UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

OFFERTE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
EN VERTU D'UN PROTOCOLE
AVEC L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PAR

SANA EL BAHLOUL

FLOW-SHOP À DEUX MACHINES AVEC DES TEMPS DE LATENCE : APPROCHE EXACTE ET HEURISTIQUE

Janvier 2008

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

RESUME

Un ordonnancement est défini comme étant une allocation, dans le temps, des ressources (machines) disponibles aux différents travaux (tâches, jobs) à réaliser, dans le but d'optimiser un ou plusieurs objectifs. La richesse de la problématique de l'ordonnancement est due aux différentes interprétations que peuvent prendre les ressources et tâches. Ainsi, les ressources peuvent être des machines dans un atelier, des pistes de décollage et d'atterrissage dans un aéroport, des équipes dans un terrain de construction, des processeurs dans les ordinateurs, etc. Les tâches, quant à elles, peuvent être des opérations dans un processus de production, le décollage et l'atterrissage dans un aéroport, les étapes d'un projet de construction, l'exécution d'un programme informatique, etc. Les différentes tâches sont caractérisées par un degré de priorité et un temps d'exécution. Les ressources, quant à elles, sont caractérisées entre autres par une capacité, des temps de réglage, etc.

Les problèmes d'ordonnancement sont généralement classés en deux modèles dépendamment du nombre d'opérations que requièrent les jobs : les modèles à une opération (modèle à machine unique et modèle à machines parallèles) et les modèles à plusieurs opérations dits modèles en ateliers (*flow-shop*, *open-shop* et *job-shop*). Bien entendu, il est également possible de trouver d'autres modèles, hybrides, qui sont des mélanges de ces deux modèles. Cette classification a permis, un tant soit peu, de mieux comprendre et cerner les problèmes d'ordonnancement réels. Toutefois, l'expérience a montré qu'il existe toujours un gouffre entre la théorie et ce qui se passe réellement dans les centres de production ou les prestations de services.

Parmi les contraintes que la théorie d'ordonnancement n'a pas considérées jusqu'à un passé récent, nous pouvons citer les temps de latence des travaux, correspondant aux différents temps nécessaires devant s'écouler entre la fin d'une opération et le début de la prochaine opération d'un même job. Les temps de latence peuvent correspondre par exemple aux temps de transport des jobs à travers les ressources ou encore aux temps de refroidissement des jobs, avant leurs prochaines manipulations. Ces temps peuvent dans certains cas prendre des proportions importantes qu'une entreprise ne doit en aucun cas ignorer. Elle devrait même revoir sa politique d'ordonnancement pour pouvoir améliorer sa productivité.

Dans ce mémoire, nous étudions le modèle de *flow-shop* à deux machines avec des temps de latence dans le but de minimiser le temps total d'accomplissement des jobs, appelé *makespan.* Pour ce faire, nous avons, tout d'abord, introduit la théorie de l'ordonnancement et survolé quelques concepts de la théorie de la NP-complétude ainsi que les différentes méthodes de résolution d'un problème d'ordonnancement en général, et du *flow-shop* à deux machines en particulier.

Pour résoudre ce problème de *flow-shop* à deux machines, nous avons utilisé, dans un premier temps la méthode de *branch and bound.* Nous avons commencé par appliquer cet algorithme sur le cas particulier des temps d'exécution unitaires. Outre les bornes inférieures et supérieures, nous y avons également présenté des règles de dominance. Limité à 900 secondes, pour l'exécution d'une instance, cet algorithme a pu résoudre efficacement des instances n'excédant pas 30 jobs. Ensuite, nous sommes passés au cas où les temps d'exécution des jobs sont quelconques. Nous y avons présentés plusieurs bornes inférieures. Pour les bornes supérieures,

nous avons conçu des heuristiques basées sur *NEH*, la règle de Johnson, la règle de Palmer et *CDS*. Au-delà de 10 jobs, l'algorithme de *branch and bound*, que nous avons implémenté, n'a pu résoudre efficacement les instances générées, même en posant 1h d'exécution pour chaque instance. Notons que les temps d'exécution des algorithmes, implémentant ces bornes inférieures et supérieures ainsi que ceux des règles de dominance, pour le cas unitaire, sont tous majorés par une complexité en $O(n\log n)$; n étant le nombre de jobs à ordonnancer. Dans un second temps, nous sommes passés à l'autre approche de résolution qu'est la méthode métaheuristique. Nous avons commencé notre étude par le développement d'un algorithme de recherche avec tabous. Des ajouts itératifs tels des procédures d'intensification et de diversification ont nettement amélioré les résultats générés par cet algorithme. Ensuite, nous avons conçu un algorithme génétique. Nous y avons incorporé une recherche locale pour améliorer les résultats. Cependant, l'algorithme de recherche avec tabous a produit de meilleurs résultats sur l'ensemble des instances testées. En guise de conclusion, nous avons discuté de nouvelles pistes de recherche à explorer.

REMERCIEMENTS

Je n'aurais pas survécu à l'aventure de ce mémoire si je n'avais pu compter en permanence sur une main tendue, prête à me relever. Les personnes qui ont partagé mon existence durant cette aventure sont la clé de voûte de cet accomplissement. Il m'est maintenant permis de les remercier au travers de ces quelques lignes... et je ne vais pas m'en priver !

Tout d'abord, je remercie **Dieu** qui m'a donné non seulement le courage mais aussi la force et la patience de réaliser ce travail et a exaucé mes prières en me donnant la chance de pouvoir vous le remettre aujourd'hui.

On n'a pas souvent l'occasion de remercier ses parents dans une vie, alors je suis contente que l'occasion m'en soit donnée ici. Eux, cela fait 25 ans qu'ils me supportent alors je leur dois bien ça. Merci maman, Merci papa. Je vous remercie tout au fond de mon cœur d'avoir cru en moi. Je vous remercie pour votre présence dans ma vie. Vous avez été et vous serez toujours la lueur de bonheur et d'espoir dans ma vie. Vous êtes toujours là pour moi. Merci pour votre confiance, gentillesse et bonté. Merci de m'avoir fourni tout ce dont j'avais besoin pour que je sois dans les meilleures conditions, allant même à vous sacrifiez. Merci pour les longues nuits où vous restiez éveillés en pensant à moi et à mes problèmes. Et pardonnez moi pour tous les soucis que je vous ai causés. Je sais maman que ces quelques mots ne sont guère suffisants pour décrire ma gratitude et mon amour pour toi et papa. Mais j'espère que tu seras ravie et fière de ta fille. Je vous dédie donc ce mémoire comme modeste marque de l'amour que je vous porte pour toujours et qui j'espère sera un jour à la hauteur de celui avec lequel vous m'avez comblé toute ma vie.

Merci aussi à mon frère Aymen, à ma sœur Amina et à tout le reste de ma famille, surtout ma tante Samouha. Que **Dieu** les garde et leur donne autant de bonheur qu'ils m'en ont donné. Je n'oublierai jamais de remercier mes grand parents Hechmi et Chefia pour leur bénédiction. À la mémoire de ma grand mère Essia disparue, qui j'espère de là-haut sera fière de moi. Que **Dieu** la protège.

Je tiens à remercier l'amour de ma vie mon mari, Aymen. Merci chéri pour tout, pour ton amour, pour ton soutien, pour ta présence quand j'en avais réellement besoin, pour ta tendresse et

ta gentillesse. Je n'oublierai j'aimais les jours où j'étais déprimée et j'avais décidé de tout laisser tomber. Tu étais là pour m'encourager, me donner la force et la confiance, me dire que je peux réaliser mes rêves et je que peux surmonter tous les obstacles. Merci, tu as a comblé ma vie par ta bonté et ton amour. Tu étais toujours là pour m'écouter et me conseiller. Je suis vraiment désolée pour mes sautes d'humeurs et tout le mal que je t'ai fait subir. Tu étais toujours là à mes côtés durant la majorité de la réalisation de ce travail et jusqu'à la dernière ligne de ce mémoire. Je dédie ce travail, en tant que premier édifice à notre vie, à mon époux Aymen et mon fils Yassine qui sera bientôt parmi nous.

Je tiens à remercier en particulier Mama Saïda et mes grandes sœurs Basma et Hinda et tout le reste de leur famille pour leur soutien moral.

Je remercie également mes beaux-parents, Tata Naïma et Tonton Mohamed, qui n'ont pas arrêté de me soutenir durant cette dernière année. J'ai beaucoup apprécié leur soutien moral. Et je remercie aussi ma belle sœur Sarra et mes beaux frères Akram et Wadii.

J'exprime ma sincère gratitude à mon directeur de recherche Djamal Rebaine pour le temps consacré à la lecture de mon travail et aux réunions qui ont rythmé les différentes étapes de mon mémoire. Un très grand merci pour votre conscience professionnelle, votre générosité et votre disponibilité.

De même, je remercie aussi tout le corps professoral et administratif du département d'Informatique et de Mathématiques de l'Université du Québec à Chicoutimi pour tous les enrichissements que j'ai eus ainsi que leur aide inestimable.

Finalement, je remercie également tous mes amis et tous ceux que j'aime, Zeinab, Ahmed, Zied, Manel, Hatem, Charaf, Ouelid, Hamadi et j'en oublie certainement qui me le pardonneront. Je dois tellement à ces personnes qui me transmettent de la joie de vivre tout le temps et qui sont à mes côtés quand j'en ai besoin.

TABLE DES MATIERES

LISTE DES FIGURES

LISTE DES TABLEAUX

À mon mari,
À mes parents, mon frère et ma sœur, mes beaux-parents,
À mes ami(e)s.

Chapitre 1 : Introduction générale

Un problème d'ordonnancement consiste à organiser dans le temps la réalisation d'un ensemble de tâches (jobs, travaux) sur un ensemble de ressources en vue de satisfaire un ou plusieurs objectifs. Ainsi, une tâche est une entité élémentaire localisée dans le temps par une date de début et/ou de fin, dont la réalisation nécessite une durée, et qui consomme un moyen selon une certaine intensité. La ressource est, quant à elle, un moyen technique ou humain destiné à être utilisé pour la réalisation d'une tâche et est disponible en quantité limitée. L'évaluation d'une solution d'ordonnancement se fait par rapport à un ou plusieurs objectifs de performance.

Parmi les problèmes d'ordonnancement qui ont été traités dans la littérature, le problème des ateliers sériels (*flow-shop*) a été l'un des premiers à être étudié vers le début des années cinquante. Johnson, en 1954, a été le premier à avoir traité le problème de *flow-shop* à deux machines [Johnson, 1954]. Nous pouvons aussi trouver des problèmes de *job-shop*, d'*open-shop* ou encore des problèmes dits flexibles, dépendamment de la conception de l'atelier. Depuis, l'expérience a montré qu'il existe un gouffre entre la théorie de l'ordonnancement et ce qui se passe réellement dans les centres de productions ou les prestations de services. Parmi les contraintes que la théorie n'a pas considérées jusqu'à un passé récent, nous pouvons citer les temps de latence des travaux, correspondant aux différents temps nécessaires devant s'écouler entre la fin d'une opération et le début de la prochaine opération d'un même job. Les temps de latence peuvent correspondre par exemple aux temps de transport des jobs à travers les ressources ou encore aux temps de refroidissement des jobs avant leur prochaine manipulation. Ces temps peuvent dans certains cas prendre des proportions importantes qu'une entreprise ne doit en aucun cas ignorer. Dans certains, elle devra même revoir sa politique d'ordonnancement pour pouvoir améliorer sa productivité.

La prise en compte des temps de latence a rendu les problèmes de *flow-shop* plus complexes à résoudre d'un point de vue computationnel. Toutefois, une panoplie de méthodes peut être utilisée pour résoudre ces problèmes. En effet, nous pouvons trouver dans la littérature des méthodes exactes comme la programmation linéaire et la relaxation

lagrangienne ou des méthodes approchées comme les heuristiques. Avec l'augmentation de la puissance de calculs des machines, les limites des méthodes exactes peuvent encore être repoussées plus loin. Et c'est dans cette optique que ce mémoire a pour but de concevoir et de mettre en œuvre un algorithme de *branch and bound* et deux métaheuristiques, à savoir l'algorithme de recherche avec tabous et l'algorithme génétique.

En plus de ce chapitre introductif, ce mémoire comporte sept autres chapitres. Ainsi le Chapitre 2 porte sur les problèmes d'ordonnancement, les temps de latence et la méthodologie de résolution d'un problème d'ordonnancement. Le Chapitre 3 décrit la résolution du problème de *flow-shop*. Une revue de la littérature et l'étude de complexité algorithmique de ce problème sont aussi présentées. Les approches heuristiques, plus précisément l'algorithme de recherche avec tabous et l'algorithme génétique, y sont décrites. Le Chapitre 4 traite le problème de *flow-shop* à deux machines avec temps de latence. Dans ce chapitre, la méthode de *branch and bound* est présentée en détails. C'est-à-dire, nous y décrivons des bornes inférieures ainsi que les différentes heuristiques utilisées. Le Chapitre 5 décrit un algorithme de recherche avec tabous et un algorithme génétique pour ce problème. Une simulation a été également présentée dans ces deux derniers chapitres afin d'évaluer la performance des méthodes qui y sont discutées. Finalement, nous concluons ce mémoire par une discussion sur le travail effectué et sur certaines pistes de recherche en découlant.

Chapitre 2 : Les problèmes d'ordonnancement

2.1 Introduction

L'ordonnancement est une allocation, dans le temps, des ressources disponibles aux différentes tâches, dans le but d'optimiser un ou plusieurs objectifs.

La richesse de la problématique de l'ordonnancement est due aux différentes interprétations que peuvent prendre les ressources et tâches. Ainsi, les ressources peuvent être des machines dans un atelier, des pistes de décollage et d'atterrissage dans un aéroport, des équipes dans un terrain de construction, des processeurs dans les ordinateurs, etc. Alors que les tâches peuvent être des opérations dans un processus de production, le décollage et l'atterrissage dans un aéroport, les étapes d'un projet de construction, l'exécution d'un programme informatique, etc. Les différentes tâches sont caractérisées par un degré de priorité et un temps d'exécution. Les ressources quant à elles, sont caractérisées par une capacité, des temps de réglage, etc.

L'objectif visé varie selon le problème d'ordonnancement à traiter : minimiser le temps total d'exécution des tâches, appelé *makespan*, minimiser le nombre des tâches devant être exécutées après leur date d'expiration, minimiser les délais d'attente, etc. Ainsi, en fonction des objectifs, un ordonnancement adéquat est déterminé. A titre d'illustration, citons l'exemple suivant :

Dans un hôpital, on a besoin d'un certain nombre d'infirmières. Ce nombre varie d'un jour à l'autre. Par exemple, le nombre d'infirmières travaillant pendant la semaine est plus important que celui des fins de semaine. En plus, le nombre d'infirmières qui travaillent durant la nuit est inférieur à celui du jour. Donc, on doit concevoir des modèles pour affecter les différents quarts de travail à chaque infirmière, pour un horizon donné.

La problématique d'ordonnancement, étudiée dans ce mémoire, est caractérisée par un ensemble $N = \{1,2,\ldots, n\}$ de n tâches appelées aussi *jobs* et $M = \{1,2,\ldots, m\}$ un ensemble de m ressources, appelées *machines*.

On peut remarquer que dans la théorie de l'ordonnancement classique, les chercheurs n'ont considéré dans leurs travaux que deux contraintes : chaque job ne peut être exécuté que par une seule machine et, à un instant donné, une machine ne peut exécuter au plus qu'un seul job.

Les problèmes d'ordonnancement sont généralement classés en trois principaux modèles dépendamment du nombre d'opérations que requièrent les jobs : des modèles à une opération (machine unique et machines parallèles) et des modèles à plusieurs opérations (*flow-shop*, *open shop* et *job shop*).

Dans un modèle à machine unique, l'ensemble des tâches à réaliser est exécuté par une seule machine. L'une des situations intéressantes où on peut rencontrer ce genre de configurations est le cas où on est devant un système de production comprenant une machine goulot qui influence l'ensemble du processus. L'étude peut alors être restreinte à l'étude de cette machine. La Figure 2-1 illustre le modèle machine unique.

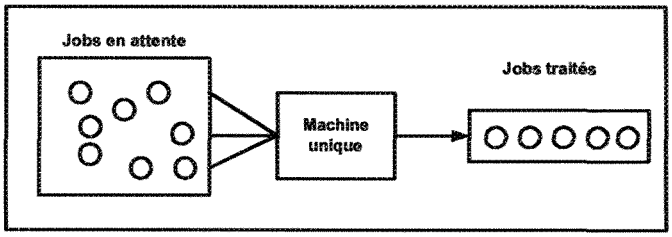


Figure 2-1 : Modèle machine unique

Avec les avancés technologiques, les systèmes de production ont considérablement évolué. On peut trouver des ateliers composés de machines organisées en parallèle. Ce modèle est utilisé surtout dans les secteurs industriels tels que : l'industrie alimentaire, les industries plastiques, les fonderies et en particulier l'industrie textile. Le processus de déroulement de ce système de production est le suivant : à chaque fois qu'une machine i se libère, ont lui affecte un job j comme illustré à la Figure 2-2. Dans le cas d'un processus d'assemblage industriel, par exemple, si l'une des étapes d'assemblage nécessite beaucoup de temps, il serait très intéressant alors d'avoir plusieurs machines parallèles qui effectuent la même tâche. D'un autre côté, les machines parallèles sont classées suivant leur rapidité. Si toutes les machines de l'ensemble M ont la même vitesse de traitement et effectuent les mêmes tâches, elles sont identiques. Si les machines ont des vitesses de traitement différentes mais linéaires alors elles sont dites uniformes. Dans

le cas où les vitesses des machines sont indépendantes les unes des autres, on parle alors de modèle de machines parallèles non reliées ou indépendantes.

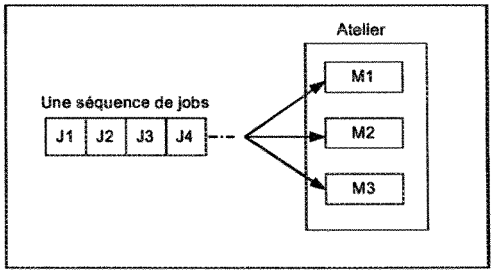


Figure 2-2 : Ordonnancement d'une séquence de jobs sur des machines organisées en parallèle

Le modèle à plusieurs opérations est constitué des cas où un job, pour se réaliser, doit passer par plusieurs machines, chacune de ces machines ayant ses spécificités. On peut rencontrer ce modèle dans les industries de montages telle que la chaîne de montage des voitures (Figure 2-3). On distingue trois modèles selon l'ordre de passage des jobs sur les machines, à savoir les modèles *d'open shop*, *flow-shop* et *job shop*.

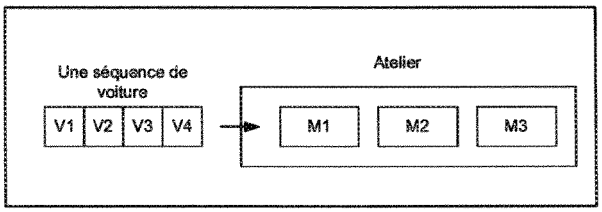


Figure 2-3 : Une chaîne de montage des voitures

Dans le modèle d'*open shop,* l'ordre de passage des *n* jobs sur les *m* machines n'est pas connu à l'avance. Cet ordre est déterminé lors de la construction de la solution. Chaque job *j* peut avoir son propre ordre de passage sur toutes les machines. Le fait qu'il n'y ait pas d'ordre prédéterminé rend la résolution du problème d'ordonnancement de ce type plus complexe, mais offre cependant des degrés de liberté intéressants. À la Figure 2-4, nous avons un ensemble de quatre jobs et un ensemble de quatre machines. À droite de la figure nous pouvons remarquer que chaque job a suivi un ordre de passage différent sur les quatre machines.

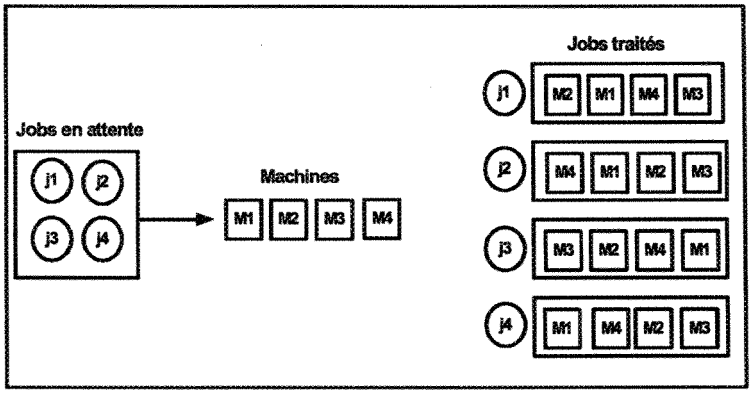


Figure 2-4 : Modèle open – shop

Dans le modèle de *flow-shop*, les ordres de fabrication visitent les machines dans le même ordre, avec des durées opératoires pouvant être différentes. Chaque job va être s'exécuter sur les *m* machines en série et tous les jobs vont suivre le même ordre de passage sur ces machines. Ce type de modèle est aussi appelé modèle linéaire. La Figure 2-5 illustre le cas d'un *flow-shop* avec quatre machines et quatre jobs. Les quatre jobs suivent le même ordre de traitement sur les quatre machines.

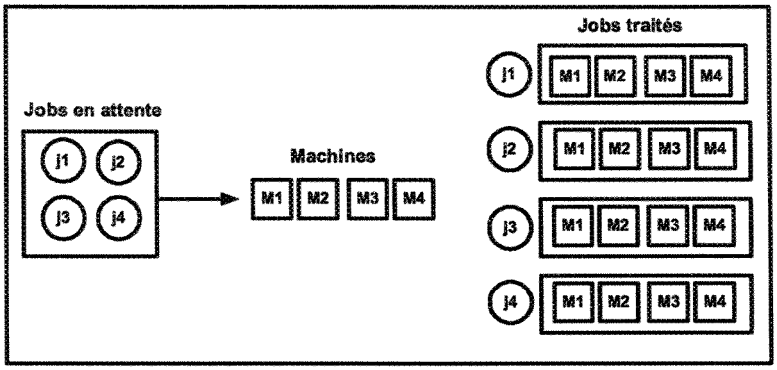


Figure 2-5 : Modèle flow-shop

Concernant le modèle de *job-shop*, chaque job à un ordre à suivre et chacun d'entre eux peut s'exécuter plusieurs fois sur la même machine; ce qui n'est pas le cas du *flow-shop*. Il s'agit dans ce cas de déterminer les dates de passage sur différentes ressources d'ordres de fabrication ayant des trajets différents dans l'atelier. Ces ordres de fabrication partageant des ressources communes, des conflits sont susceptibles de survenir, résultant des croisements de flux illustrés à la Figure 2-6. Dans cette figure, l'ordre de passage de chaque job est indiqué par un chemin de même couleur que celui du job. Par exemple, le

job j_1 passe par toutes les machines, alors que le job j_3 ne passe que par la deuxième et la quatrième machine. Dans son expression la plus simple, le problème consiste à gérer ces conflits tout en respectant les contraintes données, et en optimisant les objectifs poursuivis. Les types de ressources et de contraintes prises en compte peuvent toutefois considérablement compliquer le problème. Plus on intégrera de contraintes, plus on se rapprochera d'un cas réel, mais moins on disposera de méthodes de résolution satisfaisantes. Parmi ces contraintes, nous pouvons citer l'ordonnancement multiressources (une opération nécessite plusieurs ressources), la possibilité d'effectuer des chevauchements, la prise en compte de temps de reconfiguration dynamique, etc.

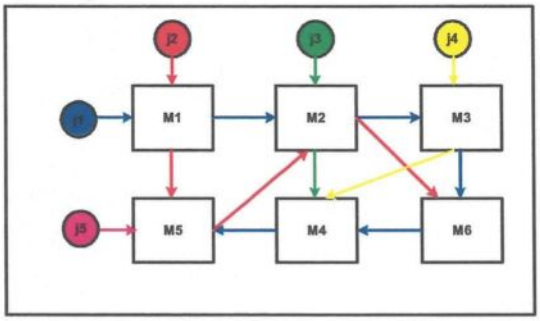


Figure 2-6 : Modèle job – shop

Pendant les dernières décennies, plusieurs ouvrages portant sur l'ordonnancement et la planification sont apparus. Le volume de Muth et Thompson [Muth et Thompson, 1963] contient un ensemble d'articles décrivant l'aspect algorithmique de l'ordonnancement. Conway, Maxwell et Miller [Conway et *al.*, 1967] traitent aussi les aspects stochastiques et les files de priorité. Baker [Baker, 1974] fournit une revue de la littérature concernant les aspects déterministes de l'ordonnancement. Dans son livre, French [French, 1982] couvre plusieurs techniques utilisées pour l'ordonnancement déterministe[1]. Blazewicz et *al.* [Blazewicz et *al.*, 1986] traitent les problèmes à ressources contraignantes et l'ordonnancement déterministe multi-objectif[2]. Plus liés à des problèmes industriels réels, Blazewicz et *al.* [Blazewicz et *al.*, 1993] couvrent aussi les aspects algorithmiques de l'ordonnancement. Dans leur livre, Pinedo et Chao [Pinedo et Chao, 1999] s'orientent vers les problèmes industriels en les classant par modèle.

[1] Ordonnancement pour lequel les données du problème sont connues d'une manière exacte à l'avance.

[2] Ordonnancement qui satisfait plusieurs objectifs à la fois.

2.2 Ordonnancement et critères d'optimalité

Dans la théorie de l'ordonnancement, sont utilisés souvent les termes suivants : « séquence », « ordonnancement » et « calendrier » pour présenter ou décrire un problème d'ordonnancement. Ces termes sont souvent interchangeables. Pour mieux distinguer les nuances entre ces mots, il faut comprendre le sens de chaque terme. Ainsi, l'exécution d'une séquence de jobs est simplement l'ordre de passage des jobs sur les machines. Dans une séquence, on n'a pas à préciser le temps de début ou de fin de chaque opération effectuée. Pour exécuter une séquence de jobs, on a besoin d'un calendrier contenant des informations concernant l'ordonnancement des jobs. Ainsi, un diagramme de Gantt[3] permet de visualiser un ordonnancement donné. Chaque bloc de ce diagramme donne le temps de début et de fin d'exécution de chaque job.

Ces termes sont souvent utilisés lors de la résolution des problèmes d'ordonnancement. Dans ces derniers, on trouve aussi les critères de performance ou encore d'évaluation de la qualité d'un ordonnancement. Ces critères sont nombreux. Mellor [Mellor, 1966] en a distingué 27. Par ailleurs, on différencie deux classes de critères de performance : les critères de performance réguliers et non réguliers.

Soit C_j et C'_j les dates de fin d'exécution d'un job j dans deux ordonnancements différents de mêmes tailles de séquence.

Définition 1

Un critère de performance est dit régulier, s'il est une fonction L qui vérifie cette condition :

$$C_1 \le C'_1, C_2 \le C'_2, ..., C_n \le C'_n \Rightarrow L(C_1, C_2, ..., C_n) \le L(C'_1, C'_2, ..., C'_n)$$

À partir de cette définition et pour un critère de performance régulier, on peut dire qu'un ordonnancement est meilleur qu'un autre [French, 1982]. Or ce n'est pas le cas pour un critère de performance non régulier, il ne vérifie pas cette condition [Raghavachari, 1988].

Comme illustré à la Figure 2-7, on peut avoir trois sortes d'ordonnancement : des ordonnancements actifs, semi-actifs et sans retard.

---

[3] Outil inventé en 1917 par GANTT [1919].

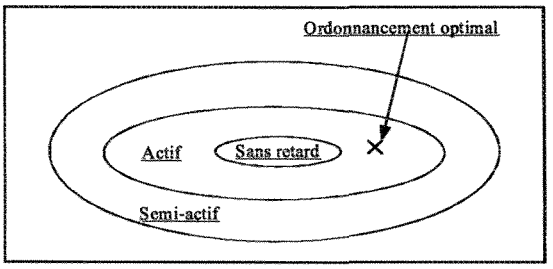


Figure 2-7: Le diagramme de Venn représentant les classes d'ordonnancement

Définition 2

Un ordonnancement est dit actif s'il est impossible d'avancer le début d'exécution d'une opération sans devoir retarder une autre tâche ou violer une contrainte (de précédence, date de début au plus tôt, ...).

Exemple 2-1

Soit, un problème de *job-shop* avec $m = 3$ et $n = 2$, le tableau suivant présente les temps d'exécution de chaque job.

	J1	*J2*
M1	1	0
M2	3	3
M3	0	2

Tableau 2-1 : Temps d'exécution du problème $m = 3$ et $n = 2$ de l'Exemple 2-1

On suppose qu'on a un ordonnancement qui permet d'exécuter sur la machine *M2* le job *J2* puis le job *J1* (voir la Figure 2-8). Il est alors clair que cet ordonnancement est actif, car si on place le *J1* avant le job *J2* sur *M2* la troisième condition de la Définition 3 (ci-dessous) ne sera pas vérifiée.

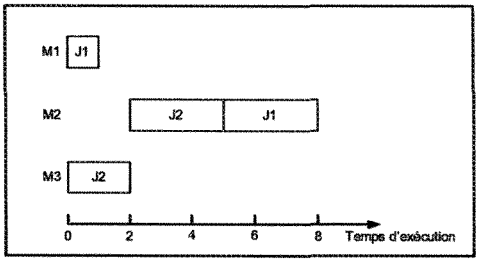


Figure 2-8 : Un ordonnancement actif

Définition 3

Un ordonnancement est dit semi-actif si aucune tâche ne peut être exécutée plus tôt sans changer l'ordre d'exécution sur les ressources ou violer une contrainte (de précédence, date de début au plus tôt, ...).

Autrement dit, dans un ordonnancement semi-actif, l'exécution de chaque opération commence aussitôt que possible.

Exemple 2-2

Soit, un problème de *job-shop* avec $m = 3$ et $n = 2$, le tableau suivant représente les temps d'exécution de chaque job.

	J1	*J2*
M1	1	0
M2	1	2
M3	0	2

Tableau 2-2 : Temps d'exécution du problème $m = 3$ et $n = 2$ de l'Exemple 2-2

J1 sera exécuté sur *M1* puis sur *M2* alors que *J2* sera exécuté sur *M2* puis sur *M3*. On suppose qu'on va ordonnancer sur la machine *M2* le job *J2* avant le job *J1* (Figure 2-9). Il est alors clair que le job *j2* commence son exécution sur *M2* à un temps $t = 2$, alors que le job *J1* à $t = 4$. Cet ordonnancement est semi – actif.

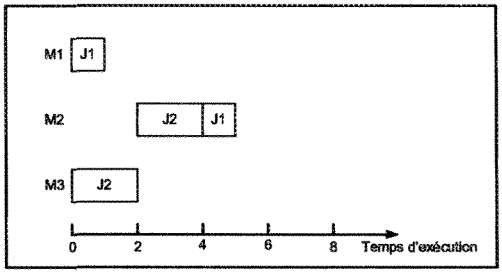


Figure 2-9 : Un ordonnancement semi-actif

Théorème 1 [French, 1982]

Pour minimiser un critère régulier, il est suffisant de considérer un ordonnancement semi-actif.

Dans ce mémoire, notre but est de minimiser la durée totale d'accomplissement des jobs, appelé également le *makespan*. Il faut alors minimiser la fonction suivante :

$$C_{\max} = \max_{1 \le i \le n} \{C_i\}$$

Le *makespan* représente le temps de fin d'exécution du dernier job dans une séquence. Il est l'un des critères les plus utilisés pour évaluer le coût d'un ordonnancement. En minimisant ce critère, on peut améliorer le rendement et réduire le temps moyen d'inactivité des machines [Chrétienne et Carlier, 1988]. La minimisation du *makespan* s'accompagne généralement de contraintes qui peuvent être temporelles ou liées aux ressources [Boivin, 2005]. Les contraintes temporelles se divisent en deux catégories : des contraintes de temps alloué (impératif de gestion : délai de livraison, disponibilité, achèvement) et des contraintes d'antériorité (cohérence technologique : gammes de fabrication, inégalité de potentiels : précédence). Les contraintes liées aux ressources peuvent être des contraintes disjonctives (une tâche *i* doit s'exécuter avant ou après une tâche *j*) ou des contraintes cumulatives (respect des capacités des ressources).

On peut aussi considérer d'autres critères de performance, tels que le temps moyen d'achèvement des jobs, le temps total de traitement, le temps de retard total, le temps d'attente des jobs, le taux d'occupation de machines, le nombre de jobs en retard, le temps de séjour d'un job dans le système avant sa réalisation, etc.

Dans ce mémoire, nous étudions le problème du *flow-shop* à deux machines et nous avons pour objectif de minimiser le *makespan*.

2.3 Diagramme de Gantt

Tout ordonnancement peut être représenté par l'intermédiaire d'un diagramme qu'on appelle diagramme de Gantt. Ce dernier, représentant un tableau mural, est un outil permettant de visualiser dans le temps les diverses tâches composant un projet. Dans un diagramme, on a deux axes perpendiculaires. L'axe horizontal représente les unités de temps, tandis que l'axe vertical représente les machines.

Exemple 2-3

Soit, $m = 2$ et $n = 3$, le tableau suivant représente les temps d'exécution de chaque job.

	J1	*J2*	*J3*
M1	4	2	3
M2	2	5	2

Tableau 2-3 : Temps d'exécution du problème $m = 2$ et $n = 3$ de l'Exemple 2-3

La Figure 2-10 représente le diagramme de Gantt associé à un ordonnancement. À partir de ce diagramme on peut déterminer la valeur du *makespan* ou d'un autre critère tel que : le temps de retard, de latence, etc. On peut alors voir si la solution en question est optimale.

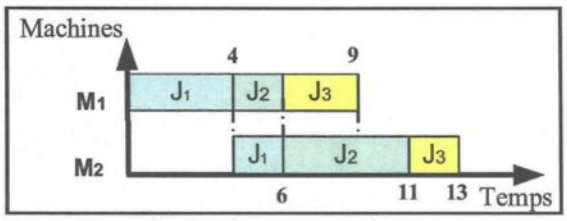


Figure 2-10: Digramme de Gantt

Le digramme de Gantt nous permet de représenter les différents ordonnancements de jobs sur les machines. Ces ordonnancements peuvent contenir des temps de latence. Ces derniers seront décrits par la suite.

2.4 Temps de latence

Jusqu'à un passé récent, il a été souvent supposé dans la littérature que les temps de latence τ_{ijk}, induits par exemple par le déplacement d'un job i d'une machine j à une autre k, sont négligeables. Soit l'exemple ci-dessous de flow-shop à deux machines et six jobs. Le tableau suivant présente les temps d'exécution de ces jobs sur les deux machines.

	J_1	J_2	J_3	J_4	J_5	J_6
M1	1	3	8	5	10	4
M2	4	2	5	1	9	6

Tableau 2-4 : Temps d'exécution du problème $m = 2$ et $n = 6$

Une solution optimale pour ce problème peut être obtenue comme illustrée par la Figure 2-11.

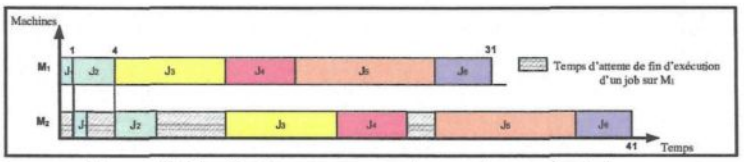


Figure 2-11 : Ordonnancement sans temps de latence

Comme on peut le remarquer sur le diagramme de la Figure 2-11, le temps pour transférer le job 2 de *M1* à *M2* est considéré comme nul. Cependant, en pratique, cette supposition est souvent non justifiée. En effet, la distance pouvant séparer les machines est en général importante. Autrement dit, le temps mis pour atteindre une machine à partir d'une autre peut être parfois plus important que les temps d'exécution eux mêmes. Dans ce cas de figure, il est nécessaire de prendre en compte ces temps de latence lors de la construction de la solution. En effet, considérons l'exemple d'un problème de *flow-shop* ci-dessus avec en plus les temps de latence associés aux jobs. Notons que pour ce problème à deux machines, le terme τ_{ijk} est réduit à τ_i.

	J_1	J_2	J_3	J_4	J_5	J_6
Temps de latence	1	3	2	1	3	2

Comme il fallait s'y attendre, le *makespan* a augmenté en considérant les temps de latence comme illustré par la Figure 2-12.

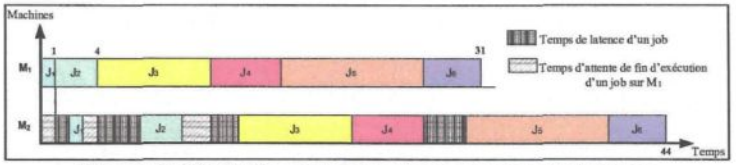


Figure 2-12: Ordonnancement avec temps de latence

Soit S un ordonnancement quelconque et C_{ij}, S_{hj} respectivement, la date de fin de l'exécution du job *j* sur la machine *i*, la date de début d'exécution d'un job *j* sur la machine *h* telle que $i \neq h$. Pour que cet ordonnancement soit valide, il est nécessaire que la relation $S_{hj} - C_{ij} \geq \tau_{ihj}$ soit vérifiée.

2.5 Brève introduction à la NP-complétude

La complexité temporelle d'un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats. Ce temps est

représenté par la fonction $T(n)$ où n est la taille des données d'entrée. Il existe deux approches pour évaluer le temps d'exécution d'un algorithme : la méthode empirique et mathématique. La méthode empirique consiste à coder et exécuter l'algorithme sur une batterie de données générées d'une manière aléatoire. À chaque exécution, le temps d'exécution de l'algorithme est mesuré. Enfin, une étude statistique est entreprise. Alors que la méthode mathématique consiste à faire le décompte des instructions de base exécutées par un algorithme. Cette manière de procéder est justifiée par le fait que la complexité d'un algorithme est en grande partie induite par l'exécution des instructions qui le composent. Cependant, pour avoir une idée plus précise de la performance d'un algorithme, il convient de signaler que la méthode expérimentale et la méthode mathématique sont en fait complémentaires. À ce moment, on s'intéresse qu'aux notions asymptotiques de la fonction $T(n)$ ou encore à sa complexité. Cet aspect est représenté par la notation O. Notons que les algorithmes de complexité polynomiale[4] sont dits efficaces.

Définition 4

Un problème de décision est un problème dont la solution est formulée en termes « oui » ou « non ».

Définition 5

Un algorithme a une complexité polynomiale si, pour tout n, l'algorithme s'exécute en moins de $c \times n^k$ opérations élémentaires (c et k étant des constantes).

Le but de la théorie de la complexité est la classification des problèmes de décision suivant leur degré de difficulté de résolution. Dans la littérature, il existe plusieurs classes de complexité, mais les plus connues sont classe P et la classe NP. Les problèmes appartenant à la classe P sont ceux dont le problème de décision correspondant donne la réponse correcte (« oui » ou « non ») en un temps polynomial. Les problèmes de la classe NP sont ceux dont on peut obtenir le résultat « oui » de leur problème de décision

[4] Les fonctions $T(n)$ associées sont polynomiales en fonction de la taille d'entrée n

selon un algorithme non déterministe en un temps polynomial. Les algorithmes non déterministes sont capables d'effectuer un choix judicieux parmi un ensemble d'alternatives.

Définition 6

Un problème A est réductible à un problème B s'il existe un algorithme « u » résolvant A qui utilise un algorithme « v » résolvant B.

Définition 7

Si l'algorithme résolvant A est polynomial, considérant les appels à l'algorithme résolvant B comme de complexité constante, la réduction est dite polynomiale. On dit que A est polynomialement réductible à B.

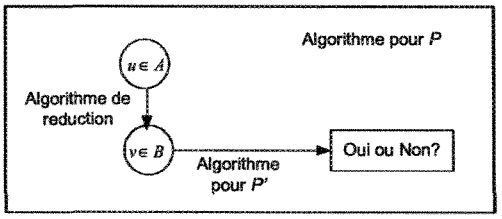


Figure 2-13 : Réduction polynomiale d'un problème

Définition 8

Un problème de décision est dit NP-complet si tout problème de la classe NP lui est polynomialement réductible.

Définition 9

Un problème d'optimisation est NP-difficile si le problème de décision qui lui correspond est NP-complet.

Un problème *L* est *NP*-difficile s'il est au moins aussi difficile que n'importe quel problème de la classe *NP*. Les problèmes *NP-complets* sont les problèmes les plus difficiles de la classe *NP*, en ce sens que si on trouve un algorithme déterministe de complexité polynomiale pour résoudre un problème *NP-complet*, alors tous les problèmes de la classe *NP* peuvent être résolus en un temps polynomial. Un tel algorithme n'a pas

encore été trouvé et la communauté scientifique accepte l'hypothèse $P \neq NP$ [Dorigo et Stützle, 2004].

Cook [Cook, 1971] a été le premier à avoir montré la *NP*-complétude d'un problème qu'est le problème de satisfiabilité.

Pour démontrer qu'un nouveau problème *A* est *NP – complet*, il suffit de montrer qu'il appartient à la classe *NP*, et de construire ensuite une réduction polynomiale d'un problème déjà *NP-complet* vers *A*.

2.6 Approches de résolution

En général, pour résoudre un problème donné, plusieurs solutions peuvent être mises à notre disposition. Il est alors intéressant d'évaluer le temps et/ou l'espace mémoire requis pour exécuter les algorithmes correspondant à ce problème. L'algorithme qui sera retenu sera le plus *efficace* (l'efficacité d'un algorithme est mesurée par sa complexité temporelle). Cette étape est celle de l'analyse des algorithmes.

Dans le cas où aucun algorithme n'est disponible, ou alors qu'on éprouve le besoin d'avoir une nouvelle solution autre celles dont on dispose, la question qui peut alors se poser est tout simplement de savoir comment on peut en concevoir une autre. Il est clair qu'il n'existe pas de recette pour résoudre un problème donné. Toutefois, des approches générales existent telles que la méthode vorace, la méthode diviser pour régner, la programmation dynamique, la réduction, la méthode de *branch and bound,* etc. Généralement, ces méthodes, quand elles sont utilisées, aboutissent souvent à des résultats. Dans ce cas de figure, il est souhaitable que la nouvelle solution soit plus performante que les solutions déjà existantes. Bien entendu, parmi les critères de performance, figurent le temps et l'espace mémoire. Si, maintenant, on est satisfait de la solution existante, il serait intéressant de savoir si le changement d'organisation des données peut améliorer ces performances [Rebaïne, 2000].

Il est clair qu'on ne peut pas améliorer indéfiniment une solution, en tout cas cela reste vrai pour les problèmes d'optimisation combinatoire. Il existe un point où cette amélioration s'arrête. Autrement dit, il est intéressant de savoir si la solution dont on dispose est optimale. On entend généralement dire par optimale, toute solution dont le temps d'exécution est minimal. Cette question est intimement liée à celle de la borne

inférieure : déterminer la borne inférieure d'un problème revient à démontrer qu'il ne pourrait exister d'algorithme dont le temps d'exécution est inférieur à cette borne. Si le problème est montré qu'il est dans la classe des problèmes «difficiles», ceci signifie qu'il est peu probable de trouver une solution polynomiale. Cela ne doit pas nous empêcher d'essayer de trouver une solution au problème en question. Comme illustré par la Figure 2-14, quatre approches peuvent être utilisées : *i*) essayer de trouver un algorithme pseudo-polynomial[5] ou de démontrer qu'il est *NP*-difficile au sens fort, *ii*) une méthode exacte, *iii*) une méthode approximative et *iv*) la relaxation [Blazewicz et *al*, 1994].

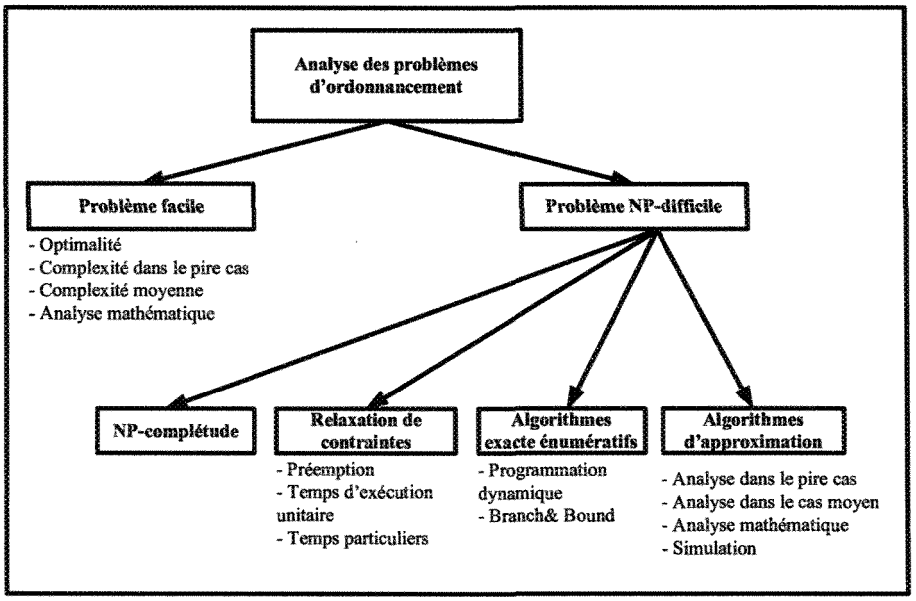


Figure 2-14: approches de résolution des problèmes d'ordonnancement

2.6.1. NP-complétude

Il s'agit dans ce cas de connaître le degré de difficulté du problème. Dans ce cas, on essaie soit de développer des algorithmes pseudo-polynomial, soit de démontrer qu'il est *NP*-difficile au *sens fort.*

[5] Un algorithme pseudo-polynomial est un algorithme dont la complexité est bornée par une fonction polynomiale de la représentation unaire des données du problème considéré.

2.6.2. Approches exactes

Dans le cas où l'on souhaite résoudre d'une manière exacte le problème considéré, des techniques comme la programmation linéaire, programmation dynamique ou la méthode de *branch and bound* sont souvent utilisées. Il est utile de rappeler que les temps d'exécution de ces méthodes deviennent de plus en plus prohibitifs à mesure que les données deviennent de plus en plus grandes.

2.6.3. Relaxation de contraintes

Cette approche consiste à relâcher certaines contraintes du problème pour bien distinguer la frontière qui puisse exister entre la résolution facile et difficile. Il est possible par exemple de permettre l'interruption des jobs ou de supposer que les temps d'exécution des jobs soient unitaires. Outre leurs intérêts théoriques, ces simplifications peuvent trouver des justifications dans de nombreuses applications pratiques [Rebaïne, 1997].

2.6.4. Approches approximatives

Dans le cas où on désire se contenter d'une solution approchée, les méthodes approximatives telles que les méthodes voraces et les méthodes itératives (recuit simulé, recherche avec tabous, etc.) sont utilisées. Ces méthodes consistent à déterminer des solutions proches de la solution optimale avec une complexité temporelle généralement raisonnable. Une étude expérimentale est généralement effectuée pour étudier la performance de ces algorithmes. Cette approche est détaillée à la section 2.7.

Il est utile de remarquer que pour les algorithmes dits voraces, une méthode mathématique est en général effectuée. Quand cela est possible, cette analyse calcule la distance qui sépare la solution générée par l'heuristique en question de la solution optimale. Cette distance est soit donnée d'une manière absolue soit relative. Pour ce faire, l'idée est de comparer la solution proposée par l'algorithme et la solution optimale. En effet, soient donc $Opt(I)$ la solution optimale pour l'instance I, $A(I)$ la solution produite par l'algorithme A et f la fonction objectif à optimiser. Le but est d'évaluer les quantités suivantes :

- Garantie absolue : Cette approche consiste à majorer, pour toute instance I, l'expression $|f(A(I)) - f(opt(I)|$. Notons que cette approche est rarement utilisée.
- Ratio de garantie : Cette approche consiste à majorer, pour toute instante I, la valeur de $\frac{|f(A(I)) - f(Opt(I))|}{f(Opt(I))}$ [Blazewicz et *al.*, 1994]. Plus ce ratio est proche de la valeur 0, meilleure est l'heuristique considérée.

2.7 Description des approches heuristiques et exacte

Dans cette partie, nous allons détailler l'approche heuristique et l'approche exacte, puisque ce sont ces deux approches que nous avons utilisées pour résoudre le problème de *flow-shop* avec des temps de latence, qui constitue, faut-il le rappeler, le problème central de ce mémoire.

2.7.1. Approche heuristique

Le mot "*heuristique*" est utilisé pour décrire un algorithme ou une procédure qui se base sur des expériences pratiques. Elle vise entre-autre à résoudre des problèmes d'optimisation *NP-difficile*.

Souvent tirées de l'expérience, les heuristiques trouvent leurs places dans les algorithmes qui nécessitent l'exploration d'un grand nombre de cas, car celles-ci permettent de réduire leur complexité moyenne en examinant d'abord les cas qui ont le plus de chances de générer la bonne réponse. Le choix d'une telle heuristique suppose auparavant la connaissance de certaines propriétés du problème à résoudre.

L'objectif d'une heuristique n'est pas d'obtenir un optimum global[6], mais seulement une « bonne » solution en un temps raisonnable. En effet, pour les problèmes *NP-difficile*, les méthodes exactes peuvent passer des semaines, voire des mois pour les résoudre. Les heuristiques peuvent en revanche produire de « bons » résultats sans garantie de l'optimalité. C'est ainsi que les programmes de jeu d'échec, par exemple, font appel de manière très fréquente à des heuristiques qui modélisent l'expérience d'un joueur.

[6] L'optimum global est la meilleure solution trouvée pour le problème considéré.

Dans ce qui suit, nous allons présenter et décrire les méthodes heuristiques les plus utilisées dans la littérature.

- **Heuristiques constructives**

L'approche, dite constructive, est probablement la plus ancienne et occupe traditionnellement une place très importante en optimisation combinatoire et en intelligence artificielle. Une heuristique constructive construit pas à pas une solution de la forme $s = (<V_1,v_1> <V_2,v_2>,..,<V_n,v_n>)$. Partant d'une solution partielle initialement vide $s = ( )$, elle cherche à étendre à chaque étape la solution partielle $s = (<V_1,v_1>,..., <V_{i-1},v_{i-1}>)$ $(i \leq n)$ de l'étape précédente. Pour cela, elle détermine la prochaine variable V_i, choisit une valeur v_i dans D_i et ajoute $<V_i,vi>$ dans s pour obtenir une nouvelle solution partielle $s = (<V_1,v_1>...<V_{i-1},v_{i-1}> <V_i,v_i>)$. Ce processus se répète jusqu'à ce que l'on obtienne une solution complète.

La performance de ces méthodes dépend largement de leur capacité à exploiter les connaissances du problème. Parmi les solutions constructives les plus utilisées, on distingue l'approche gloutonne.

Un algorithme glouton fait toujours le choix qui semble être le meilleur localement, dans l'espoir que ce choix mènera à la solution optimale globale. Les méthodes gloutonnes sont généralement rapides, mais fournissent le plus souvent des solutions de qualité médiocre. Elles ne garantissent l'optimum que dans des cas particuliers.

- **Heuristiques d'amélioration locale**

Le principe d'une méthode d'amélioration locale est le suivant: à partir d'une solution de départ x_0, considérée temporairement comme étant la valeur minimale x_{min}, on engendre par transformations élémentaires une suite finie de voisins. C'est pour cela que de telles méthodes sont aussi appelées méthodes par voisinage. En effet, pour utiliser de telles méthodes, il faut introduire une structure de voisinage qui consiste à spécifier un voisinage pour chaque solution. Le voisinage d'une solution x est représenté par l'ensemble des solutions x' atteignables à partir de x en y apportant une ou plusieurs modifications.

La recherche locale vise à déterminer une solution $s(x)$ dans le voisinage de la solution courante x, telle que $f(s(x)) < f_{min}$ (f_{min} désigne la valeur minimale courante de f ou encore on peut avoir $f(s(x)) > f_{max}$ et f_{max} désigne la valeur maximale courante de f). La méthode consiste à engendrer, à chaque itération, un N-échantillon, suivant un procédé aléatoire ou cyclique, ou suivant une loi de distribution donnée, dans le voisinage de la solution courante x. La fonction objectif f est évaluée en chaque point de l'échantillon, et la solution x' correspond à la plus petite valeur de f obtenue, $f(x) = f(s(x)) = \min[f(s_i(x))]$.

Cette nouvelle valeur $f(x')$ est comparée à la valeur minimale courante f_{min}. Si elle est meilleure, cette valeur est enregistrée, ainsi que la solution correspondante, et le processus se répète, ainsi de suite. Sinon l'algorithme prend fin. Cependant, rien ne nous garantit qu'on obtienne ainsi la meilleure solution (au plus un optimum local).

2.7.2. Métaheuristiques

Les métaheuristiques sont généralement des algorithmes stochastiques, qui progressent vers un optimum par échantillonnage d'une fonction objectif. Elles utilisent des méthodes génériques pouvant optimiser une large gamme de problèmes différents, sans nécessiter de changements profonds dans l'algorithme employé. En pratique, elles sont utilisées sur des problèmes ne pouvant être optimisés (d'une manière efficace) par des méthodes mathématiques.

Certaines métaheuristiques sont théoriquement robustes, c'est-à-dire convergentes (capable de trouver l'optimum global si le temps de calcul tend vers l'infini) sous certaines conditions. Cependant, ces conditions sont rarement vérifiées en pratique. On distingue plusieurs métaheuristiques telles que : *la recherche avec tabou, le recuit simulé, l'algorithme génétique, la recherche par colonies de fourmis*, etc.

On peut regrouper les métaheuristiques en deux grandes classes : des métaheuristiques simples et des métaheuristiques à population. De même, chacune de ces classes peuvent être subdivisées en métaheuristiques constructives et amélioratives. La Figure 2-15 illustre une classification des métaheuristiques.

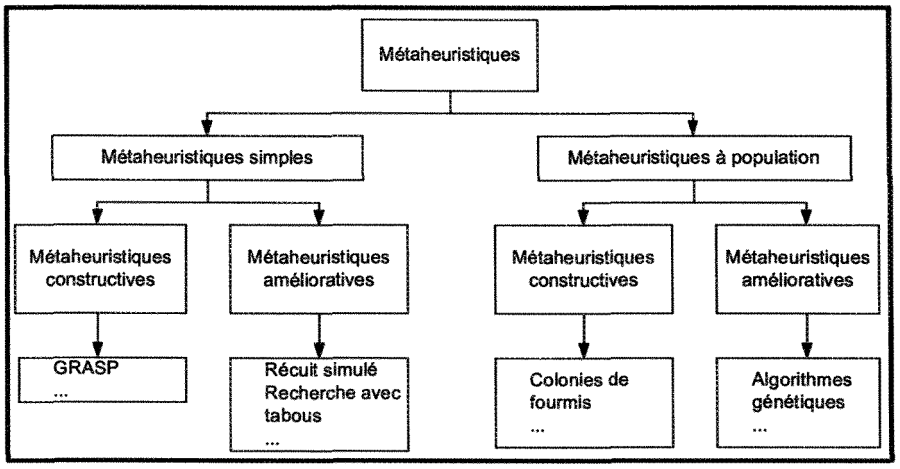


Figure 2-15 : Classification des métaheuristiques

- **Algorithmes de recuit simulé**

Le recuit simulé (*RS*) trouve ses origines dans la thermodynamique. Cette méthode est issue d'une analogie entre le phénomène physique de refroidissement lent d'un corps en fusion, qui le conduit à un état solide, de basses énergies. Il faut abaisser lentement la température, en marquant des paliers suffisamment longs pour que le corps atteigne l'équilibre thermodynamique.

L'analogie exploitée par le recuit simulé consiste à considérer une fonction f à minimiser comme fonction d'énergie, et une solution x peut-être considérée comme un état donné de la matière dont $f(x)$ est l'énergie. Le recuit simulé exploite généralement le critère défini par l'algorithme de Metropolis et *al.* [1953] pour l'acceptation d'une solution obtenue par perturbation de la solution courante. Pour une température T donnée, à partir d'une solution courante x, on considère une transformation élémentaire qui changerait x en $N(x)$. Si cette perturbation induit une diminution de la valeur de la fonction objectif f, $\delta = f(N(x)) - f(x) < 0$, elle est acceptée. Dans le cas contraire, si $\delta = f(N(x)) - f(x) \geq 0$, la perturbation est acceptée tout de même avec une probabilité $p = exp(-\delta/T)$ [Metropolis et *al.*, 1953]. Le paramètre de contrôle T est la "température" du système, qui influe sur la probabilité d'accepter une solution plus mauvaise.

L'algorithme équivaut alors à une marche aléatoire dans l'espace des configurations possibles. Cette température est diminuée lentement au fur à mesure du déroulement de

l'algorithme pour simuler le processus de refroidissement des matériaux, et sa diminution est suffisamment lente pour que l'équilibre thermodynamique soit maintenu. Cette diminution est indépendante de nombre d'itérations.

- **Algorithmes génétiques**

Les principes fondamentaux des algorithmes génétiques (*AG*) ont été exposés par Holland [1975]. Ces algorithmes s'inspirent du fonctionnement de l'évolution naturelle, notamment la sélection de Darwin, et la procréation selon les règles de Mendel. Ce type d'algorithme est aussi appelé algorithmes évolutifs.

Le principe de fonctionnement d'un algorithme génétique est décrit comme suit ; On part d'une population de solutions. Ces solutions sont aussi appelés individus ou chromosomes. On associe à chaque individu son évaluation qui représente sa fonction objectif. On sélectionne des individus de cette population appelée génération parents. La sélection est effectuée par groupe de 2 individus formant ainsi ce qu'on appelle les deux parents. Une sélection est dit élitiste si on garde à chaque fois les individus les plus forts. Dans ce cas, pour chaque individu, la probabilité d'être sélectionnée est proportionnelle à son adaptation à son environnement. Après avoir choisi les deux parents, on passe à la phase de croisement et de mutation. Cette phase est appelée application des opérateurs génétiques. Le croisement consiste à garder une partie du patrimoine génétique des deux parents. Ainsi une partie du père et une autre de la mère seront hybridées ensemble pour former un nouvel individu dit enfant. La mutation consiste à apporter de la diversité. Ainsi, certains enfants seront mutés : il y aura un changement dans leur patrimoine génétique. Il existe plusieurs variantes de ces opérateurs génétiques et les recherches ont autant concerné le croisement que la mutation. Mais comme toute métaheuristique, l'*AG* a été surtout appliqué au *TSP*. Dans un *AG*, on attribut à chaque opérateur une probabilité qui représente sa fréquence d'application. Ainsi si on dit que la probabilité associée à la mutation est de 33 %, alors un enfant sur trois en sera affecté.

Après l'application des opérateurs génétiques, vient la phase de remplacement. Cette phase consiste à appliquer une stratégie qui permet d'introduire les enfants dans la population. Plusieurs schémas peuvent avoir lieu : changer toute la population, garder les

meilleurs parents et enfants, garder les meilleurs parents et choisir aléatoirement les enfants, etc. La représentation des individus représente aussi une autre étape d'un *AG*. On peut avoir recours à une représentation binaire, mais il est tout à fait possible d'utiliser une représentation scalaire. Finalement, on associe à un *AG*, un critère d'arrêt qui peut être le nombre de générations produites, le nombre d'évaluation ou tout simplement le temps. En répétant cette opération à travers plusieurs générations, où une génération représente une itération des différentes étapes d'un *AG*, nous remarquons que les différentes caractéristiques de chaque population ou génération présentent un aspect adaptatif à l'environnement. Donc, tant que la population est bien choisie nous aurons des solutions très proches de la solution optimale. Le principal désavantage de cette métaheuristique est l'ensemble de paramétrage à effectuer pour l'obtention d'un algorithme efficace. Cet ensemble varie d'un problème à un autre.

La plupart des recherches concernant cette approche sont concentrées sur les recherches des règles empiriques. Leur but est de déterminer une solution qui rend cet algorithme plus performant et facilement implémentée.

1. Population de base générée aléatoirement
Générer P individus avec une représentation donnée où P est la taille de la population. 1 individu correspond à 1 chromosome.

2. Évaluation
On associe à chaque individu *I* une évaluation représentant la fonction objectif *F(I)*

3. Sélection
Sélectionner un ensemble de parents aléatoirement ou avec une roulette biaisée, on parle alors d'élitisme

4. Croisement et mutation
Chaque couple de parents donne 2 enfants.
- Appliquer le croisement (*Crossing-over*) avec une certaine probabilité.

L'emplacement du *crossing-over* est choisi aléatoirement.
- Appliquer la mutation avec une certaine probabilité.

Permutation de deux emplacements au hasard
- Ces deux probabilités peuvent être fixes ou évolutive (auto-adaptation).

5. Remplacement
Choisir une politique de remplacement afin d'intégrer les nouveaux enfants dans la population

6. Arrêt
Itérer les étapes de 2 à 5 jusqu'à avoir atteint un critère d'arrêt (nombre d'évaluations, temps, etc.)

Algorithme 2-1 : L'algorithme génétique

- **Algorithmes de colonies de fourmis**

L'optimisation par colonies de fourmis a été élaborée initialement pour résoudre le problème du voyageur du commerce [Dorigo, 1995]. Pour la résolution de ce problème nous disposons de *m* fourmis qui sont éparpillées sur différentes villes au départ (le choix peut être aléatoire). À chaque fois qu'une fourmi voudra visiter une autre ville, elle devra en choisir une, avec une certaine probabilité, parmi les villes qu'elle n'a pas encore visitées. Ce choix se base sur la distance qui la sépare de la ville. Chaque fourmi est forcée de parcourir toutes les villes et ce, en ajoutant chaque ville visitée à une liste dite taboue. La fourmi n'aura pas le droit de revisiter une ville qui se trouve dans cette liste. Toute fourmi qui achève un tour complet, laisse une valeur sur chaque arc (on parle alors de mémoire distribuée à long terme) qu'elle a parcouru. Cette valeur est appelée intensité de phéromone. Le parcours de toutes les villes forme une fourmi. La construction de toutes les fourmis forme un cycle. Un cycle contient *n* itérations, *n* étant le nombre de villes. À chaque cycle, les listes taboues de chaque fourmi sont effacées. La liste taboue est une mémoire pour conserver pendant un moment la trace des dernières meilleures ville déjà visitées. Après *j* itérations, toutes les fourmis emprunteront le même chemin. C'est alors que l'algorithme s'arrêtera; ça sera le plus court chemin. On appelle cet état l'état de stagnation.

L'optimisation par colonie de fourmis est une métaheuristique constructive qui aborde plusieurs solutions à la fois. Elle est caractérisée par sa rapidité à trouver la meilleure solution. Elle possède aussi une mémoire à long terme qui lui permet de converger vers la meilleure solution. Envisagée pour résoudre le *TSP* [Dorigo, 1995], elle a pu être utilisée pour résoudre d'autres problèmes et fait preuve de très bons résultats [Dorigo et Stützle, 2004].

- **Algorithmes de recherche avec tabous**

L'algorithme de recherche avec tabous (*RT*) est une métaheuristique développée par Glover [1986], et indépendamment par Hansen [1986]. Cette méthode combine une procédure de recherche locale avec un certain nombre de règles et de mécanismes permettant à celle-ci de surmonter l'obstacle des optima locaux, tout en évitant de cycler.

La méthode de recherche avec tabous peut être vue comme une généralisation des méthodes d'amélioration locales. En effet, en partant d'une solution quelconque s, appartenant à l'ensemble de solutions X, on se déplace vers une solution s' située dans le voisinage $N(s)$ de s.

Afin de choisir le meilleur voisin s' dans $N(s)$, l'algorithme évalue la fonction objectif f en chaque point s', et retient le voisin qui améliore la valeur de la fonction objectif f, ou au pire celui qu'il la dégrade le moins. L'originalité de cette méthode par rapport aux méthodes locales, qui s'arrêtent dès qu'il n'y a plus de voisin s' permettant d'améliorer la valeur de la fonction objectif f, réside dans le fait que l'on retient le meilleur voisin, même si celui-ci est plus mauvais que la solution d'où l'on vient. Ce critère autorisant les dégradations de la valeur de la fonction objectif évite à l'algorithme d'être piégé dans un minimum local. Mais il induit un risque de cycler. Pour régler ce problème, l'algorithme a besoin d'une mémoire pour conserver pendant un moment la trace des dernières meilleures solutions déjà visitées. Elles sont stockées dans une liste de longueurs L donnée, appelée liste taboue. Une nouvelle solution n'est acceptée que si elle n'appartient pas à cette liste taboue. Ce critère d'acceptation d'une nouvelle solution évite d'avoir un cycle durant la visite d'un nombre de solutions au moins égal à la longueur de la liste taboue, et il dirige l'exploration de la méthode vers des régions du domaine de solutions non encore visitées. La liste taboue est généralement gérée comme une liste circulaire de longueur fixe, mais elle n'empêche pas de cycler. Donc, le choix d'une bonne longueur n'est pas facile à déterminer. Pour améliorer ce concept, quelques chercheurs comme Glover [1990], Skorim-Kapov [1990] et Taillard [1991] ont proposé de varier la longueur de la liste taboue en cours d'exploration. On peut aussi fixer la taille de la liste, mais on tire aléatoirement la longueur active de la liste indiquant quels éléments tabous seront maintenus.

Le statut tabou interdit parfois d'explorer certaines solutions qui ne sont pas visitées auparavant. Il faut donc corriger ce défaut en utilisant le concept de critère d'aspiration. Ce dernier est une fonction qui révoque le statut tabou d'une transformation à condition qu'elle donne une solution intéressante ou encore elle évite le risque de cycler. Cette correction permet aussi de revenir à une solution déjà visitée et de

redémarrer la recherche dans une autre direction. Le processus de la recherche s'arrêtera si l'un des critères d'arrêt est vérifié. Les critères d'arrêt les plus courants sont :

- Arrêt de l'exploration dès l'obtention d'une solution optimale (si elle est connue) ou d'une solution ayant une valeur prédéterminée.
- Arrêt après un certains nombre d'itérations ou un intervalle de temps prédéfinis.
- Arrêt après un certain nombre d'itérations sans amélioration de la meilleure solution courante.

Dans la version standard de la recherche avec tabous, la fonction objectif doit être évaluée pour chaque voisin de la solution courante s. Mais il s'est avéré que cette évaluation peut être très coûteuse. Donc, pour remédier à ce problème, on considère un sous ensemble $N'(s) \subset N(s)$ choisi aléatoirement. Ensuite, on choisit la prochaine solution dans $N'(s)$. Cette alternative est très rapide et le fait de choisir aléatoirement les échantillons permet à la liste taboue d'avoir un mécanisme anti-cyclage. Mais, on peut manquer de bonnes solutions, voire même l'optimum.

Début
s :=Solution aléatoire
f^* :=$f(s)$
s^* :=s
Taboue :=liste de solutions $N(s)$, de longueur L
Taboue :=vide
Répéter
Générer un T-échantillon tel que $n_i(s) \in$ voisinage $N(s)$ et $\{s, n_i(s)\} \notin$ Taboue
$f(n(s)) = \min_{1 \le i \le T} [f(n_i(s))]$
Ajouter ($\{s, n_i(s)\}$, Taboue)
$s := n(s)$
Si $f(s) < f^*$
$f^* := f(s)$
$s^* := s$
Fin Si
Jusqu'à la condition soit satisfaite
Fin

Algorithme 2-2 : L'algorithme de recherche avec tabous

La version standard de la recherche avec tabous peut résoudre plusieurs problèmes difficiles. Mais on peut l'améliorer en lui ajoutant quelques stratégies de recherche. Les stratégies les plus utilisées sont :

- L'intensification est un concept de recherche qui permet d'intensifier l'effort de recherche dans une région de *s* qui paraît prometteuse. Pour appliquer ce concept, on arrête périodiquement la recherche pour effectuer une phase d'intensification d'une durée limitée. Ce concept est basé sur des mémoires à court terme. L'intensification est utilisée pour l'implémentation de plusieurs algorithmes de la recherche avec tabous. Elle a plusieurs avantages tels que : la fixation des bonnes caractéristiques des solutions, de biaiser la fonction objectif pour favoriser certains types de solutions, changer l'heuristique interne par une heuristique plus puissante, etc. Malgré tous ces avantages, elle n'est pas toujours nécessaire. On peut rencontrer des cas où la simple recherche est efficace.
- La diversification est un concept diamétralement opposé à l'intensification. Elle permet de focaliser les recherches dans une portion restreinte de *s* dont le but est de rediriger le processus de recherche vers d'autres régions non encore explorées. La diversification est basée sur des concepts de mémoires à long terme puisqu'elle garde en mémoire toutes les itérations effectuées. On peut utiliser deux techniques de diversification. La première technique consiste à redémarrer la recherche dans la même portion en utilisant quelques composantes de la solution courante ou la meilleure solution trouvée. La deuxième technique intègre des méthodes de diversification dans le processus de recherche.

Ces stratégies de recherches permettent l'amélioration de l'algorithme de recherche avec tabous.

Notons qu'il existe plusieurs autres métaheuristiques telles le *Scatter Search* [Glover, 1997] ou le *GRASP* [Feo et Resende, 1995]. Cependant, ces dernières années, la recherche s'oriente de plus en plus vers l'hybridation des métaheuristiques [Talbi, 2002].

2.7.3. Approche exacte : méthode de branch and bound

Parmi les méthodes de résolution des problèmes d'ordonnancement, nous avons cité plus haut la méthode de *branch and bound.*

La résolution optimale de problèmes d'optimisation combinatoire *NP-difficiles* nécessite une mise en œuvre de méthodes plus complexes. Il est possible d'énumérer toutes ces solutions, et ensuite d'en prendre la meilleure. L'inconvénient majeur de cette approche est le nombre prohibitif des solutions que cela peut générer.

La procédure par évaluation et séparation progressive (en anglais, *branch and bound*) est parmi l'une de ces méthodes d'énumération. C'est une méthode générique de résolution de problèmes d'optimisation combinatoire. On parle aussi de méthode d'énumération implicite : toutes les solutions possibles du problème peuvent être énumérées, mais l'analyse des propriétés du problème permet d'éviter l'énumération de larges classes de solutions ne pouvant pas mener à l'optimum. Dans ces algorithmes, seules les solutions potentiellement bonnes sont donc énumérées. De ce fait, on arrive souvent à obtenir la solution recherchée en un temps raisonnable, bien qu'on puisse se trouver dans le pire cas où on explore toutes les solutions du problème.

La méthode de *branch and bound* a été utilisée pour la première fois par Dantzig, Fulkerson et Johnson [1959], lors de la résolution du problème de voyageur de commerce (*TSP).* Eastman [1958] a pu résoudre ce problème en utilisant une approche énumérative, semblable à celle de Dantzig, Fulkerson et Johnson.

La méthode de *branch and bound* utilise deux concepts: la séparation et l'évaluation. La séparation consiste à diviser un ensemble de solutions en sous-ensembles plus petits. Quant à l'évaluation, elle consiste à borner ou exclure des solutions partielles.

- **Séparation**

La phase de séparation consiste à diviser le problème en un certain nombre de sous problèmes qui ont chacun leur ensemble de solutions réalisables de telle sorte que tous ces ensembles forment un recouvrement (idéalement une partition) de l'ensemble S (l'ensemble de solutions réalisables). Ainsi, en résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Ce principe de séparation peut être appliqué de manière récursive à chacun des sous-

ensembles de solutions obtenus, et ceci tant qu'il y a des ensembles contenant plusieurs solutions. Les ensembles de solutions (et leurs sous-problèmes associés) ainsi construits ont une hiérarchie naturelle en arbre. Cette arborescence est appelée *l'arbre de recherche.*

- **Évaluation**

L'évaluation d'un nœud de l'arbre de recherche a pour but de déterminer l'optimum de l'ensemble des solutions réalisables associées au nœud en question ou, au contraire, de prouver mathématiquement que cet ensemble ne contient pas de solution intéressante pour la résolution du problème. Lorsqu'un tel nœud est identifié dans l'arbre de recherche, il est inutile d'effectuer la séparation de son espace de solutions.

On peut distinguer pendant le déroulement de l'algorithme trois types de nœuds dans l'arbre de recherche : le *nœud courant* qui est le nœud en cours d'évaluation, des *nœuds actifs* qui sont dans la liste des nœuds qui doivent être traités, et des *nœuds inactifs* qui ont été élagués au cours du calcul.

Pour appliquer la méthode de *branch and bound,* au moins un des points suivants doit être vérifié [Bouzgarrou, 1998]:

- Le calcul d'une borne inférieure: la borne inférieure permet d'éliminer certaines solutions de l'arbre de recherche afin de faciliter l'exploration du reste de l'arbre et de faire converger la recherche vers la solution optimale.
- Le calcul d'une borne supérieure : la borne supérieure peut être une heuristique qui permet d'élaguer certaines branches de l'arbre de recherche.
- Application des règles de dominance : dans certains cas et pour certains problèmes, il est possible d'établir des règles qui permettent d'élaguer des branches de l'arbre de recherche.
- Stratégie de recherche : Il existe plusieurs techniques pour l'exploration de l'arbre de recherche. On peut utiliser soit une exploration en profondeur d'abord (Depth First Search - DFS), soit une exploration en largeur d'abord (Breadth First Search-BFS), soit encore une recherche qui utilise une file de priorité.

- Choix du critère d'évaluation : Lors d'une application de *branch and bound*, on a besoin d'avoir une fonction qui permet d'évaluer chaque nœud traité, et éventuellement élaguer ceux qui sont inutiles. De même, un nœud peut être élagué dans trois cas possibles : dans le premier cas, on arrive à un stade où la valeur de la borne inférieure d'un nœud courant est plus grande ou égale à la valeur de la borne supérieure qu'on avait établie auparavant. Dans le deuxième cas, la solution n'est pas réalisable, l'un des critères d'évaluation n'a pas été respecté. Le troisième cas est le cas où on a obtenu une solution réalisable, tous les critères sont valides, mais la solution obtenue est supérieure à la borne inférieure.

L'algorithme de *branch and bound* commence par examiner le problème de départ, la racine de l'arbre, avec son ensemble de solutions. Ensuite, on applique des procédures de bornes inférieures et supérieures à la racine. Si ces deux bornes sont égales, alors une solution optimale est trouvée, et on arrête l'exploration. Sinon, on divise l'ensemble des solutions en deux ou plusieurs sous problèmes qui vont être par la suite, les enfants de la racine et les nouveaux sous problèmes. Cette méthode sera appliquée de façon récursive à ces sous problèmes engendrant ainsi une arborescence. A chaque fois qu'on obtient une meilleure solution, elle sera utilisée pour élaguer toute sa descendance. La recherche continue jusqu'à ce que tous les nœuds sont, soit explorée ou élaguée [Bouzgarrou, 1998]. Gourmand en consommation temporelle, il est possible d'arrêter l'algorithme de *branch and bound* avant d'atteindre la solution optimale. En effet, il est possible d'arrêter la procédure après avoir visité un certains nombres de nœuds ou après une certaine période fixée de temps [Bouzgarrou, 1998].

La méthode de *branch and bound* a été utilisée pour résoudre plusieurs problèmes tels le problème du sac à dos (*Knapsack Problem*) [Kozanidis et *al.*, 2002], celui du voyageur de commerce (*Traveling Salesman Problem*) [Lawler et *al.*, 1985], celui d'affectation (*Assignment Problem*), etc.

Chapitre 3 : Résolution du problème de flow-shop

3.1 Introduction

Nous nous concentrons dans ce chapitre au problème de *flow-shop*. Dans un premier lieu nous étudions ces différentes propriétés et restrictions. Ensuite nous explorons l'algorithme de Johnson avec et sans temps de latence pour terminer ce chapitre avec une revue de littérature concernant la résolution du problème de *flow-shop*.

3.2 Le problème de flow-shop

Un problème d'ordonnancement *flow-shop* met en œuvre m machines, notées $M_1, M_2, ..., M_m$, et n jobs, notés $j_1, j_2, ..., j_n$. Chaque job doit être exécuté, au plus une seule fois, sur M_1, puis M_2, et ainsi de suite, jusqu'à ce qu'il soit exécuté sur la dernière machine M_m dans cet ordre. Chaque job est donc composé de m opérations élémentaires $O_{j1}, O_{j2}, ..., O_{jm}$. Pour chaque opération O_{ji}, on désigne par P_{ij} son temps d'exécution.

Sur cette définition du problème de *flow-shop* peuvent venir se greffer des nombreuses notions pour rendre compte des contraintes réelles d'un atelier. Nous ne présentons ici que les plus usuelles, même si cette liste non exhaustive peut être très largement complétée :

- Les stocks inter-machines : les jobs transitent par un stock limité pour aller d'une machine à l'autre. La politique de gestion de ce stock, ainsi que le nombre de jobs qui peuvent y être entreposées modifient la structure du problème.
- Les temps de montage : lorsqu'une machine finit d'exécuter un job pour en commencer une autre, elle peut avoir besoin de subir un changement quelconque de son mode opératoire. La durée de ces changements peut alors être prise en compte dans les solutions.
- Les moyens et les temps de transports ou de latence : pour qu'un job puisse passer d'un centre à un autre, ce dernier doit emprunter un moyen de transport. La distance

entre les centres, et donc le temps de transport des opérations d'un centre à l'autre, est une contrainte à prendre en compte lors de la construction d'une solution.

- La disponibilité des machines : les machines peuvent être soumises à des périodes d'inactivité qui peuvent être des périodes de maintenance.
- La nature des machines : certaines machines d'un même centre peuvent par exemple être plus rapides que d'autres

Pour les besoins de notre étude, nous supposons également ce qui suit :

- La non préemption des opérations : une fois que l'exécution d'un job a débuté sur une machine, celle-ci ne peut pas être interrompue. Aucune opération ne peut commencer sur cette machine avant la fin de l'opération en cours.
- Les durées des opérations sont entières. Cette hypothèse n'est pas réductrice en général. Elle permet simplement de traiter le problème en évitant la manipulation de nombres réels.
- Les machines ne peuvent exécuter qu'une opération à la fois. Ces machines sont disponibles sans restriction du début à la fin de l'ordonnancement.

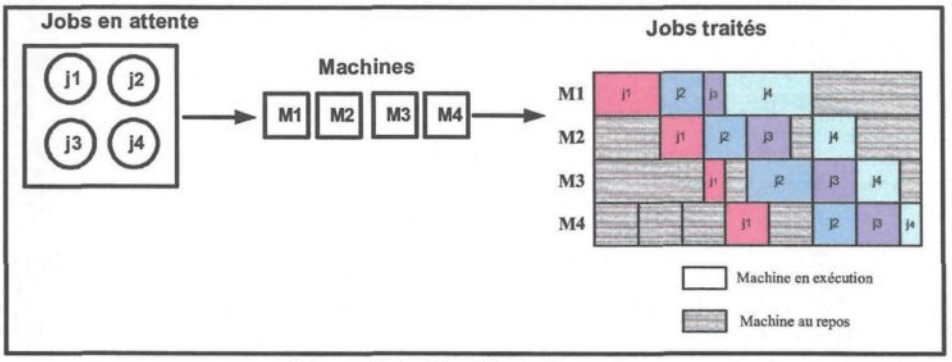


Figure 3-1 : Modèle de flow-shop

L'objectif est d'optimiser le *makespan* $C_{\max} = \max\{C_j ; j = 1, \ldots, n\}$, où C_j est la date de fin d'exécution du job *j*.

La classification des problèmes d'ordonnancement distingue les problèmes statiques versus les problèmes dynamiques d'une part et d'autre part déterministes versus stochastiques. Dans le cas où le problème à traiter est statique, il faut spécifier l'ensemble des *n* jobs qui est figé dans le temps. Dans le cas déterministe, les données du problème,

telles que les temps d'exécution des jobs, sont connues d'une manière précise. Dans ce mémoire, on restreint notre étude au problème du *flow-shop* statique et déterministe.

3.2.1. Propriétés du problème de *flow-shop*

Les premières recherches effectuées sur le problème de *flow-shop* se sont essentiellement focalisées sur les problèmes de permutations. Sous-classe du problème de *flow-shop*, un problème de *flow-shop* de permutation est un problème de *flow-shop* où l'ordre de passage des jobs sur chacune des machines est identique. Résoudre ces problèmes revient à trouver une permutation $\pi = (\pi(1), \pi(2), ..., \pi(n))$ minimisant le *makespan*.

La restriction aux *flow-shop* de permutation permet de simplifier la structure des solutions et des preuves [Pinedo, 2002]. En effet, d'un point de vue pratique, l'ordonnancement obtenu a une structure simple pouvant être facilement implémentée.

Pour des temps de latence nuls, le problème de *flow-shop* est dominé par la permutation pour un nombre de machines $m \leq 3$. Johnson [Johnson, 1954] a prouvé que le problème à deux machines est résoluble en $O(n \log n)$. Le problème devient *NP*-difficile au sens fort pour $m \geq 3$ [Pinedo, 2002]. Concernant le problème de permutation de *flow-shop* avec des temps de latence, Mitten [1959] a montré que ce problème à deux machines peut être résolu en $O(n \log n)$. Pour le cas unitaire, il est résolu en $O(n)$. Pour $m = 3$, le problème peut être résolu en $O(n \log n)$ et en $O(n^2)$ pour $m = 4$. La complexité de ce problème pour $m \geq 5$ reste encore inconnue [Munier-Kordon et Rebaïne, 2006]. Yu et *al.* [2004] ont montré que la minimisation du *makespan* pour le cas unitaire du problème du *flow – shop* à deux machines avec des temps de latence est *NP*-difficile au sens fort. Rebaïne [2005] a montré que, dans le cas des temps d'exécution unitaires et m machines, la meilleure permutation est pire d'un facteur m que le meilleur ordonnancement d'un *flow-shop*. Cependant, il existe des cas où la permutation est encore dominante [Yu, 1996].

3.2.2. Problème à deux machines avec des temps d'exécution quelconques

Pour des temps de latence nuls et $m = 2$, le problème de *flow – shop* est une permutation. Quand ces temps ne sont pas nuls, alors la dominance des permutations n'est plus vraie, même pour $m = 2$ et des temps d'exécution unitaires. En effet, soit l'instance suivante à 4 jobs et des temps d'exécution unitaires à deux machines [Rayward-Smith et Rebaïne, 1997].

Job *j*	1	2	3	4
τ_j	5	3	3	1

Tableau 3-1 : Temps d'exécution du problème $n = 4$ avec des temps de latences

La séquence optimale de *flow-shop* produit un *makespan* de valeur 8, alors que, dans le cas d'un *flow-shop* de permutation, la valeur du *makespan* optimal est 11.

Théorème 2 [Rebaïne, 2005]

Si $C_{\max}(flow-shop)$ *et* $C_{\max}(permutation)$ *sont les makespan respectifs du problème de flow – shop et de flow-shop de permutation alors le ratio suivant est vérifié:* $\frac{C_{\max}(permutation)}{C_{\max}(flow-shop)} \leq 2$ *; et la borne est atteinte.*

Dans le cas unitaire, les résultats restent presque similaires, comme résumés ci-dessous.

Théorème 3 [Rebaïne, 2005]

Si le nombre de machine est limité à 2 et les temps d'exécution unitaires, alors $\frac{C_{\max}(permutation)}{C_{\max}(flow-shop)} \leq 2-\frac{3}{n+2}$ *et la borne est atteinte.*

Théorème 4 [Rebaïne, 2005]

Si le nombre de machine est m et les temps d'exécution sont unitaires, alors $\frac{C_{\max}(permutation)}{C_{\max}(flow-shop)} \leq m$ *et la borne est atteinte.*

3.2.3. Algorithme de Johnson sans temps de latence

Johnson [1954] considère un problème de *flow-shop* à deux machines. L'objectif est la minimisation du *makespan*. Il proposa un algorithme et démontra que la même permutation des jobs pouvait être utilisée sur les deux machines. Il démontra aussi que s'il y a m machines, un ordonnancement optimal consiste à exécuter des séquences identiques de jobs sur les deux premières et deux dernière machines.

Théorème 5 [Johnson, 1954]

Dans un problème de flow-shop à deux machines, un job i doit précéder un job j dans une séquence optimale si $\min(A_i, B_j) \leq \min(A_j, B_i)$ *; A_j et B_j étant les temps d'exécution du job j respectivement sur la machine A et B.*

Il est facile de déduire de ce théorème l'algorithme suivant, connu sous le nom d'algorithme de Johnson. Cet algorithme consiste à diviser l'ensemble des jobs à traiter en deux sous-ensembles U et V : U contient les jobs i tels que $p_{1i} \leq p_{2i}$ et V contient les jobs i tels que $p_{1i} > p_{2i}$. Ces deux ensembles sont triés : U suivant l'ordre croissant des p_{1i} et V suivant l'ordre décroissant des p_{2i}. U et V sont ensuite fusionnés de manière à ajouter les jobs de V à la fin de l'ensemble U. Ces jobs seront ensuite exécutés dans cet ordre sur la première machine *M1* puis sur la deuxième machine *M2*. En d'autres termes, les jobs ayant les plus petits temps d'exécution sur la première machine seront exécutés en premier ; ceux qui ont les plus petits temps sur la deuxième machine seront exécutés à la fin. L'algorithme de Johnson est le suivant :

Début

Tant que la liste des jobs non vide

Placer dans l'ensemble U les jobs pour lesquels $p_{1j} <= p_{2j}$
Placer dans l'ensemble V les jobs pour lesquels $p_{1j} > p_{2j}$

Fin tant que

Ordonnancer les jobs de l'ensemble U dans l'ordre décroissant des p_{1j} ;
Ordonnancer les jobs de l'ensemble V dans l'ordre décroissant des p_{2j} ;
On fusionne les deux ensembles U et V . On exécute la permutation obtenue sur les deux machines tout en respectant le même ordre.

Fin

Algorithme 3-1 : L'algorithme de Johnson pour le problème de flow-shop à deux machines

Exemple 3-1

L'exemple ci-dessous illustre l'algorithme de Johnson. Soit 4 jobs j_1, j_2, j_3 et j_4 et leurs temps d'exécution respectifs sur les deux machines M_1 et M_2.

	J_1	j_2	J_3	j_4
M1	1	3	8	5
M2	4	2	7	6

Tableau 3-2 : Temps d'exécution du problème $n = 4$ et m = 2 de l'Exemple 3-1

On affecte les différents jobs à la liste U et V tout en respectant l'Algorithme 3-1. On aura :

- La liste U va contenir les jobs : j_1, j_4, j_3
- La liste V va contenir les jobs : j_2

L'ordre final sera alors j_1, j_4, j_3, j_2.

Le diagramme de Gantt associé à cet exemple est illustré par la Figure 3-2.

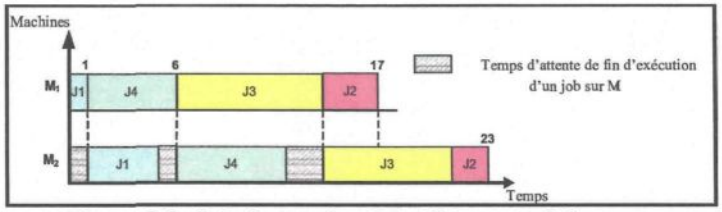


Figure 3-2: Solution optimale de l'Exemple 3-1

La complexité temporelle de cet algorithme est en $O(n \log n)$. Dans le cas où $m = 3$, on peut utiliser dans certains cas la règle de Johnson. En effet, pour résoudre ce problème, Johnson travaille avec « deux machines virtuelles ». Il obtient alors deux nouveaux temps d'exécution qui sont :

$$Q_{i1} = P_{i1} + P_{i2} \text{ et } Q_{i2} = P_{i2} + P_{i3}$$

Cette approximation est optimale si $\min(P_{i1}) \geq \max(P_{i2})$ ou $\min(P_{i3}) \geq \max(P_{i2})$. La deuxième machine n'est ainsi pas un goulot. Mais si cette condition n'est pas vraie, la méthode devient une heuristique.

3.2.4. Algorithme de Johnson modifié avec des temps de latence

Dans le cas d'un problème de permutation avec des temps de latence, l'algorithme de Johnson peut être utilisé pour résoudre ce problème. En effet, Mitten [1958] a obtenu le résultat suivant.

Théorème 6 [Mitten, 1958]

La permutation optimale est obtenue en appliquant l'ordre de Johnson aux temps d'exécution pour $p'_{ij} = p_{ij} + \tau_j$.

Exemple 3-2

L'exemple ci-dessous illustre l'application de l'algorithme de Johnson modifié sur un problème de *flow-shop*. Le nombre de job est égal à 4, leurs temps d'exécution respectifs sur les deux machines M_1 et M_2 et le temps de latence τ_j pour (j = 1, 2, 3 et 4).

j	j_1	j_2	j_3	j_4
M1	1	3	8	5
M2	4	2	7	6
τ_j	1	2	0	1

Tableau 3-3 : Temps d'exécution du problème n = 4 et m = 2 avec des temps de latences

Après avoir appliqué le Théorème 6 sur la séquence de jobs de notre exemple, on aura des nouveaux temps d'exécutions p'_{1j} et p'_{2j}, comme suit:

j	j_1	j_2	j_3	j_4
p'_{1j}	2	5	8	6
p'_{2j}	5	4	7	7

Tableau 3-4 : Les nouveaux temps d'exécution

La solution optimale est la permutation suivante : j_1, j_4, j_3 et j_2.

3.3 Revue de la littérature

Nous présentons dans ce qui suit une revue de littérature non exhaustive des heuristiques ainsi que des algorithmes de *branch and bound* utilisés pour la résolution du problème de *flow-shop*. Ce problème étant *NP-difficile*, ces deux approches sont justifiées pour sa résolution. Notons par ailleurs que les travaux mentionnés ci-dessous traitent des problèmes de *flow-shop* de permutation et sans temps de latence.

3.3.1. Approche heuristique pour le problème de flow-shop

Dans cette section, nous survolons la littérature qui a trait à l'approche heuristique pour la résolution du problème de *flow-shop* de permutation sans temps de latence. Notons qu'il y a peu de travaux effectués sur le problème de *flow-shop* avec ou sans temps de latence.

Pour construire sa solution, Palmer [1965] affecte des priorités à une séquence de jobs, puis il ordonnance les jobs suivant l'ordre décroissant de priorité. La complexité de son algorithme est $O(nm + n\log n)$. Campbell et *al.* [1970] ont développé une heuristique qui est une extension de l'algorithme de Johnson. Leur algorithme, appelé *CDS*, génère m-1 ordonnancements en regroupant les m machines en deux lots de machines virtuelles. Ils utilisent ensuite l'algorithme de Johnson pour déterminer les séquences et en garde la meilleure. Gupta [1972] a proposé, quant à lui, trois heuristiques : la première minimise le temps d'oisiveté des machines, la deuxième minimise le temps de fin d'exécution et la dernière est un algorithme qui alterne les deux. Les deux premières sont basées sur l'échange des paires de jobs et la troisième utilise les règles de Johnson. Dannenbring [1977] a proposé une heuristique rapide qui est un mélange de l'algorithme de Johnson et Palmer. Dans cette heuristique, on définit tout d'abord deux lots de machines virtuelles, comme définis dans l'heuristique *CDS*. Ensuite, on affecte les priorités aux différents jobs et on applique enfin l'algorithme de Johnson. Stinson et Smith [1982] ont utilisé des heuristiques du problème de voyageur de commerce. Nawaz et *al.* [1983] ont mis en œuvre une autre heuristique. Cette dernière est considérée comme étant la meilleure heuristique constructive pour la résolution du problème de *flow-shop*. Cette heuristique est basée sur une idée simple : le job qui a le plus grand temps d'exécution sur toutes les

machines sera exécuté le plus tôt possible. La complexité de cette heuristique appelée *NEH* est $O(n^3m)$. Cependant, Taillard [1990] a réduit la complexité de *NEH* en $O(n^2m)$.

Outre les heuristiques constructives, certains chercheurs ont utilisé les approches métaheuristiques. Ainsi, Widmer et Hertz [1989] ont proposé une méthode appelée *SPIRIT*. Cette dernière procède en deux phases : la première consiste à générer une analogie du problème de voyageur de commerce pour créer une solution initiale. Alors que dans la deuxième phase, ils créent une métaheuristique, la recherche avec tabous, pour générer le voisinage. Taillard [1990] a présenté une procédure similaire à celle de Widmer et Hertz. En plus de la recherche avec tabous, Taillard a utilisé une version améliorée de l'algorithme *NEH* pour créer plusieurs voisinages. Mais, il a remarqué que le meilleur voisinage est celui où on change la position d'un seul job. Werner [1993] a construit une méthode itérative rapide qui a donné des résultats remarquables. Cette méthode consiste à générer un nombre restreint de chemins qui donnent des solutions réalisables lors de la génération de voisinage. Il est intéressant de remarquer que cette méthode permet de générer un large voisinage. Toutefois, seules les solutions les plus intéressantes sont évaluées.

Ishibuchi et *al.* [1995] ont présenté deux algorithmes utilisant le recuit simulé. Ces deux algorithmes ont une performance robuste qui respecte la température de refroidissement d'une séquence. Notons que les algorithmes d'Ishibuchi et *al.* donnent des résultats assez proches de ceux du recuit simulé de Osman et Potts [1989].

La recherche avec tabous de Moccellin [1995] est basée sur l'algorithme *SPIRIT* de Widmer et Hertz [1989]. La seule différence réside dans le calcul de la solution initiale. Plus récemment, Moccellin et Dos Santos [2000] ont présenté un algorithme hybride qui utilise la recherche avec tabous et le recuit simulé. Ponnambalam et *al.* [2001] ont créé un algorithme génétique en utilisant des croisements GPX (*Generalised Position Crossover*) et incluant d'autres critères tels que : la mutation et le choix de la solution initiale.

Le Tableau 3-5 de Ruiz et *al* [2005] présente un résumé de quelques heuristiques utilisées pour résoudre le problème de permutation de *flow shop*.

3.3.2. Approche exacte pour le problème de flow-shop

Plusieurs chercheurs ont utilisé la méthode de *branch and bound* pour résoudre le problème de *flow-shop* de permutation. Nous présentons dans ce qui suit une revue de littérature non exhaustive reliée à l'utilisation de cette méthode.

Levner [1969] montre que rechercher le *makespan* associé à un problème de *flow-shop* est équivalent à la recherche du plus long chemin dans un graphe orienté. Basé sur ce graphe, l'auteur présente deux bornes inférieures. L'auteur n'a pas donné de résultats expérimentaux, mais il a mentionné avoir trouvé des solutions optimales pour des problèmes au-delà de 15 jobs. Suhani et Mah [1981] propose un algorithme de *branch and bound* qui utilise des bornes inférieures basées sur des estimations et variances des *makespan* calculés à partir de séquences partielles de jobs. Dans le but de réduire les temps de calculs de ces estimations, les auteurs proposent l'utilisation d'un paramètre empirique qui ajuste la valeur de la borne inférieure. Pour des problèmes à trois machines et un nombre de jobs allant de 6 à 10, cette heuristique trouve des solutions à moins de 1% de la solution optimale. Nagar et *al.* [1995] proposent, quant à eux, un algorithme de *branch and bound* qui minimise le *makespan* et le *flowtime* d'un *flow-shop*. Ils utilisent une recherche gloutonne pour déterminer une borne supérieure. Ils utilisent également la relaxation lagrangienne pour le calcul de la borne inférieure. Dans Wang et *al.* [1995], les auteurs traitent le problème de minimisation du *flowtime* pour un problème de *flow-shop* à deux machines. Ils utilisent une borne inférieure issue du travail de Ignall et Schrage [1965] où la programmation dynamique est utilisée pour le calcul du *makespan* minimal. Les deux bornes supérieures sont quant à elles issues des travaux de [Gupta, 1972]. Dans Hariri et Potts [1997], les auteurs traitent le problème de *flow-shop* pour minimiser le *makespan*. L'algorithme de *Banch and Bound* adopté utilise une borne inférieure basée sur la résolution d'un problème de *flow-shop* à deux machines artificielles. Les auteurs utilisent aussi quatre règles de dominance basées sur l'établissement de permutations partielles. Rios-Mercado et Bard [1999] minimisent le *makespan* pour le problème de permutations de *flow-shop* avec des temps de latence dépendants de la séquence d'entrée avec un algorithme de *branch and bound.* Les deux bornes inférieures qu'ils utilisent sont basées sur le calcul des temps de fin d'exécution de séquences partielles. Une règle de dominance a été aussi utilisée. Cette dernière décompose l'ensemble de jobs en deux sous

ensembles et calcule les temps de fin d'exécution des sous séquences partielles. Pour les bornes supérieures, ils ont utilisé le *Greedy Randomize Search Procedure* (*GRASP*) et des heuristiques hybrides [Rios-Mercado et Bard, 1999 b]. Gupta et *al.* [2001] utilisent un algorithme de *branch and bound* pour minimiser à la fois le *makespan* et le *flowtime*. Ils utilisent l'algorithme de Johnson et la dominance de la permutation pour établir une borne inférieure et deux règles de dominances. Pour les bornes supérieures, les auteurs utilisent des heuristiques constructives basées sur les séquences de Johnson. T' Kindt et *al.* [2001] ont développé un algorithme de *branch and bound* pour minimiser le *flowtime* (*temps total de séjour des jobs dans l'atelier*). Pour leurs bornes supérieures, ils ont utilisé une heuristique basée sur une recherche gloutonne et une deuxième basée sur l'insertion. Leurs bornes inférieures sont basées respectivement, sur une relaxation linéaire et une relaxation lagrangienne. La règle de dominance qu'ils ont utilisé est une adaptation de celle utilisée par Gupta et *al.* [2001].

Année	Nom des auteurs	Acronyme	Commentaire sur l'heuristique utilisée
1954	Johnson	Johns	Appliquée dans le cas de deux machines
1964	Dudek et Teuton	-	Utilise les règles de Johnson
1965	Palmer	Palme	Basée sur les index de priorités
1970	Campbell et al	CDS	Basée sur les règles de Johnson
1972	Gupta	-	Utilise trois heuristiques
1983	Nawaz et *al.*	NEH	Basée sur les priorités des jobs et l'insertion
1988	Hundal and Rajgopal	HunRa	Basée sur l'algorithme de Palmer
2000	Suliman	Sulim	Basée sur l'échange des pairs de jobs
2003	Framinan et *al*	-	Des études sur *NEH*

Tableau 3-5 : Les différentes heuristiques constructives pour le problème de *flow-shop de permutation*

Chapitre 4 : Approche exacte pour le problème de flow-shop à deux machines avec des temps de latence

4.1 Introduction

Nous nous intéressons dans ce chapitre au problème de *flow-shop* à deux machines avec des temps de latence. Dans un premier temps nous étudions le problème de *flow-shop* à deux machines avec des temps d'exécution unitaires et des temps de latence quelconques. Dans un deuxième temps, nous étudions le problème de *flow-shop* à deux machines avec des temps de latence et d'exécution quelconques.

4.2 Branch and bound avec des temps d'exécution unitaires

Rappelons que le problème de *flow-shop* à deux machines avec des temps d'exécution unitaires et des temps de latence quelconques est *NP-difficile* au sens fort. L'utilisation de la méthode énumérative de *branch and bound* est par conséquent justifiée. Comme mentionné au Chapitre 3, le développement de cette méthode est basé sur l'utilisation des bornes inférieures et supérieures et une stratégie de branchement. Notons que cette partie est basée sur la thèse de doctorat de Yu [1996] et sur l'article de Moukrim et Rebaïne [2005]. Nous débutons par la description des quatre bornes inférieures utilisées pour élaguer les branches non potentielles. Ensuite, nous présentons les bornes supérieures qui sont un ensemble d'heuristiques qui servent à limiter l'exploration de certaines branches. Après cela, nous décrivons les règles de dominances qui vont nous orienter vers une meilleure exploration des branches

4.2.1. Bornes inférieures

Les bornes inférieures sont toujours utilisées dans les algorithmes de *branch and bound*. En effet, à chaque nœud de l'arbre de recherche, on calcule une borne inférieure, correspondant au plus petit *makespan* qui peut être trouvé à partir de ce nœud. Si la valeur de cette borne inférieure est plus grande que la valeur du *makespan* courant, alors

on élague toute la branche issue de ce nœud. Dans ce qui suit, nous calculons quatre bornes inférieures ; la borne qui représente la plus grande valeur sera gardée.

- **Première borne inférieure (LB1)**

La première borne inférieure *LB1* est calculée à la racine de l'arbre de *recherche.* Cette borne génère la valeur du *makespan* minimale [Lenstra, 1994]. Notons que si le nombre de jobs n est inférieur ou égal à 5, alors la borne LB1 donne toujours le *makespan* optimal [Yu, 1996]. Mais, si n est supérieur ou égal à 6, alors il existe des instances pour lesquelles la borne inférieure n'est pas atteinte. Par exemple, la séquence des jobs 4-4-4-0 0-0 produit une borne inférieure égale à 9, tandis que l'optimum génère un *makespan* de valeur 10.

Lemme 1 [Yu, 1996]

Si Opt représente la valeur optimale, alors la relation suivante est vérifiée :

$$Opt \geq LB1 = \underset{1 \leq k \leq n}{Max} \left\{ \left\lceil \sum_{i=1}^{k} \frac{\tau_i}{k} \right\rceil + k + 1 \right\}, \qquad (1)$$

où $\tau_1 \geq \tau_2 \geq \ldots \geq \tau_n$.

Exemple 4-1

Soit le nombre de job suivant : $n = 6$. Les temps de latence des différents jobs sont :

Jobs *j*	1	2	3	4	5	6
Temps de latence τ_j	1	2	3	4	5	6

Tableau 4-1 : Temps d'exécution du problème pour $n = 6$ avec des temps de latences de l'Exemple 4-1

Le calcul de la première borne inférieure *LB*1 est comme suit :

$$LB1=\max\{6+1+1, \lceil (6+5)/2 \rceil + 2+1, \lceil (6+5+4)/3 \rceil + 3+1, \lceil (6+5+4+3)/4 \rceil + 4+1,$$
$$\lceil (6+5+4+3+2)/5 \rceil + 5+1, \lceil (6+5+4+3+2+1)/6 \rceil + 6+1\} = \max\{8, 9, 9, 10, 10, 11\} = 11.$$

- **Deuxième borne inférieure (LB2) [Yu, 1996]**

La deuxième borne inférieure *LB*2 peut être calculée à chaque nœud interne de l'arbre de recherche. En fait, elle est utile pour les solutions partielles qu'on construit au fur et à mesure qu'on avance dans cet arbre.

Corollaire 1 [Yu, 2004]

Soit $\delta = (\alpha, \beta)$ *une permutation de N jobs où* α *est une sous séquence fixée. Alors la relation suivante est vérifiée.*

$$Opt \geq Max\{LB1(N), |\alpha| + LB1(N-\alpha)\},$$

où LB1(K) est la borne inférieure LB1 appliquée aux jobs de l'ensemble K.

Preuve :

Il est clair que les jobs restants appartenant à la séquence N-α vont commencer leur exécution à partir d'un temps $t = \alpha$. De plus, la borne inférieure, correspondant aux jobs appartenant à la séquence N-α, est clairement *LB1* (N -α). Par conséquent, le *makespan* de l'ensemble de jobs, étant donné la séquence α, est clairement, $|\alpha| + LB1(N-\alpha)$ (Figure 4-1).

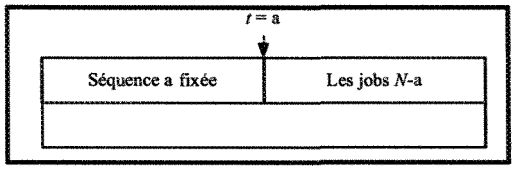


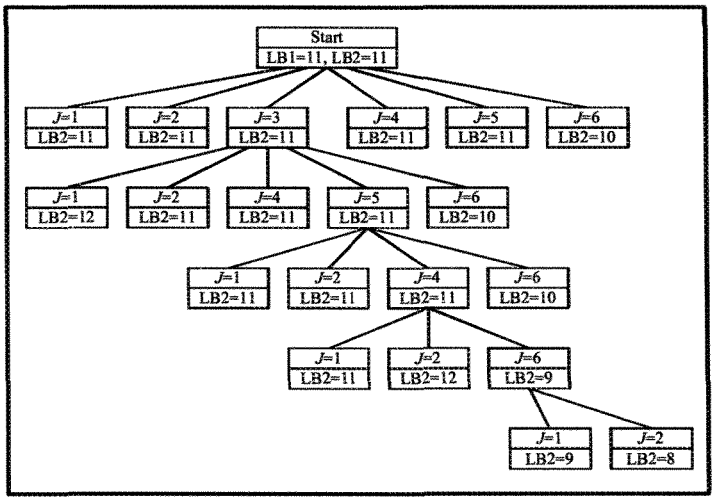
Figure 4-1: Le makespan de l'ensemble des jobs dans le cas unitaire

Exemple 4-2

Soit l'instance de l'Exemple 4-1 et $\alpha = (3,5,4)$. Les temps de latence des jobs restants qui n'appartiennent pas à α sont : 1, 2 et 6. On les ordonne suivant l'ordre décroissant. On place tout d'abord la séquence α, ensuite les N -α jobs restants. Le temps de latence des jobs seront placés dans cet ordre : 3, 5, 4, 6, 2 et 1. On obtient :

LB1 (6, 2, 1)= max $\{\lceil(6)/4\rceil+4+1, \lceil(6+2)/5\rceil+5+1, \lceil(6+2+1)/6\rceil+6+1\}$= max {7, 8, 9}=9 alors ***LB2***=$|\alpha|$ + *LB1* (6, 2, 1)=3+9=12.

En appliquant la borne inférieure *LB2* sur quelques branches de l'arbre de recherche, on obtient l'Arbre 4-1 :



Arbre 4-1 : LB2 sur quelques branches de l'arbre dans le cas unitaire

- **Troisième borne inférieure (LB3) [Moukrim et Rebaïne, 2005]**

Soit, *r(j)*, la date au plus tôt d'un job *j* sur la machine *M2*. Étant donné α jobs déjà ordonnancés, la date au plus tôt d'un job *j* est $\Pi(j) + \tau_j$ où $\Pi(j)$ est la position d'exécution du job $j \in \alpha$ sur *M1*. Pour les jobs restants qui appartiennent à la séquence $n-\alpha$, leur date au plus tôt ne peut pas être inférieure à $|\alpha| + 1 + \tau_j$. Il est clair que l'ordonnancement des jobs suivant leur date au plus tôt va générer une borne inférieure du *makespan*.

Pour tout *p* de 1 à $|\alpha|$, le job $\alpha(p)$ est ordonnancé sur *M1* à la case *p*. Pour tout job *j*, nous allons définir sa date de début au plus tôt sur *M2*.

Pour p = 1 to $|\alpha|$ faire
début
$j = \alpha(p)$
$r(j) = p + \tau_j$
Fin pour

Algorithme 4-1 : Première partie de l'algorithme de la troisième borne inférieure dans le cas unitaire

Ensuite, pour tout job j n'appartenant pas à la sous séquence α, sa date d'exécution sur *M1* est au plus tôt $|\alpha|$. Donc, sa date de début d'exécution sur *M2* est au plus tôt : $|\alpha| + 1 + \tau_j$. En exécutant les jobs selon ces dates de début d'exécution au plus tôt sur *M2*, on obtient une borne inférieure à la fin de traitement de tous les jobs sur *M2*. La date au plus tôt de l'ensemble des jobs est donnée par l'Algorithme 4-2 [Moukrim et Rebaïne, 2005]:

```
▪Trier les jobs dans un tableau s selon l'ordre croissant les r(j). On aura:
▪r(s(1)) <= r(s(2)) <= ... <=r(s(n)).
▪Poser t(s(1)) = r(s(1))
▪Répéter
        i = i + 1;
        t(s(i)) = t(s(i-1)) + 1;
        Si t(s(i)) < r(s(i )) alors t(s(i)) = r(s(i));
Jusqu'à ( i > n)
```

Algorithme 4-2 : L'algorithme de la troisième borne inférieure dans le cas unitaire

La valeur de la borne inférieure *LB3* sera alors égale à $t(s(n)) + 1$.

Exemple 4-3

L'exemple ci-dessous illustre le calcul de la borne inférieure *LB*3. Notons que nous on traite ici le même exemple donné pour illustrer *LB*1 et *LB*2. Soit $\alpha = (3,5,4)$. Les jobs restants qui n'appartiennent pas à α sont : 1, 2 et 6.

	$j \in \alpha$			$j \notin \alpha$		
Jobs *j*	3	5	4	1	2	6
Temps de latence τ_j	3	5	4	1	2	6
r(j)	4	7	7	5	6	10

Tableau 4-2 : Temps de latence pour $n = 6$ de l'Exemple 4-3

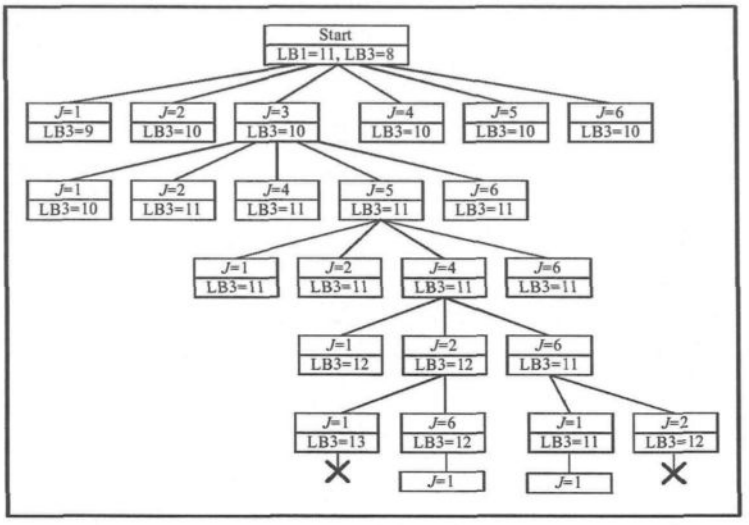
On trie les jobs dans un tableau s selon l'ordre croissant des $r(j)$. Ensuite, on applique l'algorithme pour obtenir une $LB3 = t(s(6)) + 1$:

Jobs *j*	3	1	2	4	5	6
Temps de latence τ_j	3	1	2	4	5	6
r(j)	4	5	6	7	7	10
s(i)	3	1	2	4	5	6
t(s(i))	4	5	6	7	8	10

Tableau 4-3 : Application de LB3 sur l'Exemple 4-3

On aura alors $LB3 = 10+1 = 11$.

En appliquant la borne inférieure *LB3* sur quelques branches de l'arbre de recherche, on obtient l'Arbre 4-2.



Arbre 4-2: Application de la borne *LB3* sur quelques branches de l'arbre de recherche dans le cas unitaire

- **Quatrième borne inférieure (LB4) [Moukrim et Rebaïne, 2005]**

La borne inférieure LB4 est similaire à la borne LB1, présentée dans le

Lemme 1. La seule différence réside dans le fait que *LB*4 tient compte des valeurs réelles des temps de latence. En effet, étant donnés α jobs déjà ordonnancés, leurs dates au plus tôt est $\Pi(j) + \tau_j$ pour tout job $j \in \alpha$, $\Pi(j)$ étant la position d'exécution du job j sur M1 dans cet ordonnancement. Si on place un job j à la position $\Pi(j) + \tau_j$ sur *M2*, on peut trouver un job $i \in \alpha$ et $i \neq j$ placé à la même position. Donc, on est obligé de modifier la position de ce dernier. Il s'ensuit que son temps de latence va augmenter. Cette modification consiste à augmenter le temps de latence correspondant au prochain emplacement libre sur M2. Les nouveaux temps de latence sont calculés à l'aide de l'algorithme [Moukrim et Rebaïne, 2005]:

Pour $p = 1$ to $|\alpha|$ faire
début
  j=$\alpha(p)$
  Soit q la période de temps où est exécuté le job j sur M2
    $\tau_j = q - p - 1$
  fin

Algorithme 4-3 : L'algorithme de la quatrième borne inférieure dans le cas unitaire

Notons que les temps de latence des jobs qui n'appartient pas à α ne sont pas modifiés. Pour calculer la borne *LB*4, on applique la même formule utilisée auparavant pour le calcul de la borne *LB*1. Cette formule sera appliquée sur l'ensemble des jobs après modification des temps de latence.

Exemple 4-4

Soit l'exemple ci-dessous :

Jobs *j*	1	2	3	4	5	6
Temps de latence τ_j	1	2	3	4	5	6

Soit $\alpha = (3,5,4)$, donc $|\alpha| = 3$. Les temps de latence des jobs restants qui n'appartiennent pas à α sont : 1, 2 et 6. En appliquant la borne *LB*4 sur notre exemple, on obtient l'ordonnancement de la Figure 4-2:

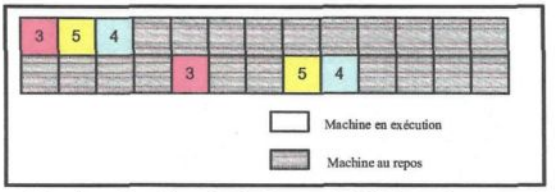


Figure 4-2: Application de la borne *LB*4 dans le cas unitaire

On traite seulement les temps de latence des jobs appartenant à α :

- Pour $p = 1$: $j = 3$, $\tau_3 = 3$. Donc, le temps modifié de $\tau_3 = 5-1-1 = 3$ (dans ce cas, on n'a pas de changement)
- Pour $p = 2$: $j = 5$, $\tau_5 = 5$. Donc, le temps modifié de $\tau_5 = 8-2-1 = 5$. (Dans ce cas, on n'a pas de changement)
- Pour $p = 3$: $j = 4$, $\tau_4 = 4$. Donc, le temps modifié de $\tau_4 = 9-3-1 = 5$. (Dans ce cas, on a fait un changement)

Après avoir modifié le temps de latence, notre tableau sera le suivant :

Jobs *j*	3	5	4	6	2	1
Temps de latence τ_j	3	5	5	6	2	1

Tableau 4-4 : Les nouveaux temps de latence de l'Exemple 4-4

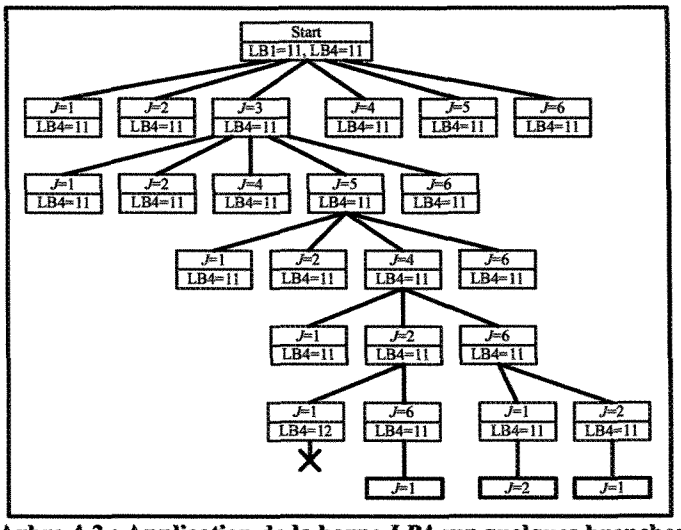
La borne *LB*4 = *LB*1 (avec temps de latence modifiés)

$$LB1=\max\{6+1+1,\lceil(6+5)/2\rceil+2+1,\lceil(6+5+5)/3\rceil+3+1,\lceil(6+5+5+3)/4\rceil+4+1,$$

$$\lceil(6+5+5+3+2)/5\rceil+5+1,\lceil(6+5+5+3+2+1)/6\rceil+6+1\}$$

LB1 = max {8, 9, 10, 10, 11, 11} = 11.

En appliquant la quatrième borne inférieure *LB*4 sur notre problème de *flow-shop,* on obtient une partie de notre arborescence. Ces branches sont décrites dans l'Arbre 4-3.



Arbre 4-3 : Application de la borne *LB4* sur quelques branches de l'arbre de recherche dans le cas unitaire

La borne inférieure retenue pour cet exemple est la borne qui nous donne la plus grande valeur entre ces quatre bornes inférieures calculées.

4.2.2. Les bornes supérieures [Moukrim et Rebaïne, 2005]

Une borne supérieure permet de rejeter certaines branches explorées, par simple propagation de contraintes. Ces bornes supérieures sont un ensemble d'heuristiques appliquées sur les nœuds de l'arborescence.

Dans cette section, on présente quatre familles d'heuristiques. Chacune d'elles ordonnance les jobs d'une séquence donnée suivant trois ordres différents. Les trois règles utilisées par chacune de ces heuristiques sont :

- Première règle :

Elle consiste à ordonnancer la séquence de jobs sur la première machine *M1* suivant l'ordre décroissant du temps de latence, et au plus tôt sur la deuxième machine *M2*.

Exemple 4-5

Soit une séquence de jobs de taille $n = 6$. S présente les temps de latence de cette séquence. On a alors $S = \{6, 5, 4, 2, 1, 3\}$. Après avoir appliqué la première règle, on aura cet ordre: 6, 5, 4, 3, 2 et1.

- Deuxième règle :

Cette règle consiste à placer les jobs d'une manière alternative sur la première machine *M1*. Elle est donnée par la stratégie qui permet d'extraire en premier une sous séquence de jobs ayant des temps de latence successifs de différence au moins de deux. Ensuite, on réapplique la même règle sur l'ensemble des jobs restants, et ainsi de suite jusqu'à épuisement de la séquence initiale. Nous obtenons ainsi un nouvel ordre qui sera exécuté au plus tôt sur la deuxième machine *M2*.

Exemple 4-6

Soient les temps de latence suivants : 6, 5, 4, 2, 1 et 3. Pour appliquer la deuxième règle sur la séquence *S*, on doit suivre les étapes suivantes :

1. Extraction d'une sous séquence de jobs ayant des temps de latence successifs de différence au moins de 2. Ces temps de latence seront stockés dans une nouvelle séquence *F*. On aura alors : $F = \{6, 4, 2\}$. Les temps de latence restant dans la séquence *S* sont : 5, 1 et 3
2. On répète le même processus, décrit par le point (1), sur les jobs restant de la séquence *S* jusqu'à son épuisement. On aura : 5 et 3. On fait une mise à jour de la séquence *F* qui sera $F = \{6, 4, 2, 5, 3\}$. Il reste seulement un job qui sera ajouté à la fin de la séquence *F*.
3. La séquence finale des temps de latence sera la suivante : $F = \{6, 4, 2, 5, 3, 1\}$.

- Troisième règle :

Elle présente le même principe que la règle précédente. Seulement dans ce cas, l'alternance ne s'applique pas sur toute la séquence. Elle consiste à prendre en premier une sous séquence qui contient seulement deux jobs ayant des temps de latence successifs de différence au moins de deux. Et on refait le même principe jusqu'au traitement de toute la séquence initiale.

Exemple 4-7

Soit les temps de latence suivants : 6, 5, 4, 3, 2 et 1. Dans ce cas, l'alternance se fait deux à deux, on aura alors : 6, 4, 5, 3, 2 et 1.

Il est clair que la complexité de ces différentes heuristiques décrites ci-dessus est $O(n\log n)$ puisqu'elle est dominée par la procédure de tri. Dans ce qui suit, on présente les différentes heuristiques utilisées pour les bornes supérieures. On distingue quatre bornes supérieures qui sont : *UB1, UB2, UB3, UB4*. Chacune de ces bornes utilise les trois règles citées ci-dessus. Au total, nous présentons 12 heuristiques. Notons que ces heuristiques sont conçues pour être utilisées pour les nœuds internes de l'arbre de recherche, c'est-à-dire qu'il est supposé qu'une sous-séquence jobs, α, est déjà fixée.

- **Première borne supérieure (UB1)**

La première heuristique utilise un ordre topologique (placer les jobs sur les n premières périodes de temps tels qu'ils se présentent). On considère une sous séquence de jobs α. Pour calculer la première borne supérieure, il faut tout d'abord compléter la sous séquence de jobs par le reste des jobs qui n'appartiennent pas à α. Ces derniers doivent être placés à la suite de la sous-séquence α. Donc, ces jobs vont être affectés au plus tôt sur la période de temps $|\alpha|$ de la première machine *M1*, suivant un certain ordre. L'ordre choisi est l'une des règles d'ordonnancement citées plus haut. La séquence finale obtenue sera exécutée sur *M1* et puis, sur *M2*. Par exemple, si on a déjà fixé l'ordre d'une sous séquence α = (1, 2 et 3) alors il reste les jobs 4, 5 et 6 ayant les temps de latence : 5, 2 et 7. Ensuite, on ordonnance ces derniers suivant l'ordre décroissant. Donc, la borne *UB1* est donnée par la valeur du *makespan* pour la séquences 1, 2, 3, 6, 4 et 5. On applique alors l'Algorithme 4-4.

p : position d'un job dans α

j : un job appartenant à la séquence α

n : nombre total des jobs

Pour $p = 1$ jusqu'à n faire

début

$$j=\alpha(p)$$

Placer j sur la période de temps p sur la machine M1

Placer j sur la première période de temps libre sur M2 à partir de la période de temps $p+1+\tau_j$

Fin pour

Algorithme 4-4 : l'algorithme de la première borne supérieure dans le cas unitaire

Mais, dans le cas où $\alpha = (1,2)$ et les temps de latence des jobs restants sont : 6, 5, 4 et 3. On applique la deuxième règle sur les jobs restants. On aura l'ordre suivant : 6, 4, 5, et 3. L'ordonnancement généré est illustré par la Figure 4-3. La première borne supérieure ***UB*1** est égale à **12.**

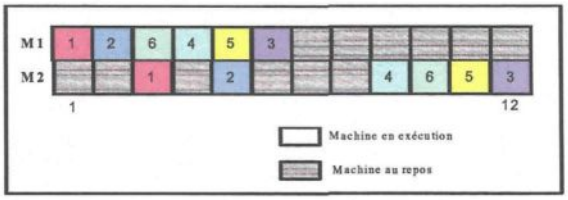


Figure 4-3: Application d'*UB*1 sur une séquence de jobs en utilisant la deuxième règle d'ordonnancement dans le cas unitaire

Mais dans le cas où on applique *UB*1 sur la séquence de jobs ordonnancée suivant la troisième règle, on obtient les mêmes résultats. Si on prend $\alpha = (6)$ alors les temps de latence des jobs restants seront : 5, 4, 3, 2 et 1. Le nouvel ordre sera : 5, 3, 4, 2 et 1. L'ordonnancement sur les deux machines est représenté par la Figure 4-4. Donc, La première borne supérieure ***UB*1** est égale à **12.**

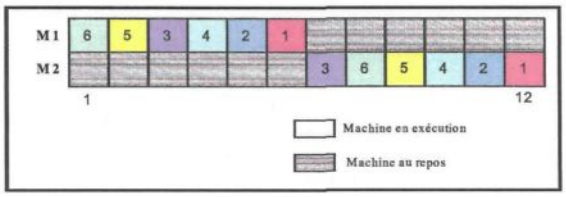


Figure 4-4: Application d'*UB*1 sur une séquence de jobs en utilisant la troisième règle d'ordonnancement dans le cas unitaire

- **Deuxième borne supérieure (UB2)**

La deuxième heuristique consiste tout d'abord, à placer les jobs appartenant à α Ensuite, on place les n-α jobs au plus tôt sur la deuxième machine M2 et au plus tard sur la première machine *M1*, par rapport à leur position sur *M2*. Donc, un job $j \in n-\alpha$ sera affecté au plus tôt à la position $|\alpha|+\tau_j+1$. On obtient l'Algorithme 4-5. $M(1,k)$ et $M(2,l)$ désignent respectivement l'emplacement d'un job i à la $k$ème position sur la première machine et la $l$ème position sur la deuxième machine [Moukrim et Rebaïne, 2005].

N : nombre total des jobs.
α : est une sous séquence de job ordonnancée.
On ordonnance les jobs n-α suivant l'une des règles d'ordonnancement
Pour i = 1 jusqu'à n faire
Début
$K = 0$
Répéter
$K = k+1$
Jusqu'à $M(1,k)$ = libre et $M(2,k+1+\tau_i)$=libre
Placer le job j à la position k sur M1 et à la position $k+1+\tau_i$ sur M2
Mettre Jusqu'à $M(1,k)$ = occupée et $M(2,k+1+\tau_i)$ = occupée
Fin pour

Algorithme 4-5 : L'algorithme de la deuxième borne supérieure dans le cas unitaire

Exemple 4-8

Pour calculer la borne *UB*2, on traite le même exemple pour illustrer le calcul d'*UB*1. Soit $\alpha = (1,2)$. Les temps de latence des jobs restants sont : 6, 5, 3 et 4. On applique les trois règles d'ordonnancement sur cet exemple:

Jobs j	1	2	3	4	5	6
Temps de latence τ_j	1	2	3	4	5	6

Tableau 4-5 : Temps de latence n = 6 de l'Exemple 4-8

- La séquence du job obtenue en appliquant la première règle est la suivante : 1, 2, 6, 5, 4 et 3 générant un *makespan* de valeur 12 comme illustré à la Figure 4-5.

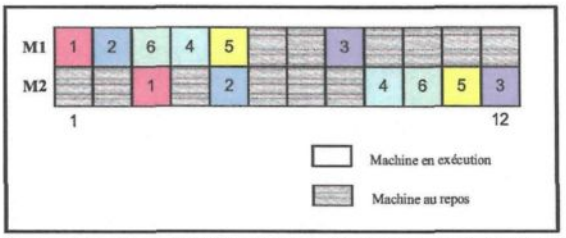


Figure 4-5: Application d'*UB2* sur une séquence de jobs en utilisant la première règle d'ordonnancement dans le cas unitaire

– En appliquant la deuxième règle d'ordonnancement, on obtient: 1, 2, 6, 3, 5 et 4. En exécutant cette séquence sur les deux machines, on obtient un *makespan* de valeur 12, (Figure 4-6). Par conséquent, *UB2 = 12*.

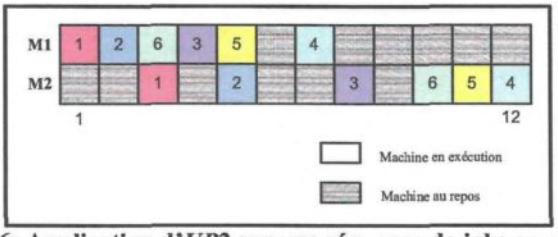


Figure 4-6: Application d'*UB2* sur une séquence de jobs en utilisant la deuxième règle d'ordonnancement dans le cas unitaire

– La troisième règle produira le même résultat que la règle précédente, puisqu'on obtient la même séquence de jobs 1, 2, 6, 3, 5 et 4.

- **Troisième borne supérieure (UB3)**

La stratégie de cette troisième borne *UB3* consiste à placer un job $j \in n-\alpha$, le plus tôt possible dans un emplacement libre sur *M1,* disons au temps *p*. Ensuite, on déduit sa place sur *M2*, car on peut se trouver dans le cas où deux jobs se confondent sur le même emplacement. À ce moment, on avance jusqu'à la prochaine position libre sur *M2*. En procédant de la sorte, le temps de latence de ce job va changer. Après avoir placé un job *j* au temps *q* sur *M2*, on doit replacer le job *j* sur *M1* (le plus tard possible) de telle manière vérifie que (*q-p*) soit le plus proche du temps de latence τ_j. L'Algorithme 4-6 illustre borne supérieure *UB3* [Moukrim et Rebaïne, 2005].

Ordonnancer les jobs n'appartenant pas à α suivant l'une des règles d'ordonnancement

Pour $i = 1$ à n faire

/* Trouver la première place libre sur M1 où le job $i \in n - \alpha$ pourrait être exécuté */

$k = 0$

Répéter

$k = k+1$

Jusqu'à $M(1, k)$ = libre

/* Déterminer la première case libre sur M2 où le job i pourrait être exécuté si on suppose que i est exécuté sur la case k sur M1 */

$q = k + \tau_i$

Répéter

$q = q+1$

Jusqu'à $M(2, q)$ = libre

Placer le job i à la période de temps q sur M2

/* Déterminer la case où le job i sera exécuté le plus tard possible sur M1 sachant qu'il est exécuté à la case q sur M2. */

$R = q - \tau_i$

Répéter

$r = r - 1$

Jusqu'à $M(1, r)$ = libre

Placer le job i sur la période de temps r sur M1

Fin pour

Algorithme 4-6 : l'algorithme de la troisième borne supérieure dans le cas unitaire

Exemple 4-9

Considérons l'instance de l'exemple précédent. On obtient l'ordonnancement illustré par la Figure 4-7. Notons que les trois règles produisent le même résultat. Donc, la valeur $UB3 = 13$.

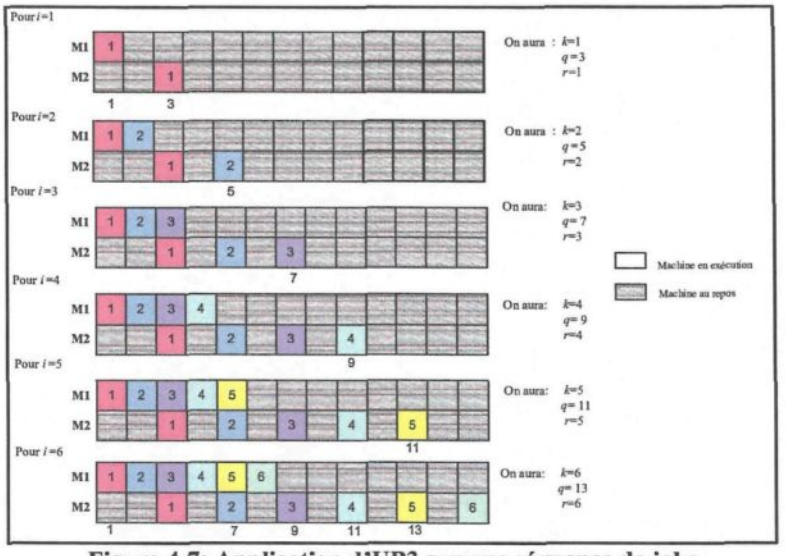


Figure 4-7: Application d'UB3 sur une séquence de jobs

- **Quatrième borne supérieure (UB4)**

La quatrième borne supérieure *UB*4 présente une heuristique qui traite séparément les jobs à temps de latence pairs et impairs. Tout d'abord, on commence par ordonner les jobs n'appartenant pas à α suivant l'une des règles d'ordonnancement. On place les jobs appartenant à α sur les deux machines. Après, on vérifie si le premier job entre les n-α est pair ou impair. Dans le cas où ce dernier est impair, on affecte le plus tôt possible sur *M1* et *M2* les jobs impairs suivis des jobs pairs. Mais dans le cas contraire, on affecte tout d'abord les jobs pairs. On suppose maintenant que le premier job de la séquence est impair. Soit un job impair $j \in n-\alpha$, placé au plus tôt à la position $|\alpha|$ sur *M1* et $|\alpha|+\tau_j$ sur M2. Ensuite, on place les jobs pairs le plus tard possible sur *M1* par rapport à leur position sur *M2*.

Soit un job *i* pair appartenant à $n-\alpha-(jobs\ impairs)$. Ce job est placé au plus tôt à la position $|\alpha|+|jobs\ impairs|+\tau_i$ sur *M2*. Si on se trouve dans un cas où deux jobs se confondent sur le même emplacement alors on passe à l'emplacement suivant. La valeur de la borne *UB4* est le *makespan* total obtenu après avoir placé les jobs sur *M1* [Moukrim et Rebaïne, 2005].

Exemple 4-10

Soit $\alpha = (3,5)$. Les temps de latence des jobs restants qui n'appartiennent pas à α sont : 6, 1, 2 et 4.

- On ordonnance les temps de latence des jobs suivants selon la première règle, notre séquence sera la suivante : 3, 5, 6, 4, 2 et 1. la valeur de la borne *UB*4 est égale à **12** (Figure 4-8).

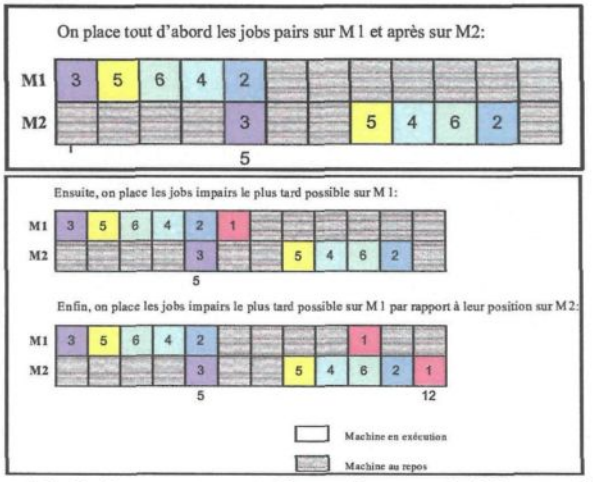


Figure 4-8: Ordonnancement d'une séquence de jobs en utilisant la première règle et la borne *UB*4

- En appliquant la deuxième et troisième règle sur les temps de latence de la séquence de jobs, on aura: 3, 5, 6, 1, 2 et 4 et la borne *UB*4 = 11. L'ordonnancement obtenu est illustré par la Figure 4-9.

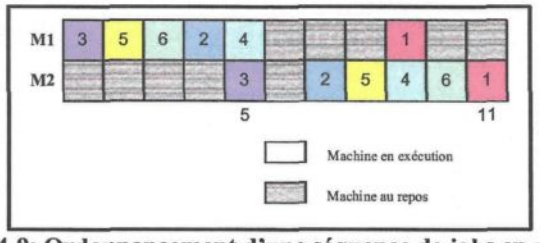


Figure 4-9: Ordonnancement d'une séquence de jobs en utilisant la deuxième règle et la borne *UB*4

4.2.3. Règles de dominance

Les règles de dominance présentent un ensemble de conditions et de critères. Elles permettent d'exclure des solutions partielles. Ces règles sont souvent utilisées pour raffiner la méthode de *branch and bound* et palier aux faiblesses de certaines bornes.

- **Première règle de dominance**

Une règle de dominance a pour but de réduire le temps de traitement et accélérer le calcul de l'algorithme de *branch and bound.*

Lemme 2 [Mokrim et Rebaïne, 2005]

Notons que M2(j) la période de temps où est exécuté le job j sur M2 sachant que j est exécuté sur M1 à la période de temps $|\alpha| + 1$. *La règle de dominance est la suivante : S'il existe un job j' dans I -* α *tel que* $\tau_{j'} > \tau_j$ *et* $M2(j) - (|\alpha| + 1) - 1 \geq \tau_{j'}$ *alors il est inutile de considérer le nœud où le job j est placé à l'emplacement* $|\alpha| + 1$ *sur M1.*

Exemple 4-11

Soit les jobs suivant $I = \{1, 2, 3, 4, 5\}$. On a $\alpha = (1, 2)$. Les jobs restants sont : 3, 4 et 5.

Jobs *j*	1	2	3	4	5
Temps de latence τ_j	2	2	2	1	0

Tableau 4-6 : Temps de latence *n* = 5 de l'Exemple 4-11

Si on place $j = 5$ alors $M2(j)=6$. On doit vérifier les deux conditions, on aura :

- $\tau'_j > \tau_j$ car $\tau'_j = 2$.
- $M2(j)-(|\alpha|+1)-1 = 6-(2+1)-1 = 2 \geq \tau'_j$

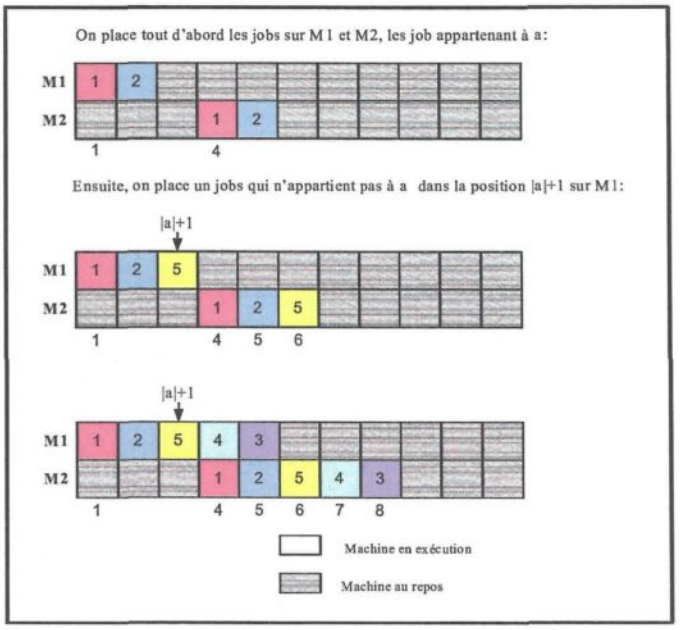


Figure 4-10 : Application de la première règle de dominance

Les deux conditions sont vraies. Donc, on n'a pas besoin d'explorer le reste de l'arbre à partir de ce nœud. On obtient alors l'ordonnancement suivant en échangeant les places des jobs 3 et 5. L'ordonnancement présenté à la Figure 4-11 a le même *makespan* que celui présenté à la Figure 4-10.

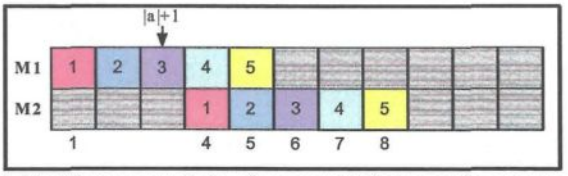
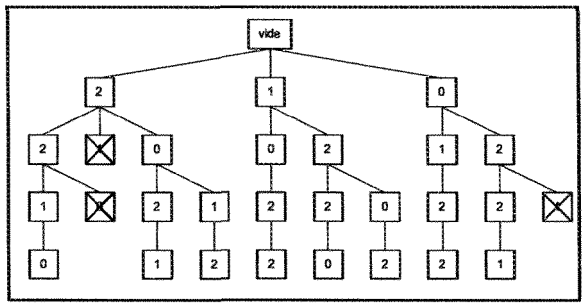


Figure 4-11: Placement de la séquence de jobs après avoir appliqué la première règle de dominance

Considérons l'exemple où les jobs ont les temps de latence suivants : 2, 2, 1, 0. On obtient l'Arbre 4-4. Les nœuds non explorés grâce à la première règle de dominance seront barrés.



Arbre 4-4 : Illustration de la première règle de dominance

- **Deuxième règle de dominance [Moukrim et Rebaïne, 2005]**

Dans cette règle, on regroupe les jobs et on les représente sous forme de blocs. Un bloc peut contenir un ou plusieurs jobs. Notons que l'ordre dans le même bloc n'est pas forcément le même sur M1 et M2. Par exemple, un bloc peut s'exécuter sur M1 dans cet ordre (3, 3, 0) alors que sur M2, il s'exécute dans l'ordre (0, 3, 3).

Par ailleurs, il a été remarqué par Moukrim et Rebaïne [2005] que les différents blocs peuvent être utilisés dans n'importe quel ordre sans que cela n'affecte la valeur du *makespan*.

Pour utiliser cette observation, on définit un ordre lexicographique entre ces différents ordonnancements trouvés et on ne générera que le plus grand d'entre eux. En appliquant cette règle de dominance, on obtient un arbre de *branch and bound* réduit.

Définition 10

Soient deux ensembles $A=(a_1,...,a_n)$ *et* $B=(b_1,...,b_m)$ *tels que* $a_1 \geq ... \geq a_n$ *et* $b_1 \geq ... \geq b_m$. *On dit que A est lexicographiquement plus grand que B si :*

1. *il existe j,* $1 \leq j < n$, $a_1 = b_1,...,a_j = b_j$ *et* $a_{j+1} > b_{j+1}$, *ou*

2. $n > m$ *et* $a_1 = b_1,...,a_m = b_m$.

Exemple 4-12

Soit les temps de latence suivants :

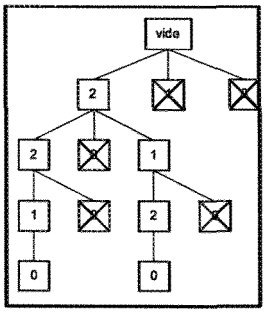
Jobs *j*	1	2	3	4	5	6	7	8	9
Temps de latence τ_j	3	3	0	5	4	1	1	4	2

Tableau 4-7 : Temps de latence *n* = 9 de l'Exemple 4-12

On crée trois blocs : (3, 3, 0), (5, 4, 1, 1) et (4, 2). En exécutant les blocs dans n'importe quel ordre on obtient le même *makespan*. Ainsi tous les ordonnancements présentés dans Figure 4-12 sont équivalents.

On va profiter de cette remarque pour ne générer qu'un seul représentant parmi ces 6 solutions. L'ordre lexicographique décroissant permet de classer ces différents blocs ainsi : (5, 4, 1, 1) > (4, 2) > (3, 3, 0). Ainsi, dans l'algorithme de *branch and bound*, on ne génère que l'ordonnancement « O4 » parmi ces 6 ordonnancements.

Considérons l'exemple suivant où les jobs ont les temps de latence suivants : 2, 2, 1, 0. On obtient l'Arbre 4-5. Les nœuds non explorés grâce à la deuxième règle de dominance seront barrés.



Arbre 4-5 : Illustration de la deuxième règle de dominance

Au début de l'ordonnancement, 0 constitue un bloc. De plus, (2,2,1)>(0). Donc, on élague la branche 0. On applique le même principe sur les autres branches jusqu'à l'obtention de l'Arbre 4-5.

4.2.4. Résultats

Dans cette section, sont résumés les résultats préliminaires de la simulation effectuée sur l'algorithme de *branch and bound* que nous venons de décrire. L'algorithme de *branch and bound* ainsi que toutes les heuristiques ont été développés avec le langage C++ sous l'environnement Visual Studio .NET 2003. L'application a été déployée sur une machine Pentium IV à 3.06 GHZ d'horloge et avec 512 Mo de mémoire vive. Nous avons limité les expérimentations à 900 secondes. Le **'X'** désigne que le temps limite de 900 secondes est dépassé. Il est utile de noter que ce problème est résolu par cet algorithme d'une manière efficace et ce, pour des instances n'excédant pas 30

jobs. Au-delà, il existe beaucoup d'instances pour lesquelles l'algorithme ne donne pas solution dans la limite fixée de 900 secondes. Ces instances ont été élaborées par Moukrim et Rebaïne [2005].

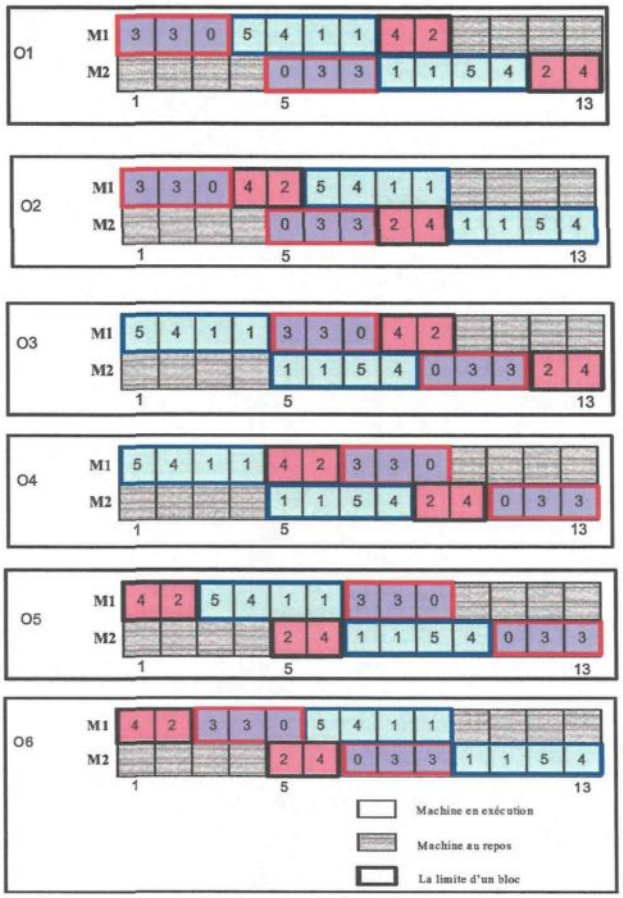


Figure 4-12: Les différents ordonnancements générés suite à la permutation des blocs

La première colonne représente la borne inférieure LB calculée à la racine de notre *Branch and Bound*. La deuxième colonne indique la meilleure borne supérieure UB des 4 bornes générées par notre algorithme. La troisième colonne représente le *Makespan* optimal. La quatrième colonne représente le nombre de nœuds visités lors de l'exploration de l'arbre de *Branch and Bound*. La dernière colonne représente le temps d'exécution de chaque instance.

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
17	17	17	0	0
18	19	18	23	0
16	17	16	21	0
17	17	17	0	0
15	16	15	23	0
17	17	17	0	0
16	16	16	0	0
15	16	15	0	0
15	16	15	21	0
17	18	17	0	0

Tableau 4-8 : Simulations pour n = 10, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
31	32	31	527	0,01
31	32	31	1815	0,02
32	35	32	133916	19,72
32	34	32	149	0
32	34	32	0	0
32	32	32	0	0
29	36	29	1368	0,02
31	31	31	0	0
31	32	31	354	0,01
32	33	32	57	19,85

Tableau 4-9 : Simulations pour n = 20, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
49	50	49	2597	0,02
47	48	47	5717	0,11
44	46	44	2295441	43,6
45	47	45	233383	4,4
48	49	48	346	0,01
45	48	45	118455	2,27
46	47	46	12079	0,23
45	47	45	176145	3,28
46	48	46	150676	2,82
44	45	44	57323	1,08

Tableau 4-10 : Simulations pour n = 30, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
63	64	63	3334873	82.98
62	64	62	6666241	165.12
62	63	62	1886276	45.15
61	64	61	29454523	712.43
64	65	64	34299	0.85
62	63	63	18845798	X
58	60	60	36507975	X
61	63	63	39355086	X
61	63	63	36565474	X
58	61	61	35060949	X

Tableau 4-11 : Simulations pour $n = 40$, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
69	70	70	32817293	X
71	73	71	2642367	69,3
70	72	72	34840502	X
68	70	70	327551841	X
68	70	70	327551841	X
66	68	68	34502465	X
70	71	71	36424816	X
70	73	73	34083049	X
67	71	71	31712970	X
65	67	67	57	X

Tableau 4-12 : Simulations pour $n = 45$, cas unitaire

Nous pouvons remarquer que le nombre de nœuds croit exponentiellement à partir de 30 jobs. Les temps de traitement deviennent similaires puisque le temps d'exécution de l'instance ne doit pas céder 900 secondes. On a trouvé jusqu'à ici des résultats optimaux.

4.3 Branch and bound avec des temps d'exécution quelconques

Nous étudions dans cette partie le problème de *flow-shop* à deux machines avec des temps de latence et d'exécution quelconques. Comme dans la partie concernant les temps d'exécution unitaires, nous décrivons les différentes bornes inférieures et supérieures utilisées pour la réalisation de l'algorithme de *branch and bound*. Notons que cette partie est partiellement basée sur le travail de Yu (1996).

4.3.1. Bornes inférieures

Rappelons que le temps d'exécution d'un job j (j = *1, 2, ..., n*) sur la machine i (i = *1, 2*) est désigné par p_{ij} et son temps de latence par τ_j.

- **Première borne inférieure (LB1)**

La première borne inférieure *LB1*, que nous présentons, est donnée par le Lemme 3.

Lemme 3 [Yu, 1996]

Si Opt désigne la valeur optimale du makespan, alors

$$Opt \geq LB1 = \max\left(\sum_{j=1}^{n} p_{1j} + \min_{1\leq j\leq n}\left(\tau_j + p_{2j}\right), \min_{1\leq j\leq n}\left(p_{1j} + \tau_j\right) + \sum_{j=1}^{n} p_{2j}\right).$$

Preuve :

Soit j le dernier job exécuté sur *M1*. Il est clair que son temps de fin est $\sum_{j=1}^{n} p_{1j}$ comme illustré à la Figure 4-13. Ce job doit premièrement subir un temps de latence τ_j et ensuite être exécuté par *M2*. Dans le meilleur des cas, ces deux temps sont inférieurs à $\min_{1\leq j\leq n}\left(p_{1j} + \tau_j\right)$. D'où le résultat.

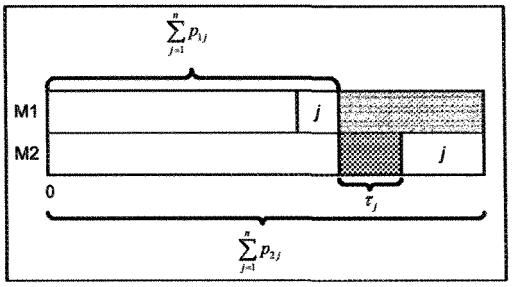


Figure 4-13 : Illustration du Lemme 3

Exemple 4-13

Soit l'instance ci-dessous avec n = 5 jobs.

Jobs *j*	**1**	**2**	**3**	**4**	**5**
Temps d'exécution p_{1j}	**13**	**12**	**2**	**1**	**1**
Temps d'exécution p_{2j}	**8**	**8**	**7**	**6**	**1**
Temps de latence τ_j	**9**	**7**	**2**	**1**	**0**
$p_{1j}+\tau_j$	**22**	**19**	**4**	**2**	**1**
$p_{2j}+\tau_j$	**17**	**15**	**9**	**7**	**1**

Tableau 4-13 : Application de LB1 sur l'Exemple 4-13 dans le cas quelconque

Les cases grises représentent respectivement le minimum de $p_{1j}+\tau_j$ et celui de $p_{2j}+\tau_j$. Le calcul de la première borne inférieure *LB1* est le suivant :

$LB1 = \max\{(13+12+2+1+1)+1, (8+8+7+6+1)+1\}=31.$

Étant donnée une sous séquence α des jobs déjà ordonnancés, nous présentons, dans ce qui suit, trois bornes inférieures *LB2*, *LB3*, *LB4*.

- **Deuxième borne inférieure (LB2)**

La deuxième borne inférieure LB2 est calculée à chaque nœud de l'arbre de recherche. Elle est donnée par le Lemme 4.

Lemme 4 [Yu, 1996]

$$Opt \geq LB2 = \max\left(P_{1j}+\tau_j+P_{2j}\right).$$

Preuve :

Quelque soit le job, il aura à s'exécuter sur *M1*, ensuite un temps de latence s'en suit avant de s'exécuter sur *M2*. Il est clair que le *makespan* ne peut être inférieur à la somme de ces trois valeurs.

Étant donné α jobs déjà ordonnancés, la date au plus tôt d'un job A sur la machine *M2* est $\sum_{j=1}^{A} p_{1j}+\tau_A$ pour tout job $A \in \alpha$. Il est clair que les jobs restants appartenant à la

séquence n - α vont commencer leur exécution après que les jobs de la séquence α soient exécutés. De plus, la borne inférieure correspondant aux jobs appartenant à la séquence n -α, est clairement *LB2* (n-α), c'est-à-dire LB2 appliquée aux jobs de l'ensemble n-α. Par conséquent, le *makespan* de l'ensemble de jobs, étant donnée la séquence de jobs α, est $\sum_{j\in\alpha} p_{1j} + LB2(n-\alpha)$. Cette borne peut être obtenue en appliquant l'Algorithme 4-7.

$|\alpha|$: **représente le cardinal de la sous séquence** α.

n - α : **étant le nombre de jobs non ordonnancés et qui n'appartiennent pas à** α.

On calcule la borne LB2 sur le reste de jobs n'appartenant pas à α **on aura alors :**

$$LB2 = \max_{j\in n-\alpha}\left(p_{1j} + \tau_j + p_{2j}\right).$$

La valeur de la borne inférieure finale est la suivante : LB2 = $\sum_{j\in\alpha} p_{1j} + LB2\left(n-\alpha\right)$

Algorithme 4-7 : Algorithme de la deuxième borne inférieure LB2 dans le cas quelconque

Exemple 4-14

Soit l'instance suivante avec n = 5 jobs.

	$j\in\alpha$		$j\in n-\alpha$		
Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$p_{1j}+\tau_j$ p_{2j}			11	8	2

Tableau 4-14 : Application de LB2 sur l'Exemple 4-14 dans le cas quelconque

Pour calculer *LB2*, on doit procède comme suit : On détermine : $\sum_{j=1}^{2} p_{1j}$ = 13+12 = 25 avec $j\in\alpha$. Ensuite, on applique la formule de *LB2* sur les n - α jobs restants. La case grise représente le maximum de ($p_{1j}+\tau_j+p_{2j}$). La valeur finale de *LB2 est* comme suit :

$$\text{LB2} = \sum_{j=1}^{2} p_{1j} + \text{LB2}\ (n - \alpha) = 25 + \max\ \{11, 8, 2\} = 36.$$

- **Troisième borne inférieure (LB3)**

Notons que les deux bornes inférieures *LB1* et *LB2* contiennent seulement un seul terme représentant le temps de latence. Il est clair qu'il est préférable d'impliquer l'ensemble de temps de latence pour espérer avoir une borne inférieure efficace.

Théorème 7 [Yu, 1996]

Soient $q_j = \min(p_{1j}, p_{2j})$, $r_j = \max(p_{1j}, p_{2j})$ *avec (j = 1, 2, ..., n). Si Opt désigne le makespan de la solution optimale, alors*

$$Opt \geq LB3 = \left\lceil \left(\sum_{j=1}^{n} q_j (\tau_j + r_j - 1) \right) \Big/ \sum_{j=1}^{n} q_j \right\rceil + 1 + \sum_{j=1}^{n} q_j .$$

Preuve :

Sans perte de généralités, nous pouvons considérer que chaque job j contient q_j jobs artificiels avec un temps d'exécution unitaire chacun. Considérons les deux cas suivants :

Cas 1 : $p_{1j} \leq p_{2j}$

Dans ce cas, nous avons : $q_j = p_{1j}$ et $r_j = p_{2j}$. Soit k = 1, 2, ..., q_j. Définissons l'exécution de chaque job artificiel J_{jk} sur *M1*, ensuite sur *M2*. Le temps d'exécution d'un job artificiel J_{jk} sera considéré comme un temps unitaire : il sera exécuté sur le $k^{ème}$ emplacement du job j sur *M1*. Puis, il poursuit son exécution sur *M2*, et il exécutera sur le $(r_j - q_j + k)^{ème}$ emplacement du job j sur *M2*. Ces jobs artificiels ont le même temps de latence τ_j'.

$$\tau_j' = (q_j - k) + \tau_j + (r_j - q_j + k - 1) = \tau_j + r_j - 1$$

Cas 2 : $p_{1j} > p_{2j}$

Dans ce cas, nous avons : $q_j = p_{2j}$ et $r_j = p_{1j}$. Soit k = 1, 2, ..., n. Similairement, définissons l'exécution de chaque job artificiel J_{jk} sur *M1*, ensuite sur *M2*. Le temps d'exécution d'un job artificiel J_{jk} sur *M1* sera le même que celui décrit par le premier

cas, alors que J_{jk} sera exécuté sur le $k^{ème}$ emplacement du job j sur $M2$. De même que le cas précédent, les jobs artificiels ont le même temps de latence τ_j' comme ci-dessus, c'est-à-dire

$$\tau_j' = (r_j - k) + \tau_j + (k-1) = \tau_j + r_j - 1.$$

En appliquant le Lemme 1 sur toute la séquence de $(q_1 + q_2 + ... + q_n)$ jobs artificiels de temps unitaires et des temps de latence, décrits précédemment, on obtient la troisième borne inférieure *LB3*.

Cette borne peut être obtenue en appliquant l'Algorithme 4-8 sur notre arborescence.

$|\alpha|$: représente le cardinal de la sous séquence α.

$n - \alpha$: représente le nombre de jobs qui n'appartiennent pas à α.

$$q_j = \min_{j \in n-\alpha}(p_{1j}, p_{2j})$$

$$r_j = \max_{j \in n-\alpha}(p_{1j}, p_{2j})$$

On calcule la borne *LB3* sur le reste de jobs n'appartenant pas à α on aura alors :

$$LB3 = \left\lceil \left(\sum_{j \in n-\alpha} q_j (\tau_j + r_j - 1) \right) \Big/ \sum_{j \in n-\alpha} q_j \right\rceil + 1 + \sum_{j \in n-\alpha} q_j.$$

La valeur de la borne inférieure finale est la suivante : LB3 = $\sum_{j \in \alpha} p_{1j} + LB3(n-\alpha)$

Algorithme 4-8 : Algorithme de la troisième borne inférieure LB3 dans le cas quelconque

Exemple 4-15

L'exemple ci-dessous illustre le calcul de la borne inférieure *LB3*. On suppose qu'on a une sous séquence α de taille 2 comme dans l'exemple précédent.

	$j \in \alpha$		$j \in n-\alpha$		
Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$r_j = \max_{j \in n-\alpha}(p_{1j}, p_{2j})$			7	6	1
$q_j = \min_{j \in n-\alpha}(p_{1j}, p_{2j})$			2	1	1
$\tau_j + r_j - 1$			8	6	0
$q_j(\tau_j + r_j - 1)$			16	6	0

Tableau 4-15 : Application de LB3 sur l'Exemple 4-15 dans le cas quelconque

On peut procéder comme suit pour le calcul de *LB3* :

1. On calcule *LB3* pour les jobs n'appartenant pas à α. On aura :

$$LB3(n-\alpha) = \lceil (16+6+0)/4 \rceil + 1 + 4 = \lceil (22)/4 \rceil + 5 = 11.$$

2. Enfin, la valeur finale de la borne *LB3* est : $LB3 = \sum_{j \in \alpha} p_{1j} + LB3(n-\alpha) =$ 25+11= 36.

- **Quatrième borne inférieure (LB4)**

Théorème 8 (Yu, 1996)

Soit p'_{ij} *la somme de j plus petites valeurs de* $\{p_{i1}, p_{i2}, ..., p_{in}\}$ *avec i = 1, 2 et j = 1, 2, ..., n, alors on aura :*

$$Opt \geq LB4 = \left\lceil \left(\sum_{j=1}^{n} \tau_j + \sum_{j=1}^{n} p'_{1j} + \sum_{j=1}^{n} p'_{2j} \right) \Big/ n \right\rceil.$$

Où p'_{ij} *désigne la somme des j plus petites valeurs de* $\{p_{i1}, p_{i2}, ..., p_{in}\}$.

Preuve :

Soit *S* un ordonnancement d'une séquence du problème de *F2D*. On suppose que σ et ε représentent, respectivement, les séquences de jobs sur *M1* et *M2*, d'une solution donnée. Il est évident que le *makespan* *C*[*S*] vérifie la condition suivante :

$$C[S] \geq \sum_{j=1}^{\sigma^{-1}(i)} p_{1j} + \tau_j + \sum_{j=\varepsilon^{-1}(i)}^{n} p_{2j} \; ; i = 1, 2, \ldots, n. \qquad (1)$$

On a $\sigma^{-1}(i)$ et $\varepsilon^{-1}(i)$ désignent respectivement la position d'un job i sur la séquence σ et ε. Soit σ et $\varepsilon \in PER_n$[7], sachant que $\alpha(i) = \sigma^{-1}(i)$ et $\beta(i) = n+1-\varepsilon^{-1}(i)$; $i \in [1, n]$. On remarque alors que la signification de p'_{ij} est la suivante :

$$\sum_{j=1}^{\sigma^{-1}(i)} p_{1j} \geq p'_{1\alpha(i)} \text{ et } \sum_{j=\varepsilon^{-1}(i)}^{n} p_{2j} \geq p'_{2\beta(i)} \text{ avec } (i = 1, 2, \ldots, n). \qquad (2)$$

Donc, si on somme (1) et (2), on obtient :

$$n * C[S] \geq \sum_{i=1}^{n} \tau_j + \sum_{i=1}^{n} p'_{1\alpha(i)} + \sum_{i=1}^{n} p'_{2\beta(i)}. \qquad (3)$$

On aura alors :

$$\sum_{i=1}^{n} p'_{1\alpha(i)} = \sum_{i=1}^{n} p_{1i}, \sum_{i=1}^{n} p'_{2\beta(i)} = \sum_{i=1}^{n} p'_{2i}. \qquad (4)$$

La borne inférieure *LB4* découle ainsi des inégalités (3) et (4).

Pour appliquer la quatrième borne inférieure *LB4*, on doit d'abord vérifier les temps de latence de chaque job appartenant à α car ils peuvent augmenter en plaçant, par exemple, deux jobs sur le même emplacement. Pour cela, on applique l'Algorithme 4-9.

q : fin de temps d'exécution d'un job sur *M2*.
Pour p = 1 to $|\alpha|$ faire
début
Soit q l'emplacement de la fin d'exécution du job j sur *M2*

$$\tau_j = q - \sum_{j=1}^{p} p_{1j} - p_{2j} \}$$

fin pour

Algorithme 4-9 : Algorithme de vérification des temps de latence

Après avoir obtenu les nouveaux temps de latence, on applique le Théorème 8 sur l'ensemble des jobs.

[7] PER_n est la permutation d'un ensemble $N = \{1, 2, \ldots, n\}$ de jobs.

Exemple 4-16

Soit l'instance suivante avec $n = 5$ jobs. On suppose qu'on a une sous séquence $\alpha = 3$ de jobs.

	$j \in \alpha$			$j \in n-\alpha$	
Jobs j	4	5	3	1	2
Temps d'exécution p_{1j}	1	1	2	13	12
Temps d'exécution p_{2j}	6	1	7	8	8
Temps de latence τ_j	1	0	2	9	7

Tableau 4-16 : Les temps d'exécution et de latence de l'Exemple 4-16

Pour calculer la borne *LB4*, on doit tout d'abord vérifier s'il a un changement dans les temps de latence des jobs appartenant à α . Donc, on doit tout d'abord ordonnancer les jobs appartenant à α comme illustré dans la Figure 4-14 :

Figure 4-14 : Application de *LB4* avec des temps d'exécution quelconques

De la Figure 4-14, on peut remarquer qu'il y'a eu des changements dans les temps de latence des jobs appartenant à α . On a donc :

- Pour $p = 1$: $j = 4$, $\tau_4 = 1$. Donc, le temps modifié de $\tau_4 = 8-1-6 = 1$ (dans ce cas, on n'a pas de changement).
- Pour $p = 2$: $j = 5$, $\tau_5 = 0$. Donc, le temps modifié de $\tau_5 = 9-2-1 = 6$. Notons que, dans ce cas, on a fait un changement.
- Pour $p = 3$: $j = 3$, $\tau_3 = 2$. Donc, le temps modifié de $\tau_3 = 16-4-7 = 5$. Notons que, dans ce cas, on a fait également un changement.

Après avoir modifié les temps de latence, notre tableau sera comme suit :

	$j \in \alpha$			$j \in n-\alpha$	
Jobs j	4	5	3	1	2
Temps d'exécution p_{1j}	1	1	2	13	12
Temps d'exécution p_{2j}	6	1	7	8	8
Temps de latence τ_j	1	6	5	9	7
P'_{1j}	1	2	4	17	29
P'_{2j}	6	7	14	22	30

Tableau 4-17 : Application de LB4 sur l'Exemple 4-16 dans le cas quelconque

On a: $LB4 = \left\lceil \left(\sum_{j=1}^{n} \tau_j + \sum_{j=1}^{n} p'_{1j} + \sum_{j=1}^{n} p'_{2j} \right) \Big/ n \right\rceil = \lceil ((1+6+5+9+7) + (1+2+4+17+29)$

$+ (6+7+14+22+30)) / 5 \rceil = 32.$

4.3.2. Bornes supérieures

Comme dans le cas unitaire, nous utilisons dans notre algorithme de *branch and bound* quatre bornes supérieures basées sur des heuristiques ci-dessous. Notons que ces heuristiques sont conçues pour être utilisées pour les nœuds internes de l'arbre de recherche, c'est-à-dire qu'il est supposé qu'une sous-séquence de jobs α est déjà fixée.

- **Première borne supérieure**

Pour calculer la première borne supérieure *UB1*, nous utilisons l'algorithme de Johnson, présenté à la section 3.2.4. Puisqu'il est à l'origine de plusieurs travaux sur deux machines, nous avons jugé utile de l'utiliser pour calculer *UB1*.

Pour calculer la première borne supérieure, on place tout d'abord les jobs appartenant à α sur machine *M1*. Ensuite, on passe à traiter les $n-\alpha$ jobs restants. Nous devrons appliquer le Théorème 6 sur les $n-\alpha$ jobs. Ce théorème consiste à modifier les temps d'exécution de chaque job j, en appliquant : $p'_{ij} = p_{ij} + \tau_j$ ($i = 1, 2, j = 1, 2, \ldots, n$). Après avoir obtenu les nouveaux temps d'exécution, on applique l'algorithme de Johnson sur les $n-\alpha$ jobs. Et enfin, on place les $n-\alpha$ jobs à la suite des α jobs déjà placés sur *M1*. L'heuristique *UB1* est obtenue en appliquant l'Algorithme 4-10.

- $|\alpha|$: **représente le cardinal de la sous séquence α .**
- $n - \alpha$ **: représente le nombre de jobs non ordonnancés et qui n'appartiennent pas à α**
- **Pour $i = 1$ jusqu'à 2 faire**

  Pour $j \in n - \alpha$ n faire

  $$p'_{ij} = p_{ij} + \tau_j$$

  Fin pour

  Fin pour

- **Appliquer l'algorithme de Johnson sur les nouveaux temps d'exécution p'_{ij} ($i = 1, 2$ et $j \in n - \alpha$)**
- **Après la fin d'exécution de chaque job j sur M1, on essaie de le placer le plus tôt possible sur M2**

Algorithme 4-10 : Algorithme de la première borne supérieure UB1 pour le cas quelconque

Exemple 4-17

L'exemple ci-dessous illustre le calcul de la borne supérieure $UB1$: soit le nombre de jobs suivant $n = 5$.

Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0

Tableau 4-18 : Les temps d'exécution et de latence de l'Exemple 4-17

- Soit la sous séquence $\alpha = (1,2)$. Les jobs restants sont : 3, 4 et 5. On doit en premier lieu calculer les nouveaux temps d'exécution en appliquant: $p'_{ij} = p_{ij} + \tau_j$ ($i = 1, 2, j = 1, 2, \ldots, n$). On obtient alors le Tableau 4-19.

On applique l'algorithme de Johnson sur les temps d'exécution modifiés. On aura alors deux sous ensembles U et V qui contiendront les jobs suivant : $U = \{4, 3\}$ et $V = \{5\}$.

	$j \in \alpha$		$j \in n-\alpha$		
Jobs *j*	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$p'_{1j} = \tau_j + p_{1j}$			4	2	1
$p'_{2j} = \tau_j + p_{2j}$			9	7	1

Tableau 4-19 : L'application de UB1 sur l'Exemple 4-17 dans le cas quelconque

La séquence finale des jobs sera ordonnancée comme suit : Après avoir placé les α jobs, on met les n -α jobs restants. On obtient la permutation: 1, 2, 4, 3 et 5. L'ordonnancement obtenu est présenté par la Figure 4-15. La valeur du *makespan* est donc $UB1 = 53$.

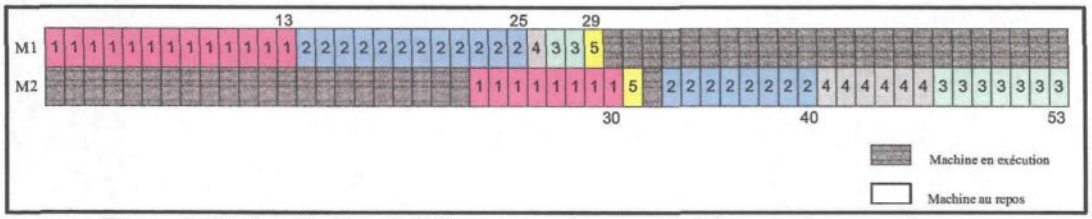


Figure 4-15 : Application de UB1 sur une séquence de jobs pour le cas quelconque

- **Deuxième borne supérieure**

La deuxième borne supérieure *UB2* est obtenue comme suit : on commence par placer les α jobs *M1*. Puis, on applique le Théorème 6 sur les n-α jobs restants. Autrement dit, les nouveaux temps d'exécution sont: $p'_{ij} = p_{ij} + \tau_j (i = 1, 2, j = 1, 2, ..., n)$. Ensuite, on ordonnance les jobs suivant ces nouveaux temps d'exécution obtenus dans l'ordre décroissant et on place la séquence obtenue sur la première machine *M1*, ensuite on place les jobs le plus tôt possible sur *M2*. Quoique très simple, cette heuristique s'est avérée efficace. Il s'agit d'appliquer l'algorithme *LPT*[8] (*Longest Processing time*) sur la première machine en incluant les temps de latence ensuite de placer au plus tôt les jobs

[8] LPT est un algorithme d'ordonnancement qui priorise les jobs avec les plus grands temps d'exécution.

sur la deuxième machine. On calcule enfin le *makespan* pour obtenir alors la deuxième borne supérieure *UB2*. Le calcul de la deuxième borne est illustré à l'Algorithme 4-11.

- $|\alpha|$: représente le cardinal de la sous séquence α.
- $n - \alpha$: étant le nombre de jobs non ordonnancés et qui n'appartiennent pas à α
- $p'_{ij} = p_{ij} + \tau_j$ avec $i = 1,2$ et $j = 1,2,...n$.
- Ordonnancer les jobs suivants l'ordre décroissant de p'_{1j} et on les place sur M1
- Après la fin d'exécution de chaque job j sur M1, on essaie de le placer le plus tôt possible sur M2

Algorithme 4-11 : Algorithme UB2 pour le cas quelconque

Exemple 4-18

Soit l'instance suivante avec $n = 5$ jobs. On doit tout d'abord appliquer la formule suivante: $p'_{ij} = p_{ij} + \tau_j$ sur l'instance. On aura alors des nouveaux temps d'exécution présentés dans le tableau ci-dessous:

	$j \in \alpha$		$j \in n-\alpha$		
Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$p'_{1j} = \tau_j + p_{1j}$			4	2	1
$p'_{2j} = \tau_j + p_{2j}$			9	7	1

Tableau 4-20 : Application de UB2 sur l'Exemple 4-18 dans le cas quelconque

Ensuite, on ordonnance les jobs suivant l'ordre décroissant de p'_{1j}, on obtient alors la séquence de jobs : 1, 2, 3, 4 et 5, illustrée par la Figure 4-16 . La valeur de la borne *UB2* est donnée par le *makespan* qui est égal à 53.

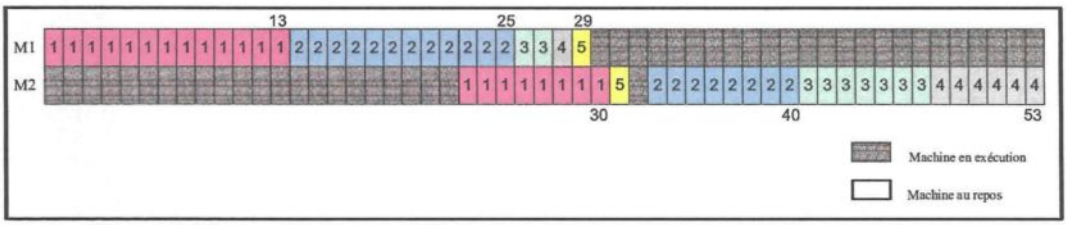


Figure 4-16: Application de UB2 sur une séquence de jobs pour le cas quelconque

- **Troisième borne supérieure**

L'heuristique pour obtenir la troisième borne supérieure *UB3* consiste à affecter des priorités aux différents jobs. On applique cette priorité au $n - \alpha$ jobs. Le job qui a la priorité la plus élevée sera exécuté en premier et ainsi de suite. Pour affecter une priorité S_j à un job J_j on calcule cette formule : $S_j = \left(\sum_{i=1}^{2} (m - 2i + 1) p_{ij} \right) + \tau_j$ avec $j = 1,2,\ldots, n$ et $i = 1,2$. Ensuite, on ordonnance les S_i suivant l'ordre décroissant. Puis on place les $n - \alpha$ jobs obtenus sur la machine *M1* à la suite des α jobs déjà ordonnancés et au plus tôt possibles sur *M2*. On calcule enfin le *makespan*. Cette formule pour le calcul des priorités est une modification de celle utilisée par Palmer [1965]. En effet, ce dernier avait utilisé la formule suivante pour calculer les priorités des jobs pour un *flow-shop* à condition que le nombre de machine m ≥ 1 : $S_j = \sum_{i=1}^{m} (m - 2i + 1) p_{ij}$ avec $j = 1,2,\ldots, n$ et $i = 1, 2$. En développant cette formule on obtient :

$$S_j = -(m-1)p_{1j} - (m-3)p_{2j} - (m-5)p_{3j} - \ldots + (m-3)p_{m-1j} + (m-1)p_{mj}, \quad \text{ou} \quad (m\text{-}1)$$

par exemple peut s'écrire sous cette forme : m-$2i$+1 pour $i = 1$.

Nous avons introduit les temps de latence τ_j pour calculer les priorités associés aux jobs. Ainsi, un job pour lequel les temps d'exécution tendent à augmenter d'une machine à une autre aura une priorité plus élevé que celle d'un job qui aura des temps d'exécution tendant à diminuer d'une machine à une autre tout en prenant compte des temps de latence. Le calcul de la troisième borne est illustré à l'Algorithme 4-12.

- **$n - \alpha$: représente le nombre de jobs non ordonnancés et qui n'appartiennent pas à α**
- **Pour $i = 1$ jusqu'à 2 faire**

  Pour $j \in n - \alpha$ n faire

  $$S_j = \left(\sum_{i=1}^{m} (m - 2i + 1) p_{ij} \right) + \tau_j$$

  Fin pour

  Fin pour
- **Ordonnancer les S_j suivant l'ordre décroissant**

Algorithme 4-12 : Algorithme de UB3 pour le cas quelconque

Exemple 4-19

Pour mieux comprendre cette borne, considérons l'exemple suivant. Soit $\alpha = (3,4)$. Les jobs restants sont : 1, 2 et 5. Le tableau ci-dessous présente la priorité de chaque job n'appartenant pas à α.

	$j \in \alpha$		$j \in n-\alpha$		
Jobs *j*	3	4	1	2	5
Temps d'exécution p_{1j}	2	1	13	12	1
Temps d'exécution p_{2j}	7	6	8	8	1
Temps de latence τ_j	2	1	9	7	0
$S_j = \left(\sum_{i=1}^{m}(m-2i+1)p_{ij}\right)+\tau_j$			14	11	0

Tableau 4-21 : L'application de UB3 sur l'Exemple 4-19 dans le cas quelconque

Le nouvel ordre de la séquence des jobs sera le suivant : 3, 4, 1, 2 et 5. La Figure 4-17 représente l'ordonnancement de la séquence sur les deux machines. La borne *UB3* est égale à 43.

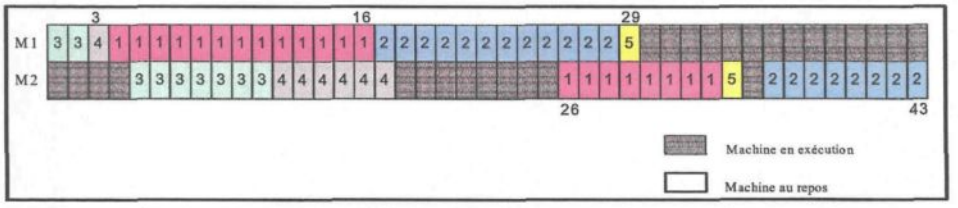


Figure 4-17: Application de UB3 sur une séquence de job pour le cas quelconque

- **Quatrième borne supérieure**

Pour le calcul de la quatrième borne supérieure *UB4*, nous appliquons l'heuristique *NEH* [Nawaz et *al.*, 1983]. Cette heuristique est basée sur l'hypothèse qu'un lot de jobs ayant un temps total d'exécution élevé est prioritaire (le lot est positionné en priorité dans un ordonnancement partiel) par rapport à un job de faible priorité. On classe les jobs dans l'ordre décroissant de leur durée de traitement totale : temps d'exécution sur les deux machines et celui de latence. Puis, on construit progressivement la séquence correspondant à la solution en insérant, à chaque étape, le job suivant dans la liste à la meilleure place dans la séquence, de manière à minimiser le *makespan*. L'adaptation de

l'algorithme *NEH* pour le calcul de la borne *UB4* est illustrée à l'Algorithme 4-13. L'utilisation de cette heuristique est justifiée par la bonne qualité des résultats donnés au sein de différents travaux, notamment ceux de Taillard [1990], Sarin et Lefoka [1993], Framinan et *al.* [2003].

- **$|\alpha|$: représente le cardinal de la sous séquence α.**
- **$n-\alpha$: représente le nombre de jobs non ordonnancés et qui n'appartiennent pas à α**

  Pour $j \in n-\alpha$ n faire

  $p_{ij} = p_{1j} + \tau_j + p_{2j}$ **avec $i = 1,2$.**

  Fin pour
- **Adaptation de NEH :**
  - **Ordonner les tâches selon l'ordre croissant de temps d'exécution des jobs p_{ij} sur les machines.**
  - **Créer une séquence vide *S*.**
  - **On ordonnance les deux premiers jobs à la suite de la séquence α.**
  - **Soit une séquence de jobs vide *T***
  - **Tant que (*T* non vide) faire**

    ***T* = *T* – premier élément *j* de *T* ;**

    Tester l'élément *j* à tous les emplacements dans *S*;

    Insérer *j* dans *S* à l'emplacement qui minimise le *makespan*.
  - **Fin Tant que**

Algorithme 4-13 : Algorithme d'adaptation NEH pour le calcul de UB4 pour le cas quelconque

Exemple 4-20

Cet exemple illustre le calcul de la borne supérieure *UB*4. Considérons l'instance suivante pour $n = 5$.

Jobs *j*	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0

Tableau 4-22 : Les temps d'exécution et de latence de l'Exemple 4-20

Soit une sous séquence $\alpha = (3,4)$. Les jobs restants n'appartenant pas à α sont : 1, 2 et 5. On doit tout d'abord calculer le temps d'exécution de chaque job sur les deux machines: $p_{ij} = p_{1j} + \tau_j + p_{2j}$ $(i = 1, 2, j = 1, 2, \ldots, n)$. On aura alors le tableau suivant :

	$j \in \alpha$		$j \in n - \alpha$		
Jobs *j*	3	4	1	2	5
Temps d'exécution p_{1j}	2	1	13	12	1
Temps d'exécution p_{2j}	7	6	8	8	1
Temps de latence τ_j	2	1	9	7	0
$p_{ij} = p_{1j} + \tau_j + p_{2j}$			30	27	2

Tableau 4-23 : Application de UB4 sur l'Exemple 4-20 dans le cas quelconque

On ordonnance les jobs n'appartenant pas à α suivant l'ordre croissant du temps p_{ij}. La séquence sera la suivante : 5, 2 et 1. Après avoir ordonnancé les jobs, on doit placer les jobs appartenant à α ainsi que les deux premiers jobs des $n - \alpha$ jobs restants. C'est-à-dire, on place les jobs 3 et 4, ensuite les jobs 5 et 2, tout en respectant l'ordre. La Figure 4-18 illustre cet ordonnancement.

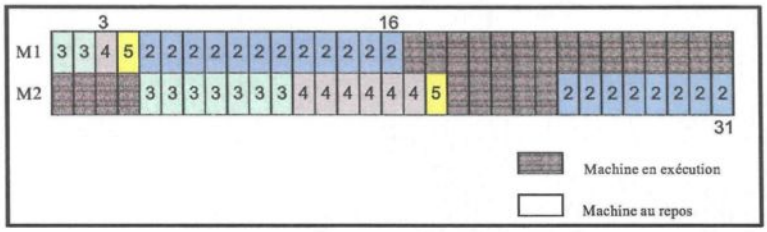


Figure 4-18 : Ordonnancement d'une partie de la séquence en appliquant UB4

On applique notre algorithme sur le reste des jobs non encore ordonnancées. Le job restant est le job 1. Ce dernier va être placé dans trois différentes positions, comme illustré dans la Figure 4-19. A partir de là, on doit choisir l'ordonnancement qui génère le plus petit *makespan*.

On a trois différents ordonnancements, (a), (b) et (c). L'ordonnancement qui a le *makespan* minimal est l'ordonnancement (b) ou (c), puisqu'ils ont le même *makespan*. Pour conclure, nous allons choisir par défaut le premier ordonnancement, (b). La valeur de la borne *UB4* est égale à 44.

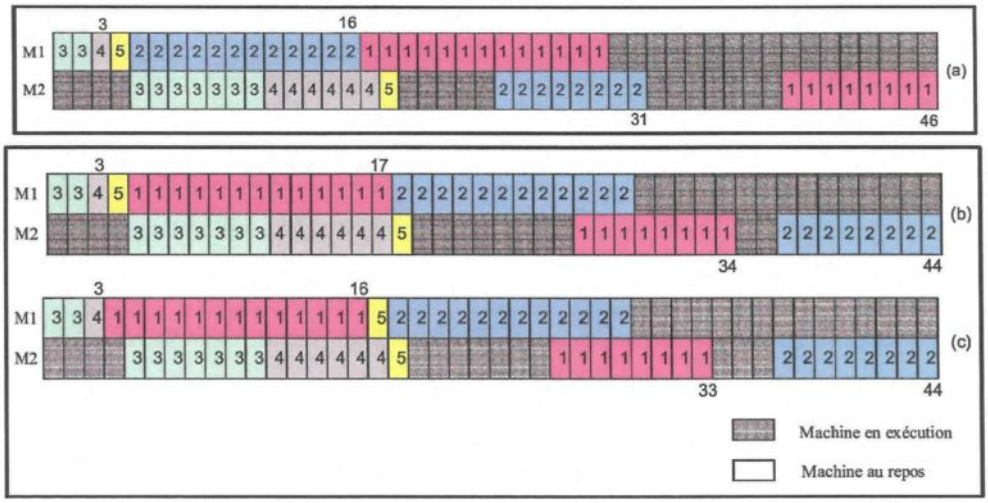


Figure 4-19 : Les différents ordonnancements induits par l'application de UB4

4.3.3. Illustration de l'algorithme de branch and bound

Appliquons l'algorithme de *branch and bound* sur l'exemple ci-dessous pour $n = 5$, où les temps de latence et d'exécution sur *M1* et *M2* de jobs sont résumés comme suit :

Jobs *j*	**1**	**2**	**3**	**4**	**5**
Temps d'exécution p_{1j}	**13**	**12**	**2**	**1**	**1**
Temps d'exécution p_{2j}	**8**	**8**	**7**	**6**	**1**
Temps de latence τ_j	**9**	**7**	**2**	**1**	**0**

Tableau 4-24 : Les temps d'exécution et de latence d'un problème de branch and bound

* On calcule d'abord la valeur de $LB1 = 31$.
* Les différentes bornes supérieures initiales sont :
  - $UB1 = 46$; $UB2 = 46$; $UB3 = 49$; $UB4 = 44$;
* On fait appel au *branch and bound*. On obtient alors une séquence minimale : 3, 1, 4, 2 et 5 et un *makespan* = 43, (Figure 4-20).

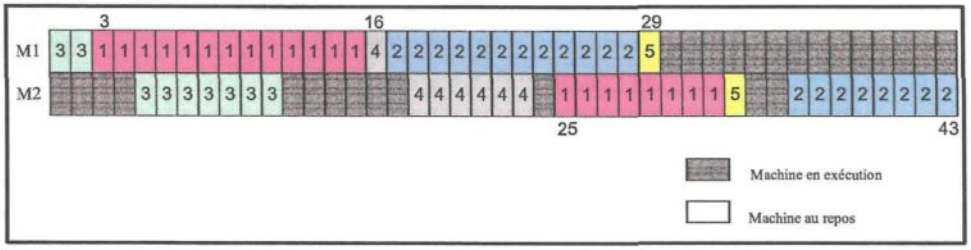
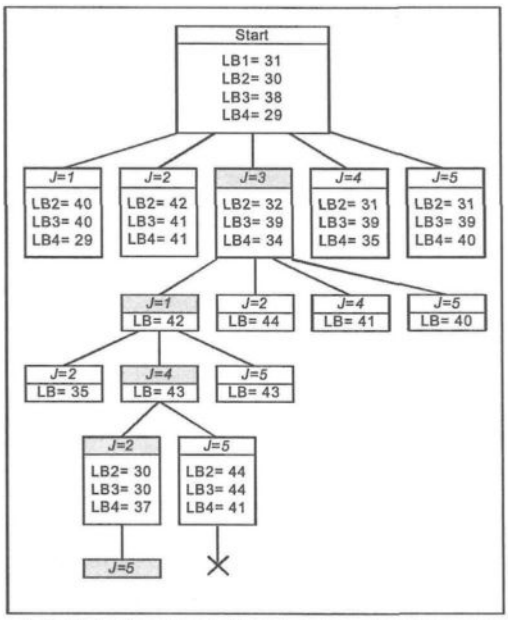


Figure 4-20 : Exécution de l'algorithme branch and bound sur une séquence de jobs dans le cas quelconque

En appliquant l'algorithme général de *branch and bound* sur notre problème, on obtient l'Arbre 4-6. Dans cet arbre, *LB* représente la valeur de la borne choisie entre *LB2*, *LB3* et *LB4*. Les cases grisées de cet arbre représentent la séquence optimale.



Arbre 4-6 : L'arbre de recherche dans le cas quelconque

4.3.4. Résultats empiriques

Nous présentons dans cette partie les différents résultats obtenus sur le problème de flow de *flow-shop* à deux machines avec des temps d'exécution et de latence quelconques. Dans un premier temps, nous rapportons les résultats relatifs aux heuristiques que nous avons utilisées. Ensuite, nous présentons les résultats de l'algorithme de *branch and bound* dont l'exécution pour une instance donnée été limitée à 3600 secondes, soit 1 heure. L'algorithme du *branch and bound* a été développé avec le

langage C++ sous l'environnement Visual Studio .NET 2003. Il a été exécuté sur une machine Pentium IV de 3.06 GHZ d'horloge et 512 Mo de mémoire vive.

- **Heuristiques**

Nous présentons dans cette partie les différents résultats relatifs aux bornes supérieures du problème de *flow-shop* à deux machines avec des temps d'exécution et de latence quelconques. Nous rappelons que les bornes supérieures, que nous avons utilisées, sont les heuristiques de Johnson modifié, ordre décroissant, *NEH* et de priorité. Le Tableau 4-25 résume les différents résultats trouvés pour les quatre heuristiques sur 10 instances de 10, 15, 20, 30, 40, 50 et 60 jobs. La deuxième colonne représente le temps CPU moyen d'exécution. Il est clair que l'heuristique de priorité (*UB4*) est la plus gourmande en temps. En moyenne *UB3* consomme moins de 6 secondes. Quant à *UB1* et *UB2*, elles consomment en moyenne moins d'un dixième de seconde. La troisième colonne du tableau représente la déviation de chaque heuristique par rapport à *LB1*. Les heuristiques *UB3* et *UB4* ont une déviation moyenne inférieure à 7 %, qui est relativement assez bonne. Au vue de ces résultats, il est clair que *UB4* est la meilleure borne supérieure, même si le rapport *déviation / temps* penche nettement en faveur de *UB3*.

UB	moyenne Temps CPU	moyenne |(Ubi-LB1)/LB1|
UB1 : Johnson	0,101557	0,109765
UB2 :Ordre décroissant	0,024957	0,121885
UB3 : NEH	5,991757	0,066290
UB4 : Priority	193,002157	0,062279

Tableau 4-25 : Résultats des heuristiques dans le cas quelconque

- **Algorithme de branch and bound**

Les tableaux ci-dessous présentent les résultats obtenus pour les classes de n = 5, 7, 9, 10, 12 et 15 jobs. Pour chaque classe, l'algorithme a été exécuté sur 10 instances générées d'une manière aléatoire. Les temps d'exécution et de latence sont générés aléatoirement dans un intervalle de temps entre 0 et 100. Nous y trouvons la borne *LB* initiale, la meilleure borne supérieure *UB* initiale, le meilleur *makespan* atteint par l'application, le nombre de nœuds visités et le temps *CPU* pour les différentes classes

d'instances sur lesquelles l'algorithme de *branch and bound* a été exécuté. Le symbole '**x**' signifie que le temps limite de 3600 secondes a été dépassé.

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
202	314	304	101	0,5
212	287	250	35	0,235
157	256	252	282	0,672
230	279	278	72	0,547
271	360	351	188	0,844
172	289	287	48	0,469
267	306	286	30	0,531
260	280	280	213	0,766
248	311	304	60	0,531
180	284	277	123	0,328

Tableau 4-26 : Simulation pour $n = 5$ dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
396	459	458	1842	5,923
329	416	410	1452	4,172
359	431	431	8683	16,627
356	415	412	8305	15,142
267	324	324	5801	9,422
328	389	386	2386	5,845
285	368	362	1535	3,203
218	263	254	2619	4,297
197	331	318	1503	2,375
316	337	337	516	2,079

Tableau 4-27 : Simulation pour $n = 7$ dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
440	485	483	16299	34,679
540	581	579	12147	32,022
580	598	598	88907	252,183
527	595	594	81137	188,64
391	490	483	216786	448,603
461	544	544	67399	160,353
383	440	438	23947	44,835
364	458	458	128987	262,364
420	560	543	168720	384,458
332	375	374	245562	441,542

Tableau 4-28 : Simulation pour $n = 9$ dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
559	597	596	79560	734,808
478	563	563	383515	X
496	523	523	416468	3482,21
396	517	517	398584	X
488	534	534	376227	X
511	556	551	216960	X
635	724	724	165542	X
560	585	584	41506	637,995
358	427	423	298785	X
325	464	446	280694	X

Tableau 4-29 : Simulation pour n = 10 dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
501	580	569	1139286	X
669	725	725	922775	X
618	650	650	877778	X
446	568	564	1147817	X
741	767	767	784949	X
657	673	673	1138119	X
459	508	508	1242814	X
622	643	643	1104795	X
632	670	664	1114856	X
477	649	608	1170930	X

Tableau 4-30 : Simulation pour n = 12 dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
701	767	767	154795	X
664	711	706	155179	X
659	768	768	152572	X
828	876	862	165754	X
736	771	771	151057	X
648	709	709	159106	X
803	858	856	183950	X
655	705	705	168341	X
796	888	888	136598	X
729	828	828	145226	X

Tableau 4-31 : Simulation pour n = 15 dans le cas quelconque

Comme nous pouvons le constater à travers ces résultats, l'algorithme de *branch and bound*, présenté ci-dessus, semble donner d'assez bons résultats, mais juste sur de petites instances, comme il fallait s'y attendre. En effet, à partir de $n = 10$, l'algorithme

ne produit pas de résultats, pour la majorité des instances, après une heure d'exécution allouée à chaque instance. Toutefois, nous restons convaincus que la conception de quelques règles de dominance améliorerait grandement l'efficacité de cet algorithme.

Chapitre 5 : Métaheuristiques pour le problème de flow-shop à deux machines avec des temps d'exécution et de latence quelconques

5.1 Introduction

Nous présentons dans ce chapitre deux métaheuristiques, l'algorithme de recherche avec tabous (*RT*) et un algorithme génétique (*AG*), pour la résolution du problème de *flow-shop* à deux machines avec des temps d'exécution et de latence quelconques. Une comparaison entre les deux méthodes est ensuite entreprise.

5.2 Algorithme de recherche avec tabous

L'algorithme que nous avons conçu est comme suit. Nous avons tout d'abord généré d'une manière aléatoire une solution initiale. En fait, il s'agit de créer une séquence aléatoire d'entrée sur la première machine : c'est notre pré-solution. Ensuite, dans une seconde étape, on génère le voisinage de cette pré-solution initiale. La taille du voisinage est égale au nombre de jobs traités. Notons que ce voisinage est généré à partir d'un ensemble de permutations sur cette pré-solution initiale. Les permutations se font entre deux jobs choisis aléatoirement. Chaque couple de jobs choisis est sauvegardé dans une *liste taboue*. La taille de cette liste est égale à 7. Le choix de cette valeur est justifié par un ensemble de tests que nous avons effectué. De plus, cette *liste taboue* est gérée comme une liste "circulaire". Après avoir généré le voisinage, on entame l'étape suivante, celle du calcul du *makespan* en prenant en compte des temps de latence. Les tests nous ont conduits aussi à faire itérer ce processus 600 fois. Notons que le passage de 600 à 700 itérations n'a pas amélioré les résultats, alors que celui de 500 à 600 les a nettement améliorés sans pour autant consommer beaucoup plus de temps d'exécution. Le critère d'aspiration consiste à vider le tiers de notre *liste taboue* lorsque cette dernière est remplie deux fois de suite.

L'algorithme de *RT* a été codé dans le langage C++ sous l'environnement Visual Studio .NET 2003. L'algorithme a été exécuté sur une machine Pentium IV à 3.06 GHZ d'horloge et avec 512 Mo de mémoire vive.

Le Tableau 5-1 présente les résultats trouvés pour l'ensemble de toutes les instances : 10 instances pour chaque classe, soit 70 instances au total. La deuxième colonne représente le temps moyen d'exécution. Le *makespan* moyen, le meilleur et le pire sont indiqués respectivement à la troisième, quatrième et cinquième colonne.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	4,3	545,1	418	725
15	11,5	791,4	703	899
20	23,7	1043,1	955	1116
30	74,3	1576,0	1397	1726
40	157,3	2085,1	1910	2202
50	295,2	2584,4	2397	2876
60	495,9	3052,4	2820	3384

Tableau 5-1 : Résultats générés par RT simple

5.3 Amélioration de RT

Nous introduisons dans ce qui suit des modifications dans le processus de *RT* afin d'améliorer les résultats trouvés.

5.3.1. Solution initiale

La première amélioration que nous pouvons effectuer est de démarrer *RT* à partir d'une bonne solution, au lieu de commencer par une solution générée aléatoirement. Nous générons cette solution à l'aide de l'heuristique *NEH* décrite au Chapitre 6. Le Tableau 5-2 présente les résultats appliqués aux mêmes instances précédentes avec une solution initiale générée par *NEH*.

Certes il y a une augmentation de la consommation temporelle qui varie de 1,4 % à 7,2 %, mais nous avons amélioré les résultats moyens de 1 %. De même, les meilleures et pires solutions ont été améliorées respectivement de 1,21 % et 1,18 % par rapport à *RT* simple pour les séquences à 60 jobs.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	4,4	546,4	422	725
15	11,8	789,3	705	890
20	24,3	1039,6	956	1112
30	75,4	1566,4	1380	1723
40	165,2	2077,7	1911	2202
50	307,2	2565,9	2378	2822
60	531,9	3032,0	2786	3344

Tableau 5-2 : Résultats générés par RT avec solution initiale

5.3.2. Diversification avec restart

Un restart consiste à effectuer, lors de la recherche, un démarrage à partir d'une nouvelle solution. Il sera ainsi possible d'explorer un autre espace du domaine de solutions. Cette technique représente l'une des techniques utilisées dans la littérature pour appliquer la diversification dans l'algorithme de recherche avec tabous.

- **Utilisation de NEH**

La deuxième amélioration que nous pouvons effectuer est d'introduire de la diversification et ce, en calculant à l'aide de l'algorithme de *NEH* la séquence associée à la meilleure trouvée dans le voisinage. Cette opération s'effectue à chaque 50 itérations. Ainsi, *RT* effectue un nouveau départ en commençant la recherche à partir d'une nouvelle séquence. Le Tableau 5-3 présente les résultats appliqués aux mêmes instances en utilisant la diversification avec *NEH*.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	4,6	545,7	417	725
15	12,6	788,6	706	890
20	27,0	1040,2	956	1111
30	86,7	1565,7	1373	1723
40	194,1	2075,1	1911	2202
50	374,6	2563,9	2378	2822
60	669,2	3030,5	2786	3344

Tableau 5-3 : Résultats générés par RT avec restart NEH

Cette amélioration a augmenté les temps d'exécution de 5% pour les petites instances à 15 % pour les plus grandes. Le *makespan* moyen a été amélioré pour tous les

ensembles d'instances, excepté ceux de 20 jobs. Nous avons pu améliorer les meilleures solutions trouvées pour les séquences de 10, 15 et 30. Le pire *makespan* des instances de 30 jobs a été lui aussi amélioré. Pour le reste, nous avons trouvé les mêmes valeurs.

- **Utilisation de NEH et Priority**

La troisième amélioration consiste à introduire de la diversification et ce, en calculant à l'aide de *NEH* ou *Priority* la séquence associée à la meilleure trouvée dans le voisinage. Cette opération s'effectue à chaque 50 itérations. Le choix de *NEH* ou *Prioriy* se fait de manière aléatoire en favorisant *NEH* qui consomme moins de temps. Nous utilisons ce qu'on appelle la technique de roulette pour effectuer ce choix [Dorigo, 1995]. Ainsi, *RT* effectue un nouveau départ en commençant la recherche à partir d'une nouvelle séquence. Le Tableau 5-4 présente les résultats appliqués aux mêmes instances que ci-dessus, en utilisant la diversification avec *NEH* et *Priority*.

Nombre de jobs N	***Temps moyen***	***Makespan moyen***	***Meilleur makespan***	***Pire makespan***
10	4,9	544,0	417	724
15	12,2	788,0	706	890
20	25,7	1039,1	955	1110
30	82,9	1566,0	1375	1723
40	182,5	2077,0	1909	2202
50	341,8	2563,4	2378	2822
60	583,7	3030,7	2786	3344

Tableau 5-4 : Résultats générés par RT avec restart NEH et Priority

Nous pouvons remarquer d'après ce même tableau que les temps moyens d'exécution se sont améliorés de 4 % pour les instances de petites tailles et de 12% pour les instances de 60 jobs. Le *makespan* moyen a été amélioré pour les instances de 10, 15, 20 et 50 jobs. Pour le reste le *makespan* moyen n' pas changé. Le meilleur *makespan* a été amélioré pour les séquences de 20 et 40 jobs. Les pires *makespan* ont été amélioré pour les instances de 10 et 20 jobs.

5.3.3. Intensification

Une intensification est une procédure qui consiste à intensifier la recherche dans une zone bien déterminée de l'espace des solutions. Nous utilisons pour cela la recherche locale (*RL*).

- **Utilisation de RL**

L'intensification consiste, dans notre cas, à appliquer toutes les 50 itérations une *RL* sur la meilleure solution trouvée dans le voisinage. *RL* consiste à trouver la meilleure séquence, suite à des permutations améliorantes. Ainsi, une solution est gardée si elle ne détériore pas la précédente. Les positions choisies pour les permutations sont tirées aléatoirement. *RL* s'arrête après avoir effectué 100 permutations. Le Tableau 5-5 présente les résultats appliqués aux mêmes instances en utilisant *RL*. Notons que dans ce cas, nous n'avons pas utilisé d'intensification.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	5,5	546,4	422	725
15	13,7	790,1	706	891
20	26,9	1040,8	956	1114
30	81,2	1566,1	1373	1723
40	174,4	2076,8	1911	2202
50	320,3	2565,3	2378	2822
60	534,0	3032,8	2791	3344

Tableau 5-5 : Résultats générés par RT avec RL

Comparé au Tableau 5-4, les temps d'exécution se sont améliorés pour les grandes instances, avec une diminution de 8 % de ceux-ci. Le *makespan* moyen a été détérioré de 0,07 % pour les instances de 50 et 60 jobs et moins de 0,5 % pour le reste. Les pires *makespan* ont été aussi détériorés pour les instances de 10, 15, 20 et 30 jobs, mais sont égaux à ceux de 40, 50 et 60 jobs. Le meilleur *makespan* des séquences de 30 jobs a été amélioré, ceux de 15 et 50 sont restés inchangés, alors que le reste, ils ont été détériorés.

- **Utilisation de RL et de la diversification avec NEH et Priority**

La dernière amélioration que nous avons effectuée consiste à utiliser à la fois la diversification et l'intensification. La diversification consiste dans ce cas à utiliser la

méthode de *restart* avec *NEH* et *Priority*. Après plusieurs tests, nous avons décidé de lancer la *RL* serait lancé toutes les 100 itérations et la diversification toutes les 30 itérations. Le Tableau 5-6 présente les résultats associés à cette dernière version de *RT*.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	5,9	543,9	417	724
15	14,6	787,8	706	890
20	29,2	1038,9	955	1109
30	86,4	1565,0	1370	1723
40	184,2	2076,7	1908	2202
50	337,0	2565,1	2378	2822
60	600,3	3030,5	2786	3344

Tableau 5-6 : Résultats générés par RT avec intensification et diversification

Comparée aux résultats du Tableau 5-5, cette version de *RT* consomme plus de temps (5% pour les petites instances et 10 % pour les plus grandes). Cependant, la valeur de tous les *makespan* moyens s'est améliorée. Il en est de même pour les pires valeurs du makespan où on les améliore trois fois et de cinq fois pour les meilleures valeurs. Pour les autres cas, on égale les précédents résultats.

5.4 Algorithme génétique

Dans cette section, nous décrivons, dans un premier temps, le contexte expérimental de l'algorithme génétique implémenté avec la plateforme *ParadisEO* [Cahon *et al.*, 2004]. Ensuite, nous détaillons l'implémentation de cet algorithme. Nous terminons cette section par la présentation des résultats obtenus de la simulation, que nous avons effectuée sur cet algorithme, et par la comparaison de ces derniers à ceux de l'algorithme avec tabous.

5.4.1. Contexte expérimental : ParadisEO

Nous avons utilisé la plateforme *ParadisEO* [Cahon et *al.*, 2004] pour implémenter l'algorithme génétique que nous avons conçu pour la résolution du problème de *flow-shop* à deux machines et avec des temps de latence. *ParadisEO* est un cadre de travail mettant en œuvre des librairies ANSI-C++ pour la conception et l'élaboration de métaheuristiques parallèles et hybrides dédiées à la résolution mono et multi-objectifs des

problèmes d'optimisation combinatoires. Ce cadre de travail est formé de quatre librairies:

- *Evolving Object* (*EO*) : Cette librairie est à la base de tout le projet. Elle fournit des outils pour élaborer des métaheuristiques basées sur des populations (Algorithme génétique, Programmation génétique, etc.).
- *Parallel Evolving Object* (*PEO*) : Cette librairie fournit des outils pour mettre ne œuvre des métaheuristiques parallèles et distribuées. Elle a été créée à la base pour assurer la parallélisation de la librairie EO de manière transparente et robuste. Nous pouvons trouver notamment le modèle en île, le modèle à parallélisation d'évaluation, le modèle cellulaire, etc.
- *Move Object* (*MO*) : La librairie MO permet d'implémenter des métaheuristiques à base de solution telles que la recherche avec tabous (*RT*), le recuit simulé (*RS*) ou encore le *hill climbing* (*HC*).
- *Multi Objectif Evolving Object* (*MOEO*) : Cette librairie permet d'implanter des métaheuristiques multi-objectifs avec plusieurs modèles tels que MOGA, NSGA-II, SPEA2, IBEA, etc. Elle permet aussi d'effectuer des calculs de performances des métaheuristiques.

Pour les besoins de notre travail, nous n'avons utilisé que la bibliothèque *EO*. Pour cela nous avons implémenté notre algorithme sur la même machine (512 Mo de RAM et 3,6 GHZ) mais sous un environnement *SuSe 10.2* avec *gc++ 4.6*. Nous l'avons exécuté sur le même ensemble des 70 instances générées précédemment pour l'algorithme de recherche avec tabous.

5.4.2. Implémentation de l'algorithme génétique

Les individus sont représentés par l'ensemble de jobs formant une séquence. Ainsi un individu représente une séquence d'exécution des différents jobs et la taille d'un individu est égale aux nombre de jobs dans une séquence. Après avoir effectué plusieurs tests, la taille de la population, la plus appropriées, est fixée à 40. Les individus seront générés aléatoirement. Le calcul du *makespan* de chaque individu représente notre évaluation. La sélection se fait d'une manière aléatoire. Nous avons gardé aléatoirement 40 individus entre les parents et les enfants générés. Le remplacement se fait d'une

manière élitiste. A chaque génération, il est produit 20 enfants issus des 40 parents. Les tests ont justifiés l'utilisation de l'opérateur de croisement COX au dépend de OX, PMX ou encore LX [Mühlenbein, 1993]. La probabilité du croisement est de 100%. Les 20 enfants générés sont issus de ce croisement. Nous avons aussi utilisé une *RL* comme opérateur de mutation. Pour un enfant sur quatre générés, nous avons appliqué une *RL* qui effectue un ensemble de permutations amélioratives. Cette recherche locale s'arrête après avoir effectué $N \times 20$ permutations où N est le nombre de jobs traités. Pour pouvoir comparer la recherche avec tabous et l'algorithme génétique, le critère d'arrêt de ce dernier est le temps consommé par la recherche avec tabous pour la même instance.

5.4.3. Comparaison des algorithmes RT et AG

Le Tableau 5-7 illustre les résultats de notre AG sur les 70 instances de données générées précédemment. La deuxième colonne représente le *makespan* moyen des 10 instances de chaque classe. La troisième et quatrième colonne représente, respectivement, le meilleur et le pire *makespan* trouvés.

N	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	544,7	423	724
15	788,6	706	884
20	1041,1	965	1113
30	1565,5	1373	1721
40	2076,9	1912	2214
50	2565,5	2396	2826
60	3031,3	2786	3344

Tableau 5-7 : Résultats AG

Au vue de ces résultats, nous pouvons tirer les conclusions suivantes : *RT* est meilleur qu'AG du point de vue de la valeur moyenne du *makespan*. Il en est de même 5 fois sur 7 concernant le pire *makespan*. Les résultats de *AG* sont comparés à la dernière version de *RT* avec de la diversification utilisant *NEH* et *Priority* et de l'intensification utilisant *RL*. Nous pouvons donc conclure que *RT* domine *AG* pour cet ensemble de données.

La Figure 5-1 présente les temps moyen d'exécution des quatre bornes supérieures ainsi que de l'algorithme de recherche avec tabous. Si les temps d'exécution

de *RT* et de *UB4* explosent avec l'accroissement du nombre de jobs, ceux de *UB1*, *UB2* et *UB3* restent relativement constants. En effet, *UB1* ne dépasse pas le tiers de la seconde, *UB2* ne dépasse pas le dixième de la seconde, et quant à *UB3*, elle ne dépasse pas 25 secondes en moyenne.

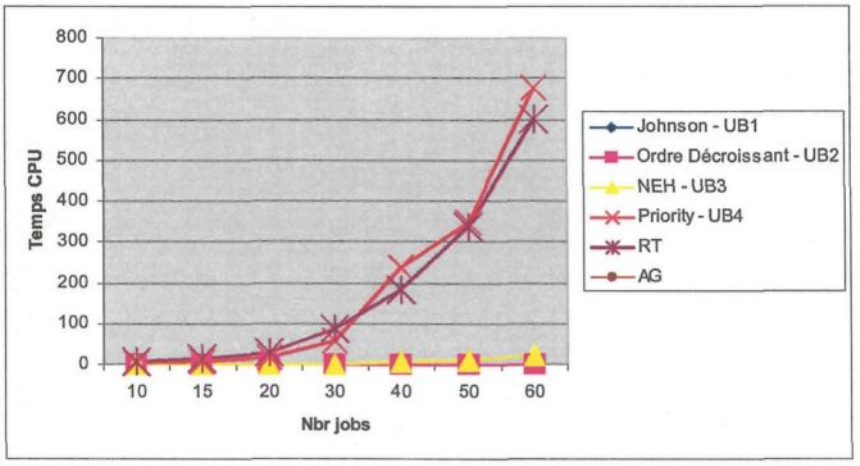


Figure 5-1: Comparaison des temps d'exécution RT et UB

La Figure 5-2 présente les déviations moyennes des quatre bornes supérieures ainsi que *RT*. La première remarque que nous pouvons faire est que la déviation moyenne diminue avec l'accroissement du nombre de jobs des séquences. Nous pouvons aussi distinguer deux groupes formés respectivement par *UB1* et *UB2* d'un côté et de *UB3*, *UB4* et *RT* d'un autre. Notons que le deuxième groupe donne de meilleurs résultats et que *UB3*, *UB4* et *RT* se confondent aux alentours de 2,5 % de déviation par rapport à *LB1* pour les séquences à 60 jobs. Nous pouvons aussi remarquer que *RT* domine les quatre bornes en termes de déviation.

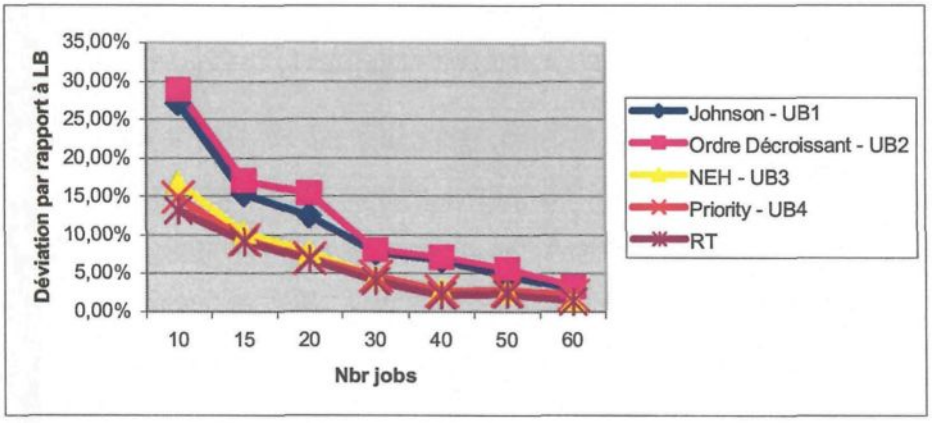


Figure 5-2: Comparaison des déviations moyennes par rapport à LB1

Conclusion

Nous avons étudié dans ce mémoire le problème de *flow-shop* à deux machines avec des temps de latence. Dans un premier temps, nous avons traité le problème avec des temps d'exécution unitaires avant de nous pencher sur les temps d'exécution quelconques. Les problèmes de type *flow-shop* ont été traités depuis la fin des années cinquante, notamment avec les travaux de [Johnson, 1954]. Après les vagues du *fordisme* et du *taylorisme*, les chaînes de production de type « ateliers » ont commencé à apparaître progressivement pour représenter de nos jours plus de la moitié des systèmes industriels des entreprises manufacturières [Pinedo, 2002]. Plusieurs variantes de ce modèle ont été utilisées dans l'industrie et dans les études théoriques. La pratique a cependant montré que les temps de latence liés aux jobs doivent être pris en considération étant donnée leur importance dans l'industrie d'aujourd'hui. En effet, Panwalkar et *al.* [1973] ont reporté que 19 % des activités industrielles subissent des temps de réglage et de transport liés aux jobs.

Après un chapitre introductif, nous avons présenté dans le Chapitre 2 les problèmes d'ordonnancement et décrit les différents modèles existant dans la littérature. Nous avons aussi motivé notre étude à l'aide d'un exemple de *flow-shop* à deux machines prenant en compte les temps de latence. Le critère que nous avons considéré dans notre travail est celui du *makespan*. Après y avoir introduit brièvement la *NP*-complétude, nous y avons survolé, à la fin de ce chapitre, les principales méthodes de résolution des problèmes d'ordonnancement, à savoir i) essayer de trouver un algorithme pseudo-polynomial ou de démontrer qu'il est *NP*-difficile au sens fort, *ii*) utiliser une méthode exacte, *iii*) utiliser une méthode approximative ou *iv*) utiliser une relaxation. Dans un deuxième temps, nous avons détaillé deux méthodes de résolution: l'approche exacte et approximative. Concernant la première approche, nous nous sommes concentrés sur la méthode de *branch and bound*. Cette méthode se base sur le calcul de bornes supérieures et inférieures à certains nœuds de l'arbre de recherche créé, afin d'élaguer certaines de ses branches au plus tôt et accélérer ainsi l'exécution de l'algorithme. Dans certains cas, il est aussi possible d'y introduire des règles de dominances pour mieux encore élaguer au plus

tôt certaines branches. Concernant l'approche approximative, nous avons abordé les heuristiques constructives, amélioratives ainsi que les métaheuristiques. Contrairement aux méthodes exactes, et pour des problèmes *NP*-difficiles, les heuristiques constituent une approche appropriée pour trouver de « bonnes » solutions en un temps « raisonnable ». Parmi les métaheuristiques, que nous avons discutées, nous avons détaillé l'algorithme de recherche avec tabous et les algorithmes génétiques.

Dans le Chapitre 3, nous avons décrit le problème du *flow-shop* et donné quelques propriétés pour le cas à deux machines avec des temps de latence. Nous avons fait également une revue de littérature de la résolution du problème du *flow-shop*. Cette revue de la littérature a concerné les méthodes heuristiques et la méthode de *branch and bound* pour la résolution d'un problème de *flow-shop*. Nous y avons ainsi décrit certaines heuristiques dont celles que nous avons utilisées, à savoir la méthodes de Johnson [1954], CDS [Campell et *al.*, 1970], Palmer [1965] et *NEH* [Nawaz et *al.*, 1983]. La revue de la littérature concernant la résolution du problème du *flow-shop* avec la méthode de *branch and bound* a, quant à elle, pu nous donner une idée sur les différentes bornes supérieures et inférieures utilisées ainsi que les règles de dominance.

Pour la résolution du problème de minimisation du *makespan* d'un problème de *flow-shop* à deux machines avec des temps de latence à l'aide de la méthode de *branch and bound*, nous avons considéré dans un premier temps, au Chapitre 4, le cas des temps d'exécution unitaires. Nous y avons présenté des bornes inférieures et supérieures ainsi que des règles de dominances pour l'implémentation d'un algorithme de *branch and bound*. Notons que cette partie est entièrement basée sur les travaux de [Moukrim et Rebaïne, 2005] et [Yu, 1996]. Dans un deuxième temps, nous sommes passés aux temps d'exécution quelconques. Dans ce cas, nous nous sommes servis de bornes inférieures de Yu [Yu, 1996], et avons conçu des bornes supérieures, qui sont des versions modifiées de l'algorithme de Johnson [1954], CDS [Campbell et *al.*, 1970], règles de priorité de Palmer [1965] et *NEH* [Nawaz et *al.*, 1983]. Nous avons tout d'abord évalué les heuristiques. Ensuite, nous avons appliqué l'algorithme de *branch and bound* en utilisant seulement les bornes inférieures avant d'y incorporer les bornes supérieures finalement. Les résultats se sont nettement améliorés du point de vue du nombre de nœuds visités et du meilleur *makespan* trouvé.

Le Chapitre 5 décrit la résolution du problème cité ci-dessus avec un algorithme de recherche avec tabous et un algorithme génétique. Concernant l'algorithme de recherche avec tabous, nous l'avons démarré avec une solution initiale générée aléatoirement, une liste taboue à 7 éléments et un voisinage de taille égal au nombre de jobs dans une séquence. Un voisinage contient des solutions issues de la solution courante après y avoir effectué une permutation de jobs. Une solution initiale générée avec *NEH* [Nawaz *et al.*, 1983], une diversification avec restart en utilisant *NEH* et la règle de priorité de Palmer [1965] et une intensification en utilisant de la recherche locale améliorative sont les différentes améliorations que nous avons ajoutées à l'algorithme de recherche avec tabous. Pour chaque amélioration, nous avons pu remarquer une amélioration des résultats. L'algorithme génétique a été implanté avec la plateforme *ParadisEO* [Cahon et *al.*, 2004] avec une population de 40 individus, un croisement COX, une mutation avec de la recherche locale améliorative, une sélection aléatoire et un remplacement élitiste. Le critère d'arrêt de l'algorithme génétique correspond au temps consommé par la recherche avec tabous pour la même instance. Les résultats ont montré que l'algorithme de recherche avec tabous domine clairement l'algorithme génétique.

Certes, l'algorithme de *branch and bound* a généré d'assez bons résultats, pour les instances de petites tailles, mais l'élaboration de meilleures bornes inférieures et supérieures pourraient améliorer les résultats. La conception de règles de dominances produira des répercussions positives sur la qualité les résultats. Concernant l'algorithme de recherche avec tabous, le passage vers un voisinage plus varié, voire multiple, pourrait améliorer les résultats. Une hybridation avec l'algorithme génétique, ou d'autres heuristiques, est aussi envisageable dans des travaux futurs de recherche.

Références

[Baker, 1974] Baker (K. R.). - Introduction to Sequencing and Scheduling . - Wiley, 1974.

[Blazewicz et *al.*, 1994] J. Blazewicz, K.H. Ecker, G. Schmidt, J. Weglarz, 1994. Scheduling in Computer and Manufacturing Systems. *Springer - Verlag, Second, Revised Edition.*

[Boivin, 2005] Simon Boivin, Mars 2005. - Résolution d'un problème de satisfaction de contraintes pour l'ordonnancement d'une chaîne d'assemblage automobile. - UQAC.

[Botta-Genoulaz, 2000] V. Botta-Genoulaz, 2000. Hybrid flow-shop scheduling with precedence constraints and time lags to minimize maximum lateness. *International Journal of Production Economics. Amsterdam.* Vol. 64, Iss. 1-3; p. 101

[Bouzgarrou, 1998] M.E. Bouzgarrou, 1998. Parallélisation de la Méthode du "Branch and Cut" pour Résoudre le Problème du Voyageur de Commerce. *Thèse réalisée à l'Institut National Polytechnique de Grenoble.*

[Cahon et *al.*, 2004] S. Cahon, N. Melab and E-G. Talbi, "ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics", *Journal of Heuristics*, vol. 10(3), pp.357-380, May 2004.

[Campbell et *al*, 1970] H. G. Campbell., R. A. Dudek et M. L. Smith, 1970. A Heuristics Algorithm For The *n* Job *m* Machine Scheduling Problem. *Mgmt. Sci* 16, 630-637.

[Carlier et *al*, 1988] Carlier J. and Chrétienne Ph., 1988."Problèmes d'Ordonnancement : Modélisation, Algorithmes et Complexité". *Editions Masson.*

[Chen, 2006] Jen-Shiang Chen, A branch and bound procedure for the reentrant permutation flow-shop scheduling problem, The International Journal of Advanced Manufacturing Technology, v 29 n.11, p 1186–1193, August 2006.

[Conway *et al.*, 1967] R. W. Conway, W.L. Maxwell, L. W. Miller., 1967. Theory of Scheduling. *Addison – Wisley, Reading, MA.*

[Cook, 1971] S. A. Cook, 1971. An Overview of Computational Complexity. *Turing award lecture, Communications of the ACM,* 26, 6, pp-. 400-408.

[Cormen et *al.*, 2002] Cormen T., Leiserson C., Rivest R., Introduction à l'algorithmique, 2e édition, Paris, Dunod, 2002

[Dantzig et *al.*, 1959] G. B Dantzig, DR Fulkerson et S. M Johnson, 1959. On a Linear-Programming, Combinatorial Approach to the Traveling-Salesman Problem *Operations Research*, 58-66.

[Dell'Amico et Vaessens, 1996] M. Dell'Amico, R. J. M. Vaessens, 1996. Flow and Open Shop Scheduling on two machines with Transportations Times and machines – independent processing times is NP – hard. *Dipartimento di Economia Politica, Universita di Modena.*

[Dell'Amico, 1993] M. Dell'Amico, 1993. Shop Problems with Two- Machines and Time Lags. *Preprint.*

[Della Croce et *al.*, 2002] F. Della Croce and M. Ghirardi and R. Tadei, 2002. An improved branch and bound algorithm for the two-machine total completion time Flow-shop problem. *European Journal of Operational Research.*

[Dorigo et Stützle, 2004] Dorigo, Stützle, 2004. Ant Colony Optimization. *Cambridge, Massachusetts et London, England: The MIT* Press.

[Dorigo, 1995] M. Dorigo, A.Colorni, V. Maniezzo, 1995. New Results of an Ant System Approach Applied to the Asymmetric TSP. *Proceedings of the Metaheuristics International Conference, Hilton Breckenridge, Colorado, USA, I. H. Osman & J. P. Kelly (Eds.), Kluwer Academic Publishers, 356–360.*

[Feo et Resende, 1995] T. Feo and M. G. Resende. Greedy randomized adaptive search procedure. *Journal of Global Optimization*, 6:109–133, 1995.

[Fondrevelle, 2002] J. Fondrevelle, 2002. DEA GSI - Aspects pratiques concernant l'ordonnancement de produits périssables : application au cas du flow-shop de permutation-. Ecole des Mines de Nancy.

[Framinan et *al.*, 2003] J .M .Framinan, R. Leisten et C. Rajendran, 2003. Different initial sequences for the heuristic of Nawaz, Enscore et Ham to minimize makespan, idletime or flowtime in the static permutation flowshop sequencing problem. *International Journal of Production Research* 41 (1), 121-148.

[French, 1982] S. French, J. Wiley & Sons, 1982. Sequencing and Scheduling. New York.

[Gantt, 1919] Henry. L. Gantt. Organizing for Work, Harcourt, Brace, and Howe, New York, 1919. *Reprinted by Hive Publishing Company, Easton, Maryland, 1973.*

[Ghallab et Allard, 1983] M. Ghallab et D.G. Allard, 1983. An efficient near admissible heuristic search algorithm. In Proceedings IJCAI-83.

[Glover, 1986] F. Glover, 1986. Future Paths for Integer Programming and Links to Artificial Intelligence. *Comput. & Ops. Res.* Vol. 13, No.5, pp. 533-549, 1986

[Glover, 1997] Glover F. A Template for Scatter Search and Path Relinking. In: Hao JK, Lutton E, Ronald E, Schoenauer M, and Snyers D (eds.) *Lecture Notes in Computer Science* 1997; 1363: 13-54.

[Graham et *al.*, 1979] R.L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan,1979. Optimization and approximation in deterministic sequencing and scheduling : Survey. Ann. Discrete Math, 5, 287 – 326.

[Gupta et *al.*, 2001] J.N.D. Gupta, V.R. Neppalli and F. Werner, 2001. Minimizing total flow time in a two-machine flowshop problem with minimum makespan. *International Journal of Production Economics, 69 pp. 323-338*

[Gupta, 1972] J.N.D.Gupta, 1972. Heuristic Algorithms for Multi-Stage Flowshop Problem. *AIIE Trans.* 4, 11-18.

[Hansen, 1986]. P. Hansen, 1986. The Steepest Ascent, Mildest Descent Heuristic for Combinatorial Programming. *Congrès sur les Méthodes Numériques en Optimisation, Capri*, Italie.

[Hariri et Potts, 1997] Hariri AMA, Potts CN. A branch and bound algorithm for the two-stage assembly scheduling problem. European Journal of Operational Research 1997;103:547-56.

[Holland, 1975]. J. H. Holland, 1975. Adaptation in Natural and Artificial Systems. *The University of Michigan Press, Ann Arbor, MI.*

[Ignall et Schrage, 1965] E. Ignall, and L.E. Schrage. Application of the Branch and Bound Technique to Some Flow-shop Scheduling Problems, Operarions Research, vol. 13, No. 3. 1965.

[Ishibuchi et *al.*, 1995] H. Ishibuchi, S. Misaki, H. Tanaka, 1995. Modified Simulated Annealing Algorithms for the Flow-shop Sequencing Problem. *European Journal of Operational Research. Vol: 81, Issue: 2.*

[Jin et *al.*, 2006] Z.Jin, Z. Yang, T.Ito, 2006. Metaheuristic algorithms for the multistage hybrid flowshop scheduling problem. *International Journal of Production Economics*. Amsterdam. Vol. 100, Iss. 2; p. 322

[Johnson, 1954] S. M. Johnson, *Optimal two- and three-stage production schedules with setup times included*, Naval Res. Logist. Quart. 1 (1954), 61–68.

[Johnson, 1958] S. M. Johnson, 1958. Discussion : sequencing *n* jobs on two jobs with arbitrary time lags. *Management Sci*, 5, 299 – 303

[Kanigel, 1997] R. Kanigel, 1997. The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency. *New York: Viking*

[Kern et Nawijn, 1991] W. Kern, W. N. Nawijn, 1991.Scheduling multi-operation jobs with time lags on a single machine. *Proceeding 2nd Twente Workshop on Graphs and Combinatorial Optimization.*

[Kozanidis et *al.*, 2002] G. Kozanidis, E. Melchrinoudis, 1 Décembre 2002. A Branch & Bound Algorithm for the 0-1 Mixed Integer Knapsack Problem with Linear Multiple Choice Constraints. *Computers & Operations Research*

[Lawler et *al.*, 1985] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, 1985. The Traveling Salesman Problem. *Wiley.*

[Lawler et *al.*, 1993] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys, 1993. Sequencing and Scheduling : Algorithms and Complexity. *Handbook in operations Research and Management Science. Vol* 4, 445-522.

[Lenstra, 1991] J.K. Lenstra, Private Communication 1991, cited in [Lawler et *al*, 1993] and [Dell'Amico 1993].

[Lenstra, 1994] J. K. Lenstra, 1994. A Note on Two Machine Flow-shop Problem with Delays. *Ce manuscript n'est pas publié*

[Levner, 1969] Levner, E.M.. Optimal Planning of Parts Machining on a Number of Machines. Automation and Remote Control 12, 1972–1978.

[Li et Tang, 2005] S.Li, L. Tang, 2005. A Tabu Search Algorithm Based on New Block Properties and Speed-up Method for Permutation Flow-Shop with Finite Intermediate Storage. *Journal of Intelligent Manufacturing. London.* Vol. 16, Iss. 4-5; p. 463

[Lomnicki, 1965] Z. Lomnicki, 1965.A Branch and Bound Algorithm for The Exact Solution of the Three-Machine Scheduling Problem. *Opnl.Res.Quart.* 16, 89-100.

[Maggu et *al.*, 1982] Maggu, P. L., M. L. Singhal, N. Mohammad et S. K. Yadav. 1982. On N-Job 2-Machine Flowshop Scheduling Problem with Arbitrary Time Lags and transportation times of jobs. *J. Opns. Res.* 25, 219-227.

[Mellor, 1966] P. A. Mellor, Juin 1966. Review of Job Shop Scheduling. *Oper Res Quart.* I7, 161-171.

[Metropolis et *al.*, 1953] N. Metropolis, , Rosenbluth, A. W., Rosenbluth, N. M., Teller, A. H., et E Teller, 1953.Equation of state calculations by fast computing machines. *J. Chemical Physics,* 21 :1087–1091.

[Mitten, 1958] L.G. Mitten, 1958. Sequencing *n* Jobs on Two Jobs with Arbitrary Time Lags. *Management Sci*, Vol 5, Issue 3, 293- 298.

[Mitten, 1959] L.G. Mitten, 1959. A Scheduling Problem. *J. Indust.* Vol 10, 131-135.

[Moccellin et Dos Santos, 2000] J. V. Moccellin, M. O. Dos Santos, 2000. An Adaptive Hybrid Metaheuristic for Permutation Flowshop Scheduling. *Journal of Control and Cybernetics,* 29 (3), 761 – 771.

[Moccellin, 1995] J. A. V. Moccelin, 1995. A New heuristic Method for the Permutation Flow-shop Problem. *Journal of Operational Research Society, 46, 883 – 886.*

[Moukrim et Rebaïne, 2005] Moukrim, A., Rebaine, D, 2005. A Branch and Bound Algorithm for the UET Two – machine Flow-shop Problem with Time Delays.

[Mühlenbein, 1993] Mühlenbein H.,"Evolutionary Algorithms: Theory and applications" (Wiley) 1993.

[Munier et Rebaine, 2006] A. Munier-Kordon, D. Rebaine, Polynomial time algorithms for the UET permutation flowshop problem with time delays. Computers & Operations Research, A apparaître.

[Muth et Thompson, 2006] Muth, J. F., Thompson, G. L.: Industrial Scheduling. Englewood Cliffs, N. J.: Prentice-Hall 1963

[Nagar et *al.*, 1995] A. Nagar, S.S. Heragu and J. Haddock. A Branch-and-Bound Approach for a Two-machine Flowshop Scheduling Problem. *Journal of the Operational Research Society*,46 721-734, 1995.

[Nagasawa et *al.*, 1989] Nagasawa, H., K. Nango, N. Hirabayashi et N. Nishiyama, 1989. Method for Distributing Tools in 2-Machine Flowshop Type FMS. *Trans. Japan Soc. Mech. Eng. Part C*, 55, 1113-1146.

[Nawaz et *al.* 1983]. M. Nawaz, E. E Enscore, I. Ham, 1983. A Heuristic Algorithm for the *m*-Machine, *n*-Job Flow-Shop Sequencing Problem- *OMEGA*

[Osman et Potts, 1989] I. H. Osman, C. N. Potts, 1989. Simulated Annealing for Permutation Flow-Shop Scheduling. *Omega (Oxford), 551-557*

[Palmer, 1965] D. S. Palmer, 1965. Sequencing Jobs Through a Multi- Stage Process in the Minimum Total Time – A Quick Method of Obtaining Near Optimum. *Opns. Res. Quart.* 16, 101-107.

[Panwalkar et *al.*, 1973] Panwalkar, S. S., R. A. Dudek and M. L. Smith (1973). "Sequencing research and the industrial scheduling problem." Symposium on the Theory of Scheduling and Its Apllications: 29-38.

[Papadimitriou et Kanellakis, 1990] C. H. Papadimitriou et P. C. Kanellakis, 1980. Flowshop scheduling with limited temporary storage. *Journal of the ACM,* 27(3), 533-549.

[Perraud-Echalier et *al.*, 1991] F. Perraud-Echalier et A. Dussauchoy, 1991; Problèmes d'ordonnancement sur machines parallèles: apport du recuit simulé (scheduling problems in parallel processor shops: contribution of simulated annealing). *Université de Lyon 1, Lyon, France*

[Pinedo, 2002] M. Pinedo, 2002. Scheduling : Theory, Algorithms, and Systems. *Pearson Education, Second Edition*

[Ponnambalam et *al*, 2001] S. G. Ponnambalam, P. Aravindan, 2001. Constructive and Improvement Flow-shop Scheduling Heuristics: An Extensive Evaluation. *Production Planning & Control* 12 (4), 335 – 344.

[Raghavachari, 1988] M. Raghavachari, 1988. Scheduling Problems with Non – Regular Penalty Functions : A Review. *Opsearch*, Vol 25, PP 144-164.

[Rajendra, 1992] C. Rajendra, 1992. Two stage Flowshop Scheduling Problem. *Journal of Operational Research Society, 43, 871 – 884.*

[Rebaïne, 1997] D. Rebaïne, 1997. Ordonnancement d'ateliers flow-shop et open shop avec des temps de transport : algorithmes et complexité. *Thèse de doctorat, Université M. Mammeri de Tizi-Ouzou, Algérie.*

[Rebaïne, 2000] D. Rebaïne, 2000. Une Introduction à l'Analyse des Algorithmes. *ENAG*, Alger.

[Rebaïne, 2005] Rebaine D. Permutation shop vs. non permutation flow-shop with delays. Journal of Computers & Industrial Engineering. Vol 48,

p357-362, 2005.

[Rinnoy Kan, 1976] A. H. G. Rinnoy Kan, 1976.Machine Scheduling Problem: Classification, Complexity and Computations. *Nijhoff, the Hague.*

[Rios-Mercado et Bard, 1999 b] R.Z. Rios-Mercado and J.F. Bard. An enhanced TSP-based heuristic for makespan minimization in a flowshop with setup times. Journal of Heuristics, vol 5, p 57-74, 1999.

[Rios-Mercado et Bard, 1999] R. Z. Rios-Mercado and J. F. Bard, A branch-and-bound algorithm for permutation flowshops with sequence-dependent setup times. IIE Transaction, vol 31, p 721-731. 1999.

[Ronconi, 2005] Ronconi D. P., A Branch-and-Bound Algorithm to Minimize the Makespan in a Flowshop with Blocking, Annals of Operations Research, Vol 138, p 53-65, 2005

[Ruiz et *al.*, 2005] R. Ruiz et C. Maroto, 2005. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research* 165 (2005) 479–494

[Santos et *al.*, 1995] D.L. Santos, J.L. Hunsucker, D.E. Deal, 1995. Global lower bounds for flow-shops with multiple processors; European Journal of Operational Research, 80:112--120.

[Sarin et Lefoka, 1993] S. Sarin et M. Lefoka, 1993. Scheduling heuristic for the *n*- job *m*-machine flowshop. *OMEGA, The International Journal of Management Science* 21 (6), 753- 764.

[Sibeyn, 2005] J. Sibeyn. Branch and Bound. 12 mars 2005.

[Stern et Vitner, 1990] H. Stern and G. Vitner, 1990.Scheduling Parts in a Combined Production Transportation Work Cell, *Journal Opl. Res. Soc.* 41, 625-632.

[Stinson et Smith, 1982] J. P Stinson, A.W Smith, 1982. Heuristic Programming Procedure for Sequencing the Static FlowShop. *International Journal of Production Research.* 20, 753-64.

[Suhani et Mah, 1981] Suhami, I. and R.S.H. Mah. . An Implicit Enumeration Scheme for the Flowshop Problem with No Intermediate Storage. Computers and Chemical Engineering 5, 83–91. 1981.

[Szwarc, 1973] W.Szwarc, 1973. Optimal Elimination Methods in the $m \times n$ Flowshop Scheduling Problem. *Opns.Res.* 21, 1250-1259.

[Taillard, 1990] E Taillard, 1990. Some Efficient Heuristic Methods for the Flow-shop Sequencing Problem. *European Journal of Operational Research. ina.eivd.ch*

[Talbi, 2002] E. G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8:541–564, 2002.

[T'Kindt, 2001] V.T'Kindt, J.N.D Gupta and J-C. Billaut, 2001. A branch and bound algorithm to solve a Two-Machine Bicriteria Flow-shop scheduling problem. ORP3.

[Tmiz et Erol, 2003] I. Temiz and S. Erol. Fuzzy branch and bound algorithm for flowshop scheduling, Journal of Intelligent Manufacturing, v.15, p.449-454, 2004.

[Tozkapan et *al.*, 2003] A. Tozkapan , O. Kirca and C. Chung, A branch and bound algorithm to minimize the total weighted flowtime for the two-stage assembly scheduling problem, Computers and Operations Research, v.30 n.2, p.309-320, February 2003

[Vaessens et Dell'Amico, 1995] R.J.M. Vaessens, M. Dell'Amico, 1995. Flow and Open Shop Scheduling on Two Machines with transportation Times and Machine- Independent Processing Times is *NP*- Hard, *Preprint*.

[Van de Velde, 1990] S. L. Van de Velde, 1990. Minimizing the sum of the job completion times in the two-machine flow-shop by lagrangean relaxation. Annals of Operations Research, 26 257-268.

[Wang et *al.*, 1995] C. Wang, C. Chu and J.-M. Proth. A branch-and-bound algorithm for n-job two machine flowshop scheduling problems. Emerging Technologies and Factory Automation, ETFA '95, Proceedings. Vol 2, p 375-383. 1995.

[Werner, 1993] F. Werner, 1993. On the Heuristic Solution of the Permutation Flow-shop Problem by Path Algorithms. *Computers and Operations Research*, 707 - 722.

[Widmer et Hertz, 1989] M.Widmer, A. Hertz, 1989. A New Heuristic Method for the Flowshop Sequencing Problem. *Eur. J. Opnl.Res.* 41, 186-193.

[Widmer et Hertz, 1989] M Widmer, A Hertz, 1989. A New Heuristic Method for the Flow-shop Sequencing Problem. *European Journal of Operational Research.* 186-193.

[Wismer, 1972] D.A. Wismer, 1972. Solution of Flowshop Scheduling Problem with No Intermediate Queues. *Opns. Res.* 20, 689-697.

[Wodecki et Bozejko, 2002] M. Wodecki, W. Bo_zejko, 2002. Solving the Flow-shop Problem by Parallel Simulated Annealing. *Springer Verlag*.

[Yu et *al.*, 2004] Yu, H. Hoogeveen and J. K. Lenstra, Minimizing *Makespan* in a Two-Machine Flow-shop with Delays and Unit-Time Operations is NP-Hard, *Journal of Scheduling* Vol 7, N°5 Septembre 2004, 333-348.

[Yu, 1996] W. Yu, 1996. The Two Machine Flow-shop Problem with Delays and the One-Machine Total Tardiness Problem. *East China University of Science and Technology.*

[Zijm et Nelissen, 1990] Zijm W.H.M et Nelissen E.H.L.B, 1990. Scheduling a Exible Machining Centre. *Eng Costs Prod Eco.*19, 249-258.