MCours.com


# 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.

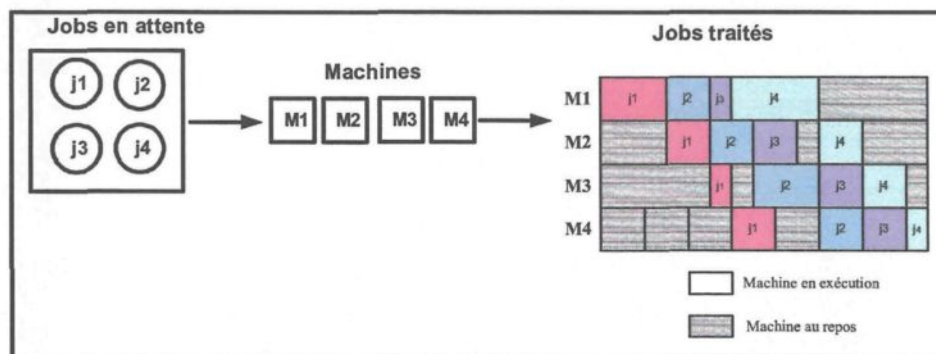


**Figure 3-1 : Modèle de flow-shop**

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 |
|---|---|---|---|---|
| $\tau_j$ | 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.

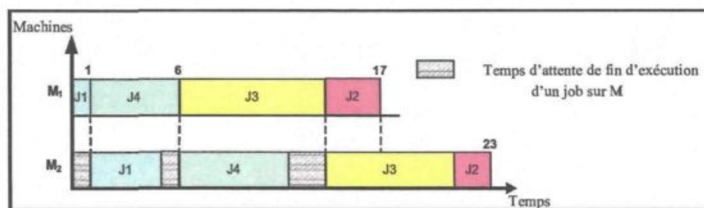


**Figure 3-2: Solution optimale de l'Exemple 3-1**

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 $\tau_j$ 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 |
| $\tau_j$ | 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***

MCours.com