales ne donnent pas de bons résultats. Enfin, les bornes de flot semblent être intéressantes quand les relaxations de la formulation indexée sur le temps ne peuvent pas être utilisées.

	n=20		**n=50**		**n=100**	
lower bound	**dist**	**cpu**	**dist**	**cpu**	**dist**	**cpu**
lb_{flow}	56,1	0,0	48,2	0,2	24,2	1,4
$lb_{flow'}$	45,9	0,0	32,4	0,5	2,1	6,9
$lb_{flow''}(15)$	46,5	0,0	33,2	0,5	3,3	6,9
$lb_{[]r+p}$	66,2	0,0	69,0	0,0	64,8	0,0
$lb_{[]spt}$	64,5	0,0	65,2	0,0	58,9	0,0
$lb_{[]split}$	64,7	0,0	65,2	0,0	58,9	0,0
$lb_{[]mip}$	64,5	0,1	65,1	0,5	58,8	3,6
$lb_{[]lp}$	64,5	0,1	65,1	0,3	58,8	1,8
$lb_{[]resort}$	64,5	0,0	65,1	0,0	58,8	0,0
$lb_{[]combo}$	64,5	0,0	65,1	0,0	58,8	0,0
lb_{t-lp}	0,4	2,5	0,2	50,9	-	-
$lb_{t-lag-mach}$	1,0	22,5	0,6	344,7	-	-
$lb_{t-lag-occ}$	1,3	1,0	1,2	14,7	-	-
$lb_{t-lag-ltd-occ}$	0,4	1,7	0,8	25,9	-	-

Tableau 3.11 – Résultats pour le retard total pondéré.

3.6.5 Considérations globales

Nous tentons ici de dresser une analyse plus générale de ces résultats numériques. La première remarque que l'on peut faire concerne la formulation indexée sur le temps dont les différentes relaxations fournissent de très bons résultats, au prix d'un temps de calcul élevé. Les bornes qui calculent des dates de fin minimales ne semblent réellement intéressantes que pour la somme des dates de fin. Il semble que la manière d'obtenir des bornes inférieures pour les autres critères détériore la qualité de ces bornes. Par exemple pour les critères pondérés, l'affectation des poids est faite dans le pire des cas, ce qui peut fortement limiter la qualité du résultat obtenu. Concernant les bornes par calcul de flot, les résultats obtenus sont plus intéressants pour la

somme des dates de fin (pondérée ou non) que pour le retard total (pondéré ou non). La manière de répartir le coût d'un job sur chacun de ses morceaux unitaires est en partie compensée pour la somme pondérée des dates de fin. Cela peut expliquer cette différence de comportement. Les bornes relâchant les dates de disponibilité paraissent assez peu intéressantes en pratique. Enfin, nous pouvons remarquer que la « meilleure » borne inférieure dépendra d'une part de l'utilisation que l'on veut en faire, et plus directement du temps dont on dispose pour la calculer, d'autre part du critère que l'on cherche à minimiser.

Méthode exacte

Sommaire

Nous présentons ici une méthode exacte dédiée aux quatre problèmes $Pm|r_i|\sum C_i$, $Pm|r_i|\sum w_i C_i$, $Pm|r_i|\sum T_i$ et $Pm|r_i|\sum w_i T_i$. Nous utilisons une méthode arborescente classique de type *Branch&Bound*, dans laquelle nous nous servons de règles de dominance ainsi que de bornes inférieures et supérieures pour élaguer l'arbre de recherche. Une description générale de cette méthode fait l'objet de la Section 4.2. Nous montrons ensuite dans la Section 4.3 comment les règles de dominance sont intégrées dans ce *Branch&Bound*. La borne supérieure fait l'objet de la Section 4.4. L'utilisation des bornes inférieures est présentée dans la Section 4.5. Dans certaines

règles de dominance comme dans certaines bornes inférieures, une borne supérieure du *makespan* de tout ordonnancement est nécessaire. Une manière de calculer cette borne est proposée dans la Section 4.6. L'algorithme de *Branch&Bound* est décrit dans la Section 4.7. Enfin, des résultats préliminaires pour chacun des critères, ainsi qu'une comparaison des règles de dominance, sont présentés dans la Section 4.8.

4.1 État de l'art

Si l'on s'intéresse en premier lieu aux problèmes à une machine, on trouve de nombreux travaux proposant des méthodes exactes de résolution. Chu a proposé un *Branch&Bound* pour le problème $1|r_i|\sum C_i$[24]. Bianco et Ricciardelli ont présenté une méthode arborescente pour le problème $1|r_i|\sum w_i C_i$ [15]. Hariri et Potts [38] puis Belouadah *et al.* [13] ont également proposé des *Branch&Bound* pour ce problème. Si les jobs ne disposent pas de dates de disponibilité mais sont soumis à des *deadlines*, *i.e.* $1|\overline{d}_i|\sum w_i C_i$, on peut citer les méthodes proposées par Potts et Wassenhove [58], ainsi que ceux de Pan [57]. Le problème $1|r_i|\sum T_i$ a été abordé par Chu, encore une fois par une méthode arborescente [25]. Concernant le retard total pondéré, Akturk et Ozdemir [2] ont proposé une méthode exacte pour résoudre le problème $1|r_i|\sum w_i T_i$. Enfin, Jouglet a généralisé et amélioré les travaux de [15, 14, 24, 25] sur une machine pour l'ensemble des quatre critères étudiés [42].

Concernant les problèmes à machines parallèles où l'on souhaite minimiser la somme des dates de fin, la littérature se décompose en deux grandes familles de contributions : la démonstration de problèmes particuliers polynomiaux et des méthodes exactes pour les problèmes difficiles. Ainsi, Bruno *et al.* ont démontré que le problème $R||\sum C_i$ était polynomial [20]. Lorsque les machines sont identiques, plusieurs situations ont été démontrées faciles à résoudre. Par exemple, si les jobs sont soumis à des *deadlines* et que la préemption est autorisée, *i.e.* $Pm|pmtn, \overline{d}_i|\sum C_i$, Azizoglu a démontré que le problème était polynomial si les durées et les *deadlines* étaient « agréables » , *i.e.* $(p_i \leq p_j) \Rightarrow (d_i \leq d_j)$ [5]. Leung et Pinedo se sont intéressés à ce même problème [48]. Lorsque tous les jobs ont la même durée, on trouve également plusieurs problèmes polynomiaux : Baptiste l'a montré le problème pour $Pm|r_i, p_i = p, size_i|\sum C_i$ où $size_i$ indique le nombre de machines nécessaires à l'exécution du job i [10] ; plusieurs algorithmes polynomiaux ont été présentés pour le problème $P|r_i, pmtn, p_i = p|\sum C_i$ (Brucker et Kravchenko [19], puis Baptiste *et al.* [11]). Des méthodes exactes ont été proposées pour les problèmes difficiles. On peut mentionner Yalaoui et Chu [68], puis Nessah *et al.* qui ont présenté des méthodes de *Branch&Bound* pour le problème $Pm|r_i|\sum C_i$ [52, 55, 53]. La somme pondérée des dates de fin a également été l'objet de différents travaux : Baptiste a démontré que lorsque les jobs sont soumis à des dates de disponibilité et qu'ils sont tous de durées égales, les problèmes $Pm|p_i = p, r_i|\sum w_i C_i$ et $Pm|p_i = p, r_i|\sum T_i$ sont polynomiaux [9]. Nous pouvons également mentionner le problème $Pm||\sum w_i C_i$, pour lequel Belouadah et Potts [14], puis Azizoglu et Kirca [7] ont proposé des méthodes

de résolution exacte. Pour le même problème avec des machines indépendantes ($R||\sum w_iC_i$), une méthode exacte s'appuyant sur la génération de colonnes a été présentée par Chen et Powell [23]. On trouve moins de travaux concernant le retard total, pondéré ou non. Toutefois, Azizoglu et Kirca ont abordé le problème $Pm||\sum T_i$ à l'aide d'un *Branch&Bound* [6]. Ce même problème a également été étudié par Yalaoui et Chu, une fois encore par une méthode arborescente [67].

4.2 Modélisation

Nous présentons ici la modélisation que nous avons adoptée. Afin de résoudre les problèmes d'ordonnancement qui nous intéressent, nous proposons une méthode de *Branch&Bound*. Lors de l'exploration, chaque sommet de l'arbre de recherche représente un ordonnancement partiel. Pour construire le fils d'un sommet, un job est ordonnancé à la suite de l'ordonnancement partiel du sommet père. Ce dernier est ordonnancé au plus tôt, par rapport à sa date de disponibilité sur la machine disponible au plus tôt.

Cette méthode arborescente balaie l'ensemble dominant des ordonnancements quasi-actifs (voir Section 2.2.1). Nous allons maintenant montrer qu'un ordonnancement quasi-actif, partiel ou non, peut être représenté, sans perte d'information, par une permutation de $[1, 2, \ldots, n]$.

Proposition 4.1. *L'ensemble Σ des permutations de $[1, 2, \ldots, n]$ peut être mis en bijection avec l'ensemble des ordonnancements quasi-actifs.*

Démonstration. Soit $\sigma \in \Sigma$ une permutation de N. On suppose les machines indicées. Pour former un ordonnancement quasi-actif, on ordonnance itérativement les jobs de N selon l'ordre fixé par σ, en choisissant toujours d'ordonnancer un job le plus tôt possible (c'est-à-dire au plus tôt sur la machine disponible au plus tôt). En cas d'égalité des dates de disponibilité des machines disponibles au plus tôt, on préfèrera celle de plus faible indice.

Cette méthode itérative, décrite plus précisement par l'Algorithme 4.2, permet d'obtenir un ordonnancement quasi-actif O à partir d'une permutation σ. On peut également obtenir σ à partir de O à l'aide de l'Algorithme 4.1. □

La Figure 4.1 montre un exemple d'ordonnancement et la séquence en bijection avec ce dernier.

L'algorithme 4.1 a une complexité linéaire, il en va de même pour l'algorithme 4.2.

À partir de maintenant, les ordonnancements que nous considérons seront tous quasi-actifs. Nous parlerons indifféremment d'ordonnancement quasi-actif ou de séquence.

Plusieurs informations sont maintenues pour chaque sommet S de l'arbre de recherche. Si l'on note σ la séquence associée à S, nous cherchons à filtrer

Algorithme 4.1 : Calcul d'un ordonnancement à partir d'une séquence

Données : Une séquence σ

Résultat : Un ordonnancement O

$O \leftarrow$ ordonnancement vide;

pour $i = 1$ *à* $taille(\sigma)$ **faire**

  Choisir la machine j disponible au plus tôt dans O. En cas d'égalité, préferer celle de plus faible indice;

  Ordonnancer dans O le job $\sigma[i]$ au plus tôt sur la machine j;

retourner O

Algorithme 4.2 : Calcul d'une séquence à partir d'un ordonnancement

Données : Un ordonnancement O de n jobs

Résultat : Une séquence σ

$\sigma \leftarrow [\,]$;

$\forall k \in [1, m], tab[k] \leftarrow 1$;

pour $i = 1$ *à* n **faire**

  $j^* \leftarrow 1$;

  Calculer la date de disponibilité Δ^* de la machine 1 avant l'exécution du job situé en $tab[1]^{\text{e}}$ position;

  pour $j = 1$ *à* m **faire**

    Calculer la date de disponibilité Δ_j de la machine j avant l'exécution du job situé en $tab[j]^{\text{e}}$ position;

    si $\Delta_j < \Delta^*$ *ou* $(\Delta_j = \Delta^*$ *et* $j < j^*)$ **alors**

      $j^* \leftarrow j$;

      $\Delta^* \leftarrow \Delta_j$;

  $job \leftarrow$le job en $tab[j^*]^{\text{e}}$ position sur la machine j^*;

  $\sigma \leftarrow (\sigma|job)$;

  $tab[j^*] \leftarrow tab[j^*] + 1$;

retourner σ

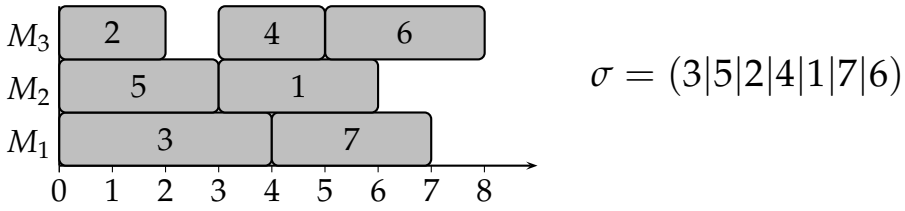


Figure 4.1 – Exemple d'un ordonnancement quasi-actif et sa séquence associée .

l'ensemble NS (pour *Not Scheduled*) des jobs non encore ordonnancés afin de déterminer un sous-ensemble PF (pour *Possible First*) des jobs qu'il est nécessaire de tester à la suite de σ. Un job i qui n'appartient pas à l'ensemble PF est un job pour lequel on sait que la séquence $(\sigma|i)$ ne mènera pas une solution de coût strictement inférieur au coût minimal des solutions ne commençant pas par cette séquence. En d'autres termes, ne pas explorer cette branche de l'arbre de recherche ne fait pas augmenter la valeur de l'optimum.

En résumé, un sommet S peut être vu comme une séquence partielle σ, un ensemble de jobs non encore ordonnancés NS contenant un sous-ensemble PF de jobs à tester à la suite de σ. Une illustration est fournie sur la Figure 4.2.

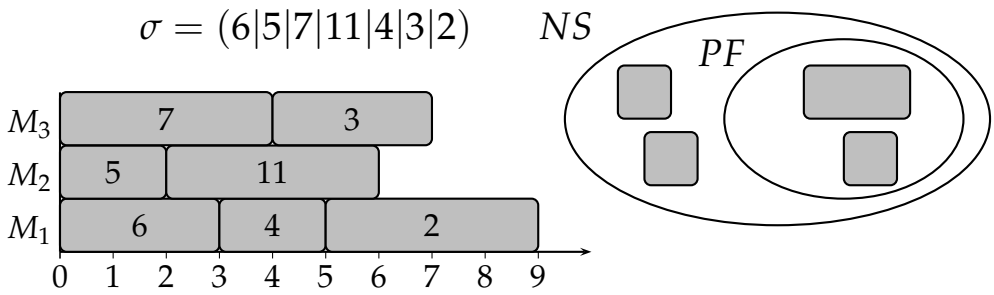


Figure 4.2 – Représentation d'un sommet de l'arbre de recherche.

Dans la suite, nous noterons S un sommet de l'arbre de recherche, σ la séquence des jobs ordonnancés au niveau du sommet S, NS les jobs non ordonnancés et PF les jobs qui doivent être testés en première position à la suite de σ. Avant les filtrages, on a $PF = NS$.

4.3 Intégration des règles de dominance

Nous abordons dans cette section l'utilisation des règles de dominance au sein de notre méthode arborescente.

4.3.1 Ordonnancements actifs et dominances locales

De par le choix de modélisation, la méthode génère et évalue l'ensemble des ordonnancements quasi-actifs. Or, on sait que l'étude des ordonnancements actifs est suffisante. Ainsi, si l'ordonnancement du job i à la suite de la séquence σ crée un temps d'inactivité qui peut être comblé par un autre job de NS, alors le job est éliminé de l'ensemble PF du sommet correspondant à σ. Ce filtrage permet de ne considérer que les ordonnancements actifs lors de l'exploration.

De la même manière, il est possible de restreindre davantage les ordonnancements à considérer, dans la mesure où les ordonnancements LOWS-actifs forment une classe dominante pour les quatre critères étudiés (voir Section 2.2.3). Lors de l'examen de S, si la séquence $(\sigma|i)$ n'est pas LOWS-active, le job i est retiré de l'ensemble PF du sommet S. Cette procédure est détaillée à l'aide de l'Algorithme 4.3. Le front d'adjacence du job i est composé des derniers jobs de chacune des machines de la séquence σ, que l'on peut connaître en $O(m)$. L'algorithme 4.3 fonctionne donc également en $O(m)$.

4.3.2 Dominance entre ordonnancements partiels

Nous décrivons dans cette section l'utilisation des règles de dominance RD_1, RD_m et RD_k basées sur les ordonnancements partiels (voir Section 2.3). Ces règles peuvent être intégrées à la méthode exacte de deux manières différentes : à l'aide d'une technique de *No Good Recording* ou dans une recherche locale. Rappelons enfin qu'un ordonnancement partiel généré lors de l'exploration peut-être représenté par une séquence.

4.3.2.1 *No Good Recording*

L'utilisation d'un *No Good Recording* [64] permet l'intégration des règles de dominance RD_1, RD_m et RD_k. Cela consiste à mémoriser tout au long de l'exploration certains états des sommets visités. Lors de l'examen d'un sommet de l'arbre de recherche, si un job i appartient à l'ensemble PF, on teste si la séquence $(\sigma|i)$ est dominée par un état mémorisé. Si tel est le cas, l'exploration du sommet est inutile et i est supprimé de l'ensemble PF. Lorsqu'un sommet de l'arbre a été exploré, on ajoute l'état correspondant à l'ensemble des états mémorisés.

Le stockage des états visités se fait à l'aide d'une table de hachage. Cette dernière nous permet d'accéder rapidement aux états qui nous intéressent, lors d'un test de dominance.

L'Algorithme 4.4 décrit la façon avec laquelle les dominances sont testées dans le *No Good Recording*.

Lors de l'utilisation d'un *No Good Recording*, le calcul d'une clé s'effectue à partir des jobs ordonnancés, qui doivent être triés par indices. Le calcul d'une clé coûte donc $O(n \log n)$. Pour savoir si deux états qui ont la même clé sont

Algorithme 4.3 : Algorithme LOWS-actif

Données : Une séquence $(\sigma|i)$ et un ensemble NS de job non encore ordonnancés

Résultat : Un booléen qui indique si la séquence est LOWS-active au niveau du job i

pour chaque *job* $j \in Adj(i)$ **faire**

  si $m(i) = m(j)$ **alors**

    Calculer la date Δ_j de disponibilité de la machine de i et j avant le job j;

    $dC \leftarrow C_i - C\big(j, C(i, \Delta_j)\big)$;

    $dF \leftarrow F(i, \Delta_i) + F(j, \Delta - j) - F(i, \Delta_j) - F(j, \Delta_i)$;

    si $non\big((dF < 0)\ ou\ (dC < 0)\ ou\ (dF = 0\ et\ dC = 0)\big)$ **alors**

      retourner *FAUX*

  sinon

    Calculer la date Δ_i de disponibilité de la machine $m(i)$ avant le job i;

    Calculer la date Δ_j de disponibilité de la machine $m(j)$ avant le job j;

    $min \leftarrow \min\big(C(i, \Delta_i), C(j, \Delta_j)\big)$; $max \leftarrow \max\big(C(i, \Delta_i), C(j, \Delta_j)\big)$;

    $min' \leftarrow \min\big(C(i, \Delta_j), C(j, \Delta_i)\big)$; $max' \leftarrow \max\big(C(i, \Delta_j), C(j, \Delta_i)\big)$;

    $dmin \leftarrow min - min'$; $dmax \leftarrow max - max'$;

    $dF \leftarrow F(i, \Delta_i) + F(j, \Delta - j) - F(i, \Delta_j) - F(j, \Delta_i)$;

    si $non\big((dF < 0)\ ou\ (dmin < 0)\ ou\ (dmax < 0)\ ou\ (dF = 0\ et\ dmin = 0\ et\ dmax = 0)\big)$ **alors**

      retourner *FAUX*

retourner *VRAI*

Algorithme 4.4 : Algorithme de *No Good Recording*

Données : Une séquence σ, une table de hachage $Hash_1$ et une table de hachage $Hash_m$

Résultat : Un booléen qui indique si la séquence σ est dominée ou non

pour chaque *séquence machine* $M_i(\sigma)$ **faire**
  pour chaque *séquence* $\sigma_1 \in Hash_1$ *composée des mêmes jobs que* $M_i(\sigma)$ **faire**
    si σ_1 *domine* $M_i(\sigma)$ *par* RD_1 **alors retourner** *VRAI*
  pour chaque *séquence* $\sigma_m \in Hash_m$ *composée des mêmes jobs que* σ **faire**
    si σ_m *domine* σ *par* RD_m **alors retourner** *VRAI*;
    Apparier les machines de σ et celles de σ_m;
    pour chaque *appariement* A **faire**
      si $A(\sigma_m)$ *domine* $A(\sigma)$ *par* RD_k **alors retourner** *VRAI*;
  retourner *FAUX*

à comparer, il faut également pouvoir comparer leurs jobs. Le plus simple est alors que les jobs soient triés, ce test coûte donc également $O(n \log n)$.

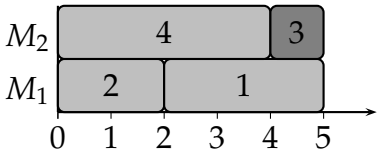
4.3.2.2 Recherche locale

Il est également possible d'intégrer les règles de dominance à la méthode exacte à l'aide d'une recherche locale. Afin de savoir si un job i peut être éliminé de l'ensemble PF, un ensemble de séquences $\mathcal{V}$ est généré à partir de la séquence $(\sigma(S)|i)$, où $\sigma(S)$ au sommet courant S. L'ensemble $\mathcal{V}$ est obtenu par exemple en échangeant le job i avec un job de σ ou en insérant ce dernier dans la séquence σ. Ces opérations permettent de former de nouvelles séquences qui composent $\mathcal{V}$. Si l'une de ces séquences domine $(\sigma(S)|i)$ selon RD_m ou RD_k, le job i est retiré de l'ensemble PF du sommet S. Dans notre cas, il y a de l'ordre de n séquences générées. La comparaison selon RD_m coûte $O(m)$, celle selon RD_k coûte $O(m)$ également, donc la recherche locale coûte $O(nm)$. Afin d'améliorer la qualité de ces séquences, l'Algorithme 4.10 est utilisé.

La Figure 4.3 montre une séquence σ et l'ensemble de séquences générées afin d'être comparées à la séquence σ. L'Algorithme 4.5 décrit en détail cette procédure.

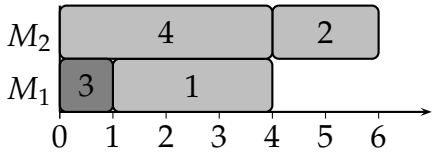
4.3.2.3 Utilisation de RD_1

La règle de dominance RD_1 (voir Section 2.3.1) caractérise une situation de dominance entre deux ordonnancements partiels sur une machine. Pour le

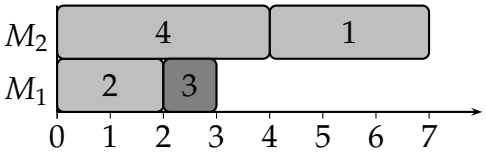


job i	1	2	3	4
r_i	1	0	0	0
p_i	3	2	1	4

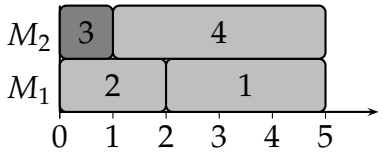
Séquence courante et données



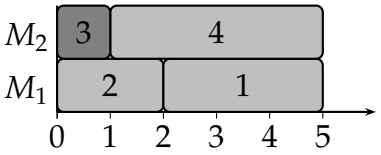
(a) échange de 2 et 3



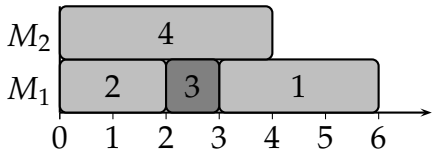
(b) échange de 1 et 3



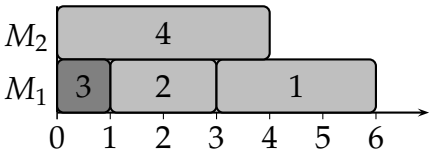
(c) échange de 4 et 3



(d) insertion de 3 avant 4



(e) insertion de 3 et 1



(f) insertion de 3 et 2

Figure 4.3 – Exemple de séquences générées lors d'une recherche locale.

Algorithme 4.5 : Algorithme de recherche locale

Données : Une séquence σ qui termine par le job i

Résultat : Un booléen qui indique si le voisinage de σ la domine ou non

pour chaque *job k de* σ *différent de* i **faire**
  Échanger les jobs k et i dans σ pour former une séquence σ';
  si σ' *domine* σ *par* RD_1, RD_m *ou* RD_k **alors**
    retourner *VRAI*
  Insérer le job i avant le job k pour former une séquence σ';
  si σ' *domine* σ *par* RD_1, RD_m *ou* RD_k **alors**
    retourner *VRAI*

retourner *FAUX*

No Good Recording, nous utilisons une table de hachage $Hash_1$ pour stocker les états déjà visités.Lorsqu'une séquence σ est étudiée, cette méthode teste pour chaque séquence machine $M_i(\sigma)$ s'il existe dans $Hash_1$ une séquence sur une machine qui la domine. Dans ce cas, la séquence σ est dominée selon RD_1. Lorsqu'une séquence doit être mémorisée, le *No Good Recording* reçoit un nouvel état composé des informations suivantes :

- les jobs de cette séquence,
- le coût de cette séquence,
- la date de fin de cette séquence.

Lorsqu'une séquence σ est testée selon la règle RD_1, il y a au plus m comparaisons effectuées. Avant chaque comparaison, il faut vérifier que les deux ensembles sont composés des mêmes jobs, ce qui coûte $O(n)$. Chaque comparaison coûte $O(1)$ dès lors que l'on a accès la somme des poids des jobs non ordonnancés en $O(1)$. Finalement, cette règle coûte $O(mn)$.

4.3.2.4 Utilisation de RD_m

De la même manière que RD_1, la règle de dominance RD_m (voir Section 2.3.2) peut être utilisée dans un *No Good Recording* ou dans une recherche locale. Le *No Good Recording* mis en place pour tester RD_m utilise également une table de hachage $Hash_m$. Lorsqu'une séquence doit être mémorisée par le *No Good Recording*, l'état suivant est ajouté à la table de hachage $Hash_m$:

- les jobs de cette séquence,
- le coût de cette séquence,
- les dates de fin des machines de cette séquence.

Afin de déterminer si une séquence est dominée selon la règle RD_m, nous savons qu'il faut calculer le décalage à droite minimal que subissent les jobs non ordonnancés, si l'on échange une séquence par une aure. L'Algorithme 4.6, qui détaille ce calcul, se comporte en $O(m)$.

Algorithme 4.6 : Calcul du décalage Δ de O_1 par rapport à O_2

Données : Deux ordonnancements partiels O_1 et O_2, où les machines sont indicées selon les dates de fin croissantes

Résultat : Δ

$\Delta \leftarrow 0$;
pour $j = 1$ *à* m **faire**
  $\delta = C(M_j(O_1)) - C(M_j(O_2))$;
  si $\delta > \Delta$ **alors** $\Delta \leftarrow \delta$;
retourner Δ

Lorsque σ_1 et σ_2 sont comparées, le test de dominance de l'une sur l'autre s'effectue en $O(m)$ si l'on connaît $\sum_{i \in NS} w_i$ en $O(1)$.

4.3.2.5 Utilisation de RD_k

La règle de dominance RD_k opère sur des sous-ensembles de machines parmi deux séquences, composés des mêmes jobs. Cette dernière n'utilise pas de *No Good Recording* spécifique, elle utilise celui de RD_m. Le point délicat dans la mise en place de cette méthode de filtrage provient du fait qu'*a priori*, on pourrait décider de comparer une séquence à n'importe quelle autre séquence. Ce n'était pas le cas avec les règles de dominance RD_1 et RD_m, où l'on comparait seulement des séquences (sur une ou sur m machines) composées des mêmes jobs. Pour RD_k, il faut seulement qu'il existe un sous-ensemble de machines sur une séquence σ_1 et un autre sur une séquence σ_2 composés des mêmes jobs pour que nous puissions tester la règle. Néanmoins, l'utilisation de cette règle à l'aide d'un *No Good Recording* ne se fait dans la méthode exacte qu'avec des séquences qui sont composées des mêmes jobs. Ainsi, si l'on cherche à savoir si une séquence σ est dominée selon la règle RD_k, elle sera uniquement comparée aux séquences qui sont composées des mêmes jobs que σ. De plus, étant données deux séquences σ_1 et σ_2, il faut au préalable trouver un sous-ensemble de machines pour chaque séquence qui soit composé des mêmes jobs que l'autre. Nous proposons pour cela l'Algorithme d'appariement 4.7. L'idée consiste à colorer les machines des deux ordonnancements de sorte qu'une couleur corresponde à un appariement. On procède de la manière suivante : tant qu'il existe des machines non colorées, on colore une machine M avec une nouvelle couleur C et on mémorise dans P tous les jobs qui s'exécutent sur M. Pour chaque job j de P, on consulte l'autre machine M_2 sur laquelle j s'exécute, on mémorise dans P tous les jobs de M_2 qui n'ont pas encore été mémorisés et on colore M_2 avec la couleur C. On itère tant que l'ensemble des jobs mémorisés P est non vide. Lorsque P est vide, on a construit un appariement représenté par la couleur C. Dans le pire des cas, comme on sait que les jobs de σ_1 sont les mêmes que ceux de σ_2, toutes les machines ont la même couleur. Autrement dit, il existe au moins un appariement qui implique l'ensemble des machines des deux ordonnancements. Une fois la coloration des machines obtenue, la règle RD_k permet de dire si l'une des séquences est dominée ou non. Si l'on suppose que l'on accède aux jobs d'une machine en $O(1)$ à l'aide d'une structure adaptée, cet algorithme a une complexité linéaire, puisqu'un job ne peut être empilé plus d'une fois au cours de l'algorithme.

On suppose ici que $n > m$. Dans l'algorithme d'appariement, une machine est visitée une fois et une seule. Globalement, la complexité de cet algorithme est donc linéaire en fonction du nombre de jobs ordonnancés. Cet algorithme coûte donc $O(n)$. Une fois les appariements établis, la règle RD_k teste pour chacun d'eux si une séquence est dominée ou non. Comme une machine appartient au plus à un appariement, la complexité du test de la règle de dominance RD_k est globalement de $O(n)$.

Algorithme 4.7 : Appariement des machines pour RD_k par coloration

Données : 2 ordonnancements O_1 et O_2 contenant les mêmes jobs

Résultat : Un ensemble d'appariements pour O_1 et O_2, représentés par une coloration des machines

Initialiser 2 tableaux $mach_1$ et $mach_2$ de sorte que $mach_1[i]$ (*resp.* $mach_2[i]$) corresponde au numéro de machine du job i dans O_1 (*resp.* O_2);
Initialiser à 0 un tableau $coul$ de taille $2m$ où $coul[i]$ représentera la couleur de la machine i dans O_1 si $i \leq m$ et la couleur de la machine $i - m$ dans O_2 si $i > m$;
Initialiser à 0 un tableau $visit$ où $visit[i]$ indiquera si le job i a déjà été visité ou non;
$j \leftarrow 1$;
$couleur \leftarrow 1$;
tant que $j \leq 2m$ **faire**
    $coul[j] \leftarrow couleur$;
    Empiler dans P tous les jobs de la machine j;
    tant que P *est non vide* **faire**
        $x \leftarrow$ dépiler (P);
        $visit[x] \leftarrow VRAI$;
        si $coul\left[mach_1[x]\right] = 0$ **alors**
            Empiler dans P tous les jobs i de la machine $mach_1[x]$ tels que $visit[i] = FAUX$;
            $coul\left[mach_1[x]\right] \leftarrow couleur$;
        si $coul\left[mach_2[x]\right] = 0$ **alors**
            Empiler dans P tous les jobs i de la machine $mach_2[x]$ tels que $visit[i] = FAUX$;
            $coul\left[mach_2[x]\right] \leftarrow couleur$;
    $couleur \leftarrow couleur + 1$;
    tant que $coul[j] \neq 0$ *et* $j \leq 2m$ **faire**
        $j \leftarrow j + 1$;

4.3.3 Dominance entre jobs non ordonnancés

À chaque sommet S de l'arbre de recherche correspond un ordonnancement partiel P et un ensemble de jobs non ordonnancés NS. La règle de dominance portant sur les jobs non ordonnancés (voir Section 2.4) est utilisée pour tenter de réduire l'ensemble PF. Pour chaque job k appartenant PF, on vérifie qu'aucun job j appartenant à NS ne domine k par rapport à P. Lorsqu'un tel job j existe, le job k est retiré de l'ensemble PF. Ce test coûte $O(n^3)$, puisque l'on va tester au plus n jobs dans le rôle de k, avec à chaque fois n jobs dans le rôle de j. Enfin, une fois k et j fixés, conclure que k est dominé par j ou non coûte $O(n)$, à cause du calcul de ω_A et de ω_B.

4.4 Borne supérieure

Nous présentons ici l'heuristique utilisée au sein de la méthode exacte pour initialiser la valeur de la borne supérieure. Cette initialisation est effectuée par un algorithme glouton, qui est présenté dans la Section 4.4.1. Deux algorithmes d'amélioration proposés par Jouglet [42] sont ensuite décrits. S'appuyant sur les règles de dominances présentées dans le Chapitre 2, ils permettent d'améliorer la solution construite par l'algorithme glouton. Nous avons choisi d'utiliser une heuristique assez simple qui intègre les résultats de dominance présentés dans le Chapitre 2. On peut néanmoins citer plusieurs travaux relatifs aux heuristiques. Sur une machine, on peut mentionner les travaux de Hariri et Potts qui ont proposé une heuristique pour le problème $1|r_i|w_iC_i$ [38]. Pour les cas non pondérés $1|r_i|\sum C_i$ et $1|r_i|\sum T_i$, Chu a proposé des algorithmes gloutons [25, 26]. Par ailleurs, Jouglet a présenté des heuristiques pour les problèmes $1|r_i|\sum C_i$, $1|r_i|\sum w_iC_i$, $1|r_i|\sum T_i$ et $1|r_i|\sum w_iT_i$ basées notamment sur une utilisation efficace de règles de dominance [42]. Dans le cas de machines parallèles identiques, Bilge *et al.* ont mis en place une méthode tabou pour le problème $Pm||T_i$ [16]. Une heuristique a également été proposée par Alidaee et Rosa pour le problème $R||\sum w_iT_i$ [3].

4.4.1 Algorithme glouton

Nous allons voir dans cette section un algorithme glouton qui peut s'appliquer aux quatre critères que nous étudions. Cet algorithme contruit des ordonnancements actifs. Partant d'un ensemble NS de jobs à ordonnancer, il construit pas à pas un ordonnancement complet. À chaque étape, une règle de décision permet d'ordonnancer un job x à la suite de l'ordonnancement en cours de construction. Dans cet algorithme se trouve la commande « *Améliorer P* », cette dernière fait référence aux algorithmes d'amélioration qui sont présentés dans la Section 4.4.2.

L'Algorithme 4.8 *HP* [25, 42] construit des ordonnancements en choisissant le job de plus grande priorité par rapport à une règle de priorité PR. À chaque itération, on ordonnance un job x de l'ensemble PF de plus grande priorité. En cas d'égalité, on choisit le job qui peut commencer le plus tôt.

Algorithme 4.8 : Algorithme *Highest Priority*

Données : Un ensemble NS de n jobs à ordonnancer

Résultat : Une valeur V_{hp}

$t \leftarrow 0$; $P \leftarrow [\,]$;

tant que $NS \neq \emptyset$ **faire**

$PF \leftarrow \left\{ j \in NS / r_j < \min_{\{l \in NS\}} \{\max(t, r_l) + p_l\} \right\}$;

Choisir le job $x \in PF$ avec une règle de priorité PR. En cas d'égalité, choisir x tel que $\max(t, r_x) = \min_{\{l \in PF\}} \{\max(t, r_l)\}$. En cas d'égalité, choisir le job x de plus faible indice;

$NS \leftarrow NS \setminus \{x\}$; $P \leftarrow (P|x)$;

Améliorer P;

Calculer la date t minimale de disponibilité d'une machine après P;

retourner $V_{hp} \leftarrow F(P)$

Nous présentons maintenant la règle de priorité utilisée à l'intérieur de cet algorithme. C'est une adaptation de la règle $CPRTWT$ [42] pour le cas à machines parallèles.

Notons $\delta(P)$ la date à laquelle une machine devient libre après l'ordonnancement partiel P et NS l'ensemble des jobs non encore ordonnancés à cette date. Notons $(P|x)$ l'ordonnancement partiel formé de l'ordonnancement partiel P ainsi que du job x, ordonnancé le plus tôt possible à la suite de P. Nous rappelons que $F(j,t)$ indique le coût du job j s'il est exécuté à la date $\max(r_j, t)$. La règle $CPRTWT$ [42] adaptée, qui peut être utilisé pour les quatre critères, s'exprime comme suit :

$$\max_{\{J_j \in NS\}} \left\{ \sum_{\{J_l \in NS\}} c_{j,l}(P) \right\},$$

avec

$$c_{j,l} = \begin{cases} 1 & \text{si } F\big(j, \delta(P)\big) + F\big(l, \delta(P|j)\big) \leq F\big(l, \delta(P)\big) + F\big(j, \delta(P|l)\big), \\ 0 & \text{sinon.} \end{cases}$$

Lorsque l'on doit ordonnancer des jobs à la suite de l'ordonnancement partiel P, la règle de priorité $CPRTWT$ adaptée compare le coût des jobs j et l lorsque j est ordonnancé avant l avec le coût des jobs j et l lorsque j est ordonnancé après l. Si le coût dans le premier cas est inférieur au coût dans le second cas, $c_{j,l} = 1$. Sinon, $c_{j,l} = 0$. Ainsi, en balayant l'ensemble des jobs l pour chaque job j, le job j qui maximise $c_{j,l}$ paraît plus intéressant que les autres. C'est donc celui qui est choisi pour suivre l'ordonnancement partiel P.

4.4.2 Algorithmes d'amélioration

Nous avons vu dans le Chapitre 2 plusieurs règles de dominance. Nous avons tout d'abord étudié une classe dominante d'ordonnancements appelés LOWS-actifs (voir Section 2.2.3). Nous avons également proposé des règles de dominance portant sur les jobs ordonnancés dans la Section 2.3. Nous présentons ici deux algorithmes proposés par Jouglet [42] qui permettent d'améliorer la qualité des solutions construites par un algorithme glouton. Ces deux algorithmes utilisent les résultats de dominance précedemment cités. Dans les deux algorithmes, on part d'une séquence $\sigma = ([1], [2], \ldots, [i])$.

Dans l'Algorithme 4.9, le dernier job i de la séquence σ est étudié. Pour chaque job j appartenant au front d'adjacence de i, on teste si σ est LOWS-active sur ces deux jobs. Si c'est le cas, rien n'est modifié. Dans le cas contraire, la séquence résultat est construite à partir de σ en échangeant les positions des jobs i et j. Cet algorithme a une complexité en $O(m)$.

Algorithme 4.9 : Algorithme *make-lows*

Données : Une séquence $\sigma = ([1], [2], \ldots, [i])$

Résultat : Une séquence σ_2

pour chaque *job* $j \in Adj([i])$ **faire**
  si *l'ordonnancement partiel obtenu à partir de* σ *n'est pas LOWS-actif sur les jobs* j *et* $[i]$ **alors**
    $\sigma_2 \leftarrow echange(\sigma, j, [i])$;
    retourner σ_2;

retourner σ;

Dans l'Algorithme 4.10, plusieurs séquences sont générées à partir de σ. Ces dernières sont créées en échangeant le dernier job $[i]$ de σ avec un autre job de la séquence, ou bien en insérant $[i]$ en deux jobs de σ. Chaque nouvelle séquence est créée de manière incrémentale. À chaque pas, l'Algorithme 4.9 est lancé sur le dernier job de la nouvelle séquence en construction. Cela permet d'améliorer la qualité de la séquence obtenue. Dès lors qu'une séquence construite est « meilleure » que σ selon RD_m ou selon RD_1 (voir Section 2.3), l'algorithme s'arrête en retourne cette séquence.

4.5 Bornes inférieures

4.5.1 Adaptation aux ordonnancements partiels

Les différentes bornes inférieures présentées dans le Chapitre 3 peuvent être utilisées au sein du *Branch&Bound*. Néanmoins, le calcul d'une borne inférieure à un sommet quelconque de l'arbre de recherche doit pouvoir se faire avec des dates de disponibilité différentes pour les machines. Si

Algorithme 4.10 : Algorithme *make-better*

Données : Une séquence $\sigma = ([1], [2], \ldots, [i])$

Résultat : Une séquence σ_2

$x \leftarrow 1$; *better* $\leftarrow FAUX$;

tant que $x \leq i - 1$ *et better* $= FAUX$ **faire**

  Échanger les jobs $[i]$ et $[x]$ dans σ pour former

  $\sigma_2 = (\{1\}, \{2\}, \ldots, \{i\})$;

  pour $k = 2$ *à* i **faire**

    make-lows$((\{1\}, \{2\}, \ldots, \{k\}))$;

  si σ_2 *est meilleure que* σ **alors**

    better $\leftarrow VRAI$;

  Insérer le job $[i]$ avant $[x]$ dans σ pour former

  $\sigma_2 = (\{1\}, \{2\}, \ldots, \{i\})$;

  pour $k = 2$ *à* i **faire**

    make-lows$((\{1\}, \{2\}, \ldots, \{k\}))$;

  si σ_2 *est meilleure que* σ **alors**

    better $\leftarrow VRAI$;

si *better* $= VRAI$ **alors**

  retourner σ_2;

sinon

  retourner σ;

certaines bornes inférieures peuvent être calculées sous cette hypothèse, d'autres bornes ne peuvent plus être appliquées. Pour parer à cette limitation, nous proposons de réduire « artificiellement », lors du calcul de la borne inférieure, les dates de disponibilité des machines à la plus petite d'entre elles. On peut justifier cette relaxation en remarquant que le nombre de machines est bien souvent faible par rapport au nombre de jobs, ce qui tend à limiter l'impact de cette relaxation.

4.5.2 Choix des bornes inférieures selon les critères

L'examen des résultats numériques des bornes inférieures présenté dans la Section 3.6 nous a conduit à faire les choix suivants. Pour la somme des dates de fin, la méthode arborescente utilise la borne inférieure $lb_{no-release-subsets}$. Cette borne inférieure s'adapte aux dates de diponibilité différentes des machines. Pour le retard total, la borne utilisée est $lb_{[\,]combo}$. Cette dernière est obtenue à partir des bornes inférieures $lb_{[\,]split}$ et $lb_{[\,]resort}$. Seule la première de ces deux bornes s'adapte facilement aux dates de diponibilité différentes des machines. Pour les critères pondérés, c'est la borne $lb_{flow'}$ qui est utilisée. Ces choix sont motivés par le bon compromis qualité / temps que ces deux bornes inférieures semblent offrir. En effet, les bornes inférieures étant calculées à chaque sommet de l'arbre de recherche, une borne trop coûteuse en temps de calcul alourdirait considérablement la méthode.

4.6 Borne supérieure du *makespan* d'un ordonnancement actif

Nous décrivons maintenant le calcul d'une borne supérieure de la date de fin d'un ordonnancement actif quelconque, que nous utilisons en tant qu'horizon. Cela nous est utile au sein de certaines règles de dominance ainsi qu'au sein de certaines bornes inférieures. Il a été prouvé récemment que le problème de maximisation du *makespan* parmi les ordonnancements actifs était NP-difficile [4]. L'heuristique que nous utilisons fonctionne de la manière suivante. On dispose d'un ensemble N de jobs à ordonnancer. On va retirer des jobs à l'ensemble N au fur et à mesure de la procédure. Les jobs sont indicés de sorte que $r_1 \leq r_2 \leq \ldots \leq r_n$. Ensuite, on fait le raisonnement suivant : entre les dates r_i et r_{i+1}, on regarde les jobs disponibles de plus grandes durées et on compte le nombre minimal n_i de ces jobs que l'on va placer sur une machine sans dépasser la date r_{i+1}. Comme on a choisi les jobs les plus grands, on sait que n_i est le cas le plus défavorable (minimal). On sait donc qu'on placera au minimum $m \cdot n_i$ jobs dès lors qu'on dispose de m machines. On retire alors que les $m \cdot n(i)$ jobs disponibles de plus courtes durées de l'ensemble N (ou moins s'il n'y a pas $m \cdot n(i)$ jobs disponibles à r_i) et on passe à l'intervalle $[r_{i+1}, r_{i+2}]$. Lorsque l'on se trouve sur l'intervalle $[r_n, +\infty[$, la valeur $r_n + \frac{1}{m}\left(\sum_{i \in N} p_i - \max_{i \in N} p_i\right) + \max_{i \in N} p_i$ est une borne supérieure du *makespan* de tout ordonnancement actif. Cette der-

nière formule mérite quelques explications. Lorsque l'on a atteint r_n, tous les jobs non encore ordonnancés sont disponibles. Autrement dit, on se retrouve face à un problème sans dates de disponibilité. On cherche alors à ordonnancer ces jobs de sorte que l'ordonnancement soit actif et que le *makespan* soit maximal. Si on appelle i^* le job de N de plus grande durée, le cas le plus défavorable (celui qui maximise le *makespan*) est alors celui représenté sur la figure 4.4. La proposition 4.2 prouve ce résultat :

Proposition 4.2. *Si l'on dispose de n jobs disponibles à la date 0, et si l'on note* $p_i^* = \max_{k\in N} p_k$, *alors la quantité* $\frac{1}{m}(\sum_{i\in N} p_i - p_i^*) + p_i^*$ *est une borne supérieure du makespan de tout ordonnancement actif.*

Démonstration. Supposons qu'il existe un ordonnancement actif O qui termine à une date C_O telle que $C_O > \frac{1}{m}(\sum_{k\in N} p_k - p_i^*) + p_i^*$. Il existe au moins un job qui termine son exécution à la date C_O. Notons i ce job. Nous allons calculer le temps d'occupation des machines $\omega(O)$ de l'ordonnancement O. La machine $m(i)$, sur laquelle s'exécute le job i, est occupée entre 0 et C_O, puisque l'ordonnancement O est actif et que tous les jobs sont disponibles à la date 0. Par ailleurs, les autres machines doivent terminer après la date $C_O - p_i$, sans quoi l'ordonnancement ne serait pas actif puisque l'on pourrait avancer l'exécution du job i sans retarder celle d'un autre job. Les machines autres que $m(i)$ sont donc occupées au minimum jusqu'à $C_O - p_i$. Par définition, on a

$$\omega(O) = \sum_{j\in[1,m]} \omega\big(M_j(O)\big).$$

Donc

$$\omega(O) = \omega\big(M_{m(i)}(O)\big) + \sum_{m(j)\neq m(i)} \omega\big(M_{m(j)}(O)\big).$$

Donc

$$\omega(O) \geq C_O + (m-1)(C_O - p_i),$$

ce qui entraîne que

$$\omega(O) \geq mC_O - (m-1)p_i.$$

Or, on a supposé que

$$C_O > \frac{1}{m}(\sum_{k\in N} p_k - p_i^*) + p_i^*,$$

donc, on peut écrire

$$\omega(O) > \sum_k p_k - p^* + mp_i^* - (m-1)p_i.$$

D'où

$$\omega(O) > \sum_k p_k + (m-1)(p_i^* - p_i).$$

Or, par définition de p_i^*, on sait que $p_i^* - p_i \geq 0$, on obtient finalement :

$$\omega(O) > \sum_k p_k.$$

Cette situation est impossible, ce qui conclut notre raisonnement par l'absurde. □

L'algorithme 4.11 décrit en détail l'ensemble du calcul d'une borne supérieure du *makespan* d'un ordonnancement actif.

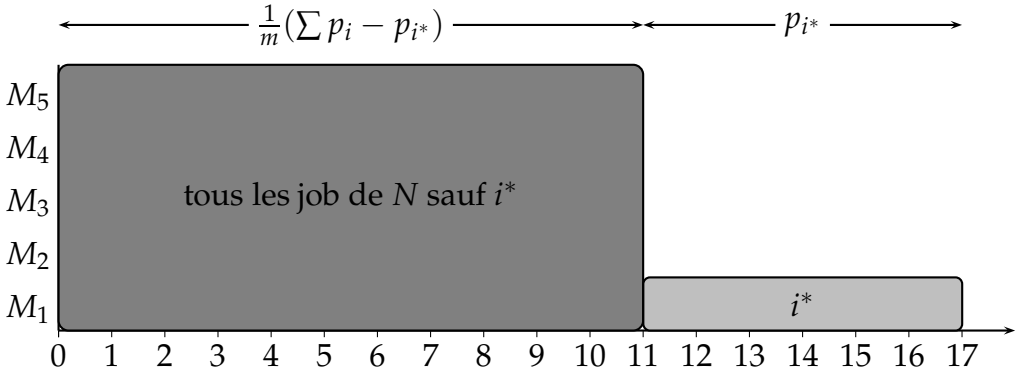


Figure 4.4 – Cas le plus défavorable après r_n (ici nulle).

4.7 Algorithme de *Branch&Bound*

Nous présentons maintenant l'algorithme de *Branch&Bound* utilisé pour la résolution exacte des quatre problèmes étudiés.

4.7.1 Stratégie d'exploration

Nous avons choisi d'adopter une stratégie en profondeur d'abord. Cela nous permet notamment de mettre à jour la valeur des bornes rapidement. Par ailleurs, lors de l'exploration d'un sommet et une fois que les règles de dominance ont filtré certains jobs, nous choisissons de tester en premier le job de plus faible durée.

4.7.2 Description de l'algorithme

L'algorithme 4.12 est décrit précisément dans cette section. La première étape consiste à initialiser la borne supérieure UB. Pour ce faire, l'algorithme glouton présenté dans la Section 4.4.1 est utilisé. L'algorithme d'amélioration

Algorithme 4.11 : Calcul d'une borne supérieure du *makespan* d'un ordonnancement actif

Données : Un ensemble de jobs N indicés selon les dates de disponibilité

Résultat : Une borne supérieure H

$i \leftarrow 1$;

tant que $i < n$ **faire**

Calculer le nombre $n(i)$ de jobs de N qu'on peut placer entre r_i et r_{i+1}. Pour cela, choisir les plus grands jobs disponibles de N;

Retirer les $m \cdot n(i)$ jobs disponibles de plus faibles durées de N (ou moins lorsque le nombre de jobs disponibles à r_i est strictement inférieur à $m \cdot n(i)$);

$i \leftarrow i + 1$;

retourner $r_n + \frac{1}{m}\left(\sum_{i \in N} p_i - \max_{i \in N} p_i\right) + \max_{i \in N} p_i$;

make-better est activé. Ensuite, l'examen des sommets de l'arbre de recherche commence, avec une stratégie en profondeur d'abord. Un sommet S est représenté par un triplet (σ, NS, PF). L'examen d'un sommet S consiste à examiner l'ensemble de ses fils. Ce dernier est obtenu à partir de l'ensemble PF, qui indique quels sont les jobs qui doivent être ordonnancés à la suite de σ. Un fils du sommet S représente la séquence $(\sigma|i)$, où un job non ordonnancé i a été ordonnancé à la suite de la séquence σ du sommet S. Avant d'examiner le sommet fils correspondant à la séquence $(\sigma|i)$, plusieurs tests sont lancés pour savoir si cet examen peut être évité. Ces tests correspondent à l'appel $Filtrer(Fils)$ dans l'algorithme. Voici l'ordre dans lequel ils sont évalués :

1. borne inférieure triviale ;
2. recherche locale.

La borne inférieure triviale calcule le coût des jobs de l'ensemble NS du sommet $(\sigma|i)$ lorsqu'ils sont ordonnancés à leur date de disponibilité corrigée , *i.e.* $\min(r_i, t)$, où t représente la date minimale de disponibilité d'une machine à la suite de la séquence $(\sigma|i)$.

Lorsque ces tests n'ont pas éliminé un sommet $(\sigma|i)$, ce dernier doit être examiné. Une nouvelle série de tests détermine les jobs de NS qui appartiendront à PF, et ceux qui n'appartiendront pas à PF. C'est l'appel $FiltrerPF(Fils)$ qui réalise ces opérations. Voici cette série de tests :

1. dominance des ordonnancements actifs ;
2. dominance des ordonnancements LOWS-actifs ;
3. *No Good Recording* et RD_1 ;
4. *No Good Recording* et RD_m ;
5. borne inférieure ;

6. *No Good Recording* et RD_k ;
7. dominance sur les jobs non ordonnancés.

La valeur de la borne supérieure est mise à jour tout au long de l'algorithme. L'algorithme termine lorsque tous les sommets ont été visités.

Algorithme 4.12 : Algorithme de *Branch&Bound*

Données : Un ensemble de jobs N
Résultat : Une valeur V^*

Initialiser une pile *Sommets*;
Initialisation de la borne supérieure : $UB \leftarrow HP(N)$;
$Racine \leftarrow (\{\}, N, N)$;
FiltrerPF(Racine);
Empiler(*Racine*,*Sommets*);
tant que *non(pilevide(Sommets))* **faire**
  $S = (\sigma, NS, PF) \leftarrow$tete$(Sommets)$;
  si $PF \neq \emptyset$ **alors**
    $i \leftarrow PF[1]$;
    $Fils \leftarrow ((\sigma|i), NS \setminus \{i\}, NS \setminus \{i\})$;
    si $Filtrer(Fils) \neq VRAI$ **alors**
      $FiltrerPF(Fils)$;
      Empiler(Fils,Sommets);
    sinon
      $PF \leftarrow PF \setminus \{i\}$;
  sinon
    $S = (\sigma, NS, PF) \leftarrow$dépiler$(Sommets)$;
    si $NS = \emptyset$ **alors**
      $UB \leftarrow \min(UB, F(\sigma))$;
      $V^* \leftarrow \min(V^*, F(\sigma))$;
    Mettre à jour les *No Good Recording* avec σ;
retourner V^*

4.8 Résultats numériques

Nous présentons dans cette section les résultats numériques préliminaires de la méthode exacte que nous avons proposée. Ces résultats sont rapportés critère par critère. Nous présentons également des résultats permettant de mesurer le rôle et l'efficacité des règles de dominance.

4.8.1 Schémas de génération des instances de test

Pour la somme des dates de fin et la somme pondérée des dates de fin, nous avons adapté les schémas de Hariri et Potts [38] et ceux de Belouadah, Posner et Potts [13]. Ces schémas dépendent d'un paramètre R. Lorsque n, m et R sont fixés, cinq instances sont générées : les valeurs de p_i, r_i et w_i suivent des distributions uniformes. Les valeurs de p_i sont distribuées dans $[1, 100]$, celles de w_i dans $[1, 10]$ et celles de r_i dans $[0, 50.5nR/m]$. Le paramètre R peut prendre dix valeurs différentes dans $\{0.2, 0.4, 0.6, 1.0, 1.25, 1.5, 1.75, 2.0, 3.0\}$. Il y a donc 50 instances pour chaque couple de valeur (n, m).

Pour le retard total et le retard total pondéré, nous avons adapté les schémas de Chu [24] ainsi que ceux d'Akturk et Ozdemir [2] pour le problème à une machine. Pour chaque taille d'instance n, et pour chaque nombre de machines m, m générations de $\lfloor \frac{n}{m} \rfloor$ jobs et éventuellement une génération de $n - m * \lfloor \frac{n}{m} \rfloor$ jobs sont effectuées pour former l'instance. Ces générations sont contrôlées par deux paramètres α et β. Les valeurs de r_i, de p_i, de d_i et de w_i (dans le cas pondéré) sont générées selon des distributions uniformes : p_i dans $[1, 100]$, w_i dans $[1, 10]$, r_i dans $[0, \alpha \sum p_i]$ et $d_i - r_i + p_i$ dans $[0, \beta \sum p_i]$. Le paramètre α prend quatre valeurs différentes $\{0, 0.5, 1, 1.5\}$ et le paramètre β trois valeurs dans $\{0.05, 0.25, 0.5\}$. Lorsque n, m, α et β sont fixées, 10 instances sont créées. Finalement, il y a 120 instances pour chaque couple de valeurs (n, m).

4.8.2 Résultats de règles de dominance

Nous présentons dans cette section des résultats numériques relatifs aux règles de dominance. Ces tests ont été effectués sur des instances de 10 jobs, avec 2, 3, 4 ou 5 machines. Plusieurs configurations ont été testées, afin de déterminer les méthodes de filtrage les plus efficaces. Ces résultats sont présentés dans deux tableaux distincts. Le Tableau 4.2 présente les résultats pour la somme des dates de fin et la somme pondérée des dates de fin. Le Tableau 4.3 rapporte les résultats pour le retard total et le retard total pondéré. La configuration « de base » utilise le filtrage des ordonnancements actifs et la borne inférieure triviale. Les résultats de cette configuration correspondent aux lignes « rien ». Les lignes suivantes rapportent les résultats de cette configuration, à laquelle une ou plusieurs méthode(s) de filtrage ont été ajoutées. Les sigles présents dans ces lignes sont expliqués dans le Tableau 4.1. Pour chaque configuration, nous rapportons le temps de calcul en millisecondes (colonne « cpu ») et le nombre de sommets visités par la méthode (colonne « sommets »).

Le comportement des méthode de filtrage semble être qualitativement identique pour les quatre critères. Nous dressons donc une analyse commune aux quatre critères.

Lorsqu'une seule méthode de filtrage est ajoutée à la configuration, on peut apprécier le potentiel de chacune des méthodes. Si le filtrage des ordonnancements qui ne sont pas LOWS-actifs est utilisé, on peut constater que le

sigle	méthode de filtrage
lows	filtrage des ordonnancements LOWS-actifs
rd_1	règle RD_1 avec *No Good Recording*
rd_m	règle RD_m avec *No Good Recording*
rd_k	règle RD_k avec *No Good Recording*
ns	règle sur les jobs non ordonnancés
local	recherche locale

Tableau 4.1 – Sigles des résultats des méthodes de filtrage

nombre de sommets diminue sensiblement. Néanmoins, ce filtrage requiert un certain temps de calcul, puisque cette configuration prend globalement plus de temps que la configuration de base. Si l'on active uniquement la règle de dominance RD_1 à l'aide d'un *No Good Recording*, les résultats sont de mauvaise qualité. En effet, alors que le nombre de sommet reste sensiblement le même, le temps de calcul est multiplié par un facteur oscillant entre 3 et 7. La règle RD_k seule obtient des résultats similaires à la règle de dominance RD_1. À l'inverse, la règle de dominance RD_m utilisée avec un *No Good Recording* fournit d'excellents résultats. Elle permet de diviser le nombre de sommets visités d'un facteur 20 environ. De plus, le temps est également réduit de manière significative. La règle portant sur les jobs non ordonnancés apporte des résultats assez concluants : elle augmente légèrement le temps de calcul mais le nombre de sommets visités diminue de moitié. Enfin, la méthode de recherche locale ne semble pas très intéressante, puisqu'elle filtre très peu de sommets tandis qu'elle demande un temps de calcul supplémentaire assez important.

La ligne « tout » rapporte les résultats lorsque toutes ces méthodes de filtrage sont utilisées conjointement. On constate que le nombre de sommets est divisé d'un rapport oscillant autour de 50 par rapport à la configuration de base. Le temps de calcul est divisé par 3 environ. Les lignes situées audessus de celle-ci rapportent les résultats de l'ensemble des méthodes de filtrage sauf une. Lorsque seul le filtrage LOWS-actif n'est pas utilisé, quelques sommets de plus sont visités, mais le temps de calcul diminue. Si tout est utilisé sauf la règle de dominance RD_1, le nombre de sommets ne diminue par rapport à la configuration où tout est utilisé. Par ailleurs, le temps de calcul est pratiquement divisé par 2. On peut dresser le même bilan concernant la règle de dominance RD_k : il n'y a pas plus de sommets visités lorsqu'elle n'est pas utilisée et elle demande un temps de calcul non négligeable. Ces résultats concernant les règles RD_1 et RD_m sont en accord avec ceux obtenus quand seules ces méthodes sont utilisées. L'utilisation de ces deux règles de dominance ne semble pas intéressante, dans la mesure où elle semble coûter et ne rien apporter. En revanche, lorsque la configuration complète est privée de l'utilisation de la règle RD_m, on constate aisément que le nombre de sommets et le temps de calcul sont multipliés par un facteur oscillant entre 10 et 20 suivant les critères. Une fois encore, ce constat corrobore les résultats

obtenus par RD_m seule, qui semble très efficace au sein de la méthode exacte. Lorsque la règle sur les jobs non ordonnancés n'est pas utilisée, on constate que le nombre de sommets augmente de manière visible. Le temps de calcul semble à peu près équivalent, même si les résultats varient d'un critère à l'autre. Enfin, la méthode de recherche locale semble, comme les règles RD_1 et RD_k, complètement inefficace. En effet, elle ne permet pas de filter des sommets et elle requiert un temps de calcul important.

Étant donnés le temps de calcul requis et l'absence de résultats, nous avons décidé de ne pas utiliser de recherche locale dans la méthode exacte. Concernant les règles RD_1 et RD_k, nous avons lancé quelques tests supplémentaires. Les résultats de ces tests apparaissent sur les trois dernières lignes de chaque tableau. La dernière ligne indique la configuration suivante : filtrage des ordonnancements LOWS-actifs, règle de dominance RD_m et règle de dominance sur les jobs non ordonnancés. C'est la configuration que nous décidé d'utiliser pour les résultats finaux. Lorsque l'on ajoute à cette configuration la règle de dominance RD_1, on voit que les résultats sont équivalents, au niveau du nombre de sommets comme au niveau du temps de calcul. Il en va de même pour la règle de dominance RD_k.

	$\sum C_i$		$\sum w_i C_i$	
méthodes	**cpu**	**sommets**	**cpu**	**sommets**
rien	554	5933	660	9363
lows	693	4096	1417	6400
rd_1	1362	5597	2472	8343
rd_m	74	244	173	491
rd_k	1323	4251	2595	6691
ns	922	3258	1404	4611
local	1844	5521	3652	8237
tout sauf lows	94	162	147	213
tout sauf rd_1	79	144	131	206
tout sauf rd_m	872	2005	1668	3002
tout sauf rd_k	77	144	129	204
tout sauf ns	77	214	227	426
tout sauf local	71	145	114	206
tout	126	144	205	204
lows+rd_1+rd_m+ns	97	145	143	206
lows+rd_m+rd_k+ns	100	145	147	210
lows+rd_m+ns	90	145	131	210

Tableau 4.2 – Résultats des méthodes de filtrage pour $Pm|r_i|\sum C_i$ et $Pm|r_i|\sum w_i C_i$

	$\sum T_i$		$\sum w_i T_i$	
méthodes	**cpu**	**sommets**	**cpu**	**sommets**
rien	5292	69763	1702	24799
lows	5977	40514	5628	14659
rd_1	28907	42121	12856	16354
rd_m	553	1960	611	1306
rd_k	24572	46227	13082	18886
ns	9088	44615	5916	14131
local	21869	44871	11275	16008
tout sauf lows	826	1234	406	432
tout sauf rd_1	718	1208	371	422
tout sauf rd_m	23528	25155	8769	8823
tout sauf rd_k	642	1209	355	423
tout sauf ns	937	1607	592	767
tout sauf local	529	1213	328	426
tout	1175	1208	551	422
lows+rd_1+rd_m+ns	560	1215	367	427
lows+rd_m+rd_k+ns	687	1217	403	429
lows+rd_m+ns	468	1219	339	429

Tableau 4.3 – Résultats des méthodes de filtrage pour $Pm|r_i|\sum T_i$ et $Pm|r_i|\sum w_i T_i$

n	m	cpu	sommets	non résolues
10	2	12	12	0
	3	16	14	0
	4	27	20	0
	5	36	26	0
	10			
20	2	472	206	0
	3	751	310	0
	4	796	376	0
	5	983	351	0
	10	17781	3032	0
30	2	11355	2085	0
	3	67772	10771	0
	4	76015	15940	1
	5	100348	18862	2
	10	161101	11635	2
40	2	189529	11860	7

Tableau 4.4 – Résultats du *Branch&Bound* pour $Pm|r_i|\sum C_i$

4.8.3 Résultats préliminaires de la méthode exacte

Nous présentons dans cette section les résultats préliminaires de la méthode exacte pour les problèmes que nous étudions. Un tableau est présenté pour chacun des critères : le Tableau 4.4 pour la somme des dates de fin, le Tableau 4.5 pour la somme pondérée des dates de fin, le Tableau 4.6 pour le retard total et le Tableau 4.7 pour le retard total pondéré. Les colonnes « cpu », « sommets » et « non résolus » indiquent respectivement le temps de calcul moyen en millisecondes, le nombre moyen de sommets explorés et le nombre d'instances non résolues dans le temps imparti, fixé à 30 minutes. Ces résultats sont présentés par nombre de jobs et par nombre de machines. Lorsque plus de 10 instances ne sont pas résolues pour une taille d'instances et un nombre de machine fixés, on considère que la méthode ne sait pas résoudre cette classe d'instances.

On peut constater que la taille des instances résolues reste assez faible. Pour la somme des dates de fin, la méthode résout des instances allant jusqu'à 40 jobs sur 2 machines. La méthode arborescente proposée par Yalaoui et Chu s'avère plus efficace au vu des résultats qu'ils présentent [68]. Pour les autres critères, ces résultats sont, à notre connaissance, les premiers présentant une résolution exacte. Bien qu'ils soient encore d'un intérêt limité étant donnée la faible taille d'instances pouvant être résolues, ils peuvent servir de base à de futurs travaux.

n	m	cpu	sommets	non résolues
10	2	116	106	0
	3	227	196	0
	4	254	150	0
	5	385	154	0
	10			
20	2	93099	16278	0
	3	111370	20712	6
	4	47873	10410	8
	5			
	10	59830	3748	8

Tableau 4.5 – Résultats du *Branch&Bound* pour $Pm|r_i|\sum w_iC_i$

n	m	cpu	sommets	non résolues
10	2	45	77	0
	3	92	170	0
	4	102	181	0
	5	81	146	0
	10			
20	2	23956	12851	0
	3	261693	162948	8
	4			
	5			
	10			

Tableau 4.6 – Résultats du *Branch&Bound* pour $Pm|r_i|\sum T_i$

n	m	cpu	sommets	non résolues
10	2	368	264	0
	3	677	481	0
	4	805	511	0
	5	623	319	0
	10			

Tableau 4.7 – Résultats du *Branch&Bound* pour $Pm|r_i|\sum w_iT_i$

Conclusion

Nous avons présenté dans cette thèse plusieurs contributions relatives aux quatre problèmes d'ordonnancement $Pm|r_i|\sum C_i$, $Pm|r_i|\sum w_i C_i$, $Pm|r_i|\sum T_i$ et $Pm|r_i|w_i T_i$. Ces contributions sont les suivantes :

- De nouvelles règles de dominance de natures différentes. À partir de travaux menés sur une machine [42], nous avons proposé une nouvelle classe d'ordonnancements dominants pour le cas à machines parallèles. Nous avons également proposé trois règles de dominance portant sur des ordonnancements partiels. Ces règles permettent de comparer deux ordonnancements partiels, sous certaines conditions, afin de déterminer si l'un est dominé par l'autre ou non. Nous avons proposé une règle de dominance portant sur les jobs non ordonnancés.
- Un panorama des bornes inférieures pour les quatre problèmes. Nous avons proposé plusieurs bornes inférieures, qui utilisent des relaxations différentes. Nous avons fournis des algorithmes efficaces pour l'ensemble des bornes inférieures. Enfin, nous avons dressé une étude numérique qui permet de comparer l'efficacité des bornes, critère par critère.
- Une méthode de résolution exacte pour les quatre critères. Cette contribution n'est pas encore achevée. Néanmoins, une première implémentation permet de résoudre des instances de petite taille. Cette implémentation permet de mesurer l'efficacité des règles de dominance, grâce à plusieurs types d'intégration.

Plusieurs pistes de recherche nous semblent intéressantes à explorer :

- Une borne supérieure du *makespan* des ordonnancements actifs est utilisée dans les règles de dominance et dans le calcul des bornes inférieures. La qualité de cette borne supérieure est déterminante dans les méthodes qui l'utilisent. Nous pensons qu'il pourrait être pertinent d'approfondir cette question afin d'améliorer la qualité de cette borne supérieure.
- Certaines des bornes inférieures ne s'adaptent pas naturellement lorsque les machines ne sont pas disponibles à la même date. Cette limite ne permet pas une utilisation efficace de ces bornes au sein d'un *Branch&Bound*. Une généralisation permettant pour prendre en compte ce cas de figure semble intéressante, puisqu'elle pourrait améliorer les résultats de la méthode arborescente.
- Il semble que de nouvelles règles de dominance peuvent être mises en place pour ces quatre problèmes. Jusqu'à maintenant, quand un job est

filtré pour succéder à une certaine séquence, la machine sur laquelle il serait ordonnancé n'a aucune importance. Il nous semble que cette information pourrait être utilisée. Nous pensons en particulier que la règle de dominance sur les jobs non ordonnancés pourrait alors être généralisée.

Annexes

Notations

Pour un job i :

r_i	date de disponibilité
p_i	durée
d_i	date échue
w_i	poids

Pour un job ordonnancé i :

C_i	date de fin
T_i	retard
F_i	coût
$start_i$	date de début
$C(i,t)$	date de fin si $start_i = \max(t, r_i)$
$F(i,t)$	coût du job si $C_i = C(i,t)$
$Adj(i)$	front d'adjacence
$m(i)$	indice de la machine sur laquelle est executé le job i
$P(i)$	ensemble des jobs qui précèdent i dans la séquence
Δ_i	date de disponibilité de la machine $m(i)$ avant l'exécution du job i

Pour une séquence σ :

$O(\sigma)$	ordonnancement obtenu à partir de la séquence σ
$(\sigma\|i)$	séquence obtenue en concaténant le job i à la fin de la séquence σ
$F_O(\sigma)$	coût associé aux jobs de σ au sein de l'ordonnancement O
$C_{max}(\sigma)$	date de fin de la séquence σ

Pour un ordonnancement O :

$\sigma(O)$	séquence en bijection avec l'ordonnancement O
$M_i(O)$	ordonnancement machine extrait de O formé des jobs de la machine i
$M(O, \eta)$	sous-ordonnancement partiel de O formés des jobs des machines j, où $j \in \eta$
$\mathcal{V}(O)$	ensemble des ordonnancements voisins de O générés par échange et insertion du dernier job de O

Diverses notations :

N	ensemble des jobs à ordonnancer
NS	ensemble des jobs non encore ordonnancés
PF	ensemble des jobs non encore ordonnancés qu'il est nécessaire de tester
η	sous-ensemble d'indices de machines
H	l'horizon

Bibliographie

[1] K. Aardal, G.L. Nemhauser, et R. Weismantel. Discrete optimization. Dans *Handbooks in Operations Research and Management Science*, volume 12. Elsevier, 2005.

[2] M.S. Akturk et D. Ozdemir. An exact approach to minimizing total weighted tardiness with release dates. *IIE Transactions*, 32 :1091–1101, 2000.

[3] B. Alidaee et D. Rosa. Scheduling parallel machines to minimize total weighted and unweighted tardiness. *Computers & Operations Research*, 24 :775–788, 1997.

[4] M.A. Aloulou, M.Y. Kovalyov, et M.-C. Portmann. Evaluating flexible solutions in single machine scheduling via objective function maximization : the study of computational complexity. *RAIRO - Operations Research*, à paraître.

[5] M. Azizoglu. Preemptive scheduling on identical parallel machines subjects to deadlines. *European Journal of Operational Research*, 148 :205–210, 2003.

[6] M. Azizoglu et O. Kirca. Tardiness minimization on parallel machines. *International Journal of Production Economics*, 55 :163–168, 1998.

[7] M. Azizoglu et O. Kirca. On the minimization of total weighted flow time with identical and uniform parallel machines. *European Journal of Operational Research*, 113 :91–100, 1999.

[8] K.R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley and Sons, 1974.

[9] Ph. Baptiste. Scheduling equal-length jobs on identical parallel machines. *Discrete Applied Mathematics*, 13 :21–32, 2000.

[10] Ph. Baptiste. A note on scheduling multiprocessor tasks with identical processing times. *Computers & Operations Research*, 30 :2071–2078, 2003.

[11] Ph. Baptiste, M. Chrobak, C. Durr, et F. Sourd. Preemptive multi-machine scheduling of equal-length jobs to minimize the average flow time. *Submitted*, 2005.

[12] Ph. Baptiste, A. Jouglet, et D. Savourey. Lower bounds for parallel machine scheduling problems. *International Journal of Operational Research*, à paraître.

[13] H. Belouadah, M.E. Posner, et C.N. Potts. Scheduling with release dates on a single machine to minimize total weighted completion time. *Discrete Applied Mathematics*, 36 :213–231, 1992.

[14] H. Belouadah et C. Potts. Scheduling identical parallel machines to minimize total weighted completion time. *Discrete Applied Mathematics*, 48 :201–218, 1994.

[15] L. Bianco et S. Ricciardelli. Scheduling of a single machine to minimize total weighted completion time subject to release dates. *Naval Research Logistics Quarterly*, 29 :151–167, 1982.

[16] U. Bilge, F. Kiraç, M. Kurtulan, et P. Pekgün. A tabu search algorithm for parallel machine total tardiness problem. *Computers & Operations Research*, 31 :397–414, 2004.

[17] P. Brucker. *Scheduling Algorithms*. Springer Lehrbuch, 1995.

[18] P. Brucker et S. Knust.
http ://www.mathematik.uni-osnabrueck.de/research/OR/class/.

[19] P. Brucker et S. A. Kravchenko. Complexity of mean flow time scheduling problems with release dates. Rapport technique, OSM Reihe P, Heft 251, Universität Osnabrück, Fachbereich Mathematik/Informatik, 2004.

[20] J. Bruno, E.G. Coffman, et R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17 :382–387, 1974.

[21] J. Carlier et Ph. Chrétienne. *Problèmes d'ordonnancement : Modélisation / Complexité*. Masson, 1988.

[22] I. Charon, A. Germa, et O. Hudry. *Méthodes d'optimisation combinatoire*. Masson, 1996.

[23] Z. Chen et W. Powell. Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing*, 11 :78–94, 1999.

[24] C. Chu. A branch and bound algorithm to minimize total flow time with unequal release dates. *Naval Research Logistics*, 39 :859–875, 1992.

[25] C. Chu. A branch and bound algorithm to minimize total tardiness with different release dates. *Naval Research Logistics*, 39 :265–283, 1992.

[26] C. Chu. Efficient heuristics to minimize total flow time with release dates. *Operations Research Letters*, 12 :321–330, 1992.

[27] C. Chu et M.-C. Portmann. Some new efficients methods to solve the $n|1|r_i|\sum T_i$. *European Journal of Operational Research*, 58 :404–413, 1991.

[28] R.W. Conway, W.L. Maxwell, et L.W. Miller. *Theory of Scheduling*. Addison-Wesley, Reading, 1967.

[29] T.H. Cormen, C.E. Leiserson, R.L. Rivest, et C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second édition, 2001.

[30] G.B. Dantzig. *Linear Programming and Extensions*. Princeton, NJ : Princeton University Press, 1963.

[31] F. Della-Croce et V. T'kindt. Improving the preemptive bound for the one-machine dynamic total completion time scheduling problem. *Operations Research Letters*, 31 :142–148, 2003.

[32] GOThA (Groupe d'Ordonnancement Théorique et Appliqué). Les problèmes d'ordonnancement. *RAIRO - Recherche Opérationnelle*, 27 :77–150, 1993.

[33] M.R. Garey et D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[34] R.E. Gomory. An algorithm for integer solutions of linear programs. *Recent Advances in Mathematical Programming*, pages 269–302, 1963.

[35] M. Gondran et M. Minoux. *Graphes et Algorithmes*. Eyrolles, third édition, 1995.

[36] R. Graham, E. Lawler, J. Lenstra, et A.R. Kan. Optimisation and approximation in deterministic sequencing and scheduling. *Annals of Discrete Mathematics*, 4 :287–326, 1979.

[37] C. Guéret, C. Prins, et M. Sevaux. *Programmation linéaire*. Eyrolles, 2000.

[38] A. Hariri et C. Potts. An algorithm for single machine sequencing with release dates to minimize total weighted completion time. *Discrete Applied Mathematics*, 5 :99–109, 1983.

[39] J.A. Hoogeveen, J.K. Lenstra, et S.L. Van de Velde. *Sequencing and scheduling*, chapter 12. John Wiley and Sons, M. Dell'Amico, F. Maffioli & S. Martello (Eds.), Annotated Bibliographies in Combinatorial Optimization édition, 1997.

[40] W.A. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21 :846–847, 1973.

[41] W.A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quaterly*, 21 :177–185, 1974.

[42] A. Jouglet. *Ordonnancer une machine pour minimiser la somme des coûts*. Thèse de doctorat, Université de Technologie de Compiègne, France, 2002.

[43] A. Jouglet, Ph. Baptiste, et J. Carlier. *Handbook of scheduling : algorithms, models, and performance analysis.*, chapter Branch-and-Bound Algorithms for Total Weighted Tardiness. Leung, J. Y-T, Chapman & Hall / CRC édition, 2004.

[44] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4 :373–395, 1984.

[45] S. Kedad-Sidhoum, Y. Rios Solis, et F. Sourd. Lower bounds for the earliness-tardiness scheduling problem on single and parallel machines. *European Journal of Operational Research*, à paraître.

[46] L.G. Khachiyan. A polynomial-time algorithm for linear programming. *Soviet Mathematics Doklady*, 20 :191–194, 1979.

[47] J. Y-T. Leung, éditeur. *Handbook of scheduling : Algorithms, Models and Performance Analysis*. Chapman & Hall / CRC, 2004.

[48] J. Y.-T. Leung et M. Pinedo. Minimizing total completion time on parallel machines with deadline constraints. *SIAM Journal on Computing*, 32 :1370–1388, 2003.

[49] C.F. Liaw, Y.K. Lin, C.Y. Cheng, et M. Chen. Scheduling unrelated parallel machines to minimize total weighted tardiness. *Computers & Operations Research*, 30 :1777–1789, 2003.

[50] R. McNaughton. Scheduling with deadlines and loss functions. *Management Sciences*, 6 :1–12, 1959.

[51] G.L. Nemhauser et L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1998.

[52] R. Nessah. *Ordonnancement de la production pour la minimisation des encours*. Thèse de doctorat, Université de Technologie de Troyes, 2005.

[53] R. Nessah, C. Chu, et F. Yalaoui. An exact method for $Pm|sds, r_i|\sum C_i$ problem. *Computers & Operations Research*, Accepté, 2005.

[54] R. Nessah, F. Yalaoui, et C. Chu. La minimisation du temps de séjours des tâches ordonnancées sur des machines parallèles identiques : Nouvelle borne inférieure. Dans *Conférence Internationale Productique CIP'03*, 2003.

[55] R. Nessah, F. Yalaoui, et C. Chu. Méthode exacte pour le problème d'ordonnancement $Pm|r_i|\sum C_i$. Dans *6e Congrès International de Génie Industriel*, Besançon (France), 2005.

[56] J. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41 :338–350, 1993.

[57] Y. Pan. An improved branch and bound algorithm for single machine scheduling with deadlines to minimize total weighted completion time. *Operations Research Letters*, 31 :492–496, 2003.

[58] C. Potts et L.V. Wassenhove. An algorithm for single machine sequencing with deadlines to minimize total weighted completion time. *European Journal of Operational Research*, 12 :379–387, 1983.

[59] D. Rivreau. *Problèmes d'Ordonnancement Disjonctifs : Règles d'Elimination et Bornes Inférieures*. Thèse de doctorat, Université de Technologie de Compiègne, France, 1999.

[60] D. Savourey, Ph. Baptiste, et A. Jouglet. Bornes inférieures pour machines parallèles. Dans *ROADEF'06, Congrès de la Société Fran caise en Recherche Opérationnelle, Lille, France*, 2006.

[61] D. Savourey, Ph. Baptiste, et A. Jouglet. Lower bounds for parallel machines scheduling. Dans *RIVF2006, Research, Innovation and Vision for the Future, Ho Chi Minh Ville, Vietnam*, 2006.

[62] D. Savourey et A. Jouglet. Dominance rules for scheduling jobs with release dates on parallel machines. Dans *Models and Algorithms for Planning and Scheduling Problems MAPSP'05, Siena, Italy*, 2005.

[63] D. Savourey, A. Jouglet, et Ph. Baptiste. Règles de dominance pour l'ordonnancement de jobs avec dates de disponibilité sur machines parallèles. Dans *ROADEF'05, Congrès de la Société Fran caise en Recherche Opérationnelle, Tours, France*, 2005.

[64] T. Schiex et G. Verfaillie. No-good recording for static and dynamic CSP. Dans *Proceeding of the 5th IEEE International Conference on Tools with Artificial Intelligence*, 1993.

[65] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1998.

[66] S. Webster. Weighted flow time bounds for scheduling identical processors. *European Journal of Operational Research*, 80 :103–111, 1995.

[67] F. Yalaoui et C. Chu. Parallel machine scheduling to minimize total tardiness. *International Journal of Production Economics*, 76 :265–279, 2002.

[68] F. Yalaoui et C. Chu. A new exact method to solve the $Pm|r_i|\sum C_i$ problem. *International Journal of Production Economics*, In press, 2005.

Cette thèse a été écrite avec GNU Emacs en LaTeX 2ε. Les figures ont été réalisées à l'aide du paquet *PSTricks*. La police utilisée est *math-pazo* en 12 points. La feuille de style a été initialement écrite par Joanny Stéphant et François Clautiaux, puis modifiée par Julien Chiquet et moi-même.

Titre Ordonnancement sur machines parallèles : minimiser la somme des coûts

Résumé Nous étudions quatre problèmes d'ordonnancement sur machines parallèles. Ces quatre problèmes diffèrent par le critère que l'on cherche à minimiser : la somme des dates de fin, la somme pondérée des dates de fin, le retard total ou le retard total pondéré. Les jobs à ordonnancer sout soumis à des dates de disponibilité. Nous avons proposé pour ces quatres problèmes plusieurs règles de dominance. Une étude des bornes inférieures a également été réalisée. Enfin, nous avons proposé une méthode de résolution exacte utilisant les règles de dominance ainsi que les bornes inférieures.

Mots-clés Recherche Opérationnelle, Ordonnancement, Machines parallèles, Bornes inférieures, Méthode arborescente, Règles de dominance, Retard total (pondéré)

Title Parallel Machines scheduling problem with minsum criteria

Abstract We study four parallel machines scheduling problems. These problems are different by the criterion to minimize : the total completion time, the total weighted completion time, the total tardiness or the total weighted tardiness. Jobs can not be processed before release dates and preemption is forbidden. We have proposed several dominance rules for these problems. A complete study of lower bounds, existing and new ones, is also provided. Finally, we have presented a *Branch&Bound* method to solve exactly those problems, which takes advantage of lower bounds and dominance rules.

Keyword Operational Research, Scheduling, Parallel machines, Lower bounds, Dominance Rules, *Branch&Bound*, Total (weighted) tardiness