République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Dr.Yahia Farès de Médéa

Faculté des Sciences

Département des Mathématiques et Informatique



MEMOIRE

Présenté pour l'obtention du

Diplôme de Master en Mathématiques

Spécialité : Recherche Opérationnelle

Présenté Par

SEMRANI Hadjer & ***ANNANE Asmaa***

Thème

Problèmes d'ordonnancements sur une seule machine leurs complexités et les différentes méthodes de résolution

---

Soutenu devant le jury composé de :

M.	K.TOUMACHE	Maître-assistant	UYF Médéa	Président
Mr.	A.BENKACI	Maître assistant	UYF Médéa	Examinateur
Mr.	R. LEMDANI	Maître assistant	UYF Médéa	Examinateur
M^{me}.	S.KOUIDER AIAD	Maître-assistant	UYF Médéa	Promotrice

Juillet 2019

Remerciements

Nous tenons à remercie **ALLAH** le tout puissant clément et miséricordieux, qui nous a donné la force et le courage pour dépasser toutes les difficultés, d'avoir suivis et accomplis ce mémoire, également nous remecions infiniment nos parents, qui nous ont encouragé et nous ont aidé à suivre toutes les étapes de nos études.

Nous exprimons toute notre gratitude à notre promotrice **Mme.S.KOUIDER AÏAD**, Maître-assistant à la faculté des sciences de l'Université de Médéa, pour nous avoir encadrés, guidés tout au long de ce mémoire et aidés par son soutien, ses précieux conseils et ses encouragements.

Nous sommes très honorées par la présence au jury de **Mr.K.TOUMACHE**, Maître-assistant à la faculté des sciences de l'Université de Médéa. Nous le remercions très sincèrement d'avoir accepté d'être présidente du jury.

Nos vifs remerciements à nos enseignants, **Mr.A.BENKACI** Maître assistant à la faculté des sciences de l'Université de Médéa, d'avoir faire parti du jury, et pour nous avoir fait partager ses connaissances durant la formation de Master 2.

Nous remercions aussi **Mr.R. LEMDANI** Maître assistant à la faculté des sciences de l'Université de Médéa, qui a bien voulu accepter d'être membre du jury de soutenance en tant qu'examinateur.

Nos profonds remerciements vont également à tous les enseignants qui nous ont enseignés tout au long de notre cursus universitaire.

Finalement, nous n'oublions pas de remercier nos amies et nos collègues et à toutes les personnes qui auront contribué de près ou de loin à l'élaboration de ce mémoire ainsi qu'à la réussite de cette formidable année universitaire.

Dédicace

ALLAH le tout puissant merci d'être toujours auprès de moi, Je dédie ce travail aux êtres les plus chers à mon coeur

A mes très chers parents
Aucune dédicace, aucun mot ne pourrait exprimer à leur juste valeur la gratitude et l'amour que je vous porte. Puisse Dieu le tout puissant, vous préserver et vous accorder santé, longue vie et bonheur.
Je souhaite prouver mon grand remerciement qui ne sera jamais suffisant à elle que j'espère la rendre fière par ce travail.

La meilleure de toutes les mères Djamila, qui m'a soutenu durant toute ma vie, qui m'a toujours poussé et motivé durant mes années d'études, qui m'a appris à aimer le travail et le bon comportement, pour son amour infini et sa bienveillance jour et nuit.

Mon très cher père Ahmed, pour être le bon exemple de père par son soutien, ses encouragement et aides des mes premiers pas d'étude jusqu'a ce jour.
A mes frères Hamza, Mohamed, Ismail, Abderrahime et Ibrahime, en témoignage de la fraternité, avec mes souhaits un avenir plein de joie, de bonheur et de réussite, ainsi que toute ma grande famille.

A ma binôme Asmaa, mon intime amie et ma soeur qui n'existait pas, merci pour tous les beaux moments que nous avons vécus, dieu m'a toujours préserver pour moi.

A mes chère amies et soeurs de cœur Hadjira et Meriem, merci pour leur encouragement, leur soutien moral et pour tous les bons moments qu'on a vécus ensemble.
J'espère que notre amitié durera éternellement.

A tous mes chers collègues de la promotion 2018-2019 et a tous ce qui ont enseigné moi au long de ma vie scolaire.

A ceux que j'ai porté dans mon cœur et je n'ai pas pu citer le nom.

Hadjer

Dédicace

Dieu le tout puissant merci d'être toujours auprès de moi, du profond de mon coeur, je dédie ce travail à tous ceux qui me son chers :

A ma très chère mère SLAMA Nacéra

Quoi que je fasse ou que je dise, je ne saurai point te remercier comme il se doit. Tu m'as comblé avec ta tendresse et affection et tu n'as cessé de me soutenir et de m'encourager durant toutes les années de mes études.

Que Dieu le tout puissant te donner santé, bonheur et longue vie

A mon très cher père ANNANE Redouane

Autant de phrases et d'expressions aussi éloquentes soit-elles ne sauraient exprimer ma gratitude et ma reconnaissance.

Tu as su m'inculquer le sens de la responsabilité, de l'optimisme et de la confiance en soi face aux difficultés de la vie.Tes conseils ont toujours guidé mes pas vers la réussite.

Que Dieu le tout puissant te préserve, t'accorde santé, bonheur

A mon cher frère Amine et son épouse Mouna pour tous mes moments d'examens par leur soutien moral et ses belles surprises sucrées. Je vous souhaite un avenir plein de joie, de bonheur, de réussite et de sérénité.

A mon adorable petit frère Sisino pour toute l'ambiance dont tu m'as entouré, pour toute la spontanéité et ton élan chaleureux. Puisse Dieu le tout puissant exhausser tous tes voeux.

A ma chère binôme Hadjer ma proche amie et ma plus belle soeur merci pour son sérieux travail, leurs indéfectibles soutiens et leurs patiences infinies. J'espère que notre amitié durera éternellement

A mes amie de coeur Meriouma & Hadjira

Qui m'ont toujours à mes côtés, et qui m'ont accompagnaient durant mon chemin d'études et ils m'ont soutenu dans toute ma carrière. En souvenir de notre sincère et profonde amitié et des moments agréables que nous avons passés ensemble, je vous aime en dieu.

A toute ma famille qui n'ont jamais cessé, de formuler des prières à mon égard, de me soutenir et de m'épauler pour que je puisse atteindre mes objectifs

A tous mes enseignants et mes autres amies.

Mercie à tous ceux qui aiment Asma et ceux qui Asma aime.

Asmaa

Résumé

La théorie de l'ordonnancement est une branche de la recherche opérationnelle qui s'intéresse à organiser l'exécution de tâches, en leur attribuant des ressources et en fixant leurs dates d'exécution. Un des objectifs de la théorie de l'ordonnancement est de déterminer une solution optimale ou une bonne solution en un temps raisonnable.

Ce modeste travail a deux objectifs principaux : le premier est d'étudier quelques problèmes d'ordonnancement sur une seule machine à savoir leurs compléxités et les différentes méthodes de résolution, la deuxième est de proposer des heuristiques pour résoudre un problème pratique et d'implémenter ces heuristiques en MATLAB.

Mot clé : Ordonnancement, compléxité, heuristique, métaheuristique.

Abstract

Scheduling Theory is a branch of operational research that is interested in organizing the execution of tasks, assigning them resources, and setting their execution dates. One of the objectives of the scheduling theory is to determine an optimal solution or a good solution in a reasonable time.

This modest work has two main objectives: the first is to study some problems of scheduling on a single machine namely their completeness and the different methods of resolution, the second is to propose heuristics to solve a practical problem and to implement these heuristics in MATLAB.

Keywords : Scheduling, complexity, heuristic, metaheuristic.

Table des matières

Introduction générale **1**

1 Principes Fondamentaux de l'ordonnancement **4**

1.1 Introduction 4

1.2 Présentation du problème d'ordonnancement 4

1.2.1 Définition du terme ‹‹ Ordonnancement ›› 4

1.2.2 Définition du problème d'ordonnancement 5

1.2.3 Eléments du problème d'ordonnancement 6

1.3 Les critères d'optimisation 10

1.3.1 Classification des critères d'optimisation 10

1.3.2 Propriétés des critères 11

1.4 Représentation des problèmes d'ordonnancement 13

1.4.1 Classification des Problèmes d'ordonnancement 13

1.4.2 Ordonnancement dans les différents types d'ateliers manufacturiers . 16

1.4.3 Outils de modélisation pour l'ordonnancement 17

1.5 Théorie de la complexité 21

1.5.1 Notion de complexité 21

1.5.2 Problème de décision 21

1.5.3 Problème d'optimisation 22

1.5.4 La complexité des problèmes 23

1.5.5 Réduction entre problèmes d'ordonnancement 25

1.6 Conclusion 27

2 Les différentes méthodes de résolution des problèmes d'optimisation **28**

2.1 Introduction 28

2.2 Méthodes exactes 28

2.2.1 Méthodes de Séparation et Evaluation 29

2.2.2 Programmation linéaire 31
2.2.3 La programmation dynamique 32
2.3 Les méthodes heuristiques constructives 34
2.4 Les métaheuristiques 36
2.4.1 Les méthodes de recherche locale 37
2.4.2 Les méthodes évolutives 43
2.4.3 Méthodes hybrides 46
2.5 Conclusion 47

3 Complexité et résolution des problèmes d'ordonnancement sur une seule machine 48
3.1 Introduction 48
3.2 Problèmes faciles 48
3.2.1 Minimisation du temps de fin de traitement moyen 48
3.2.2 Minimisation de la grande Tardivité 53
3.2.3 Minimisation du nombre de tâches en retard. 57
3.2.4 Minimisation du grand temps de flow 60
3.3 Problèmes difficiles 63
3.3.1 Minimisation du temps de fin de traitement 63
3.3.2 Minimisation de la tardivité 69
3.3.3 Minimisation du retard pondéré total 70
3.4 Conclusion 74

4 Etude et Implémentation d'un problème pratique sur MATLAB 75
4.1 Introduction 75
4.2 Présentation du problème 75
4.2.1 Description du problème 75
4.2.2 Principe 76
4.2.3 Formulation mathématique du problème 76
4.2.4 Définition du problème 77
4.3 Les heuristiques proposés 77
4.3.1 Première heuristique H1 77
4.3.2 Deuxième heuristique H2 79
4.3.3 Troisième heuristique H3 82
4.4 Résolution du problème et implémentation 83

4.4.1 Choix du langage 84
4.4.2 Généralités sur le langage Matlab 84
4.4.3 Exemple illustratif 85
4.4.4 Résultats numériques 86
4.4.5 Discussion 93
4.5 Conclusion 94

Conclusion générale **95**

Annexe **97**

Liste des figures

1.1 Caractéristiques d'une tâche T_i. 7
1.2 Classification des ressources. 8
1.3 Ordonnancement sur une seule machine. 16
1.4 Ordonnancement sur des machine parallèle. 17
1.5 Diagramme de Gantt d'un ordonnancement. 17
1.6 Graphe Potentiel-Tâches d'un ordonnancement. 19
1.7 Modélisation par un réseau de PERT. 20
1.8 Représentation des classes de complexités. 22
1.9 Transformation Polynomial (réduction entre problèmes d'ordonnancement). . 26

2.1 Schéma de résolution. 34
2.2 Exploration de l'espace X par approche constructive. 35
2.3 Exploration de l'espace X par une approche de recherche locale. 37
2.4 Exploration de l'espace X par une approche évolutive. 43
2.5 Différentes stratégies d'hybridation. 46

3.1 Ordonnancement de S et S^* pour le problème $1/pmtn, r_i/\sum C_i$. 49
3.2 Ordonnancement de S et S' par échange des deux tâches T_i et T_k. 50
3.3 Application de l'algorithme de 1/outtree/$\sum w_i C_i$ 52
3.4 Solution optimale de l'exemple 3.2 53
3.5 Les contraintres de précédence de l'exemple 3.3. 54
3.6 Solution optimale de l'exemple 3.3. 55
3.7 Ordonnancement de S et S^* pour le problème $1/prec, pmtn, r_i/L_{max}$ 56
3.8 Solution optimale de l'exemple 3.4. 60
3.9 Les contraintes de précédences de l'exemple 3.5. 62
3.10 Un ordonnancement généré par l'algorithme de scharge del'exemple 3.6. . . . 65
3.11 Partition réduite au problème $1/r_i, d_i/C_{max}$. 66

3.12 Arbre de recherche par Branch *et* Bound de Bratley et al. 67
3.13 Partition réduite au problème $1/r_i/\sum C_i$. 68
3.14 Partition réduite au problème $1/r_i/L_{max}$. 69
3.15 Partition réduite au problème $1//\sum w_i T_i$ 71
3.16 Solution de l'exemple 3.7 de première borne. 73
3.17 Solution de l'exemple 3.7 de deuxième borne. 73
3.18 Procédure de Branch *and* Bound pour l'exemple 3.7. 74

4.1 Solution de l'exemple par l'heuristique $H1$. 79
4.2 Solution de l'exemple par l'heuristique $H2$. 81
4.3 Solution de l'exemple par l'heuristique $H3$. 83
4.4 Résultats de l'heuristique $H1$ pour le 1^{er} Cas. 86
4.5 Diagramme de gantt de l'heuristique $H1$ pour le 1^{er} Cas. 87
4.6 Résultats de l'heuristique $H2$ pour le 1^{er} Cas. 87
4.7 Diagramme de gantt de l'heuristique $H2$ pour le 1^{er} Cas. 88
4.8 Résultats de l'heuristique $H3$ pour le 1^{er} Cas. 88
4.9 Diagramme de gantt de l'heuristique $H3$ pour le1^{er} Cas. 89
4.10 Résultats de l'heuristique $H1$ pour le 2^{eme} Cas. 89
4.11 Diagramme de gantt de l'heuristique $H1$ pour le 2^{eme} Cas. 90
4.12 Résultats de l'heuristique $H2$ pour le 2^{eme} Cas. 90
4.13 Diagramme de gantt de l'heuristique $H2$ pour le 2^{eme} Cas. 91
4.14 Résultats de l'heuristique $H3$ pour le 2^{eme} Cas. 91
4.15 Diagramme de gantt de l'heuristique $H3$ pour le 2^{eme} Cas. 92

Liste des tableaux

1.1 Présentation des critères élémentaires. 10
1.2 Classification des critères de performances. 11
1.3 Signification et valeur des champs dans la notation de Graham. 15

4.1 Les données des exemples. 85
4.2 Les résultats des exemples. 92
A.1 Classification de quelques problèmes d'ordonnancent sur une seule machine. . 97

Liste des algorithmes

2.1 Descente Simple. 38
2.2 Recuit simulé. 41
2.3 Tabou classique. 42
2.4 Génétique simple. 44
2.5 Colonies de fourmis. 45
3.1 $1/outtree/\sum w_iC_i$ 51
3.2 $1/prec, p_i = 1/L_{max}$ 54
3.3 $1/p_i = 1/\sum w_iU_i$. 57
3.4 Pour le calcul de E. 58
3.5 $1/prec/F_{max}$. 61
3.6 $1/r_i, deliverytimes/C_{max}$ 64
4.1 $1/r_i, w_i/\sum (S - r_i) + p_i$ 78
4.2 $1/r_i, w_i/\sum (S - r_i) + p_i$ 80
4.3 $1/r_i, w_i/\sum (S - r_i) + p_i$ 82

Liste des abréviations

B&B	Branch & Bound.
EDD	Earliest Due Date.
FIFO	First In First Out.
LPT	Longest Processig Time.
MATLAB	Matrix Laboratory.
NP	NP-Polynomial.
P	Polynomial.
PL	Programmation Linéaire.
PERT	Program Evaluation and Research Task.
SAT	SATisfaisabilité.
SEP	Séparation Et évaluation.
SPT	Shortest Processing Time.
SRPT	Shortest Remaining Process Time.
WSPT	Weighted Shortest Processing Time.

Introduction générale

La recherche opérationnelle, appelée aussi science du management ou science de la décision, est une discipline relativement récente, elle apparaît principalement dans les années 1940. Elle s'inscrit aujourd'hui pleinement dans le cadre de la compétitivité économique.

La recherche opérationnelle peut être définie comme l'ensemble des méthodes et techniques d'analyse des problèmes de nature combinatoire liés à des phénomènes observés pour élaborer de meilleurs politiques de décisions menant à leur résolution, programmation mathématique, théorie des graphes, théorie des jeux, processus stochastiques, ou encore théorie de l'ordonnancement. Dans le cas de notre étude, la théorie de l'ordonnancement constitue notre centre d'intérêt.

La théorie de l'ordonnancement est une branche de la recherche opérationnelle et de l'informatique, visant à répondre aux questions de planification. Elle est incontestablement importante dans l'analyse et la résolution des problèmes posés par la gestion de production. Les premières recherches théoriques sur les problèmes d'ordonnancement remontent à la fin des années 50 qui sont probablement les plus liés au monde industriel puisqu'il s'agit de concilier, de façon optimale, des ressources limitées avec des activités dans le temps.

L'ordonnancement consiste à organiser dans le temps l'exécution d'un ensemble de tâches au moyen d'un ensemble de ressources toute en satisfaisant un ensemble de contraintes et en optimisant un ou plusieurs objectifs, un des objectifs de la théorie de l'ordonnancement est de déterminer une solution optimale ou une bonne solution en un temps raisonnable. C'est une fonction qui joue un rôle essentiel et peut être rencontrée dans de nombreux domaines les systèmes industriels de production (gestion de production, gestion de projets), les systèmes informatiques (ordonnancement de processus), les systèmes administratifs (gestion hospitalière), etc... .

Un problème d'ordonnancement désigne tout problème dans lequel l'objectif est l'allocation de ressources au cours du temps, de façon à réaliser un ensemble d'activités. De fait, la no-

tion de problème d'ordonnancement regroupe une grande variété de problèmes différents selon la nature des tâches à réaliser, les caractéristiques des ressources, les contraintes portant sur les tâches, et les critères à optimiser.

De nombreux problèmes d'intérêt rencontrés dans la pratique sont des problèmes combinatoire NP-difficiles, et de ce fait intraitables par des méthodes classiques. L'énumération de toutes les solutions réalisables pour trouver la meilleure solution s'avère impossible même pour un problème de taille moyenne. Plusieurs études se sont portées sur la résolution de problèmes déterministes, les chercheurs considèrent dans ce cas que les données du problème sont parfaitement connues, et proposent des méthodes pour générer un ordonnancement unique satisfaisant les contraintes du système et ayant des performances optimales ou proches des optimums. Ces méthodes sont alors, ou bien exactes peuvent suffire et aboutir à une solution optimale (si le problème est de petite taille et de complexité réduite), ou bien approchées constituent le moyen le plus efficace de se rapprocher le plus possible de la solution optimale (si le problème est de taille importante). Notons, toutefois, que ces problèmes demeurent ardus dans la plupart des cas et le développement de nouvelles méthodes d'ordonnancement déterministes est toujours d'intérêt dans le domaine théorique.

Dans cette étude, la contribution à la résolution des problèmes d'ordonnancement a évolué selon deux axes, le premier axe abordé dans ce mémoire concerne des problèmes d'ordonnancement à une machine pour lesquels nous présontons quelques problèmes faciles et difficiles avec leurs complexité et résolution. Le second axe concerne l'étude et l'élaboration d'un problème pratique en proposant quelques heuristiques pour avoir la résolution satisfaite.

Pour exposer notre mémoire, nous avons structuré la suite du manuscrit comme suit :

Le chapitre 1 est consacré à la présentation des notions générales liées à notre travail, nous commençons par présenter brièvement les problèmes d'ordonnancement ainsi que les notations utilisées dans la suite de ce mémoire, nous rappelons d'abord les différents éléments du problème et nous présentons les critères d'optimisation et quelques outils de modélisation. Ensuite, nous discutons sur quelque concept de la théorie de complexité.

Le chapitre 2 porte les différentes méthodes de résolution des problèmes d'ordonnancement nous nous sommes intéressés à la présentation de quelques méthodes d'optimisation développées dans la littérature, classées en trois catgéories : les méthodes exactes, les heuristiques constructives et les métaheuristiques.

Dans le chapitre 3, nous considérons quelques problèmes d'ordonnancement sur une seule machine, qui sont distingués comme des problèmes faciles et difficiles. Pour les problèmes faciles, nous avons précisé leurs complexité ainsi que quelques algorithmes de résolution et pour les problèmes difficiles, nous avons présenté leurs complexité utilisant le problème de 3-Partition pour les prouver.

Le chapitre 4 présente une implémentation d'un problème pratique qui planifier des navires au post à quai de tel sort le temps de manutention soit minimal qui est un problème difficile, pour cela nous avons proposé des heuristiques pour le résoudre, ensuite nous reportons des résultats numériques qui ont été obtenus par l'éxecution de notre programme Matlab.

Enfin, nous terminons par une conclusion générale et quelques perspectives.

Chapitre 1
Principes Fondamentaux de l'ordonnancement

1.1 Introduction

Dans ce chapitre nous nous intéressons aux problèmes d'ordonnancement d'une manière générale afin de situer la problématique de notre travail.

Dans un premier temps, il est nécessaire de définir les principes fondamentaux du problème d'ordonnancement en précisant sa définition et les différents éléments du problème, dans la deuxième section nous présentons les critères d'optimisation et ses propriétés. La troisième section représente une classification des problèmes et quelques outils de modélisation. La dernière section est consacrée à quelques concepts sur la théorie de la complexité.

1.2 Présentation du problème d'ordonnancement

1.2.1 Définition du terme << Ordonnancement >>

Due à la disponibilité d'un grand nombre de définitions de l'ordonnancement dans la littérature, nous présentons uniquement trois définitions brèves mais typiques comme illustré ci-dessous :

Définition 1.1 *Ordonnancer un ensemble des tâches, c'est programmer leur exécution en leur allouant les ressources requises et en fixant leur date de début [10].*

Définition 1.2 *Le problème d'ordonnancement consiste à organiser dans le temps la réalisation des tâches, compte tenu de contraintes temporelles (délais, contraintes d'enchainement,...) et de contraintes portant sur l'utilisation et la disponibilité des ressources requises [21].*

Définition 1.3 *L'ordonnancement concerne l'allocation de ressources limitées aux activités avec l'objectif d'optimiser un ou plusieurs mesures de performance [20].*

En partant de ces trois définitions, les propriétés élémentaires d'un problème d'ordonnancement découlent directement. Un problème d'ordonnancement fait évoquer généralement les points suivants :

- Existence d'un nombre fini de tâches.
- Partage de ressources limitées entre ces tâches.
- Détermination de la date de début d'exécution pour chaque tâche.
- Respect des contraintes imposées sur les tâches et les ressources.

1.2.2 Définition du problème d'ordonnancement

Les problèmes d'ordonnancement apparaissent dans tous les domaines : informatique, construction, industrie, administration,etc... .

Nombreuses sont les définitions proposées au problème d'ordonnancement. Dans ces définitions, on retrouve l'aspect commun de l'affectation de ressources aux tâches, en recherchant une certaine optimisation.

On peut évoquer parmi ces définitions ce qui suit :

« Etant donné un ensemble de tâches à accomplir, le problème d'ordonnancement consiste à déterminer quelles opérations doivent être exécutées, et à assigner des dates et des ressources à ces opérations de façon à ce que les tâches soient, dans la mesure du possible, accomplies en temps utile, au moindre coût et dans les meilleures conditions ».

« Résoudre un problème d'ordonnancement consiste à ordonnancer i.e. programmer ou planifier dans le temps, l'exécution des tâches en leur attribuant les ressources nécessaires, matérielles ou humaines de manière à satisfaire un ou plusieurs critères préalablement définis, tout en respectant les contraintes de réalisation » [15, 21].

D'une façon générale, il y a problème d'ordonnancement quand il existe :

- Un ensemble de travaux (tâches ou encore jobs ou lots) à réaliser.
- Un ensemble de ressources à utiliser par ces travaux.
- Un programme (calendrier) à déterminer, pour allouer convenablement les ressources aux tâches [13].

1.2.3 Eléments du problème d'ordonnancement

Dans la définition du problème d'ordonnancement, quatre éléments fondamentaux interviennent : les tâches, les ressources, les contraintes et les objectifs.Alors, la formulation et la description de ce problème se fait par la détermination de ces quatre éléments dits de base.

1.2.3.1 Les tâches

Une tâche est un travail mobilisant des ressources et réalisant un progrès significatif dans l'état d'avancement du projet compte tenu du niveau de détail retenu dans l'analyse du problème [15].

Exemple

- En construction : Pose des dalles, peinture, plomberie.
- En informatique : Exécution d'un programme, impression d'un document, etc... .
- En transport ferroviaire : Occupation d'une portion de voie par un train.

Généralement chaque tâche est notée par T_i, et elle est décrite par l'ensemble des caractéristiques suivantes :

- $\mathbf{t}_i$: date de début d'exécution de la tâcheT_i.
- $\mathbf{r}_i$: date de disponibilité de la tâche T_i (dates de début au plus tôt).
- $\mathbf{p}_i$: durée d'exécution de la tâche T_i.
- $\mathbf{C}_i$: date de fin d'exécution de la tâche T_i, $\mathbf{C}_i = \mathbf{t}_i + \mathbf{p}_i$.
- $\mathbf{d}_i$: date échue de la tâche T_i (dates de fin au plus tard).
- $\mathbf{w}_i$: poids associé à la tâche.
- $\mathbf{L}_i$: la tardivité de la tâche T_i, $\mathbf{L}_i = \mathbf{C}_i - \mathbf{d}_i$.
- $\mathbf{T}_i$: retard d'exécution de la tâche T_i, $\mathbf{T}_i = \max\{\mathbf{C}_i - \mathbf{d}_i, \mathbf{0}\}$.
- $\mathbf{U}_i$: indicateur de retard de la tâche, $\mathbf{U}_i = \begin{cases} \mathbf{1} & \textbf{si } \mathbf{C}_i > \mathbf{d}_i \\ \mathbf{0} & \textbf{sinon} \end{cases}$
- $\widetilde{\mathbf{d}}_i$: la date limite (la tâche doit etre obligatoirement achevé avant la date $\widetilde{\mathbf{d}}_i$).

D'une façon plus schématique on présente la figure suivante.

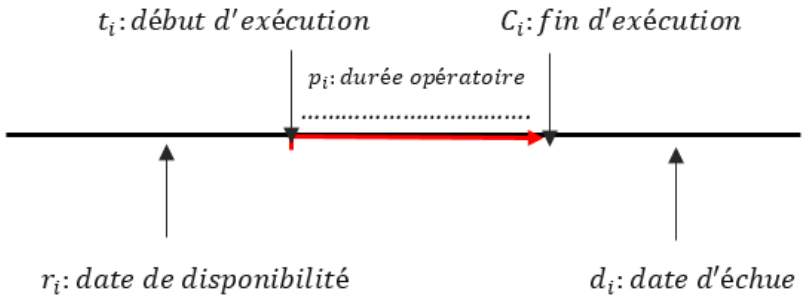


Figure 1.1 Caractéristiques d'une tâche T_i.

1.2.3.2 Les ressources

Une ressource est un moyen destiné à être utilisé pour la réalisation d'une tâche, et disponible en quantités limitées.

On distingue deux types des ressources :

a. Ressources renouvelables vs. ressources consommables

Une ressource est dite renouvelable, si après avoir été utilisée par une ou plusieurs tâches, elle est à nouveau disponible en même quantité comme : les hommes, les machines, l'espace, etc... .

Par contre, une ressource est consommable, lorsque elle devient non disponible après avoir été utilisée, comme la matière première, l'énergie, etc... .

b. Ressources disjonctives vs. ressources cumulatives

Une ressource est dite disjonctive (ou non partageable), si elle ne peut être affecté qu'à une seule tâche à la fois. Ce type concerne surtout les ressources renouvelables.

Par contre, une ressource cumulative (partageable) peut être utilisée par plusieurs tâches simultanément (équipe d'ouvriers, poste de travail, ...) [21].

Dans notre problème, les ressources qui sont des machines, sont renouvelables et disjonctives.

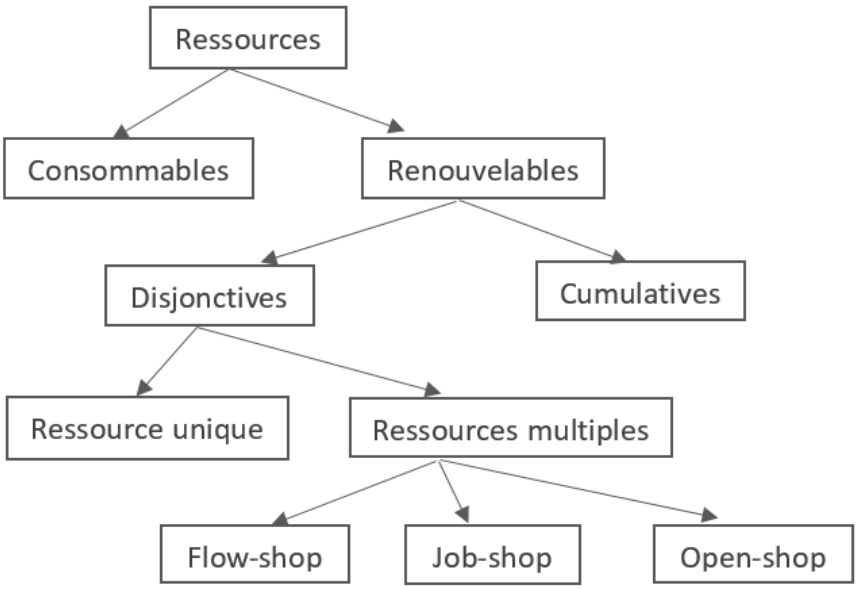


Figure 1.2 Classification des ressources.

1.2.3.3 Les contraintes

Les contraintes expriment des restrictions sur les valeurs que peuvent prendre conjointement une ou plusieurs variables de décision. Donc les contraintes représentent les limites imposées par l'environnement, tandis que l'objectif est le critère d'optimisation.

Plusieurs types de ces contraintes doivent être respectés. On distingue notamment :

a. Les contraintes de localisation temporelle

Sont issues des impératifs de gestion, et relatives aux dates limites des tâches ou du projet entier. On a surtout :

- La date de disponibilité r_i (avant laquelle la tâche ne peut pas commencer).
- La date échue d_i (avant laquelle la tâche doit être achevée).

b. Les contraintes d'enchaînement

Nous qualifions de contrainte d'enchaînement ou de succession, une contrainte qui lie le début ou la fin de deux activités par une relation linéaire.

Ce sont des contraintes imposées généralement par la cohérence technologique (les gammes opératoires dans le cas d'ateliers) qui décrivent des positionnements relatifs devant être respectés entre les tâches.

D'autres contraintes plus spécifiques entre deux tâches ou plus, telles que : la synchronisation, la simultanéité, le recouvrement,... sont également imposées dans certains systèmes.

c. Les contraintes de ressources

Une contrainte de ressources représente le fait que les activités utilisent une certaine quantité de ressources, tout au long de leur exécution.

Ces contraintes sont essentiellement soit :

- Des contraintes d'utilisation des ressources qui expriment la nature, la quantité et les caractéristiques d'utilisation de ces ressources.
- Des contraintes de disponibilité des ressources qui déterminent les quantités des ressources disponibles au cours du temps [15, 21].

On peut distinguer les contraintes suivant qu'elles sont strictes ou non. Les contraintes strictes sont des obligations à respecter. Alors que, les contraintes dites "relâchables" (appelées aussi préférences) peuvent éventuellement n'être pas satisfaites [19].

1.2.3.4 Les objectifs

Tout ordonnancement est guidé par un ou plusieurs objectifs qu'il doit chercher leur optimisation. Les objectifs que doit satisfaire un ordonnancement sont variés. D'une manière générale, on distingue plusieurs classes d'objectifs concernant un ordonnancement donné .

- **Les objectifs liés au temps** on trouve par exemple, la minimisation du temps total d'exécution, du temps moyen d'achèvement, des durées totales de réglage ou des retards par rapport aux dates de livraison, etc...
- **Les objectifs liés aux ressources** par exemple, maximiser la charge d'une ressource ou minimiser le nombre de ressources nécessaires pour réaliser un ensemble de tâches.
- **Les objectifs liés au coût** ces objectifs sont généralement de minimiser les coûts, de lancement, de production, de stockage, de transport, etc...[21].

Dans la section suivante nous reviendrons avec une description plus poussée sur les différents critères que doit satisfaire l'ordonnancement.

1.3 Les critères d'optimisation

Dans la résolution d'un problème d'ordonncemanent, on peut choisir entre deux grands types d'objectifs, visant respectivement à l'optimalité des solutions (trouver une solution optimale) ou plus simplement à leur admissibilité (trouver une meilleure solution en un temps raisonnable).

La recherche de l'optimalité impose à mesurer la qualité d'une solution de répondre aux exigences du problème étudié. Dans ce but, des critères d'optimisation sont définis et utilisés.

1.3.1 Classification des critères d'optimisation

Généralement, les critères d'optimalité les plus utilisés dans la littérature font intervenir les dates de fin d'exécution des tâches C_i comme paramètre de base en combinaison avec d'autres paramètres auxiliaires.

Des opérateurs de maximum (critère goulet) ou de somme sont adoptés pour formuler ces critères où il est envisageable sous certaines situations d'introduire une pondération pour différencier entre les priorités des tâches.

Dans les Tableaux 1.1 et 1.2, nous clarifions les critères élémentaires ainsi que la procédure de création des critères d'optimalités les plus rencontré dans le domaine de l'ordonnancement.

Dans le contexte d'une optimisation multi-objectif, il est possible de proposer d'autres critères sophistiqués utilisant l'approche somme pondérée dont le but est d'élaborer un compromis entre deux ou plusieurs fonctions objectifs intégrées dans une seule formulation [2].

Critère	Notation	Expression
Date fin	C_i	$Ci = S_i + p_i$
Temps de séjour	F_i	$Fi = Ci - r_i$
Retard algébrique	L_i	$L_i = Ci - d_i$
Retard	T_i	$T_i = \max(Ci - d_i, 0)$
Indicateur de retard	U_i	$U_i = \begin{cases} 1 \ si \ \ T_i > 0 \\ 0 \quad\quad sinon \end{cases}$
Avance	E_i	$E_i = \max(d_i - Ci, 0)$

Tableau 1.1 Présentation des critères élémentaires.

Maximum	Maximum pondéré	Somme	Somme pondérée
$\max_{1\leq i\leq n}(C_i)$	$\max_{1\leq i\leq n}(w_i \times C_i)$	$\sum_{i=1}^{i=n} C_i$	$\sum_{i=1}^{i=n}(w_i \times C_i)$
$\max_{1\leq i\leq n}(F_i)$	$\max_{1\leq i\leq n}(w_i \times F_i)$	$\sum_{i=1}^{i=n} F_i$	$\sum_{i=1}^{i=n}(w_i \times F_i)$
$\max_{1\leq i\leq n}(L_i)$	$\max_{1\leq i\leq n}(w_i \times L_i)$	$\sum_{i=1}^{i=n} L_i$	$\sum_{i=1}^{i=n}(w_i \times L_i)$
$\max_{1\leq i\leq n}(T_i)$	$\max_{1\leq i\leq n}(w_i \times T_i)$	$\sum_{i=1}^{i=n} T_i$	$\sum_{i=1}^{i=n}(w_i \times T_i)$
$\max_{1\leq i\leq n}(U_i)$	$\max_{1\leq i\leq n}(w_i \times U_i)$	$\sum_{i=1}^{i=n} U_i$	$\sum_{i=1}^{i=n}(w_i \times U_i)$
$\max_{1\leq i\leq n}(E_i)$	$\max_{1\leq i\leq n}(w_i \times E_i)$	$\sum_{i=1}^{i=n} E_i$	$\sum_{i=1}^{i=n}(w_i \times E_i)$

Tableau 1.2 Classification des critères de performances.

1.3.2 Propriétés des critères

Deux propriétés importantes des critères de performance qui sont la régularité des critères et la notion de dominance décrivent certaines relations entre un critère donné et l'ordonnancement correspondant.

1.3.2.1 Critères réguliers

Soient $C_i(x)$ et $\delta(x)$, respectivement la date de fin de la dernière opération de la tâche T_i et la valeur d'un critère δ appliqué à un ordonnancement x, δ est fonction des dates de fin d'ordonnancement C_i.

Considérons deux ordonnancements $x, x^{'}$si:

$$C_i(x) \leq C_i(x^{'}) \Rightarrow \delta(x) \leq \delta(x^{'}) \qquad \forall i = \overline{1,n}$$

Alors le critère δ est dit régulier.

Autrement dit, un critère est régulier, s'il est une fonction croissante des dates de fin d'exécution des opérations. Donc, on peut le dégrader en avançant l'exécution d'une tâche.

Parmi les critères réguliers $C_{\max}, T_{\max}, L_{\max}, \bar{L}, \bar{T}$ Par contre, $\bar{E}$, $E_{\max}$ ils sont des critères non réguliers [19].

1.3.2.2 Notion de dominance

Un sous-ensemble de solutions est dit dominant pour l'optimisation d'un critère donné, s'il contient au moins un optimum pour ce critère.

De manière analogue, on définit la dominance par rapport à un ensemble de contraintes lorsque le sous-ensemble dominant contient au moins une solution admissible (qui satisfait toutes les contraintes), si elle en existe. La recherche d'une solution optimale ou admissible peut ainsi être limitée à un sous-ensemble dominant [15].

1.3.2.3 Relations entre les critères

Le traitement de l'ordonnancement dans la littérature s'est tout d'abord orienté vers une optimisation monocritère, l'environnement manufacturier évoluant rapidement, les objectifs des entreprises se sont diversifiés et le processus d'ordonnancement est devenu de plus en plus multicritères.

Les critères que doit satisfaire un ordonnancement sont variés. L'existence de plusieurs relations d'équivalence, de similitude et de réduction entre ces critères permet de déduire certains des autres. En effet, une solution optimale pour un critère donné peut s'avérer optimale pour un autre [19].

1.3.2.4 Caractéristiques générales des ordonnancements

- Ordonnancement admissible : Un ordonnancement est dit admissible s'il respecte toutes les contraintes du problème (dates limites, précédences, limitation des ressources).

- Ordonnancement actif : Dans un ordonnancement actif aucune tâche ne peut commencer plutôt sans reporter les débuts d'une autre.

- Ordonnancement semi-actif : Est un ordonnancement dans lequel l'opération commence plutôt sans altérer la séquence. Dans un ordonnancement semi actif on ne peut avancer une tache sans modifier la séquence sur la ressource.

- Ordonnancement sans retard : Dans un ordonnancement sans retard, on ne doit pas retarder l'exécution d'une tache, si celle-ci est en attente si la ressource est disponible [2].

1.4 Représentation des problèmes d'ordonnancement

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.

1.4.1 Classification des Problèmes d'ordonnancement

À cause de l'existence d'une très grande variété de problèmes d'ordonnancement, la notation à trois champs $\boldsymbol{\alpha}/\boldsymbol{\beta}/\boldsymbol{\gamma}$ proposée initialement par Graham et all [18].

- **Champ $\boldsymbol{\alpha}$: Ressources**

Permet d'obtenir les informations relatives aux ressources, le champ $\boldsymbol{\alpha}$ se compose de deux sous paramètres $\boldsymbol{\alpha}_1$ et $\boldsymbol{\alpha}_2$.

$\boldsymbol{\alpha}_1 \in \{\phi, P, Q, R, F, O, J\}$: Type de machine.

$\boldsymbol{\phi}$: Une seule machine.

P : Machine parallèle identique, la vitesse d'exécution est la même pour toutes les machines M_j et tous les tâches T_i.

Q : Machine parallèle uniforme, chaque machine M_j a une vitesse d'exécution propre et constante et la vitesse d'exécution est la même pour tous les tâches T_i d'une même machine M_j.

R : Machine parallèle générale, la vitesse d'exécution est différente pour chaque machines M_j et pour chaque tâche T_i.

F : Machine spécialisée (système flow shop), chaque tâche doit être exécutée sur l'ensemble des machines disposées en série, selon un même routage.

O : Machine spécialisée (système open shop), les tâches doivent être exécutées sur certaines machines sans restriction sur le routage des tâches.

J : Machine spécialisée (système job shop), les tâches doivent être exécutées sur l'ensemble des machines disposées en série, mais peuvent avoir des routages différents.

$\boldsymbol{\alpha}_2 \in \{\phi, m\}$: Nombre de machine.

$\boldsymbol{\alpha}_2 = \phi$: Le nombre de machine est variable.

$\boldsymbol{\alpha}_2 = m$: Le nombre de machine est constant et égal à m [15, 32].

- **Champ $\boldsymbol{\beta}$: contraintes**

Permet d'obtenir des informations sur les contraintes, il est composé de 6 sous champs $\boldsymbol{\beta}_1\boldsymbol{\beta}_2\boldsymbol{\beta}_3\boldsymbol{\beta}_4\boldsymbol{\beta}_5\boldsymbol{\beta}_6$:

$\boldsymbol{\beta}_1 \in \{\phi, pmtn\}$ indique l'autorisation ou non de la préemption.

$\boldsymbol{\beta}_1 = \phi$: La préemption n'est pas autorisée (lorsqu'une opération est commencée, elle doit être terminée avant de pouvoir exécuter une autre opération sur la même machine)

$\boldsymbol{\beta}_1 = pmtn$: La préemption est autorisée (l'exécution d'une opération peut être interrompue puis reprise sur la même machine ou sur une autre).

$\boldsymbol{\beta}_2 \in \{\phi, prec, tree, chain, etc...\}$ décrit les contraintes de précédence.

$\boldsymbol{\beta}_2 = \phi$: Pas de contraintes de précédence
$\boldsymbol{\beta}_2 = prec$: Contraintes de précédence quelconque.
$\boldsymbol{\beta}_2 = tree$: Contraintes de précédence sont données sous forme d'un arborescence.
$\boldsymbol{\beta}_2 = chain$: Contraintes de précédence sont données sous forme de chaines.

$\boldsymbol{\beta}_3 \in \{\phi, r_i\}$ décrit les dates de disponibilité des différentes tâches.

$\boldsymbol{\beta}_3 = \phi$: Toutes les dates de disponibilité sont nulles.
$\boldsymbol{\beta}_3 = r_i$: Il existe des dates de disponibilité pour les tâches non identique.

$\boldsymbol{\beta}_4 \in \{\phi, p_i = p, \underline{p} \leq p \leq \overline{p}\}$ définit les particularités des durées de traitement.

$\boldsymbol{\beta}_4 = \phi$: Les temps de traitement des tâches sont quelconques et non identique.
$\boldsymbol{\beta}_4 = p_i = p$: Les temps de traitement des tâches sont égaux à p.
$\boldsymbol{\beta}_4 = \underline{p} \leq p \leq \overline{p}$: Le temps de traitement de chaque tâches est compris entre $\underline{p}$ et $\overline{p}$.

$\boldsymbol{\beta}_5 \in \{\phi, d_i, \widetilde{d}_i\}$

$\boldsymbol{\beta}_5 = \phi$: Il n'y a pas des tâches échues pour les tâches, ou il existe des dates échues pour les tâches dans le cas ou les critères d'optimisation est en fonction de ces derniers.
$\boldsymbol{\beta}_5 = d_i$: Il existe des dates échues pour les tâches.
$\boldsymbol{\beta}_5 = \widetilde{d}_i$: Il existe une date limite pour chaque tâche.

$\boldsymbol{\beta}_6 \in \{\phi, S_i\}$

$\boldsymbol{\beta}_6 = \phi$: Il n'existe pas de temps de préparation.
$\boldsymbol{\beta}_6 = S_i$: S'il existe un temps de préparation.

- **Champ γ : critères**

Représente le critère d'optimisation, l'objectif est de minimiser Y tel que Y est les critères de Tableau 1.2

Et reprise par la suite par plusieurs auteurs s'est rapidement émergée en tant que cadre de référence pour donner une classification claire tenant compte des caractéristiques de la plateforme et de l'environnement du problème d'ordonnancement étudié comme indiqué sur Le Tableau 1.3.

Champ	Sous Champ	Valeurs
Environnement(α)	Type de machine (α_1)	{Ø, P, Q, R, F, O, J}
	Nombre de machine (α_2)	{Ø, m}
Contraintes(β)	Préemption (β_1)	{Ø , pmtn}
	Précédence (β_2)	{Ø, prec, tree, chain, . . , etc}
	Dates de disponibilité (β_3)	{Ø, r_i}
	Durées opératoires (β_4)	{Ø, $p_i = p , \underline{p} \leq p \leq \bar{p}$ }
	Date de fin (β_5)	{Ø, $d_i, \widetilde{d_i}$}
	Temps de préparation(β_6)	{Ø, S_i}
Critères(γ)	Objectif	les critères de Tableau 1.2

Tableau 1.3 Signification et valeur des champs dans la notation de Graham.

Exemple

1. Le problème à une machine pour ordonnancer un ensemble des tâches indépendantes avec préemption, ayant le temps de traitement et des dates des disponibilités arbitraires, l'objectif est la minimisation de la date de fin de traitement est noté $\mathbf{1/pmtn}, \mathbf{r}_i/\mathbf{C}_{\max}$.

2. Le problème sur 4 machines spécialisées dans un système flow shop pour ordonnancer un ensemble des tâches indépendantes sans préemption, ayant le temps de traitement égaux à 1 et des dates des disponibilités et des dates échue arbitraires, l'objectif est de minimiser le retard moyen est noté $\mathbf{F4}/\mathbf{r}_i, \mathbf{p}_i = \mathbf{1}/\overline{\mathbf{T}}$.

1.4.2 Ordonnancement dans les différents types d'ateliers manufacturiers

Une classification très répandue des ateliers, du point de vue ordonnancement est basée sur les différentes configurations des machines. Les modèles les plus connus sont ceux d'une machine unique, de machines parallèles, d'un atelier à cheminement unique ou d'un atelier à cheminement multiple.

- **Machine unique**

Dans ce cas l'ensemble des tâches réalisées est fait par une seule machine. Les tâches alors sont composées d'une seule opération qui nécessite la même machine. L'une des situations intéressantes où on peut rencontrer ce genre de configuration est le cas où on est devant un système de production comprenant une machine goulot qui influence l'ensemble du processus. L'étude peut alors être restreinte à l'étude de cette machine.

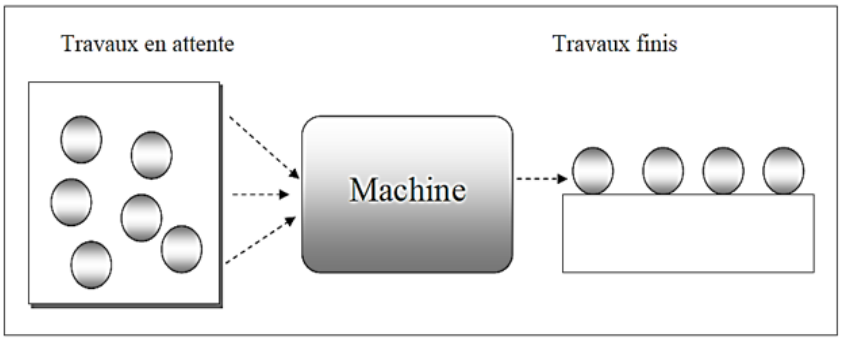


Figure 1.3 Ordonnancement sur une seule machine.

- **Machines parallèles**

Dans ce cas on dispose d'un ensemble de machines identiques pour réaliser les travaux. Les travaux se composent d'une seule opération et un travail exige une seule machine.

L'ordonnancement s'effectue en deux phases : la première phase consiste à affecter les travaux aux machines et la deuxième phase consiste à établir la séquence de réalisation sur chaque machine [32].

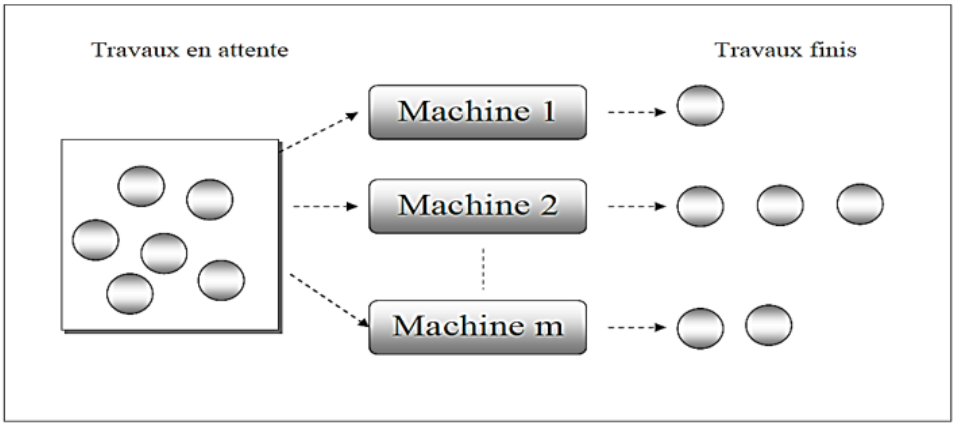


Figure 1.4 Ordonnancement sur des machine parallèle.

1.4.3 Outils de modélisation pour l'ordonnancement

Dans le domaine d'ordonnancement, plusieurs outils sont disponibles non seulement pour la représentation graphique des solutions mais également pour la spécification sans ambiguïté des données et des contraintes du problème. Dans cette section, nous présentons quelques outils de base largement employés dans la littérature des problèmes d'ordonnancement.

1.4.3.1 Diagramme de Gantt

Le diagramme de Gantt est un outil permettant de modéliser la planification des tâches nécessaires à la réalisation d'un projet, il s'agit d'un outil élaboré en 1917 par Henry L.Gantt qui permet de représenter graphiquement l'avancement du projet et constitue également un bon moyen de communication entre les différents acteurs d'un projet.

Le diagramme de Gantt présente en ordonnée la liste des tâches, notées T_i à exécuter par les machines notées M_j et en abscisse l'échelle du temps, comme le montre la figure 1.5 dans le cas où $i = 1, 2, \ldots, 10$ *et* $j = 1, 3$ [4].

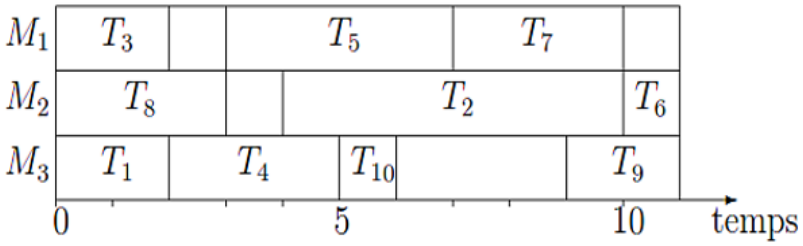


Figure 1.5 Diagramme de Gantt d'un ordonnancement.

1.4.3.2 Graphe Potentiel-Tâches

Cette outil graphique a été développé grâce à la théorie des réseaux de Pétri qui ont surtout servi à modéliser les systèmes dynamiques à évènements discrets.

Dans ce genre de modélisation, les tâches sont représentées par des nœuds et les contraintes par des arcs, comme le montre la figure 1.6 .

Ainsi, les arcs peuvent être de deux types

- **Les arcs conjonctifs** illustrant les contraintes de précédence et indiquant les durées des tâches
- **Les arcs disjonctifs** indiquant les contraintes de ressources [4].

Exemple

Tâches	Durée	Contraintes
a	3 Jours	
b	7 Jours	1achevée
c	2 Jours	2 et 5 achevée
d	4 Jours	6 achevée
e	6 Jours	4 achevée
f	5 Jours	
g	6 Jours	6 achevée
h	7 Jours	7 achevée

Pour qu'une tâche puisse commencer, il est nécessaire que toutes les tâches qui la relient à la tâche du début S du projet, soient réalisées, on définit donc :

- **La date au plus tôt de la tâche** qui correspond à la date de début, au plus tôt, de l'exécution de la tâche.

Par exemple la tâche c ne peut s'exécuter que si b ou e ont été réalisées. Donc, pour réalisées b il faut $3 + 7$ $jours$ et pour réalisées e il faut $4 + 6$ $jours$ ou $5 + 4 + 6$ $jours$.

La tâche c ne pourra commencer au plus tôt que 15 $jours$ après le début du projet c'est donc on a le plus long chemin entre c, e, d et f.

- **La durée du projet** qui correspond au plus long chemin entre s (tâche de début du projet) et $s^{'}$ (tâche de fin du projet).

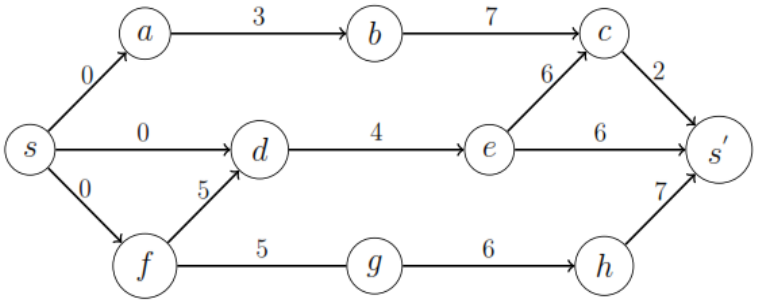


Figure 1.6 Graphe Potentiel-Tâches d'un ordonnancement.

1.4.3.3 Réseau de PERT (Program Evaluation and Research Task)

Le PERT est la technique d'évaluation et de contrôle des programmes, c'est un outil qui permet d'élaborer, de mettre à jour et de suivre un projet, il présente de façon visuelle l'enchaînement logique des tâches en vue d'en faciliter la coordination et le contrôle et d'améliorer des prévision de durée et de coût.

La méthode PERT est systématiquement utilisé pour des projets dans l'entreprise pour des problèmes d'organisation, d'ordonnancement, etc... .

Le PERT se présente donc comme un réseau compose les éléments suivant :

- Tâche : représente par une flèche.
- Etape : c'est le début et la fin d'une tâche qui représente par une cercle.
- Tâche fictive: représente par une flèche en pointilles.

Cette représentation, semblable à la précédente, permet de représenter une tâche par un arc, auquel est associé un chiffre qui représente la durée de la tâche. Entre les arcs, figurent des cercles, appelés sommets ou événements, qui marquent l'aboutissement d'une ou de plusieurs tâches. Ces cercles sont numérotés afin de suivre l'ordre de succession des divers évènements [4].

Exemple On prend les même données de tableau de l'exemple précédent, on va regroupée les tâche dans un autre tableau en deux classes, tâches antérieures et tâches postérieures.

Tâches antérieures	*Tâches*	*Tâches postérieures*
/	T_1	T_2
T_1	T_2	T_3
T_2, T_5	T_3	/
T_6	T_4	T_5
T_4	T_5	T_3
/	T_6	T_4, T_7
T_6	T_7	T_8
T_7	T_8	/

On modélise ces données par un réseau de PERT tel que:

- Nombre des étapes est 7 (nombre des cercles).
- Nombre des tâches est 8 (nombre des flèches)

Le réseau est représenté dans la figure suivante.

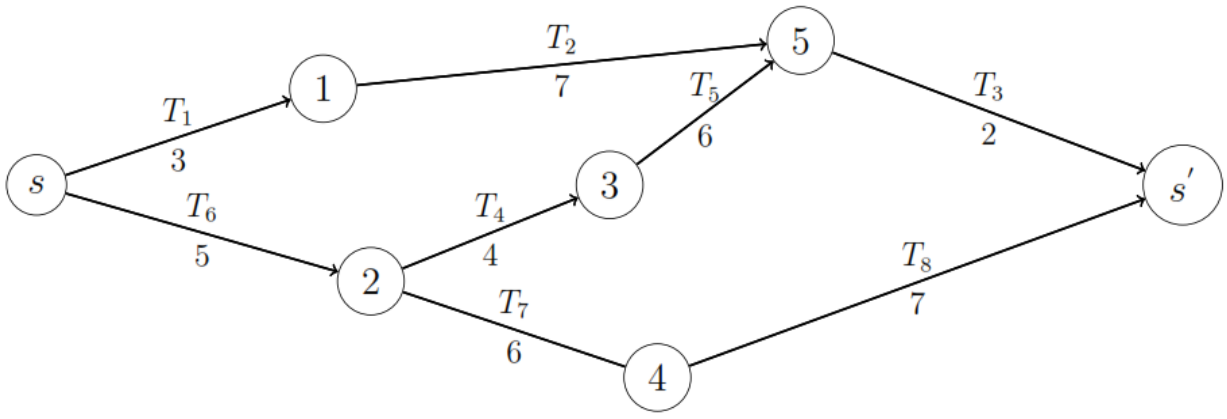


Figure 1.7 Modélisation par un réseau de PERT.

1.5 Théorie de la complexité

1.5.1 Notion de complexité

La notion de complexité apparait logiquement dans les années 1960, après la création des premiers ordinateurs. Elle a pour objectif de mesurer l'efficacité d'un algorithme, c'est-à-dire le nombre nécessaire d'étapes pour qu'une machine de Turing puisse résoudre un problème.

La théorie de la complexité a pour but d'apporter des informations sur la difficulté théorique d'un problème à résoudre, elle permet de classer les problèmes selon leur dureté algorithmique.

La complexité algorithmique est un concept fondamental qui permet de mesurer les performances d'un algorithme et de le comparer avec d'autres algorithmes réalisant les mêmes fonctionnalités.

Les problèmes d'ordonnancement correspondent soit à des problèmes de décision, soit à des problèmes d'optimisation combinatoire...[33, 25].

1.5.2 Problème de décision

Définition 1.4 *Les problèmes de décision sont des problèmes pour lesquels on cherche à répondre par "oui" ou "non" à une question donnée [28].*

Les problèmes de décision sont classés selon les trois classes suivantes :

- **La classe P**

Définition 1.5 *La classe P contient tous les problèmes de décision solubles en temps polynomial, c'est-à-dire, les problèmes de décision pour lesquels il existe un algorithme polynomial permettant de répondre à la question par un OUI [27].*

- **La classe NP**

Définition 1.6 *La classe NP contient tous les problèmes de décision solubles en temps polynomial sur une machine de Turing non déterministe, c'est-à-dire, les problèmes de décision pour lesquels il existe un algorithme tel que si on lui fournit une solution dont la réponse est ‹OUI› il est capable de le vérifier en temps polynomial [27].*

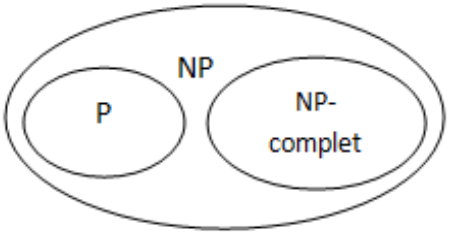


Figure 1.8 Représentation des classes de complexités.

A partir de ces deux définitions, il est clair que $P \subseteq NP$, mais jusqu'à aujourd'hui, rien ne peut affirmer que $P = NP$ ni infirmer que $P \neq NP$.

Avant de définir la classe *NP-Complet*, il est nécessaire de présenter en premier ce que c'est *une transformation polynomial.*

Définition 1.7 *Une transformation d'un problème π_1 en problème π_2 notée $\pi_1 \rightarrow \pi_2$, est dite polynomiale, s'il existe un algorithme polynomial qui transforme n'importe quelle instance I_1 du problème π_1 à une instance I_2 du problème π_2, telle que la réponse à l'instance I_1 est OUI* si et seulement si *la réponse à l'instance I_2 est OUI. Cette transformation est aussi appelée* ***réduction*** *[27].*

- **La classe NP – Complet**

Définition 1.8 *Un problème π est dit dans la classe* NP-Complet*, si π appartient à la classe NP et tous les problèmes de la classe* NP-Complet *peuvent être transformés en problème π [27].*

Comme la transformation polynomiale est transitive, il suffit de chercher un seul problème $\pi^{'}$de la classe *NP-Complet* qui peut être réduit polynomialement au problème π, afin que ce dernier devienne *NP-Complet.*

1.5.3 Problème d'optimisation

Définition 1.9 *Un problème d'optimisation exige une réponse à une question de style 'trouver un objet tel que. . . ', la réponse ici fournit l'objet recherché.*

Les problèmes d'optimisation étant les plus couramment étudiés, ils sont des problèmes pour lesquels on cherche à déterminer une solution qui optimise un critère, chaque problème d'optimisation peut être décrit sous forme d'un problème de décision, qui n'est pas nécessairement plus facile que le problème d'optimisation. Ainsi, il est possible d'analyser la difficulté d'un problème d'optimisation en considérant sa version de décision [28, 27].

Les problèmes d'optimisation ne sont pas des problèmes de décision, et ne peuvent donc appartenir à NP, à fortiori à *NP-complet*. La notion de réduction entre ces problèmes s'impose pour définir intuitivement qu'un problème π est au moins aussi difficile qu'un problème $\pi^{'}$ qui se réduit à π.

Un problème $\pi^{'}$ se réduit à π si l'on peut écrire une procédure pour résoudre $\pi^{'}$ en connaissant une sous-procédure pour résoudre π. On peut alors définir la *NP-difficulté* de la manière suivante.

Définition 1.10 *Un problème π est dit* NP-difficile *ou* NP-dur*,* si et seulement si *pour tout problème $\pi_0 \propto \pi$.*

Pour un problème d'optimisation π, on a clairement $\pi_D \propto \pi_C$. Pour cette raison on utilise en pratique le lemme suivant :

Lemme 1.1 *Soit π un problème d'optimisation. Si son problème de décision associé π_D est* NP-complet*, alors π est* NP-difficile *[32].*

La plupart des problèmes d'ordonnancement sont NP-difficile car ils sont des problèmes d'optimisation, toutefois il existe quelques cas particuliers pour lesquels on a trouvé des algorithmes polynomiaux.

Dans le domaine de l'optimisation combinatoire l'appartenance à la classe des problèmes *NP-difficile* ne signifie pas du tout la fin d'étude d'un problème [32].

1.5.4 La complexité des problèmes

Lorsque nous considérons des problèmes d'ordonnancement, la question importante est la complexité d'un problème.

Pour un nouveau problème, nous essayons souvent d'abord de développer un algorithme qui résout le problème de manière efficace, si nous ne pouvons pas trouver un tel algorithme,

il peut être utile de prouver que le problème est NP-difficile, ce qui implique qu'aucun algorithme efficace n'existe.

Dans cette section, nous passons en revue les aspects les plus importants de la théorie de la complexité qui nous permettent de décider si un problème est «facile» ou «difficile» [3].

Définition 1.11 *La complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.*

L'efficacité d'un algorithme est mesurée par l'augmentation du temps de calcul en fonction du nombre des données.

Nous avons donc deux éléments à prendre en compte :

- la taille des données.
- le temps de calcul [34].

Comme dans la plupart des cas, il est très difficile de déterminer le temps d'exécution moyen d'un algorithme pour une entrée de taille n. Pour cela, la fonction T (n) peut être définie comme une limite supérieure du temps d'exécution de l'algorithme pour toute entrée de taille n, donc on doit chercher l'efficacité d'un algorithme par rapport à un autre [3].

Définition 1.12 *Un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable à un ordre de grandeur inférieur.*

C'est pour ces raisons la notation O est souvent utilisée pour la mesure de la complexité car elle décrit une borne supérieure du cas le plus défavorable.

La complexité des algorithmes a abouti à une classification des problèmes en fonction des performances des meilleurs algorithmes connus qui les résolvent.

Les principales classes des problèmes en fonction de la complexité sont :

- $O(1)$ complexité constante (temps constant).
- $O(\log(n))$ complexité logarithmique.
- $O(n)$ complexité linéaire.
- $O(n\ \log(n))$ complexité n-logarithmique.
- $O(n^2)$ complexité quadratique.
- $O(n^p)$ complexité polynomiale.
- $O(2^n)$ complexité exponentielle (problèmes très difficiles) [34].

Un problème est classé comme «facile» (facile à résoudre) s'il peut être résolu par **un algorithme polynomial.**

Et il est classé dans la catégorie «Difficile» (insoluble) si le problème de décision est NP-complet, donc :

Pour prouver la NP-complétude d'un problème A, il suffit :

- de prouver qu'il est dans NP.
- et de prouver que $B \leq A$ pour un problème B que l'on sait déjà NP-complet [34].

Il s'avère que la classe NP possède une multitude de problèmes complets.

SAT a été le premier problème montré NP-complet par Cook-Levin en 1971, La NP-complétude du SAT assure qu'aucun algorithme pour ce problème ne peut être efficace.

Tous les problèmes NP- difficiles ne sont pas de difficulté identique, on rencontre par exemple des problèmes qui ne peuvent pas être résolus en temps polynomial avec un codage binaire, mais qui peuvent l'être avec un codage unaire, ces problèmes sont dits *NP-difficiles au sens ordinaire* ou simplement NP- difficiles, les algorithmes pour cette classe de problèmes sont appelés *pseudo-polynomiaux*, pour d'autres problèmes, on ne peut pas trouver d'algorithme polynomial, quel que soit le codage, ceux-là sont dits *NP- difficiles au sens fort* [31].

1.5.5 Réduction entre problèmes d'ordonnancement

Nous donnons quelques réductions (transformation polynomiales)(La figure 1.9) entre problèmes d'ordonnancement.

Une flèche de X à Y signifie que la résolution d'un modèle avec le paramètre Y est plus difficile que celle du même modèle avec le paramètre X, ces relations nous aide à déterminer la complexité des problèmes [18].

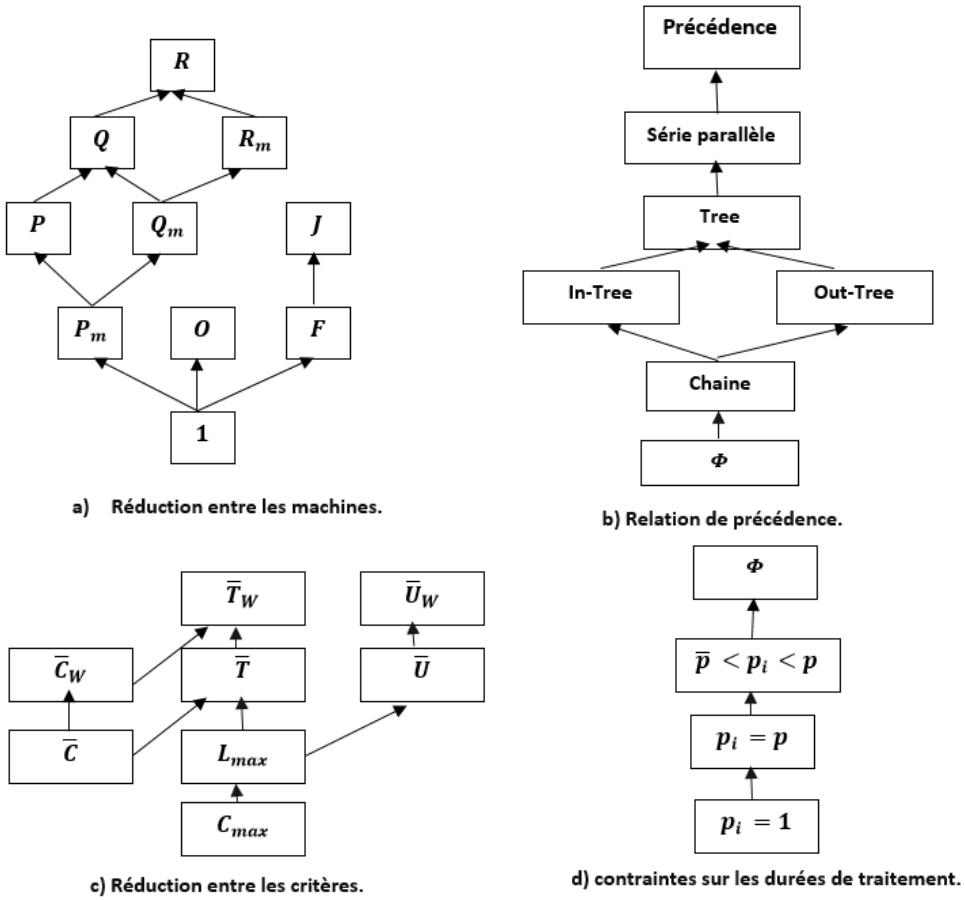


Figure 1.9 Transformation Polynomial (réduction entre problèmes d'ordonnancement).

Il est intéressant de voir comment la modification d'un seul élément de la classification d'un problème peut affecter sa complexité, la figure précédente représente certaine réduction

- **Le cas a)** représente les réductions entre les machines. Le cas le plus simple est le cas à une machine, le système devient plus complexe dans le cas de plusieurs machines en parallèle, la difficulté augmente encore, si le nombre de machines est non borné.
- **Le cas b)** indique les réductions entre les contraintes de précédence, le cas le plus simple est le cas où on n'a aucune contrainte de précédence.
- **Le cas c)** donne la relation existant entre les principaux critères d'optimisation.
- **Le cas d)** représente les réductions entre les contraintes sur les durées de traitement.

Exemple

Le problème $P//C_{\max}$ est un cas particulier du problème $Q//C_{\max}$ ou toutes les machines auraient la même vitesse. Tout algorithme développé pour $Q//C_{\max}$ peut être appliqué au problème $P//C_{\max}$ [18].

1.6 Conclusion

La partie que vous venez de lire était une introduction aux problèmes d'ordonnancement.

D'abord, en exposant notamment les notions de base concernant ces problèmes et précisant ces différents éléments : les tâches, les ressources, les contraintes et les objectifs.

Le deuxième point nous avons exposé la classification des critères d'optimisation et ses propriétés, ainsi que la représentation des problèmes qui définit les approches de classification qui s'appuient sur un ensemble des paramètres portant sur différentes caractéristiques du problème dans les différents types d'ateliers et quelques outils de modélisation (diagramme de Gantt, graphe potentiel, réseau de Pert).

Au final, nous avons élaboré un survol de la complexité des problèmes d'ordonnancement (les classes de complexité, réduction des problèmes, etc. . .).

Chapitre 2
Les différentes méthodes de résolution des problèmes d'optimisation

2.1 Introduction

Du à l'importance des problèmes d'optimisation combinatoire, de nombreux chercheurs ont porté un grand intérêt à leur résolution, ainsi que la résolution optimale d'un problème d'ordonnancement est de trouver une séquence d'exécution des tâches sur des machines de sorte qu'un certain critère d'optimisation atteigne sa valeur optimale.

La diversité des problèmes combinatoires a conduit à une grande variété de méthodes de résolution alors dans ce chapitre nous exposons les grandes lignes des méthodes les plus connues classées en trois catégories : les méthodes exactes qui garantissent la complétude de la résolution, les heuristiques constructives et les métaheuristiques qui perdent la complétude pour gagner en efficacité.

2.2 Méthodes exactes

On peut définir une méthode exacte comme étant une méthode qui fournit une solution optimale pour un problème d'optimisation.

L'utilisation de ce type de méthodes s'avère particulièrement intéressante dans le cas des problèmes de petites tailles. La méthode par séparation et évaluation (Branch & Bound), d'autres méthodes telles que la programmation linéaire ou la programmation dynamique

Toutes ces méthodes examinent d'une manière implicite, la totalité de l'espace de recherche pour produire la solution optimale [4].

2.2.1 Méthodes de Séparation et Evaluation

Description de la méthode

La méthode de séparation et évaluation (SEP) est basée 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'un arbre d'énumération généré dynamiquement, qui contient initialement le nœud racine représentant le problème d'origine. Pour chaque sous-problème (noeud de l'arbre), 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 calculée 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 atteinte.

Ainsi, l'exploration de certaines branches de l'arbre est coupée, ce qui permet de ne pas énumérer réellement toutes les solutions. Il faut donc remarquer que l'efficacité d'un algorithme de séparation et d'évaluation est déterminée par la qualité des bornes utilisées et par de bonnes conditions de branchement [23].

L'arbre de recherche

La solution d'un problème résolu par la méthode SEP est traditionnellement décrite comme une recherche dans un arbre, dans le quel le nœud racine correspond au problème d'origine à résoudre et chacun des autres nœuds correspond à un sous-problème du problème original.

Soit R un nœud de l'arbre, les descendants de R sont des sous-problèmes dérivés de R, et constituent d'autres sous-problèmes (d'autres nœuds) qui vont être explorés ultérieurement. Les feuilles correspondent aux solutions sous-optimales.

Si le passage du nœud R à un de ses descendants est incompatible avec les contraintes du problème, il sera supprimé, sinon sa borne est calculée, et il sera, par la suite élagué s'il ne contient aucune solution meilleure que celle disponible, sinon, il sera mémorisé avec sa borne pour une visite ultérieure [23].

Un algorithme B&B pour un problème de minimisation est composé des éléments suivants :

1. Une fonction qui calcule pour chaque nœud la meilleure borne inférieure de ce nœud.

2. Une stratégie pour sélectionner le nœud adéquat pour la recherche de la solution.

3. Une règle de branchement appliquée au nœud, en le subdivisant en plusieurs sous-ensembles de nœuds à visiter.

Calcul de la borne inférieure

La procédure qui calcule la borne inférieure est l'élément clé de n'importe quel algorithme B&B, car une qualité médiocre de la borne inférieure ne peut pas être compensée par de bons choix de la stratégie de recherche ou de la règle de branchement.

Soit f la fonction objectif, la fonction g qui calcule la borne inférieure doit vérifier les conditions suivantes :

1. $g\ (N_i) \leq f(N_i)$ pour tous les nœuds N_i de l'arbre .
2. $g\ (N_i) = f(N_i)$ pour tous les nœuds feuilles de l'arbre.
3. $g\ (N_i) \geq g(N_j)$ si le nœud i est un descendant du nœud j.

Les techniques de relaxation sont souvent utilisées dans le calcul d'une borne inférieure. Voici dans ce qui suit, les principales techniques de relaxation

- **Relaxation des contraintes :** Une technique simple de relaxation consiste à ignorer certaines contraintes du problème. On obtient alors un problème dont la solution optimale est plus facile à calculer, par exemple, certains problèmes utilisent l'autorisation de la préemption comme une relaxation de la contrainte de non préemption des tâches.
- **Relaxation lagrangienne :** La relaxation lagrangienne s'articule sur l'idée de relâcher les contraintes difficiles, non pas en les supprimant totalement, mais en les prenant en compte dans la fonction objectif de sorte qu'elles pénalisent la valeur des solutions qui les violent.

Stratégies de recherche

Les stratégies permettant de choisir dans l'arbre de recherche, le nœud à explorer, sont résumées dans ce qui suit :

a) Sélectionner le nœud dont la borne inférieure est la plus petite, avec l'idée que ce nœud aura le plus de chance de contenir une solution optimale.

b) Sélectionner le nœud le plus récemment créé, cette stratégie correspond à l'exploration dite « profondeur d'abord ».

c) Sélectionner le nœud le plus anciennement créé, cette stratégie correspond à l'exploration dite « largeur d'abord ».

Règles de branchement

Les règles de branchement dans un algorithme B&B, sont utilisées pour subdiviser l'ensemble des solutions possibles.

Deux règles sont utilisées dans le branchement, qui consistent à affecter les tâches une à une à partir de la fin (branchement en arrière), ou au contraire, affecter les tâches à partir du début (branchement en avant) [23].

2.2.2 Programmation linéaire

On appelle programmation Linéaire, le problème mathématique qui consiste à optimiser (maximiser ou minimiser) une fonction linéaire de plusieurs variables qui sont reliées par des relations linéaires appelées contraintes [1].

C'est l'une des techniques classiques de recherche opérationnelle, il existe de nombreux algorithmes s'inspirant des problèmes de la programmation linéaire, en particulier l'algorithme le plus utilisé qui est "l'algorithme du simplexe", on le définit comme suit:

L'algorithme du simplexe

L'algorithme du simplexe permet de résoudre les problèmes de PL en construisant tout d'abord une solution faisable qui est un sommet du polytope puis en se déplaçant selon les arêtes du polytope pour atteindre des sommets pour lesquels la valeur de l'objectif est de plus en plus grande, jusqu'à atteindre l'optimum.

Bien que cet algorithme soit efficace en pratique et qu'il soit assuré de trouver l'optimum, son comportement dans le pire cas peut être mauvais. Il est ainsi possible de construire un PL pour lequel la méthode du simplexe requiert un nombre d'étapes exponentiel en la taille du problème

2.2.3 La programmation dynamique

La « Programmation dynamique » est un paradigme de programmation, c'est-à-dire une façon particulière d'appréhender un problème algorithmique donné.

Cette méthode décompose un problème de dimension n en n problèmes de dimension. Le système est alors constitué de n étapes que l'on résout séquentiellement, le passage d'une étape à une autre se faisant à partir des lois d'évolution du système et d'une décision.

Le principe d'optimalité de Bellman 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. Ainsi, pour appliquer la programmation dynamique à un problème combinatoire, le calcul du critère pour un sous-ensemble de taille k nécessite la connaissance de ce critère pour chaque sous-ensemble de taille k -1 et porte le nombre de sous-ensembles considérés à 2n (où n est le nombre d'éléments considérés dans le problème) [1].

La programmation dynamique est donc de complexité exponentielle. Pourtant pour les problèmes NP-difficiles au sens faible, il est souvent possible de construire un algorithme de programmation dynamique pseudo-polynomial, pouvant alors être utilisé pour des problèmes de taille raisonnable [23].

Exemple 2.1 *Un problème d'ordonnancement sur une seule machine.*

On considère un ensemble de n tâches à exécuter sur une seule machine avec des temps de traitement p_i , des dates d'échues d_i et des gain g_i.

Si, dans une solution, l'exécution d'une tâche T_i se termine au plus tard à la date d_i, la tâche T_i est dit « *en avance* », le gain associé est g_i.

Le gain d'une solution est la somme des gains réalisés, donc il faut trouver *une solution de gain maximal*.

On défenit les solutions S telles que :

1) $S = HR$ où : H est une sous-liste de tâches à l'heure, R est une sous-liste de tâches en retard.

2) Les tâches de H sont placées dans l'ordre croissant de leurs dates d'échues.

On peut donc poser le problème sous la forme : Déterminer une liste H de tâches à l'heure de gain maximal.

On suppose sans perte de généralité que: $d_1 \leq d_2 \leq \ldots.. \leq d_n$.

Et on résout par *la programmation dynamique* un problème « dual » du problème initial:

Si H est une liste de tâches à l'heure, on note :

$g(H) = \sum_{T_i \in H} g_i$ le gain associé à H.

$p(H) = \sum_{T_i \in H} p_i$ la durée de H.

Pour chaque valeur de gain total e, on cherche une liste H des tâches à l'heure telle que

$g(H) = e$ et $p(H)$ soit minimum.

Schéma de récurrence avant.

On a n étapes décisionnelles $i = \overline{1,n}$ tel que l'étape numéro i concerne la tâche T_i.

elle consiste à prendre (ou non) la tâche T_i dans l'ensemble des tâches à l'heure.

$$B_i = \begin{cases} 1 & si\ T_i\ est\ prise \\ 0 & sinon \end{cases}$$

Variable d'état : e_i est le gain réalisé après les i premières décisions.

$$\textit{Coût immédiat :} \quad z_i = \begin{cases} p_i & si\ B_i = 1 \\ 0 & sinon \end{cases}$$

Ensemble $\Delta_i(e)$ des décisions possibles si $e_{i-1} = e$.

On note $P_{i-1}(e)$ la durée minimale d'une liste de tâches à l'heure appartenant à $\{T_1, \ldots., T_{i-1}\}$ et dont le gain est égal à e.

Convention :

si une telle liste n'existe pas : $P_{i-1}(e) = +\infty$

Si $P_{i-1}(e) + p_i > d_i$, alors il n'est pas possible de prendre la tâche T_i et donc $\Delta_i(e) = \{0\}$

Si $P_{i-1}(e) + p_i \leq d_i$, alors $\Delta_i(e) = \{0, 1\}$

Fonction objectif : Min $\sum z_i$.

Equation de récurrence avant :

$$p_i(e) = \begin{cases} P_{i-1}(e) & si \quad [(P_{i-1}(e) + p_i > d_i)\ ou\ (e < g_i)] \\ \min\{P_{i-1}(e), P_{i-1}(e - g_i) + p_i\} & sinon. \end{cases}$$

Initialisation :

$P_0(e) = +\infty \ \ si\ e > 0$

$P_0(0) = 0$

La valeur maximale du gain est: $\max\{e \ / \ p_n(e) < +\infty\}$

Exemple illustratif : soit le problème de 5 tâches traités sur une seule machine

T_i	T_1	T_2	T_3	T_4	T_5
p_i	5	1	4	3	6
d_i	8	8	9	9	10
g_i	3	2	2	3	4

Pour trouver *la séquence optimale* et *le gain maximal,* on doit suivre les procédures précédentes.

Les tâches sont déjà triées dans l'ordre croissant de leurs dates d'échue.

La résolution de cet exemple est présentée dans la Figure 2.1 .

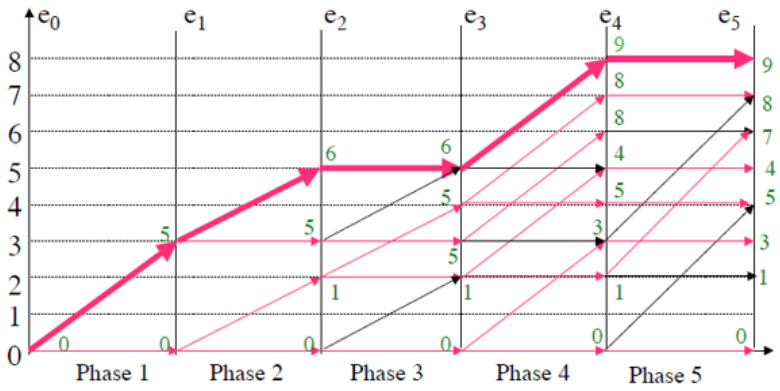


Figure 2.1 Schéma de résolution.

La séquence optimale est : $H = (T_1, T_2, T_4)$
Le gain maximal est : $g(H) = 8$

2.3 Les méthodes heuristiques constructives

Pour les problèmes de grandes tailles, les méthodes exactes ne sont pas envisageables à cause de leur temps de calcul. Il est dans ce cas possible d'utiliser des méthodes approximatives qui donnent des solutions certes sous-optimales, mais obtenues rapidement.

En pratique, l'objectif n'est pas d'obtenir un optimum absolu, mais seulement une bonne solution et la garantie de l'inexistence d'une solution sensiblement meilleure.

Pour atteindre cet objectif au bout d'un temps de calcul raisonnable, il est nécessaire d'avoir recours à des méthodes appelées "heuristiques".

Un grand nombre d'heuristiques, qui produisent des solutions proches de l'optimum, ont été développées pour les problèmes d'optimisation combinatoire difficiles, la plupart d'entre elles sont conçues spécifiquement pour un type de problème donné.

D'autres, au contraire, désormais appelées "métaheuristiques", sont capables de s'adapter à différents types de problèmes, combinatoires ou même continus.

Les méthodes par construction progressive sont des méthodes itératives où à chaque itération, une solution partielle est complétée, la plupart de ces méthodes sont de type "glouton". A chaque étape, la solution courante est complétée de la meilleure façon possible sans tenir compte de toutes les conséquences que cela entraîne au niveau du coût de la solution finale.

Le type de recherche qui est à la base d'une méthode constructive est représenté dans la figure 2.2 , l'idée consiste à diminuer la taille du problème à chaque étape, ce qui revient à se restreindre à un sous-ensemble X^K inclus dans X toujours plus petit [23].

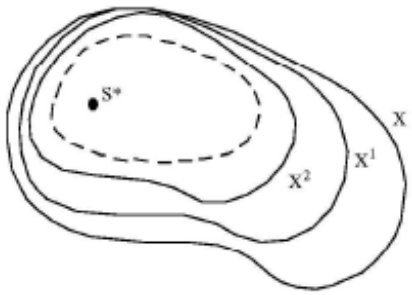


Figure 2.2 Exploration de l'espace X par approche constructive.

Parmi les méthodes constructives nous en exposons deux types :

- **Les algorithmes de liste :** Ces algorithmes consistent, dans une première phase, à calculer une liste qui donnera l'ordre de prise en compte des éléments.

Cette liste construite a priori, à partir d'un critère bien défini, n'est pas remise en cause au cours de l'ordonnancement. La deuxième phase de l'algorithme se réduit à considérer les tâches dans l'ordre de la liste pour construire l'ordonnancement.

- **Les règles de priorité :** Un deuxième type de méthode par construction progressive, consiste à choisir, au cours de la construction, l'affectation d'une tâche sur une machine en utilisant des règles de priorité. Ces méthodes peuvent également être vues comme des méthodes sérielles dynamiques où les listes sont reconstruites à chaque étape, selon un critère qui peut évoluer dans le temps.

Les règles de priorités se classent en deux familles : les règles statiques qui ne dépendent que des données de la tâche (durée, graphe de précédence) et les règles dynamiques qui dépendent de l'ordonnancement partiel construit :

- Parmi les règles statiques SPT, WSPT (règles de Smith), LPT, EDD, et FIFO.

- Les règles dynamiques tiennent compte de l'ordonnancement construit jusqu'à présent.

On peut, par exemple, chercher à placer les tâches qui génèrent le moins de temps mort dans l'ordonnancement.

Les méthodes constructives se distinguent par leur rapidité et leur grande simplicité.

On obtient en effet très rapidement une solution admissible pour un problème donné sans avoir recours à des techniques hautement sophistiquées.

Le principal défaut de ces méthodes réside malheureusement dans la qualité des solutions obtenues. Le fait de vouloir opérer à tout prix le meilleur choix à chaque étape est une stratégie dont les effets peuvent être catastrophiques à long terme.

2.4 Les métaheuristiques

Les Métaheuristiques sont des méthodes générales destinées principalement à la résolution des problèmes d'optimisation combinatoire dont le temps de résolution croît exponentiellement.

Le terme de Métaheuristiques introduit pour la première fois par Glover (1986), vient des deux mots grecs : heuriskein qui veut dire trouver et meta qui signifie au-delà [15].

Les métaheuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global, c'est-à-dire l'extremum global d'une fonction, en évaluant une certaine fonction objectif, elles se comportent comme des algorithmes de recherche, tentant d'apprendre les caractéristiques d'un problème afin d'en trouver une approximation de la meilleure solution d'une manière proche des algorithmes d'approximation.

En outre, elles n'ont pas de critère d'arrêt naturel. Elles utilisent des critères d'arrêt tels le temps de calcul maximum ou le nombre d'itérations [7].

Elles sont représentées essentiellement par les méthodes de recherche locale comme le recuit simulé et la recherche tabou, et les algorithmes évolutifs comme les algorithmes génétiques et les colonies de fourmis. Grâce à ces métaheuristiques, on peut proposer aujourd'hui des solutions approchées pour des problèmes d'optimisation classiques de plus grande taille et pour de très nombreuses applications qu'il était impossible de traiter auparavant.

On constate, depuis ces dernières années, que l'intérêt porté aux métaheuristiques augmente continuellement en recherche opérationnelle et en intelligence artificielle [23].

2.4.1 Les méthodes de recherche locale

Nous exposons, dans cette section, les méthodes d'améliorations locales les plus connues. Ces méthodes sont des algorithmes itératifs qui explorent l'espace X en se déplaçant pas à pas d'une solution à une autre, une méthode de ce type débute à partir d'une solution $x_0 \in X$ choisie arbitrairement ou alors obtenue par le biais d'une méthode constructive.

Le fonctionnement d'une méthode de recherche locale est illustré de manière générale dans la figure 2.3 .

Les passages successifs d'une solution à une solution voisine définissent un chemin au travers de l'espace des solutions admissibles. La modélisation d'un problème d'optimisation et le choix du voisinage doivent être effectués de telle sorte qu'il existe au moins un "chemin" entre chaque solution $x \in X$ et une solution optimale x^*.

En effet, l'existence de tels chemins permet à la méthode de recherche locale d'atteindre une solution optimale à partir de n'importe quelle solution admissible [23].

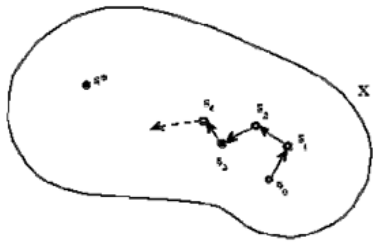


Figure 2.3 Exploration de l'espace X par une approche de recherche locale.

2.4.1.1 Méthodes de descente

C'est l'une des heuristiques de recherche locale les plus simples. Elle procède de manière itérative, en choisissant à chaque itrération un point dans le voisinage de la solution courante. S'il est meilleur, il devient la nouvelle solution courante, sinon un autre point est choisi, et ainsi de suite.

Toutefois elle comporte deux obstacles majeurs qui limitent leur efficacité:

- Suivant la taille et la structure du voisinage $N(S)$ considéré, la recherche de la meilleure solution voisine est un problème qui peut être aussi difficile que le problème initial.

- Une méthode de descente est incapable de progresser au-delà du premier minimum local rencontré, or les problèmes d'optimisation combinatoire comportent typiquement de nombreux optima locaux pour lesquels la valeur de la fonction objectif peut être fort éloignée de la valeur optimale [1, 23].

Algorithm 2.1 Descente Simple.

Début
Choisir une solution admissible initiale $s \in X$;
Poser $s^* := s$;
Tant que le critère d'arrêt n'est pas satisfait faire
Début
générer $N(s)$;
déterminer $s^{'} \in N(s)$ telle que
$f(s^{'}) = \min f(s^{''}) \ \ \forall \ \ s^{''} \in N(s)$;
$s := s^{'}$;
si $f(s) < f(s^*)$ alors $s^* := s$;
sinon le critère d'arrêt est satisfait
Fin;
Fin.

2.4.1.2 Recuit simulé

Est une méthode empirique (métaheuristique) destinée à résoudre au mieux les problèmes dits d'optimisation. Cette méthode est une technique de recherche locale inspirée du processus utilisé en métallurgie. Ce processus alterne des cycles de refroidissement lents et de réchauffage ou de recuit qui tendent à minimiser l'énergie physique. Le recuit simulé compter sur l'algorithme de Metropolis-Hastings, qui permettre la description du développement du système thermodynamique [1].

Le principe général de la méthode

Le principe de la méthode s'inspire du processus d'amélioration de la qualité d'un métal solide par recherche d'un état d'énergie minimum correspondant à une structure stable de ce métal. L'état optimal correspondrait à une structure moléculaire régulière parfaite.

En partant d'une température élevée à laquelle le solide est devenu liquide, la phase de refroidissement conduit la matière liquide à retrouver sa forme solide par une diminution progressive de la température. Chaque température est maintenue jusqu'à ce que la matière trouve un équilibre thermodynamique.

L'application de ce principe à l'optimisation commence par générer une solution initiale. A chaque itération, on calcule l'énergie de l'état (de la solution), et on diminue la température. Le passage de la solution actuelle à une nouvelle (même plus mauvaise) se fait avec une certaine probabilité qui dépend de la nature des deux solutions et l'avancement de la méthode.

Processus et ingrédients de la méthode

Le processus du Recuit Simulé répète une procédure itérative qui cherche des configurations de coût plus faible tout en acceptant de manière contrôlée des configurations qui dégradent la fonction de coût.

On utilise une méthode stochastique pour générer une suite d'états successifs du système en partant d'un état initial donné. Tout nouvel état est obtenu en faisant subir un déplacement (une perturbation) aléatoire au système. Soit :

- $\Delta E = E_2 - E_1$: la différence d'énergie occasionnée par une telle perturbation.
- T : la température absolue du système.
- k : une constante physique appelée : *constante de Boltzmann*, $k = 1,3805.10^{-23} J/K$.

Le nouvel état est accepté si l'énergie du système diminue ($\Delta E \leq 0$); sinon, il est accepté avec une probabilité définie par : $p = e^{-\Delta E/kT}$.

A chaque nouvelle itération, un voisin $s^{'} \in N(s)$ de la configuration courante s est généré de manière aléatoire, selon les cas, ce voisin sera soit retenu pour remplacer celle-ci, soit rejeté.

- Si ce voisin est de performance supérieure ou égale à celle de la configuration courante, i.e. $f(s^{'}) \leq f(s)$, il est systématiquement retenu.
- Dans le cas contraire, $s^{'}$ est accepté avec une probabilité p qui dépend de deux facteurs :

– De l'importance de la dégradation $\Delta f = f(s^{'}) - f(s)$, les dégradations plus faibles sont plus facilement acceptées.

– D'un paramètre de contrôle T (la température), une température élevée correspond à une probabilité plus grande d'accepter des dégradations.

La température est contrôlée par une fonction décroissante qui définit un schéma de refroidissement, le schéma de refroidissement de la température est l'une des paramètres les plus difficiles à régler

Parmi les difficultés de la méthode, la détermination de certains paramètres : la valeur initiale de la température (T_0) et le coefficient de décroissance de la température (γ).

Le réglage de ces paramètres est assez délicat et repose sur des essais, certains utilisateurs de cet algorithme posent $\gamma \in [0,85, 0,95]$.

L'algorithme suivant présente la version simplifiée du *Recuit Simulé* [15].

Algorithm 2.2 Recuit simulé.

Début
Déterminer la solution initiale s_0 et la température initiale T_0;
Poser $\hat{s} = s_0$ et $T_i = T_0$;
Calculer $f(s_0)$;
Répéter pour chaque itération $i/i = 1, ..., n$ (ou jusqu'à ce que T est proche de 0);
    Début
    choisir $s^{'} \in N(s_i)$;
    calculer $\Delta f = f(s^{'}) - f(s_i)$;
    si $\Delta f < 0$ alors $\hat{s} = s_i$;
    sinon Début
        tirer p dans [0, 1] suivant une distribution uniforme;
        si $p \leq e^{(-\Delta f/T)}$ alors $\hat{s} = s_i$;
        sinon s_i est rejetée;
        Fin;
    calculer $T = \gamma T$;
    Fin;
Retourner $\hat{s}$.
Fin.

2.4.1.3 La Recherche Tabou

Principe de base

Dans une première phase, la méthode de recherche tabou peut être vue comme une généralisation des méthodes d'amélioration locales, en partant d'une solution quelconque x appartenant à l'ensemble de solutions X, on se déplace vers une solution $x\prime$ située dans le voisinage $N(x)$. Donc l'algorithme explore itérativement l'espace de solutions X.

Afin de choisir le meilleur voisin $x\prime$ dans $N(x)$, l'algorithme évalue la fonction objectif f en chaque point $x\prime$, et retient le voisin qui améliore la valeur de la fonction objectif, ou au pire celui qui la dégrade le moins.

L'originalité de la méthode de recherche tabou, par rapport aux méthodes locales, qui s'arrêtent dès qu'il n'y a plus de voisin $x\prime$ permettant d'améliorer la valeur de la fonction

objectif f, réside dans le fait que l'on retient le meilleur voisin, même si celui-ci est plus mauvais que la solution d'où l'on vient. Ce critère autorisant les dégradations de la fonction objectif évite à l'algorithme d'être piégé dans un minimum local. Mais il induit un risque de cyclage. En effet, lorsque l'algorithme a quitté un minimum quelconque par acceptation de la dégradation de la fonction objectif, il peut revenir sur ses pas, à l'itération suivante.

Pour régler ce problème, l'algorithme a besoin d'une mémoire pour conserver pendant un moment la trace des dernières meilleures solutions déjà visitées. Ces solutions sont déclarées taboues, d'où le nom de la méthode.

Elles sont stockées dans une liste de longueur L donnée, appelée liste taboue, une nouvelle solution n'est acceptée que si elle n'appartient pas à cette liste taboue, ce critère d'acceptation d'une nouvelle solution évite le cyclage de l'algorithme, durant la visite d'un nombre de solutions au moins égal à la longueur de la liste taboue, et il dirige l'exploration de la méthode vers des régions du domaine de solutions non encore visitées [23].

L'algorithme donne les principes de base de la méthode tabou classique.

Algorithm 2.3 Tabou classique.

Début
$x :=$ *solution initiale*, $f_{\min} := f(x)$, $X_{\min} := x$;
$Tabou :=$ *liste de solution* x, *de longueur* L;
$Tabou :=$ *vide*;
répéter
générer un voisinage $N(x)$ tel que $x_i \in N(x)$ et $x_i \notin Tabou$;
$f(x^{'}) = \min_{1 \leq i \leq N} [f(x_i)]$;
ajouter $(x^{'}, Tabou)$
$x := x^{'}$
si $f(x) < f_{\min}$
$f_{\min} := f(x)$;
$X_{\min} := x$;
fin si;
jusqu'à conditions d'arrêt satisfaites
Fin.

2.4.2 Les méthodes évolutives

Contrairement aux méthodes de recherche locale qui font intervenir une solution unique, les méthodes évolutives manipulent un groupe de solutions admissibles à chacune des étapes du processus de recherche. L'idée centrale consiste à utiliser régulièrement les propriétés collectives d'un ensemble de solutions distinguables, appelé population, dans le but de guider efficacement la recherche vers de bonnes solutions dans l'espace X. En règle générale, la taille de la population reste constante tout au long du processus. Après avoir généré une population initiale de solutions, aléatoirement ou par l'intermédiaire d'une méthode constructive, une méthode évolutive tente d'améliorer la qualité moyenne de la population courante en ayant recours à des principes d'évolution naturelle.

Le processus cyclique qui est à la base d'une méthode évolutive est composé d'une phase de coopération et d'une phase d'adaptation individuelle qui se succèdent à tour de rôle. Ce formalisme nouveau s'applique à la plupart des méthodes évolutives développées à ce jour.

Le mécanisme de recherche qui est à la base d'une approche évolutive est représenté sommairement dans la figure 2.4 .

Le but est de repérer des solutions aussi bonnes que possible en manipulant à chaque étape un ensemble de solutions localisées dans différentes régions prometteuses de l'espace X.

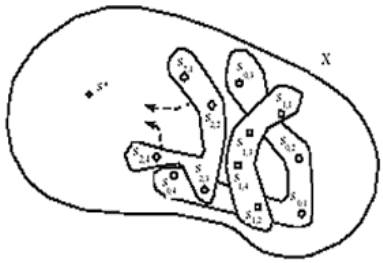


Figure 2.4 Exploration de l'espace X par une approche évolutive.

On peut distinguer deux grandes classes d'algorithmes évolutifs : les algorithmes génétiques et les colonies de fourmis. Ces méthodes se différencient par leur manière de représenter l'information et par leur façon de faire évoluer la population d'une génération à l'autre [23].

2.4.2.1 Algorithmes génétiques

Cette classe de méthodes est basée sur une imitation des phénomènes d'adaptation des êtres vivants. Les algorithmes génétiques fonctionnent sur une analogie avec la reproduction des êtres vivants.

De manière générale, les algorithmes génétiques utilisent un même principe, une population d'individus (correspondants à des solutions) évolue en même tempcomme dans l'évolution naturelle en biologie. Pour chacun des individus, on mesure sa faculté d'adaptation au milieu extérieur par le fitness. Les algorithme génétiques s'appuient alors sur trois fonctionnalités :

-La sélection : qui permet de favoriser les individus qui ont un meilleur fitness (pour nous le fitness sera le plus souvent la valeur de la fonction objectif de la solution associée à l'individu).

-Le croisement : qui combine deux solutions parents pour former un ou deux enfants (offspring) en essayant de conserver les "bonnes" caractéristiques des solutions parents.

-La mutation : qui permet d'ajouter de la diversité à la population en mutant certaines caractéristiques (gènes) d'une solution.

La représentation des solutions (le codage) est un point critique de la réussite d'un algorithme génétique, il faut bien sûr qu'il s'adapte le mieux possible au problème et à l'évaluation d'une solution [23].

Algorithm 2.4 Génétique simple.

Début

Générer une population initiale P de solutions de taille $|P| = n$

Répéter

Selection : choisir 2 solutions x et $x\prime$ par une technique de sélection

Croisement : combiner les solutions parents x et $x\prime$ pour former une solution enfant y

Mutation de y sous conditions

Choisir une solution individuelle $y\prime$ pour être remplacée dans la population

Remplacer $y\prime$ par y dans la population

Jusqu'à critère d'arrêt satisfait

Fin.

2.4.2.2 Algorithmes de colonies de fourmis

Cette nouvelle métaheuristique imite le comportement de fourmis cherchant de la nourriture. A chaque fois qu'une fourmi se déplace, elle laisse sur la trace de son passage une odeur (la phéromone). Comme la fourmi est rarement une exploratrice isolée, avec plusieurs de ses congénères, elle explore une région en quête de nourriture.

Face à un obstacle, le groupe des fourmis explore les deux côtés de l'obstacle et se retrouvent, puis elles reviennent au nid avec de la nourriture. Les autres fourmis qui veulent obtenir de la nourriture elles aussi vont emprunter le même chemin. Si celui-ci se sépare face à l'obstacle, les fourmis vont alors emprunter préférablement le chemin sur lequel la phéromone sera la plus forte.

Mais la phéromone étant une odeur elle s'évapore. Si peu de fourmis empruntent une trace, il est possible que ce chemin ne soit plus valable au bout d'un moment, il en est de même si des fourmis exploratrices empruntent un chemin plus long (pour le contournement de l'obstacle par exemple). Par contre, si le chemin est fortement emprunté, chaque nouvelle fourmi qui passe redépose un peu de phéromone et renforce ainsi la trace, donnant alors à ce chemin une plus grande probabilité d'être emprunté [23].

Algorithm 2.5 Colonies de fourmis.

```
Début
  Créer une population initiale de fourmis
  Répéter
  Pour chaque fourmi faire
    Début
      Construire une solution par une procédure de construction à
       l'aide des traces de phéromone
      Mise à jour des traces de phéromone basée sur la qualité des
      solutions trouvées
    Fin;
 Jusqu'à critère d'arrêt satisfait
Fin.
```

2.4.3 Méthodes hybrides

Que ce soit, pour élargir le spectre d'application d'une méthode, ou augmenter ces performances, le recours au principe d'hybridation est envisagé depuis longtemps, ce qui permet de créer de nouvelles méthodes hybrides plus sophistiquées et plus performantes.

Les méthodes hybrides sont des méthodes de recherche constituées d'au moins deux méthodes de recherche distinctes. Il est possible d'hybrider toutes les méthodes, y compris heuristiques et méthodes exactes, mais il faut être prudent dans le choix des méthodes à utiliser pour obtenir une bonne coopération.

Par exemple, l'hybridation d'une méthode évolutionnaire qui renforce la diversification, et une méthode de recherche locale qui assure l'intensification de la recherche.

L'hybridation peut se faire par plusieurs stratégies Figure 2.5 :

- **Hybridation séquentielle :** Consiste à exécuter séquentiellement différentes méthodes de recherche, de telle manière que les résultats d'une méthode servent de solutions initiales à la suivante.

- **Hybridation parallèle synchrone :** Est obtenue par incorporation d'une méthode de recherche particulière dans un opérateur d'une autre. C'est à dire, une méthode est englobée dans une autre. C'est une technique plus complexe que la précédente.

- **Hybridation parallèle asynchrone :** Consiste à faire évoluer en parallèle plusieurs méthodes de recherche qui peuvent être identiques. Cette coévolution nécessite un mécanisme coordinateur qui permet d'assurer une bonne coopération des méthodes.

Il est bien entendu possible de combiner plusieurs stratégies d'hybridation au sein de la même méthode [15].

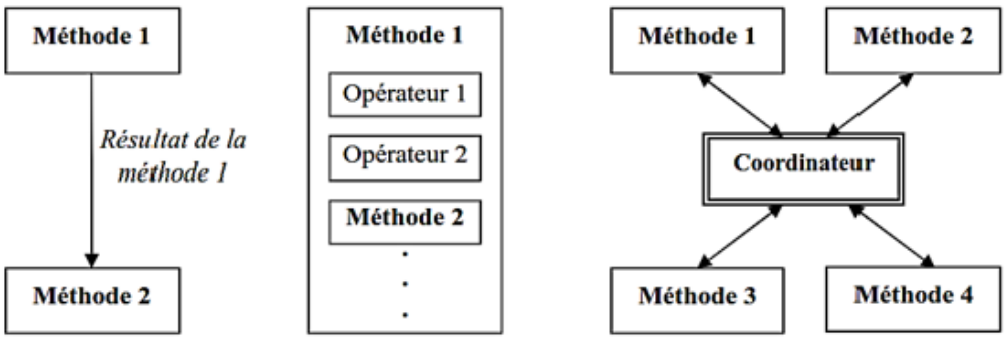


Figure 2.5 Différentes stratégies d'hybridation.

2.5 Conclusion

Dans ce chapitre nous avons présenté les méthodes de résolution des problèmes d'ordonnancement les plus connues qui sont classées en trois catégories : les méthodes exactes, les heuristiques constructives et les métaheuristique.

Parmi les méthodes exactes, nous avons distingué la méthode par séparation et évaluation (Branch and Bound), la programmation linéaire et la programmation dynamique.

Parmi les métaheuristiques, nous avons présenté les méthodes de recherche locale telles que la méthodes de descente, le recuit simulé et la méthode de recherche tabou, et les méthodes évolutionnistes, telles que les algorithmes génétiques, les algorithmes de colonies de fourmis.

L'existence de plusieurs méthodes a pour la résolution des problèmes d'ordonnancement d'une manière pertinente.

Chapitre 3
Complexité et résolution des problèmes d'ordonnancement sur une seule machine

3.1 Introduction

Nous considérons dans ce chapitre les problèmes d'ordonnancement sur une seule machine, nous précisons la complexité et la résolution de quelques problèmes '*faciles*' et '*difficiles*'.

3.2 Problèmes faciles

3.2.1 Minimisation du temps de fin de traitement moyen

3.2.1.1 Problème $1/pmtn, r_i/\sum C_i$

Ce problème peut être résolu en construisant un ordonnancement de gauche à droite en utilisant :

Règle de Smith modifiée (SRPT) :

À chaque date de disponibilité ou une date de fin de traitement d'une tâche, ordonnancer parmi les tâches restantes la tâche disponible ayant le plus petit temps de traitement restant.

La complexité est : $O(n\log(n))$

Théorème 3.1 *Un ordonnancement construit par la règle de Smith modifiée est optimale pour le problème* $1/pmtn, r_i/\sum C_i$ *[3].*

Preuve. Supposons qu'un ordonnancement actif optimal S^* coïncide avec un ordonnancement S construit par la règle SRPT jusqu'au temps t, ensuite nous avons une situation comme le montre la figure 3.1

Soit la tâche T_i dans S, qui commence au temps t et être traiter jusqu'au temps $t^{'}$.

Soit la tâche T_j dans S^* qui commence au temps t. Selon la règle SRPT, le temps de traitement restant de T_j n'est pas inférieur au temps restant de traitement du T_i.

De plus, nous avons $r_i, r_j \leq t$. Maintenant en S^* nous éliminons tous les intervalles des deux tâches T_i et T_j qui ne commencent pas avant t.

Après cela, nous ordonnançons ces tâches dans les créneaux vides à partir de temps t par traiter d'abord les parties restantes de T_i, puis les autres parties restantes de T_j. L'ordonnancement créé de cette manière est encore optimal.

Nous répétons ce processus d'échange jusqu'à ce que S et le nouvel ordonnancement optimal coïncide Jusqu'au temps $t^{'}$.

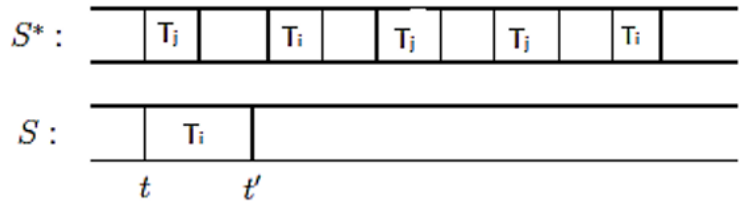


Figure 3.1 Ordonnancement de S et S^* pour le problème $1/pmtn, r_i/\sum C_i$.

3.2.1.2 Problème $1//\sum w_i\, C_i$

Le premier objectif à prendre en compte est le temps de réalisation total pondéré $\sum w_i\, C_i$. Le poids w_i de la tâche T_i peut être considéré comme un facteur d'importance, il peut représenter soit un coût de possession par unité de temps, soit la valeur déjà ajoutée à la tâche T_i. Ce problème donne lieu à l'une des règles les mieux connues en matière de la théorie d'ordonnancement WSPT. Selon cette règle, les tâches sont ordonnées dans l'ordre décroissant de w_i/p_i.

La complexité est : $O(n\log(n))$

Théorème 3.2 *La règle WSPT est optimale pour* $1//\sum w_i\, C_i$ *[23].*

Preuve. Par contradiction, supposons qu'un ordonnancement S qui n'est pas selon WSPT, soit optimal. Dans cet ordonnancement, il doit y avoir au moins deux tâches adjacentes, par exemple la tâche T_i suivi de la tâche T_k , tel que :

$\frac{w_i}{p_i} < \frac{w_k}{p_k}$ Formons un nouvel ordonnancement S' à partir de S en traitant la tâche T_k.

À l'instant t ensuite la tâche T_i . Toutes les autres tâches restent dans leur position d'origine. Le temps de réalisation total pondéré des tâches traitées avant les tâches T_i et T_k n'est pas affecté par l'échange. Le temps d'achèvement pondéré des tâches T_i et T_k sous S est :

$(t + p_i)\ w_i + (t + p + p_k)\ w_k$

Alors que sous S' est :

$(t + p_k)\ w_k + (t + p_k + p_i)\ w_i.$

Il est facile de vérifier que si $\frac{w_i}{p_i} < \frac{w_k}{p_k}$ la somme des deux temps d'achèvements pondérés de S' est strictement inférieur à cel de S. Ceci contredit l'optimalité de S

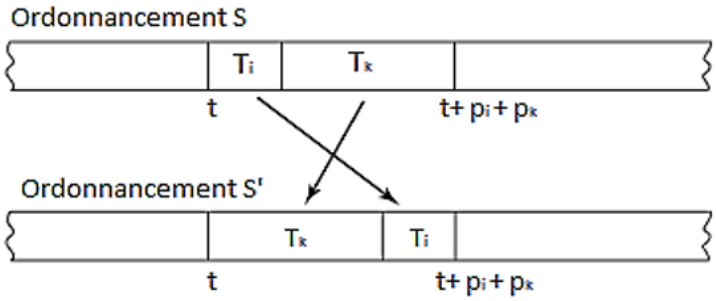


Figure 3.2 Ordonnancement de S et S' par échange des deux tâches T_i et T_k.

3.2.1.3 Problème $1/\text{outtree}/\sum \mathbf{w}_i\ \mathbf{C}_i$

Pour chaque tâche T_i, nous définissons : $q(I) = \frac{w(i)}{p(i)}$, et soit $S(I)$ l'ensemble des successeurs (pas nécessairement directs) de T_i , incluant T_i .

– Pour un ensemble de tâches $I \in \{1, ..., n\}$:

$w(I) = \sum_{i \in I} w_i$, $p(I) = \sum_{i \in I} p_i$ et $q(I) = \frac{w(I)}{p(I)}$.

– Deux sous-ensembles I, $J \subseteq \{1, ..., n\}$ sont parallèles $I \sim J$ si, pour tout $i \in I$, $j \in J$, T_i n'est pas un successeur de T_j et T_j n'est pas un successeur de T_i.

– Chaque ordonnancement est donné par une séquence π [9].

La complexité est : $O(n \log(n))$

Théorème 3.3 *Soient T_i et T_j des tâches telles que $T_i \to T_j$ et $q_j = \max\{q_k : k \in S(i)\}$. Alors il existe un ordonnancement optimal dans lequel T_i est exécuté immédiatement avant T_j [3].*

– Les conditions du théorème sont satisfaites si nous choisissons une tâche T_j différente de la racine avec une valeur q_j maximale, avec son unique parent T_i.

– Puisqu'il existe un ordonnancement où T_i est exécuté immédiatement avant T_j, nous fusionnons les nœuds i et j et faisons tous les fils de j fils supplémentaires de i.

– Le nouveau nœud i représente la sous-séquence π_i : T_i,T_j et $q(i) := q(J_i)$ *avec* $J_i = \{T_i, T_j\}$.

– Ce processus peut être appliqué récursivement.

Dans l'étape générale, chaque noeud i représente un ensemble des tâches J_i et une séquence π_i des tâches dans J_i, où T_i est la première tâche de cette séquence.

Nous sélectionnons un sommet j différent de la racine avec un valeur maximum de $q(j)$. Soit f le père unique de j dans l'arbre original. ensuite nous devons trouver un nœud i de l'arbre actuel avec $f \in J_i$.

Nous fusionnons j et i, remplaçant Ji et π_i par $J_i \cup J_j$ et $\pi_i \circ \pi_j$, où $\pi_i \circ \pi_j$ est le concaténation des séquences π_i et π_j.

Les détails sont donnés par l'algorithme suivant dans lequel :

$E(i)$: Dernier tâche de π_i.

$P(i)$: Prédécesseur de T_i dans la relation de précédence, puis prédécesseur de T_i dans une séquence optimale.

Nous supposons que $i = 1$ est la racine de l'arbre.

Algorithm 3.1 $1/outtree/\sum w_i C_i$

$w(1) = -\infty$;
pour $(i = 1$ *à* n $)$ $faire$
$\quad E(i) = i$; $J_i = \{i\}$; $q(i) = w(i)/p(i)$;
$L = \{1, \ldots, n\}$;
$Tantque(L \neq \{1\})$ $faire$
Début
trouver $T_j \in L$ *avec la valeur* $q(j)$ *la plus grande*;
$f = p(j)$;
$Trouver$ T_i *tel que* $f \in J_i$;
$w(i) = w(i) + w(j)$; $p(i) = p(i) + p(j)$;
$q(i) = w(i)/p(i)$; $P(j) = E(i)$;
$E(i) = E(j)$; $J_i = J_i \cup J_j$;
$L = L \setminus \{T_j\}$;
$Fin.$

Exemple 3.2 Considérer le problème $1/outtree/\sum w_i\, C_i$ avec 7 tâches telles que:

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7
w_i	3	6	1	4	10	3	6
p_i	3	2	5	1	2	1	3
q_i	1	3	1/5	4	5	3	2

La figure 3.3 montre le fonctionnement de l'algorithme 3.1 :

(a) représente notre problème, (b) présente l'arbre après deux itérations. Le résultat après les opérations suivantes est présenté aux (c). et (d).

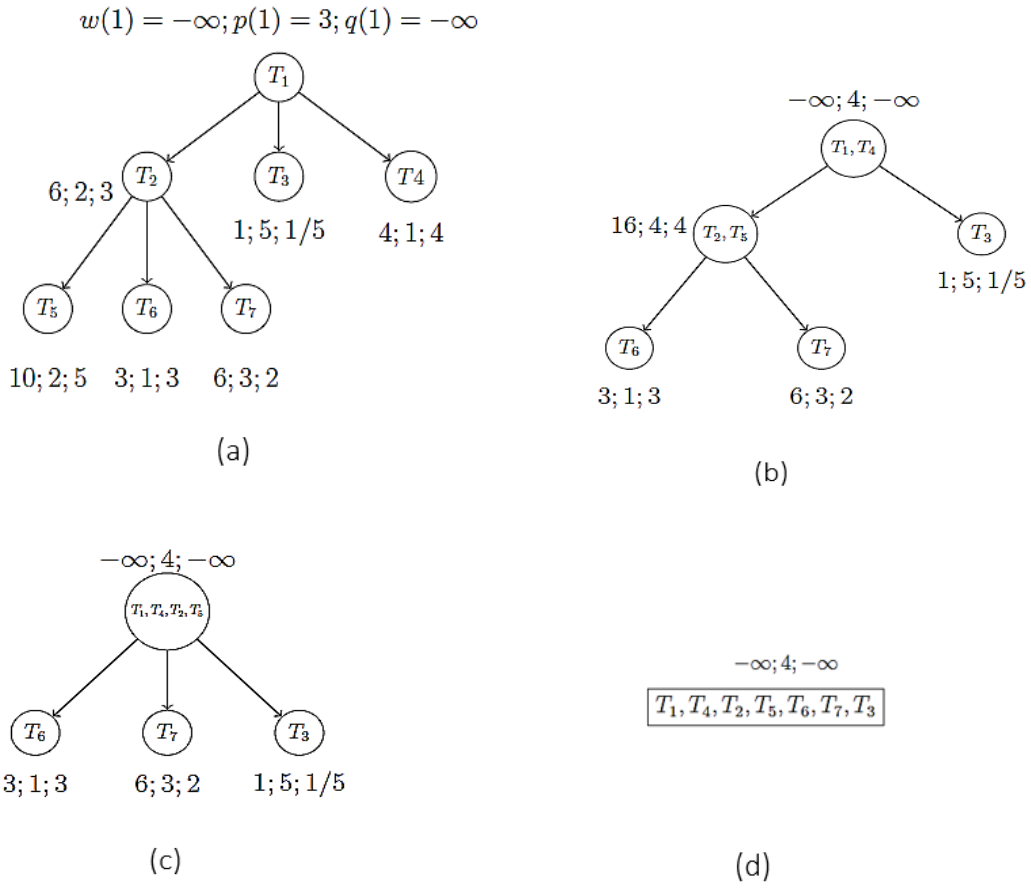


Figure 3.3 Application de l'algorithme de 1/outtree/$\sum w_i C_i$

La séquence optimale est $(T_1, T_4, T_2, T_5, T_6, T_7, T_3)$ qui est présentée dans la figure 3.4
Avec $\sum_{i=1}^{7} w_i\, C_i = 257$

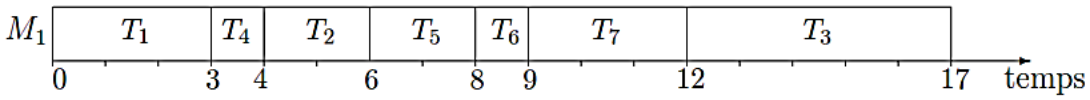


Figure 3.4 Solution optimale de l'exemple 3.2

3.2.2 Minimisation de la grande Tardivité

3.2.2.1 Problème $1/prec/L_{\max}$

Pour résoudre le problème $1/prec/L_{\max}$ il suffit de fait une modification des dates échues de telle sorte qu'ils soient compatibles avec les contraintes de précédence.

On a si $T_i \rightarrow T_j$ et $d_i < d_j$ alors la régle EDD respecte les contraintes de précédence, tel que :

La régle EDD range les tâches suivant l'ordre croissant de leurs dates échues, c'est-à-dire que, dans ce cas, EDD est optimal.

Donc pour que la fonction $L_{\max}$ n'augmente pas dans aucun ordonnancement réalisable, on a la régle de modification suivante [26]:

$$Si\ T_i \rightarrow T_j\ alors\ d_i^{'} := \min\{d_i, d_j - p_j\}\ avec\ j = n\ bas\ 1 \tag{3.1}$$

D'où l'algorithme de ce problème consiste deux étapes:

- Construire des dates échues cohérente : $d_i^{'} := \min\{d_i, \min\limits_{j/T_i \rightarrow T_j} d_j - p_j\}$; $j = n$ bas 1.
- Appliquer la règle EDD avec des dates échues modifiées.

La complexité est : $O(n^2)$

3.2.2.2 Problème $1/\mathbf{prec}, \mathbf{p}_i = 1/\mathbf{L}_{\max}$

On ajoute la contrainte $p_i = 1$ au problème précédent, et on le résolu plus efficacement en utilisant une idée de Monma 1982, définit comme suit [3]:

- Commencer par modifier les dates échues, on suppose que toutes sont des entiers.
- Toutes les tâches sont ordonnancées dans l'intervalle $[0, n]$ (car $p_i = 1$), ce qui n'implique qu'aucune tâche T_i avec $d_i \geq n$ n'est en retard.

- Pour trier les tâches, nous pouvons utiliser la méthode de *Bucket sorting,* on construit les ensembles :

$$B_k := \begin{cases} \{T_i : d_i = k\} & si\ 0 \leq k \leq n-1 \\ \{T_i : d_i \geq n\} & si\ k = n \end{cases}$$

- Peut être construit une séquence optimale π par l'algorithme suivant :

Algorithm 3.2 $1/prec, p_i = 1/L_{max}$

Pour $k = 0$ *à* n *faire* $B_k = \emptyset$;
Pour $j = 1$ *à* n *faire*
  Si $(d_j < n)$ *alors* $B_{d_j} := B_{d_j} \cup \{T_j\}$
  Sinon $B_n = B_n \cup \{T_j\}$;
$j = 1$;
Pour $k = 0$ *à* n *faire*
Tantque $(\exists\ T_i \in B_k)$ *faire*
Début
$\pi(j) = i$;
$B_k = B_k \setminus \{T_i\}$;
$j = j + 1$;
Fin.

La complexité est : $O(n)$.

Exemple 3.3 *On considère 5 tâches avec* $p_i = 1$ *et :*

T_i	T_1	T_2	T_3	T_4	T_5
d_i	8	13	2	11	7

- Les contraintres de précédence sont représentées dans la figure suivante :

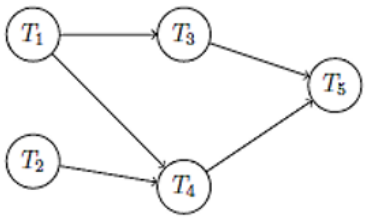


Figure 3.5 Les contraintres de précédence de l'exemple 3.3.

1) On fait la modification des dates échues d_i par la régle de modification 3.1 on obtient:

T_i	T_1	T_2	T_3	T_4	T_5
$d_i^{'}$	1	10	2	6	7

2) On applique l'algorithme 3.2 :

$B_k = \emptyset \ \forall k = \overline{0,n}$, $n = 5$;

$j = 1 \rightarrow d_1 = 1 < 5 \quad alors \quad B_1 := \{T_1\}$;

$j = 2 \rightarrow d_2 = 10 > 5 \quad alors \quad B_5 := \{T_2\}$;

$j = 3 \rightarrow d_3 = 2 < 5 \quad alors \quad B_2 := \{T_3\}$;

$j = 4 \rightarrow d_4 = 6 > 5 \quad alors \quad B_5 := \{T_2, T_4\}$;

$j = 5 \rightarrow d_5 = 7 > 5 \quad alors \quad B_5 := \{T_2, T_4, T_5\}$;

$j := 1$

$B_0 = \emptyset$;

$B_1 = \{T_1\} \rightarrow \pi(1) = 1 \rightarrow B_1 = \emptyset \rightarrow j := 2$;

$B_2 = \{T_3\} \rightarrow \pi(2) = 3 \rightarrow B_2 = \emptyset \rightarrow j := 3$;

$B_3 = \emptyset$;

$B_4 = \emptyset$;

$B_5 = \{T_2, T_4, T_5\} \rightarrow \pi(3) = 2 \quad \rightarrow \quad B_5 = \{T_4, T_5\} \rightarrow j := 4$;

$B_5 = \{T_4, T_5\} \quad \rightarrow \quad \pi(4) = 4 \quad \rightarrow \quad B_5 = \{T_5\} \quad \rightarrow j := 5$;

$B_5 = \{T_5\} \quad \rightarrow \quad \pi(5) = 5 \quad \rightarrow \quad B_5 = \emptyset$; *...Stop*

Donc la séquence optimale est: $(T_1, T_3, T_2, T_4, T_5)$ qui est présentée dans la figure 3.6 .

Avec $L_{\max} = 0$ tel que: $L_{\max} = \max\{l_i\} = \max\{C_i - d_i^{'}\}$

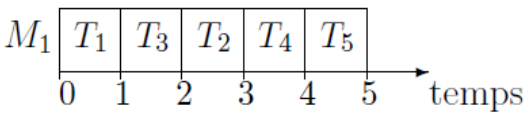


Figure 3.6 Solution optimale de l'exemple 3.3.

3.2.2.3 Problème $1/\mathbf{prec}, \mathbf{pmtn}, \mathbf{r}_i/\mathbf{L}_{\max}$

Résolution directe par *la règle EDD préemptive (Earliest Due Date)*

La règle EDD préemptive (Earliest Due Date) :

Ordonnancer les tâches en commençant à la plus petite valeur de r_i. À chaque point de décision t donné par une date de disponibilité ou par la fin d'une tâche, ordonnancer une tâche T_i telle que :

- $r_i \leq t$.
- tous les prédécesseurs de T_i sont ordonnancés.
- T_i à la plus petite date limite modifiée [26].

La complexité est : $O(n^2)$.

Théorème 3.4 *Un ordonnancement construit par* La règle EDD préemptive *est optimal pour le problème* $1/prec, pmtn, r_i/L_{\max}$ *[3].*

Preuve. Supposons que les deux ordonnancements S et S^* coïncident jusqu'au moment t, Ensuite nous avons une situation comme indiqué dans la figure 3.7 , où $s > t$ est le temps minimal auquel la tâche T_i est traité dans S^*.

Si nous éliminons la partie de la tâche T_i dans S^* à partir du moment s, déplacer la tâche T_k et les successeurs possibles de T_k ordonnancée entre t et s à droite, et réordonnancée la partie de T_i avec longueur $l = r - s$ dans les intervalles de temps créés vides, puis encore nous avons un ordonnancement optimal.

En continuant ce processus nous obtenons après un nombre fini des étapes, un ordonnancement optimal qui coïncide avec S.

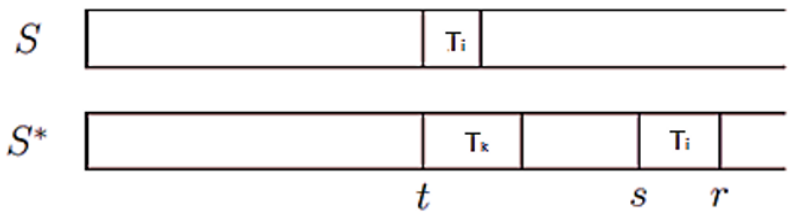


Figure 3.7 Ordonnancement de S et S^* pour le problème $1/prec, pmtn, r_i/L_{max}$.

3.2.3 Minimisation du nombre de tâches en retard.

3.2.3.1 Problème $1/\mathbf{p}_i = 1/\sum w_i\ \mathbf{U}_i$

L'algorithme suivant construit un ensemble S optimal des tâches à temps, pour obtenir un ordonnancement optimal, nous ordonnançons les tâches de S en fonction des dates échues en ordre croissant.

Tous les tâches n'appartenant pas à S sont en retard et seront ordonnancées dans un ordre arbitraire après les tâches de S.

L'idée de l'algorithme est d'essayer d'ordonnancer les tâches dans l'ordre des dates échues plus tôt, si une tâche est ordonnancée et soit en retard, alors T_i est ordonnancée et une tâche T_k avec la plus petite valeur w est supprimé de l'ensemble des tâches ordonnancées.

Dans l'algorithme suivant, nous supposons que $1 \leq d_1 \leq d_2 \leq ... \leq d_n$; et t indique temps actuelle [9].

Algorithm 3.3 $1/p_i = 1/\sum w_i U_i$.

$S := \emptyset$; $t := 0$;
Pour $j = 1$ *à* n *faire*
Début
Si $d_j \geq t$ *alors*
Début
Ajouter T_j *à* S;
$t := t + 1$;
Fin
Sinon
S'il existe une tâche $T_k \in S$ *avec* $w_k < w_j$ *alors*
Début
Supprimer la tâche T_k *de* S *où* k *est le plus grand indice tel que la valeur de* w_k *est minimal* ;
Ajouter T_j *à* S;
Fin
Fin.

La complexité est : $O(n\log(n))$

3.2.3.2 Problème $1/\mathbf{r}_i < \mathbf{r}_j \Rightarrow \mathbf{d}_i < \mathbf{d}_j / \sum \mathbf{U}_i$

Pour un ensemble de tâches donné, les dates de disponibilités et les dates échues sont appelées cohérentes, si $r_i < r_j$ implique $d_i \leq d_j$ pour toutes les tâches T_i, T_j.

Nous commençons par ordonner des tâches selon les deux, des temps de disponibilité et des dates échues non décroissants, sans perte de généralité, nous pouvons supposer que les tâches sont déjà correctement indexées, c'est-à-dire $r_1 \leq r_2 \leq ... \leq r_n$ et $d_1 \leq d_2 \leq ... \leq d_n$.

Tout ordonnancement S peut encore être décrite par une partition du groupe de tâches τ dans le temps fixé $\tau^{\leq}$ et ensemble tardif $\tau^{>}$. Les tâches de $\tau^{\leq}$ sont traitées dans l'ordre EDD, elles sont donc classées en fonction de leurs indices.

Soit $\tau^{\leq} = \{T_{k_1}, ..., T_{k_m}\}, k_1 < ... < k_m$. La date de fin de traitement C_{k_i} de la tâche T_{k_i} dans cet ordonnancement est donné par :

$C_{k_1} = r_{k_1} + p_{k_1}$

$C_{k_i} = \max\{C_{k_{i-1}}, r_{k_i}\} + p_{k_i}$; $(i = 2, ..., m)$.

Ensuite, la dernière tâche de $\tau^{\leq}$ est terminée au moment $C(\tau^{\leq}) = C_{k_m}$.

L'algorithme suivant génère des ordonnancements optimales pour les sous-ensembles $\{T_1, ..., T_i\}$ dans l'ordre de $i = 1, ..., n$.

Soit un ordonnancement optimal pour $\{T_1, ..., T_i\}$ être spécifié par les sous-ensembles $E_i \subseteq \{T_1, ..., T_i\}$ des tâches à temps. Ensuite, ensemble E_n donnera un ordonnancement optimal pour τ [3, 24].

Algorithm 3.4 Pour le calcul de E.

Debut

Ordonner les tâches en fonction des dates de disponibilités et des dates échues non décroissantes;

$E_0 := \phi$;

pour $j := 1$ *à* n *faire*

$E_j := E_{j-1} \cup \{T_j\}$; $\%S(E_j)$ *est obtenue en séquencant les tâches de* E_j *dans l'ordre EDD.*

Si $C(E_j) > d_j$ $\%$ $C(E_j)$ *est la date de fin de traitement de la dernière tâche en* $S(E_j)$.

Alors

Choisir $T_k \in E_j$ *tel que* $S(E_j - \{T_k\})$ *dans l'ordre EDD soit de longueur minimale*;

$E_j := E_j - \{T_k\}$;

Fin.

La complexité est : $O(n \log(n))$.

Exemple 3.4 *On considère 6 tâches avec :*

T_i	T_1	T_2	T_3	T_4	T_5	T_6
r_i	0	0	4	4	5	8
p_i	4	3	3	7	7	4
d_i	4	5	7	11	14	15

Les 6 tâches sont déjà ordonné en fonction des dates de disponibilités et dates échues croissant $(T_1\ T_2\ T_3\ T_4\ T_5\ T_6)$.

Initialisation :

Itération 1 :

$E_1 := E_0 \cup \{T_1\}$.

Test: $C(E_1) = 4 \geq d_1 = 4$.

$E_0 = \emptyset$.

Itération 2 :

$E_2 := E_1 \cup \{T_2\}$.

Test: $C(E_2) = 7 \geq d_2 = 5$.

Alors: on choisit $T_k \in E_j$ de telle sorte que la sous-ordonnancement obtenue pour $E_j \backslash \{T_k\}$

Dans l'ordre EDD soit de longueur minimale.

$E_2 := E_2 \backslash \{T_1\}$.

$E_2 = \{T_2\}$.

Itération 3 :

$E_3 := E_2 \cup \{T_3\}$.

$E_3 = \{T_2, T_3\}$.

Test: $C(E_3) = 7 \geq d_3 = 7$.

Itération 4 :

$E_4 := E_3 \cup \{T_4\}$.

$E_4 = \{T_2, T_3, T_4\}$.

Test: $C(E_4) = 14 > d_4 = 11$.

Alors: on choisit $T_k \in E_j$ de telle sorte que la sous-ordonnancement obtenue pour $E_j \backslash \{T_k\}$

Dans l'ordre EDD soit de longueur minimale.

$E_4 := E_4 \setminus \{T_4\}$.

$E_4 = \{T_2, T_3\}$.

Itération 5:

$E_5 := E_4 \cup \{T_5\}$.

$E_5 = \{T_2, T_3, T_5\}$.

Test: $C(E_5) = 14 \geq d_5 = 14$.

Itération 6:

$E_6 := E_5 \cup \{T_6\}$.

$E_6 = \{T_2, T_3, T_5, T_6\}$.

Test: $C(E_6) = 18 > d_6 = 15$.

Alors: on choisit $T_k \in E_j$ de telle sorte que la sous-ordonnancement obtenue pour $E_j \setminus \{T_k\}$

Dans l'ordre EDD soit de longueur minimale.

$E_6 := E_6 \setminus \{T_5\}$.

$E_6 = \{T_2, T_3, T_6\}$.

D'où :

l'ensemble des tâches en temps est $\tau^{\leq} = \{T_2, T_3, T_6\}$ et l'ensemble des tâches en retard est $\tau^{>} = \{T_1, T_4, T_5\}$.

L'ordonnancement optimal est $(T_2, T_3, T_6, T_1, T_4, T_5)$ qui est présentée dans la figure 3.8

Avec $\sum U_i = 3$.

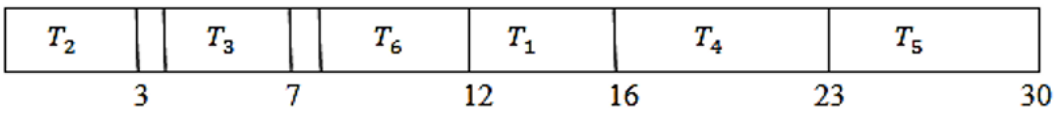


Figure 3.8 Solution optimale de l'exemple 3.4.

3.2.4 Minimisation du grand temps de flow

3.2.4.1 Problème $1/prec/F_{max}$

$F_{\max} = \max F_j(c_j)$; $j = 1, \ldots, n$. F_j monotone por tout j.

- $N = \{1, \ldots, n\}$ ensemble des tâches.

- $S \subseteq N$: tâches non ordonnancées.

- $p(s) := \sum_{j \in S} p_j$.

- $A = (a_{ij})$: matrice d'adjacence des précédences.

- $n(i)$: nombre de successeurs directs de T_i.

- $\pi(k) = i$: la tâche T_i dans la position k.

La complexité est : $O(n^2)$

Idées de base de l'algorithme

- Il suffit de construire la séquence optimale des tâches.

$$\pi := (\pi(1), \pi(2), ..., \pi(n)) .$$

- Construction en arrière.

- Règle : ordonnancer une tâche $T_j \in S$ qui n'a pas de successeur dans S et a une valeur $f_j(p(S))$ minimale comme dernière tâche de S [9].

Algorithm 3.5 $1/prec/F_{max}$.

Pour $i = 1$ *à* n *faire*
$n(i) = \sum_{j=1}^{n} a_{ij}$;
$S = \{T_1, \ldots, T_n\}$; $p = \sum_{j=1}^{n} p_j$;
Pour $k = n$ *bas à* 1 *faire*
Debut
Trouver $T_j \in S$ *avec* $n(j) == 0$ *et* $f_j = (p)$ *minimum*;
$S := S \backslash \{T_j\}$;
$\pi(k) = j$;
$p := p - p_j$;
Pour $i = 1$ *à* n *faire*
Si $a_{ij} = 1$
Alors $n(i) := n(i) - 1$;
Fin.

Exemple 3.5 *Considérer le problème* $1/prec/F_{\max}$ *avec* 5 *tâches telles que :*

T_i	T_1	T_2	T_3	T_4	T_5
r_i	0	7	1	6	3
p_i	5	5	7	6	11

- Les contraintres de précédence sont représentées dans la figure suivante :

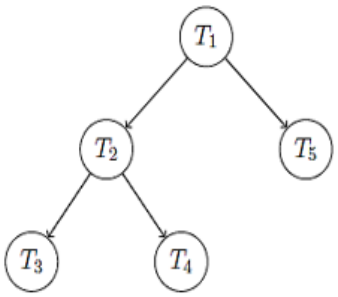


Figure 3.9 Les contraintes de précédences de l'exemple 3.5.

Pour résoudre ce problème on suit les étapes de l'algorithme précédent :

Initialisation

$n(1) = 2$; $n(2) = 2$; $n(3) = 0$; $n(4) = 0$; $n(5) = 0$.

$S = \{T_1, T_2, T_3, T_4, T_5\}$; $\quad p = \sum_{j=1}^{5} p_j = 38$;

k = 5

$j = 3, 4, 5.$ avec $n(j) = 0$ et $\begin{cases} f_3(p) = p - r_3 = 37 \\ f_4(p) = p - r_4 = 32 \\ f_5(p) = p - r_5 = 35 \end{cases}$

On choisit la tâche T_4 avec $(n(4) = 0$ et $f_4(p)$ soit minimum).

$S := S \backslash \{T_4\} \rightarrow \pi(5) = 4 \rightarrow p := p - p_4 = 28 \rightarrow \underset{i=1,3,4,5}{a_{i4} = 0}$ et $a_{24} = 1$

Alors $n(2) = 1$.

k = 4

On choisit la tâche T_5 avec $(n(5) = 0$ et $f_5(p)$ soit minimum).

$S := \{T_1, T_2, T_3\} \rightarrow \pi(4) = 5 \rightarrow p := p - p_5 = 17 \rightarrow a_{15} = 1$

Alors $n(1) = 1$.

k = 3

On choisit la tâche T_3 avec $n(3) = 0$.

$S := \{T_1, T_2\} \rightarrow \pi(3) = 3 \rightarrow p := p - p_3 = 10 \rightarrow a_{23} = 1$

Alors $n(2) = 0$.

k = 2

On choisit la tâche T_2 avec $n(2) = 0$

$S := \{T_1\} \rightarrow \pi(2) = 2 \rightarrow p := p - p_2 = 5 \rightarrow a_{12} = 1$

Alors $n(1) = 0$.

k = 1

On choisit la tâche T_5 avec ($n(5) = 0$ et $f_5(p)$ soit minimum).

$S := \emptyset \rightarrow \pi(1) = 1 \rightarrow p := p - p_1 = 0 \rightarrow a_{1j} = 0 \ ; j = 1, .., 5.$

La séquence optimale est : $(T_1, T_2, T_3, T_5, T_4)$,

Avec $f_{\max} = \max\limits_{1 \leq i \leq 5} \{c_i - r_i\} = \max\{5; 5; 18; 27; 34\}$

D'où : $f_{\max} = 34$.

3.3 Problèmes difficiles

3.3.1 Minimisation du temps de fin de traitement

3.3.1.1 Problème $1/r_i, \text{deliverytimes}/C_{\max}$

Dans ce type de probléme la tâche T_i est disponible pour le traitement au moment r_i a besoin de temps de traitement p_i et enfin doit passer *du temps de livraison (delivery times)* q_i dans le système après son traitement.

Ce type de problème d'ordonnancement soit *NP- difficile au sens fort.*

Scharge a présenté une heuristique qui repose sur l'idée qu'une tâche de délai de livraison q_i maximal (**delivery times)** parmi celles de temps de traitement p_i le plus ancien est choisie [12].

Algorithm 3.6 $1/r_i, deliverytimes/C_{max}$

Debut

$t := \min_{T_i \in \tau}\{r_i\}; C_{\max} := t;$

Tant que $\tau \neq \phi$ *faire*

Debut

$\tau' := \{T_i \backslash T_i \in \tau, \text{et } r_i \leq t\};$

choisir $T_i \in \tau'$ *tel que* $p_i = \max_{T_k \in \tau'}\{p_k \backslash q_k = \max_{T_l \in \tau'}\{q_l\}\};$

ordonnancer T_i *à l'instant* t;

$\tau = \tau - \{T_i\};$

$C_{\max} = \max\{C_{\max}, t + p_i + q_i\};$

$t := \max\{t + p_i, \min_{T_l \in \tau}\{r_l\}\};$

Fin;

Fin.

Exemple 3.6 *considérer* 7 *tâches avec :*

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7
r_i	10	13	11	20	30	0	30
p_i	5	6	7	4	3	6	2
q_i	7	26	24	21	8	17	0

On a chaque tâche i définit par : $T_i(t_i, t_i + p_i, t_i + p_i + q_i)$

τ' *l'ensemble des tâches disponibles*

Itération 1

$t := 0$; $C := 0$; $\tau' := \{T_6\}$

Ordonnancer la tâche T_6 *avec* $T_6(0, 6, 23)$

Itération 2

$t := 10$; $\tau' := \{T_1\}$

ordonnancer la tâche T_1 *avec* $T_1(10, 15, 22)$

Itération 3

$t := 15$; $\tau' := \{T_2, T_3\}$

Choisir la tâche qu'ayant la grande durée de livraison q_i *donc*

Ordonnancer la tâche T_2 *avec* $T_2(15, 21, 47)$

Itération 4

$t := 21$; $\tau^{'} := \{T_3, T_4\}$

Ordonnancer la tâche T_3 *avec* $T_3(21, 28, 52)$

Itération 5

$t := 28$; $\tau^{'} := \{T_4\}$

Ordonnancer la tâche T_4 *avec* $T_4(28, 32, 53)$

Itération 6

$t := 32$; $\tau^{'} := \{T_5, T_7\}$

Ordonnancer la tâche T_5 *avec* $T_5(32, 35, 43)$

Reste T_7 *avec* $T_7(35, 37, 37)$

Itération 7

$\tau = \phi$ *stop....*$C_{\max} := \max\{t_i + p_i + q_i\} = 53$

Donc on obtenu l'ordonnancement $(T_6, T_1, T_2, T_3, T_4, T_5, T_7)$ de longueur 53, qui est montré dans la figure 3.10 .

L'exécution sur une seule machine est représentée par des lignes continues et les délais de livraison par des lignes en pointillés.

L'ordonnancement optimale, serait $(T_6, T_3, T_2, T_4, T_1, T_5, T_7)$, et sa longueur totale est 50.

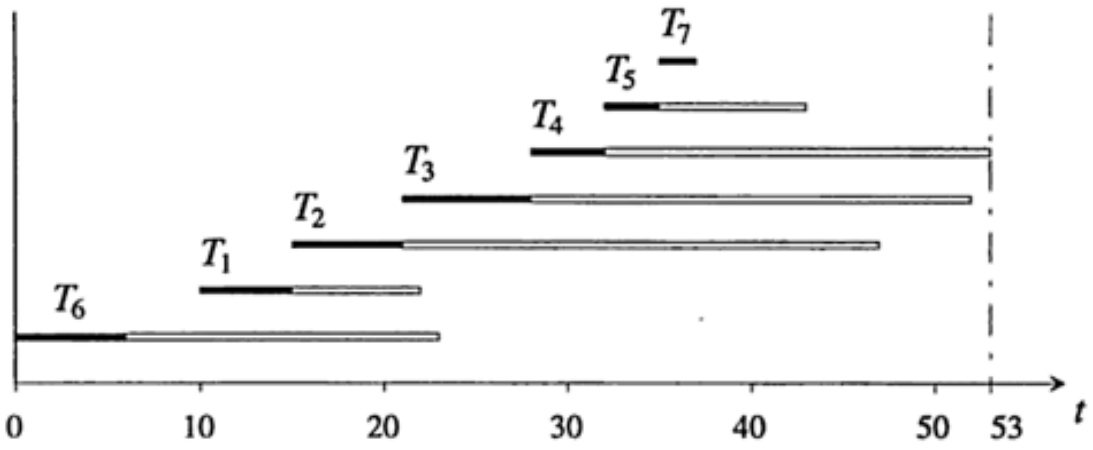


Figure 3.10 Un ordonnancement généré par l'algorithme de scharge del'exemple 3.6.

3.3.1.2 Problème $1/r_i,\tilde{d}_i/C_{\max}$

En cas de ce problème si les tâches sont autorisées à avoir des temps de traitement inégaux, une transformation du problème en 3-Partitions montre que ce problème est *NP-difficile au sens fort*, même pour les temps de disponibilité entiers et les dates limites.

Théorème 3.5 *Le problème* $1/r_i, \tilde{d}_i/C_{\max}$ *est* NP-difficile au sens fort *[35].*

Par réduction du problème des 3-Partition qui est fortement NP-Complet.

l'énonce de 3-Partition problème

Instance : $A = \{a_i\}_{1 \leq i \leq 3m}$ de $3m$ ensemble tel que $\frac{B}{4} \leq a_i \leq \frac{B}{2}$
et $\sum_A a_i = mB$, avec la borne $B \in \mathbb{N}$,

Question : Peut-on A en m ensemble disjoint $A_1, A_2, ..., A_m$
tel que: $\sum_{A_k} a_i = B$, $\forall\ k \in \{1, ..., m\}$ (chaque ensemble A doit contenir exactement 3 élément de A)

Réponse : "Oui" si A peut être partitionné en m ensemble disjoint $A_1, A_2, ..., A_m$
tel que: $\sum_{A_k} a_i = B$ sinon "Non"

Si nous montrons qu'il existe un sous-ensemble A qui contient des entiers totalisent B, il doit alors contient trois entiers cela découle l'hypothése $\frac{B}{4} \leq a_i \leq \frac{B}{2}$ (essaye de résumer 4 entiers ou 2 entiers).

Preuve. Il est clair que le problème est dans NP, a partir de l'énonce de 3-Partition.

Nous construisons un exemple d'un problème d'ordonnancement composé de $4m$ tâches tel que $T_i = (p_i, r_i, \tilde{d}_i)$ comme suit :

- $\forall i \in \{1, ..., m\}$, $T_i = (1, (B+1)\ (i-1), (B+1)(i-1)+1)$ Ce sont des tâches supplémentaires utilisées pour séparer les sous-ensemble.
- $\forall i \in \{m+1, ..., 4m\}$, $T_i = (a_j, 0, \infty)$; $j = i - m$ chacun de ces tâches T_i correspond à l'élément a_j de problème de 3-Partition.

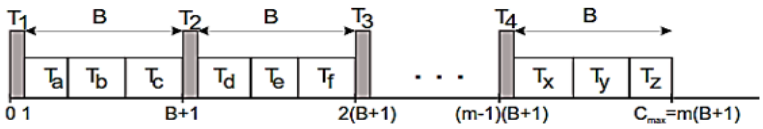


Figure 3.11 Partition réduite au problème $1/r_i, d_i/C_{max}$.

Il est facile de prouver que le problème 3-Partition a une solution *si et seulement si* la solution optimale a la valeur de $C_{\max} = m.(B+1)$.

Le problème général peut être résolu en appliquant *un algorithme de Branch & Bound*. Bratley et al ont proposé un algorithme qui sera brièvement décrit dans la figure suivante :

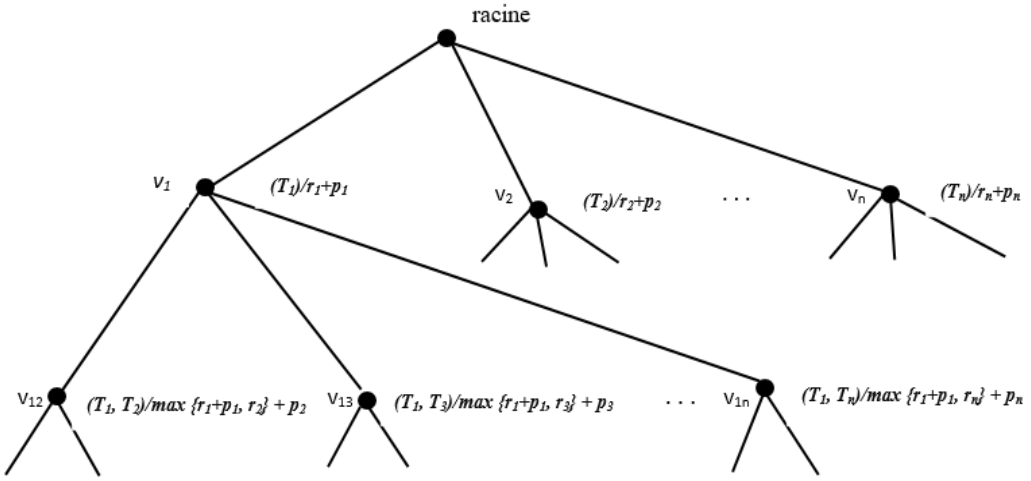


Figure 3.12 Arbre de recherche par Branch *et* Bound de Bratley et al.

Il s'agit d'une méthode énumérative qui crée un arbre de solution. Chaque nœud est étiqueté par : (l'ordre des tâches) /(heure de fin de la dernière tâche).

L'ordre dans lequel les nœuds de l'arbre examiné est basé sur une stratégie de recherche de retours en arrière, pourtant, l'algorithme utilise deux critères pour réduire le nombre d'étapes de recherche.

i. **Dépassement des dates limites** : Considérons le noeud v au niveau $k-1$ et ses $n-k+1$ successeurs immédiats au niveau k de l'arbre. Si le temps d'exécution associé à au moins l'un de ces nœuds dépasse la date limite de la tâche ajoutée au niveau k, puis tous les $n-k+1$ nœuds peuvent être exclus de toute considération ultérieure.

ii. **Problème de décomposition** : Considérons le niveau k de l'arbre de recherche, si le temps d'exécution C_i de T_i est inférieur ou égal au temps de disponibilité le plus petit $r_{\min}$ parmi les tâches non encore ordonnancée, puis, le problème se décompose au niveau k et il n'est pas nécessaire de saisir une autre branche de l'arbre de recherche, la raison de cette forte exclusion est que la meilleure ordonnancement pour les tâches $n-k$ restantes ne peut pas être démarré avant la plus petite temps de disponibilité parmi ces tâches, et donc pas plus tôt que le temps d'exécution C_i des k premières tâches [12].

3.3.1.3 Problème $1/r_i/\sum C_i$

Théorème 3.6 *Le problème $1/r_i/\sum C_i$ est de complexité NP- difficile au sens fort [8].*

Preuve. La preuve est faite en réduisant 3-Partition à $1/r_i/\sum C_i$.

Étant donné une instance de 3-Partition, nous construisons une instance de $1/r_i/\sum C_i$:

- Pour chaque a_i, nous créons une tâche Ti avec $p_i = a_i$ et $r_i = 0$.

-Pour chaque $n = 3t + k$ avec $k \in \{1, 2, ..., t\}$, nous créons un ensemble E_k de L tâches (où $L \gg B$), chacune avec temps de traitement 1 et date de disponibilité $kb + (k-1)L$, i.e. $E_k = \{T_{k,1}, T_{k,2}, ..., T_{k,L}\}$ avec $r_{k,i} = kb + (k-1)L$ et $p_{k,i} = 1$.

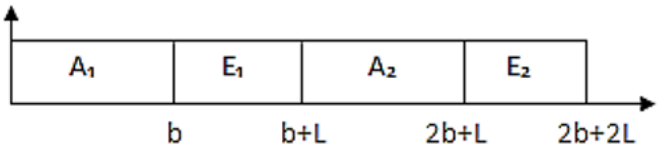


Figure 3.13 Partition réduite au problème $1/r_i/\sum C_i$.

Notez que chaque ensemble E_k se termine à l'heure $kb + kL$ et que dans chaque groupe A_i, les 3 tâches sont ordonnancés dans l'ordre croissant de leurs temps de traitement.

A_{nf} est l'ensemble des tâches non fictives (ceux avec temps de traitement p_k) et soit A_f l'ensemble des tâches fictives (celles avec des temps de traitement $p_{k,i}$). On a :

$$\sum_{i \in A_{nf}} C_i \approx \frac{3(b+L)n^2}{2}, \quad \sum_{i \in A_f} C_i \approx \frac{L^2 n^2}{2}$$

Puisque $(\sum_{i \in A_f} C_i) \gg (\sum_{i \in A_{nf}} C_i)$, nous avons mieux de faire les tâches fictives à mesure qu'elles sont **libérées traitées**. Plus loin, les tâches non fictives peuvent être placées exactement dans les espaces entre les tâches fictives et les déplacées plus en arrière ne ferait qu'augmenter la somme des temps de fin de traitement. Ainsi, l'ordonnancement ci-dessus est optimal.

Maintenant, étant donné l'absence de instance de 3-Partition, il existe un espace dans un groupe A_i qui ne peut pas être rempli par une tâche particulier non fictive leur temps de traitement p_i. Nous avons deux options:

-Reculez p_i sans affecter le reste de l'ordonnancement. Dans ce cas, nous augmentons le temps de traitement p_i et ainsi nous augmentons la somme totale des temps de fin de traitement.

-Poussez les tâches fictives en arrière, ce qui augmente également la somme totale des temps de fin de traitement.

Dans les deux cas, nous n'avons pas un ordonnancement optimal.

3.3.2 Minimisation de la tardivité

3.3.2.1 Problème $1/r_i/L_{\max}$

Ce problème avec des tâches ayant des dates de disponibilité arbitraire et qui ne permet pas la préemption, est notamment plus difficile. L'ordonnancement optimal n'est pas nécessairement un ordonnancement sans retard.

Théorème 3.7 *Le problème* $1/r_i/L_{\max}$ *est de complexité* NP- difficile au sens fort *[23].*

Preuve. La preuve est basé sur le fait que 3-Partition est réduit à $1/r_i/L_{\max}$.

Soient $a_1, \ldots, a_{3t}, b$, tel que $b/4 < a_i < b/2$ et $\sum_{i=1}^{3t} a_i = tb$ l'instance suivante de $1/r_i/L_{\max}$ peut être construit. Le nombre des tâches n est égale à $4t-1$

Et

$r_i = ib + (i-1)$, $p_i = 1$, $d_i = ib + i$, $i = 1, \ldots t-1$.

$r_i = 0$, $p_i = a_i - t + 1$, $d_i = tb + (t-1)$, $i = t, \ldots 4t-1$.

Un ordonnancement avec $L_{\max} \leq 0$ existe si et seulement si chaque tâche $i = 1, ..., t-1$, peut être traité entre r_i et $d_i = r_i + p_i$.

Cela peut être fait si et seulement si les tâches restantes peuvent être partitionnées sur les t intervalles de longueur b, ce qui peut être fait si et seulement si 3-Partition admet une solution.

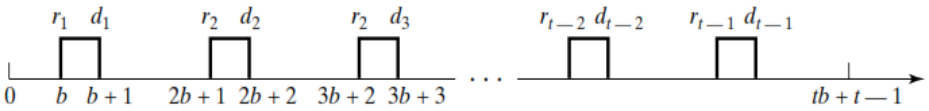


Figure 3.14 Partition réduite au problème $1/r_i/L_{max}$.

Plusieurs algorithmes exacts ont été proposés pour ce problème, mais ils sont tous basés sur des méthodes énumératives et leur temps de calcul augmente de manière exponentielle avec la taille du problème.

Une procédure de séparation et évaluation est proposé pour résoudre ce problème $1/r_i/L_{\max}$.

3.3.3 Minimisation du retard pondéré total

3.3.3.1 Problème $1//\sum w_iT_i$

Le problème $1//\sum w_iT_i$ est une généralisation importante du problème $1//\sum T_i$.

Des dizaines de chercheurs ont travaillés sur ce problème et ont expérimenté de nombreuses approches différentes.

Lemme 3.2 *S'il y a deux tâches T_i et T_k avec $d_i \leq d_k$, $p_i \leq p_k$ et $w_i \geq w_k$, il existe alors une séquence optimale dans laquelle la tâche T_i apparaît avant la tâche T_k.*

Malheureusement, aucun algorithme efficace ne peut être obtenu pour $1//\sum w_iT_i$ avec des poids arbitraires.

Théorème 3.8 *Le problème $1//\sum w_iT_i$ est de complexité* NP- difficile au sens fort *[23].*

Preuve. La preuve est faite en réduisant 3-Partition à $1//\sum w_iT_i$.

La réduction est basée sur la transformation suivante. Encore une fois, le nombre des tâches n est choisi égal à $4t-1$

Et

$$d_i = 0, \qquad p_i = a_j, \qquad w_i = a_j, \qquad i = 1, \ldots, 3t.$$
$$d_i = (j-3t)(b+t), \qquad p_i = 1, \qquad w_i = 2, \qquad i = 3t+1, \ldots, 4t-1.$$

Soit $$z = \sum_{i,k=1}^{3t} a_i a_k + \tfrac{1}{2}(t-1)tb.$$

On peut montrer qu'il existe un ordonnancement avec une valeur objective z si et seulement s'il existe une solution au problème de 3-Partition. Les 3 premières tâches ayant un rapport w_i/p_i égal à 1 et sont dus au moment 0. Il y a $t-1$ tâches avec w_i/p_i rapport égal à 2 et leurs dates d'échues sont à $b+1$, $2b+2$, et ainsi de suite.

Une solution avec la valeur z peut être obtenue si ces $t-1$ tâches peuvent être traités exactement pendant les intervalles $[b, b+1], [2b+1, 2b+2], ..., [(t-1)b+t-2, (t-1)b+t-1]$.

Pour adapter ces $t-1$ tâches à ces $t-1$ intervalles, le premier $3t$ tâches doit être partitionnées en t sous-ensembles de trois tâches chacune avec la somme des trois temps de traitement dans chaque sous-ensemble étant égal à b. Il peut être vérifié que dans ce cas la somme des retards pondérés est égale à z.

Si une telle partition n'est pas possible, il existe au moins un sous-ensemble dont la somme des trois temps de traitement est supérieure à b et un autre sous-ensemble de dont

la somme des trois temps de traitement est inférieure à b. On peut vérifier que dans ce cas la somme des retards pondérés est supérieure à z.

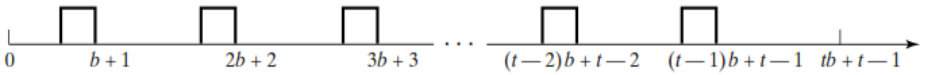


Figure 3.15 Partition réduite au problème $1//\sum w_i T_i$.

Généralement une méthode approchée est utilisée pour le problème $1//\sum w_i T_i$. Le plus souvent, les ordonnancements sont construits à partir de la fin. Au $i^{ième}$ niveau de l'arbre de recherche, les tâches sont placés dans la $(n-i+1)$ ème position. Donc de chaque nœud au niveau $i-1$ il y a $n-i+1$ branches allant au niveau i. il n'est pas nécessaire d'évaluer tous les nœuds possibles. Les résultats de dominance tels que celui décrit dans le lemme 3.2 peut éliminer un certain nombre des nœuds. La borne supérieure sur le nombre des nœuds au niveau j est $n!/(n-i)!$. L'argument pour construire la séquence en arrière est que les termes plus grands dans la fonction objective sont susceptibles de correspondre à des tâches qui se positionnent plus vers la fin de l'ordonnancement. Il semble avantageux de traiter ces derniers en premier.

Il existe de nombreuses techniques de délimitation différentes. L'un des plus élémentaires techniques de délimitation est basé sur une relaxation du problème à un problème de transport. Dans cette procédure, chaque tâche i avec un temps de traitement (entier) p_i est divisée en p_i tâches, chacun avec un temps de traitement unitaire. Les variables de décision x_{ik} sont 1 si une unité du tâche i est traitée pendant l'intervalle de temps $[k-1,k]$ et 0 sinon. Ces variables de décision x_{ik} doivent satisfaire deux ensembles de contraintes :

$$x_{ik} = \sum_{k=1}^{c_{\max}} p_i, \qquad i = 1, ..., n$$

$$x_{ik} = \sum_{j=1}^{n} p_i, \qquad k = 1, ..., c_{\max}$$

Clairement, une solution satisfait ces contraintes ne garantit pas un réalisable ordonnancement sans préemptions. Définir les coefficients de coût c_{ik} qui satisfont

$$\sum_{k=l-p_i+1}^{l} c_{ik} \leq w_i \max(c_i - d_i, 0). \quad \text{pour} \quad i = 1, ..., n; \quad l = 1, ..., C_{\max}.$$

Ensuite, la solution de coût minimum fournit une borne inférieure, puisque pour toute solution du problème de transport avec $x_{ik} = 1$ Pour $k = c_i - p_i + 1, \ldots, c_i$, la relation suivante :

$$\sum_{k=1}^{c_{\max}} c_{ik}\ x_{ik} = \sum_{k=c_j-p_j+1}^{c_j} c_{ik} \leq w_i\ \max(c_i - d_i, 0).$$

Il est assez facile de trouver des fonctions de coût qui répondent à cette relation.

Par exemple, l'ensemble

$$c_{ik} = \begin{cases} 0, & \text{pour} \quad k \leq d_i \\ w_i, & \text{pour } k > \ d_i \end{cases}$$

La solution du problème des transports prévoit une borne inférieure pour $1//\sum wiTi$.

Cette technique englobant est appliquée à l'ensemble des tâches non planifiés à chaque nœud de l'arbre. Si la borne inférieure est plus grande que la solution de l'ordonnancement connu, alors le nœud peut être éliminé [23].

Exemple 3.7 *Considérons les 4 tâches suivantes :*

Tâches	1	2	3	4
w_i	4	5	3	5
p_i	12	8	15	9
d_i	16	26	26	28

D'après le lemme 3.2, il s'ensuit immédiatement que, dans une séquence optimal la tâche T_4 suit la tâche T_2 et la tâche T_3 suit la tâche T_1. L'arbre de branch & bound est construit en arrière dans le temps. Seulement deux tâches doivent être considérées comme candidats pour la dernière position, à savoir les tâches T_3 et T_4. Les nœuds de l'arbre qui doivent être étudiés sont illustrés à la figure 3.18 . Pour sélectionner une branche à rechercher en premier, les bornes sont déterminées pour les deux nœuds au niveau 1.

Une borne inférieure pour une séquence optimale parmi les descendants du nœud $(*, *, *, T_4)$ peut être obtenue en considérant le problème de transport décrit avant d'appliquer aux tâches T_1, T_2 et T_3. Les fonctions de coût sont choisies comme suit :

$c_{1k} = 0, \qquad k = 1, \ldots, 16$
$c_{1k} = 4, \qquad k = 17, \ldots, 35$
$c_{2k} = 0, \qquad k = 1, \ldots, 26$
$c_{2k} = 5, \qquad k = 27, \ldots, 35$
$c_{3k} = 0, \qquad k = 1, \ldots, 26$
$c_{3k} = 3, \qquad k = 27, \ldots, 35$

L'affectation optimale des segments de tâche aux créneaux horaires affecte la tâche T_1 dans la première 12 emplacements, la tâche T_2 dans les emplacements 19 à 26 et tâche T_3 dans les emplacements 13 à 18 et 27 à 35 (cette solution optimale peut être trouvée en résolvant un problème de transport mais peut, bien sûr, également être trouvé par essais et erreurs). Le coût de cette affectation de ces trois tâches est 3×9 (coût de l'attribution de la tâche T_3 aux créneaux 27 à 35). Afin d'obtenir une borne inférieure pour le nœud, le retard de la tâche T_4 doit être ajouté; cela donne la borne inférieure $27 + 80$, ce qui équivaut à 107.



Figure 3.16 Solution de l'exemple 3.7 de première borne.

De manière similaire, une borne inférieure peut être obtenue pour le nœud $(*, *, *, T_3)$. Une borne inférieure pour un ordonnancement optimal pour les tâches T_1, T_2 et T_4, donne 8, tandis que le retard de la tâche T_3 est de 54, ce qui donne une limite de 62.

Comme le nœud $(*, *, *, T_3)$ semble être le nœud le plus prometteur, la progéniture de ce nœud est considéré en premier. Il s'avère que le meilleur ordonnancement accessible de ce nœud est (T_1, T_2, T_4, T_3) avec une valeur objective de 64.

D'après le fait que la limite inférieure de $(*, *, *, T_4)$ est 107, il s'ensuit que $(T_1, T_2, 4, T_3)$ est le meilleur programme global.

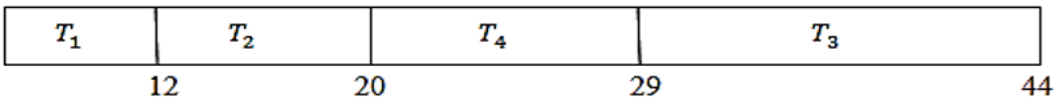


Figure 3.17 Solution de l'exemple 3.7 de deuxième borne.

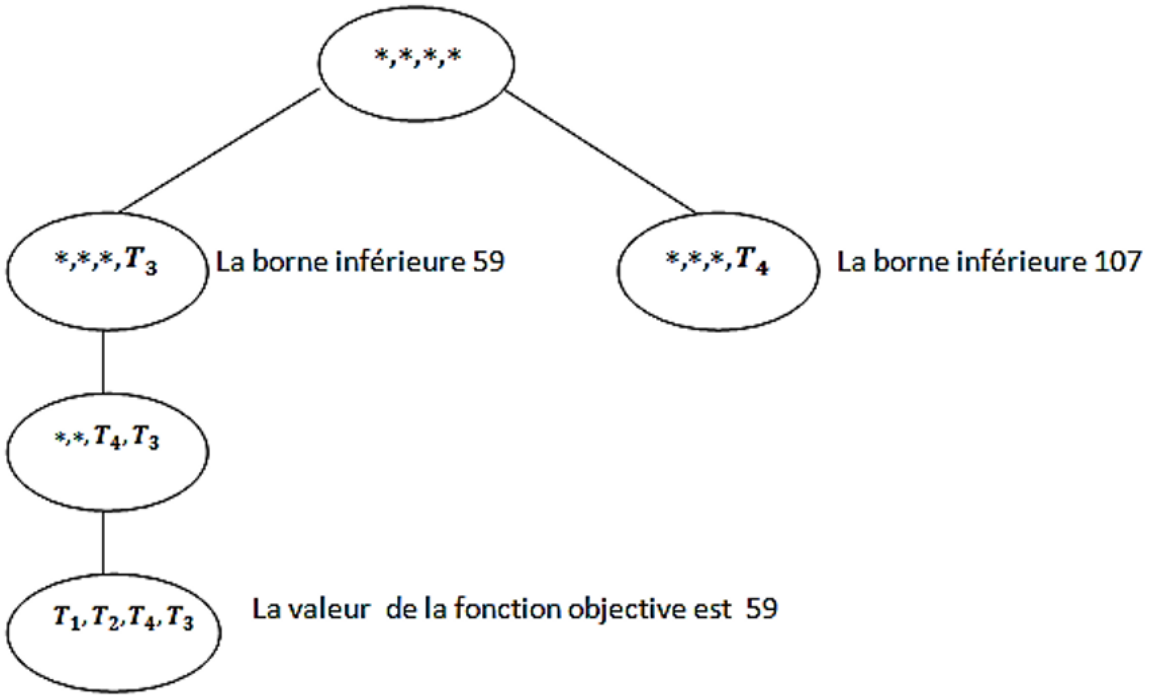


Figure 3.18 Procédure de Branch *and* Bound pour l'exemple 3.7.

3.4 Conclusion

Dans ce chapitre nous avons exposé quelques problèmes d'ordonnancement sur une seule machine qui sont distingués comme des problèmes *faciles et difficiles*, pour les problèmes faciles, nous avons présenté leurs complexité ainsi que quelques algorithmes de résolution. Pour les problèmes difficiles, nous avons présenté leurs complexité et nous avons utilisé le problème de 3-Partition pour les prouver.

Les résultats de ce chapitre sont résumés dans un tableau dans l'annexe.

Chapitre 4
Etude et Implémentation d'un problème pratique sur MATLAB

4.1 Introduction

Dans ce chapitre nous exposons un problème pratique à être traités par la recherche opérationnelle pécisement l'ordonnancement pour avoir la résolution satisfaite.

Pour atteindre cet objectif nous présentons l'implémentation de quelques heuristiques proposées et nous les illustrons par des exemples réels de notre problème, ensuite nous reportons leurs résultats numériques obtenus par l'éxecution de notre programme Matlab.

4.2 Présentation du problème

4.2.1 Description du problème

Une grande entreprise de metallurgie possède un site en bord de mer. L'approvisionnement en matière première (minerais, charbon, etc...) s'éffectue par voie maritime.

Dans notre étude nous allons traiter le problème de planifier des navire au post à quai dans un terminal à conteneurs .Une meilleure solution de ce problème se représente par un plan de service de (chargement/ déchargement) des navires entrants et des durées de séjour de longueurs minimales, et cherche aussi à planifer de tel sort le temps de manutention soit minimal.

4.2.2 Principe

Le principe consiste à pouvoire réaliser une bonne résultat d'optimisation des temps de manutention des navires, dans le but d'augmenter le rendement et en reduisant les pénalités demandées.

4.2.3 Formulation mathématique du problème

Les variables

Nous considérons les paramètres suivantes :

- M : Poste à quai (une seule site).
- $T_i : i \in \{1, .., n\}$:Ensemble des navires entrants.
- S : Le moment où le poste à quai devient libre, pour la planification des navires au poste à quai.
- r_i : Le temps d'arrivée du navire T_i (date de disponibilité).
- p_i : Le temps de traitement du navire T_i sur le poste à quai.
- $A_i = S - r_i$: Le temps d'attente de navire T_i.
- w_i : Facteur relatif au type de matière première qui est dans le navire T_i.
- k : Facteur relatif au conditions de travail.
- pe : Une pénalité demandé pour une durée d'attente d'un navire.

Les contraintes

- Le quai de déchargement du site ne peut acceuillir qu'un seul navire à la fois.
- Tous les navires n'arrivent pas en même temps sur le site (les navires ayant des dates de disponibilité arbitraires)
- Les autres navires doivent attendre en mer.
- Une durée d'attente d'un navire est facturé par une pénalité demandée .

Fonction objectif :

C'est le but visé par l'entreprise, qui est représenté par une ou plusieurs critères d'optimisation (temps d'attente, temps de manutention et la pénalité demandé).

L'objectif est de minimiser le temps moyen de manutention des navires qui égale à

$$TM = \min \sum_{i=1}^{n} (S - r_i) + p_i$$

4.2.4 Définition du problème

La représentation de ce problème dans notre domaine d'ordonnancement est :

$1/r_i, w_i/ \sum_{i=1}^{n} (S - r_i) + p_i$ qui est défini comme suit :

Planifier un ensemble des navires indépendants (les tâches) ayant des dates d'arrivés r_i (dates de disponibilité), des temps de traitement P_i et des facteurs Relatifs au type de matière première w_i arbitraires sur le seule post à quai qui devient libre au moment S.

L'objectif est de minimiser le temps moyen de manutention des navires.

Ce problème se réduit au problème $1/r_i/ \sum_{i=1}^{n} C_i$ qui est déja montrer NP-difficile en preuve de théorème 3.6.

Ce pourquoi, nous allons développer trois heuristiques pour sa résolution.

4.3 Les heuristiques proposés

Pour résoudre notre problème on propose les heuristiques suivantes qui sert à minimiser notre fonction objectif.

4.3.1 Première heuristique H1

Le principe de cette heuristique est de traiter les navires selon la règle FIFO de leurs temps d'arrivée.

La règle FIFO où la première tâche arrivée est la première à être ordonnancée

Algorithm 4.1 $1/r_i, w_i/\sum(S - r_i) + p_i$

Début

Ranger les tâches selon l'ordre croissant de leurs dates de disponibilités dans une liste L.

Tant que $L \neq \emptyset$ *faire*

Début

Si $r_i \geq S$ *alors*

Début

$A_i = 0;$

$c_i = r_i + p_i;$

$S := c_i + \omega_i/k;$

Fin

Sinon *Début*

$A_i = S - r_i;$

$c_i = S + p_i;$

$S := c_i + \omega_i/k;$

Fin

$L := L \backslash \{T_i\}$

Fin

$A = \sum_{i=1}^{n} A_i;$

$pn = pe.A;$

$TM = \sum_{i=1}^{n} (A_i + p_i);$

Fin.

Exemple d'application

On prend 5 navires $n = 5$ sur une seule poste à quai dont les temps de traitement p_i *par minute*, les temps d'arrivée r_i *par heure* et les facteurs ralatifs au type de matière première w_i sont donnés dans le tableau suivant avec $S = 6h$, $k = 3$ *et* $pe = 500$ *ua* (*unité d'argent*)

	T_1	T_2	T_3	T_4	T_5
r_i	12	6	15	8	13
p_i	20	15	30	45	35
w_i	2	1	4	3	5

On appliquant l'heuristique $H1$ à notre exemple on obtient la liste $L = (T_2, T_4, T_1, T_5, T_3)$ par le rangement des navires dans l'ordre croissant des temps d'arrivée r_i.

Itération1 : $r_2 = 6 \geq S = 6 \rightarrow A_2 = 0 \rightarrow C_2 = 6.25 \rightarrow S = 6.58, L = L/\{T_2\}$.
Itération2 : $r_4 = 8 \geq S = 6.58 \rightarrow A_4 = 0 \rightarrow C_4 = 8.75 \rightarrow S = 9.75, L = L/\{T_4\}$.
Itération3 : $r_1 = 12 \geq S = 9.75 \rightarrow A_1 = 0 \rightarrow C_1 = 12.33 \rightarrow S = 13, L = L/\{T_1\}$.
Itération4 : $r_5 = 13 \geq S = 13 \rightarrow A_5 = 0 \rightarrow C_5 = 13.58 \rightarrow S = 15.25, L = L/\{T_5\}$.
Itération5 : $r_3 = 15 \leq S = 15.25 \rightarrow A_3 = 0.25 \rightarrow C_3 = 15.75 \rightarrow S = 17.08$,
$L = \emptyset ... Stop$.

On obtient les rèsultat $\begin{cases} At = 0.25h \\ pnt = At^{*}500 = 125ua \\ TM = 2.66h \end{cases}$

La sèquence obtenue $(T_2, T_4, T_1, T_5, T_3)$ est représentée sur la figure suivante :

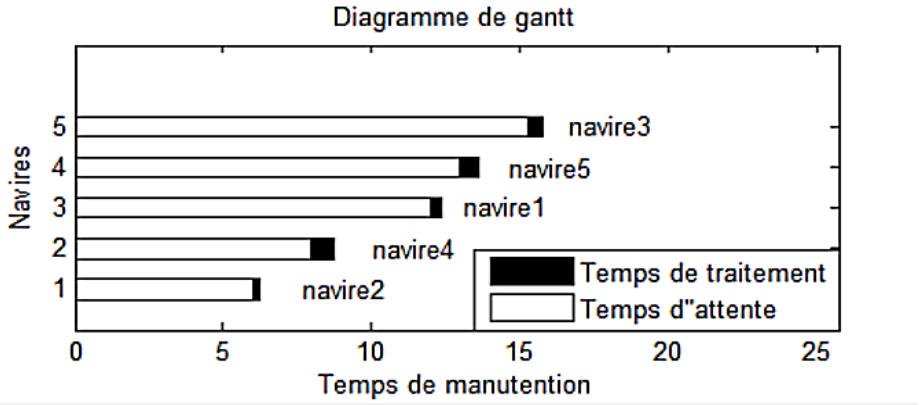


Figure 4.1 Solution de l'exemple par l'heuristique $H1$.

4.3.2 Deuxième heuristique H2

Le principe de cette heuristique est de traiter à chaque fois le navire ayant le plus petit temps de manutention.

Algorithm 4.2 $1/r_i, w_i/\sum(S - r_i) + p_i$

Début

Tant que $N \neq \emptyset$ *faire*

$A_i = S - r_i$;

Si $A_i > 0$ *alors* $c_i = S + p_i$;

$TM = A_i + p_i$;

Sinon $A_i = 0$;

$c_i = r_i + p_i$;

$TM = A_i + p_i$;

Choisir le navire T_i *ayant le plus petit temps de manutention* TM=min TM_i

$S := c_i + \omega_i/k$;

$N := N \backslash \{T_i\}$;

Si $N = \emptyset$ *alors* *Début*

$A = \sum_{i=1}^{n} A_i$;

$pn = pe.A$;

$TM = \sum_{i=1}^{n}(A_i + p_i)$;

Fin

Fin.

Exemple d'application

Reprenons l'exemple précédent et on appliquant cette heuristique :

Itération1 :

$A_1 = -6 \rightarrow A_1 = 0 \rightarrow C_1 = 12.33 \rightarrow TM = 0.33.$

$A_2 = 0 \rightarrow C_2 = 6.25 \rightarrow TM = 0.25.$

$A_3 = -9 \rightarrow A_3 = 0 \rightarrow C_3 = 15.5 \rightarrow TM = 0.5.$

$A_4 = -2 \rightarrow A_4 = 0 \rightarrow C_4 = 18.75 \rightarrow TM = 0.75.$

$A_5 = -7 \rightarrow A_5 = 0 \rightarrow C_5 = 13.58 \rightarrow TM = 0.58.$

$\min TM = 0.25$ *le navire* $T_2 \rightarrow S = 6.58, N = N/\{T_2\}.$

Itération2 :

$A_1 = -5.41 \rightarrow A_1 = 0 \rightarrow C_1 = 12.33 \rightarrow TM = 0.33.$

$A_3 = -8.42 \rightarrow A_3 = 0 \rightarrow C_3 = 15.5 \rightarrow TM = 0.5.$

$A_4 = -1.42 \rightarrow A_4 = 0 \rightarrow C_4 = 18.75 \rightarrow TM = 0.75.$

$A_5 = -6.42 \rightarrow A_5 = 0 \rightarrow C_5 = 13.58 \rightarrow TM = 0.58.$

$\min TM = 0.33$ *le navire* $T_1 \rightarrow S = 13, N = N/\{T_1\}.$

Itération3 :

$A_3 = -2 \rightarrow A_3 = 0 \rightarrow C_3 = 15.5 \rightarrow TM = 0.5.$

$A_4 = 5 \rightarrow C_4 = 13.75 \rightarrow TM = 5.75.$

$A_5 = 0 \rightarrow C_5 = 13.58 \rightarrow TM = 0.58.$

$\min TM = 0.5$ *le navire* $T_3 \rightarrow S = 16.83, N = N/\{T_3\}.$

Itération4 :

$A_4 = 8.83 \rightarrow C_4 = 17.58 \rightarrow TM = 9.58.$

$A_5 = 3.83 \rightarrow C_5 = 17.41 \rightarrow TM = 4.41.$

$\min TM = 4.41$ *le navire* $T_5 \rightarrow S = 19.08, N = N/\{T_5\}.$

Itération5 :

$A_4 = 11.83 \rightarrow C_4 = 19.83 \rightarrow TM = 11.83.$

$\min TM = 11.83$ *le navire* $T_4 \rightarrow S = 20.83, N = \emptyset ...Stop.$

On obtient les rèsultat : $\begin{cases} At = 14.91h \\ pnt = At^*500 = 7458.3ua \\ TM = 17.33h \end{cases}$

La séquence obtenue $(T_2, T_1, T_3, T_5, T_4)$ est représentée sur la figures suivante :

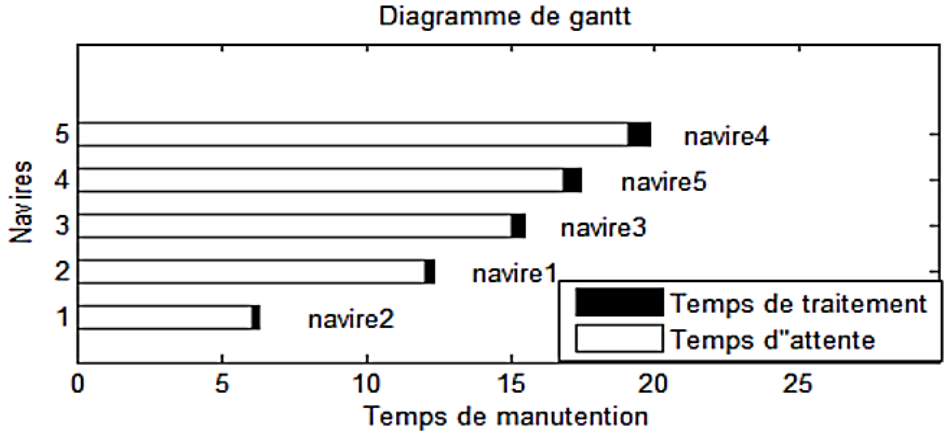


Figure 4.2 Solution de l'exemple par l'heuristique $H2$.

4.3.3 Troisième heuristique H3

Le principe de cette heuristique est de traiter à chaque fois le navire ayant le plus petit temps d'attente.

Algorithm 4.3 $1/r_i, w_i/\sum(S - r_i) + p_i$

Début

Tant que $N \neq \emptyset$ *faire*

$A_i = S - r_i$;

si $A_i < 0$ *alors* $A_i = 0$;

Choisir le navire T_i *ayant le plus petit temps d'attente* A_i.

si $A_i > 0$ *alors* $c_i = S + p_i$;

$S := c_i + \omega_i/k$;

sinon $c_i = r_i + p_i$;

$S := c_i + \omega_i/k$;

$N := N \backslash \{T_i\}$

Si $N = \emptyset$ *alors* *Début*

$A = \sum_{i=1}^{n} A_i$;

$pn = pe.A$;

$TM = \sum_{i=1}^{n} (A_i + p_i)$;

Fin

Fin.

Exemple d'application

On prend le même exemple et on appliquant cette heuristique

Itération1 :

$A_1 = -6$, $A_2 = 0$, $A_3 = -9$, $A_4 = -2$, $A_5 = -7$.

$A_1 = 0$, $A_2 = 0$, $A_3 = 0$, $A_4 = 0$, $A_5 = 0$.

$\min A_i = 0$ *le navire* $T_1 \rightarrow C_1 = 12.33 \rightarrow S = 13$, $N = N/\{T_1\}$.

Itération2 :

$A_2 = 7$, $A_3 = -2 = 0$, $A_4 = 5$, $A_5 = 0$.

$\min A_i = 0$ *le navire* $T_3 \rightarrow C_3 = 15.5 \rightarrow S = 16.83$, $N = N/\{T_3\}$.

Itération3 :

$A_2 = 10.83,\ A_4 = 8.83,\ A_5 = 3.83.$

$\min A_i = 3.83$ *le navire* $T_5 \rightarrow C_5 = 17.41 \rightarrow S = 19.08,\ N = N/\{T_5\}.$

Itération4 :

$A_2 = 13.08,\ A_4 = 11.08.$

$\min A_i = 11.08$ *le navire* $T_4 \rightarrow C_4 = 19.83 \rightarrow S = 20.83,\ N = N/\{T_4\}.$

Itération5 :

$A_2 = 14.83$,$C_4 = 21.08 \rightarrow S = 21.41,\ N = \emptyset ...Stop.$

On obtient les résultats suivantes : $\begin{cases} At = 29.75h \\ pnt = At^{*}500 = 14875ua \\ TM = 32.16h \end{cases}$

La séquence obtenue $(T_1, T_3, T_5, T_4, T_2)$ est représentée sur la figures suivante :

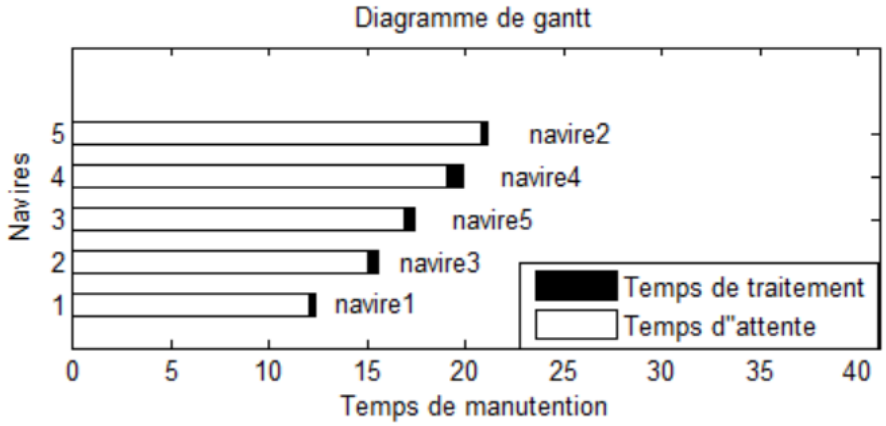


Figure 4.3 Solution de l'exemple par l'heuristique $H3$.

4.4 Résolution du problème et implémentation

Nous aboutissons maintenant à l'étape finale à savoir l'élaboration d'un langage muni d'une interface claire et accessible, facilitant son utilisation.

4.4.1 Choix du langage

Le choix s'est porté sur l'emploi du langage du logiciel Matlab, car il répond aux critères suivants :

- La maniabilité du langage : constitué d'un ensemble de possibilités faisant en sorte que le programmeur travaille avec aisance, assuré d'une part par la syntaxe du langage et d'autre part par un aspect visuel clair représentatif à la fois du détail et du global.
- Le bagage du langage : il contient une interface graphique puissante ainsi qu'une grande variété de méthodes scientifiques implémentées (prédéfinies).

4.4.2 Généralités sur le langage Matlab

Matlab est un logiciel parfaitement dédié à la résolution de problèmes d'analyse numérique ou de traitement du signal. Permet d'effectuer des calculs matriciels ou de visualiser les résultats sous forme graphique. La formulation des problèmes s'apparente à la formulation mathématique des problèmes à résoudre. L'utilisation de ce logiciel consiste à lancer des lignes de commandes, qui peuvent le plus souvent ressembler à la programmation en C.

Le nom Matlab vient de MATrix Laboratory, les éléments de données de base manipulés par Matlab étant des matrices (mais pouvant évidemment se réduire à des vecteurs et des scalaires) qui ne nécessitent ni dimensionnement ni déclaration de type. Contrairement aux langages de programmation classiques, les fonctions du Matlab permettent de manipuler directement et interactivement ces données matricielles, le rendant ainsi particulièrement efficace en calcul numérique, analyse et visualisation de données en particulier. Il existe deux modes de fonctionnement sur Matlab :

(a) Le mode interactif : Les instructions sont exécutées au fur et à mesure qu'elles sont données par l'usager.

(b) Le mode exécutif : Dans ce cas, l'utilisateur utilise un fichier "M-file" contenant toutes les instructions à exécuter.

4.4.3 Exemple illustratif

Pour illustrer notre heuristiques on propose trois exemples avec les données suivantes :

- Le nombre des navires entrants $n = 10, 30, 50$.
- Le moment où le poste à quai devient libre est $S = 6h$.
- Le facteur relatif au conditions de travail $k = 3, 5$.
- La pénalité demandée pour une durée d'attente d'un navire $pe = 500$ ua.
- Les paramètres des navires sont présentés dans le tableau 4.1 suivant tel que :
  - Le temps d'arrivée du navire r_i par *heure.*
  - Le temps de traitement du navire p_i *par minutes.*
  - Le facteur relatif au type de matière première qui est dans le navire w_i.
- Pour chaque exemple, on traite les n premières navires du tableau.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}
r_i	12	6	15	8	13	11	14	10	7	9	13	9	7	11	14	7	10
p_i	20	15	30	45	35	50	30	40	25	35	50	30	40	25	35	20	15
w_i	2	1	4	3	5	2	4	1	5	2	3	5	2	4	5	2	1

	T_{18}	T_{19}	T_{20}	T_{21}	T_{22}	T_{23}	T_{24}	T_{25}	T_{26}	T_{27}	T_{28}	T_{29}	T_{30}	T_{31}	T_{32}	T_{33}
r_i	9	8	14	11	13	15	16	6	8	12	9	11	10	12	6	15
p_i	30	45	30	20	25	30	35	40	50	45	55	30	60	20	15	30
ω_i	4	3	5	2	4	1	5	2	3	5	2	4	1	2	1	4

	T_{34}	T_{35}	T_{36}	T_{37}	T_{38}	T_{39}	T_{40}	T_{41}	T_{42}	T_{43}	T_{44}	T_{45}	T_{46}	T_{47}	T_{48}	T_{49}	T_{50}
r_i	8	6	11	14	10	7	9	13	9	7	11	14	7	10	9	8	14
p_i	45	35	50	30	40	25	35	50	30	40	25	35	20	15	30	45	30
ω_i	3	5	2	4	1	5	2	3	5	2	4	5	2	1	4	3	5

Tableau 4.1 Les données des exemples.

4.4.4 Résultats numériques

Dans cette section, nous avons utilisés notre programme MATLAB sur notre exemples, tel que on a une seule poste à quai (machine) et nous fixons le nombre des navires (tâches) à $n = 10, 30, 50$ et le facteur relatif au conditions de travail $k = 3, 5$.

Les paramètres des navires qui sont les temps d'arrivée r_i, les temps de traitement p_i et le facteur relatif au type de matière première qui est dans le navire ω_i seront distribués aléatoirement.

Pour chaque nombre de tâche n nous avons fait varier le facteur k, ainsi comparé les résultats des critères obtenus pour chaque heuristiques pour laissez-nous conclure quelle heuristique donne une bonne solution.

Donc nous adressons les résultats obtenus du programme pour l'exemple $n = 10$ en deux cas $k = 3$ *et* 5.

Cas 1 $n = 10$, $k = 3$

- *Heuristique* 1

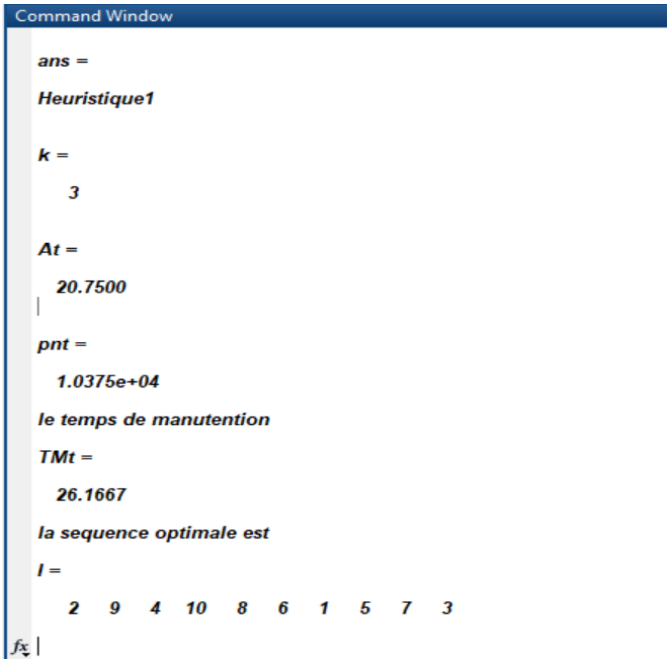


Figure 4.4 Résultats de l'heuristique $H1$ pour le 1^{er} Cas.

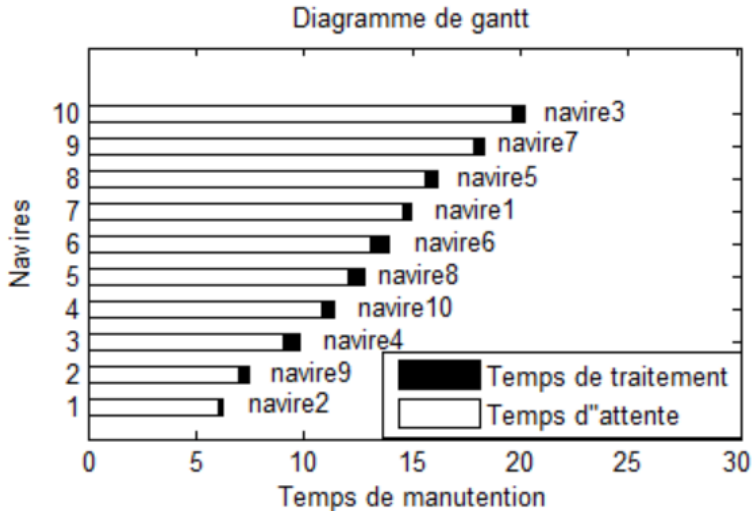


Figure 4.5 Diagramme de gantt de l'heuristique $H1$ pour le 1^{er} Cas.

- *Heuristique* 2

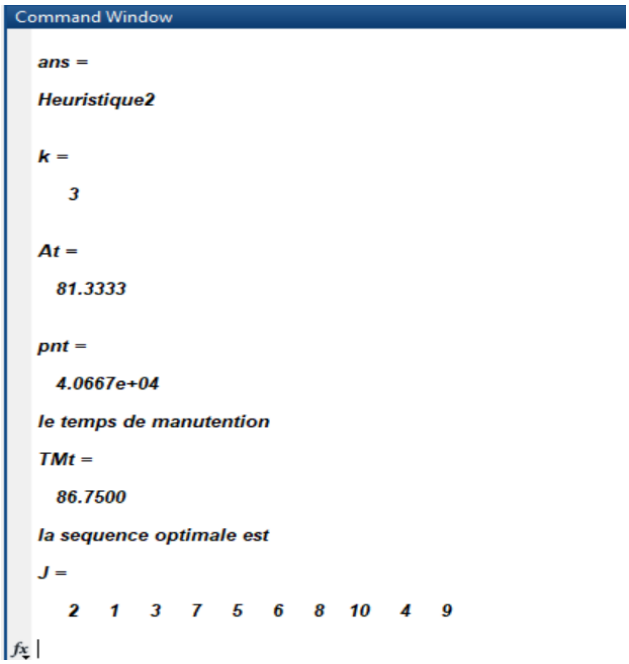


Figure 4.6 Résultats de l'heuristique $H2$ pour le 1^{er} Cas.

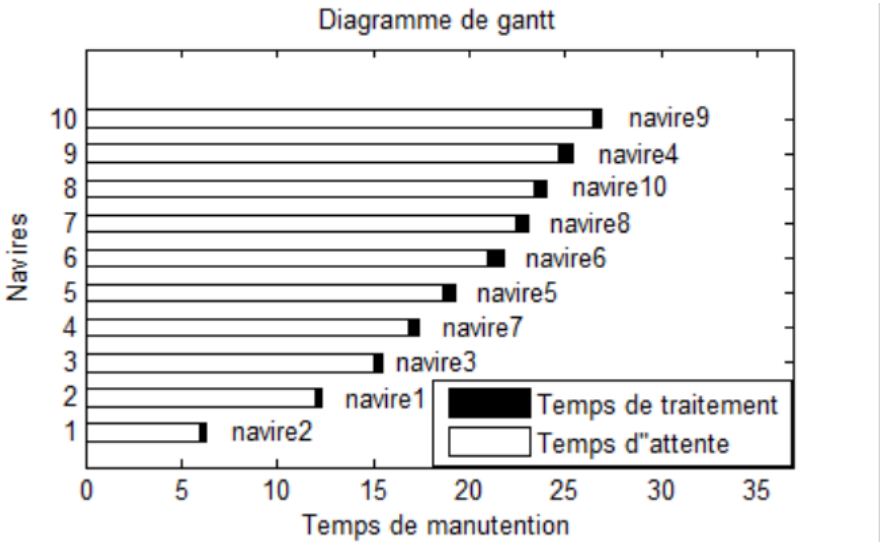


Figure 4.7 Diagramme de gantt de l'heuristique $H2$ pour le 1^{er} Cas.

- *Heuristique* 3

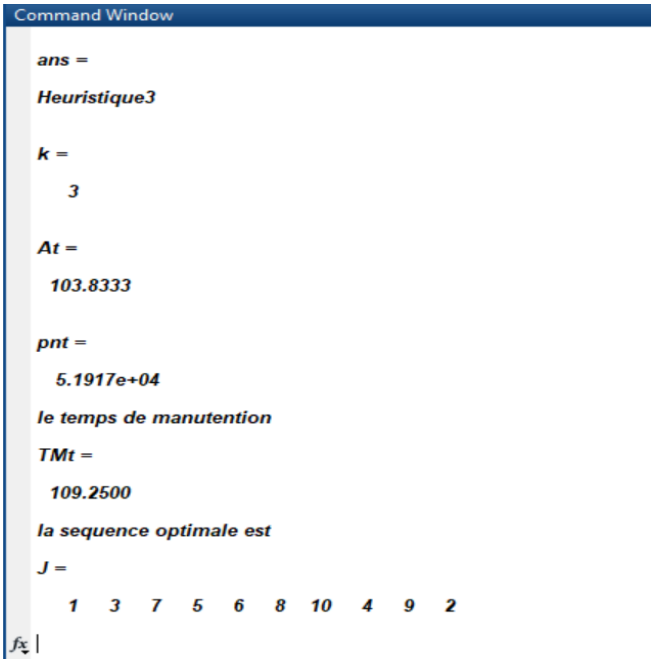


Figure 4.8 Résultats de l'heuristique $H3$ pour le 1^{er} Cas.

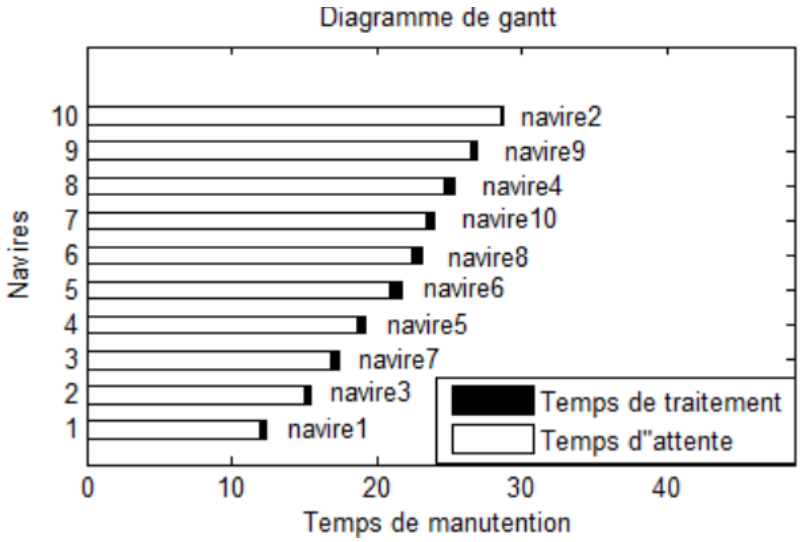


Figure 4.9 Diagramme de gantt de l'heuristique $H3$ pour le1^{er} Cas.

Cas 2 $n = 10$, $k = 5$

- *Heuristique* 1

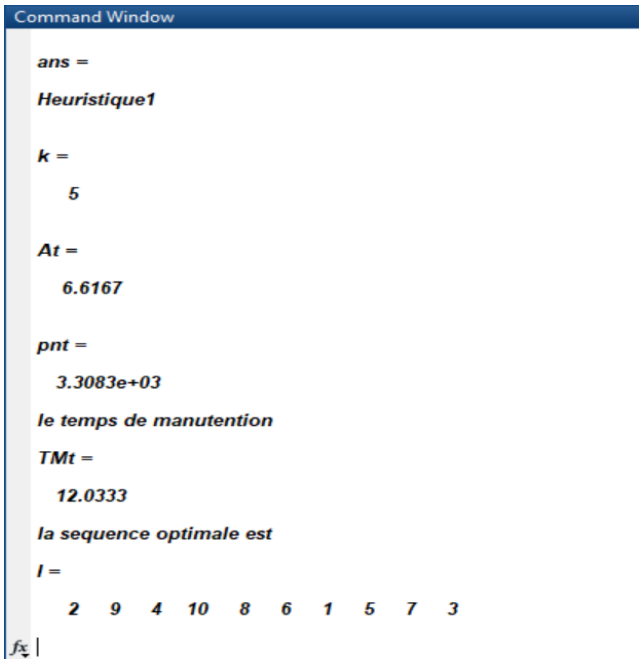


Figure 4.10 Résultats de l'heuristique $H1$ pour le 2^{eme} Cas.

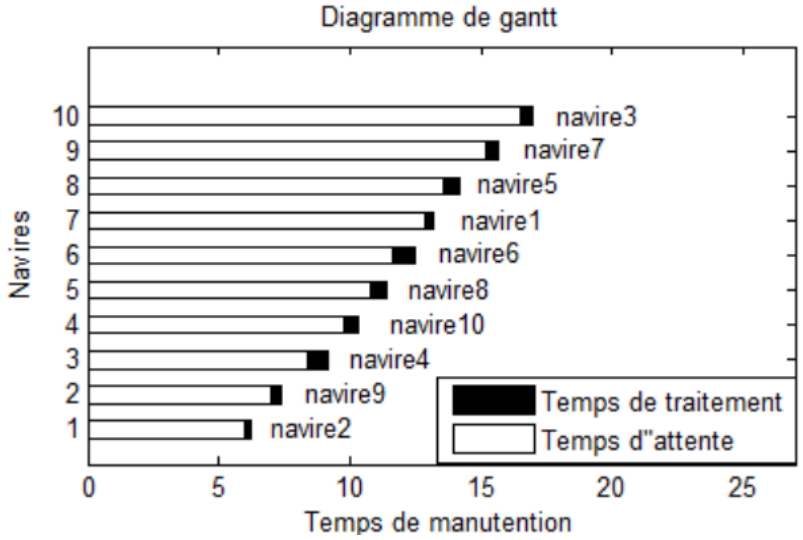


Figure 4.11 Diagramme de gantt de l'heuristique $H1$ pour le 2^{eme} Cas.

- *Heuristique* 2

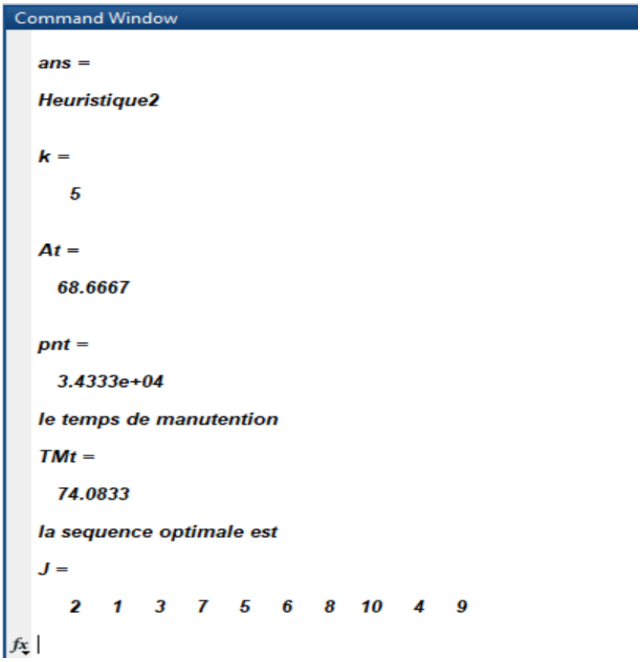


Figure 4.12 Résultats de l'heuristique $H2$ pour le 2^{eme} Cas.

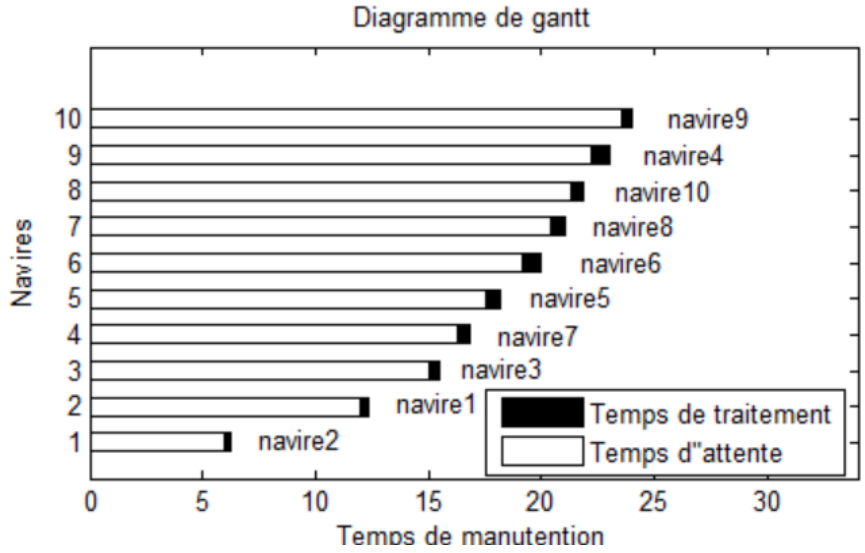


Figure 4.13 Diagramme de gantt de l'heuristique $H2$ pour le 2^{eme} Cas.

- *Heuristique* 3

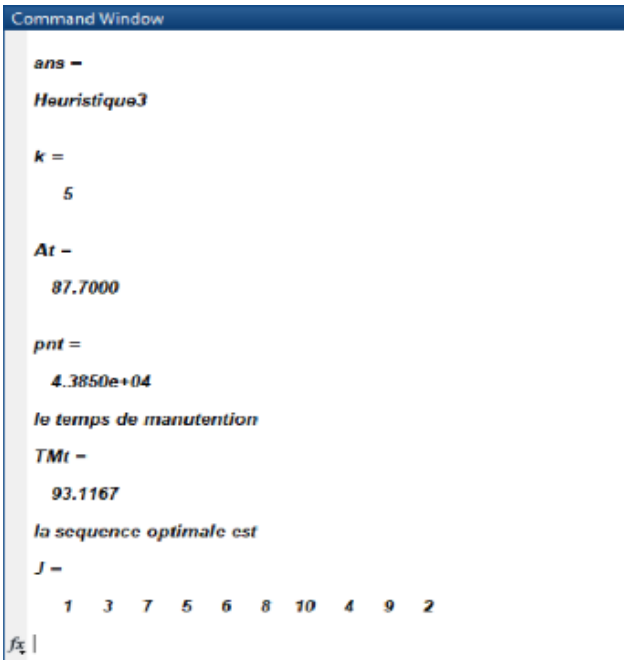


Figure 4.14 Résultats de l'heuristique $H3$ pour le 2^{eme} Cas.

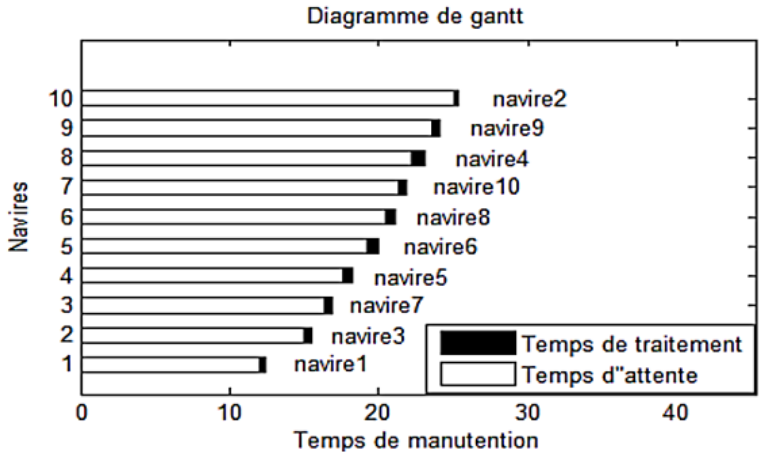


Figure 4.15 Diagramme de gantt de l'heuristique $H3$ pour le 2^{eme} Cas.

Nous complétons les autres exemples pour $n = 30$ et 50 avec $k = 3$ et 5, les résultats sont présentées dans le tableau suivant :

		Critères d'optimalités	Heuristique 1	Heuristique 2	Heuristique 3
$k = 3$	$n = 10$	Temps d'attente At	20.75	81.33	103.83
		Temps de manutention TM	26.1667	86.75	109.25
		Pénalité totale Pnt	10375	40667	51917
	$n = 30$	Temps d'attente At	533.833	683.833	822.5
		Temps de manutention TM	551.083	701.083	839.75
		Pénalité totale Pnt	266920	341920	411250
	$n = 50$	Temps d'attente At	1655.8	1978.8	2208.6
		Temps de manutention TM	1683.8	2006.8	2236.6
		Pénalité totale Pnt	827920	989420	1104300
$k = 5$	$n = 10$	Temps d'attente At	6.6167	68.66	87.7
		Temps de manutention TM	12.03	74.08	93.11
		Pénalité totale Pnt	3308.3	34333	43850
	$n = 30$	Temps d'attente At	368.76	535.96	641.96
		Temps de manutention TM	386.01	553.21	659.21
		Pénalité totale Pnt	184380	267980	320980
	$n = 50$	Temps d'attente At	1182.9	1513.2	1687.4
		Temps de manutention TM	1210.9	1541.2	1715.4
		Pénalité totale Pnt	591450	756620	843690

Tableau 4.2 Les résultats des exemples.

4.4.5 Discussion

A la lecture du tableau 4.2, il apparaît clairement que l'heuristique $H1$ est plus intréssant que les heuristiques $H2$ et $H3$. Nous remarquons que les critères d'optimalités (temps d'attentes, temps de manutention et la pénalité) dépendent de nombre des navires entrants, facteur relatif au type de matière première et au conditions des travails.

L'application des trois heuristiques proposées nous montrent que à chaque fois on augmente le nombre des navires nous avons une augmentation dans la valeur de la fonction objectif, d'autre part une variation de facteur relatif au conditions de travail k a beaucoup d'influence sur la solution de tel sorte que chaque fois la valeur k est grande le temps de manutention diminue. Par contre, une variation de facteur relatif au type de matière première w_i augmente le temps de manutention.

On peut ainsi conclure sur l'importance de considérer le facteur relatif au conditions des travails k.

L'heuristique $H1$ génère des meilleures solutions que les heuristiques $H2$ et $H3$ pour notre exemples. Par ailleurs, pour évaluer les trois heuristiques proposées nous avons utilisé l'indicateur de performance de chaque heuristique par rapport à la valeur de pénalité. Ce dernier est donné par le quotient $\frac{pnt^{>}-pnt^{<}}{pnt^{>}}$ avec $pnt^{>}$ et $pnt^{<}$ la grande et petite valeure du pénalité respectivement donnée par les deux heuristiques évaluées.

L'heuristique $H1$ qui donne des meilleures solutions que l'heuristique $H2$ pour $n = 10$, 30, 50 navires et $k = 3$ avec des pourcentages : 74.5%, 22.3%, 16.3% et 90.4%, 31.2%, 21.8% pour le cas où $k = 5$, c'est aussi l'heuristique $H1$ fournit des meilleurs résultats que l'heuristique H3 avec 80%, 35%, 25% respectivement lorsque le $k = 3$ et 92.4%, 42.5%, 29.9% lorsque le $k = 5$.

Finalement, d'après ces exemples, il apparaît que l'heuristique $H1$ qui traite les navires selon la règle FIFO de leurs temps d'arrivée est la plus appropriée, puisque c'est elle qui fournit le meilleur résultat plus elle minimise bien la valeur de temps de manutention ainsi que la pénalité demandée.

Il semble alors que dans un objectif de minimisation notre critères d'optimalités, cette heuristique soit la plus performante des autres.

4.5 Conclusion

Ce chapitre à été consacré à l'implémentation et l'élaboration de notre heuristiques pour minimiser le temps de manutentuion. Nous avons reporté aussi, des résultats numériques obtenus par notre programme MATLAB sur un ensemble des exemples variés.

Nous avons présenté une comparaison des résultats obtenus pour obtenir la bonne solution qu'affirme l'efficacité d'une heuristique.

Conclusion générale

Le travail présenté dans ce mémoire se situ dans le cadre de l'étude de l'ordonnancement qui se pose dans un environnement à une seule machine en tenant compte des différents caractéristiques des tâches, l'objectif est de déterminer la complexité de quelques problème d'ordonnancement et leurs résolutions.

Au cours de ce mémoire, nous avons présenté rapidement les problèmes d'ordonnancement et les principes fondamentaux utilisés concernant ces problèmes en précisant les différents éléments : les tâches, les ressources, les contraintes et les objectifs. Le deuxième point nous avons exposé la classification des critères d'optimisation et ses propriétés, ainsi que quelques outils de modélisation (diagramme de Gantt, graphe potentiel, réseau de PERT). Au final, nous avons abordé quelques notions sur la théorie de la complexité (les classes de complexité, réduction des problèmes, etc...).

Nous nous sommes intéressés dans le deuxième chapitre d'un état de l'art des méthodes de résolution pour trouver de bonnes solutions aux problèmes correspondants en utilisant pour leur résolution, des méthodes exactes (branch & bound, la programmation linéaire et la programmation dynamique) et des métaheuristiques (méthodes de descente, recuit simulé, recherche tabou, algorithmes génétiques et algorithmes de colonies de fourmis).

Nous avons consacré le troisième chapitre à étudier quelques problèmes d'ordonnancement sur une seule machine qui sont distingués comme des problèmes faciles et difficiles, tout d'abord nous avons présenté la complexité des problèmes faciles avec quelques algorithmes de résolution. Ainsi que certains approches de résolutions des problèmes diffciles et leurs complexité en utilisant le problème 3-Partition pour les prouver, nous avons également illustré quelques problèmes étudiés par des exemples.

Au final, nous avons implémenté un problème pratique qui vise à améliorer l'efficacité d'un port qui planifier des navires au post à quai en termes d'optimiser le temps de manutention et de traitement des navires.

Durant notre étude, nous avons proposé des heuristiques pour atteindre notre objectif, ensuite nous avons illustré ces heuristiques par des exmples énumératifs codé en langage Matlab. Les résultats numériques obtenus ont été apportés et commentés.

Nous avons conclu que l'heuristique $H1$ génère des meilleures solutions que les heuristiques $H2$ et $H3$ pour des exemples variés, puisque elle minimise bien la valeur de temps de manutention ainsi que la pénalité demandée. Il semble alors que dans un objectif de minimisation notre critères d'optimalités, cette heuristique soit la plus performante des autres.

La famille des problèmes d'ordonnancement offre un large éventail de recherche. Le problème que nous avons étudié n'en est qu'une petite ramification. Pour approfondir l'étude du problème d'ordonnancement sur une seule machine, il serait intéressant d'étudier l'effet sur plusieurs machines.

Le domaine de notre problème pratique choisie est vaste et nous aimerions conclure en indiquant un certain nombre de perspectives de recherches futures :

- Nous envisageons la solution proposée dans ce mémoire pour les problèmes d'allocations d'un postes à quai dans un terminal à conteneur, peut être utilisée pour les autres types de terminaux.
- Utiliser un autre logiciel par une interface graphique qui met en oeuvre les heuristiques proposés pour bien visionner les résultats.

Une autre perspective intéressante serait non seulement d'améliorer les heuristiques mais également la recherche de nouvelles méthodes de résolution, et de tester ses performances avec d'autre outils de résolution exacte.

Nous espérons qu'avec ce travail, nous avons atteint les objectifs tracé et que cette recherche soit aussi d'un intérêt pour les esprits curieux dans le domaine de l'ordonnancement.

Annexe

Dans cette annexe on va citer les résultats de chapitre 3 qui sont résumés dans le tableau suivant :

Problème	*Facile/NP-difficile*	*Compléxité*
$1/pmtn, r_i/\sum C_i$	*Facile*	$O(n\log(n))$
$1//\sum w_iC_i$	*Facile*	$O(n\log(n))$
$1/outtree/\sum w_iC_i$	*Facile*	$O(n\log(n))$
$1/prec/L_{\max}$	*Facile*	$O(n^2)$
$1/prec, p_i = 1/L_{\max}$	*Facile*	$O(n)$
$1/prec, pmtn, r_i/L_{\max}$	*Facile*	$O(n^2)$
$1/p_i = 1/\sum w_iU_i$	*Facile*	$O(n\log(n))$
$1/r_i < r_j \Rightarrow d_i < d_j/\sum U_i$	*Facile*	$O(n\log(n))$
$1/prec/F_{\max}$	*Facile*	$O(n^2)$
$1/r_i, deliverytimes/C_{\max}$	*NP-difficile au sens fort*	/
$1/r_i, \tilde{d}_i/C_{\max}$	*NP-difficile au sens fort*	/
$1/r_i/\sum C_i$	*NP-difficile au sens fort*	/
$1/r_i/L_{\max}$	*NP-difficile au sens fort*	/
$1//\sum w_iT_i$	*NP-difficile au sens fort*	/

Tableau A.1 Classification de quelques problèmes d'ordonnancent sur une seule machine.

Bibliographie

[1] H.ADJIRI, *Les problèmes d'ordonnancement d'atelier M-Machine identique en parallèle*, Mémoire de Master en Mathématique et Informatique, 2018.

[2] T.BENTRCIA, *Ordonnancement des systèmes de production sous contraintes technologiques ou contextuelles : Modélisation, étude de complexité et approches intégrées de résolution*, Mémoire de doctorat en Génie Industriel, 2017.

[3] P.BRUCKER, *Complex scheduling*, édition Spring, 2006.

[4] H.BOUKEF, *Sur l'ordonnancement d'ateliers job-shop flexibles et flow-shop en industries pharmaceutiques : optimisation par algorithmes génétiques et essaims particulaires*, Mémoire de doctorat en Automatique et Informatique Industrielle, 2009

[5] A.CHERGUI, A.DAHMANI, *Ordonnancement d'un flow-shop par métaheuristique hybride*, Mémoire de Master en Génie industriel, 2017.

[6] D.DAI, R.FERTAS, *Modélisation et Optimisation du Temps de Manutention Au Niveau de BMT*, Mémoire de Master en Recherche Opérationnelle, 2017.

[7] M.DAOUDI, *Approches de résolution par les métaheuristiques de problèmes d'optimisation combinatoire NP-Difficiles*, Mémoire de doctorat en Informatique, 2012.

[8] L.Eugene, Q.Maurice, *Weighted sum of completion times.*

[9] B.FORTZ, *Modèles mathématiques et algorithmes pour l'ordonnancement*, INFO-F-425 , 2009

[10] GOTHA, *Les problèmes d'ordonnancements*, Revue française d'automatique, d'informatique et de recherche opérationnelle, 1993.

[11] H.ISSEMLELT, K.KELLAL, *Les méthodes PERT et MPM pour l'ordonnancement des taches d'un projet*, Mémoire de Master en recherche opérationnelle, 2016.

[12] B.Jacek, P.Erwin, W.Jan, *Scheduling computer and Manufacturing processes*, édition Spring-verlag, 2001.

[13] JEFFREY W. HERRMAN et al, *Handbook of Production Scheduling*, édition Springer, 2006.

[14] R.Jörg, *Complexity theory and cryptology*, Université Düsseldorf Germany, 2005.

[15] M.KEBABLA, *Utilisation des stratégies Métaheuristiques pour l'ordonnancement des ateliers de type Job Shop*, Mémoire de magister en Génie Industriel, 2008.

[16] KENNEth R. Baker, D.Trietsch, *Principles of sequencing and scheduling*, édition à john wiley sons, 2009.

[17] R.KIVIN, J.Billaut, *New objective functions based on jobs positions for single machine scheduling with deadlines*, International Conférence on Industriel Engineering et Systèmes Management, 2017.

[18] S.KOUIDER AID, *Cour d'ordonnancement*, master université Yahia Farés, 2019.

[19] A.LETOUZEY, *Ordonnancement interactif basé sur des indicateurs : Applications à la gestion de commandes incertaines et à l'affectation des opérateurs*, Thèse de Doctorat en Systèmes Industriels, 2001.

[20] J. Y-T LEUNG, *Handbook of scheduling algorithms, models, and performance analysis*, édition Sartaj Sahni, 2004.

[21] LOPEZ et ESQUIROL, *L'ordonnancement*, édition Economica, 1999.

[22] T.MEHENNI, *Utilisation des métaheuristiques pour résoudre un problème d'ordonnancement sur machine a contrainte de ressource non renouvelable*, Mémoire de magister en Informatique industrielle, 2006.

[23] M.L. Pinedo, *Scheduling Theory, Algorithms and Systems*, édition Spring, 1995.

[24] Muminu O. Adamu, *A survey of single machine scheduling to minimize weighted number of tardy jobs*, University of Kwazulu-Natal, 2014.

[25] H.T.Nguyen, *Complexité et algorithmes pour l'ordonnancement multicritere de travaux indépendants : problèmes juste-à-temps et travaux interférants*, Mémoire de Doctorat en Informatique, 2009.

[26] NIEBERG Tim, *Scheduling Single Machine Scheduling, Research Institute for Discrete Mathematics*, 2010.

[27] A.OULAMARA, *Contribution à l'étude des problèmes d'ordonnancement flow hop avec contraintes supplémentaires : Complexité et méthodes de résolution*, Mémoire de doctorat en Informatique, 2011.

[28] S.OURARI, *De l'ordonnancement déterministe à l'ordonnancement distribué sous incertitudes*, Mémoire de magister en Systèmes Industriels, 2011.

[29] CH.Philippe, *Ordonnancement sur une machine*, Module MOP-Cours 2 Spécialité IAD, 2015.

[30] CH.RAPINE, *Ordonnancement : entre théorie et applications*, université de Lorraine, 2018.

[31] F.SADI, *Algorithmes exacts et approchés pour les problèmes d'ordonnancement multi-agent à machines parallèles*, Mémoire de Doctorat en Informatique, 2015.

[32] O.SELT, *Méthaheuristique pour résoudre les problèmes d'ordonnancement des tâches sur des machines parallèles*, Mémoire de magister en Mathématiques discrètes 2008.

[33] E.TOURNIAIRE, *Problèmes NP-difficiles : approximation modérément exponentielle et complexité paramétrique*, Mémoire de doctorat en Informatique ,2013.

[34] E.TRICHET, *Introduction à la complexité algorithmique*, 2009.

[35] H.Zdenek, *Scheduling*, CTU in Prague, 2018.