# 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.

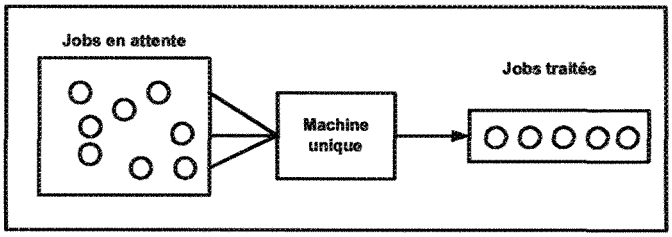


**Figure 2-1 : Modèle machine unique**

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.

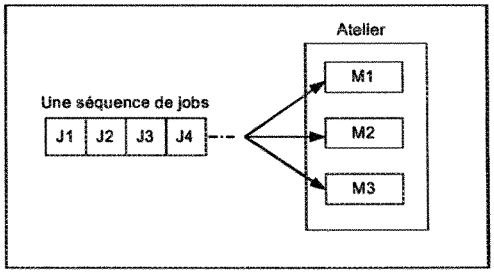


**Figure 2-2 : Ordonnancement d'une séquence de jobs sur des machines organisées en parallèle**

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*.

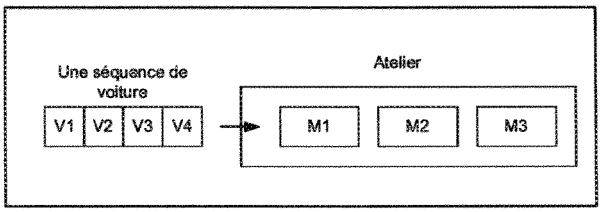


**Figure 2-3 : Une chaîne de montage des voitures**

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.

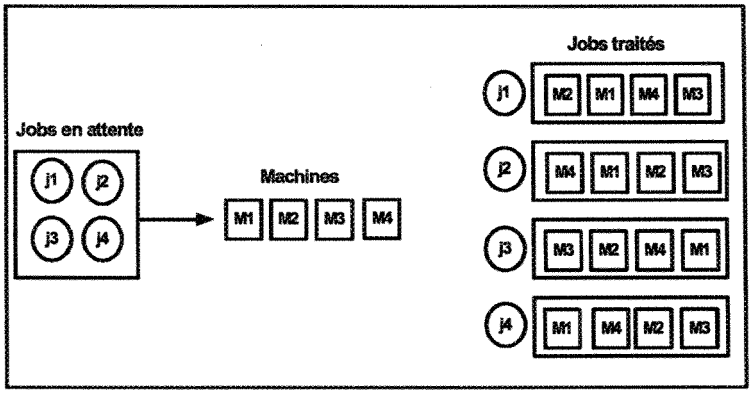


**Figure 2-4 : Modèle open – shop**

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.

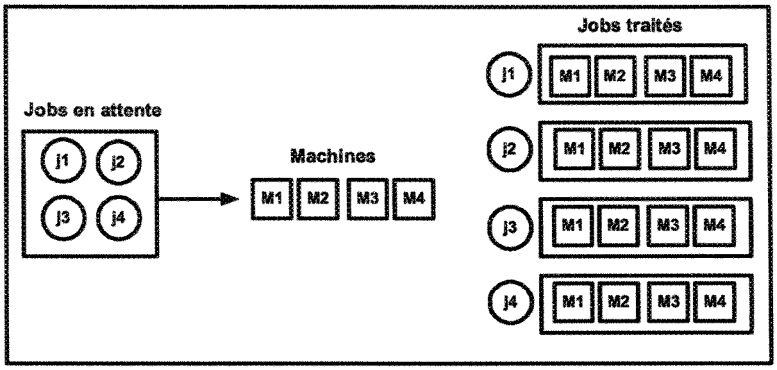


**Figure 2-5 : Modèle flow-shop**

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.

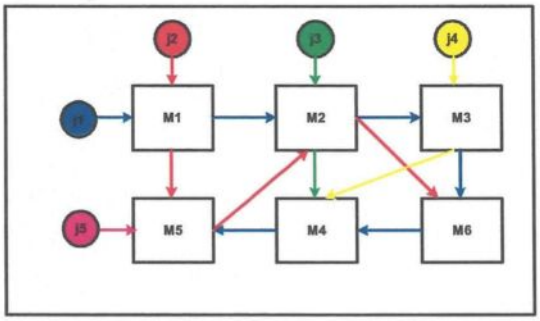


**Figure 2-6 : Modèle job – shop**

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].

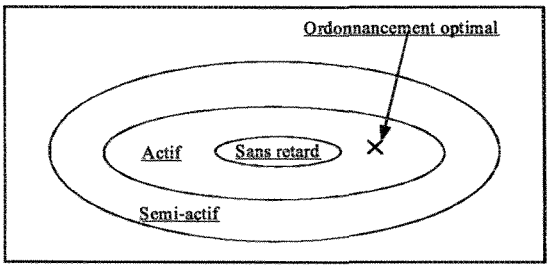


**Figure 2-7: Le diagramme de Venn représentant les classes d'ordonnancement**

### 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.

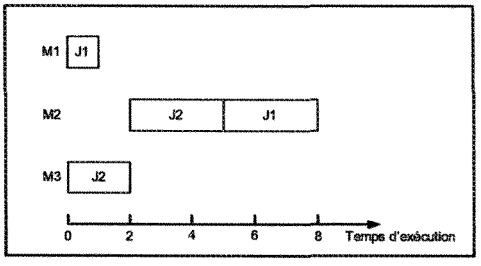


**Figure 2-8 : Un ordonnancement actif**

**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.

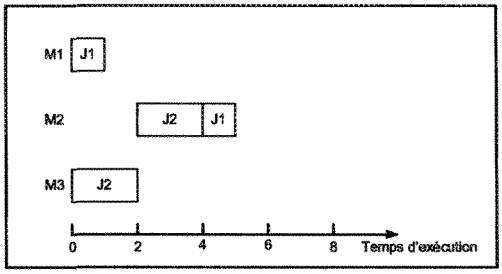


**Figure 2-9 : Un ordonnancement semi-actif**

**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.

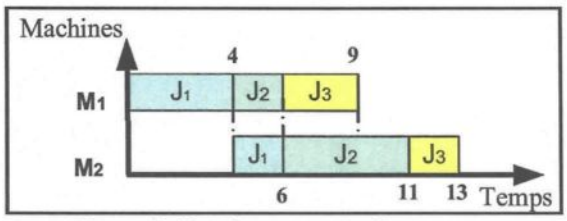


**Figure 2-10: Digramme de Gantt**

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 $\tau_{ijk}$, 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.

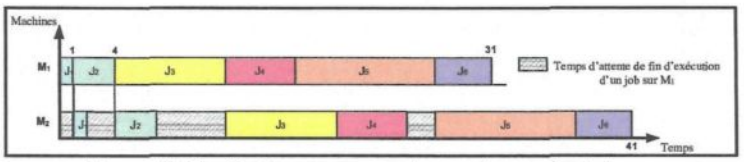


**Figure 2-11 : Ordonnancement sans temps de latence**

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 $\tau_{ijk}$ est réduit à $\tau_i$.

| | $J_1$ | $J_2$ | $J_3$ | $J_4$ | $J_5$ | $J_6$ |
|---|---|---|---|---|---|---|
| **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.

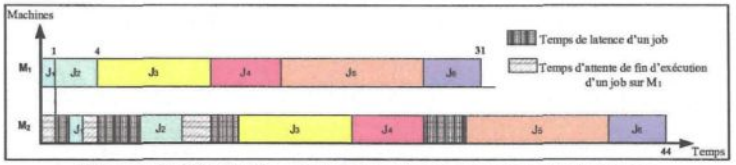


**Figure 2-12: Ordonnancement avec temps de latence**

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.*

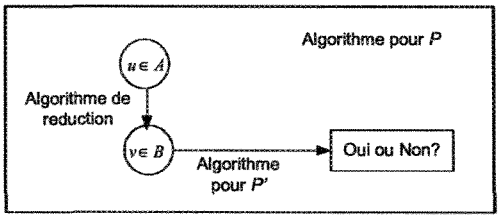


**Figure 2-13 : Réduction polynomiale d'un problème**

**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].

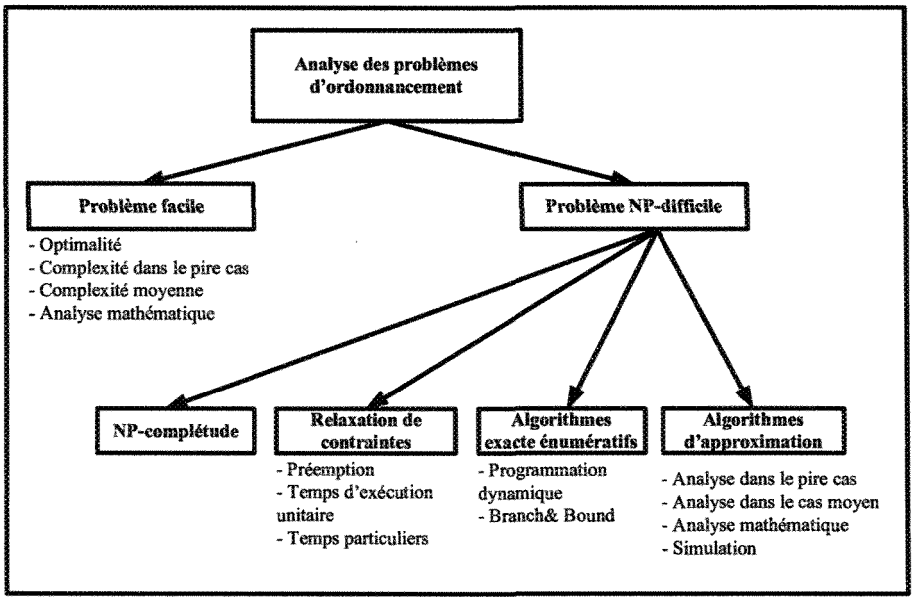


**Figure 2-14: approches de résolution des problèmes d'ordonnancement**

### 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.

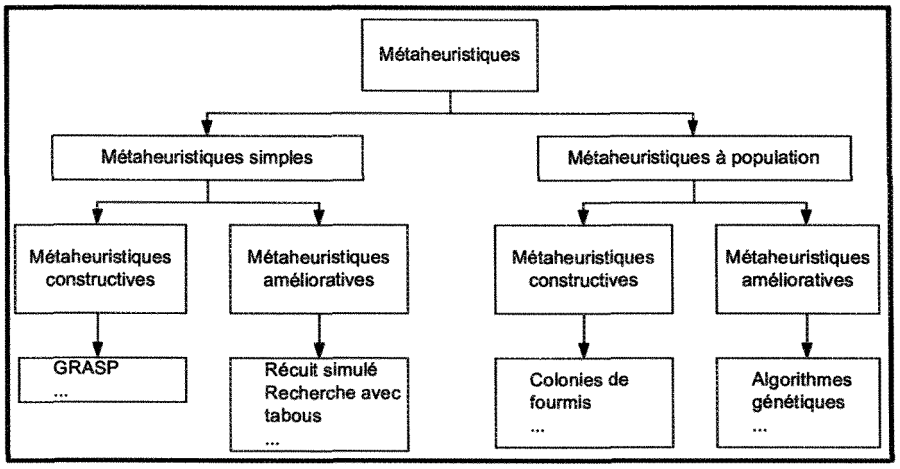


**Figure 2-15 : Classification des métaheuristiques**

- **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.