N° d'ordre : 01/2016-D/MT

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE « HOUARI BOUMEDIENE »
FACULTE DE MATHEMATIQUES

THESE
Présentée pour l'obtention du grade de DOCTEUR EN SCIENCES

EN : MATHEMATIQUES

Spécialité : Recherche Opérationnelle

Par : Amrouche Karim

Sujet

ORDONNANCEMENT SUR UNE LIGNE DE PRODUCTION AVEC RECIRCULATION

Soutenue Publiquement le 06/01/2016, devant le jury composé de :

M. BELBACHIR Hacène,	Professeur,	U.S.T.H.B.,	Président
M. BOUDHAR Mourad,	Professeur,	U.S.T.H.B.,	Directeur de thèse
M. YALAOUI Farouk,	Professeur,	UTT, France,	Examinateur
M. REBAINE Djamel,	Professeur,	UQAC, Canada,	Examinateur
M. CHERGUI M. El Amine,	Maitre de Conférences A,	U.S.T.H.B.,	Examinateur
M. AIT ZAI Hakim,	Maitre de Conférences A,	U.S.T.H.B.,	Examinateur

Remerciements

Je remercie, tout d'abord ***ALLAH****, le tout puissant, qui m'a donné le courage et la patience pour réaliser ce travail.*

Je tiens à exprimer ma gratitude et ma reconnaissance à Monsieur Mourad Boudhar, mon directeur de thèse, Professeur à l'U.S.T.H.B pour m'avoir proposé ce sujet, guidé dans mes recherches et dirigé cette thèse. Je le remercie également pour, sa patience, sa disponibilité, son soutien continu et pour son aide précieuse. Je lui suis redevable de m'avoir aidé à mieux comprendre les problèmes d'ordonnancement et la théorie de la complexité.

Mes sincères remerciements s'adressent aux personnes qui m'ont fait l'honneur de faire partie du Jury de ma thèse.
à Monsieur Hacène Belbachir, Professeur à l'USTHB, qui m'a fait l'honneur de présider ce Jury. à Messieurs Farouk Yalaoui (Professeur à l'Université de technologie de Troyes, France qui m'a toujours bien accueilli au sein de son laboratoire LOSI (Laboratoire des systèmes industriels)), Djamel Rebaine (Professeur à l'université de Chicoutimi, Canada),Mohamed El Amine Chergui (Maitre de conférence à l'USTHB) et Hakim Ait zai (Maitre de conférence à l'USTHB) pour avoir accepté de juger ce travail et d'être membre de ce Jury.

Je remercie mes collègues de l'USTHB : Wafaa Labbi, Haned Amina, et surtout Mohamed Bendraouche que je considère comme un frère et qui m'a beaucoup aidé à la rédaction de mes articles en Anglais.
Enfin, je remercie bien sûr ceux qui ont fait de moi ce que je suis, ceux grâce à eux tant d'années d'études ont été possibles, ceux envers qui j'ai une dette imprescriptible : ma Mère et mon père. Un grand merci à ma femme Sana, mon frère Said et ma sœur Amina et à mon petit bébé Abderrahim que dieu le protège.

Résumé

Le travail présenté dans cette thèse, traite du problème d'ordonnancement de tâches en mode non préemptif, sur un atelier de type réentrant flowshop. L'atelier étudié est constitué de deux machines et avec recirculation des tâches sur la première machine, et en présence d'un temps de latence exact séparant les deux opérations sur la première machine. L'objectif est de minimiser la date de fin de traitement de l'ensemble des n tâches (makespan). Ce problème est NP-difficile. Nous montrons que certains sous problèmes sont aussi NP-difficiles tandis que d'autres sont résolubles en des temps polynomiaux. En outre, nous proposons des heuristiques de type "algorithme de liste" et de type "recherche aléatoire", ainsi qu'une métaheuristique de type algorithme génétique suivies d'une étude expérimentale.

Mots clés : Ordonnancement, flow shop réentrant, temps de latence, makespan, complexité, heuristique, métaheuristique.

Abstract

In this thesis, we consider the problem of scheduling jobs non-preemptively, in which we seek to minimize the overall finish time (makespan). The workshop is of type reentrant flowshop. The studied problems consist of two machines and a set of n jobs. Each job goes through the first machine, then the second machine and finaly returns back to the first machine. The two operations on the first machine must be separated by an exact time lag. This problem is known to be NP-hard. We prove that some subproblems remain NP-hard, and others are well solvable. Furthermore, in the approximation front, we propose list algorithm type heuristics, random search based heuristics and a genetic algorithm metaheuristic with empirical experiments.

Keywords : Scheduling, reentrant flowshop, time lag, makespan, complexity, heuristic, metaheuristic.

Table des matières

Liste des tableaux

Table des figures

Introduction

Vu les exigences du marché qui se traduisent par une grande variété de produits manufacturés, une entreprise ne peut se maintenir que si ses produits sont compétitifs, pour cela une réduction des prix et des délais de réalisation de ses produits est indispensable. D'où, une bonne gestion de production est devenue une nécessité de plus en plus préoccupante pour les entreprises. Pour cette raison les problèmes d'ordonnancement d'atelier ont connus tant de succès, et en particulier les problèmes d'ordonnancement à cheminement unique connus aussi dans la littérature comme problèmes d'atelier de type flow shop.

Depuis que Johnson [27] proposa un algorithme polynomial pour résoudre le cas à deux machines, le problème d'ordonnancement dans les ateliers de type flow shop a retenu l'attention de plusieurs chercheurs. Plusieurs articles ont été publiés sur ce sujet, en considérant à chaque fois de nouveaux types de contraintes, parmi celles étudiées on peut citer par exemple les contraintes d'écart temporelle dites aussi temps de latence ou "time lag" en Anglais et les contraintes de recirculation.

Dans ce travail, on s'intéresse aux problèmes d'ordonnancement d'atelier de type flow-shop avec recirculation sur la première machine, en présence d'un temps de latence exact. Chaque tâche T_j doit se traiter d'abord sur la première machine M_1, ensuite sur la deuxième machine M_2 et puis revient après un temps de latence exact l_j à la première machine M_1 pour sa dernière opération. L'objectif est de minimiser la date de fin de traitement de l'ensemble des tâches. Ce problème est noté $F2|chain - reentrant, l_j|C_{max}$. Les problèmes d'ordonnancement avec temps de latence peuvent se rencontrer dans les blocs opératoires [9], ou dans les ateliers de fabrication de meubles où chaque composant doit subir une opération de peinture puis une opération de polissage et enfin une autre opération de peinture où un délai de séchage entre ses opérations est imposé [4].

Cette thèse est constituée de sept chapitres et elle est organisée comme suit.

Dans le premier chapitre nous présentons les principaux termes et notions utilisés tout au long de cette thèse. En premier lieu, nous donnons quelques notions de base de la théorie des graphes, ensuite nous donnons une introduction à la théorie de la complexité.

Dans le second chapitre, nous présentons quelques problèmes d'ordonnancement, leurs domaines, leur classification ainsi que leurs techniques de résolution.

Le chapitre 3 est consacré aux définitions, notations du problème général ainsi que les contraintes considérées. Nous exposons dans ce chapitre un état de l'art du problème traité et nous citons les principaux résultats connus dans la littérature.

Le quatrième chapitre est dédié à l'étude de la complexité du sous-problème (noté P_2) où les temps de latence sont identiques. Nous montrons que ce dernier est NP-difficile au sens fort, ensuite nous proposons plusieurs sous problèmes résolubles en des temps polynomiaux.

Le cinquième chapitre est consacré à la résolution du problème étudié au chapitre 4. On propose des heuristiques de type "algorithme de liste" avec des expérimentations numériques.

Dans le chapitre 6, nous considérons le sous-problème (noté P_3) où les durées de traitement sur la deuxième machine sont égaux au temps de latence. Nous montrons que ce dernier est NP-difficile au sens fort, ensuite nous proposons plusieurs sous problèmes résolubles en des temps polynomiaux. Afin de le résoudre, des heuristiques basées sur la recherche aléatoire et une métaheuristique ont été proposées au chapitre 7. Ces heuristiques ont été comparées expérimentalement sur des instances générées aléatoirement.

Enfin, dans la conclusion nous récapitulons les principaux résultats présentés dans cette thèse et nous proposons des perspectives.

Chapitre 1

Notions de base

Dans ce chapitre, nous abordons les notions principales utilisées tout au long de la thèse. D'abord nous présentons quelques notions de base de la théorie des graphes. Ensuite, nous introduisons les notions de problèmes, d'algorithmes et leurs complexités ainsi que la complexité des problèmes.

1.1 Notions de graphes

Un graphe G simple (sans boucles) est défini par la donnée d'un ensemble fini V, appelé ensemble de sommets et d'un ensemble E de paires de sommets distincts et non ordonnés de V, dites arêtes. On note ce graphe par $G = (V, E)$.
Soit $G = (V, E)$ un graphe, si $e = \{x, y\}$ est une arête de G, e est représentée par un segment de droite. On dira dans ce cas que les deux sommets x et y sont reliés ou adjacents. L'arête e a pour extrémités x et y.
Une arête e est dite incidente à un sommet x si ce sommet est une extrémité de cette arête.
Un sommet y est dit un voisin de x si x et y sont adjacents. Si x est un sommet de G, le voisinage de x est l'ensemble $V(x)$ formé par tous les voisins de x. Un sommet de G est dit isolé si son voisinage est un ensemble vide. Le degré $d(x)$ de x est la cardinalité du voisinage de x : $d(x) = |V(x)|$.

Exemple 1.1. $V = \{v_1, v_2, ..., v_6\}$, $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_5\}, \{v_3, v_4\}, \{v_4, v_5\}\}$
le sommet v_6 est un sommet isolé

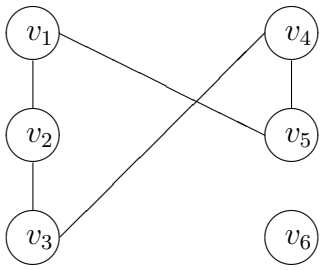


FIGURE 1.1 – Graphe $G = (V, E)$ de l'exemple 1.1

Le complémentaire de G est le graphe noté $\overline{G} = (V, \overline{E})$ tel que $\{x, y\} \in \overline{E}$ si et seulement si $\{x, y\} \notin E$.

Le sous-graphe de G induit par un sous-ensemble $V' \subseteq V$ est le graphe $G' = (V', E')$ tel que E' est l'ensemble des arêtes de G ayant les deux extrémités dans V'.

Une chaîne de longueur l, est une séquence de sommets $x_0, x_1, ...x_l$ tels que les sommets x_{i-1} et x_i sont reliés par une arête, pour tout i ($i = \overline{1, l}$), x_0 et x_l sont appelés les extrémités de la chaîne. Un cycle est une chaîne fermée (les extrémités coïncident) dont toutes les arêtes sont distinctes.

On peut représenter un graphe $G = (V, E)$ par sa matrice d'incidence sommet-arête A, où le coefficient a_{ij} en ligne i et en colonne j vaut :

$$a_{ij} = \begin{cases} 1, & \text{si l'arête } e_j \text{ est incidente au sommet } v_i; \\ 0, & \text{sinon.} \end{cases}$$

Un couplage dans un graphe $G = (V, E)$ est un sous-ensemble d'arêtes C tel que deux arêtes de C ne soient pas incidentes à un même sommet. On peut aussi dire qu'un couplage est un sous-ensemble d'arêtes deux à deux non-adjacentes.
Un couplage maximum est un couplage contenant le plus grand nombre d'arêtes possibles.

1.2 Problèmes, algorithmes et complexité

Un problème est une question générale à laquelle on veut apporter une réponse (ou une solution), possédant généralement des paramètres. Il est formalisé par (i) une description

de tous ses paramètres formels dont les valeurs ne sont pas spécifiées et (ii) une indication précisant les propriétés qui doivent être vérifiées par la réponse (ou la solution).

Une instance d'un problème est obtenue en attribuant une valeur à chacun de ses paramètres. Résoudre un problème c'est donner une solution à toutes ses instances. Cette résolution s'effectue par la définition d'un algorithme, c'est-à-dire une suite d'opérations élémentaires séquentielles à effectuer.

Parmi les problèmes qui nous intéressent, il existe généralement deux grands types :

- **Les problèmes de décisions :** ce sont les problèmes posant une question dont la réponse est "oui" ou "non". Ils sont formalisés par deux composantes : une instance générique précisant les paramètres formels du problème et une question.

  Exemple : **Partition**
  Instance : n entiers positifs $a_1, a_2, ..., a_n$.
  Question : existe-t-il un sous-ensemble $J \subseteq I = \{1, 2, ..., n\}$ tel que $\sum_{i \in J} a_i = \sum_{i \in I \setminus J} a_i$?
- **Les problèmes d'optimisation :** Ce sont les problèmes cherchant à optimiser (maximiser ou minimiser) une certaine fonction f définie de $R^n \to R$ dite fonction objectif, sur un sous-ensemble non-vide X de R^n dit ensemble de contraintes.

  Exemple : **"Le problème du voyageur de commerce"**
  Etant donné un ensemble de n villes et pour chaque paire de villes $\{v_i, v_j\}$, la distance de la ville v_i à la ville v_j est donnée. Trouver un parcours fermé passant par toutes les villes (une et une seule fois) minimisant la distance totale parcourue.

1.2.1 Complexité d'un algorithme, efficacité et taille d'entrée

Un algorithme qui résout un problème prend en entrée une instance I, dite une entrée du problème. D'autre part pour implémenter cet algorithme sur un ordinateur, cette instance est codée sous forme de chaînes binaires de 0 et 1 (codage binaire). La longueur de cette chaîne définit la taille de l'entrée notée $|I|$.
La complexité d'un algorithme est une fonction $T(|I|)$, qui est égale au nombre maximum d'opérations élémentaires requises par l'exécution de l'algorithme sur une entrée de taille $|I|$ (on considère donc le pire des cas possibles).

L'efficacité d'un algorithme pour un problème se caractérise par sa complexité : on dira qu'un algorithme est polynomial ou efficace si sa complexité est en $O(|I|^k)$ pour une certaine constante k ($|I|$ étant la taille d'entrée). En revanche, un algorithme dont la complexité ne peut pas être bornée polynomialement est qualifiée d'exponentiel, par exemple des algorithmes de complexité en $O(2^{|I|})$ et en $O(|I|!)$.

La complexité d'un algorithme va dépendre non seulement de l'algorithme lui même mais aussi de la manière dont est codée l'entrée. Toutefois, d'après la définition précédente un algorithme efficace pour un codage raisonnable de l'entrée (le restera pour tout codage qui lui est polynomialement équivalent (c'est à dire que la taille par rapport au premier codage est majorée par une fonction polynomiale de celle du second, et inversement).
Ainsi, dans la théorie de la complexité on peut remplacer la taille d'entrée (comme elle a été définie ci-dessus par les chaînes de 0 et 1) par un paramètre "intuitif" qui, souvent et par abus de langage représentera **la taille d'entrée**. Voici quelques exemples :

- si l'entrée est un graphe $G = (V, E)$: tel que $|V| = n$, $|E| = m$ la taille d'entrée peut être soit n, soit m ou $n + m$.
- si l'entrée est une matrice mxn : la taille d'entrée est le maximum, la somme ou le produit de m et n.
- si l'entrée est un polynôme : la taille d'entrée est son degré ou le nombre de coefficients.

Lors du calcul de la complexité d'un algorithme, on cherche un ordre de grandeur plutôt que le nombre exact. On utilise pour ce faire, la notation de Landau O. Si la taille d'entrée est n, un algorithme dont la complexité est $T(n) = 2n^3 + 5n^2 + 4$ est en $O(n^3)$.

A titre d'exemple, la complexité de l'algorithme classique du produit de deux matrices carrées d'ordre n se calcule comme suit : il existe n^2 éléments à calculer. Or chacun de ces éléments nécessite n multiplications et $n - 1$ additions, ce qui fait $2n - 1$ opérations élémentaires. Par conséquent, la complexité de cet algorithme est en $O(n^3)$.

1.2.2 Classes de problèmes

La classification usuelle des problèmes se fait en considérant leurs complexités. Celle-ci correspond à la complexité de l'algorithme le plus efficace pour résoudre ce problème. A cause des difficultés rencontrées, la théorie de la complexité ne traite fondamentalement que les problèmes de décision, car leur formalisme Oui-Non, Vrai-Faux a permis de les

étudier avec les outils de la logique mathématique. Cependant on étend la notion de complexité aux problèmes d'optimisation. En effet, il est facile de transformer un problème d'optimisation en un problème de décision. Par exemple : chercher à optimiser une certaine fonction f revient à traiter un problème de décision qui consiste à comparer f à un certain k, puis en générant un certain nombre de valeurs de k, on peut déterminer la valeur optimale (par exemple en utilisant l'algorithme de dichotomie).

Les trois classes de complexité de problèmes les plus courantes sont P, NP et NP-Complet :

La classe P (Polynomial)

Cette classe se compose de tous les problèmes de décision qui sont résolus par des algorithmes polynomiaux par rapport à la taille d'entrée.

La classe NP (Nondeterministic Polynomial)

Elle regroupe tous les problèmes de décision qui peuvent être résolus en temps polynomial par des algorithmes non déterministes. Un algorithme est dit non déterministe s'il comporte des instructions de choix. Pour ces algorithmes, si à chaque instruction, le bon choix est effectué, le temps de calcul est polynomial. Si au contraire tous les choix sont énumérés l'algorithme devient déterministe et son temps de calcul devient exponentiel. De façon informelle, un problème de décision appartient à NP si on peut vérifier en un temps polynomial, qu'une solution proposée (ou devinée) permet d'affirmer que la réponse est "oui".

Remarque 1.1. *Les algorithmes polynomiaux sont évidemment des cas particuliers des algorithmes non déterministes. Alors tout problème de décision de la classe P, appartient également à la classe NP, d'où* $P \subseteq NP$.

1.2.3 Classe des problèmes NP-Complets

Parmi les problèmes de la classe NP, il existe une large classe de problèmes : les problèmes NP-Complets, qui sont équivalents entre eux quant à l'existence d'un algorithme polynomial pour les résoudre. C'est-à-dire, s'il existe un algorithme polynomial pour résoudre un seul de ces problèmes, alors il en existe un pour chaque problème de la classe NP.

Les problèmes NP-Complets sont les problèmes les plus difficiles de la classe NP. Afin de décrire cette classe d'équivalence, définissons tout d'abord la transformation (réduction) polynomiale entre deux problèmes.

Définition 1.1. *Soient D_1 et D_2 deux problèmes de décision. Une transformation polynomiale de D_1 vers D_2 peut être vue comme une fonction f de l'ensemble des instances de D_1 vers l'ensemble des instances de D_2 qui satisfait les deux conditions suivantes :*

1. *f est calculable par un algorithme polynomial*
2. *Pour toute instance I de D_1, I a pour réponse "oui" (pour D_1) si et seulement si $f(I)$ a pour réponse "oui" (pour D_2).*

Si une telle transformation existe, on dira que D_1 se transforme (réduit) polynomialement en D_2, et on écrit $D_1 \propto D_2$.

D'une manière équivalente, on dit que D_1 se transforme (réduit) polynomialement en D_2, s'il existe un algorithme de résolution de D_1, qui fait appel à un algorithme de résolution de D_2 et qui est polynomial lorsque la résolution de D_2 est comptabilisée comme une opération élémentaire.

On en déduit que : si $D_1 \propto D_2$ et s'il existe un algorithme polynomial pour résoudre D_2, alors il existe un algorithme polynomial pour résoudre D_1.

Définition 1.2. *Un problème de décision D est NP-Complet si :*

1. *D est dans NP*
2. *Tout problème D' de NP se réduit polynomialement à D*

La classe des problèmes de décision NP peut être partitionnée en 3 sous classes : les problèmes de la classe P, les problèmes NP-Complets et les autres, voir figure 1.2.

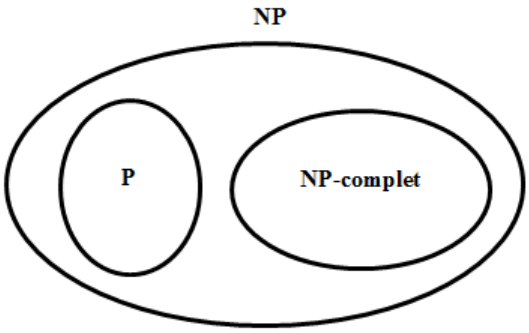


FIGURE 1.2 – Géographie de la classe NP

Le problème SAT :
Le premier problème qui a été prouvé NP-Complet par S.A. Cook en 1970 [24], est le problème de Satisfiabilité (SAT), qui est un problème de logique, défini comme suit :
Instance : Un ensemble U de variables booléennes et une collection C de clauses sur U.
Question : Existe-t-il une affectation en "vrai" et "faux" des variables telle que toutes les clauses de C prennent la valeur "vrai" ?

Si la réponse est "oui" on dit que C est satisfiable. Exemple : Pour l'instance $I : U = \{u_1, u_2\}, C = \{\{u_1, \overline{u_2}\}, \{\overline{u_1}, u_2\}\}$, la réponse est oui, il suffit de prendre valeur "vrai" pour u_1 et pour u_2. Par contre pour l'instance $I' : U = \{u_1, u_2\}, C' = \{\{u_1, u_2\}, \{u_1, \overline{u_2}\}, \{\overline{u_1}\}\}$, la réponse est non.

1.3 Prouver la NP-complétude d'un problème

La démonstration d'appartenance d'un problème de décision à la classe des problèmes NP-Complets est très importante car, une fois cette démonstration faite, on peut considérer comme peu vraisemblable l'existence d'un algorithme polynomial pour résoudre ce problème.

Montrer qu'un problème de décision D est NP-Complet consiste en les trois étapes suivantes :

- Montrer que D appartient à NP
- Choisir D', un problème NP-Complet

– Construire une transformation polynomiale f de D' vers D

1.3.1 Quelques problèmes NP-Complets de base

Nous citons six problèmes NP-Complets dont la NP-complétude n'a pu être démontrée que grâce au problème de Satisfiabilité. Ces problèmes sont considérés comme des problèmes de base et sont exposés dans le livre de Garey et Johnson [24].

3-SAT(3-satisfiabilité) : Etant donné un ensemble U de variables booléennes et une collection C de m clauses sur U, contenant chacune 3 variables. Existe-t-il une affectation en "vrai" et "faux" des variables telle que toutes les clauses de C prennent la valeur "vrai" ?

3-DM(3-Dimensional Matching) : Etant donné un ensemble M de triplets, $M \subseteq X \times Y \times Z$ où X, Y, Z sont des ensembles mutuellement disjoints et de même cardinalité q. Existe-t-il un sous ensemble $M' \subseteq M$ tel que $|M'| = q$ et les triplets de M' sont deux à deux disjoints ?

Recouvrement : Etant donné un graphe $G = (V, E)$ et un entier k. Existe-t-il $X \subseteq V$ tel que $|X| \leq k$ et toute arête de G a au moins une de ses extrémités dans X ?

Clique : Etant donné un graphe $G = (V, E)$ et un entier k. Existe-t-il $X \subseteq V$ tel que $|X| \geq k$ et tous les sommets de X sont deux à deux adjacents ?

Cycle Hamiltonien : Etant donné un graphe $G = (V, E)$. Existe-t-il un cycle passant une et une seule fois par tous les sommets de G ?

Partition : Etant donné n entiers positifs $a_1, a_2, ..., a_n$. Existe-t-il un sous ensemble $J \subseteq I = \{1, 2, ..., n\}$ tel que $\sum_{i \in J} a_i = \sum_{i \in I \setminus J} a_i$?

La figure 1.3 montre les différentes réductions polynomiales utilisées pour prouver la NP-Complétude de ces six problèmes.

1.3.2 Conjecture fondamentale P $\neq$ NP

On a vu dans la remarque précédente que $P \subseteq NP$, mais on ne sait pas si P est égale ou non à NP. S'il s'avérait que $P = NP$, alors on pourrait résoudre tous les problèmes de la

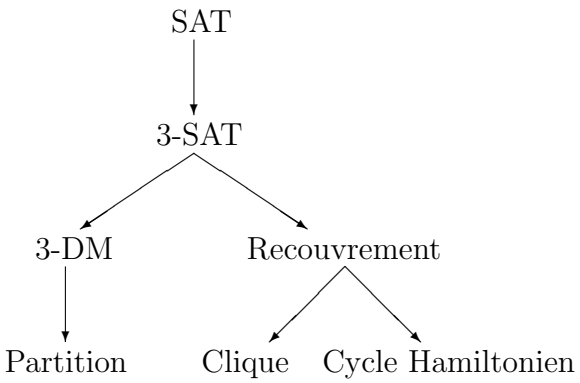


FIGURE 1.3 – Enchaînement des preuves de la NP-complétude.

classe NP en un temps polynomial. Or, les problèmes NP-Complets sont très fréquents et pour aucun d'entre eux on n'a pas réussi à trouver un algorithme polynomial le résolvant. La communauté scientifique pense que $P \neq NP$, mais ce n'est pas prouvé et que cette conjecture reste un problème ouvert depuis 1971.

1.3.3 NP-complétude au sens fort

On rappelle qu'un algorithme résolvant un problème de décision (D) est dit polynomial si sa complexité est en $O(|I|^k)$, où $|I|$ représente la taille d'entrée, pour un codage raisonnable, ou d'une manière équivalente que sa complexité est bornée par un polynôme en $|I|$. Un algorithme est dit pseudo-polynomial si sa complexité est bornée par une fonction polynomiale de deux variables $|I|$ et max(I), où max(I) est la valeur du plus grand nombre dans I. Par exemple, en ordonnancement, un algorithme est dit polynomial si sa complexité est bornée par un polynôme du nombre de tâches, tandis que si sa complexité est bornée par un polynôme de deux variables : le nombre de tâches et la durée de traitement maximale, il sera dit pseudo-polynomial.

Quand un problème est NP-complet, il n'est pas raisonnable d'espérer de construire un algorithme polynomial le résolvant, mais on peut trouver dans certains cas des algorithmes pseudo-polynomiaux, comme celui de Partition. Dans d'autres cas, il sera difficile de trouver un algorithme pseudo-polynomial qu'un algorithme polynomial, on parlera alors des problèmes NP-Complets au sens fort.

Définition 1.3. *Un problème de décision D est dit NP-Complet au sens fort si :*

1. *D est NP-Complet*
2. *L'existence d'un algorithme pseudo-polynomial pour le résoudre entraîne l'existence d'un algorithme polynomial.*

Exemple 1.2. *Le problème suivant est NP-Complet au sens fort [24].*

3-Partition : *Etant donné un ensemble A à $3m$ éléments, une borne entière B et une taille a_i pour chaque élément $a \in A$, telles que $\frac{B}{4} < a_i < \frac{B}{2}$ et $\sum_{a \in A} a_i = mB$. Existe-t-il m sous ensembles disjoints $S_1, ..., S_m$ tels que pour tout j $(j = 1, 2, ..., m) : \sum_{a \in S_j} a_i = B$?*

Un problème NP-Complet dont les paramètres formels sont bornés est évidemment NP-Complet au sens fort.

1.3.4 Problèmes NP-difficiles

Définition 1.4. *Un Problème d'Optimisation Combinatoire (POC en abrégé) est défini par la donnée d'un ensemble fini dénombrable D de solutions réalisables et une fonction objectif*
$f : D \to \mathbb{R}$ où on cherche un élément $x_0 \in D$ tel que $f(x_0) = \min_{x \in D} f(x)$.

Ce problème s'appelle un problème de minimisation et est noté $\begin{cases} Minf(x) \\ x \in D \end{cases}$
Le problème consistant à chercher un élément maximum au lieu d'un élément minimum est de même nature puisque $\max_{x \in D} f(x) = -\min_{x \in D}(-f(x))$.

D'abord, notons qu'à chaque problème d'optimisation combinatoire on peut lui associer un problème de décision, par exemple le problème de décision associé au problème $\begin{cases} Minf(x) \\ x \in D \end{cases}$ est le suivant : étant donné un ensemble fini D, une fonction objectif f définie sur D et une borne K , existe t-il $x \in D$ tel que $f(x) \leq K$?

Si l'on dispose d'un algorithme polynomial pour un POC, alors on dispose d'un algorithme polynomial pour le problème de décision associé, et vice versa.

La difficulté d'un problème d'optimisation combinatoire réside dans le fait que la solution doit être cherchée dans un ensemble de très grande cardinalité.

Définition 1.5. *Un problème d'optimisation combinatoire "POC" est dit NP-difficile, si un problème de décision NP-complet se réduit (au sens de Turing) à "POC".*

Par exemple, le problème d'optimisation combinatoire associé au problème de décision Deux Processeurs, est NP-difficile.

Deux Processeurs : Etant donné n tâches $T_1, T_2, ..., T_n$ indépendantes et non morcelables telles que chaque tâche T_i possède une durée de traitement p_i, et un nombre k . Existe-t-il un ordonnancement de ces tâches sur deux processeurs de durée inférieure ou égale à k ?

Remarque 1.2.

Deux problèmes d'optimisation seront dits polynomialement équivalents si le problème de décision associé à l'un peut se réduire polynomialement à celui de l'autre et vice versa.

Chapitre 2

Ordonnancement, définitions et techniques de résolution

Dans ce chapitre, nous abordons les notions principales des problèmes d'ordonnancement où nous donnons leurs notations ainsi que leurs classifications, ensuite nous donnons un aperçu des méthodes de résolution de ces problèmes, on commence par présenter quelques problèmes d'optimisation combinatoire ensuite nous donnons quelques méthodes de résolution exactes et approchées.

2.1 Problèmes d'ordonnancement

La définition suivante est due à Carlier et Chrétienne [17] :

Définition 2.1. *Ordonnancer c'est programmer l'exécution d'une réalisation en attribuant des ressources aux tâches et en fixant leurs dates d'exécution*

Les problèmes d'ordonnancement apparaissent dans de nombreux domaines : informatique (tâches : jobs ; ressources : processeurs), chantiers (suivi de projets), l'industrie (problèmes d'atelier, gestion de production), administration, écoles et universités (personnel, emplois du temps).

Les éléments essentiels d'un problème d'ordonnancement sont : les tâches, les ressources, les contraintes et les objectifs.

2.1.1 Tâches

Une tâche est définie par un ensemble d'opérations qui doivent être exécutées. Les tâches peuvent représenter des pièces, des programmes ou des activités. Une tâche notée J_i peut être définie par ses caractéristiques temporelles :

- p_i : la durée de traitement de la tâche J_i.
- r_i : la date de disponibilité de la tâche J_i.
- d_i : la date échue de la tâche J_i.
- etc.

2.1.2 Ressources

Une ressource est un moyen matériel (machine) ou humain intervenant dans la réalisation d'une tâche.

Il existe deux principaux types de ressources :

les ressources renouvelables

Elles sont disponibles en quantité constante tout au long de l'exécution des tâches. Les ressources renouvelables usuelles sont les machines, les processeurs, le personnel, etc.

Les ressources consommables

Une ressource est consommable si, après avoir été allouée à une tâche, elle n'est plus disponible pour les tâches restantes, c'est le cas des matières premières et de l'énergie par exemple.

2.1.3 Contraintes

Les contraintes représentent les limitations imposées par l'environnement. Dans la plupart des problèmes d'ordonnancement les tâches à exécuter sont soumises à des contraintes

qu'il faut satisfaire au moment de la recherche d'une solution optimale, on distingue trois principales types de contraintes :

- **Les contraintes potentielles :** Ce sont les contraintes de localisation temporelle par exemple " la tâche i doit précéder la tâche j " ou "la tâche i doit être achevée avant telle date".

- **Les contraintes disjonctives :** Lorsque deux tâches ne peuvent pas être exécutées en même temps, on dit qu'il y a une contrainte disjonctive que doivent satisfaire ces deux tâches.

- **Les contraintes cumulatives :** Elles concernent l'évolution dans le temps du volume total des moyens humains ou matériels consacrés à l'exécution des tâches.

2.1.4 Ordonnancement

Un ordonnancement constitue une solution au problème d'ordonnancement. Il décrit l'exécution des tâches et l'allocation des ressources au cours du temps et vise à satisfaire un ou plusieurs objectifs. Les variables intervenant le plus souvent dans un ordonnancement sont :

- t_i : la date de début de la tâche J_i.
- C_i : la date de fin de traitement de la tâche J_i.
- $l_i = C_i - d_i$: la tardivité de la tâche J_i.
- $t_i = \max\{l_i, 0\}$: le retard de la tâche J_i.
- U_i : l'indicateur de retard de la tâche J_i, $U_i = \begin{cases} 1 & \text{si } C_i > d_i \\ 0 & \text{sinon} \end{cases}$
- $F_i = C_i - r_i$: le temps de séjour (flow time).

Selon les problèmes, les tâches peuvent être exécutées par morceaux, le mode de traitement est dit préemptif. Il est dit non préemptif si celles-ci doivent être exécutées sans interruption.

2.1.5 Critères

Dans un problème d'ordonnancement on cherche un ordonnancement minimisant une fonction objectif, appelé critère.

Les critères couramment étudiés dans la littérature sont : $X_{max}= \max_{1\leq i\leq n}\{X_i\}$, $\sum X_i= \sum_{i=1}^{n} X_i$, $\sum w_iX_i= \sum_{i=1}^{n} w_iX_i$ avec $X \in \{C, F, L, U\}$.

- C_{max} : date de fin de traitement de l'ensemble des tâches (makespan).
- $\sum C_i$: la somme des dates de fin de traitement.
- $\sum w_iC_i$: la somme pondérée des dates de fin de traitement.
- T_{max} : le grand retard.
- L_{max} : la grande tardiveté.
- F_{max} : le grand temps de séjour.
- $\sum U_i$: le nombre de tâches en retard.
- $\sum w_iU_i$: la somme pondérée du nombre de tâches en retard.
- d'autres critères peuvent être définis.

Soit C_j et C'_j les dates de fin d'exécution d'une tâche T_j dans deux ordonnancements S et S' différents de mêmes tailles de séquence.

Définition 2.2. *Un critère de performance f est dit régulier, s'il vérifie la condition :* $C_1 \leq C'_1, C_2 \leq C'_2, ..., C_n \leq C'_n \Rightarrow f(C_1, C_2, ..., C_n) \leq f(C'_1, C'_2, ..., C'_n)$.

2.1.6 Diagramme de Gantt

Les ordonnancements sont généralement représentés par des diagrammes dits de Gantt, voir l'exemple de la figure 2.1. Ceux ci indiquent, selon une échelle temporelle donnée, l'occupation des machines par les différentes tâches, les temps morts (parties hachurées) dus, par exemple aux éventuelles indisponibilités des tâches ou contraintes potentielles.

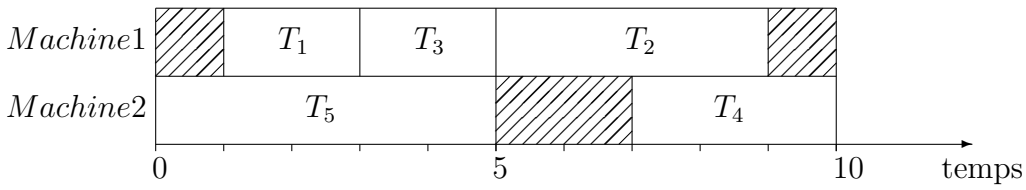


FIGURE 2.1 – Exemple de diagramme de Gantt

2.1.7 Ordonnancement d'atelier

Dans les problèmes d'ordonnancement d'atelier, les ressources sont les machines, ne pouvant réaliser qu'une tâche à la fois, celle-ci pouvant être composée d'une ou plusieurs opérations. Parmi les problèmes fréquemment rencontrés dans la littérature, on trouve les problèmes à une machine unique et les problèmes multi-machines. Dans le second cas on distingue les problèmes à machines parallèles et ceux à machines dédiées. Dans le premier cas, chaque tâche est réduite à une seule opération.

Problème à une machine unique

Chaque tâche est composée d'une seule opération. Toutes les tâches à réaliser sont alors exécutées par une seule machine.

Problème à machines parallèles

Ce problème se subdivise en trois classes selon les vitesses d'exécution des machines :

- **Machines identiques :** Toutes les machines ont la même vitesse de traitement quelque soit la tâche.
- **Machines uniformes :** Chaque machine a sa propre vitesse qui ne dépend pas des tâches à exécuter.
- **Machines générales :** Contrairement au cas précédent, la vitesse de traitement des tâches dépend de la machine et de la tâche à exécuter.

Le problème à machines dédiées (spécialisées)

Elles sont spécialisées à l'exécution de certaines opérations. Dans cette catégorie, chaque tâche est constituée de plusieurs opérations. En fonction du mode de passage des opérations sur les différentes machines, trois ateliers spécialisés classiques sont différenciés :

- **Le problème à cheminement unique (Flow shop) :** Les tâches doivent être traitées par toutes les machines, et l'ordre de passage des opérations sur ces machines est le même pour toutes les tâches.

- **Le problème à cheminements multiples (Job shop) :** Chaque tâche est traitée par un sous ensemble de machines et l'ordre de passage des opérations de chaque tâche est propre à celle-ci et est spécifié à l'avance.

- **Le problème à cheminement quelconque (Open shop) :** Les tâches doivent être traitées par toutes les machines, mais l'ordre de passage des opérations sur ces machines est arbitraire.

2.2 Classification et notations

Pour des raisons de clarté et de compréhension, il est important d'adopter un descripteur qui soit basé sur l'existant et homogénéisant les notations actuelles.

La notation $(\alpha|\beta|\gamma)$ introduite par Graham et al. [25] est couramment utilisée pour distinguer les différents problèmes d'ordonnancement entre eux et les classifier.

Champ α

Renseigne sur les informations relatives aux ressources, à savoir le nombre de machines, type de machines, et type d'atelier. Il est composé de deux sous champs et désigné par $\alpha_1\alpha_2$:

- $\alpha_1 \in \{1, P, Q, R, F, O, J\}$
  1 : machine unique.

P : machines parallèles identiques.
Q : machines parallèles uniformes.
R : machines parallèles générales.
F : flow shop.
O : open shop.
J : job shop.

– $\alpha_2 \in\{\phi$, m $\}$ (pour $\alpha_1 \neq 1$)
Lorsque $\alpha_2 = \phi$, le nombre de machines est variable, tandis que lorsque $\alpha_2 = m$, le nombre de machines est fixé à m.

Champ β

Permet d'obtenir des informations sur les contraintes prises en compte sur les tâches qui doivent être ordonnancées. Il est composé de 5 sous champs $\beta_1, \beta_2, \beta_3, \beta_4, \beta_5$:

– $\beta_1 \in \{\phi, pmtn\}$
$\beta_1 = \phi$: la préemption des tâches n'est pas autorisée.
$\beta_1 =$ pmtn : la préemption est autorisé.

– $\beta_2 \in \{\phi$, prec, tree, chain$\}$
$\beta_2 = \phi$: pas de contraintes de précédence.
$\beta_2 =$ prec : les contraintes de précédence sont quelconques.
$\beta_2 =$ tree : les contraintes de précédence sont données sous forme d'arborescence.
$\beta_2 =$ chain : les contraintes de précédence sont données sous forme de chaînes.

– $\beta_3 \in\{\phi, r_i\}$
$\beta_3 = \phi$: toutes les dates de disponibilité sont nulles.
$\beta_3 = r_i$: il existe des dates de disponibilité pour les tâches.

– $\beta_4 \in\{\phi, p_i = p, \underline{p} \leq p_i \leq \overline{p}\}$
$\beta_4 = \phi$: les temps de traitement des tâches sont quelconques.
$\beta_4 = p_i = p$: les temps de traitement des tâches sont égaux à p.
$\beta_4 = \underline{p} \leq p_i \leq \overline{p}$: le temps de traitement de chaque tâche est compris entre $\underline{p}$ et $\overline{p}$.

– $\beta_5 \in\{\phi, d_i, \tilde{d}_i\}$
$\beta_5 = \phi$: il n'y a pas des dates échues pour les tâches, ou il existe des dates échues pour les tâches dans le cas ou le critère d'optimisation est en fonction de ces derniers.

$\beta_5 = d_i$: il existe des dates échues pour les tâches.
$\beta_5 = \tilde{d}_i$: il existe une date limite pour chaque tâche.

Champ γ

Le troisième champ γ décrit le critère d'optimalité. On cherche à minimiser γ avec $\gamma \in \{ C_{max}, \sum C_i, \sum w_i C_i, T_{max}, L_{max}, F_{max}, \sum U_i, \sum w_i U_i ... \}$.

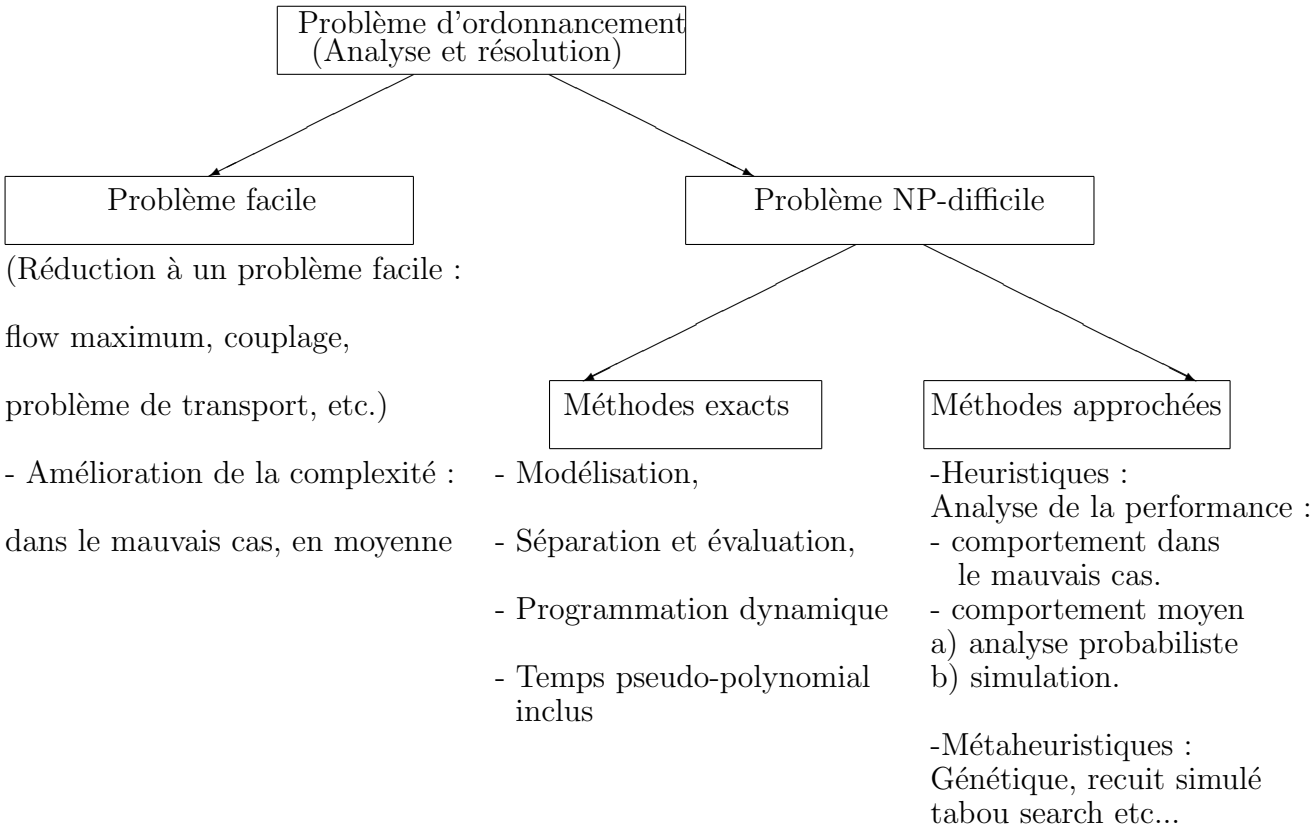
2.2.1 Exemples

Exemple 2.1. *Le problème* $1|r_i|C_{max}$ *est le problème d'ordonnancement de tâches indépendantes sur une seule machine avec des durées de traitement et des dates de disponibilité arbitraires, l'objectif est la minimisation du makespan.*

Exemple 2.2. *Le problème* $F2||C_{max}$ *est le problème d'ordonnancement de tâches indépendantes sur un atelier du type flow shop à deux machines avec des durées de traitement arbitraires, l'objectif est la minimisation du makespan.*

2.3 Analyse des problèmes d'ordonnancement

Les problèmes de l'ordonnancement déterministe forment donc une classe de problèmes d'optimisation combinatoire qui peuvent être analysés et résolus comme le montre le schéma suivant :



2.4 Techniques de résolution des problèmes d'ordonnancement

Certains problèmes d'ordonnancement peuvent être résolus efficacement en les réduisant à des problèmes d'optimisation combinatoire bien connus, tels que le problème de flot maximum, le problème de couplages de poids maximum, le problème de transport, etc... d'autres peuvent être résolus en utilisant des techniques standards, telles que la programmation dynamique et les méthodes branch-and-bound, ou bien en faisant appelle à des méthodes approchées telles que les heuristiques et les métaheuristiques.

2.5 Exemples de problèmes d'optimisation combinatoire

Le problème de transport

Le problème de transport consiste à déterminer le réseau de distribution permettant d'expédier de m origines une certaine quantité d'un produit à n destinations et ceci à un coût minimum. Considérons que la quantité disponible à l'origine i est de a_i unités et que la demande à la destination j est de b_j unités. La quantité à expédier de l'origine i à la destination j est identifiée par x_{ij} et le coût d'expédition correspondant par unité, par c_{ij}.

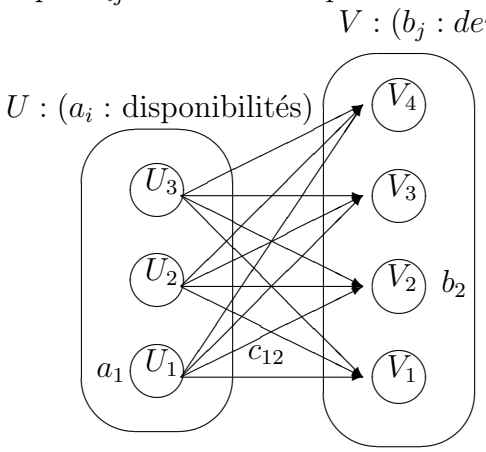


FIGURE 2.2 – Le réseau de transport $R = (U, V, c)$

Le modèle mathématique d'un problème de transport est :

$$\begin{cases} Min\ Z = \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} \quad \text{(minimiser le coût total de transport)} \\ s.c. \\ \sum_{i=1}^{n} x_{ij} = b_j, \forall j = 1, \cdots, m \quad : \text{contraintes des disponibilités} \\ \sum_{j=1}^{m} x_{ij} = a_i, \forall i = 1, \cdots, n \quad : \text{contraintes des demandes} \\ x_{ij} \in N, \forall j = 1, \cdots, m, \forall i = 1, \cdots, n \quad : \text{contraintes de non-négativité} \end{cases}$$

Le problème de transport peut être résolu à l'aide de l'algorithme du simplexe, mais il existe des algorithmes spécifiques par exemple la méthode du Stepping-Stone [39] qui tiendront compte des particularités du problème pour simplifier sa résolution. Le principe de cette méthode consiste d'abord a trouvé une solution réalisable en utilisant la règle du coin nord-ouest par exemple ensuite de l'améliorer itérativement.

Le problème d'affectation

Le problème d'affectation peut être considéré comme un cas particulier du problème de transport où $a_i = 1, i = 1, \ldots, m$ et $b_j = 1, j = 1, \cdots n$; de plus $m = n$. Le problème d'affectation consiste à déterminer par exemple l'affectation optimale de n ouvriers à n postes de travail qui minimiserait le temps de fabrication.
Le modèle d'un problème d'affectation est le suivant :

$$\begin{cases} Min\ Z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \quad \text{(minimiser le coût d'affectation)} \\ \quad s.c \\ \sum_{i=1}^{n} x_{ij} = 1, \forall j = 1, \cdots, n \quad : \text{un poste par ouvrier} \\ \sum_{j=1}^{n} x_{ij} = 1, \forall i = 1, \cdots, n \quad : \text{un ouvrier pour chaque poste} \\ x_{ij} \in \{0,1\} \forall j = 1, \cdots, m, \forall i = 1, \cdots, n. \end{cases}$$

Une méthode de résolution d'un tel problème est l'algorithme de Kuhn ou méthode hongroise [39]. Dans un problème d'affectation, nous avons toujours une matrice carrée et la solution optimale n'aura qu'une seule affectation dans une ligne ou dans une colonne donnée.

Le problème de flot maximum

Soit $G = (X, U)$ un graphe. A chaque arc $u \in U$, on associe un nombre réel, noté $f(u)$, appelé flux sur l'arc u. $F = \{f(u)/u \in U\}$ est un flot si et seulement si en chaque sommet $x \in X$, la somme des flux sur les arcs entrant est égal à la somme des flux sur les arcs sortant.
A partir du graphe précédent on construit un réseau $R = (V, U, c)$ en ajoutant à l'ensemble X des sommets, deux sommets particuliers s et p appelés source (entrée du graphe) et puits (sortie du graphe), et en associant aux arcs des capacités inférieures b_u et supérieures c_u avec $b_u \leq c_u$. On dit alors qu'un flot f sur R est compatible avec les capacités si : $b_u \leq f(u) \leq c_u$
Etant donné un réseau $R = (V, U, c)$, la détermination d'un flot maximum sur R est celui de la détermination d'un flot compatible avec les capacités, de valeur totale maximale.

Considérons la matrice d'incidence A aux arcs du graphe $G = (V, U)$. Si nous considérons f comme étant un vecteur, on montre que f est un flot sur le réseau R avec $b(u) = 0, \forall u$ si et seulement si $Af = 0$. Par conséquent le problème du flot maximum de s à p s'écrit :

$$\begin{cases} Max\ Z = f(u_r) \\ Af = 0 \\ 0 \leq f \leq c \ \ ou_r = (p, s) \end{cases}$$

Une des méthodes de résolution du problème de flot maximum est l'algorithme de Ford et Fulkerson.

Le problème de couplage de poids maximum

Dans la classe des problèmes d'optimisation combinatoire résolubles en temps polynomial, le couplage avec poids est un des plus «difficiles». Un algorithme pour la recherche du couplage maximum dans un graphe a été donné par Edmonds en 1965 [22]. Cet Algorithme est basé sur les deux théorèmes suivants.

Définition 2.3. *Considérons un couplage M dans $G = (V, E)$; on appelle chaîne alternée une chaîne simple (c'est-à-dire n'utilisant pas deux fois la même arêtes) dont les arêtes sont alternativement dans M et dans $E - M$.*

Théorème 2.1. *(Claude berge 1957 [14]) Un couplage M est maximum si et seulement s'il n'existe pas de chaîne alternée reliant deux sommets insaturés distincts.*

Théorème 2.2. *(Edmonds 1965 [22]) On peut trouver les chemins alternants en temps polynomial.*

On a donc un algorithme simple pour le couplage maximum :
Algorithme Couplage-Maximum

1. $M \leftarrow \emptyset$
2. Boucle
3. $P \leftarrow$chemin augmentant(G, M)
4. si $P = \emptyset$ alors retourner M sinon $M \leftarrow M \oplus P$
5. Fin de la boucle

Ici, $M \oplus P$ désigne la différence symétrique des deux graphes (c'est-à-dire les arêtes qui sont dans l'un ou l'autre mais pas dans les deux). Le problème reste donc de trouver des chemins augmentants.

2.6 Méthodes exactes

Ce sont des algorithmes qui fournissent une solution optimale au prix d'un temps de calcul souvent important pour les problèmes NP-difficile. Ces méthodes se caractérisent par un temps de calcul exponentiel, ce qui explique qu'elles ne sont utilisables que sur des problèmes de petite taille. Trois familles de méthodes exactes sont à distinguer : la résolution à partir d'une modélisation analytique, la programmation dynamique et la méthode de séparation et évaluation.

2.6.1 Modélisation analytique

La modélisation analytique d'un problème permet, non seulement de mettre en évidence l'objectif et les différentes contraintes du problème, mais également de le résoudre. L'idéal est de modéliser le problème sous la forme d'un programme linéaire dont les variables sont réelles. Il existe des solveurs comme CPLEX permettant de résoudre ce type de modèles. Les modèles deviennent plus difficiles à résoudre si on utilise des variables entières, et lorsque le modèle n'est pas linéaire.

2.6.2 La programmation dynamique

Introduite par Bellman dans les années 50 [12] la programmation dynamique décompose un problème P de dimension n en sous-problèmes de même nature que P. Le système est alors constitué de n étapes que l'on résout séquentiellement, le passage d'une étape à une autre se fait à partir des lois d'évolution du système et d'une décision. Une fois la valeur optimale trouvée, la méthode construit une solution optimale à partir des informations calculées.

Le principe d'optimalité est basé sur l'existence d'une équation récursive permettant de décrire la valeur optimale du critère à une étape en fonction de sa valeur à l'étape précédente.

2.6.3 Procédures par séparation et évaluation

Les procédures de type séparation et évaluation - PSE (Branch and Bound en anglais) sont des méthodes génériques de résolution des problèmes d'optimisation combinatoires. Ces méthodes reposent sur une énumération implicite et intelligente de l'ensemble des solutions réalisables. Pour cela, la séparation consiste à décomposer le problème initial en plusieurs sous-problèmes qui sont à leur tour décomposables. Ce processus peut se visualiser sous forme d'une arborescence d'énumération. Pour chaque sous-problème (noeud de l'arborescence), la procédure d'évaluation calcule (dans le cas d'un problème de minimisation) une borne inférieure de la solution obtenue à partir de ce sous-problème. Au préalable, une borne supérieure de la solution optimale a été calculé et est utilisée pour éviter l'exploration de noeuds dont la valeur de la borne inférieure est supérieure à la valeur de la borne supérieure. Cette borne supérieure est réactualisée lorsqu'une solution réalisable de valeur inférieure est trouvée. Ainsi, l'exploration de certaines branches de l'arbre est coupée, ce qui permet de ne pas énumérer réellement toutes les solutions.

2.7 Méthodes approchées

Pour des problèmes NP-difficiles de grande taille, les méthodes exactes sont peu envisageables à cause de leur temps de calcul. Il est alors possible d'utiliser des méthodes approximatives qui donnent des solutions dont l'optimalité n'est pas garantie, mais en un temps de calcul raisonnable. La performance de telles méthodes est généralement calculée par le rapport entre la valeur de la solution fournie et la valeur de la solution optimale. Si la solution optimale est non calculable, il est possible d'étudier expérimentalement le comportement de la méthode approchée en comparant ses performances soit à celles d'autres méthodes, soit à des bornes inférieures de la solution optimale.

Pour un problème de minimisation (resp. maximisation) et pour toute instance, nous définissons le rapport $R_A(I)$ pour un algorithme approché A par :
$R_A(I) = \frac{A(I)}{OPT(I)}$ (resp. $RA(I) = \frac{OPT(I)}{A(I)}$) où $A(I)$ est la valeur de la solution construite par l'algorithme A pour l'instance I et $OPT(I)$ est la solution optimale pour l'instance I.
Le rapport de performance absolue pour un algorithme approché A est donné par :
$R_A = inf\{r \geq 1 / RA(I) \leq r$ pour toute instance I du problème$\}$.
Le rapport de performance asymptotique pour un algorithme approchée A est donné par :
$R_A^{\infty} = inf\{r \geq 1 /$ pour un entier positif $k, RA(I) \leq r$ pour toute instance I du problème satisfaisant $OPT(I) \geq k\}$.

Méthodes heuristiques constructives

Les méthodes par construction progressive sont des méthodes itératives, où à chaque itération, une solution partielle est complétée. La plupart de ces méthodes sont des algorithmes gloutons car elles considèrent les éléments (travaux ou processeurs) dans un certain ordre sans jamais remettre en question un choix une fois qu'il a été effectué. Ces méthodes permettent de trouver une solution rapidement.

Un premier type de telles méthodes consiste à établir une liste initiale qui donne l'ordre de prise en compte des éléments. Cette liste est construite à priori, à partir d'un critère bien défini. La deuxième phase de l'algorithme construit une solution à partir des éléments dans l'ordre de la liste construite pour l'ordonnancement.

Un deuxième type de méthode consiste à choisir, au cours de la construction, l'affectation d'un travail à une machine en utilisant des règles de priorité.

Méthodes amélioratrices

Ces méthodes sont initialisées par une solution réalisable, calculée soit aléatoirement soit à l'aide de l'une des heuristiques constructives. Elles recherchent à chaque itération, une amélioration de la solution courante par des modifications locales. Cet examen se poursuit jusqu'à ce qu'un critère d'arrêt soit satisfait. L'utilisation de ces heuristiques itératives suppose que l'on puisse définir, pour toute solution S, un voisinage de solution N(S) contenant les solutions voisines. En général, le voisinage d'une solution est généré en appliquant plusieurs fois et de façon différente une petite transformation. Ce voisinage est ensuite évalué et comparé à la solution courante. Nous présentons ci-dessous les méthodes amélioratrices les plus connues :

Méthode de descente

Cette méthode se caractérise par le fait qu'elle ne considère, à chaque itération, que des solutions évoluant dans le sens de l'amélioration. Cette méthode ne conduit pas en général au minimum absolu mais seulement au minimum local qui constitue la meilleure des solutions accessibles compte tenu de la solution initiale. Pour améliorer l'efficacité, on peut l'appliquer plusieurs fois avec des conditions initiales différentes et retenir comme solution

finale le meilleur des minimums locaux obtenus, cette procédure augmente sensiblement le temps de calcul de l'algorithme et ne garantit pas de trouver une configuration optimale.

Recuit simulé

Cette méthode, due aux physiciens Kirkpatrick, Gellat et Vecchi, s'inspire des méthodes de simulation de Métropolies en mécanique statique. Les inconvénients du recuit simulé résident d'une part dans les réglages, comme la gestion de la décroissance par palier de la température. D'autre part, les temps de calcul peuvent devenir selon les cas très importants. Par contre, les méthodes de recuit simulé ont l'avantage d'être souples et rapidement implémentables lorsque l'on veut résoudre des problèmes d'optimisation combinatoire, le plus souvent de grande taille.

Recherche tabou

La principale particularité de la méthode tient dans la mise en œuvre de mécanismes inspirés de la mémoire humaine dont le but est de tirer les leçons du passé.
Le principe de base de la méthode tabou est simple, à partir d'une solution initiale quelconque, la méthode tabou engendre une succession de solutions ou configurations qui doivent aboutir à une solution optimale. A chaque itération, le mécanisme de passage d'une configuration (s) à la suivante (t) est le suivant :

- On construit l'ensemble des voisins de s, c'est-à-dire l'ensemble des configurations accessibles en un seul mouvement élémentaire à partir de s, soit V(s) l'ensemble de ces voisins.
- On évalue la fonction objectif f du problème pour chacune des configurations appartenant à V(s). La configuration t, qui succède à s dans la chaîne construite par tabou est la configuration de V(s) pour laquelle f prend la valeur minimale.

La procédure ne fonctionne généralement pas bien car il y a un risque de retourner à une configuration déjà retenue lors d'une itération précédente, ce qui provoque un cyclage. Pour éviter ce phénomène on tient à jour à chaque itération une liste tabou de mouvements interdits, cette liste circulaire contient m mouvements inverses. La recherche du successeur de la configuration courante se restreint alors aux voisins de s qui peuvent être atteints sans utiliser de mouvement de la liste tabou. La procédure est stoppée dès qu'on a effectué un nombre donné d'itérations sans améliorer la meilleure solution courante.

Algorithmes génétiques

Les algorithmes génétiques sont des techniques de recherche inspirées de l'évolution biologique des espèces et basées sur une imitation des phénomènes d'adaptation des êtres vivants. On part d'une population initiale sur laquelle des opérations de reproduction vont être réalisées dans l'objectif d'exploiter au mieux les caractéristiques et les propriétés de cette population. Un algorithme génétique consiste à faire évoluer progressivement, par génération successive la composition de cette population en maintenant sa taille constante.

Chapitre 3

Définition du problème, notations et état de l'art

Dans ce chapitre, nous présentons les différentes contraintes prises en compte dans les problèmes d'ordonnancement traités dans cette thèse. La section 1 est dédiée aux problèmes d'ordonnancement de type flow shop classiques et aux problèmes d'ordonnancement de type flow shop en présence des contraintes d'écarts temporels (temps de latences ou time lags) et les contraintes de recirculation. La section 2 est consacrée aux définitions et notations utilisées dans cette thèse. Nous citons également quelques applications de notre problème. Dans le dernier paragraphe, nous présentons un état de l'art.

3.1 Les contraintes considérées

Dans le problème du flow shop classique avec m machines $Fm||C_{max}$, les tâches sont ordonnancées sur les machines, selon le même ordre $(M_1, M_2, ..., M_m)$ et le but est de minimiser le makespan. Dans ce problème, les tâches passent par une machine, seulement une fois, la préemption des tâches n'est pas autorisée et il n'y a pas de contraintes d'écart temporel entre les opérations ni de contraintes de précédence entre les tâches.

3.1.1 Contraintes de recirculation

La contrainte de recirculation signifie qu'une tâche peut visiter une machine ou un ensemble de machines plusieurs fois. Les problèmes où ce type de contraintes est imposé sont

appelés "reentrant flow shops". Selon le schéma d'exécution des tâches, on peut distinguer plusieurs types de contraintes de recirculation.

Contrainte de type : "cyclic-reentrant"

Le schéma d'exécution des tâches dans les problèmes où la contrainte de type cyclic-reentrant est imposée est selon le nombre de passage r. Par exemple, si le nombre de passage est égale à trois, $r = 3$, alors les tâches sont ordonnancées dans cet ordre $(M_1, M_2, \ldots, M_m, M_1, M_2, \ldots, , M_m, M_1, M_2, \ldots, M_m,)$.

Contrainte de type : "chain-reentrant"

Dans notre thèse on s'intéresse à ce type de contrainte. L'ordre d'exécution des tâches dans les problèmes de type chain-reentrant shop est comme suit $(M_1, M_2, \ldots, M_m, M_1)$. On remarque qu'une fois le passage sur les m machines terminé, il y a un retour sur la première machine.

Contrainte de type : "V-reentrant"

Les problèmes qui ont ce type de contrainte sont aussi connus dans la littérature comme les problèmes de type "V-shop" voir (Lev and Adiri [30]). L'ordre d'exécution des tâches dans ces problèmes est sous forme d'un V $(M_1, M_2, \ldots, M_{m-1}, M_m, M_{m-1} \ldots, M_2, M_1)$. On remarque que pour $m = 2$, "chain-reentrant" et "V-reentrant" sont équivalentes.

3.1.2 Contraintes de temps de latence

Les contraintes de temps de latence ou d'écarts temporels (time lags en anglais), connues aussi dans la littérature comme des contraintes de délais ou contraintes d'attente, sont une généralisation des contraintes de précédence classiques. Dans le cas classique, une tâche peut débuter juste après la fin de la tâche qui la précède. La présence de contraintes d'écarts temporels oblige une tâche à ne commencer qu'après un certain temps à la suite du début ou la fin de la tâche qui la précède. Dans ce cas, le temps d'attente est minoré, appelé aussi "minimum time lag ". Il est parfois possible que la seconde tâche doit absolument commencer avant un temps donné après le début ou la fin de la première tâche, l'attente est alors majorée, appelée ici "maximum time lag". Un autre cas regroupe les deux situations,

où l'attente est à la fois minorée et majorée, dans ce cas l'attente se situe dans un intervalle de temps. Dans notre thèse on considère le cas où l'écart séparant les deux opérations de chaque tâche correspond à un time-lag minimal et un time-lag maximal de même valeur. Ce type de contrainte est dit temps de latence exact ou time-lag exact.

3.2 Définition du problème traité

Dans ce travail, nous considérons le problème d'ordonnancement d'un ensemble de n tâches, $T = \{T_1, T_2\ , .., T_n\}$ sur 2 machines. L'atelier est du type chain-reentrant shop et en présence des temps de latence exacts. c-à-d. chaque tâche T_j commence son traitement sur la première machine appelée aussi machine primaire ensuite elle passe sur la deuxième machine et elle revient une deuxième fois sur la première machine, après un temps de latence exact l_j. On suppose qu'à tout instant une machine n'est allouée qu'au plus une tâche et chaque tâche n'est traitée que par au plus une machine. Le mode de traitement des tâches est non-préemptif, c'est-à-dire l'interruption des tâches n'est pas autorisée.
Afin de faciliter la description et la classification du problème d'ordonnancement étudié, nous allons adopter la classification proposé par Graham et al. [25] qui structure les données d'un problème d'ordonnancement sur la base d'une notation à trois champs distincts $\alpha/\beta/\gamma$.

- $\alpha = F2$: l'atelier est de type flow shop à 2 machines.
  Le champ β est décomposé en trois sous-champs $\beta = \beta_1\beta_2\beta_3$ où :
- $\beta_1 = chain - reentrant$: qui signifie qu'une tâche doit revisiter la première machine une deuxième fois après avoir été traitée sur les deux machines.
- $\beta_2 \in\{\phi, p_j = p\ \}$: les temps d'exécution des tâches sont soit égaux à p, soit quelconques.
- $\beta_3 \in\{\phi, l_j = L\ \}$: les temps de latence sont soit égaux à L, soit quelconques.
- $\gamma = C_{max}$: L'objectif est de minimiser la date de fin de traitement de l'ensemble des tâches.

Le problème général est donc noté $P_1 : F2|chain{-}reentrant, l_j|C_{max}$ où : $F2|ChR, l_j|C_{max}$.
Pour simplifier, les opérations et les durées de traitement sont notées comme suit (voir Figure 1) :

- $a_{[j]}$: La première opération de la tâche T_j sur la première machine.
- $b_{[j]}$: L'opération de la tâche T_j sur la deuxième machine.
- $c_{[j]}$: La deuxième opération de la tâche T_j sur la première machine.
- a_j : Temps de traitement de la première opération de la tâche T_j sur la première machine.
- b_j : Temps de traitement de la tâche T_j sur la deuxième machine $b_j \leq l_j$.
- c_j : Temps de traitement de la deuxième opération de la tâche T_i sur la première machine.

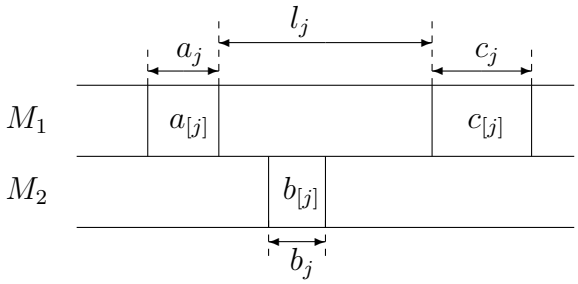


FIGURE 3.1 – Schéma d'exécution d'une tâche dans le problème $P_1 : F2|ChR, l_j|C_{max}$

3.2.1 Motivation

Dans le problème de flow shop classique les tâches passent une seule fois sur une machine. En pratique, cette supposition n'est pas toujours justifiée. En fait, dans la vie pratique, il existe des situations où une tâche peut visiter une machine ou un ensemble de machines plus d'une fois. Par exemple, on peut rencontrer ce type de problème dans les ateliers de fabrication des semi-conducteurs où chaque composant revisite la même machine plusieurs fois (Wang et al. [42]). Cette caractéristique de recirculation peut être également rencontrée dans les ateliers de fabrication des cartes d'impression [42], ou encore dans les ateliers de fabrication de meubles où chaque composant subit une opération de peinture, puis une opération de polissage et enfin une opération de peinture.
Une application du problème de flow shop avec recirculation et en présence d'un temps de latence exact peut être rencontrée dans les blocs opératoires. Lorsqu'un malade subit une opération chirurgicale celui-ci, passe par les étapes suivantes :
1. Le chirurgien (la machine primaire) opère le malade T_j et l'opération dure un temps a_j.
2. Le malade reste dans la salle de réanimation où il est surveillé (par l'infirmière) et ceci prend un temps b_j.
3. Après une période de temps de latence exact l_j, le chirurgien doit revisiter son malade T_j, et ceci dure un temps c_j.

3.3 Etat de l'art

Cette section examine la complexité des problèmes d'ordonnancement de type flowshop. On commence par présenter les principaux résultats de littérature sur le problème flow shop classique. Ensuite, nous nous focalisons sur les problèmes d'ordonnancement de type flowshop de recirculation et en présence de contraintes d'écarts temporels et on termine par les problèmes d'ordonnancement de type flowshop avec contrainte no-wait.

3.3.1 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 taches sur chacune des machines est identique. Résoudre ces problèmes revient à trouver une permutation $\pi = (\pi(l), \pi(2), \ldots, \pi(n))$ minimisant le makespan. La restriction aux flow-shop de permutation permet de simplifier la structure des solutions et des preuves Pinedo [34]. En effet, d'un point de vue pratique, l'ordonnancement obtenu a une structure simple pouvant être facilement implémentée. Le problème de flow-shop est dominé par la permutation pour un nombre de machines $m \leq 3$. Johnson [27] a prouvé que le problème à deux machines est résoluble en $O(nlogn)$. Le problème devient NP-difficile au sens fort pour $m \geq 3$ Pinedo [34].

3.3.2 Flow shop avec recirculation

Le problème $F2|chain-reentrant|C_{max}$ a été étudié par Wang et al. [42]. Ils ont donné une caractérisation de la solution optimale, et ils ont développé un algorithme de type séparation et évaluation et des heuristiques pour résoudre ce problème. Lev et Adiri [30] ont traité le problème $Fm|V-reentrant|C_{max}$, ils ont montré que ce problème est NP-difficile et ils ont proposé des sous problèmes résolubles en des temps polynomiaux. Choi et Kim [19] ont proposé plusieurs heuristiques pour résoudre le problème $Fm|cyclic-reentrant|C_{max}$ et ils ont développé des propriétés de dominances et des bornes inférieures. Pan et Chen [33] ont traité le même problème pour lequel seules les séquences de permutation sont considérées, ils ont développé trois formulations mathématiques sous forme de programmes linéaires en variables mixtes et six heuristiques. Chen et al. [18] ont adopté un algorithme de type tabou search hybride pour résoudre ce problème. Méziani et Boudhar [16] ont traité le problème de flow shop à deux machines avec recirculation sur la deuxième machine (M_1, M_2, M_2), ils ont prouvé que ce problème est résoluble en un temps polynomial et ils ont proposé un algorithme exact pour le résoudre.

3.3.3 Flow shop en présence des temps de latence

Pour le flowshop à deux machines, le critère du makespan est le plus étudié dans la littérature. Le cas le plus simple se présente quand toutes les tâches ont le même time-lag L. On peut facilement montrer que ce problème est équivalent au cas sans time-lags en

décalant toutes les tâches sur la deuxième machine de L unités de temps, par conséquent l'algorithme de Johnson [27] donne la solution optimale. Considérons maintenant le cas où tous les time-lags min sont différents. Le problème $F2|lj|C_{max}$ est étudié par Mitten [31]. Il a montré que si on se restreint aux ordonnancements de permutation, la solution optimale peut être obtenue par l'algorithme de Johnson [27] moyennant des modifications sur les durées opératoires des tâches : pour chaque tâche, il ajoute la valeur de son "time-lag min" aux durées opératoires sur les deux machines. Par contre, si on s'intéresse au cas général (pas de restriction aux ordonnancements de permutation), la minimisation du makespan est NP-difficile au sens fort, Dell'Amico [20], Lawler et al. [29]. Yu et al. [43] ont montré que ce problème reste NP-difficile au sens fort même si toutes les durées opératoires des tâches sont unitaires, ou si les durées opératoires ne dépendent que des tâches mais pas des machines, et que les time-lags ne peuvent prendre que deux valeurs. Yu [44] a proposé une condition pour laquelle l'ordonnancement de permutation devient dominant, à savoir, il faut que $\forall i, j \quad l_i \leq l_j + max\{p_{1,j}; p_{2,j}\}$. Par ailleurs, quand le flowshop est composé de plus de deux machines et en présence de contraintes de time-lags min, la plupart des problèmes de minimisation de critères réguliers sont NP-difficile au sens fort, même si on se restreint aux ordonnancements de permutation. En fait, déjà dans le cas classique (sans les time-lags) hormis le makespan, les autres critères sont déjà difficiles. Rayward-Smith et Rebaine [35] ont proposé deux heuristiques avec des rapports de performance pour le problème de flow shop à deux machines en présence des contraintes temporelles où les temps de traitement sont unitaires et les time lag sont arbitraires. Munier-Kordon et Rebaine [28] ont présenté un algorithme polynomial pour le même problème lorsque le nombre de machines est égale à 3 ou 4. Rebaine [36] a donné une évaluation du rapport de performance entre la meilleur solution connue pour le problème flow shop classique et le problème flow shop de permutation avec temps de latence. Lorsque une tâche est composée de deux opérations séparées par un exact time lag le problème revient au cas de coupled-task noté $1|coupled - task|C_{max}$, ce dernier a été introduit par Shapiro [40]. Il a étudié des cas particulier et il a proposé des heuristiques avec des expérimentations numériques. Orman et Potts [32] ont étudié le même problème, ils ont présenté plusieurs résultats de complexité. Le seul problème resté ouvert pour quelques années est celui où les durées opératoires des premières opérations et les durées opératoires des secondes opérations sont respectivement égales et les time-lags exacts identiques ($1|coupled - task, a_j = a, l_j = L, b_j = b|C_{max}$). Ce problème a été résolu polynomialement plus tard par Ahr et al. [3] pour des petites instances en utilisant un algorithme basé sur le plus court chemin. Cet algorithme a été adapté par Brauner et al. [15] pour résoudre un autre type de problème de coupled-task. Ageev et Baburin [1] ont proposé un 7/4 et un 3/2-approximation algorithme pour résoudre les problèmes $1|coupled - task, a_j = b_j = 1, l_j = L|C_{max}$ et $F2|a_j = b_j = 1, l_j = L|C_{max}$ respectivement. Ageev et Kononov [2] ont donné plusieurs résultats d'approximation selon les valeurs a_j et b_j. Quelques travaux ont été considérés en ajoutant

d'autres contraintes aux problèmes de coupled-tasks. Blazewicz et al. [13] ont montré que le problème polynomial $1|coupled - task, a_j = b_j = 1, l_j|C_{max}$ devient NP-difficile si on ajoute la contrainte de précédence entre les taches. Fondrevelle et al. [23] ont considéré un nouveau critère pour les problèmes d'ordonnancement de type flow shop en présence des contraintes temporelles à savoir la minimisation de la somme des dates de fin sur les machines, noté $\sum MCk$, ils ont montré que ce critère généralise le makespan et ils ont donné plusieurs résultats de complexité.

3.3.4 Flow shop à contrainte no-wait

La contrainte sans attente " no-wait " signifie qu'aucune attente entre deux opérations d'une même tâche n'est permise. Le problème $F2/no - wait/C_{max}$ est résolu en temps polynomial par l'algorithme de Gilmore et Gomory [26]. Cependant, ce problème est NP-difficile au sens fort lorsque le nombre de machine est égale à trois Hans Rock [37]. Ultérieurement, des contraintes additionnelles ont été introduites comme la contrainte de blocage qui exprime qu'une tâche ne doit libérer une machine que pour passer directement sur la machine suivante par exemple le problème $F2/block(1,2)/C_{max}$ signifie qu'il existe une contrainte de blocage entre la première et la deuxième machine.

Chapitre 4

Analyse de complexité du problème $P_2 : F2|ChR, l_j = L|C_{max}$

Dans ce chapitre, nous considérons le problème $P_2 : F2|ChR, l_j = L|C_{max}$ où les temps de latence sont identiques $l_j = L$. On commence par donner un exemple pour illustrer ce problème. Ensuite nous démontrons la NP-complétude de ce dernier, et nous proposons des sous problèmes résolubles en des temps polynomiaux, certains d'entre eux sont équivalents au problème du couplage de poids maximum et les autres ont été prouvés en utilisant leurs propres particularités. La liste des problèmes traités est résumée dans la Table 4.1.

TABLE 4.1 – Liste des problèmes étudiés

Problème	Complexité
$P_{21} : F2\|ChR, a_j = \frac{L}{2}, c_j = \frac{L}{2}, l_j = L\|C_{max}$	$NP - difficile$
$P_{22} : F2\|ChR, a_j, c_j > \frac{L}{2}, l_j = L\|C_{max}$	$Polynomial$, $O(n^{2.5})$
$P_{23} : F2\|ChR, a_j = \frac{L}{2}, b_i + b_j \leq L, c_j \leq \frac{L}{2}, l_j = L\|C_{max}$	$Polynomial, O(nlogn)$
$P_{24} : F2\|ChR, a_j > L\|C_{max}$	$Polynomial, O(n)$
$P_{25} : F2\|ChR, c_j > L\|C_{max}$	$Polynomial, O(n)$

Exemple illustratif Pour illustrer le problème $P_2 : F2|ChR, l_j = L|C_{max}$, on considère l'instance suivante : 5 tâches T_1, T_2, T_3, T_4 et T_5 à ordonnancer sur 2 machines. Les temps de traitement sont donnés dans la Table 4.2, avec $L = 3$.

Une solution réalisable du problème est donnée à la Figure 4.1, avec $C_{max} = 19$.

TABLE 4.2 – Durées de traitement des 5 tâches

T_i	T_1	T_2	T_3	T_4	T_5
a_i	2	3	2	1	1
b_i	2	1	3	1	3
c_i	1	2	1	2	3

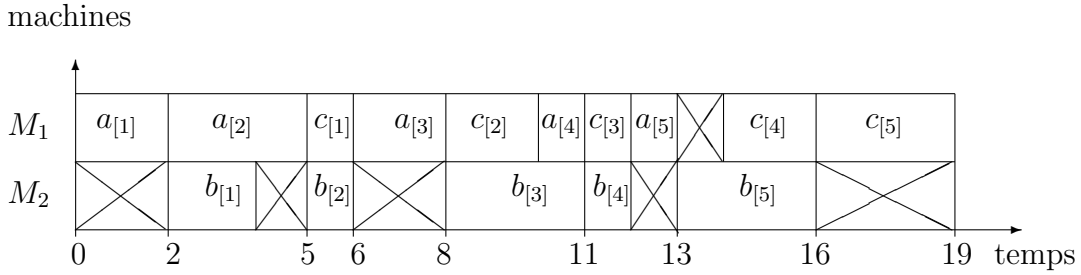


FIGURE 4.1 – Une solution du problème P_2.

4.1 NP-difficulté

Dans cette section, nous montrons que le problème $P_2 : F2|ChR, l_j = L|C_{max}$ est NP-difficile au sens fort, même si les temps de traitement sur la première machine sont identiques $a_j = c_j = \frac{L}{2}$.

Avant de donner les différents résultats obtenus, nous parlerons d'abord de la notion d'entrelacement entre les tâches ainsi que des conditions d'entrelacement et de la durée de traitement d'un batch.

Définition 4.1. *On dit que la tâche T_i peut être entrelacée avec la tâche T_j, si, et seulement si l'opération $a_{[j]}$ de la tâche T_j est ordonnancée entre les deux opérations $a_{[i]}, c_{[i]}$ de la tâche T_i et l'opération $c_{[i]}$ de la tâche T_i est ordonnancée entre les deux opérations $a_{[j]}, c_{[j]}$ de la tâche T_j voir Figure 4.2.*

De la définition ci-dessus découle la condition d'entrelacement suivante : la tâche T_i est entrelacée avec la tâche T_j (on dit aussi que les deux tâches T_i, T_j sont ordonnancées dans le même batch), si, et seulement si $a_j \leq L$ et $c_i \leq L$.

Définition 4.2. *Si les deux tâches T_i et T_j sont ordonnancées dans le même batch alors la durée de traitement de ce batch est notée d_{ij}, et est égale au temps qui s'écoule entre le début de l'opération a_i et la fin de l'opération c_j. voir Figure 4.2.*

Lemme 4.1. *Si la tâche T_i peut être entrelacée avec la tâche T_j et T_j peut aussi être entrelacée avec T_i, alors la durée de traitement optimale du batch formé en entrelaçant les*

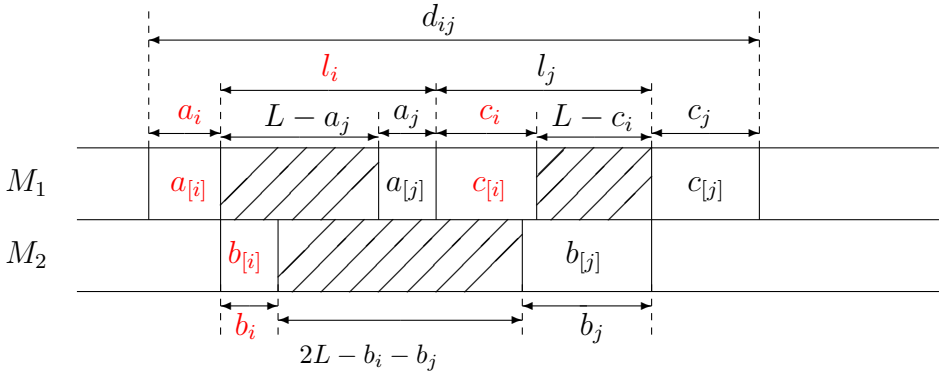


FIGURE 4.2 – Exemple d'un batch contenant deux tâches entrelacées T_i et T_j dans cet ordre

deux tâches T_i, T_j est égale à $d^* = \min\{d^*_{ij}, d^*_{ji}\}$. où d^*_{ij} (resp d^*_{ij}) est la durée de traitement minimale du batch contenant les deux tâches T_i, T_j (resp T_j, T_i) dans cet ordre.

Preuve. Si la tâche T_i peut étre entrelacée avec la tâche T_j ceci implique que $a_j \leq L$ et $c_i \leq L$ alors on a :
$d^*_{ij} = a_i + c_j + 2L - \min\{L - a_j, L - c_i, 2L - b_i - b_j\}$. où $L - a_j, L - c_i, 2L - b_i - b_j$ sont des temps morts, voir la partie hachurée de la Figure 4.2
$= a_i + c_j + L + \max\{a_j, c_i, b_i + b_j - L\}$.
De même, si la tâche T_j peut étre entrelacée avec la tâche T_i ceci implique que $a_i \leq L$ et $c_j \leq L$ on a alors :
$d^*_{ji} = a_j + c_i + 2L - \min\{L - a_j, L - c_i, 2L - b_j - b_i\}$.
$= a_j + c_i + L + \max\{a_i, c_j, b_j + b_i - L\}$.
D'où la séquence optimale entre T_i et T_j est $d^* = \min\{d^*_{ij}, d^*_{ji}\}$. ■

- Un batch contenant plusieurs tâches $T_1, ..., T_i$ ordonnancées dans cet ordre est noté par $\{T_1, ..., T_i\}_B$.

Théorème 4.1. *[10] Le problème $P_{21} : F2|ChR, a_j = c_j = \frac{L}{2}, l_j = L|C_{max}$ est NP-difficile au sens fort.*

Preuve.

Considérons le problème de décision suivant connu sous le nom de 3-partitions : étant donné un ensemble $\bar{A} = \{\bar{a}_1, \bar{a}_2, ..., \bar{a}_n\}$ d'entiers positifs chacun compris strictement entre $\frac{B}{4}$ et $\frac{B}{2}$ tel que $\sum_{\bar{a}_i \in \bar{A}} \bar{a}_i = mB$. Existe-t-il m sous-ensembles disjoints $S_1, S_2, ..., S_m$ de cardinalité 3 et de poids B ? 3-partitions est NP-difficile au sens fort. [24]
Montrons que ce problème se réduit polynomialement au problème de décision suivant : étant donné $3m$ tâches indépendantes et non morcelables $n = 3m$ de temps de traitement

$a_i = \frac{B}{4}$, $b_i = \bar{a}_i, c_i = \frac{B}{4}$ et un temps de latence $L = \frac{B}{2}$.
Existe-t-il un ordonnancement de ces $3m$ tâches sur les deux machines M_1 et M_2 de durée totale inférieure ou égale à $\frac{3mB}{2}$?
Notre problème appartient à la classe NP car on peut vérifier en temps polynomial si une permutation des tâches vérifie toutes les conditions.
Nous prouvons que le problème d'ordonnancement a une solution si et seulement si le problème de 3-partitions a une solution.
Si le problème de 3-partitions a une solution, donc il existe une partition de A en m sous ensembles disjoints de cardinalité 3 et de poids B à laquelle nous associons l'ensemble des tâches $T = \{T_{[11]}, T_{[12]}, T_{[13]}\} \cup ... \cup \{T_{[m1]}, T_{[m2]}, T_{[m3]}\}$ tel que : $T_{[i,j]}$ désigne la tâche numéro j du batch i. Nous construisons donc une solution pour le problème d'ordonnancement comme le montre la figure suivante, avec $C_{max} = m(\frac{3B}{2}) = \frac{3mB}{2}$.

machines

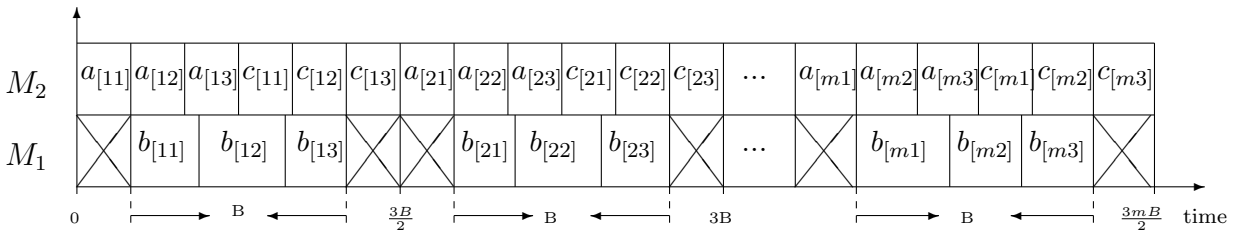


FIGURE 4.3 – m batchs, contenant chacun 3 tâches entrelacées

Cette solution est réalisable car :
– $b_{[i1]}$: termine avant le début de $c_{[i1]}$ car sa durée est strictement inférieure à $\frac{B}{2} = L$.
– $b_{[i2]}$: termine avant le début de $c_{[i2]}$ car la durée de $b_{[i3]}$ est strictement supérieure à $\frac{B}{4}$.
– $b_{[i2]}$: débute après la fin de $a_{[i2]}$ car $b_{[i1]}$ est strictement supérieure à $\frac{B}{4}$.
– $b_{[i3]}$: débute après la fin de $a_{[i3]}$ car la durée de $b_{[i1]} + b_{[i2]}$ est strictement supérieure à $\frac{B}{2} = L$.

Si le problème d'ordonnancement a une solution inférieure ou égale à $\frac{3mB}{2}$ donc la solution est comme le montre la figure précédente. Nous avons m batchs de tâches où chaque batch est composé de 3 tâches entrelacées entre elles, la durée de traitement de chaque batch est égale à $\frac{3B}{2}$. Aucune tâche n'est traitée seule car sa durée est égale B. Deux tâches ne peuvent pas être traitées seules car leur durée est égale à $\frac{5B}{4}$. Dans ces deux cas, nous avons $B + \frac{5B}{4} = \frac{9B}{4} > \frac{3B}{2}$: De même, si trois tâches sont traitées seules, leur durée totale est égale à $3B > \frac{3B}{2}$ (voir la figure 4.4).

On a aussi $b_{[i1]} + b_{[i2]} + b_{[i3]} = B$ $(\forall i = \overline{1, m})$ car il ne peut pas exister un batch i pour lequel $b_{[i1]} + b_{[i2]} + b_{[i3]} > B$ car la solution dans ce cas n'est pas réalisable (les times lags ne seront pas respectés pour ce batch) et il ne peut pas exister aussi un batch i pour le quel $b_{[i1]} + b_{[i2]} + b_{[i3]} < B$ car si ce batch existe il y'aura forcément un autre batch j pour

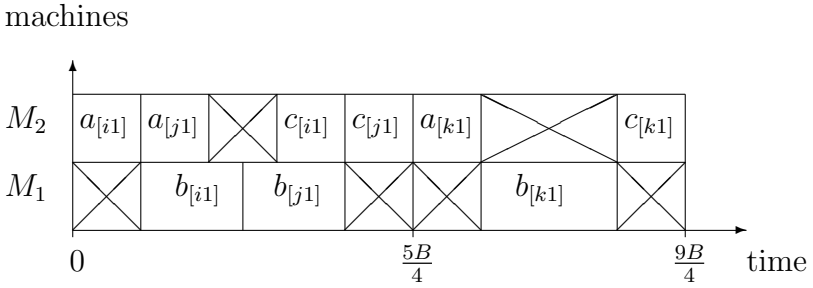


FIGURE 4.4 – batchs composés par 2 tâches entrelacées et d'une seule tâche

lequel $b_{[j1]} + b_{[j2]} + b_{[j3]} > B$ par ce que les $b_{[ij]} = \bar{a}_i$ et $\sum_{\bar{a}_i \in \bar{A}} \bar{a}_i = mB$ et la solution dans ce cas n'est pas aussi réalisable (car les temps de latence ne seront pas respectés pour le batch j) donc les batchs formés donnent une solution pour le problème de 3-partitions. ■

4.2 Sous problèmes faciles

Dans cette section nous proposons des sous-problèmes résolubles en des temps polynomiaux, nous considérons les problèmes où les temps de traitement sont constants ou bien supérieurs à une certaine constante. Nous considérons d'abord le problème P_{22} : $F2|ChR, a_j, c_j > \frac{L}{2}, l_j = L|C_{max}$, ensuite nous donnons d'autres sous-problèmes résolubles en des temps polynomiaux : P_{23}, P_{24} et P_{254} que nous définissons par la suite.

4.2.1 Problème $P_{22} : F2|ChR, a_j, c_j > \frac{L}{2}, l_j = L|C_{max}$

Dans le problème P_{22}, les temps de traitement sur la première machine sont strictement supérieurs à $\frac{L}{2}, (a_j, c_j > \frac{L}{2})$ ceci nous amène à donner la caractérisation suivante de la solution optimale.

Lemme 4.2. *La solution optimale du problème P_{22} est formée d'une suite de batchs de cardinalité 2 ou 1, ordonnancées sans temps mort.*

Preuve.

Si les temps de traitement sur la première machine sont strictement supérieurs à $\frac{L}{2}$ pour toutes les tâches $(\forall j, a_j, c_j > \frac{L}{2})$, alors on ne peut pas entrelacer plus que deux tâches dans un même batch, car la somme des temps de traitement de n'importe quelles deux opérations sur la première machine est strictement supérieure à L, ($l_j = L \ \forall j$), de plus les batchs formés ne peuvent pas ce chevaucher. ■

Théorème 4.2. *[10] Le problème P_{22} se réduit au problème de couplage de poids maximum.*

Preuve. On construit un graphe $G = (V, E)$ tel que :
V : l'ensemble des sommets égal à l'ensemble des tâches T.
E : l'ensemble des arêtes tel que $(T_l, T_{l\prime}) \in E$ ssi la condition d'entrelacement est vérifiée c-à-d. : ($a_{l\prime} \leq L$ et $c_l \leq L$) ou ($a_l \leq L$ et $c_{l\prime} \leq L$).
Soit la matrice d'incidence sommet-arête du graphe $G = (V; E)$ où

$$\lambda_{lk} = \begin{cases} 1, & \text{si l'arête k est incidente au sommet l;} \\ 0, & \text{sinon.} \end{cases}$$

Pour $k = 1, \ldots, q$, $l = 1, \ldots, n$ (q le nombre d'arêtes).

Soit $\rho_k = min\{d_{s_k t_k}, d_{t_k s_k}\}$ le poids de l'arête k, si l'arête k est incidente aux sommets s_k et t_k.

Où
$d_{s_k t_k}$: La durée de traitement du batch contenant les deux tâches entrelacées s_k et t_k dans cette ordre.
$d_{t_k s_k}$: La durée de traitement du batch contenant les deux tâches entrelacées t_k et s_k dans cette ordre.

$$\text{On a} \begin{cases} d_{s_k t_k} = a_{s_k} + c_{t_k} + L + \max\{a_{t_k}, c_{s_k}, b_{s_k} + b_{t_k} - L\} \\ d_{t_k s_k} = a_{t_k} + c_{s_k} + L + \max\{a_{s_k}, c_{t_k}, b_{s_k} + b_{t_k} - L\} \end{cases}$$

Comme la solution optimale est une suite de batchs de cardinalité 2 ou 1, alors le programme linéaire, correspondant à ce problème, s'écrit comme suit :

$$min\ C_{max} = (\sum_{k=1}^{q} \rho_k x_k) + \sum_{l=1}^{n} (1 - \sum_{k=1}^{q} \lambda_{lk} x_k)\delta_l.$$

$$\text{s.c.} \begin{cases} \sum_{k=1}^{q} \lambda_{lk} x_k \leq 1 \text{ for } l = \overline{1, n} \\ x_k \in \{0, 1\} \text{ for } k = \overline{1, q} \end{cases}$$

où

$$\text{- } x_k = \begin{cases} 1, & \text{si les deux tâches reliées par l'arête k sont entrelacées entre elles dans} \\ & \text{le même batch ;} \\ 0, & \text{sinon.} \end{cases}$$

- $\sum_{k=1}^{q} \rho_k x_k$: la somme des durées de traitement des batchs contenant deux tâches entrelacées.

- $\sum_{l=1}^{n}(1 - \sum_{k=1}^{q} \lambda_{lk}x_k)\delta_l$: la somme des durée de traitement des batchs contenant une seule tâche.

- $\delta_l = a_l + c_l + L$: temps de traitement du batch qui contient, la tâche l.

- $\sum_{k=1}^{q} \lambda_{lk}x_k \leq 1$ indique que chaque tâche doit appartenir à au plus, un batch.

Nous avons

$$(\sum_{k=1}^{q} \rho_k x_k) + \sum_{l=1}^{n}(1 - \sum_{k=1}^{q} \lambda_{lk}x_k)\delta_l = (\sum_{l=1}^{n} \delta_l) - (\sum_{l=1}^{n}\sum_{k=1}^{q} \lambda_{lk}x_k\delta_l - \sum_{k=1}^{q} \rho_k x_k)$$

$$= (\sum_{l=1}^{n} \delta_l) - (\sum_{k=1}^{q}\sum_{l=1}^{n} \lambda_{lk}\delta_l x_k - \sum_{k=1}^{q} \rho_k x_k)$$

$$= (\sum_{l=1}^{n} \delta_l) - \sum_{k=1}^{q}(\sum_{l=1}^{n} \lambda_{lk}\delta_l - \rho_k)x_k.$$

$$= (\sum_{l=1}^{n} a_l + c_l + L) - \sum_{k=1}^{q}(\sum_{l=1}^{n} \lambda_{lk}(a_l + c_l + L) - \min\{d_{s_k t_k}, d_{t_k s_k}\})x_k.$$

$$= (\sum_{l=1}^{n} a_l + c_l + L) - \sum_{k=1}^{q}\left(\sum_{l=1}^{n} \lambda_{lk}(a_l + c_l + L) - \min\left\{\begin{array}{l} a_{s_k} + c_{t_k} + L + \max\{a_{t_k}, c_{s_k}, b_{s_k} + b_{t_k} - L\}, \\ a_{t_k} + c_{s_k} + L + \max\{a_{s_k}, c_{t_k}, b_{s_k} + b_{t_k} - L\} \end{array}\right\}\right) x_k.$$

$$= nL + \sum_{l=1}^{n} a_l + \sum_{l=1}^{n} c_l - \sum_{k=1}^{q}(a_{s_k} + c_{s_k} + L + a_{t_k} + c_{t_k} + L - \min\left\{\begin{array}{l} a_{s_k} + c_{t_k} + L + \max\{a_{t_k}, c_{s_k}, b_{s_k} + b_{t_k} - L\}, \\ a_{t_k} + c_{s_k} + L + \max\{a_{s_k}, c_{t_k}, b_{s_k} + b_{t_k} - L\} \end{array}\right\})x_k.$$

$$= nL + \sum_{l=1}^{n} a_l + \sum_{l=1}^{n} c_l - \sum_{k=1}^{q} \max\left\{\begin{array}{l} 2L + a_{s_k} + c_{s_k} + a_{t_k} + c_{t_k} - a_{s_k} - c_{t_k} \\ 2L + a_{s_k} + c_{s_k} + a_{t_k} + c_{t_k} - a_{t_k} - c_{s_k} \end{array}\right.$$

$$\left.\begin{array}{l} -L - \max\{a_{t_k}, c_{s_k}, b_{s_k} + b_{t_k} - L\}, \\ -L - \max\{a_{s_k}, c_{t_k}, b_{s_k} + b_{t_k} - L\} \end{array}\right\} x_k.$$

$$= nL + \sum_{l=1}^{n} a_l + \sum_{l=1}^{n} c_l - \sum_{k=1}^{q} \max \left\{ \begin{array}{l} L + c_{s_k} + a_{t_k} - \max\{a_{t_k}, c_{s_k}, b_{s_k} + b_{t_k} - L\}, \\ L + a_{s_k} + c_{t_k} - \max\{a_{s_k}, c_{t_k}, b_{s_k} + b_{t_k} - L\} \end{array} \right\} x_k.$$

$$= nL + \sum_{l=1}^{n} a_l + \sum_{l=1}^{n} c_l - \sum_{k=1}^{q} \max \left\{ \begin{array}{l} \min\{L + c_{s_k}, L + a_{t_k}, 2L + c_{s_k} + a_{t_k} - b_{s_k} - b_{t_k}\}, \\ \min\{L + c_{t_k}, L + a_{s_k}, 2L + a_{s_k} + c_{t_k} - b_{s_k} - b_{t_k}\} \end{array} \right\} x_k.$$

$nL + \sum_{l=1}^{n} a_l + \sum_{l=1}^{n} c_l$ est une constante supérieure à
$$\sum_{k=1}^{q} \max \left\{ \begin{array}{l} \min\{L + c_{s_k}, L + a_{t_k}, 2L + c_{s_k} + a_{t_k} - b_{s_k} - b_{t_k}\}, \\ \min\{L + c_{t_k}, L + a_{s_k}, 2L + a_{s_k} + c_{t_k} - b_{s_k} - b_{t_k}\} \end{array} \right\} x_k.$$

Alors, minimiser C_{max} est équivalent à maximiser
$$\sum_{k=1}^{q} \max \left\{ \begin{array}{l} \min\{L + c_{s_k}, L + a_{t_k}, 2L + c_{s_k} + a_{t_k} - b_{s_k} - b_{t_k}\}, \\ \min\{L + c_{t_k}, L + a_{s_k}, 2L + a_{s_k} + c_{t_k} - b_{s_k} - b_{t_k}\} \end{array} \right\} x_k.$$

Ainsi, le problème se réduit au problème du couplage de poids maximum. ■

L'algorithme suivant résout le problème $F2|chain{-}reentrant, a_j > \frac{L}{2}, c_j > \frac{L}{2}, l_j = L|C_{max}$ en temps polynomial.

Algorithme $\mathcal{A}1$ pour le problème P_{22}

1. A partir du graphe $G = (V, E)$ construire un nouveau graphe pondéré $H = (V, E)$ où chaque arête de E possède un poids égal à :
$$\max \left\{ \begin{array}{l} \min\{L + c_{s_k}, L + a_{t_k}, 2L + c_{s_k} + a_{t_k} - b_{s_k} - b_{t_k}\}, \\ \min\{L + c_{t_k}, L + a_{s_k}, 2L + a_{s_k} + c_{t_k} - b_{s_k} - b_{t_k}\} \end{array} \right\}$$
si l'arête est incidente aux sommets s_k et t_k.
2. Trouver le couplage du poids maximum M du graphe H.
3. Pour chaque paire de M, entrelacer les deux tâches correspondantes dans un même batch.
4. Mettre chaque tâche restante dans un seul batch.
5. Ordonnancer les batchs formés dans un ordre quelconque sans temps mort.

Corollaire 4.1. *L'algorithme A1 résout le problème* $F2|chain - reentrant, a_j > \frac{L}{2}, c_j > \frac{L}{2}, l_j = L|C_{max}$ *en* $O(n^{2.5})$.

Preuve. L'optimalité est garantie par le théorème 2. La complexité de l'algorithme $A1$ est

donnée par la deuxième étape 2, qui fournit un couplage de poids maximum en $O(n^{2.5})$, [22]. Donc la complexité de l'algorithme $A1$ est $O(n^{2.5})$. ■

Exemple 4.1. *Nous disposons de* 5 *tâches* $T_1, ..., T_5$. *Les durées de traitement de ces tâches sur les deux machines sont données dans la table 4.3 avec* $L = 4$

TABLE 4.3 – Durée de traitement des tâches

T_i	T_1	T_2	T_3	T_4	T_5
a_i	2	3	5	2	5
b_i	2	4	3	4	3
c_i	5	2	2	5	3

1- On a les tâches entre parenthèses $(1,2)$, $(1,3)$, $(1,4)$, $(1,5)$, $(2,3)$, $(2,5)$, $(3,5)$, $(4,1)$, $(4,2)$, $(4,3)$, $(4,5)$, $(5,3)$, *ne peuvent pas être entrelacées dans un même batch, donc les arcs correspondants ne sont pas formés (car la condition d'entrelacement n'est pas vérifiée). Ainsi le graphe* $G = (V, E)$ *contient 5 sommets (nombre de tâches) et 8 arêtes :* $\{(2,1),(2,4),(3,1),(3,2),(3,4),(5,1),(5,2),(5,4)\}$

2- Construction du graphe H *et évaluation des arêtes (voir Figure 4.5) :*

$d(2,1) = \min\{4+2, 4+2, 8+4-6\} = 6.$

$d(2,4) = \min\{4+2, 4+2, 8+4-8\} = 4.$

$d(3,1) = \min\{4+2, 4+2, 8+4-6\} = 6.$

$d(3,2) = \min\{4+2, 4+3, 8+5-7\} = 6.$

$d(3,4) = \min\{4+2, 4+2, 8+4-7\} = 5.$

$d(5,1) = \min\{4+3, 4+2, 8+5-5\} = 6.$

$d(5,2) = \min\{4+3, 4+3, 8+6-7\} = 7.$

$d(5,4) = \min\{4+3, 4+2, 8+5-7\} = 6.$

3- Le couplage de poids maximum est : $M = \{(5,2),(3,1)\}$.

4- La séquence optimale est : $(5,2),(3,1),4$ *et le temps de fin de traitement est égal à* 41 *(voir Figure 4.6).*

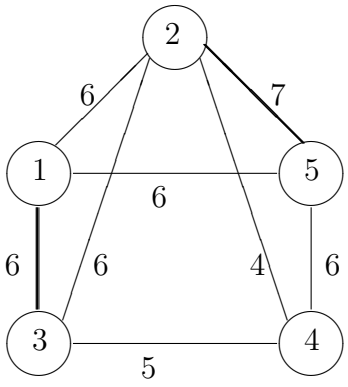


FIGURE 4.5 – Graphe valué H

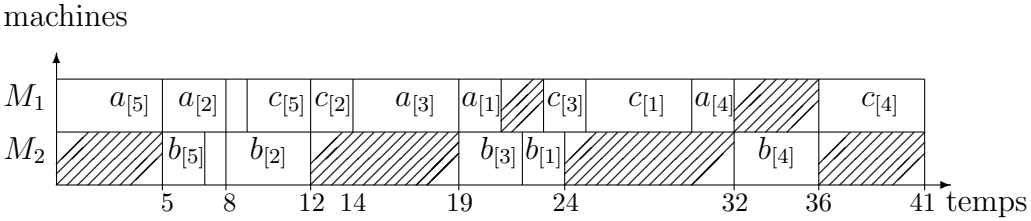


FIGURE 4.6 – La solution obtenue en utilisant l'algorithme $\mathcal{A}1$

4.2.2 Problème $P_{23} : F2|ChR, a_j = \frac{L}{2}, b_i + b_j \leq L, c_j \leq \frac{L}{2}, l_j = L|C_{max}$

Dans la première section, nous avons prouvé la NP-complétude du problème $P_{21} : F2|ChR, a_j = \frac{L}{2}, c_j = \frac{L}{2}, l_j = L|C_{max}$ ceci implique que le problème $F2|ChR, a_j = \frac{L}{2}, c_j \leq \frac{L}{2}, l_j = L|C_{max}$ est aussi NP-difficile au sens fort. Dans ce qui suit, nous montrons que ce dernier devient polynomial lorsqu'on lui rajoute la condition $b_i + b_j \leq L$.

Théorème 4.3. *Le problème $P_{23} : F2|ChR, a_j = \frac{L}{2}, b_i + b_j \leq L, c_j \leq \frac{L}{2}, l_j = L|C_{max}$ est résoluble en temps polynomial par l'algorithme $\mathcal{A}2$.*

La complexité de l'algorithme $\mathcal{A}2$ est de $O(nlogn)$. En effet, la première étape de l'algorithme $\mathcal{A}2$ est une opération de tri de complexité $O(nlogn)$ et la deuxième étape de l'algorithme $\mathcal{A}2$ consiste à former des batches de cardinalité 3 qui est de complexité $O(n)$.

Algorithme $\mathcal{A}2$ pour le problème P_{22}

Début

1. Ranger des tâches T par ordre croissant des c_i, soit $T_1, T_2, ..., T_i, ..., T_n$ cet ordre.
2. **Pour** $i := 1$ à $\lfloor \frac{n}{3} \rfloor$ **Faire**
   - Entrelacer la tâche T_i avec les deux tâches T_{n-2i+2} et T_{n-2i+1} afin de former un batch de cardinalité 3. (voir Figure 4.7)

Fait

Si $|T| = \emptyset$ alors **stop**.

Sinon $(1 \leq |T| \leq 2)$

Dans ce cas le batch formé a une cardinalité égale à 2 ou à 1.

3. Ordonnancer les batchs selon un ordre quelconque sans temps mort.

Fin.

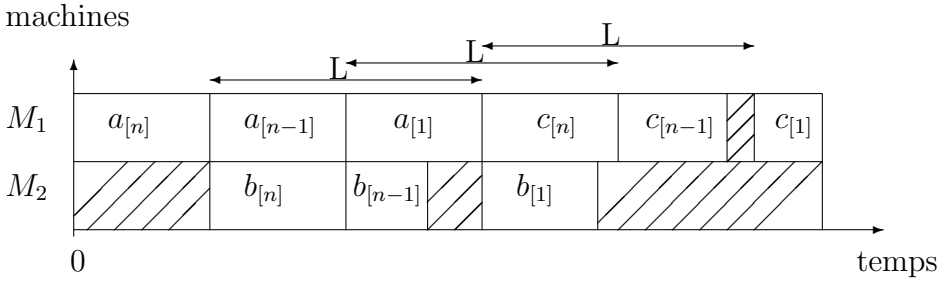


FIGURE 4.7 – Exemple d'un batch formé de 3 tâches entrelacées et qui se termine avec le plus petit c_i.

Preuve. On a $\forall j,\ a_j = \frac{L}{2}, c_j \leq \frac{L}{2}, l_j = L$ alors les tâches peuvent étre ordonnancées dans des batchs de cardinalité au plus trois. Pour n'importe quelle solution réalisable σ du problème P_4, il existe une solution σ' formée seulement des batchs de cardinalité trois telle que $C_{max}(\sigma') \leq C_{max}(\sigma)$.

Notons que :

– Le temps de traitement d'un batch $\{T_i, T_j, T_k\}_B$ de cardinalité 3 est égal à $\frac{L}{2} + L + L + c_k = \frac{5L}{2} + c_k$.

– Le temps de traitement d'un batch $\{T_i, T_j\}_B$ de cardinalité 2 est égal à $2L + c_j$.

– Le temps de traitement d'un batch contenant une seule tâche $\{T_i\}_B$ est égal à $\frac{3L}{2} + c_j$.

Soit $\alpha = \lfloor \frac{n}{3} \rfloor$ le nombre maximum de batchs de cardinalité trois qu'on peut former. La date de fin de traitement C_{max} de l'ensemble des batchs est donc égale à :

$$\begin{cases} -C_{max} = \alpha(\frac{5L}{2}) + \sum_{k=1}^{\alpha} c_k, \text{ si reste de } \frac{n}{3} = 0 \\ -C_{max} = \alpha(\frac{5L}{2}) + 2L + \sum_{k=1}^{\alpha+1} c_k, \text{ si reste de } \frac{n}{3} = 2 \\ -C_{max} = \alpha(\frac{5L}{2}) + \frac{3L}{2} + \sum_{k=1}^{\alpha+1} c_k, \text{ si reste de } \frac{n}{3} = 1 \end{cases}$$

Comme l'ensemble des tâches T est rangé par ordre croissant des c_k, donc chaque batch

se termine avec la tâche qui a le plus petit c_k, ce qui permet d'obtenir la solution optimale du problème. Minimiser $\sum_{k=1}^{\alpha} c_k$ revient à prendre les α premières valeurs des c_k. ■

Exemple 4.2. *Nous disposons de* 5 *tâches* $T_1, ..., T_5$ *Les durées de traitement de ces tâches sur les deux machines sont données dans la Table suivante et le time lag* $L = 6$.

TABLE 4.4 – Durée de traitement des 5 tâches

T_i	T_1	T_2	T_3	T_4	T_5
a_i	3	3	3	3	3
b_i	2	1	4	2	2
c_i	3	1	3	2	1

La première étape de l'algorithme A2 consiste à ordonancer les tâches selon l'ordre croissant des c_i. *donc l'ordre dans le cas de cet example est :* T_2, T_5, T_4, T_1, T_3. *La solution optimale* S^* *est donnée par des triplets ou bien des batchs de cardinalité trois sauf le dernier batch qui contient seulement les tâches restantes.* $S^* = (\{T_3, T_1, T_2\}, \{T_4, T_5\})$. *Le temps de traitement du batch* $\{T_3, T_1, T_2\}_B$ *est égal à* $\frac{5L}{2} + c_2 = 15 + 1 = 16$ *le temps de traitement du batch* $\{T_4, T_5\}_B$ *est égal à* $2L + c_5 = 12 + 1 = 13$, *donc le temps de fin de traitement de l'ensemble des tâches est égal à* $16 + 13 = 29$ *(voir Figure 4.8).*

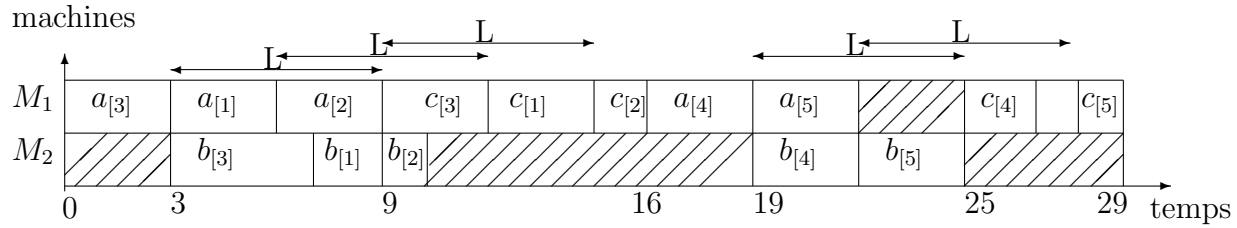


FIGURE 4.8 – La solution optimale de l'exemple 4.2

4.2.3 Problèmes P_{24}, P_{25}, P_{26}

Proposition 4.1. *Les problèmes suivants sont résolubles en des temps polynomiaux :*
$P_{24} : F2|ChR, a_j > L, l_j = L|C_{max}$.
$P_{25} : F2|ChR, c_j > L, l_j = L|C_{max}$.
$P_{26} : F2|ChR, a_j = b_j = c_j = c, l_j = L|C_{max}$.

Preuve. Pour le problème P_{24}, nous avons $a_j > L$, $\forall j$, donc les tâches ne peuvent pas être entrelacées entre elles car $l_j = L$ (la condition d'entrelacement n'est pas vérifiée). On en déduit que la solution optimale σ est obtenue en ordonnançant les tâches l'une après l'autre sans temps mort et en prenant n'importe quel ordre de tâches. La date de fin de

traitement de l'ensemble des tâches est égale à $C_{max} = nL + \sum_{i=1}^{n}(a_i + c_i)$. c'est la même chose pour le problème P_{25} puisque c'est le cas symétrique.
Comme tous les temps de traitement sont identiques $a_j = b_j = c_j = c$ pour le problème P_{26}, alors, la solution optimale est obtenue en ordonnançant les tâches dans des batchs de cardinalité $\lfloor \frac{L}{c} \rfloor + 1$, sauf le dernier batch qui contient les tâches restantes. Bien sûr on suppose que $c \leq L$ car dans le cas où $c > L$ on tombe sur le problème P_{25}. Le temps de traitement de chaque batch formé est égal à $c(2 + \lfloor \frac{L}{c} \rfloor) + L$ ■

Chapitre 5

Résolution du problème $F2|ChR, l_j = L|C_{max}$

Dans ce chapitre, nous proposons des heuristiques de type "algorithme de liste" pour le problème $P_2 : F2|ChR, l_j = L|C_{max}$. D'abord, nous donnons quelques bornes inférieures de la valeur optimale du makespan.

5.1 Bornes inférieures

Nous proposons dans ce qui suit trois bornes inférieures pour le problème P_2 : LB_1, LB_2 et LB_3. On commence d'abord par partitionner l'ensemble des tâches T en deux sous ensembles A et B tels que $A = \{T_j / \min\{a_j, c_j\} > L\}$ et $B = T \setminus A$.

Proposition 5.1. $LB_1 = \sum_{j=1}^{n}(a_j + c_j)$ *est une borne inférieure.*
$LB_2 = \min_{1\leq j\leq n}\{a_j\} + \sum_{j=1}^{n} b_j + \min_{1\leq j\leq n}\{c_j\}$ *est une borne inférieure.*
$LB_3 = |A|\, L + \sum_{j/T_j\in A}(a_j + c_j) + \min_{i/T_j\in B}\{a_j\} + \sum_{j/T_i\in B} b_j + \min_{i/T_j\in B}\{c_j\}$ *est une borne inférieure.*
$LB = max\{LB_1, LB_2, LB_3\}$ *est une borne inférieure.*

Notons que ces bornes inférieures sont utilisées afin d'évaluer la performance des heuristiques proposées.

Preuve. Il est claire que la somme des temps de traitement sur la première machine ($i.e \sum_{j=1}^{n}(a_j + c_j)$) nous donne une borne inférieure. La somme des temps de traitement sur

la deuxième machine à laquelle on ajoute $\min\limits_{1\leq j\leq n}\{a_j\} + \min\limits_{1\leq j\leq n}\{c_j\}$ est aussi une borne inférieure car chaque ordonnancement doit forcément commencer par une opération du type $a_{[j]}$ et se terminer avec une opération de type $c_{[j]}$. Pour la borne LB_3 nous avons déjà vu que la condition d'entrelacement n'est pas vérifiée pour les tâches qui ont $\min\{a_j, c_j\} > L$, donc ces tâches sont ordonnancées seules. par conséquent, on ajoute la valeur L au calcul de la borne inférieure pour ce type de tâches. ■

5.2 Heuristique d'entrelacement HE

Pour résoudre le problème P_2 : $F2|ChR, l_j = L|C_{max}$ nous proposons neuf méthodes approchées. Leur principe est d'appliquer l'heuristique d'entrelacement que nous allons définir par la suite avec les règles de priorité suivantes :

- L_1 : Les tâches sont rangées selon l'ordre croissant des a_j.
- L_2 : Les tâches sont rangées selon l'ordre croissant des c_j.
- L_3 : Les tâches sont rangées selon l'ordre croissant des $a_j + c_j$.
- L_4 : Les tâches sont rangées selon l'ordre décroissant des a_j.
- L_5 : Les tâches sont rangées selon l'ordre décroissant des c_j.
- L_6 : Les tâches sont rangées selon l'ordre décroissant des $a_j + c_j$.
- L_7 : Les tâches sont rangées selon l'ordre croissant des b_j.
- L_8 : Les tâches sont rangées selon l'ordre décroissant des b_j.
- L_{RP} : Dans ce dernier cas on génère plusieurs listes aléatoirement, (c-à-d. On génère une population de listes, dans chaque liste les tâches sont rangées dans un ordre aléatoire). La taille de la population générée est égale au nombre de tâches.

Le principe de l'heuristique d'entrelacement est comme suit :
Tant que l'ensemble des tâches n'est pas vide :
- Former un nouveau batch en choisissant de la liste L_i la première tâche disponible et l'entrelacer avec un nombre maximum de tâches.
- Calculer la durée de traitement du batch notée P_{batch}.
- Le temps de fin de traitement de l'ensemble des tâches est égal à la somme des temps de traitement des batchs formés.

Heuristique HE_{L_i}

Début

1. Initialiser le nombre de batchs à zéro $b := 0$;
2. De la liste L_i choisir la première tâche disponible T_j.
3. **Tant que** $T \neq \emptyset$ **faire**
   - De la liste L_i choisir une autre tâche disponible T_k afin de l'entrelacer avec T_j.
   Tant que la condition d'entrelacement est vérifiée pour la tâche T_k **faire**
   - Entrelacer T_j avec T_k comme dans la Figure 5.1.
   - $T = T \setminus T_k$.
   - Choisir de la liste L_i une nouvelle tâche T_k non entrelacée.
   Fin tant que
   - Calculer la durée de traitement P_{batch_b} du batch b formé , $b := b + 1$. $T = T \setminus T_i$.

   Fin tant que
4. Ordonnancer tous les batchs dans n'importe quel ordre.
5. Le temps de traitement de l'ensemble des tâches est égal à la somme des durées de traitement des batchs formés.

Fin.

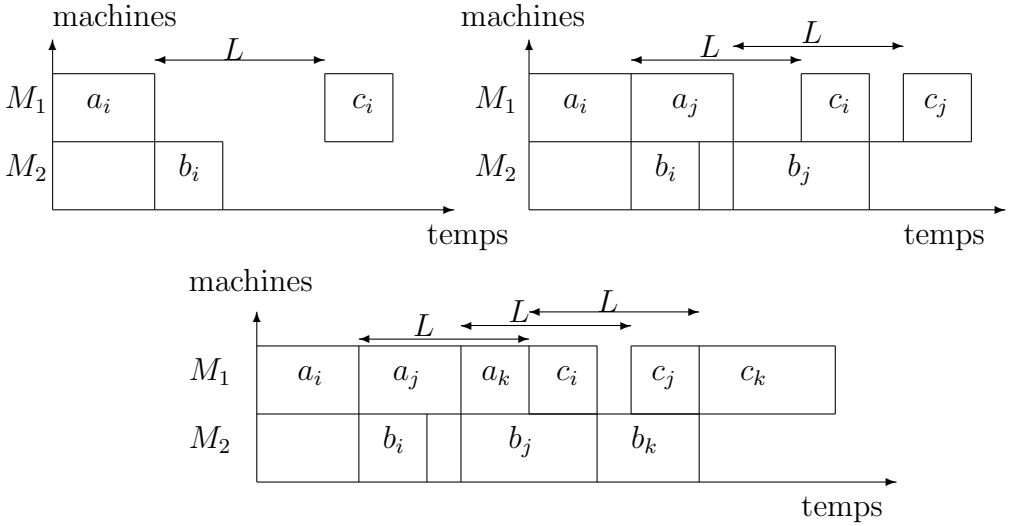


FIGURE 5.1 – Exemple d'entrelacement de 3 tâches T_i, T_j, T_k dans un même batch.

La complexité de ces heuristiques est de $O(nlogn)$. En effet, la première étape consiste à ranger les taches selon une liste donnée, c'est une opération de tri de complexité $O(nlogn)$ et la deuxième étape est une application de la technique d'entrelacement qui est de complexité $O(n)$.

5.3 Expérimentations numériques

Pour évaluer l'efficacité des heuristiques, nous avons effectué des expérimentations numériques. Pour cela, nous avons considéré trois types d'instances distinctes.

Le premier type est caractérisé par des temps d'exécution sur la première machine égaux à la moitié des temps de latence ($a_j = c_j = \frac{L}{2}$) et des temps d'exécution sur la deuxième machine inférieurs au time lag ($b_j \leq l_j = L$) ; comme ce dernier a été prouvé d'être NP-difficile au sens fort ceci justifie notre choix. Pour ce premier type d'instance nous n'avons considéré que les listes L_7, L_8 et L_{RP} car les autre listes donnent le même ordre, vu que les temps de traitement sur la première machine sont identiques (voir Table 5.2).

Le second type est caractérisé par des petits temps d'exécution sur la première machine inférieurs au temps de latence, dans ce cas les temps d'exécution sont générés aléatoirement comme suit : (i) $a_j \in [1, 20]$, $c_j \in [1, 30]$, $b_j \leq l_j = L \in \{30, 70\}$, (ii) $a_j \in [1, 30]$, $c_j \in [1, 50]$, $b_j \leq l_j = L \in \{50, 100\}$.

Dans le troisième type, nous générons de grands temps d'exécution. Les temps d'exécution des tâches sont générés de façon aléatoire suivant une loi uniforme comme suit : $a_j \in [20, 80]$, $c_j \in [10, 60]$, $b_j \leq l_j = L = 70$, (ii) $a_j \in [20, 80]$, $c_j \in [10, 60]$, $b_j \leq l_j = L = 100$.

Nous avons généré 100 instances aléatoires pour chaque problème de taille $n = 20$, $n = 50$, $n = 100$, $n = 500$ et $n = 1000$.

La performance des heuristiques basée sur les différentes règles développées dans la section précédente a été comparé à la borne inférieure. Pour chaque instance résolue, nous reportons le nombre de fois où une heuristique H fournit de meilleures solutions par rapport aux autres notées #Best, et le nombre de fois où elle coïncide avec la borne inférieure notée #LB. La performance d'une heuristique donnée est mesurée par l'écart relatif entre la valeur de la solution obtenue C_{max} et la borne inférieure LB ($Dev = \frac{Cmax(H)-LB}{LB}$). Pour chaque type d'instances nous présentons les déviations moyennes et maximales (Devmoy) et (Devmax), respectivement et nous présentons les performances des meilleures heuristiques.

D'après les expérimentations numériques, nous avons observé que les heuristiques proposées donnent en général de bonnes solutions. Quand les temps d'exécution sur la première machine prennent des valeurs égales à la moitié des temps de latence ($a_j = c_j = \frac{L}{2}$) et des temps d'exécution sur la deuxième machine inférieurs au time lag ($b_j \leq l_j = L$) nous avons remarqué que les solutions sont proches de l'optimum avec une déviation moyenne inférieure à 0.1 pour la plupart des instances, ceci est obtenu lorsque le temps de latence est petit ($L = 70$). Nous avons aussi constaté que la liste L_{RP} est la plus performante. Par ailleurs, plus le temps de latence augmente ($L = 100$), plus les solutions deviennent moins bonnes mais restent toujours qualifiées comme de bonnes solutions avec une déviation moyenne inférieure à 0.15. Dans ce cas, nous avons observé que la liste L_8 basée sur le

TABLE 5.1 – $a_j = c_j = \frac{L}{2}$

		$L = 70 \quad b_i \leq L$			$L = 100 \quad b_i \leq L$		
		IHL_7	IHL_8	IHL_{RP}	IHL_7	IHL_8	IHL_{RP}
$n = 20$	*Best*	66	94	100	15	88	100
	opt	0	0	0	0	0	0
	$aver_{dev}$	0.095	0.056	0.056	0.237	0.157	0.155
	max_{dev}	0.140	0.044	0.025	0.366	0.366	0.366
$n = 50$	*Best*	19	100	100	5	100	100
	opt	0	0	0	0	0	0
	$aver_{dev}$	0.041	0.010	0.010	0.195	0.141	0.141
	max_{dev}	0.147	0.149	0.147	0.401	0.401	0.401
$n = 100$	*Best*	14	97	100	5	100	100
	opt	0	0	0	0	0	0
	$aver_{dev}$	0.077	0.034	0.033	0.191	0.142	0.142
	max_{dev}	0.148	0.150	0.1485	0.436	0.419	0.419
$n = 500$	*Best*	12	95	100	5	100	100
	opt	0	0	0	0	0	0
	$aver_{dev}$	0.079	0.026	0.025	0.196	0.133	0.133
	max_{dev}	0.151	0.149	0.149	0.422	0.422	0.422
$n = 500$	*Best*	12	95	100	5	100	49
	opt	0	0	0	0	0	0
	$aver_{dev}$	0.080	0.026	0.022	0.197	0.133	0.134
	max_{dev}	0.145	0.149	0.142	0.423	0.421	0.423

tri des tâches selon l'ordre croissant des b_j et la liste L_{RP} qui génère plusieurs listes aléatoirement, sont les plus performantes et donnent presque les mêmes résultats. En ce qui concerne le second type d'instances, c.-à-d. le cas où les temps d'exécution sur la première machine sont petits et inférieurs au temps de latence, nous avons observé que la liste L_{RP} domine les autres listes en terme de performance, nous avons remarqué que les solutions obtenues sont d'une qualité moyenne comparées aux solutions obtenues pour le premier type d'instance avec une déviation moyenne qui tourne au tour de 0.4 (voir Table 5.2 et Table 5.4) à notre avis ceci est peut-être dû aux qualités des bornes inférieures. D'autre part, les solutions s'améliorent lorsque le temps de latence augmente (voir Table 5.3 et Table 5.5) ceci est justifié car les batchs formées dans ce cas contiennent plus de tâches, donc moins de temps mort, ainsi les solutions obtenues sont plus proches des bornes inférieures. Pour le dernier type d'instance où nous avons généré de grands temps d'exécution, nous avons constaté que la liste L_6 est la plus performante, ceci est bien illustré dans les deux diagrammes ci-dessous (voir Figure 5.2).

TABLE 5.2 – $a_j \in [1, 20], c_j \in [1, 30]$

				$L = 30 \quad b_j <= L$					
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	12	5	7	12	14	29	8	11	87
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.40	0.45	0.43	0.40	0.38	0.36	0.41	0.40	0.33
max_{dev}	0.75	0.87	0.87	0.75	0.75	0.75	0.81	0.75	0.75
$n = 50$ *Best*	6	9	11	8	4	14	5	4	78
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.38	0.43	0.39	0.38	0.36	0.35	0.39	0.38	0.32
max_{dev}	0.67	0.75	0.75	0.64	0.65	0.65	0.69	0.71	0.60
$n = 100$ *Best*	3	4	2	4	4	35	7	5	69
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.38	0.44	0.40	0.38	0.38	0.35	0.39	0.39	0.33
max_{dev}	0.64	0.84	0.84	0.63	0.70	0.70	0.68	0.71	0.59
$n = 500$ *Best*	3	2	4	3	2	27	2	2	74
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.35	0.42	0.38	0.34	0.36	0.33	0.37	0.35	0.31
max_{dev}	0.62	0.86	0.80	0.57	0.67	0.67	0.65	0.66	0.56
$n = 1000$ *Best*	2	2	3	2	2	29	2	2	72
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.35	0.42	0.39	0.34	0.36	0.34	0.38	0.35	0.31
max_{dev}	0.62	0.81	0.81	0.57	0.68	0.68	0.65	0.64	0.56

TABLE 5.3 – $a_j \in [1, 20], c_j \in [1, 30]$

				$L = 70 \quad b_j <= L$					
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	9	3	2	1	4	0	21	3	63
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.16	0.18	0.17	0.21	0.16	0.18	0.23	0.25	0.12
max_{dev}	0.39	0.36	0.38	0.48	0.36	0.60	0.48	0.67	0.27
$n = 50$ *Best*	17	8	11	6	6	8	1	2	59
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.12	0.13	0.13	0.13	0.14	0.15	0.25	0.20	0.10
max_{dev}	0.22	0.26	0.26	0.28	0.30	0.29	0.45	0.34	0.20
$n = 100$ *Best*	14	1	3	6	3	2	0	2	77
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.10	0.11	0.10	0.11	0.11	0.12	0.18	0.15	0.09
max_{dev}	0.17	0.21	0.20	0.22	0.22	0.22	0.36	0.26	0.17
$n = 500$ *Best*	18	5	8	3	2	0	0	1	64
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.10	0.10	0.10	0.10	0.10	0.11	0.20	0.17	0.09
max_{dev}	0.17	0.19	0.19	0.19	0.21	0.20	0.36	0.27	0.18
$n = 1000$ *Best*	1	1	3	4	1	5	0	1	90
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.10	0.10	0.10	0.10	0.11	0.11	0.21	0.17	0.09
max_{dev}	0.19	0.21	0.21	0.20	0.23	0.22	0.39	0.29	0.17

TABLE 5.4 – $a_j \in [1, 30], c_j \in [1, 50]$

				$L = 50 \quad b_j <= L$					
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	2	2	4	2	20	22	2	4	70
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.38	0.44	0.42	0.40	0.34	0.35	0.41	0.39	0.33
max_{dev}	0.89	89	0.89	0.89	0.85	0.85	0.85	0.89	0.85
$n = 50$ *Best*	10	0	10	4	2	6	2	0	90
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.39	0.45	0.42	0.40	0.37	0.37	0.42	0.41	0.33
max_{dev}	0.70	0.81	0.81	0.70	0.70	0.70	0.70	0.80	0.70
$n = 100$ *Best*	2	0	0	0	0	0	0	0	98
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.38	0.49	0.45	0.41	0.40	0.38	0.45	0.41	0.34
max_{dev}	0.71	0.86	0.86	0.71	0.75	0.75	0.73	0.74	0.67
$n = 500$ *Best*	0	0	0	0	0	0	0	0	100
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.36	0.47	0.42	0.37	0.38	0.37	0.43	0.39	0.33
max_{dev}	0.66	0.85	0.85	0.66	0.73	0.73	0.68	0.70	0.64
$n = 1000$ *Best*	0	0	0	0	0	0	0	0	100
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.36	0.48	0.43	0.36	0.39	0.37	0.43	0.39	0.33
max_{dev}	0.6	0.85	0.85	0.64	0.74	0.74	0.70	0.70	0.64

TABLE 5.5 – $a_j \in [1, 30], c_j \in [1, 50]$

				$L = 100 \quad b_j <= L$					
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	9	45	8	0	7	0	14	0	69
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.23	0.24	0.22	0.29	0.22	0.24	0.34	0.32	0.19
max_{dev}	0.38	0.42	0.35	0.45	0.35	0.37	0.55	0.50	0.30
$n = 50$ *Best*	18	2	9	3	2	2	0	0	68
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.18	0.23	0.21	0.20	0.23	0.25	0.37	0.19	0.16
max_{dev}	0.30	0.35	0.36	0.38	0.38	0.41	0.55	0.48	0.27
$n = 100$ *Best*	17	0	3	5	2	2	0	0	73
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.14	0.16	0.16	0.15	0.19	0.19	0.28	0.22	0.13
max_{dev}	0.23	0.26	0.25	0.26	0.29	0.31	0.42	0.33	0.21
$n = 500$ *Best*	31	0	0	8	0	10	0	0	64
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.14	0.16	0.16	0.15	0.17	0.17	0.30	0.24	0.14
max_{dev}	0.21	0.26	0.25	0.25	0.27	0.27	0.43	0.34	0.21
$n = 1000$ *Best*	0	0	0	0	0	0	0	0	100
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.14	0.18	0.17	0.15	0.19	0.19	0.31	0.26	0.13
max_{dev}	0.22	0.28	0.27	0.24	0.30	0.30	0.46	0.36	0.21

TABLE 5.6 – $a_j \in [20, 80]\,, c_j \in [10, 60]$

				$L = 70 \quad b_j <= L$					
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	0	0	0	20	0	36	8	2	38
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.34	0.25	0.32	0.23	0.28	0.23	0.27	0.27	0.23
max_{dev}	0.41	0.31	0.37	0.32	0.34	0.32	0.41	0.34	0.29
$n = 50$ *Best*	0	0	0	0	0	56	2	4	38
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.28	0.21	0.24	0.20	0.22	0.18	0.21	0.21	0.18
max_{dev}	0.31	0.26	0.30	0.26	0.28	0.26	0.26	0.28	0.22
$n = 100$ *Best*	0	0	0	16	0	68	0	0	16
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.28	0.22	0.23	0.20	0.22	0.18	0.22	0.22	0.20
max_{dev}	0.31	0.25	0.27	0.24	0.25	0.25	0.25	0.27	0.25
$n = 500$ *Best*	0	0	0	0	0	98	0	0	2
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.28	0.22	0.22	0.20	0.21	0.17	0.21	0.20	0.19
max_{dev}	0.30	0.24	0.25	0.24	0.24	0.23	0.23	0.24	0.23
$n = 1000$ *Best*	0	0	0	0	0	100	0	0	0
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.28	0.22	0.21	0.19	0.21	0.17	0.21	0.20	0.19
max_{dev}	0.29	0.24	0.25	0.23	0.24	0.23	0.23	0.24	0.23

TABLE 5.7 – $a_j \in [20, 80]\,, c_j \in [10, 60]$

				$L = 100 \quad b_j <= L$					
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	0	3	0	15	2	17	0	0	63
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.35	0.27	0.33	0.25	0.30	0.25	0.33	0.30	0.23
max_{dev}	0.41	0.41	0.40	0.40	0.37	0.40	0.42	0.40	0.36
$n = 50$ *Best*	0	4	4	0	0	36	0	0	56
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.29	0.23	0.25	0.24	0.24	0.21	0.30	0.25	0.20
max_{dev}	0.37	0.41	0.38	0.38	0.39	0.38	0.40	0.40	0.36
$n = 100$ *Best*	0	6	1	10	0	40	0	0	43
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.30	0.24	0.26	0.23	0.24	0.22	0.32	0.28	0.21
max_{dev}	0.41	0.40	0.40	0.41	0.40	0.40	0.40	0.40	0.37
$n = 500$ *Best*	0	0	1	0	1	85	0	0	13
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.29	0.23	0.24	0.22	0.23	0.20	0.31	0.25	0.21
max_{dev}	0.41	0.41	0.40	0.40	0.40	0.41	0.40	0.40	0.40
$n = 1000$ *Best*	0	0	0	0	0	91	0	0	9
opt	0	0	0	0	0	0	0	0	3
$aver_{dev}$	0.29	0.23	0.23	0.21	0.22	0.20	0.31	0.25	0.21
max_{dev}	0.41	0.41	0.41	0.41	0.40	0.40	0.40	0.40	0.40

TABLE 5.8 – $a_j \in [10, 30]\,, c_j \in [20, 50]$

					$L = 50 \quad b_j <= L$				
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	37	3	11	16	14	14	4	18	47
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.29	0.35	0.32	0.28	0.29	0.30	0.32	0.28	0.27
max_{dev}	0.40	0.48	0.48	0.402	0.401	0.40	0.42	0.39	0.39
$n = 50$ *Best*	11	2	2	6	29	17	9	5	64
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.30	0.35	0.34	0.32	0.30	0.301	0.31	0.31	0.29
max_{dev}	0.41	0.45	0.45	0.42	0.43	0.41	0.40	0.41	0.39
$n = 100$ *Best*	0	6	1	10	0	40	0	0	43
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.28	0.35	0.33	0.29	0.29	0.29	0.30	0.29	0.28
max_{dev}	0.41	0.40	0.40	0.41	0.40	0.40	0.40	0.40	0.37
$n = 500$ *Best*	5	2	2	3	15	50	4	9	42
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.29	0.34	0.33	0.29	0.29	0.28	0.30	0.29	0.28
max_{dev}	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
$n = 1000$ *Best*	12	2	2	12	8	57	7	5	42
opt	0	0	0	0	0	0	0	0	3
$aver_{dev}$	0.29	0.34	0.33	0.29	0.29	0.28	0.30	0.29	0.29
max_{dev}	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

TABLE 5.9 – $a_j \in [20, 50]\,, c_j \in [10, 50]$

					$L = 50 \quad b_j <= L$				
	IHL_1	IHL_2	IHL_3	IHL_4	IHL_5	IHL_6	IHL_7	IHL_8	IHL_{RP}
$n = 20$ *Best*	0	10	8	18	10	18	39	10	46
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.37	0.34	0.36	0.32	0.32	0.32	0.31	0.33	0.31
max_{dev}	0.48	0.41	0.48	0.41	0.39	0.40	0.39	0.41	0.40
$n = 50$ *Best*	0	9	0	19	11	22	13	6	52
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.33	0.28	0.32	0.27	0.28	0.27	0.29	0.28	0.26
max_{dev}	0.43	0.35	0.43	0.33	0.35	0.33	0.38	0.36	0.33
$n = 100$ *Best*	0	7	0	17	20	24	11	14	54
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.33	0.28	0.32	0.27	0.28	0.27	0.29	0.28	0.26
max_{dev}	0.43	0.35	0.43	0.33	0.35	0.33	0.38	0.36	0.33
$n = 500$ *Best*	0	10	0	9	4	49	13	5	31
opt	0	0	0	0	0	0	0	0	0
$aver_{dev}$	0.33	0.26	0.31	0.27	0.27	0.26	0.28	0.27	0.26
max_{dev}	0.42	0.33	0.42	0.32	0.33	0.32	0.35	0.33	0.32
$n = 1000$ *Best*	0	0	0	0	0	91	0	0	9
opt	0	0	0	0	0	0	0	0	3
$aver_{dev}$	0.33	0.26	0.31	0.26	0.26	0.26	0.28	0.26	0.25
max_{dev}	0.42	0.32	0.42	0.432	0.32	0.32	0.34	0.33	0.31

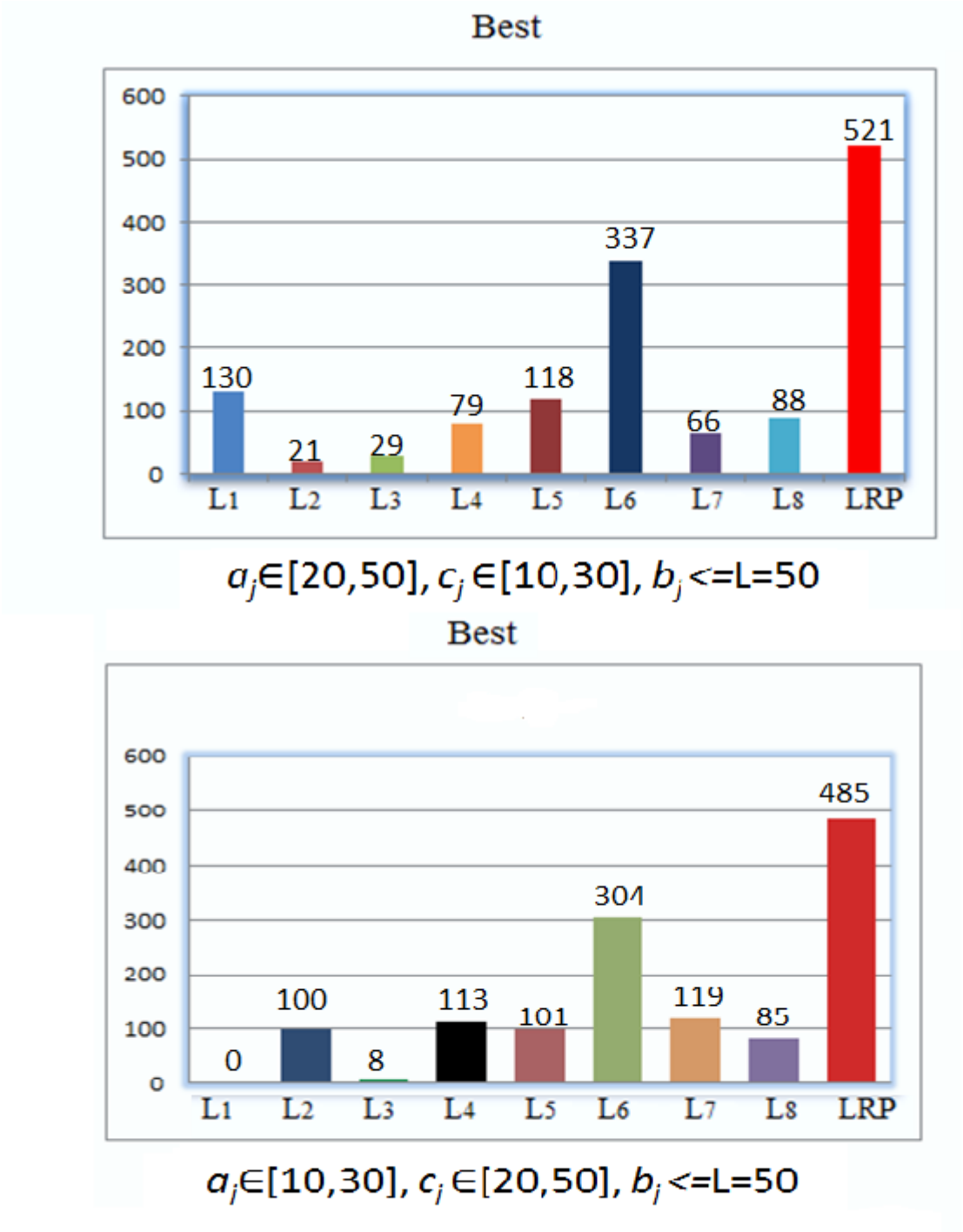


FIGURE 5.2 – Le nombre de fois où une heuristique Fournit les meilleurs solutions

Chapitre 6

Analyse de complexité du Problème $P_3 : F2|ChR, b_j = l_j|C_{max}$

Dans ce chapitre, nous considérons le cas où les temps de traitement sur la deuxième machine sont égaux aux temps de latence $b_j = l_j$, ce problème est noté $P_3 : F2|ChR, b_j = l_j|C_{max}$, et peut être vu comme un problème d'ordonnancement avec recirculation auquel on ajoute la contrainte de sans attente (no-wait constraint en Anglais) $F2|ChR, no - wait|C_{max}$. On commence par donner un exemple illustratif de ce problème, ensuite nous démontrons sa NP-complétude, enfin nous proposons plusieurs sous problèmes résolubles en des temps polynomiaux. La liste des problèmes traités est donné à la Table 6.1 ci-dessous.

TABLE 6.1 – Liste des problèmes étudiés

Problème	Complexité
$P_3 : F2\|ChR, b_j = l_j\|C_{max}$	$NP - difficile$
$P_{31} : F2\|ChR, b_j = l_j = L, a_i + c_j > L\|C_{max}$	$Polynmial$, $O(n^{2.5})$
$P_{32} : F2\|ChR, b_j = l_j = L, a_i + c_j \leq L\|C_{max}$	$Polynmial$, $O(nlogn)$
$P_{33} : F2\|ChR, a_j = b_j = l_j = L\|C_{max}$	$Polynmial$, $O(nlogn)$
$P_{34} : F2\|ChR, b_j = c_j = l_j = L\|C_{max}$	$Polynmial$, $O(nlogn)$

6.1 Caractérisation de la solution optimale

Comme il a été déjà introduit au début de ce chapitre, le problème $P_3 : F2|ChR, b_j = l_j|C_{max}$ est équivalent au problème $F2|ChR, no - wait|C_{max}$. On suppose que les temps

de traitement ne peuvent pas être nuls. Le schéma d'exécution d'une tâche est illustré par la figure suivante.

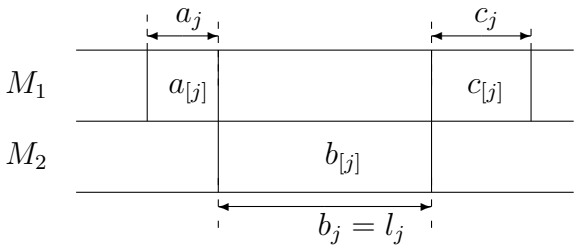


FIGURE 6.1 – Schéma d'exécution d'une tâche dans le problème P_3.

Exemple illustratif

On considère l'instance suivante : 5 tâches T_1, T_2, T_3, T_4 et T_5 à ordonnancer sur 2 machines. Les temps de traitement sont donnés dans la table 6.2, avec $L = 3$.

TABLE 6.2 – Temps de traitement des tâches

T_i	T_1	T_2	T_3	T_4	T_5
a_j	1	2	1	1	1
$b_j = l_j$	3	3	4	2	1
c_j	2	2	2	2	1

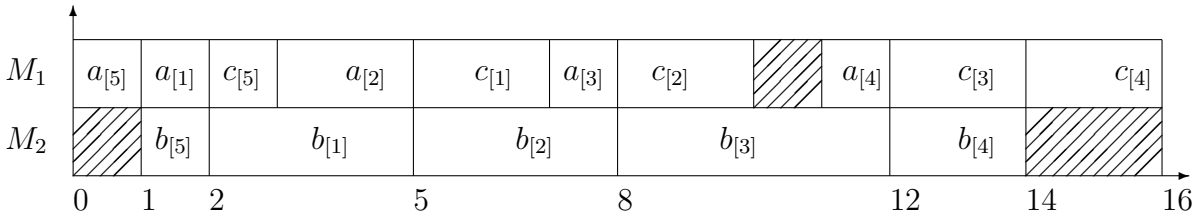


FIGURE 6.2 – Solution réalisable du problème P_3.

Lemme 6.1. *Le problème $P_3 : F2|ChR, b_j = l_j|C_{max}$ est un problème de permutation.*

Preuve. Comme les temps de traitement sont supposés différents de zéro, et les opérations d'une tâche sont ordonnancées sur les machines sans attente. Alors, c'est facile de voir qu'une permutation des tâches est un ensemble dominant. ■

La condition d'entrelacement de deux tâches T_i, T_j dans le problème P_3 et la durée de traitement du batch contenant ces deux tâches entrelacées sont données dans la définition suivante.

Définition 6.1. *La tâche T_i est entrelacée avec la tâche T_j dans cet ordre ssi $a_j \leq b_i$ et $c_i \leq b_j$ et la durée de traitement du batch formé est égale à : $a_i + b_i + b_j + c_j$.*

Ci-dessous on définit une nouvelle notion, c'est la notion d'une chaîne de tâches.

Définition 6.2. *On dit que l'ensemble des tâches $\{T_1, T_2, \dots T_k\}$ forme une chaîne si et seulement si ses tâches sont ordonnancées comme dans la figure 6.3.*
Dans ce cas on a : $a_2 \leq b_1$ et $c_{k-1} \leq b_k$ et $c_{i-2} + a_i \leq b_{i-1}, i = 2, \dots k$. et la durée de traitement de cette chaîne est égale à : $a_1 + c_k + \sum_{i=1}^{k} b_i$.

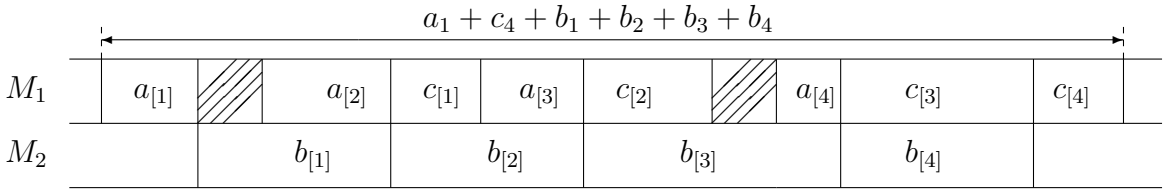


FIGURE 6.3 – Une chaîne de longueur 4 contenant les tâches $\{T_1, T_2, T_3, T_4\}$

Définition 6.3. *La longueur d'une chaîne est égale au nombre de tâches contenant cette chaîne.*

Remarque 6.1. *Un batch contenant deux tâches entrelacées est une chaîne de longueur 2.*

Soit l le nombre de tâches T_i qui vérifient la condition suivante : $a_i > \max_{j=1,n}\{b_j\}$, et q le nombre de tâches T_i qui vérifient la condition : $c_i > \max_{j=1,n}\{b_j\}$

Lemme 6.2. *La solution optimale du problème P_3 est formée d'une suite de chaînes ordonnancées l'une à coté de l'autre sans temps mort, et le nombre de chaînes dans cette solution est supérieur ou égal au maximum entre l et q ($\max\{l, q\}$).*

Preuve. Il suffit de voir qu'à l'intérieur d'une chaîne (c-à-d. entre les positions 2 et k de la chaîne) on ne peut pas avoir une tâche T_i qui a une durée de traitement sur la première machine (a_i ou c_i) supérieure à $\max_{j=1,n}\{b_j\}$ à cause de la condition suivante $c_{i-2} + a_i \leq b_{i-1}, i = 2, \dots k$.
Donc on ne peut pas avoir deux tâches T_x, T_y pour les quelles $a_x > \max_{j=1,n}\{b_j\}$ et $a_y > \max_{j=1,n}\{b_j\}$ dans la même chaîne. De même, on ne peut pas avoir aussi deux tâches T_z, T_p pour les quelles $c_z > \max_{j=1,n}\{b_j\}$ et $c_p > \max_{j=1,n}\{b_j\}$ dans la même chaîne.
Donc, si on suppose l le nombre de tâches qui vérifie la condition
$a_i > \max_{j=1,n}\{b_j\}$, et q le nombre de tâches qui vérifie la condition $c_i > \max_{j=1,n}\{b_j\}$.
Alors, le nombre de chaînes dans une solution optimale est supérieur ou égal au maximum entre l et q ($\max\{l, q\}$). Et comme on cherche à minimiser le makespan alors les chaînes formées doivent être ordonnancées sans temps mort. ■

Dans ce qui suit une analyse de complexité du problème P_3 est présentée.

6.2 Problème NP-difficile

Théorème 6.1. *[5] Le problème $F2/ChR, l_j = b_j/C_{max}$ est NP-difficile au sens fort.*

Preuve. Nous montrons que le problème P_3 est NP-difficile au sens fort par la réduction du problème NMTS (Numerical Matching with Target Sums) qui est NP-difficile au sens fort (Garey et Johnson [65]) au problème P_3. Le problème NMTS est défini comme suit : étant donnés deux ensembles d'entiers X et Y, avec $X = \{x_1, \ldots, x_n\}$ et $Y = \{y_1, \ldots, y_n\}$, et un ensemble $Z = \{z_1, \ldots, z_n\}$ d'entiers positifs. Est-il possible de partitionner l'ensemble $X \cup Y$ en n sous-ensembles disjoints $N_1, \ldots, N_n$, où chaque $N_j \quad (j = 1, \ldots, n)$ contient exactement un élément de X et un élément de Y, tel que $x_{Nj} + y_{Nj} = z_j$? Soit une instance arbitraire de $NMTS$, nous construisons une instance du problème d'ordonnancement avec $3n$ tâches comme suit :
L'ensemble des tâches T est défini comme suit.
Soit L et S deux entiers positifs tel que $L > \max_{i=1,n}\{z_i\}$ et $S = \sum_{i=1}^{n} z_j$.

- Les tâches de type X : $T_X = \{Tx_1, Tx_2, \ldots, Tx_n\}$, sont en correspondance avec les éléments de l'ensemble X. Leurs temps de traitement sur les deux machines sont notés ax_j, bx_j et cx_j.
- Les tâches de type Y : $T_Y = \{Ty_1, Ty_2, \ldots, Ty_n\}$, sont en correspondance avec les éléments de l'ensemble Y. Leurs temps de traitement sur les deux machines sont notés ay_j, by_j and cy_j.
- Les tâches de type Z : $T_Z = \{Tz_1, Tz_2, \ldots, Tz_n\}$, sont en correspondance avec les éléments de l'ensemble Z. Leurs temps de traitement sur les deux machines sont notés az_i, bz_i and cz_i.

Les temps de traitement des tâches sont donnés dans la table 6.3
Existe-t-il un ordonnancement des $3n$ tâches sur les deux machines M_1, M_2 telle que la durée totale de l'ordonnancement $C_{max} \leq 2(2L+1)n + S$?

TABLE 6.3 – Durée de traitement des tâches

tâches	a_i	b_i, l_i	c_i
Tx_i	$L+1$	L	x_i
Ty_i	y_i	L	$L+1$
Tz_i	L	z_i	L

Notre problème appartient à la classe NP car on peut vérifier en temps polynomial si une permutation des tâches vérifie toutes les conditions.
Nous prouvons que le problème d'ordonnancement a une solution si et seulement si le problème de NMTS a une solution.

Si NMTS a une solution, alors il existe une partition de $X \cup Y$ en n sous ensembles disjoints de cardinalité 2 (x_i, y_j) tels que $x_i + y_j = z_k$ comme indiqué dans la définition du NMTS. A cette solution, on construit une instance à notre problème d'ordonnancement avec n batchs où chaque batch contient exactement trois tâches Tx_i, Ty_j, Tz_k ordonnancées dans cet ordre. Notons qu'un batch a un temps de traitement égal à $4L + z_k + 2$ (voir Figure 6.4). Donc, le temps de fin de traitement de l'ensemble des tâches est : $4Ln + S + 2n$.

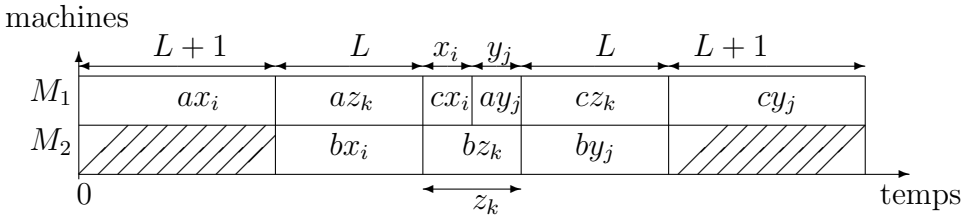


FIGURE 6.4 – Representation d'une solution avec NMTS.

Inversement, supposons qu'il existe un ordonnancement réalisable σ pour le problème P_3 tel que $C_{max} \leq 4Ln + 2n + S$. cet ordonnancement ne contient pas un temps mort sur la première machine.

Notons que l'entrelacement des tâches : $(Tx_i, Tx_{i'})$, $(Ty_j, Ty_{j'})$, (Tz_k, Tx_i) et (Ty_j, Tz_k) dans cet ordre n'est pas possible à cause de la durée de traitement $L + 1$. Aussi, l'entrelacement des tâches : (Ty_j, Tx_i) et (Tx_i, Ty_j) dans cet ordre induit un temps mort dans les batchs. donc, le seul entrelacement possible est :(Tx_i, Tz_k) et (Tz_k, Ty_j) dans cet ordre qui donne le batch (Tx_i, Tz_k, Ty_j). Comme on a $\sum_{i=1}^{n}\{x_i + y_i\} = \sum_{i=1}^{n} z_i = S$, et la valeur du makespan est égale à $4Ln + 2n + S$. d'où, pour tous les batchs formés on a $cx_i + ay_j = bz_k$. Donc on obtient une solution pour le problème NMTS.

■

6.3 Sous-problèmes faciles

Dans cette section, nous proposons des sous problèmes résolubles en des temps polynomiaux du problème P_3.

6.3.1 Problème $P_{31} : F2|ChR, b_j = l_j = L, a_i + c_j > L|C_{max}$

Théorème 6.2. *[11] Le problème P_{31} se réduit au problème de couplage de poids maximum.*

Preuve. Si pour toutes les tâches on a $b_j = l_j = L$ et $a_i + c_j > L$, alors la solution du problème P_{31} est formée d'une suite de batchs de cardinalité 2 ou 1, voir Figure 6.5. Comme on cherche à minimiser le makespan, donc les batchs formés sont ordonnancés sans temps mort et la valeur du makespan est égale à la somme des durées de traitement des batchs formant la solution.

Ceci nous permet de modéliser le problème P_{31} par un programme linéaire. Avant de définir les variables de décision, On considère d'abord le graphe $G = (V, E)$, où l'ensemble des sommets $V = \{T_1, T_2, \ldots, T_n\}$ correspond à l'ensemble des tâches et l'ensemble des arêtes $E = \{e_1, e_2, \ldots, e_q\}$ est défini par :
$(T_l, T_{l'}) \in E \Leftrightarrow (a_{l'} \leq L$ et $c_l \leq L)$ ou $(a_l \leq L$ et $c_{l'} \leq L)$.

Pour chaque arête e_k avec les extrémités T_{s_k} et T_{t_k}, on définit son poids ρ_k comme suit. Les extrémités de e_k peuvent étre entrelacée dans cet ordre T_{s_k}, T_{t_k} si $(a_{t_k} \leq L$ et $c_{s_k} \leq L)$, dans ce cas la durée de traitement du batch formé est égale à $d_{s_k t_k} = a_{s_k} + c_{t_k} + 2 * L$, voir Figure 6.5 (a). Inversement, ses extrémités peuvent être entrelacées dans cet ordre T_{t_k}, T_{s_k} si $(a_{s_k} \leq L$ et $c_{t_k} \leq L)$, dans ce cas la durée de traitement du batch formé est égale à $d_{t_k s_k} = a_{t_k} + c_{s_k} + 2 * L$. D'où, la durée de traitement minimale du batch contenant les deux tâches T_i, T_j est notée $\rho_k = min\{d_{s_k t_k}, d_{t_k s_k}\}$. Pour chaque tâche T_l, on associe un poids. $\delta_l = a_l + c_l + L$ voir Figure 6.5 (b).

Soit λ la matrice d'incidence sommet-arête du graphe $G = (V; E)$ où

$$\lambda_{lk} = \begin{cases} 1, & \text{si l'arête } e_k \text{ est incidente au sommet l;} \\ 0, & \text{sinon.} \end{cases}$$

Pour $k = 1, \ldots, n$, $l = 1, \ldots, q$ (q le nombre d'arête). On définit q variables de décision $x_1, x_2, \ldots, x_q$ telles que

$$x_k = \begin{cases} 1, & \text{si les deux tâches reliées par l'arête } e_k \text{ sont entrelacées entre elles dans} \\ & \text{le même batch ;} \\ 0, & \text{sinon.} \end{cases}$$

Le programme linéaire est donné par :

$$min\ C_{max} = (\sum_{k=1}^{q} \rho_k x_k) + \sum_{l=1}^{n} (1 - \sum_{k=1}^{q} \lambda_{lk} x_k) \delta_l.$$

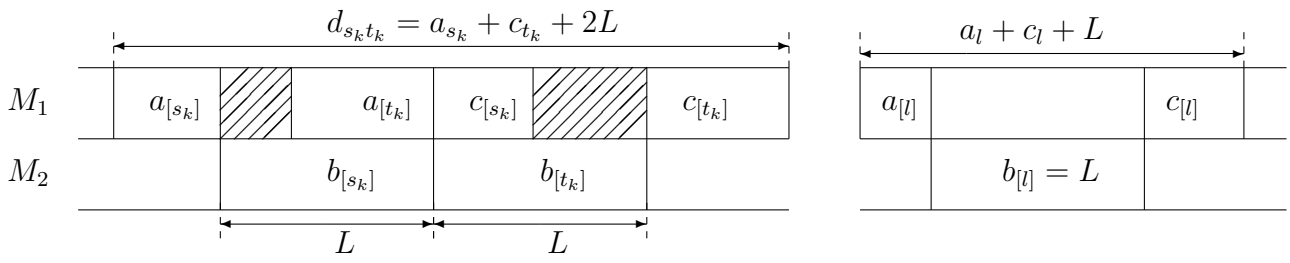


FIGURE 6.5 – (a) Batch contenant deux tâches entrelacées T_{s_k}, T_{t_k} dans cet ordre ; (b) Batch contenant une seule tâche

$$\text{s.c.} \begin{cases} \sum_{k=1}^{q} \lambda_{lk} x_k \leq 1 \text{ for } l = \overline{1,n} \ldots\ldots (1) \\ x_k \in \{0,1\} \text{ for } k = \overline{1,q} \end{cases}$$

La quantité $\sum_{k=1}^{q} \rho_k x_k$ représente la somme des durées de traitement des batchs contenant deux tâches entrelacées, tandis que la somme $\sum_{l=1}^{n}(1 - \sum_{k=1}^{q} \lambda_{lk} x_k)\delta_l$ est la somme des durées de traitement des batchs contenant une seule tâche.
Les contraintes (1) indiquent que chaque tâche doit appartenir à au plus, un batch.
La fonction objective C_{max} peut étre calculée comme suit :

$$C_{max} = \sum_{k=1}^{q} \rho_k x_k + \sum_{l=1}^{n}(1 - \sum_{k=1}^{q} \lambda_{lk} x_k)\delta_l = (\sum_{l=1}^{n} \delta_l) - (\sum_{l=1}^{n}\sum_{k=1}^{q} \lambda_{lk} x_k \delta_l - \sum_{k=1}^{q} \rho_k x_k)$$

$$= \sum_{l=1}^{n} \delta_l - (\sum_{k=1}^{q}\sum_{l=1}^{n} \lambda_{lk}\delta_l x_k - \sum_{k=1}^{q} \rho_k x_k) \text{ et donc}$$

$C_{max} = \sum_{l=1}^{n} \delta_l - \sum_{k=1}^{q}(\sum_{l=1}^{n} \lambda_{lk}\delta_l - \rho_k)x_k$. A partir de (1), On a
$\sum_{l=1}^{n} \lambda_{lk}\delta_l - \rho_k = \delta_{s_k} + \delta_{t_k} - \rho_k = \delta_{s_k} + \delta_{t_k} - \min\{d_{s_k t_k}, d_{t_k s_k}\}$

$$= a_{s_k} + c_{s_k} + L + a_{t_k} + c_{t_k} + L - \min\{a_{s_k} + c_{t_k} + 2L, a_{t_k} + c_{s_k} + 2L\}$$

$$= a_{s_k} + c_{s_k} + a_{t_k} + c_{t_k} - \min\{a_{s_k} + c_{t_k}, a_{t_k} + c_{s_k}\} = \max\{a_{s_k} + c_{t_k}, a_{t_k} + c_{s_k}\}$$

Donc :

$$C_{max} = \sum_{l=1}^{n} \delta_l - \sum_{k=1}^{q} \max\{c_{s_k} + a_{t_k}, a_{s_k} + c_{t_k}\}x_k.$$

$$C_{max} = \sum_{l=1}^{n}(a_l + c_l + L) - \sum_{k=1}^{q} \max\{c_{s_k} + a_{t_k}, a_{s_k} + c_{t_k}\}x_k.$$

$\sum_{l=1}^{n}(a_l + c_l + L)$ est une constante supérieure à $\sum_{k=1}^{q} \max\{c_{s_k} + a_{t_k}, a_{s_k} + c_{t_k}\}x_k$.
Donc, minimiser C_{max} est équivalent à maximiser $\sum_{k=1}^{q} \max\{c_{s_k} + a_{t_k}, a_{s_k} + c_{t_k}\}x_k$. Ainsi, le problème se réduit au problème du couplage de poids maximum. ■

Algorithme $\mathcal{A}3$ pour le problème P_{31}

1. A partir du graphe $G = (V, E)$ construire un nouveau graphe pondéré $H = (V, E)$ où chaque arête de E possède un poids égal à : $\max\{c_{s_k} + a_{t_k}, a_{s_k} + c_{t_k}\}$ si l'arête est incidente aux sommets s_k et t_k.
2. Trouver un couplage de poids maximum M du graphe H.
3. Pour chaque paire de M, entrelacer les deux tâches correspondantes dans le même batch.
4. Mettre chaque tâche restante dans un seul batch.
5. Ordonnancer les batchs formés dans un ordre quelconque sans temps mort.

Corollaire 6.1. *L'algorithme $\mathcal{A}3$ résout le problème $F2|ChR, b_j = l_j = L, a_i + c_j > L|C_{max}$ en $O(n^{2.5})$.*

Preuve. L'optimalité est garantie par le théorème 6.1. La complexité de l'algorithme A_3 est donnée par la deuxième étape 2, qui fournit un couplage de poids maximum en $O(n^{2.5})$, [22]. Donc la complexité de l'algorithme A_3 est $O(n^{2.5})$. ■

Exemple 6.1. *Nous disposons de 5 tâches $T_1, ..., T_5$ Les durées de traitement de ces tâches sur les deux machines sont données dans la table 6.4 suivante avec $b_j = l_j = L = 8$.*

TABLE 6.4 – Durée de traitement des 5 tâches

T_j	T_1	T_2	T_3	T_4	T_5
a_j	4	5	6	9	9
c_j	7	5	5	6	10

1- On a : , (T_1, T_4), (T_1, T_5), (T_2, T_4), (T_2, T_5), (T_3, T_4), (T_3, T_5), (T_4, T_5), (T_5, T_1), (T_5, T_2), (T_5, T_3), (T_5, T_4), ne peuvent pas être entrelacées dans un même batch car la condition d'entrelacement n'est pas vérifiée. Donc le graphe $G = (V, E)$ contient 5 sommets (nombre de tâches) et 6 arêtes :$\{(T_1, T_2), (T_1, T_3), (T_2, T_3), (T_4, T_1), (T_4, T_2), (T_4, T_3)\}$

2- Construction du graphe H et évaluation des arêtes (voir Figure 6.6) :

$d(T_1, T_2) = \{7 + 5\} = 12$. $d(T_1, T_3) = \{7 + 6\} = 13$.

$d(T_2, T_1) = \{5 + 4\} = 9$. $d(T_2, T_3) = \{5 + 6\} = 11$.

$d(T_3, T_1) = \{5 + 4\} = 9$. $d(T_3, T_2) = \{5 + 5\} = 10$.

$d(T_4, T_1) = \{6 + 4\} = 10$. $d(T_4, T_2) = \{6 + 5\} = 11$. $d(T_4, T_3) = \{6 + 6\} = 12$.

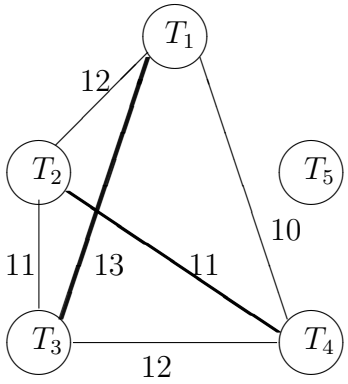


FIGURE 6.6 – Graphe valué H

3- Le couplage de poids maximum est : $M = \{(T_1, T_3), (T_4, T_2)\}$.

4- La solution optimale est :$(T_1, T_3), (T_4, T_2), T_5$. *La durée de traitement du batch* (T_1, T_3) *est égale à* $a_1 + c_3 + 2L = 25$, *La durée de traitement du batch* (T_4, T_2) *est égale à* $a_4 + c_2 + 2L = 30$ *et la durée de traitement du batch contenant la tâche* (T_5) *est égale à* $a_5 + c_5 + L = 27$. *Donc, la durée de fin de traitement de l'ensemble des 5 tâches est égale à* $C_{max} = 25 + 30 + 27 = 82$.

6.3.2 Problème $P_{32} : F2|ChR, b_j = l_j = L, a_i + c_j \leq L|C_{max}$

Théorème 6.3. *[10] Le problème* P_{32} *est résoluble en temps polynomial.*

L'algorithme suivant résout le problème P_{32} en $O(nlogn)$.

Algorithme $\mathcal{A}4$ pour le problème P_{32}

Construire la solution comme suit :
1. Trouver deux tâches k et l telles que : $a_k + c_l = \min_{i \neq j}\{a_i + c_j\}$.
2. Ranger les tâches selon l'ordre suivant : $T_k, \ldots, T_i, T_{i+1}, \ldots, T_l$ où la tâche T_k est ordonnancée à la première position et la tâche T_l à la dernière position.
3. Ordonnancer les deux tâches T_i, T_{i+1} $(i = k, \ldots, l-1)$ comme dans la Figure 6.7.

Preuve. On a $b_j = L$, $\forall j$, donc, la cardinalité d'un batch formé est inférieure ou égale à deux, et les batchs formés dans ce cas peuvent être entrelacées (chaînés entre eux),

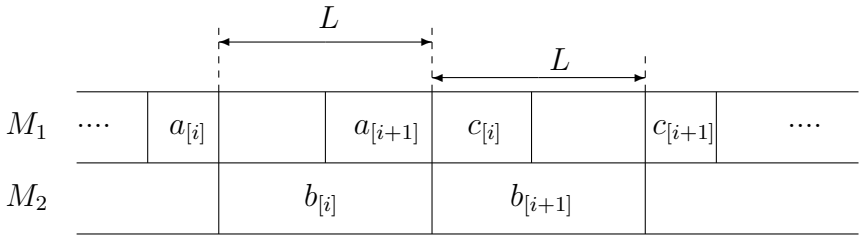


FIGURE 6.7 – Entrelacement de deux tâches.

et comme nous avons aussi la condition que la somme des temps de traitement sur la première machine est toujours inférieure à L, alors tous les batchs peuvent être entrelacés entre eux. Donc, pour minimiser la date de fin de traitement de l'ensemble des tâches, il suffit de mettre les deux tâches T_i, T_j qui donnent la plus petite somme $\min_{i \neq j}\{a_i + c_j\}$ au début et à la fin de l'ordonnancement respectivement, et mettre les autres tâches au milieu. La valeur du makespan est égale à $C_{max} = \min_{i \neq j}\{a_i + c_j\} + nL$. ■

Exemple 6.2. *Nous disposons de* 5 *tâches* $T_1, ..., T_5$. *Les durées de traitement de ces tâches sur les deux machines sont données dans le tableau suivant et le time lag* $b_j = l_j = L = 8$

TABLE 6.5 – Durée de traitement des 5 tâches

T_j	T_1	T_2	T_3	T_4	T_5
a_j	1	4	3	2	4
c_j	2	4	1	2	3

On a toutes les tâches peuvent être entrelacées entre elles, car $a_i + c_j \leq L$. *On choisit donc deux tâches* T_l, T_k *telles que* $a_l + c_k = \min_{i \neq j}\{a_i + c_j\}$. *Dans notre cas* $l = 1$ *et* $k = 3$. *L'ordonnancement optimal est donné par la séquence suivante* T_1, T_2, T_4, T_5, T_3 *avec* $C_{max} = a_1 + c_3 + 5L = 42$ *(voir Figure 6.8).*

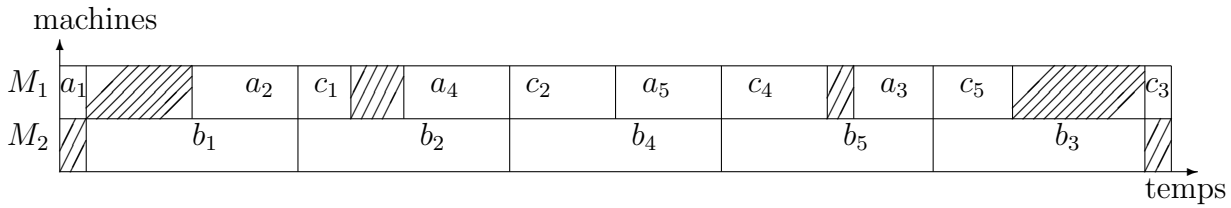


FIGURE 6.8 – Solution de l'exemple 6.2

6.3.3 Problème $P_{33} : F2|ChR, a_j = b_j = l_j = L|C_{max}$

On termine ce chapitre par le dernier sous problème résoluble en un temps polynomial à savoir le problème $P_{33} : F2|ChR, a_j = b_j = l_j = L|C_{max}$. Ce dernier peut être résolu en utilisant l'algorithme A_5 proposé dans la section 4.2 car la condition $a_i + c_j \geq L$ est vérifiée pour ce problème mais on préfère donner un nouvel algorithme polynomial plus rapide (L'algorithme A_5) qui est de complexité $O(nlogn)$ au lieu de l'algorithme A_4 qui est de complexité $O(n^{2.5})$.

Théorème 6.4. *Le problème P_{33} est résoluble en temps polynomial.*

Algorithme $\mathcal{A}5$ pour le problème P_{33}

Construire la solution comme suit :
1. Ordonnancer les tâches selon l'ordre croisant des c_j. Soit $T_1, \ldots, T_{j-1}, T_j, \ldots, T_n$ cet ordre.
2. S'il existe une tâche T_j à partir de laquelle $c_j > L$, alors partitionner l'ensemble des tâches en deux sous ensembles $T = E_1 \cup E_2$ tel que $E_1 = \{T_1, T_2, \ldots T_{j-1}\}$ et $E_2 = \{T_j, T_{j+1}, \ldots T_n\}$.
3. Entrelacer chaque tâche de l'ensemble E_2 avec une tâche de l'ensemble E_1 qui a le plus grand c_j afin de former des batchs de cardinalité 2.
4. Si $|E_1| = \emptyset$ alors Entrelacer les tâches de l'ensemble E_2 entre elles, deux à deux, afin de former des batchs de cardinalité 2 (on entrelace la dernière tâche de l'ensemble E_2 avec la première tâche de l'ensemble E_2) jusqu'à ce que $|E_2| = \emptyset$.
5. Si $|E_2| = \emptyset$ alors ordonnancer les tâches de l'ensemble E_1 dans des batchs de cardinalité 1 jusqu'à ce que $|E_1 = \emptyset|$.
6. Ordonnancer les batchs formés selon un ordre quelconque sans temps mort.

Preuve. Comme on a $a_j = b_j = L$ pour toutes les tâches alors la cardinalité maximale d'un batch est égale à deux. Un batch de cardinalité deux de la forme (T_l, T_k) a une durée de traitement égale à $3L + c_k$. Tandis qu'un batch contenant une seule tâche T_k a une durée de traitement égale à $2L + c_k$. Comme $3L$ et $2L$ sont des constantes alors minimiser C_{max} revient à minimiser la somme des c_k des batchs formés. L'étape 1 de l'algorithme $\mathcal{A}5$ ordonnance les tâches selon l'ordre croissant des c_j et la manière dans laquelle les batchs sont formés garantie l'optimalité. Car la somme des c_k des batchs formés est la plus petite somme possible. ■

Exemple 6.3. *Nous disposons de 7 tâches $T_1, ..., T_7$. Les durées de traitement de ces tâches sur les deux machines sont données par la table 6.6 avec $a_j = b_j = l_j = L = 8$*

TABLE 6.6 – Durée de traitement des tâches

T_j	T_1	T_2	T_3	T_4	T_5	T_6	T_7
c_j	1	2	5	2	6	10	9

La première étape de l'algorithme $\mathcal{A}5$ consiste à ranger les tâches selon l'ordre croissant des c_j : $T_1, T_2, T_4, T_3, T_5, T_7, T_6$ ensuite à partitionner l'ensemble des tâches $T = E_1 \cup E_2$ tels que $E_1 = \{T_1, T_2, T_4, T_3, T_5\}$ et $E_2 = \{T_7, T_6\}$. La dernière étape est de former des batchs et à les ordonnancer sans temps mort. L'ordonnancement optimal est donc donné par la séquence $(T_5, T_7), (T_3, T_6), (T_4, T_2), T_1$ avec $C_{max} = 3L + c_7 + 3L + c_6 + 3L + c_2 + +2L + c_1 = 110$.

Notons que le problème $P_{34} : F2|ChR, b_j = c_j = l_j = L|C_{max}$ peut être résolu avec le même algorithme $A5$ car c'est le cas symétrique, il suffit de remplacer dans cet algorithme le terme c_j par a_j.

Chapitre 7

Résolution du problème $P_3 : F2|ChR, b_j = l_j|C_{max}$

Dans ce chapitre, nous proposons une heuristique basée sur la recherche aléatoire de type Multi-start descente, et une métaheuristique pour résoudre le problème $P_3 : F2|ChR, b_j = l_j|C_{max}$.

7.1 Bornes inférieures

Afin de tester la performance des heuristiques proposées, nous proposons des bornes inférieures relatives au problème $P_3 : F2|ChR, b_j = l_j|C_{max}$.
Le calcul des bornes inférieures se fait en deux étapes. La première étape consiste à partitionner l'ensemble des tâches T en plusieurs sous ensembles et la deuxième étape consiste à ranger les sous ensembles obtenus dans un ordre bien précis.
1ère étape :
On commence d'abord par partitionner l'ensemble des tâches T en deux sous ensembles E_1 et E_2 tels que :
$E_1 = \{T_j \in T / a_j, c_j > \max\limits_{i=1,n}\{b_i\}\}$, l'ensemble E_2 est le complémentaire de l'ensemble E_1 donc $E_2 = T \setminus E_1$.
Ensuite, on partitionne l'ensemble E_2 en quatre sous ensembles $E_{21}, E_{22}, E_{23}, E_{24}$ tel que :
$E_{21} = \{T_j \in T / a_j > \max\limits_{i=1,n}\{b_i\}, c_j \leq \max\limits_{i=1,n}\{b_i\}\}$, l'ensemble E_{23} est le complémentaire de l'ensemble E_{21} donc $E_{23} = E_2 \setminus E_{21}$.
Et enfin les ensembles E_{22}, E_{24} sont définis comme suit :
$E_{22} = \{T_j \in T / c_j > \max\limits_{i=1,n}\{b_i\}, a_j \leq \max\limits_{i=1,n}\{b_i\}\}$, l'ensemble E_{24} est le complémentaire de

l'ensemble E_{22} donc $E_{24} = E_2 \setminus E_{22}$.

2ème étape :

On range l'ensemble E_{23} selon l'ordre croissant des c_j soit (π_1) cet ordre, et π_2 l'ordre obtenu en rangeant l'ensemble E_{24} selon l'ordre croissant des a_j.

On note par :

-$c_{\pi_1 j}$: La valeur du temps de traitement de l'opération c de la tâche en position j dans la permutation π_1.

-$a_{\pi_2 j}$: La valeur du temps de traitement de l'opération a de la tâche en position j dans la permutation π_2.

Proposition 7.1. $LB_1 = \sum_{j=1}^{n}(a_j + c_j)$,

$LB_2 = \min_{1\leq j\leq n}\{a_j\} + \sum_{j=1}^{n} b_j + \min_{1\leq j\leq n}\{c_j\}$,

$LB_3 = \sum_{j=1}^{n} b_j + \sum_{j\in E_1}(a_j + c_j) + \sum_{j\in E_{21}} a_j + \sum_{j=1}^{|E_{21}|} c_{\pi_1 j}$,

$LB_4 = \sum_{j=1}^{n} b_j + \sum_{j\in E_1}(a_j + c_j) + \sum_{j\in E_{22}} c_j + \sum_{j=1}^{|E_{22}|} a_{\pi_2 j}$ *et*

$LB_5 = \sum_{j=1}^{n} b_j + \sum_{j\in E_1}(a_j + c_j) + \sum_{j\in E_{21}} a_j + \sum_{j\in E_{22}} c_j$.

sont des bornes inférieures.

$LB = max\{LB_1, LB_2, LB_3, LB_4, LB_5\}$ *est aussi une borne inférieure.*

Notons que ces bornes inférieures sont utilisées afin d'évaluer la performance des heuristiques proposées.

Preuve. Les bornes inférieures LB_1 et LB_2 du problème P_2 sont aussi des bornes inférieures pour le problème P_3.

Les bornes LB_3, LB_4, LB_5 sont une conséquence directe du lemme 6.1 du chapitre précédent. Puisque le nombre de chaînes de tâches est supérieur ou égal au maximum entre l et q.

Donc, il existe forcément des chaînes qui commencent par une tâche qui a un temps de traitement sur la première a_j machine supérieur à $max\{b_i\}$ et des chaînes qui se terminent par une tâche qui a un temps de traitement sur la première c_j machine supérieur à $max\{b_i\}$. Donc, on doit comptabiliser ces temps dans le calcul des bornes inférieures. ■

7.2 Approche heuristique

Pour résoudre le problème $P_3 : F2|ChR, b_j = l_j = L|C_{max}$ nous proposons d'abord deux méthodes approchées. Leur principe est d'appliquer une recherche aléatoire dans le voisinage des solutions réalisables ; une fois le critère d'arrêt rencontré, on retourne la meilleure solution trouvée.

7.2.1 Heuristique simple descente

La première heuristique proposée est le squelette d'une méthode de descente simple. A partir d'une solution initiale x (le problème P_3 est dominé par la permutation, donc une solution à notre problème est une permutation de tâches de taille n), on construit une solution x' qui est voisine à x (x' est obtenue en permutant le contenu de deux positions de la solution initiale x. Notons que ces deux positions sont choisies d'une manière aléatoire). Si la solution x' obtenue est meilleure que x (c-à.d. $f(x') \leq f(x)$) alors on accepte cette solution comme nouvelle solution et on recommence le processus jusqu'à ce qu'il n'y ait plus aucune solution améliorante dans le voisinage de x.

Le critère d'arrêt pour notre heuristique simple descente est le nombre de voisins de la solution x générée. Pour notre algorithme ce nombre est fixé à 1000.

Heuristique simple descente

Début
1. Initialisation : trouver une solution initiale x, $i = 0$.
2. **Tant que** le critère d'arrêt est non rencontré ($i \leq 1000$) faire :
3. Trouver une solution x' dans le voisinage de x
   Si $f(x') < f(x)$ alors : $x = x'$, $i = i + 1$.
4. **Fin Tant que**

Fin.

Cette méthode se base sur une amélioration progressive de la solution et donc reste bloquée dans un minimum local dès qu'elle le rencontre. Il existe d'une manière évidente une absence de diversification. L'équilibre souhaité entre intensification et diversification n'existe donc plus. Un moyen très simple de diversifier la recherche peut consister à exécuter cet algorithme en prenant un autre point de départ. Comme l'exécution de cette méthode est souvent très rapide, on peut alors inclure cette répétition au sein d'une boucle générale. On obtient alors un algorithme de type multi-start descente.

7.2.2 Heuristique multi-start descente

Heuristique multi-start descente
Début
1. Initialisation : trouver une solution initiale x, $f(meilleur) = +\infty$.
2. **Tant que** le critère d'arrêt est non rencontré faire :
3. x' est le résultat de l'heuristique simple descente.
Si $f(x') < f(meilleur)$ alors : $meilleur = x'$.
4. Générer une autre solution (un autre point de départ) x.
5. **Fin Tant que**
Fin.

On a choisit d'arrêter notre heuristique quand le temps d'exécution dépasse 5 minutes.

7.3 Approche méta-heuristique

Cette section se concentre sur la présentation des approches métaheuristiques proposées pour l'amélioration des heuristiques. Les métaheuristiques sont des heuristiques stochastiques itératives qui progressent vers un optimum local en se comportant comme des algorithmes de recherche, ces méthodes sont généralement hybridées avec une recherche locale. Nous allons adopter l'algorithme génétique.

7.3.1 Algorithmes génétique

Les algorithmes génétiques font partie de la famille des algorithmes évolutionnaires. Ils s'inspirent de l'évolution biologique des espèces et basées sur une imitation des phénomènes d'adaptation des êtres vivants. Avec ce type de méthodes, il ne s'agit pas de trouver une solution analytique exacte mais de trouver une bonne solution satisfaisante dans un temps de calcul raisonnable. La première description du processus des algorithmes génétiques a été donnée par Holland en 1975, puis Goldberg [31] les a utilisées pour résoudre des problèmes concrets d'optimisation.

Le but de ces algorithmes génétiques est d'optimiser une fonction prédéfinie, appelée fonction objectif, ou fitness ; ils travaillent sur un ensemble de solutions candidates, appelé *population* d'individus ou chromosomes. Ces derniers sont constitués d'un ensemble

d'éléments, appelés *gènes*, qui peuvent prendre plusieurs valeurs, appelées *allèles*. Un chromosome est une représentation ou un codage d'une solution du problème donné. Une première population est choisie soit aléatoirement, soit par des heuristiques ou par des méthodes spécifiques au problème, soit encore par mélange de solutions aléatoires et heuristiques. Cette population doit être suffisamment diversifiée pour que l'algorithme ne reste pas bloqué dans un optimum local. C'est ce qui se produit lorsque trop d'individus sont semblables. Les algorithmes génétiques génèrent de nouveaux individus, de telle sorte qu'ils soient plus performants que leurs prédécesseurs. Le processus d'amélioration des individus s'effectue par utilisation d'opérateurs génétiques, qui sont la sélection, le croisement et la mutation.

Pour mettre en oeuvre un algorithme génétique, il est nécessaire de disposer :

– D'une représentation génétique du problème, c'est-à-dire un codage de solutions utilisé sous la forme de chromosomes.

– D'un mécanisme de génération de la population initiale.

– D'une fonction qui permet d'évaluer l'adaptation d'un chromosome à son environnement, ce qui offre la possibilité de comparer des individus.

– D'un mode de sélection des chromosomes à reproduire.

– d'opérateurs de croisement et de mutation permettant de diversifier la population au cours des générations.

Les étapes de l'algorithme génétique

L'algorithme génétique commence par une génération d'une population initiale de Psize individus, pour lesquels nous calculons leurs fitness et nous sélectionnons les individus par une méthode de sélection. Ces individus seront manipulés par un opérateur de croisement qui les choisit selon une probabilité Pcrois. Leurs résultats peuvent être mutés par un

opérateur de mutation avec une probabilité de mutation *Pmut*. Les phases de sélection et de recombinaison (croisement et mutation) permettent de générer une nouvelle population d'individus, qui ont de bonne chances d'être plus forts que ceux de la génération précédente. Les individus issus de la phase de recombinaison seront insérés par une méthode d'insertion dans la nouvelle population, dont nous évaluons la valeur de la fonction objectif de chacun de ses individus. De génération en génération, la force des individus de la population augmente et un test d'arrêt sera effectué pour décider quand arrêter l'algorithme. La Figure 7.1 présente un schéma de fonctionnement général de l'algorithme génétique.

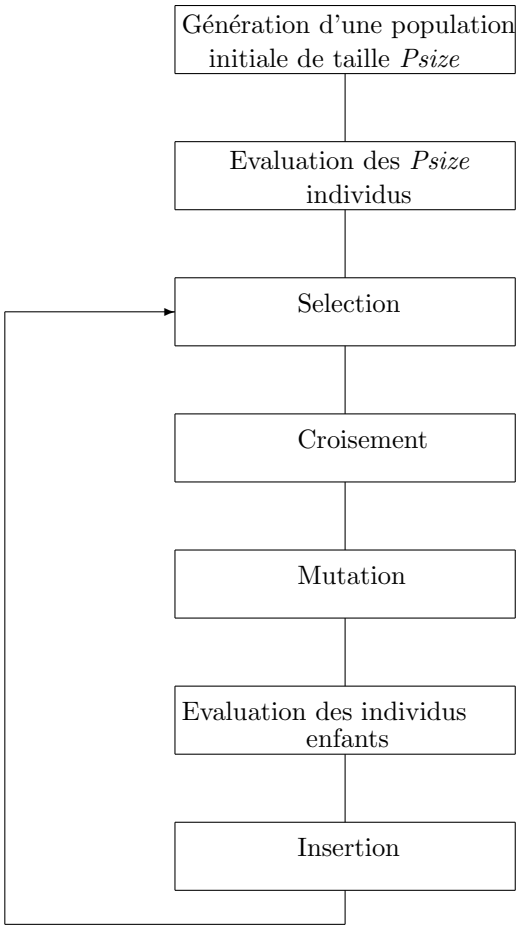


FIGURE 7.1 – Fonctionnement général de l'algorithme génétique.

Adaptation de l'algorithme génétique au problème $P3$

– Codage : Le premier pas dans l'implantation des algorithmes génétiques est de créer une population initiale d'individus. Le codage que nous avons retenu est de longueur égale au nombre total de tâches à réaliser. L'individu est alors représenté par une séquence de

tâches. La Figure 7.2 présente le codage d'une solution quelconque. La tâche numéro 5 sera lancée en production la première, suivie de la tâche 7... etc. et enfin la tache numéro 6, placée dans le dernier chromosome du code, sera lancée la dernière.

5	7	4	2	1	8	3	6

FIGURE 7.2 – Codage d'une solution.

- Population initiale : la population initiale que nous avons considérée est composée de séquences de tâches générées aléatoirement. Le problème principal dans cette étape est le choix de la taille de la population. Si la taille de la population est trop grande, le temps de calcul augmente et demande un espace mémoire important. Par contre, pour une population de taille très petite, la solution obtenue n'est pas satisfaisante. Il faut donc trouver le bon compromis. Pour cela, nous avons testé le paramètre *Tint* qui représente la taille de notre population avec les valeurs $\{10, 20, 50, 100\}$ et d'après les résultats obtenus, la bonne valeur est $Tint = 50$.

- La procédure de sélection : la sélection permet d'identifier les individus susceptibles d'être croisés dans une population. La procédure de sélection que nous avons utilisée est la sélection par la roulette (Roulette Wheel Selection, RWS en Anglais). La méthode RWS exploite la métaphore d'une roulette de casino, qui comporterait autant de cases que d'individus dans la population et où la taille de ces cases serait proportionnelle à la performance de chaque individu. Le jeu étant lancé, l'individu sélectionné est désigné par l'arrêt de la bille sur sa case. Donc, si on note f_i la performance de l'individu i (la fonction objectif de l'individu i) et p_i la probabilité de le sélectionner. Alors, $p_i = \frac{\sum f_i - f_i}{\sum f_i}$.

– Le croisement : le croisement permet d'enrichir la population en manipulant les composantes des chromosomes. Un croisement est envisagé avec deux parents et génère un ou deux enfants. Les individus survivants à la phase de sélection vont subir le croisement avec une probabilité *Pcrois*, dans notre algorithme génétique, nous utilisons le croisement en 2-points. Ce croisement s'effectuera de la manière suivante :

– Choisir deux individus de la population actuelle comme parents pour générer deux enfants.

– Générer aléatoirement deux positions $k_1, k_2 \in [1, \ldots, n]$ avec $k_1 < k_2$.

– Générer aléatoirement une probabilité P.

– Si $P \leq Pcrois$: on fait une opération de croisement. Ce croisement conserve dans l'enfant i la zone interne du parent j (zone comprise entre k_1 et k_2) et pour l'enfant j la zone interne du parent i. Ensuite, compléter les cases vides restantes de l'enfant i par les éléments du parent i et les cases vides restantes de l'enfant j par les éléments du parent j qui ne provoquent pas de doublon. voir Figure 7.3

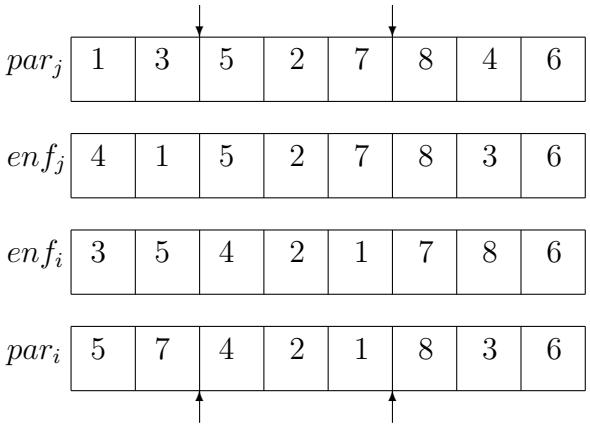


FIGURE 7.3 – Opération de croissement avec $k_1 = 2$ et $k_2 = 5$

– La mutation : Classiquement, l'opérateur de mutation modifie aléatoirement un individu pour en former un autre qui le remplacera. La mutation nous garantit que l'algorithme génétique sera susceptible d'atteindre la plupart des points du domaine réalisable. L'opérateur de mutation utilisé dans notre cas est l'opérateur d'échange réciproque qui permet de sélectionner au hasard deux gènes et de les échanger.
Si la probabilité P, générée aléatoirement, appartient à l'intervalle $]Pcroi, Pmut]$ nous appliquons l'opérateur de mutation sur les deux enfants i et j pour avoir deux enfants mutés $imut$ et $jmut$, Sinon si la probabilité générée est strictement supérieure à $Pmut$ ($P > Pmut$) : dans ce cas, on copie le parent i à l'enfant i et le parent j à l'enfant j.

– Fonction d'évaluation : pour chaque chromosome, qui est une permutation de n tâches, nous calculons la valeur du makespan correspondant.

– Insertion : les individus qui sont générés aléatoirement et les enfants mutés et croisés

sont triés selon leur fonction d'évaluation dans un ordre croissant. Seule la moitié supérieure de la population, correspondant aux meilleurs individus, est sélectionnée. Il est à noter que la taille de la population reste fixe, égale à *Tint*, de génération en génération.

– Critère d'arrêt : l'algorithme génétique s'arrête après un nombre fixé de générations noté *Itermax*.

Finalement, les étapes de notre algorithme sont données par l'algorithme suivant :

Algorithme génétique pour le problème P_3

1. Initialiser : $Tint$, $Itermax$, $Pmut$, $Pcroi$, δ.
2. Générer la population initiale de taille $Tint$.
3. Evaluer tous les chromosomes par la fonction d'évaluation.
4. **Répéter**
5. $i = 0$
6. **Tant que** $\delta \leq \frac{Tint}{2}$ **faire**
7. Sélectionner par la roulette deux parents de la population.
8. Générer une probabilité P
9. Si $P \leq Pcroi$ alors
10. Croiser les deux parents sélectionnés afin d'obtenir deux enfants.
11. Si $P \in]Pcroi, Pmut]$ alors
12. Muter les deux parents sélectionnés afin d'obtenir deux enfants mutés.
13. Si $P > Pmut$ alors
12. Copier les deux parents sélectionnés aux deux enfants.
13. $\delta = \delta + 1$.
14. **Fin Tant que**.
15. Ranger les parents et les enfants dans l'ordre croissant selon leur fonction d'évaluation.
16. Supprimer les $Tint$ chromosomes faibles et enregistrer les $Tint$ meilleures chromosomes selon leur fonction d'évaluation.
17. $i = i + 1$.
18. **Jusqu'à** $i = Itermax$.

7.4 Expérimentations numériques

Pour évaluer l'efficacité de l'heuristique multi-start descente et de l'algorithme génétique proposés, nous avons effectué des expérimentations numériques. Pour cela, nous avons considéré deux types d'instances distinctes.
Le premier type est caractérisé par des temps d'exécution sur la première machine générés aléatoirement comme suit : (i) $a_j \in [1, 30]$, $c_j \in [1, 40]$, (ii) $a_j \in [20, 60]$, $c_j \in [10, 50]$, et

des temps d'exécution sur la deuxième machine identiques et égaux au time lag $b_j = l_j = L = \{40, 60\}$.

Le second type est caractérisé par des temps d'exécution sur la première machine générés aléatoirement comme suit : (i) $a_j \in [1, 30]$, $c_j \in [1, 40]$, (ii) $a_j \in [20, 60]$, $c_j \in [10, 50]$, et des temps d'exécution sur la deuxième machine identiques et égaux au time lag $b_j = l_j \in [40, 70], b_j = l_j \in [70, 100]$.
Nous avons généré 100 instances aléatoires pour chaque problème de tailles $n = 20$, $n = 50$, $n = 100$, $n = 500$ et $n = 1000$. Afin d'évaluer la performance des heuristiques et de la métaheuristique proposée, nous avons effectué une série de tests sur des instances générées aléatoirement.

TABLE 7.1 – $a_j \in [1, 30] c_j \in [1, 40]$

	$b_j = l_j = L = 40$			$b_j = l_j = L = 60$		
	HSD	$HMSD$	AG	HSD	$HMSD$	AG
$n = 20$ $Best$	61	73	99	88	90	98
opt	10	11	11	21	21	22
$aver_{dev}$	0.085	0.081	0.075	0.005	0.005	0.004
max_{dev}	0.35	0.31	0.31	0.05	0.05	0.04
$n = 50$ $Best$	37	51	94	83	86	99
opt	15	15	15	12	12	15
$aver_{dev}$	0.09	0.08	0.07	0.003	0.006	0.056
max_{dev}	0.32	0.32	0.30	0.015	0.013	0.012
$n = 100$ $Best$	29	39	92	52	56	86
opt	5	5	6	4	6	4
$aver_{dev}$	0.09	0.09	0.09	0.003	0.003	0.02
max_{dev}	0.30	0.30	0.28	0.04	0.03	0.03
$n = 500$ $Best$	20	44	77	46	57	91
opt	5	5	8	10	12	13
$aver_{dev}$	0.10	0.09	0.08	0.001	0.001	0.000
max_{dev}	0.31	0.30	0.30	0.018	0.015	0.013
$n = 1000$ $Best$	21	40	84	53	69	89
opt	18	20	30	19	25	39
$aver_{dev}$	0.10	0.09	0.09	0.000	0.000	0.000
max_{dev}	0.31	0.29	0.29	0.016	0.014	0.013

D'après les expérimentations numériques, nous avons observé que l'heuristique multi-start descente (HMSD) et l'algorithme génétique (AG) donnent de bonnes solutions. Et que l'algorithme génétique domine l'heuristique Multi-start descente et l'heuristique simple descente (HSD) en terme de performance dans la plupart des cas. Pour le premier type d'instance, c.-à-d. le cas où les temps de traitement sur la deuxième machine sont identiques et égaux au temps de latence $b_j = l_j = L$, nous avons remarqué que les solutions obtenues par l'algorithme génétique sont très proches de l'optimum avec une déviation

TABLE 7.2 – $a_j \in [20, 60] c_j \in [10, 50]$

	$b_j = l_j = L = 40$			$b_j = l_j = L = 70$		
	HSD	$HMSD$	AG	HSD	$HMSD$	AG
$n = 20$ $Best$	32	50	96	63	74	97
opt	5	6	7	5	6	6
$\%aver_{dev}$	0.22	0.21	0.20	0.063	0.055	0.050
$\%max_{dev}$	0.39	0.38	0.37	0.23	0.21	0.21
$n = 50$ $Best$	10	22	94	45	48	99
opt	2	2	3	3	4	4
$\%aver_{dev}$	0.24	0.24	0.22	0.072	0.07	0.06
$\%max_{dev}$	0.35	0.34	0.32	0.31	0.31	0.28
$n = 100$ $Best$	7	21	88	25	34	90
opt	2	2	3	2	5	2
$\%aver_{dev}$	0.24	0.23	0.23	0.07	0.064	0.066
$\%max_{dev}$	0.35	0.35	0.36	0.236	0.231	0.22
$n = 500$ $Best$	97	98	100	35	47	88
opt	4	4	5	5	5	10
$\%aver_{dev}$	0.44	0.42	0.42	0.08	0.07	0.07
$\%max_{dev}$	0.84	0.80	0.75	0.30	0.29	0.28
$n = 500$ $Best$	90	95	100	38	49	88
opt	3	4	5	4	5	9
$\%aver_{dev}$	0.45	0.41	0.39	0.09	0.08	0.07
$\%max_{dev}$	0.80	0.75	0.70	0.31	0.29	0.28

moyenne inférieure à 0.10 et une déviation maximal inférieure à 0.3 pour la plupart des instances. Par ailleurs, plus les temps de traitement sur la première machine augmentent, plus la qualité des solutions obtenues se détériorent mais restent toujours qualifiées comme de bonnes solutions ; les déviations moyennes obtenues dans ce cas sont inférieures à 0.25 et les déviations maximales inférieures à 0.4. En ce qui concerne le second type d'instances, c.-à-d. le cas où les temps de traitement sur la deuxième machine sont variables, nous avons observé que les solutions obtenues sont d'une qualité moyenne comparée aux solutions obtenues pour le premier type d'instance avec une déviation moyenne qui tourne au tour de 0.4 (voit Table 6.3 et Table 6.4).

TABLE 7.3 – $a_j \in [1, 30] c_j \in [1, 40]$

		$b_j = l_j \in [40, 70]$			$b_j = l_j \in [70, 100]$		
		HSD	$HMSD$	AG	HSD	$HMSD$	AG
$n = 20$	$Best$	53	73	99	80	85	98
	opt	10	10	13	20	25	24
	$aver_{dev}$	0.095	0.081	0.065	0.005	0.005	0.004
	max_{dev}	0.35	0.29	0.27	0.05	0.05	0.04
$n = 50$	$Best$	37	51	94	83	86	99
	opt	13	15	18	12	12	15
	$aver_{dev}$	0.10	0.08	0.07	0.003	0.006	0.056
	max_{dev}	0.33	0.32	0.30	0.015	0.013	0.012
$n = 100$	$Best$	29	37	92	50	56	87
	opt	5	5	6	4	6	4
	$aver_{dev}$	0.11	0.09	0.09	0.004	0.003	0.02
	max_{dev}	0.31	0.30	0.28	0.04	0.02	0.03
$n = 500$	$Best$	20	44	77	45	57	90
	opt	5	5	8	11	12	15
	$aver_{dev}$	0.10	0.09	0.07	0.000	0.000	0.000
	max_{dev}	0.31	0.30	0.30	0.018	0.015	0.013
$n = 1000$	$Best$	21	40	84	53	69	89
	opt	18	20	30	19	25	39
	$aver_{dev}$	0.12	0.10	0.08	0.000	0.000	0.000
	max_{dev}	0.31	0.29	0.29	0.016	0.014	0.013

TABLE 7.4 – $a_j \in [20, 60] c_j \in [10, 50]$

		$b_j = l_j \in [40, 70]$			$b_j = l_j \in [70, 100]$		
		HSD	$HMSD$	AG	HSD	$HMSD$	AG
$n = 20$	$Best$	30	50	96	63	74	97
	opt	4	6	7	5	5	6
	$aver_{dev}$	0.26	0.25	0.21	0.063	0.055	0.050
	max_{dev}	0.39	0.38	0.37	0.25	0.21	0.21
$n = 50$	$Best$	10	22	94	45	47	97
	opt	2	2	3	3	4	4
	$aver_{dev}$	0.25	0.24	0.22	0.082	0.07	0.06
	max_{dev}	0.35	0.34	0.32	0.32	0.30	0.28
$n = 100$	$Best$	8	18	88	23	36	90
	opt	2	2	3	2	5	2
	$aver_{dev}$	0.24	0.23	0.23	0.08	0.065	0.076
	max_{dev}	0.35	0.35	0.36	0.25	0.241	0.21
$n = 500$	$Best$	88	93	100	32	41	88
	opt	1	3	6	2	5	11
	$aver_{dev}$	0.41	0.42	0.39	0.10	0.07	0.06
	max_{dev}	0.84	0.80	0.75	0.30	0.31	0.29
$n = 500$	$Best$	90	95	100	37	48	89
	opt	2	4	6	4	5	9
	$aver_{dev}$	0.41	0.40	0.34	0.10	0.08	0.06
	max_{dev}	0.80	0.75	0.70	0.33	0.31	0.30

Conclusion

Dans cette thèse, nous avons considéré le problème d'ordonnancement de tâches dans un atelier de type chain-reentrant en présence d'un temps de latence dont l'objectif est de minimiser la date de fin de traitement des tâches. Nous avons traité le cas de deux machines, ce problème est noté $P_1 : F2|ChR, l_j|C_{max}$.

Comme ce problème est nouveau, on s'est focalisé sur l'étude de sa complexité. Selon les valeurs des temps de traitement des tâches et les temps de latence, deux sous-cas du problème général ont été considérées. La première concerne le cas où les temps de latence sont identiques, ce problème est noté $P_2 : F2|ChR, l_j = L|C_{max}$ tandis que la deuxième extension concerne le cas où les durées de traitement sur la deuxième machine sont égaux au time lag noté $P_3 : F2|ChR, b_j = l_j|C_{max}$.

Pour ces deux sous-cas, nous avons établi des résultats de type NP-difficulté et des algorithmes résolubles en des temps polynomiaux pour résoudre de nombreux sous-problèmes faciles.

Pour la résolution du problème, P_2, nous avons proposé des heuristiques de type algorithme de liste. Tous ces algorithmes ont été testés et comparés sur des instances générées aléatoirement.

Dans le cas du problème, P_3, nous avons proposé des heuristiques basées sur une recherche aléatoire, et une métaheuristique de type algorithme génétique avec des expérimentations numériques.

Beaucoup de problèmes ont été étudiés dans cette thèse, certains sont difficiles, d'autres

sont faciles (au sens de la complexité) et plusieurs algorithmes ont été proposés, toutefois beaucoup de questions restent en attente : améliorer les algorithmes existants, étudier d'autres sous-problèmes et proposer d'autres approches de résolution. Les résultats obtenus ouvrent la voie à de nouvelles pistes de recherches intéressantes :

1. Etendre ce travail au cas de m machines.
2. Etendre ce travail à d'autres critères d'optimalité comme la somme des dates de fin de traitement (pondérée ou non).
3. Considérer d'autres types d'ateliers open shop et job shop.
4. Résoudre le problème avec des méthodes exactes comme la méthode de Branch and Bound et la programmation dynamique.
5. Comparer les performances des algorithmes de liste avec la solution optimale pour des problèmes de petites tailles.
6. Améliorer les bornes inférieures.
7. etc.

Bibliographie

[1] A.A. Ageev and A.E. Barburin. Approximation algorithms for UET scheduling problems with exact delays. Operations Research Letters 35 (2007) 533–540.

[2] A.A. Ageev and A.V. Kononov. Approximation algorithms for scheduling problems with exact delays, In WAOA, 2006. pp.1–14.

[3] D. Ahr, J. Békési, G. Galambos, M. Oswald and G. Reinelt. An exact algorithm for scheduling identical coupled tasks. Mathematical Methods of Operations Research 59 (2004) 193–203.

[4] K. Amrouche. Flow shop à deux machines avec recirculation. Mémoire de Magister en mathématiques, spécialité : Recherche Opérationnelle, U.S.T.H.B, 2010.

[5] K. Amrouche and M. Boudhar. A genetic algorithm for a chain reentrant shop with an exact time lag. 26^{th} EURO/INFORMS July 1-4, 2103, Rome, Italie.

[6] K. Amrouche and M. Boudhar. A metaheuristic for a chain reentrant shop with an exact time lag, Conférence Internationale sur la Conception et Production intégrées 21-23 octobre 2013, Tlemcen, Algérie.

[7] K. Amrouche and M. Boudhar. Reentrant flow shop with an exact time lag, Journées Scientifiques du laboratoires RECITS, 21-22 Avril 2014, Alger, Algérie.

[8] K. Amrouche, M. Boudhar and F. Yalaoui. Complexity results of a chain reentrant shop with an exact time lag. Operational Research Practice in Africa April 20-22, 2015, Alger, Algérie.

[9] K. Amrouche, M. Boudhar and F. Yalaoui. A multi-start descent heuristic for a chain reentrant shop. Metaheuristics International Conference, June 7-10, 2015 Agadir, Marroc.

[10] K. Amrouche and M. Boudhar. Two machines flow shop with reentrance and exact time lag. RAIRO, Operation Research 50 (2016) 223-232.

[11] K. Amrouche, M. Boudhar, M. Bendraouche and F. Yalaoui. Chain-reentrant shop with an exact time lag : new results. International Journal of Production Research. under revision.

[12] R. Bellman. Dynamic programming. Princeton University Press, Princeton, USA, 1972.

[13] J. Blazewicz, K. Ecker, T. Kis, CN. Potts, M. Tanas and J. Whitehead. Scheduling of coupled tasks with unit processing times. Journal of Scheduling 13 (2010) 453–461.

[14] C. Berge. Graphes et hypergraphes, Dunod, 1971.

[15] N. Brauner, G. Finke, V. Lehoux-Lebacque, C. Potts and J. Whithead. Scheduling of coupled tasks and one-machine no-wait robotic cells. Computers and Operational Research 36 (2009) 301–307.

[16] M. Boudhar and N. Meziani. Two-stage hybrid flow shop with recirculation. International Transactions in Operational Research 17 (2010) 239–255.

[17] J. Carlier and P. Chretienne. Problèmes d'ordonnancement : modélisation, complexité et algoithmes. Masson, Paris (France) ; 1984.

[18] J. S. Chen, J. C. H. Pan, C. K. Wu. Hybrid tabu search for re-entrant permutation flow-shop scheduling problem, Expert Systems with Applications 34(3) (2008) 1924–1930.

[19] S. W. Choi and Y. D. Kim. Minimizing makespan on an m-machine re-entrant flowshop, Computers and Operations Research, 2008, 35(5) : 1684-1696.

[20] M. Dell'Amico. Shop problems with two machines and time lags. Operations Research 44 (1996) 777–787.

[21] E. Dhouib, J. Teghem and T. Loukir. Minimizing the Number of Tardy Jobs in a Permutation Flowshop Scheduling Problem with Setup Times and Time Lags Constraints, Journal of Mathematical Modelling and Algorithms 12 (2013) 85–99.

[22] J. Edmonds. Matching and a polyhedron with 0,1 vertices. Journal of Research N.B.S.B 69 (1956) 125–30.

[23] J. Fondrevelle, A. Oulamara and M.C. Portmann. Permutation flowshop scheduling problems with time lags to minimize the weighted sum of machine completion times. International journal of production economics 33 (2007) 168–176.

[24] M.R. Garey and D.S. Johnson. Computers and Intractability. W. H. Freeman and Company, New York, 1979.

[25] R.E. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan. Optimisation and approximation in deterministic sequencing and scheduling : a survey, Ann. Discrete Math. 4 (1979) 287–326.

[26] P.C. Gilmore and R.E. Gomory. Sequencing a one-state variable machine : A solvable case of the travehng salesman problem. Oper. Res 12 (1964), 655–679.

[27] S.M. Johnson. Optimal two and three-stage production schedules with setup times included. Naval Research Logistics Quarterly 1 (1954) 61–68.

[28] A.M. Kordon and D. Rebaine. Polynomial time algorithms for the UET permutation flowshop problem with time delays. Computers & Operations Research 35 (2008) 525—537.

[29] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and scheduling theory : algorithms and complexity. In S.C. Graves, P.H. Zipkin, and A.H.G Rinnooy Kan editors, Handbooks in Operations Research and Management Science, North Holland : Amesterdam, 1993.

[30] V. Lev and I. Adiri. V-shop scheduling. European J. Opnl. Res. 18 (1984) 51–56.

[31] L.G. Mitten. Sequencing n jobs on two machines with arbitrary time lags. Management Science 5 (1959) 293–298.

[32] A.J. Orman and C.N. Potts. On the Complexity of Coupled-task Scheduling. Discrete Applied Mathematics 72 (1997) 141–54.

[33] J. C.-H. Pan, J.-S. Chen. Minimizing makespan in reentrant permutation flow-shops, Journal of Operational Research Society, 2003, 57 : 642-653.

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

[35] V.J. Rayward-Smith and D. Rebaine. Analysis of heuristics for the UET two-machine flow shop problem with time delays. Computers & Operations Research 35 (2008) 3298—3310.

[36] D. Rebaine. Flow shop vs. permutation shop with time delays. Comuters & industrial engineering 48 (2005) 357—362.

[37] H. Rock. The Three-Machine No-Wait Flow Shop Is NP-Complete. Journal of the Assoclatton for Computmg Machinery 1984 31, 336–345.

[38] M. Sakarovitch. Optimisation combinatoire : théorie des graphes. Hermann, Paris, 1984.

[39] M. Sakarovitch. Optimisation combinatoire : programmation linéaire. Hermann, Paris, 1983.

[40] R.D. Shapiro. Scheduling coupled tasks. Naval Research Logistics Quarterly 27 (1980) 489–97.

[41] F.D. Vargas-Villamil and D.E. Rivera. A model predictive control approach for real-time optimization of reentrant manufacturing lines. Comput. Ind. 45 (2001) 45–57.

[42] M.Y. Wang, S.P. Sethi and S.L. Van De Velde. Minimizing makespan in a class of reentrant shops. Oper. Res. 45 (1997) 702–712.

[43] W. Yu. The two-machine flowshop problem with delays and the one-machine total tardiness problem. PhD thesis, Technische Universiteit Eindhoven, 1996.

[44] W. 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 7 (2004) 333–348.

[45] X. Zhang and S. Van de Velde. On-line two-machine open shop scheduling with time lags, European Journal of Operational Research 204 (2010) 14–19.