A-PDF MERGER DEMO

N° d'ordre:

UNIVERSITE DE M'SILA

FACULTE DES SCIENCES ET DES SCIENCES DE L'INGENIEUR

Département de Mathématiques

MEMOIRE

Présenté pour l'obtention du diplôme de Magistère

Spécialité: Mathématique

Option: Mathématique discrètes

P a r

SELT Omar

SUJET

Méthaheuristique, pour résoudre les problèmes d'ordonnancement des tâches sur des machines parallèles

Soutenu publiquement le ..... **devant le jury composé de:**

Mr. NADIR Mostefa	Pr.	Université de M'sila	Président
Mr. BELOUADAH Hocine	M. C.	Université de M'sila	Rapporteur
Mr. BOUDERAH Brahim	M. C.	Université de M'sila	Examinateur
Mr. BOUBETRA Abdelhak	M. C.	Centre universitaire BBA	Examinateur
Mr. BENTERKI Djamel	M. C.	Université de Sétif	Examinateur

Promotion : 2007/2008

A-PDF MERGER DEMO

Remerciements

Je remercie tous les enseignants du département de mathématique, pour leur aide précieuse, surtout Mr. LKHALI Belkacem, Mr. BOUDERAH Brahim, Mr. BENHAMIDOUCHE Noureddine, Mr. AMROUNE Abelaziz, Mr. Mihoubi Douadi, Mr. BOUDAOUD Abdelmajid, Mr. MOSTFA Nadir et un remerceiment spécial pour mon encadreur Mr. Belouadah Hocine pour ses conseils et son suivi durant toute la réalisation de ce mémoire et Mr. LAKEHAL Meftah et Mr. Lounas pour leur aide.

Résumé:

Le travail de cet mémoire concerne l'étude du problème d'ordonnancement des tâches sur machines parallèles avec périodes d'indisponibilité où chaque tâche a une durée d'exécution et un poids lié à son importance et l'objectif est de déterminer une séquence de tâche sur les machines afin de minimiser la somme pondérée des dates de fin. Le problème est connu d'être NP- difficile. Il n'a jamais été résolu par la méthode dite taboue –à notre meilleur connaissance- ici on a proposé l'application de cette métaheuristique pour déterminer une solution approchée et moins coûteuse de point de vue temps l'exécution.

Le cas d'une seule machine est aussi étudié séparément.

Mots clés: Ordonnancement – machines parallèles – métaheuristique.

Abstract :

The work of this thesis concerns the study of the parallel machine scheduling with the maintenances term periods, where each job has a processing time and a weight related to its urgency and the objective is to determine a sequence of the tasks on the machines so that the sum of weighted completion time is minimized. This problem is known to be NP- difficult.

To the best of our knowledge, this problem has not been solved using the tabou search method, here we propose an approximate solution based on this metaheuristic that is cheaper in terms of completion time.

The case of the unique machine is also handled separately.

Key words: Scheduling **-** parallel machine **–** metaheuristic.

مـــلخص:

الهدف من هذه المذكرة هو دراسة مشكلة الترتيبات لمجموعة من الأشغال على آلات متوازية بأدوار زمنية غير متاحة بحيث يرتبط كل عمل بزمن تنفيذ ووزن يحدد أهميته، بحيث نحصل على الترتيبة المناسبة لتقليص دالة المجموع الترجيحي لزمن انتهاء الأشغال، وللعلم أنه لا يمكن تحديد خوارزمي حدودي يضبط الحل للمشكل المطروح. لذلك لجأنا إلى استعمال طريقة البحث (تابو) مع تحديد وسائط هذه الطريقة التقريبية من أجل إيجاد حل تقريبي للمشكل، مع مراعاة تقليل زمن التنفيذ، وقد تطرقنا كذلك إلى المشكل على آلة واحدة كحالة خاصة.

كلمات مفتاحية: ترتيبات - آلات متوازية – ما وراء التقريبية.

Sommaire

Chapitre III : Résolution des problèmes d'ordonnancements par une métaheuristique

INTRODUCTION

Dans le problème d'une machine, le problème d'ordonnancement se résume à un problème de séquencement des opérations sur la machine.

Le problème à machines parallèles se caractérise par le fait que plusieurs machines sont possibles pour l'exécution d'un travail qui n'en nécessite qu'une d'un point de vue théorique, ce problème est une généralisation du problème à une machine et un cas particulier de problème d'atelier multi-machines.

Pratiquement, c'est un cas fréquemment rencontré dans des applications réelles, en particulier dans l'ordonnancement de traitement informatique. Les problèmes d'ordonnancement à machines parallèles se divise en trois classes.

- **Identiques:** La durée de traitement est la même sur toutes les machines (notre cas).
- **Uniforme:** La durée de traitement varie uniformément en fonction de la performance de la machine
- **Indépendantes:** La durée est totalement variable entre les différentes machines.

Dans le chapitre I, nous abordons les définitions générales concernant la théorie d'ordonnancement sur une machine, ainsi que l'ordonnancement sur des machines parallèles.

Dans le chapitre II, nous illustrons par distinction et explication les méthodes de résolution des problèmes d'ordonnancement des taches sur une seule machine et machines parallèles en citant les méthodes exactes et les méthodes approchées, nous comparons spécifiquement ces deux derniers types de méthodes en évoquant les avantages et les inconvénients de chacune.

Dans le chapitre III, nous discutons en détail les métaheuristiques comme méthodes approchées en spécifiant la méthode de recherche taboue appliquée sur les différents types de problèmes d'ordonnancement. Citons que la méthode de recherche taboue c'est une métaheuristique originalement développée par **[Glo 86]** et indépendamment par **[Han 89],** cette méthode combine une procédure de recherche locale avec un certain

nombre de règle et de mécanisme permettant à celle-ci de surmonter l'obstacle des optima locaux, tout en évitant de cycler. Il y a plusieurs travaux de recherche sur ce même sujet citons **[Sch 84]** qui a étudié le problème d'ordonnancement sur machines parallèles avec différents intervalles d'indisponibilité. Dans **[Kac 05]**, une étude comparative entre deux méthodes exactes qui sont la méthode de la programmation en nombre entier et la méthode de Branch and Bound a été effectuée pour minimiser la somme pondérée des dates de fin des tâches, en fin s'arrive notre proposition qui consiste une tentative de développer la méthode de recherche taboue amélioré sur des machines parallèles avec contraintes d'indisponibilité en faisant des permutations par block et a la permutation entre deux tâches adjacents.

Définition générale des problèmes d'ordonnancement.

1- Introduction:

- Le problème d'ordonnancement consiste à organiser le temps de réalisation de taches, compte tenu des contraintes temporelles (délais, contraintes d'enchaînement) et de contraintes portant sur l'utilisation et la disponibilité des ressources requises.
- Un ordonnancement constitue une solution au problème d'ordonnancement, il décrit l'exécution des taches et l'allocation des ressources au cours du temps et vise à satisfaire un ou plusieurs objectifs. De manière plus précise, on parle d'ordonnancement lorsqu'on réserve le terme séquencement au cas où seul l'ordre relatif des tâches est fixé indépendamment des dates d'exécution.

2- 1-1 Les tâches:

- Une tache **i** est une entité élémentaire de travail (opération ou ensemble d'opérations). localisée dans le temps par une date de début ri ou fin C_i , dont la réalisation nécessite une durée $p_i = C_i - r_i$ et qui consomme des moyens (matériels, personnels, monétaires…).

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.

- L'ensemble des tâches est généralement noté **I**, le nombre de tâches noté par **n** et chaque tâche est noté par **i**. Et elle est décrite par les caractéristiques suivantes:

- r_i**:** date de disponibilité de tâche i (date de début au plus tôt)
- t_i**:** date de début d'exécution de la tâche **i**.
- C_i**:** date de fin d'exécution de la tache **i**.
- p_i**:** durée d'exécution de la tache **i**.
- d_i**:** date échue de la tâche i (date de fin au plus tard: deadline).
- w_i**:** poids de la tache i si elle est achevée après sa date échue **($C_i > d_i$)**

- **T_i:** retard d'exécution de la tâche **i** , **$T_i = \max\{C_i - d_i , 0\}$** , (Tardiness)
- **E_i:** avance de la tâche **i, $T_i = \max\{d_i - C_i , 0\}$, (Earliness).**

  Il est clair que $\forall i \in I, \quad P_i = C_i - t_i, \quad r_i \leq t_i < C_i \leq d_i$

1-2 Les contraintes:

- Dans la plupart des problèmes d'ordonnancement les tâches à exécuter sont soumises à des contraintes qu'il faut satisfaire au moment de la recherche d'une solution optimale, on distingue trois types de contraintes.

1-2-1 Les contraintes potentielles:

- Ce sont les contraintes de localisation temporelle par exemple " la tache **i** doit précéder la tache **j** " ou la tache **i** doit être achevée avant telle date. Avec ce type de contrainte, le problème est dit "problème centrale d'ordonnancement".

1-2-2 Les contraintes disjonctives:

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

1-2-3 Les contraintes cumulatives:

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

1-3 Les ressources:

- Les ressources sont les moyens requis pour l'exécution des tâches, il sont de deux types.
- - Une ressource est consommable si, après avoir été allouée à une tache, elle n'est plus disponible pour les autres tâches. Comme le cas d'une matière première, de l'argent.
- - Une ressource est renouvelable si, après avoir été allouée à une tâche, elle redevient disponible, à l'achèvement de celle-ci, pour les autres tâches, comme le cas d'une machine, un processus, une imprimante…

2- Les critères:

Le critère d'un problème d'ordonnancement est la fonction économique qu'on vise à optimiser. On l'appelle également dans un autre sens (fonction objectif).

En général, le critère est considéré comme une application **F** de l'ensemble ∑. des permutations des tâches du problème posé vers $\mathbf{R^+}$ qui, à chaque permutation $\mathbf{(j_1, j_2,\ldots j_n)}$ fait associer un réel positif $\mathbf{F(j_1, j_2, \ldots .j_n)}$ et qui exprime, en outre l'utilisation efficace des ressources, le délai global d'exécution et le respect du plus grand nombre de contraintes, la fonction objectif associé à un problème d'ordonnancement est définie par:

$$F\,obj = \underset{\sigma\in\Sigma}{OPT}\ F(\sigma)$$

Exemples de critères:

- Durée totale: C_{max}

Dans les problèmes d'ordonnancement où on vise à minimiser la durée totale d'exécution des tâches, le critère défini par $C_{\max}(\sigma) = \max_{i\in I}\{C_i\}$

Tel que : **I** l'ensemble des tâches.

i est une tache et $\mathbf{C_i}$ la date d'achèvement de la tâche **i** , $\mathbf{C_i = t_i + P_i}$

c'est donc la date d'exécution de la dernière tâche exécutée.

La fonction objective s'écrit:

$$Fobj = \min_{\sigma\in\Sigma}\{C_{\max}(\sigma)\}$$

- Retard global:

Dans les problèmes d'ordonnancement où on vise à minimiser le retard global d'exécution des tâches, le critère est défini par:

$$T_{\max}(\sigma) = \max_{i\in I}\{T_i\}$$

où: σ est une permutation des tâches, **i** est une tâche et $\mathbf{T_i}$ le retard d'exécution de la tâche **i**

$\mathbf{T_i = (0, C_i - d_i)}$.

La fonction objectif s'écrit:

$$Fobj = \min_{\sigma\in\Sigma}\{T_{\max}(\sigma)\}$$

- Somme des retards: ΣT_i

La fonction objectif s'écrit:

$$Fobj = \min_{i \in I} \left\{ \sum T_i \right\}$$

- Somme pondérée des retards: $\Sigma w_i T_i$

La fonction objective s'écrit:

$$Fobj = \min_{i \in I} \left\{ \sum w_i T_i \right\}$$

w_i coût unitaire (par unité de temps) de retard de la tache **i** si elle est achevée en retard **($C_i > d_i$)**

- Nombre de tâches en retard: $\sum U_i$

La fonction objectif s'écrit: $Fobj = \min_{i \in I} \left\{ \sum U_i \right\}$

$$U_i = \begin{cases} 1 & \text{si la tâche i est achevée en retard } (C_i > d_i) \\ 0 & \text{si non} \end{cases}$$

- Nombre pondéré de taches en retard: $\sum w_i U_i$

$$Fobj = \min_{i \in I} \left\{ \sum w_i U_i \right\}$$

w_i Le coût unitaire (par unité de temps) de retard de la tache **i** si elle est achevée en retard **($C_i > d_i$)**

- Le retard moyen:

$$\overline{T} = \sum T_i / n$$

$Fobj = \min_{i \in I} \left\{ \sum T_i / n \right\}$ où "n" est le nombre de taches.

- La somme des coûts des avances et des retards:

$$Fobj = \min_{i \in I} \left\{ \sum \left(\alpha_i . E_i + \beta_i T_i \right) \right\}$$

- **E_i :** avance de la tache **i** (Earliness)
- **T_i :** retard de la tache **i** (Tardiness)
- **α_i :** coût unitaire d'avance associé à la tache **i** .
- **β_i :** coût unitaire de retard associé à la tache **i**.

2-1 Critère d'optimisation régulier:

Habituellement, les critères d'optimisation sont exprimés en fonction de l'ensemble des dates de fin d'un ordonnancement, de telle sorte que leur forme générale est toujours:

$Z(\sigma)=R(C_{\sigma(1)},\ldots\ldots,C_{\sigma(n)})$ où σ est un ordonnancement $\sigma(i)$ est la tache qui est affectée à la $i^{ème}$ position dans σ et C_i est la date de fin de la tache **i** dans σ.

Soit σ' un ordonnancement, on désigne $C_{i'}$ par la date de fin de tache i' dans σ'. Le critère Z est régulier si est seulement si l'implication suivante est vraie, quel que soit l'ordonnancement σet σ'

$$\left[C_{\sigma(1)} \le C_{\sigma'(1)},\ldots\ldots C_{\sigma(i)} \le C_{\sigma'(i)}\ldots\ldots\ldots \; et \;\; C_{\sigma(n)} < C_{\sigma'(n)}\right] \Rightarrow \left[Z(\sigma) \le Z(\sigma')\right]$$

2-2 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.

2-3 Représentation des solutions:

Le diagramme de Gantt est certainement le type de représentation le plus ancien. Le plus répondu et le plus simple pour visualiser graphiquement l'exécution des taches et l'occupation des ressources au cours du temps.

Chaque tâche est symbolisée par un rectangle dans lequel sont inscrits le code et la durée de la tache, commençant par représenter les taches avec antériorité, une tache est portée dans le diagramme lorsque toutes ses antériorités y ont déjà portées cette représentation indique bien le développement des travaux.

3- Représentation des problèmes d'ordonnancement:

Une classification peut s'opérer selon:

Le nombre de machines et leurs ordre d'utilisation pour fabriquer un produit qui dépend de la nature de l'atelier. Un atelier est définit par le nombre de machines qu'il contient et par son type. Les différents types possibles sont les suivants:

- **Une machine (ϕ):** chaque travail est constitué d'une seule opération.
- **Machines parallèles:** elles remplissent, à priori, toutes les mèmes fonctions, selon leur vitesse d'exécution, on distingue:

  *** Machine identique (P):** la vitesse d'exécution est la même pour toutes les machines $\mathbf{M_i}$ et tous les travaux $\mathbf{J_i}$.

  *** Machine uniforme (Q):** chaque machine $\mathbf{M_i}$ a une vitesse d'exécution propre et constante, la vitesse d'exécution est la même pour tous les travaux $\mathbf{J_i}$ d'une même machine $\mathbf{M_j}$

  *** Machines indépendantes (R):** la vitesse d'exécution est différente pour chaque machine $\mathbf{M_j}$ et pour chaque travail $\mathbf{J_i}$.

  *** Machines dédiées:** elles sont spécialisées à l'exécution de certaines opérations. Dans cette catégorie, chaque travail est constitué de plusieurs opérations. En fonction du mode de passage des opérations sur les différentes machines, trois ateliers spécialisés sont différenciés à savoir:

Flow Shop (F): c'est un cas particulier du problème d'ordonnancement d'atelier pour lequel le cheminement des travaux est unique:

Les **n** travaux utilisent les **m** machines dans l'ordre **1, 2, ……, m** (ligne de production).

La plupart de la littérature est limitée au cas particulier du Flow Shop de permutation. Un ordonnancement de permutation est un ordonnancement qui conserve le même ordre des travaux sur toutes les machines.

Job Shop (J): Les **n** travaux doivent être exécutés sur les **m** machines, sous des hypothèses identiques à celles du Flow Shop, la seule différence est que les séquences opératoires relatives aux différents travaux peuvent être distinctes et sont propre à chaque travail.

Open Shop (O): Est un modèle d'atelier moins contraint que le Flow Shop car l'ordre d'exécution des opérations n'est pas fixé.

Notation:

- Afin d'identifier d'une manière synthétique et précise le type du problème d'ordonnancement abordé, on utilise une notation à trois champs α / β / γ:

$\alpha = \alpha_1 \alpha_2$: décrits les caractéristiques des machines:

$\alpha_1 \in$ { b, P, , R, O, F, J}

b: ordonnancement sur une seule machine.

P: ordonnancement sur plusieurs machines parallèles et identiques.

q: ordonnancement sur plusieurs machines et uniforme.

R: ordonnancement sur plusieurs machines parallèles et différentes.

O: il s'agit d'un problème (Open Shop).

F: il s'agit d'un problème (Flow Shop).

J: il s'agit d'un problème (Job Shop).

α_2^E { b, K}

b: le nombre de machines est variable.

K: le nombre de machines est K (K entier positif).

B: l'ensemble des contraintes.

γ: le critère à optimiser.

Exemples:

* $1//\sum w_i C_i$: Représente le problème de minimisation de la somme pondérée des dates de fin des taches indépendantes sur une seule machine.

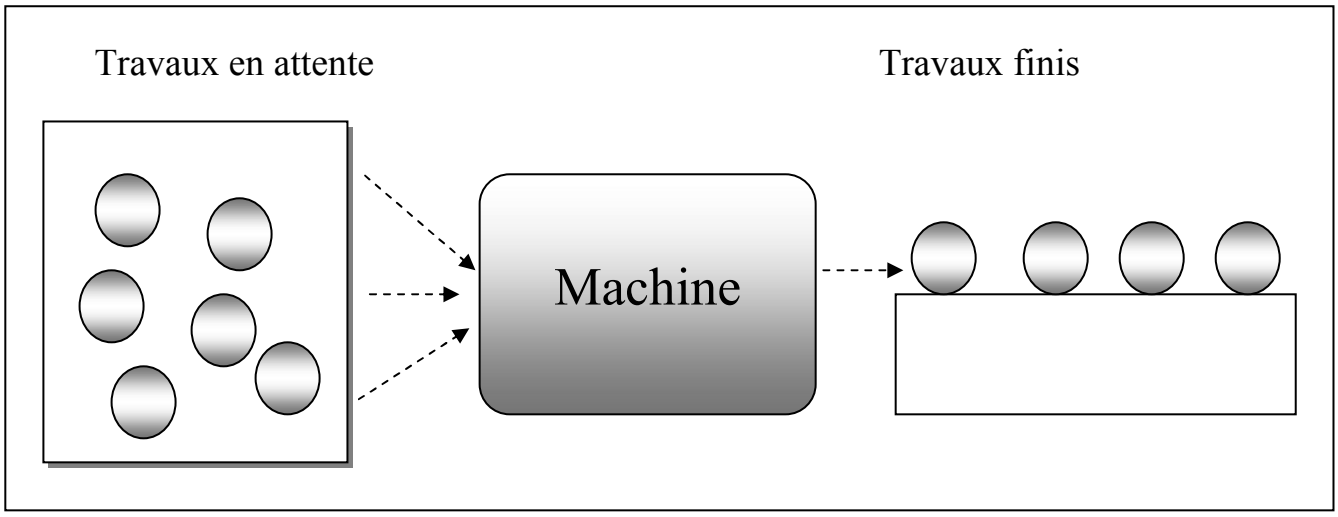
* $F3 \mid V_i \mid \sum T_i$: Représente les problèmes de minimisation de la somme des retards en Flow Shop à 3 machines avec contrainte des dates de disponibilité.

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 models les plus connus sont ceux d'une machine unique, de machines parallèles, d'un atelier à cheminement unique ou d'un atelier acheminement 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.



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 (voir figure 01)

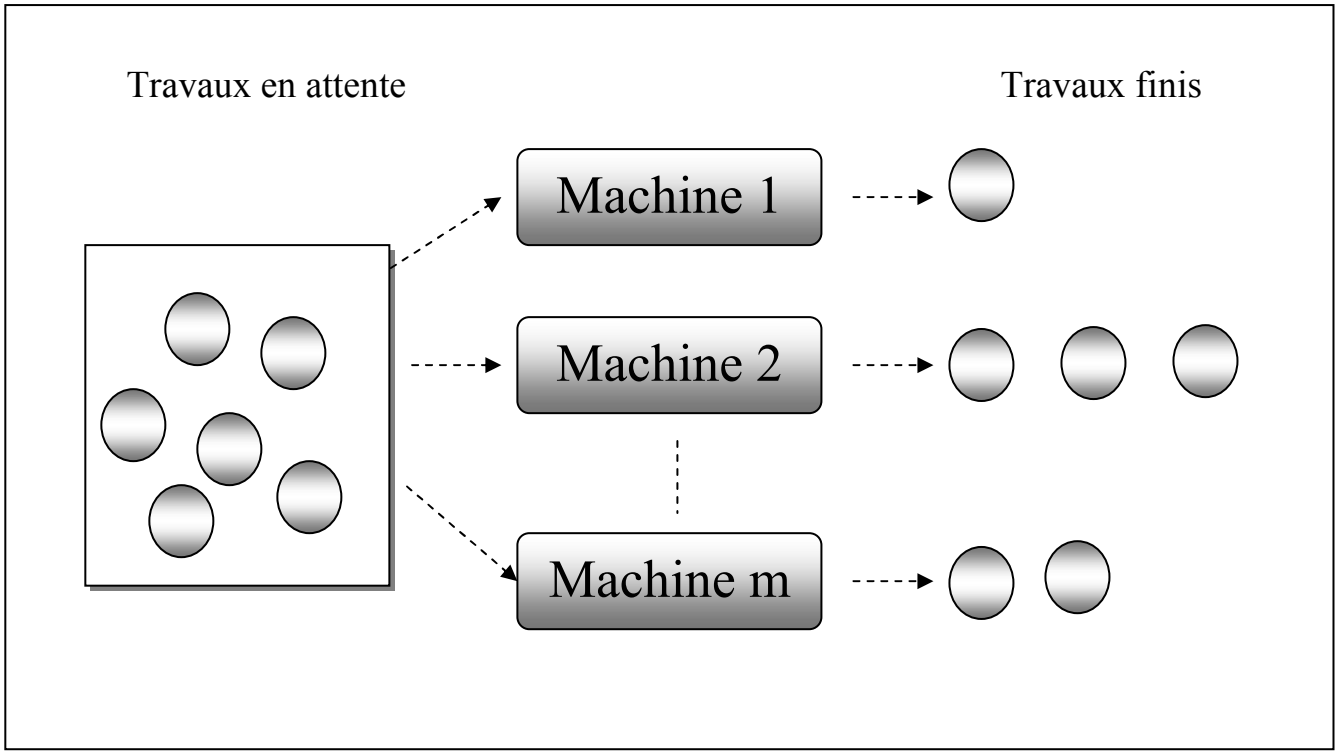


Figure (01)

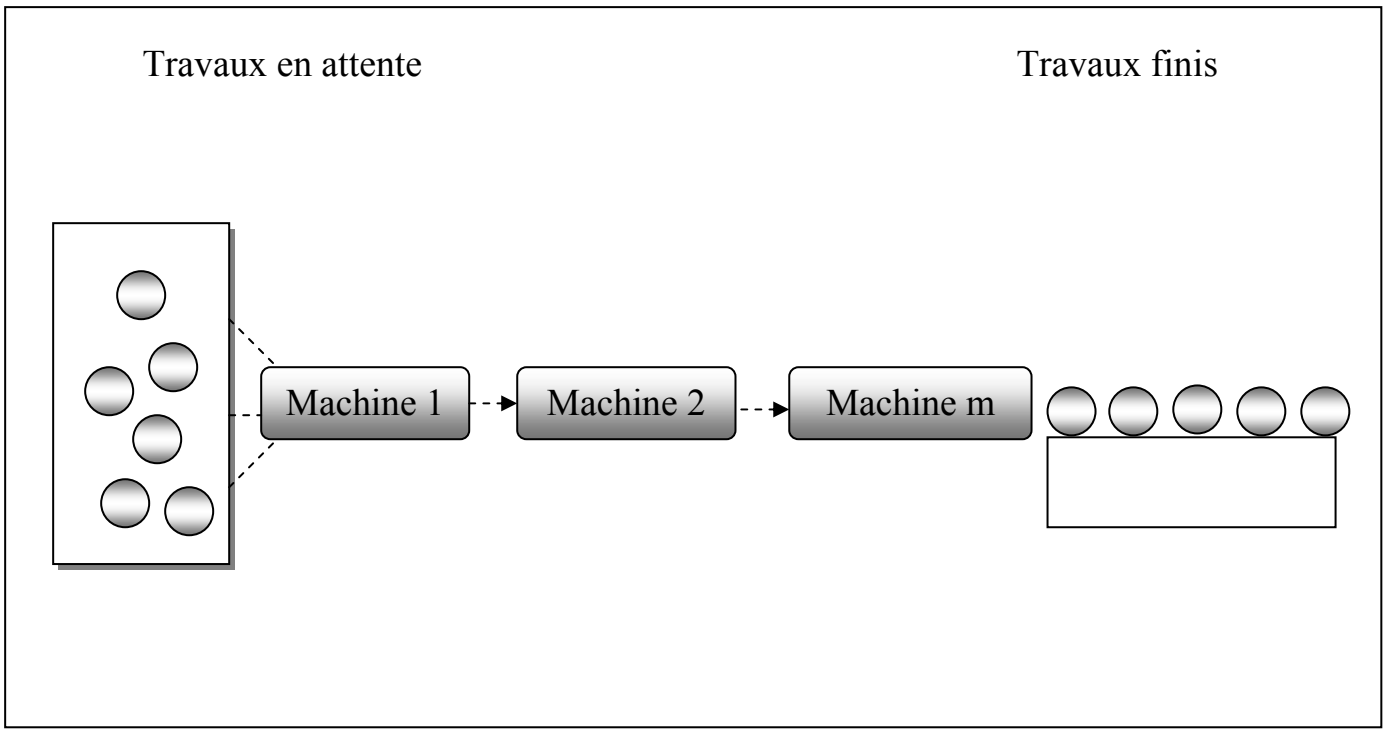
Atelier à cheminement unique (Flow Shop)

Un atelier à cheminement unique est un atelier où le processus d'élaboration de produits est dit linéaire c'est-à-dire lorsque les étapes de transformation sont identiques pour tous les produits fabriqués. Selon les types de produits élaborés, on distingue la production continue et la production discrète. La production continue est caractérisée par la fluidité de son processus et l'élimination de stockage. C'est le cas notamment dans les raffineries, les cimenteries, les papeteries… . la production discrète de masse s'applique principalement aux produits de grande consommation fabriqué à la chaîne (la majorité du domaine du textile, machine-outils).dans les deux cas les machines peuvent être dédiées à une opération précise, et sont implantées en fonction de leur séquence d'intervention dans la gamme de production.

L'un des objectifs principaux dans le cas d'atelier à cheminement unique est de trouver une séquence des tâches en main qui respecte un ensemble de contraintes et qui minimise le temps total de production. Parmi les caractéristiques d'un problème de cette catégorie.

- Il existe au minimum n! différentes solutions où n est le nombre de travaux réalisés. Notons que n! = n*(n-1)*(n-2)….*1

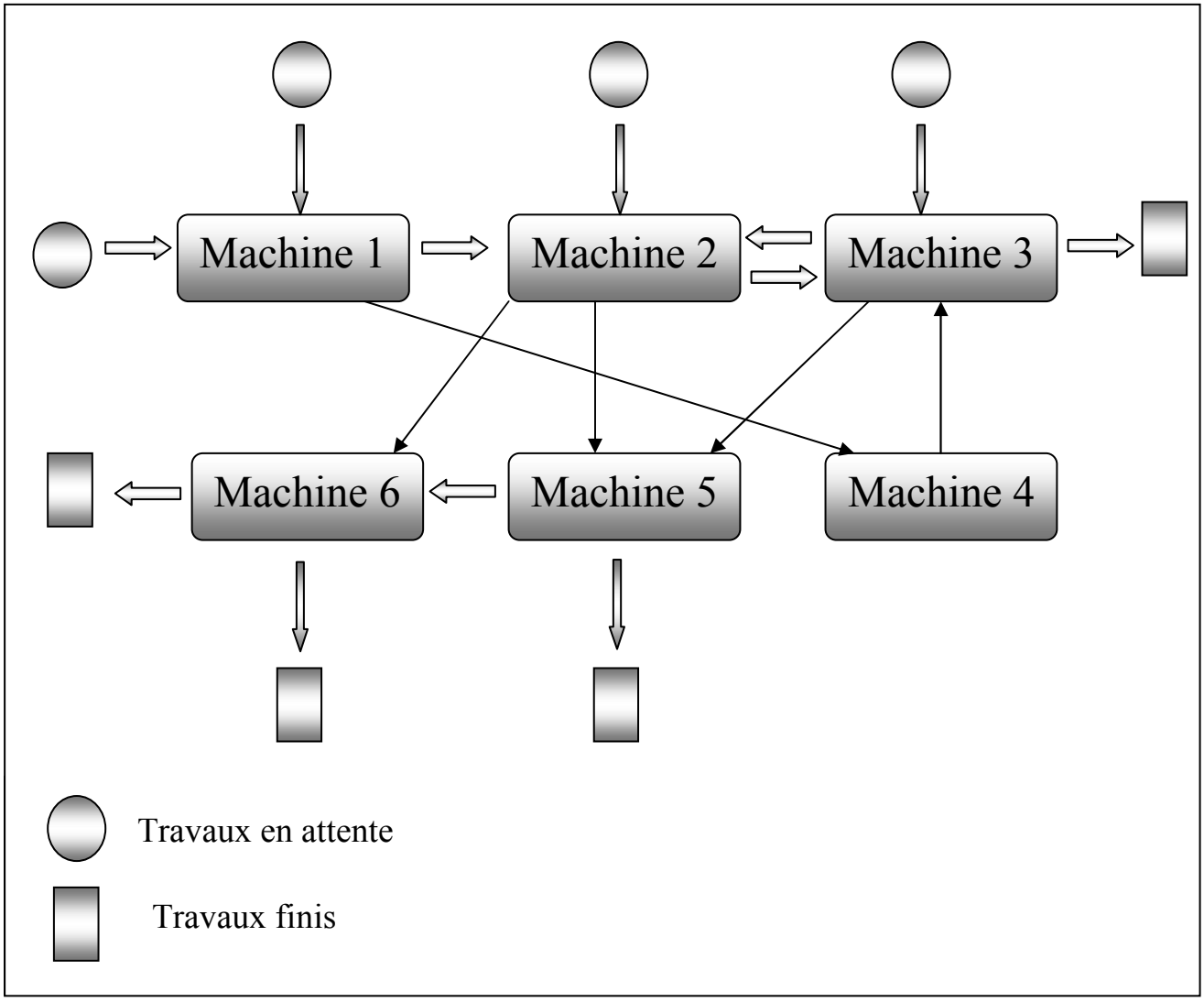
- Le problème est NP-difficile à l'exception des versions avec deux machines et certains cas particuliers avec trois machines.
- Une grande productivité



Atelier à cheminement multiple (Job Shop):

Les ateliers à cheminement multiple (ACM) sont des unités manufacturières traitant une variété de produits individuels dont la production requière divers types de machines dans des séquences variées. L'une des caractéristiques d'un atelier à cheminement multiple est que la demande pour un produit particulier est généralement d'un volume petit ou moyen. Une autre caractéristique est la variabilité dans les opérations et un mix produit constamment changeant. Ainsi, il est nécessaire que le système soit de nature flexible. Dans un sens général, la flexibilité et la capacité d'un système de répondre aux variations dans l'environnement. L'objectif le plus considéré dans le cas d'un atelier à cheminement multiple est le même que celui considéré pour un atelier à cheminement unique, à savoir trouver une séquence de tâches sur les machines qui minimisent le temps total de production.

La figure suivante montre un exemple d'un atelier à cheminement multiple avec quatre travaux et six machines.



Parmi les caractéristiques d'un problème d'ordonnancement dans un atelier à cheminements multiples:

- Le nombre de solutions possibles est de l'ordre de $(n!)^m$, où n est le nombre de tâches à effectuer et m le nombre de machines. Notons qu'une tâche veut dire la même chose qu'un travail.
- Le problème est NP-difficile et est considéré parmi les problèmes les plus difficiles à traiter.

3-1 Notion de dominance:

Un sous ensemble, 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 au moins une solution admissible, s'il en existe. La recherche d'une solution optimale ou admissible peut ainsi être limitée à un sous ensemble dominant.

4- Théorie de la complexité des l'algorithmes:

Les problèmes d'ordonnancement d'ateliers, sont des problèmes combinatoires, extrêmement difficiles et il n'existe pas de méthode universelle permettant de résoudre efficacement tous les cas, beaucoup d'entre eux peuvent prendre un temps considérable pour être résolus, la théorie de la complexité des algorithmes a donné uns sens précis au terme d'algorithme efficace et de problème difficile, citons quelques concepts de base de cette théorie.

- **La complexité d'un algorithme:**

Evaluer la complexité d'un algorithme consiste à trouver en fonction de la taille des données, le nombre d'opération élémentaires nécessitées par cet algorithme.

- **Un algorithme polynomial:**

Est un algorithme dont le nombre d'opérations élémentaires nécessaire pour résoudre un exemple de taille **n** est une fonction polynomiale en **n**.

- **Algorithme non polynomial:**

Est un algorithme dont le nombre d'opérations n'est pas donné par un polynôme de **n**.

- **Un algorithme efficace:**

On dit qu'un algorithme est efficace si le nombre des opérations nécessaires pour résoudre un problème est donné par une fonction polynomiale d'un paramètre caractérisant la taille du problème considéré.

4-1 Problème de décision:

Est un problème dont les résultats ne peuvent prendre que l'une des deux valeurs: VRAI ou FAUX.

- **La classe NP:**

La classe des problèmes NP (NP pour Non détermination Polynomiale) est la classe des problèmes des décisions qui peuvent être résolus par une machine de Turing non déterministe en temps polynomial.

Parmi la classe de problème NP, on distingue deux grandes classes:

- La classe des problèmes polynomiaux (la classe P) et la classe des problèmes NP-Complet (la classe NPC).

- **La classe P:** Un Problème est dit appartenir à la classe P s'il existe un algorithme polynomial pour le résoudre. On dit que les problèmes de la classe P sont faciles.
- **La classe NP-Complet:**

Un problème de décision est dit NP-Complet si tout problème de la classe NP peut se rendre polynomialement à lui.

- 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 peut ne pas trouver d'algorithme polynomial, quelque soit le codage, ceux-là sont dits NP- difficiles au sens fort.

Problèmes d'optimisation:

La théorie de la complexité est construite à partir des problèmes de décisions. Pour les problèmes d'optimisation, il s'agit d'optimiser une mesure sur un ensemble de solutions réalisables. Un problème d'optimisation Π sera défini par la donnée de:

- Un ensemble I d'instances.
- Une fonction **S** telle que pour tout $\mathbf{x} \in \mathbf{I}$, **S(x)** représente l'ensemble des solutions réalisables pour **x**.
- Une fonction **m** à valeur entière définie sur tous les couples $\mathbf{x} \in \mathbf{I}$ et $\mathbf{y} \in \mathbf{S(x)}$. cette mesure **m** représente la fonction objectif d'une solution **y** l'instance **x**.
- Un objectif $\in \{\min, \max\}$ précisant si Π est un problème de minimisation ou de maximisation.

A toute instance **x**, on associe une solution **y*(x)** et la valeur **m*(x)** définissant un optimum. Si Π est un problème de minimisation, $\mathbf{m^*(x)=\min\{m(x, y) \mid y \in S(x)\}}$ et **y*(x)** est une solution vérifiant $\mathbf{m(x, y^*(x)) = m^*(x)}$ [RAP 02].

A un tel problème d'optimisation Π, on peut associer 2 versions différentes selon la nature du résultat attendu (fonction calculée par un algorithme de résolution):

- Π_C (par défaut Π), le problème de construction. Pour chaque instance **x**, on doit fournir en sortie la valeur **m*(x)** et une solution **y*(x)** réalisant cet optimum.

- Π_D, le problème de décision associé. Une instance de Π_D est une instance **x** de Π plus entier **K**. La question est: existe-il une solution **y** vérifiant $\mathbf{m(x, y) \leq K}$ pour le problème de **min** **(resp. $\mathbf{m(x, y) \geq K}$ pour max)**.

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 q'un problème Π est au moins aussi difficile qu'un problème Π' qui se réduit à Π.

Un problème Π' se réduit à Π si l'on peut écrire une procédure pour résoudre Π' en connaissant une sous-procédure pour résoudre Π. On peut alors définir la NP-difficulté de la manière suivante [RAP 02]

Définition:

Un problème Π est dit NP-difficile ou NP-dur, ssi seulement 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 la propriété suivante [RAP 02].

Propriété:

Soit Π un problème d'optimisation. Si son problème de décision associé $\Pi_D \in$ NP-complet, alors $\Pi \in$ NP-difficile.

La plupart des problèmes d'ordonnancement sont NP-difficile car ils sont des problèmes d'optimisation. Toutefois il existe quelques cas particuliers pour les quels on a trouvé des algorithmes polynomiaux, comme pour le problème :

$1| r_i , pmtn | \sum C_i$ résolu par Baptiste et al.

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.

Introduction:

La résolution optimale d'un problème d'ordonnancement est de trouver une séquence d'exécution des tâches sur machine de sorte qu'un certain critère d'optimisation atteigne sa valeur optimale.

Les méthodes de résolutions des problèmes d'ordonnancements puissent dans toutes les techniques de l'optimisation combinatoires (programmation mathématique, programmation dynamique, théorie des graphes …).

Ces méthodes garantissent en général l'optimalité de la solution fournie. Mais les algorithmes dont la complexité n'est pas polynomiale ne peuvent pas être utilisés, des problèmes de grande taille, d'où la nécessité de construire des méthodes de résolution approchée, qu'on appelle méthodes heuristiques, efficaces pour les problèmes NP- difficiles, dans la suite nous présenterons les méthodes de résolutions les plus connues.

1- Les méthodes exactes:

La résolution se ramène à l'exploration de l'ensemble des séquences possibles à travers d'un arbre de recherche. Le temps de calcul nécessaire d'une telle méthode augmente en général exponentiellement avec la taille du problème à résoudre. Elle sont issues de la recherche opérationnelle, le but de ces méthodes est de trouver en un temps de calcul, le plus court possible, la solution optimale du problème, ainsi la résolution se ramène à l'exploration de l'ensemble de séquences possibles au travers d'un arbre de recherche.

Pour améliorer l'efficacité de la recherche, on utilise des techniques pour calculer des bornes permettant d 'élaguer le plus tôt possible des branches conduisent à un échec, de plus on emploi des heuristiques pour guider les choix de variables et de valeurs durant l'exploration de l'arborescence. Parmi les méthodes exactes on distingue.

1-1 Enumération complète:

On recherche un ordonnancement optimale à travers tous les ordonnancements possibles, l'énumération complète génère les ordonnancements possibles. Dans le cas des problèmes d'ordonnancement non préemptif sur une machine, il existe **n!** séquences

d'exécution des tâches possibles, par conséquent pour le problème de **m** machines correspondant il y a $(n!)^m$ séquences d'exécution possible, ce nombre est très grand. Et bien qu'en pratique à cause des contraintes du problème, beaucoup d'ordonnancement peuvent être inadmissibles et que des procédures d'élimination peuvent parfois être utilisées pour voir si la non optimalité d'un ordonnancement implique la non-optimalité d'autre, non encore gênés, le temps de résolution peut être prohibitivement long même si on utilise le plus puissant et le plus rapide des ordinateurs, il est donc clair que, chercher une solution optimale utilisant l'énumération complète n'est pas convenable même pour des problèmes de petite taille. C'est pour cela qu'il était nécessaire de faire appel à des méthodes intelligentes.

1-2 Analyse combinatoire: (Cette méthode examine l'effet d'un changement mineur sur une séquence particulière, sur la valeur de cette séquence).

L'analyse combinatoire peut mener à des algorithmes très efficaces qui produisent un ordonnancement optimal en un nombre prévisible d'étapes où de dernier grandit au plus polynomialement en fonction de la taille du problème, cette méthode examine l'effet d'un changement mineur sur une séquence particulière, sur la valeur de cette séquence.

1-3 Programmation dynamique:

C'est une méthode fondée sur une approche récursive, s'applique à n'importe quel problème qui peut être divisé en une séquence de problèmes, la solution d'un problème d'ordre **n** s'exprime simplement en fonction de solutions de problèmes d'ordre**(n – 1)**.

La méthode de la programmation dynamique est une méthode générique pour les problèmes d'optimisation et en particulier il a été prouvé que c'est un outil à la fois souple et puissant pour traiter les problèmes de l'ordonnancement **[ABD 87]**, elle a été appliquée pour la première fois aux problèmes d'ordonnancement par **CARP** et **[HEL 62]**, et ensuite par **[LAW 64]**.

Cette méthode consiste à décomposer le problème posé en sous problèmes (ou phases), puis à établir une équation récursive exprimant la solution ou du sous problème d'ordre **K** en fonction de celle du sous problème d'ordre **(K- 1)**. Ainsi, le dernier sous problème

serait d'ordre **(n)**. en l'occurrence, c'est le problème à résoudre dans l'exemple ci-dessous nous illustrons l'approche de la programmation dynamique.

1// f_i, ou on minimise le coût totale $\sum f(C_J)$

Cette fonction récursive a été suggérée par **HEL** et **KARP** et ensuite **[Law 64]**

soit **J** $\subseteq$ **I = {1, 2, 3……..,n}** un ensemble de taches et **f *(J)** le coût total minimal de l'ordonnancement des taches de **J** sur la période $\left[0, \sum_{i \in j} P_i\right]$,

l'objectif est de trouver **f*(J)** en résolvant les équations récursives.

f*(J) = $\min_{i \in j}$ **{f*(J-{i}) + f_i (**$(\sum_{i \in J} Pi)$**}.**

qui sont initialisés par **f*(∅) = 0**

La programmation dynamique est une méthode d'énumération implicite, meilleur que l'énumération complète, mais dont les besoins en espèce mémoire sont souvent très grands.

1-4 Méthode par séparation et évaluation:

Ces techniques ont été appliquées pour la première fois aux problèmes d'ordonnancement par LaniNicKI et IGNAL.

Cette méthode représente les solutions possibles par un "arbre de recherche" qu'il faut explorer du haut en bas, en se servant d'une borne inférieure (Lower Bound), de la solution optimale, déterminée au préalable, on évolue à chaque nœud plus que la borne inférieure et n'explorer que celles qui améliorent cette borne inférieure, cette méthode permet donc de réduire le nombre de cas à explorer. Ce nombre dépend essentiellement de la qualité de la borne inférieure adoptée au départ.

- **L'arbre de recherche:**

L'arbre de recherche est la représentation schématique permettant de recenser systématiquement et explicitement toutes les **n!** permutations de **n** tâches. elle et apparut comme un ensemble de nœuds disposés en couches hiérarchiques et numérotées respectivement de 0 à (n-1) du plus haut au plus bas niveau.

La couche 0 contient un seul nœud, dit nœud racine, qu'on note **S**. La couche **n-1** est composé de **n!** nœud dits nœuds feuilles, les nœuds des autres couches sont dits nœuds internes.

Chaque nœud est un ensemble d'ordonnancement. En particulier **S** est l'ensemble de tous les ordonnancements (card s = n!), et tout nœud – feuille est un singleton nous désignons par I_y l'ensemble des taches déjà affectées dans le nœud **Y** ($I_S = \phi$ et $I_F = I$ si **F** est nœud – feuille) par 0 un nœud de la couche **i**, et par **D** un nœud de la couche **i+1** où **i** $\neq$ **n-1**

D est dit successeur (0) s'il en dérive directement c'est-à-dire que $I_D - I_0 = \{t\}$, où (t) est la tâche de branchement du nœud 0 au nœud **D** et que les tâches sont disposées identiquement dans **D** que dans 0.

Tout nœud possède $\mathbf{(I - I_0)}$ successeurs immédiats qui forment une partition de lui.

Un nœud est un descendant d'un autre s'il en dérive directement ou indirectement.

Tout nœud est le sommet d'un sous arbre de l'arbre de recherche.

Tout sous arbre contient tous les descendants de son sommet, par conséquent, pour supprimer un sous arbre, il suffit de supprimer son sommet.

En explorant l'arbre de recherche, la méthode par séparation et évaluation utilise une procédure de séparation pour partitionner successivement les nœuds.

Si le passage d'un nœud(0) à un successeur immédiat **D** est incompatible avec les contraintes du problème à résoudre, le nœud **D** sera supprimé. Sinon la méthode par séparation et évaluation permet de décider d'explorer ou de supprimer le sous-arbre de sommet **D**. Cet arbre est supprimé s'il a été jugé inutile de l'explorer car il ne contient aucune solution meilleur que celle disponible jusqu'à présent la méthode par séparation et évaluation est un exemple typique de l'approche d'énumération implicite, qui peut trouver une solution optimale en examinant systématiquement des ensembles de solutions admissibles.

- **Procédure de séparation:**

Cette procédure décrit la méthode utilisée pour partitionner un sous ensemble de solutions possibles, les procédures de séparations les plus usuelles sont:

- Branchement en arrière (backwards branching):

Les taches sont affectées une par une à partir de la fin.

- Branchement en avant (Forwards branching):

Les tâches sont affectées une par une du début.

- A chaque étape, une tache est choisie pour être effectuée soit au début ou à la fin selon une certaine méthode heuristique basée sur les données des problèmes (voir Hairari [HAR 81], chapitre 8 et 10)

- A chaque étape , une tâche est choisie pour être effectuée, soit avant soit après une autre tâche. Une heuristique a peut être été utilisée pour déterminer cette paire de tâches (voir Potts [POT 85]).

- A chaque étape, une tâche est choisie pour être affectée soit avant soit après une autre tâche. Une heuristique peut être utilisée pour déterminer cette paire de tâches (voir Hariri, [HAR 81]).

Pour minimiser le critère **f** d'un problème d'ordonnancement particulier, la méthode par séparation et évaluation utilise une procédure de calcul d'une borne inférieure dite fonction d'évaluation par défaut.

- **Procédure de calcul d'une borne inférieure:**

Il s'agit d'une fonction définie sur l'ensemble des nœuds de l'arbre de recherche, et qu'on note par **Lb** si **Y** un nœud quelconque, nous devrions être capable de calculer **Lb(Y)** et avoir: $\forall y \in \mathrm{Y} \;\; Lb\,(\mathrm{Y}) \leq f\,(y)$

Parmi les méthodes de calcul de bornes inférieures pour les problèmes d'ordonnancement on distingue (pour détails, voir [HAR 81], et Belouadah [Bel 85].

- Relaxation des contraintes du problème.
- Relaxation lagrangienne des contraintes (Bel 88).
- Relaxation "State Space" de la programmation dynamique ([CHR 81])
- Relaxation de l'objectif [ABD 87].

- **Borne supérieure:**

Si aucune solution admissible n'est connue, la borne supérieure **UB** est initialisée jusqu'à ce que la première solution admissible soit trouvée.

Pour un nœud **Y** n'ayant pas encore été supprimé, on calcule sa borne inférieure

Lb (Y). Si **Lb (Y) ≥ UB** alors on supprime **Y**, sinon **Y** est conservé en vue de l'explorer éventuellement ultérieurement.

L'arbre de recherche se réduit remarquablement lorsque les bornes inférieures sont aussi grandes que possibles pour poursuivre la recherche, le prochain nœud à explorer est choisi selon une stratégie de recherche.

- **Stratégie de recherche:**

Une stratégie de recherche permet de sélectionner dans l'arbre le prochain nœud à explorer, soit **Y** ce nœud cherché. Par convention le premier nœud à être exploré est le nœud racine **S**. généralement le nœud **Y** peut être choisi comme étant:

1- Le nœud ayant la plus petite borne inférieure.

2- Le nœud le plus récemment crée.

3- Le nœud ayant la plus petite borne inférieure parmi les nœuds des récemment crées

Les stratégies (2) et (3) sont la stratégie profondeur d'abord, alors que la stratégie (1) est la stratégie largeur d'abord.

La stratégie profondeur d'abord nécessite peut (très peut) d'espace mémoire, mais une quantité considérable de calculs. Par contre la stratégie largeur d'abord nécessite moins de calculs car elle choisi la branche à explorer d'une façon plus intelligente et donc, trouve une solution optimale plus vite. Malheureusement, cette performance exige des besoins plus grands en espace mémoire.

Quelque soit la stratégie de recherche utilisée, le déroulement de la méthode par séparation et évaluation sera terminée une fois que l'arbre de recherche est entièrement exploré et l'ordonnancement essai (trial Shedul) qui reste sera optimal.

- **Amélioration de l'efficacité:** il y a plusieurs façons pour améliorer l'efficacité de la méthode par séparation et évaluation, citons quelques ânes:

a/ Chois d'une stratégie de recherche: la stratégie utilisée est un déterminant important du chois temps nécessaire pour résoudre un problème, généralement largeur d'abord trouve un ordonnancement optimal plus vite qu'une stratégie profondeur d'abord [FRE 82]

b/ Choix de bornes inférieures:

La qualité des bornes inférieures est aussi un facteur important dans le temps de calcul nécessaire pour résoudre un problème, la borne inférieur **Lb (Y)** du nœud **Y** est bonne si elle

n'est pas trop petite que la plus petite valeur du critère d'optimisation au niveau de ce nœud, la méthode par séparation et évaluation trouve généralement un ordonnancement optimal après avoir examiner moins de nœuds avec bonnes bornes inférieures qu'avec mauvaises. Les bornes inférieures éliminant les nœuds qui sont en haut de l'arbre, ce qui réduit considérablement la recherche. Car lorsque un nœud situé en haut de l'arbre est éliminé, plusieurs nœuds subséquents sont éliminés au même temps.

c/ Règle de dominance:

Si on peut montrer qu'une solution optimale peut toujours être générée sans trouver un nœud particulier de l'arbre de recherche, ce nœud est dit dominé et peut être éliminé.

En résumé, les règles de dominance sont utiles dans la réduction du.

- Besoins en espace mémoire sur l'ordinateur
- Temps de calcul (car elles évitent des calculs pour les descendants de tout nœud éliminé).

d/ Initialisation de la borne supérieur (UB):

démarrer avec un ordonnancement presque-optimal (Near Optimal Shedule) peut entraîner l'élimination de beaucoup de nœuds lors des premières étapes de la recherche. On utilise des méthodes heuristiques, pour trouver un ordonnancement presque optimal, dans le cas échéant, un ordonnancement admissible.

2- Méthodes approchées:

Les méthodes approchées constituent une alternative très intéressante pour traiter les problèmes d'optimisation de grande taille si l'optimalité n'est pas primordiale, en effet, ces méthodes sont utilisées depuis longtemps par de nombreux praticiens.

Pour les grands problèmes NP-durs, il est même probablement impossible de trouver une solution optimale durant une durée de vie humaine, donc, dans le cas où on ne peut pas trouver un ordonnancement optimal au bout d'un temps raisonnable, on ne doit pas quitter

l'analyse mais plutôt utiliser notre expérience pour trouver en temps polynomial une solution réalisable, la meilleur que possible.

Les méthodes approchées sont fondées principalement sur diverses heuristiques, souvent spécifiques à un type de problème, les métaheuristiques constituent une autre partie importante des méthodes approchées et ouvrent des voies très intéressantes en matière de conception des heuristiques pour l'optimisation combinatoire.

2-1 Les méthodes de voisinage:

Les méthodes de voisinage sont fondées sur la notion de voisinage.

Définition:

Soit **X** l'ensemble des configurations admissibles d'un problème, on appelle voisinage toute application $N : x \to 2^x$, on appelle mécanisme d'exploration du voisinage toute procédure qui précise comment la recherche passe d'une configuration: **S** $\in$ **X** à une configuration **S′** $\in$ **N(s)**.

On dit que la configuration **S** est un optimum (minimum) local par rapport au voisinage **N** si:

$\forall S' \in N(S) : f(S) \leq f(S')$

Une méthode typique de voisinage débute avec une configuration initiale, et réalise ensuite un processus itératif qui consiste à remplacer la configuration courante par l'un de ses voisins en tenant compte de la fonction coût, ce processus s'arrête et retourne la meilleur configuration trouvée quand la condition d'arrêt est réalisée, cette condition peut être généralement une limite pour le membre d'itérations.

2-3 Algorithmes évolutifs:

Ces algorithmes sont basés sur le principe de processus d'évolution naturelle et les mécanismes d'évolution des espèces vivantes. Un algorithme évolutif typique est composé de trois éléments essentiels:

- une population constitué de plusieurs individus représentant des solutions du problème donné.

- Un mécanisme d'évaluation de l'adaptation de chaque individu de la population à l'égard de son environnement extérieur.
- Un mécanisme d'évolution composé d'opérateurs permettant d'éliminer certains individus et de produire de nouveaux individus à partir des individus sélectionnés.

Un algorithme évolutif débute avec une population initiale souvent générée aléatoirement et répète un cycle dévolution composée de trois étapes séquentielles.

- Mesurer l'adaptation (la qualité) de chaque individu de la population par le mécanisme d'évaluation.
- Sélectionner une partie des individus. Produire de nouveaux individus ou des recombinaisons d'individus sélectionnés.

Ce processus se termine quand la condition d'arrêt est vérifiée, par exemple quand un membre maximum de génération est atteint. Selon l'analogie de l'évolution naturelle, la qualité des individus de la population devrait tendre à s'améliorer ou fur et à mesure des processus.

- Un algorithme évolutif comporte: un ensemble d'opérateur tel que la sélection, la mutation et éventuellement le croisement.

La sélection a pour objectif de choisir les individus qui vont pouvoir suivre et se reproduire pour transmettre leur caractéristiques à la génération suivante elle est basée généralement sur la conservation des individus les mieux adaptés et l'élimination des moins adaptés.

Le croisement cherche à combiner les caractéristiques des individus parents pour créer des individus enfants dans la génération futur et la mutation effectuée de légères modifications de certains individus.

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

a- Algorithme génétique: 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 (correspondant à des solutions) évoluées en même temps comme dans l'évolution

naturelle en biologie pour chacun des individus, on mesure sa faculté d'adaptation en milieu extérieur par le fitness. Les algorithmes 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 sauvent la valeur de la fonction objectif de la solution associée à l'individu.

- Le croissement: 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.

Algorithme (1): un algorithme génétique simple

1: Initialisation: générer une population initiale **P** de solutions de taille **|P| = n**

2: Répéter

3: Sélectionner: choisir deux solutions **X** et **X′** par une technique de sélection

4: Croisement: combiner les deux solutions parents **X** et **X′** pour former une solution enfant y

5: Mutation de **y** sous conditions

6: Choisir une solution individuelle y′ pour être remplacé dans la population

7: Remplacer **y′** par y dans la population

8: Jusqu'à critère d'arrêt satisfait.

b- 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 la région en quête de nourriture face à l'obstacle, le groupe des fourmis explorent les deux cotés 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 le phéromone sera la plus forte mais la phéromone étant une odeur elle s'évapore. Si peu de fourmis empruntant 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 empruntant un chemin plus long (pour 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 possibilité d'être emprunté.

Algorithme 2: Métaheuristique ACO:

1: Initialisation: Créer une population initiale de fourmis

2: Répéter

3: Pour chaque fourmi faire

4: Construire une solution par une procédure de construction à l'aide des traces de phéromones.

5: Mise à jour des traces de phéromones basées sur la qualité des solutions trouvées.

6: Fin pour

7: Jusqu'à critère d'arrêt satisfait

3-Les méthodes hybrides:

l'idée essentielle de cette hybridation consiste à exploiter pleinement la puissance de recherche de méthodes voisinage et de recombinaison des algorithmes évolutifs sur une population de solution, un tel algorithme utilise, une ou plusieurs méthodes de voisinage sur les individus de la population pendant un certain membre d'itération ou jusqu'à la découverte d'un ensemble d'optima locaux et invoque ensuite un mécanisme de recombinaison pour créer un nouveau individu comme pour les algorithmes génétiques spécialisés, la recombinaison doit impérativement être adaptée au problème traité.

Les algorithmes hybrides sont sans doute parmi les méthodes les plus puissantes malheureusement le temps de calcul nécessaire pouvant devenir prohibitif à cause des membres d'individus manipulés dans la population.

Méthode de descente:

c'est l'une des heuristiques de recherche locale les plus simples. Elle consiste à rechercher dans le voisinage de la solution courante, une solution de coût plus faible. Elle procède ainsi jusqu'à arriver à un optimum local.

A partir d'une solution trouvée par une heuristique par exemple, on peut très facilement implémenter des méthodes de descente. Ces méthodes s'articulent toutes autour d'un principe simple. Partir d'une solution existante, chercher une solution dans le voisinage et accepter cette solution si elle améliore la solution courante.

L'algorithme 1 présente le squelette d'une méthode de descente simple. A partir d'une solution initiale **x**, on choisit une solution **x′** dans le voisinage **N(x)** de **x**. Si cette solution est meilleure que **x**, **f(x′)** alors on accepte cette solution comme nouvelle solution **x** et on recommence le processus jusqu'à ce qu'il n'y ait plus aucune solution améliorante dans le voisinage de **x**.

Algorithme 1 Descente Simple:

1: initialisation: trouver une solution initiale **x**

2: répéter

3: recherche de voisinage: trouver une solution $\mathbf{x' \in N(x)}$

4: si $\mathbf{f(x') < f(x)}$ alors

5: $\mathbf{x' := x}$

6: fin si

7: jusqu'à $\mathbf{f(y) \geq f(x)}$, $\forall \mathbf{y \in N(x)}$

l'avantage principal de ces méthodes réside dans leur grande simplicité et leur rapidité. Toutefois elles comportent deux obstacles majeurs qui limitent considérablement leur efficacité:

- suivant la taille et la structure du voisinage **N(x)** considéré, la recherche de la meilleur solution voisine est un problème qui peut être aussi difficile que le problème initiale;
- 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.

Pour faire faces à ces carences, la solution la plus simple est la méthode de relance aléatoire qui consiste à générer une nouvelle configuration de départ de façon aléatoire et à recommencer une descente. Une autre solution consiste à accepter des voisins de même performance que la configuration courante. Cette approche permet à la recherche de se déplacer sur les plateaux, mais n'est pas suffisante pour ressortir de tous les optima locaux. D'autres raffinements plus élaborés sont également possibles, par exemple: l'introduction de voisinages variables [SEV04], et les techniques de réduction ou d'élargissement du voisinage[HAO 03].

Recuit simulé:

Cette classe de méthodes d'optimisation s'inspire des méthodes de simulation de Metropolis (années 50) en mécanique statistique. L'analogie historique s'inspire du recuit des métaux en métallurgie: un métal refroidi trop vite présente de nombreux défauts microscopiques, c'est l'équivalent d'un optimum local pour un problème d'optimisation combinatoire. Si on le refroidi lentement, les atomes se réarrangent, les défauts disparaissent, et le métal a alors une structure très ordonnée, équivalent à un optimum global.

L'analogie avec une méthode d'optimisation est trouvée en associant une solution à un état métal, son équilibre thermodynamique est la valeur de la fonction objectif de cette solution. Passer d'un état du métal à un autre correspond à passer d'une solution à une solution voisine.

Pour passer à une solution voisine, il faut respecter l'une des deux conditions:

- Soit le mouvement améliore la qualité de la solution précédente, **i.e.** en minimisation, la variation de coût négative ($\Delta C < 0$).
- Soit le mouvement détériore la qualité de la solution précédente et la probabilité **p** d'accepter un tel mouvement est inférieur à une valeur dépendante de la température courante **t ($p < e^{-\Delta c / t}$)**.

Le schéma de refroidissement de la température est une des parties les plus difficiles à régler dans ce cas. Ces schémas sont cruciaux pour l'obtention d'une implémentation efficace. Un refroidissement trop rapide mènerait vers un optimum local pouvant être de très mauvaise

qualité. Un refroidissement trop lent serait très coûteux en temps de calcul. Le réglage des différents paramètres (température initiale, nombre d'itérations par palier de température, décroissance de la température, …) peut donc être long et difficile. Sans être exhaustif, on rencontre habituellement trois grandes classes de schémas de refroidissement [HAO 99]:

- réduction par paliers: chaque température est maintenue égale pendant un certain nombre d'itérations, et décroît ainsi par paliers.
- Réduction continue: la température est modifiée à chaque itération.
- Réduction polynomial: la température décroît à chaque itération avec des augmentations occasionnelles.

L'algorithme 2 présente les principales caractéristiques d'un recuit simulé.

Algorithme 2 Recuit simulé:

1: initialisation: trouver une solution initiale **x**, poser une température initiale **t**

2: répéter

3: recherche de voisinage: trouver une solution $\mathbf{x' \in N(x)}$

4: déterminer $\mathbf{\Delta C = f(x') - f(x)}$

5: obtenir $\mathbf{p \sim U(0.1)}$

6: si $\mathbf{\Delta C < 0}$ ou $\mathbf{e^{-\Delta C / t} > p}$ alors

7: $\mathbf{x' := x}$

8: fin si

9: réduire la température **t** selon un schéma de refroidissement

10: jusqu'à un critère d'arrêt satisfait.

4- Comparaison entre méthodes exactes et méthodes approchées (voir la table 01)

Le tableau ci-dessous représente quelques différences entre les méthodes exactes et les méthodes approchées.

Méthode approchée	Méthode exacte
Efficacité pour les problèmes de grande taille	Efficacité pour des cas particuliers des problèmes de taille raisonnable
Espace mémoire raisonnable	Espace mémoire considérable
Durée de résolution très petite par rapport aux méthodes exactes	Durée de temps de résolution est généralement considérable pour les problèmes de grande taille
Aucune garantie d'optimalité	Garantie d'optimalité
Pratiquement simple à implanter et à comprendre	Pratiquement difficile à implanter et à comprendre.

Table (01)

INTRODUCTION:

Dans ce Chapitre, nous allon représenté les études et les solutions du problème d'ordonnancement sur une machine à **n** tâches, avec toutes les tâches **i** sont disponibles au début de l'ordonnancement **($r_i = 0$).**

En présence des périodes d'indisponibilités de la machine pour la minimisation de la somme pondérée des dates de fin des tâches $\sum w_i C_i$.

- L'ordonnancement sous contraintes des disponibilités des machines a largement été traité par plusieurs auteurs, citons le cas 1, NC//$\sum C_i$ et 1, NC // $\sum w_i C_i$ par Belouadah Bel [88] et m, NC //C_{max} par [Lee 96], [Lee 97] et [Lee 99] et plus récemment par Schmidt [SCH 00].

Et puisque il est NP- dur au sens fort, les méthodes exactes requièrent un effort calculatoire qui croit exponentiellement avec la taille du problème dans un temps, raisonnable.

- Les méthodes approchées permettent de résoudre ce type de problème en un temps raisonnable.

Dans ce chapitre on présente la métaheuristique recherche tabou; pour trouver une solution approchée du problème considéré.

Dans cette section nous avons présenté la méthode de recherche Tabou et détaillé les paramètres et les grandes étapes de la méthode ainsi les différentes techniques qui permettent son amélioration et nous avons appliqué la méthode de recherche au problème considéré ainsi que les résultats obtenus.

- Finalement une étude comparative entre les solutions avec les deux techniques de génération de voisinage.

Représentation du problème:

Le problème d'ordonnancement des tâches avec prise en compte des contraintes de disponibilité de machine se définit de la manière suivante:

- une machine qui n'est pas disponible pendant **M** périodes

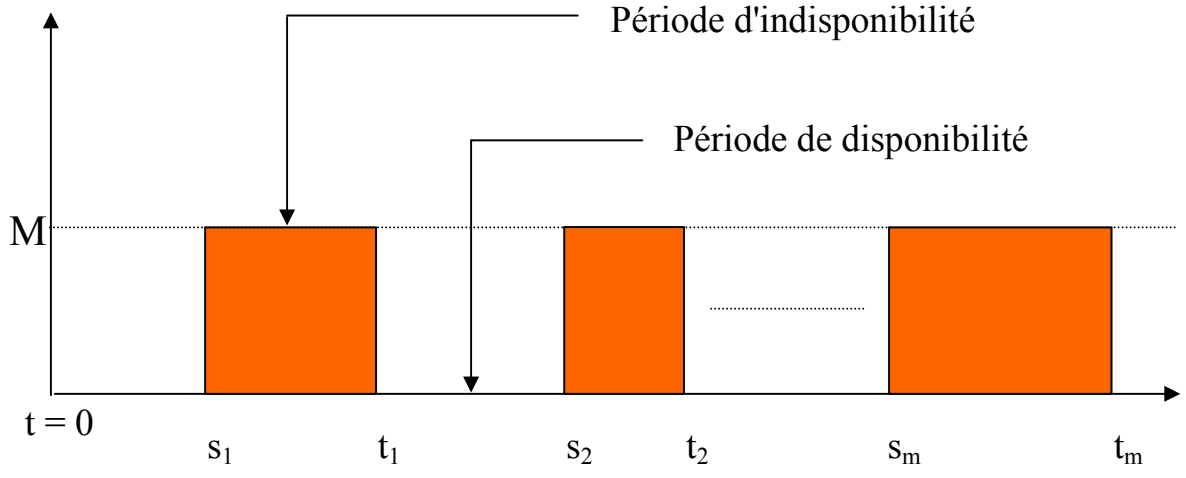
  $[s_i, t_i]$ tel que $S_i < t_i$ et $t_i < S_{i+1}$ pour i = 1, 2, 3…., M-1

  dont les dates et les durées sont fixées et connues à priori.

- La machine ne peut réaliser qu'une tâche à la fois et chaque tâche nécessite une seule machine.
- Toutes les tâches i sont disponible au début de l'ordonnancement (r_i=0).
- Les tâches sont caractérisées par:
  - Leur temps de traitement p_i.
  - Leur poids w_i.
- La collection de **n** jobs: J = (1, 2, ……, n) est indépendant où la tâche ne s'exécute qu'une seule fois.
- Les tâches à ordonnancer sont strictement non préemptives, ce qui signifie que l'exécution de toute opération ne peut être interrompue ni par une période d'indisponibilité, ni par la réalisation d'une autre tâche.

L'objectif est de déterminer la séquence d'entrée des tâches sur la machine tel que la somme pondérée des dates de fin des tâches ($\sum w_i C_i$) soit minimale.

Le problème considéré se décrit par le schéma suivant:



Recherche Taboue

La recherche tabou, est une méthaheuristique , originalement développée par Glover, 1986 et indépendamment par [Han86], [GL 097] et [GL099].

Cette méthode combine une procédure de recherche locale avec un certain nombre de règles et de mécanisme permettant à celle-ci de surmonter l'obstacle des optima locaux, tout en évitant de cycler.

Principe de base:

La méthode de la recherche tabou peut être vue comme une généralisation des méthodes d'amélioration locales, en effet, en partant une solution quel conque **x**, on se déplace vers une solution **x'** située dans le voisinage **N(x)**. donc l'algorithme explore itérativement l'espace de solution **X**.

A fin de choix le meileur voisin **x'** dans **N(x)** l'algorithme évalue la fonction objectif **(f)** en chaque point **x'**, 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 apport aux méthodes locales qui s'arrêtent des qu'il n'y a plus de voisin **x'** 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 le 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 piège dans un minimum local, Mais il induit un risque de cyclage, en effet lorsque l'algorithme a quitte un minimum quelque par acception de la dégradation de la fonction objectif, il peut revenir sur ses pas, a l'itération suivant.

- Pour régler ce problème, l'algorithme a besoin d'une mémoire pour conserver pendent un moment la trace des dernières meilleures solutions déjà visitées. Ces solution sont déclarées tabous, d'où le nom de la méthode.
- Elles sont stockée dans une liste de longueur **L** donnée, appelée liste taboue.

Une nouvelle solution n'est accepté que ci elle n'appartient pas a cette liste taboue, ce critère d'acceptation dune nouvelle solution évite le cyclage de l'algorithme durant la visite

d'un nombre de solution au moins égal à la longueur de la liste taboue, et il dirige l'exploration de la méthode vers des régions du domaine de solution non encore visitées.

- L'algorithme (1) (présenté ci-dessous) donne les principes de base de méthode tabou classique.

1- Initialisation:

χ := solution initial, $f_{min} = f(x)$, X:=*x*

TABOU := liste de solution *x*, de longueur L

TABOU := vide

2: répéter

3: générer un voisinage N(*x*) tel que $x1 \in N(X)$ et $x_i \notin$ TABOU

4: $f(x') = \min_{1 \le i \le N} [f(x_i)]$

5: ajouter (*x'*, TABOU)

6: *x*:=*x'*

7: Si $f(x) < f_{min}$

8: f_{min}:=f(*x*)

9: X_{min}:=*x*

10: fin Si

11- jusque à condition d'arrête satisfait.

La liste de TABOUE est généralement comme une liste circulaire, on élimine à chaque itération la solution taboue la plus ancienne en la remplaçant par la nouvelle solution retenue mais le codage d'une telle liste est encombrant , car il faudrait garder en mémoire tous les éléments qui définissent une solution, pour pallier cette contrainte, on remplace la liste taboue de solution interdites par une liste de (transformation interdit, en interdisant la transformation inverse d'une transformation faite récemment).

2- Algorithme générale de la méthode Taboue.

- Soit ***f*** une fonction à minimiser, l'algorithme générale de la méthode de recherche Tabou peut être résumé comme une suite: (Algorithme 2)

Algorithme (2)

Début

Etape1:

Choisir une solution initial:x_0

Poser $x^* \leftarrow x_0$

$f^* \leftarrow f(x_0)$

$K \leftarrow 0$

Liste Tabou $\leftarrow \phi$

Etape 2:

Répéter tant qu'un critère de fin n'est pas vérifié.

Etape 2.1: choisir parmi le voisinage N(x_0) de x_0, *le* meilleur voisin non Tabou meilleur x_K

Etape 2.2: poser $x_{K+1} \leftarrow meilleur\ x_k$

Etape 2.3: si $f(x_{k+1}) < f^* alors$

$f^* \leftarrow f(x_{k+1})$

$x^* \leftarrow x_{k+1}$

Etape 3.4: Mise à jour de la liste tabou

$k \leftarrow k+1$

fin

3- la liste Tabou

L'un des aspects critique d'une recherche Tabou est la nécessité d'ajuster la longueur de la liste Taboue pour parcourir efficacement l'espace des solutions.

Si la liste est trop courte, la recherche finit par explorer un optimum local de rayon légèrement supérieur, les différentes solutions explorées forment un cycle qui va se répéter indéfiniment.

A l'inverse, si la liste est trop longue, tous les mouvements peuvent devenir Tabous et sans nouveau voisins la recherche s'arrête, c'est un blocage.

La liste Tabou est généré selon la stratégie FIFO (First in First out), on élimine le plus vieux Tabou et on insère la nouvelle solution, en générale la liste Tabou doit être maintenue à une longueur minimale permettant d'éviter un cycle.

La complexité d'une approche Tabou dépend essentiellement de:

- La taille du voisinage de la solution.
- La méthode d'évaluation de chacun.

De ces voisins afin de déterminer celui qui minimise la fonction coût.

Comme le temps de cette évaluation est considérable pour un voisinage plus large, il est intéressant d'utiliser les voisinages aussi contraints que possible afin de diminuer au maximum possible le complexité de résolution en travaillant généralement sur un sous ensemble de voisinage.

4- Sélection du voisinage:

La sélection du meilleur voisin et souvent selon la politique du best fit qui permet de choisir le meilleur voisin non Tabou, ou la politique du first fit Dans ce dernier cas on sélectionne le premier voisin qui satisfait les contraintes Tabous, cette politique est souvent utilisée lorsque la taille du voisinage ne permet pas d'effectuer une évaluation complété.

4-1 Critère d'aspiration

Les interdictions engendrées par la méthode Taboue peuvent s'avérer trop fortes et restreindre l'ensemble des solutions admises à chaque itération d'une manière jugée trop brutal, le mécanisme d'aspiration, et mis en place a fin de pallier cet inconvénient.

Ce mécanisme permet de lever le statut Tabou d'une configuration sans pour autant introduire un risque de cycles dans le processus de recherche.

La fonction d'aspiration la plus simple consiste à révoquer le statut Tabou d'un mouvement si ce dernier permet d'atteindre une solution de coût inférieur à celui de la meilleur solution trouvée jusqu'à présent.

4-2 Techniques d'amélioration:

Parmi les stratégies qui ont été proposés récemment pour améliorer l'efficacité de la méthode Tabou la stratégie de l'intensification et celle de la diversification.

- L'intensification: on mémorise les meilleurs solutions rencontrées et l'on essaie d'en dégager quelques propriétés communes pour définir des régions intéressantes vers lesquelles on oriente la recherche, par exemple en rendant Tabou tous les mouvements qui font sortir de cette région.

Donc l'intensification permet de stopper périodiquement le processus normal d'exploration et intensifier l'effort de recherche dans une région qui parait prometteuse, une autre manière d'appliquer l'intensification consiste à mémoriser une liste de solution de bonne qualité à retourner vers une de ces solutions.

- La diversification: a un objectif inverse de l'intensification, elle cherche à diriger la recherche vers des zones inexplorées. Sa mise en œuvre consiste souvent à mémoriser les solutions les plus fréquemment visitées et imposer un système de pénalités, a fin de favoriser les mouvements les moins souvent utilisés.

4-3 Structure de voisinage

La connaissance de l'ensemble des voisins d'une séquence $\sigma = (J_1, J_2 .....J_{n-1}, J_n)$ permet d'en sélectionner le meilleur qui apporte une amélioration de la solution.

Parmi les techniques les plus connues on peut citer:

43-1 Voisinage par swapping

(permutation simple de deux tâches)

Cette technique est basée sur les deux stratégies suivantes.

a/ La permutation simple de tâche adjacent: ce type de voisinage consiste à générer (*n-1*) séquences obtenues par la permutation de l'ordre de traitement de chaque paire de tâches adjacent dans la séquence initiale.

b/ La permutation simple de deux tâches, cette stratégie consiste à générer $\frac{n(n-1)}{2}$ séquences obtenues par la permutation entre chaque paire de tâches (non nécessairement adjacent) dans la séquence initiale.

ILLUSTRATION

Soit: $\sigma = (J_1, J_2, J_3 .....J_n)$ séquence initiale.

Alors la permutation simple deux tâche (adjacentes)

$(J_2, J_1, J_3 \ldots\ldots\ldots\ldots\ldots J_{n-1}, J_n)$
$(J_1, J_3, J_2 \ldots\ldots\ldots\ldots\ldots J_{n-1}, J_n)$
$(J_1, J_2, J_4 \ldots\ldots\ldots\ldots\ldots J_{n-1}, J_n)$
.
.
.
$(J_1, J_2, J_3 \ldots\ldots\ldots\ldots\ldots J_n, J_{n-1})$

et la permutation simple de deux tâches (non nécessairement adjacentes).

$(J_2, J_1, J_3 \ldots\ldots\ldots\ldots\ldots J_{n-1}, J_n)$
$(J_3, J_2, J_1 \ldots\ldots\ldots\ldots\ldots J_{n-1}, J_n)$
$(J_4, J_2, J_3 \ldots\ldots\ldots\ldots\ldots J_{n-1}, J_n)$
.
.
.
.
$(J_1, J_2, J_3 \ldots\ldots\ldots\ldots\ldots J_n, J_{n-1})$

Exemple 1:

Soit une séquence σ = (1, 2, 3, 4), le voisinage N(σ) est:

N (σ) = { (2, 1, 3, 4), (3, 2, 1,4), (4, 2, 3, 1), (1,3, 2, 4), (1, 4, 3, 2), (1, 2, 4, 3)}

Le voisinage par swapping (voir tabeau 01)

Tâche i	Tâche j	Séquence obtenu
1	2	(2, 1, 3, 4)
	3	(3, 2, 1, 4)
	4	(4, 2, 3, 1)
2	3	(1,3, 2, 4)
	4	(1, 4, 3, 2)
3	4	(1, 2, 4, 3)

Tableau (1)

4-3-2Le voisinage par insertion de tâche: on choisi deux positions ***i*** et ***j*** tel que $i \neq j$ à partir de la séquence courante et déplacer la tâche de la position ***i*** et l'incérer dans la position ***j***, et on peut distinguer deux cas possibles: ***i <j*** ou ***i > j*** (cette stratégie consiste a générer $(n-1)^2$ séquences).

ILLUSTRATION

1er cas i < j

$(.....,J_i,J_k,J_1,J_m,J_j.....)$

Après le changement

$(.....,J_k,J_1,J_m,J_j,J_i.....)$

2eme cas *i* > *j*

séquence courante:

$(.....,J_j,J_k,J_1,J_m,J_i.....)$

après le chargement

$(.....,J_i,J_j,J_k,J_1,J_m.....)$

Exemple 2:

Soit une séquence σ = (1, 2, 3, 4)

N (σ) = { (2, 1, 3, 4), (2, 3,1, 4), (2, 3, 4, 1), (1,3, 2, 4), (1, 3, 4, 2), (3, 1, 2, 4), (1, 2, 4, 3), (4, 1, 2, 3), (1, 4, 2, 3)} .

Le voisinage par insertion: (voir tableau 02)

Position tâches	1	2	3	4
1	-	(2, 1, 3, 4)	(2, 3,1, 4)	(2, 3, 4, 1)
2	(2,1 , 3, 4)	-	(1, 3, 2, 4)	(1, 3, 4, 2)
3	(3, 1, 2, 4)	(1, 3, 2, 4)	-	(1, 2, 4, 3)
4	(4, 1, 2, 3)	(1, 4, 2, 3)	(1, 2, 4, 3)	-

Tableau (02)

2 Application de la méthode de Taboue au problème (P):

2-1 Choix des paramètres de la méthode.

- Le voisinage retenu pour la méthode recherché taboue, trois types de voisinage:

a- Génération aléatoire.

b- Génération par permutation de deux tâches avec un mouvement systématique.

c- Le voisinage par insertion avec un mouvement aléatoire pour passer d'un voisin à un autre.

- La stratégie de sélection du voisin utilisé est la sélection du premier voisin non Tabou dans le voisinage de la solution (séquence) courante.

- La taille de la liste Tabou est prise égale à 10 Cette valeur est suffisante et fréquemment utilisée dans la littérature car elle évite le cyclage et le blocage du processus de recherche.

- Le critère d'arrêt est choisi comme un nombre d'itérations à effectuer.
- Ce nombre est fixé à $\mathbf{n^2}$ itérations.

Algorithme TS:

```
Début
    Liste tabou ← ϕ
    i ← 1
    Sol cour ← solution initiale.
    Meil solution ← sol cour
    Tant que (i < = nb iter max) faire
        Début
            Voisinage ← détermine voisinage (sol cour)
            Meil voisin ← meilleur voisin (voisinage)
            Ajouter dans liste tabou (meil voisin)
            Sol-cour ← meil voisin
                Si (coût (sol cour)) < coût (meil sol)) alors
                Meil sol ← sol cour
                F si
                i ← i + 1
    Fin tant que
Fin
```

Description de l'algorithme TS:

- La fonction "déterminer un voisinage", retourne l'ensemble des voisins de la solution courante suivant le type de voisinage utilisé.

- La fonction "meilleur voisin" retourne le meilleur voisin non Tabou parmi les éléments de voisinage générés par la fonction " déterminer un voisinage".

- La procédure "ajoute dans liste tabou" permet la mise à jour de la liste Tabou suivant la stratégie **(FIFO**) et l'ajout du voisin choisi.

- La fonction "Coût" retourne la valeur objective d'une solution.

Exemple illustratif:

Soit le problème de 05 tâches, on considère la taille de la liste Tabou = 02:

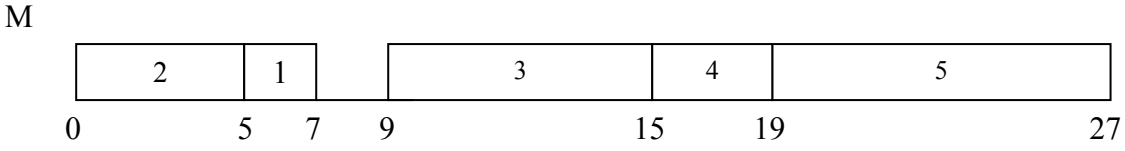
- Toutes les tâches sont disponibles au début de l'ordonnancement ($r_i = 0$)

Job	1	2	3	4	5
P_i	2	5	6	4	8
w_i	3	4	2	1	2

- Pour $t \in [7, 9]$ la machine est indisponible

On va minimiser la fonction $\sum w_i C_i$

On choisi aléatoirement la séquence: $\sigma_0 = (2,1, 3, 4, 5)$ d'où $f_0 = 144$



Initialisation:

nb itération max = 6 - le nombre d'itérations

$\sigma^* \leftarrow \sigma_0$ et $f^* \leftarrow f_0$

liste tabou = ϕ - initialement la liste tabou est vide

$i \leftarrow 1$

Itération 01: le voisinage est:

σ_1:	(1, 2, 3, 4, 5)	(2, 3, 1, 4, 5)	(2, 1, 4, 3, 5)	(2, 1, 3, 5, 4)
f_1:	137	180	141	144
	meilleur voisin			
	non tabou			

σ_0 = (1, 2, 3, 4, 5)

liste tabou ← {(1, 2, 3, 4, 5)}

i ← 2

Itération 02: le voisinage est:

σ_1 :	(2, 1, 3, 4, 5)	(1,3, 2, 4, 5)	(1, 2, 4, 3, 5)	(1, 2, 3, 5, 4)
f_1:	144	204	139	137
				meilleur voisin mais tabou

σ_0 = (1, 2, 4, 3, 5)

liste tabou: {(1, 2, 4, 3, 5)}

i ← 3

Itération 03: le voisinage est:

σ_1:	(2, 1, 4, 3, 5)	(1, 4, 2, 3, 5)	(1, 2, 3, 4, 5)	(1, 2, 4, 5, 3).
f_1:	146	164	137	143
			meilleur voisin	
			mais tabou	

σ_0 = (1, 2, 4, 5, 3)

liste Tabou {(1, 2, 4, 5, 3)}

i ← 4

Itération 04: Le voisinage est:

σ_1:	(2, 1, 4, 5, 3)	(1, 4, 2, 5, 3)	(1, 2, 5, 4, 3)	(1, 2, 4, 3, 5)
f_1:	150	168	143	139
				meilleur voisin
				non Tabou

σ_0 (1, 2, 4, 3, 5)

Liste Tabou = { (1, 2, 4, 3, 5), (1, 2, 4, 5, 3)}

i ← 5

Itération 05: Le voisinage est:

σ_1:	(2, 1, 4, 3, 5)	(1, 4, 2, 3, 5)	(1, 2, 3, 4, 5)	(1, 2, 4, 5, 3)
f_1:	146	164	137	143
			meilleur voisin	**Tabou**
			mais Tabou	

σ_0 (2, 1, 4, 3, 5)

liste Tabou ← { (2, 1, 4, 3, 5), (1, 2, 4, 3, 5)} .

i ← 6

Itération 06: le voisinage est:

σ_1:	(1, 2, 4, 3, 5)	(2, 4, 1, 3, 5)	(2, 1, 3, 4, 5)	(2, 1, 4, 5, 3)
f_1:	139	178	144	150
	meilleur voisin			
	mais Tabou			

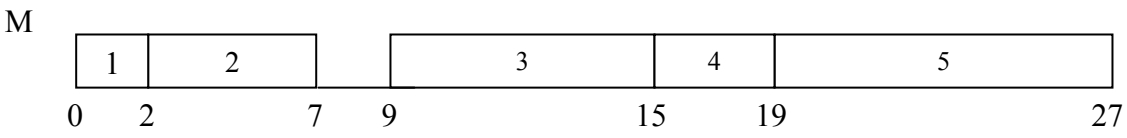
σ_0 (2, 1, 3, 4, 5)

liste Tabou ← { (2, 1, 4, 5, 3), (2, 1, 4, 3, 5)} .

i ← 7…….stop.

donc le processus de recherche s'arrête et la meilleur solution trouvée est:

σ^* = (1, 2, 3, 4, 5) et f * = $\sum_{i=1}^{5} W_i C_i$ = 137



2-2 Type de voisinage:

Le paramètre N est un élément de l'ensemble {I. block} tel que:

I: indique que le voisinage est généré par permutation de deux tâches.

Block: indique que le voisinage est généré par block

2-3 Le mouvement dans le voisinage:

Le paramètre de passage au voisin M est un élément de l'ensemble { R,S } où R dénote que Le mouvement est aléatoire par contre S indique que le passage d'un voisin à un autre est systématique.

2-4 Le choix de la séquence initiale:

Le paramètre de la séquence initiale S est un élément de l'ensemble { R, WSPT } tel que:

R: indique la génération aléatoire de la séquence initiale.

WSPT: indique que la séquence initiale est générée à l'aide de règles WSPT (weighted shorts est processing time), les tâches sont ordonnées suivant l'ordre décroissant des rapports p_i / w_i.

La complexité de cette heuristique est 0 (n log n).

Ces éléments permettent de démarrer avec une bonne séquence initiale pour rechercher les meilleurs améliorations possibles dans un certain temps raisonnable.

Périodes d'indisponibilité:

Nous avons choisi les périodes d'indisponibilité selon les manières suivantes:

- Pour une seule période d'indisponibilité:
  - début de période = $\sum p_i / n$ (tel que n est le nombre de tâches).
  - la durée de période = 10.
- Pour deux périodes d'indisponibilité:
  - début de première période = $\sum p_i / 3$
  - début de la deuxième période = $2 \sum p_i / 3$
  - la durée de période = 10
- Pour trois périodes d'indisponibilité:
  - début de première période = $\sum p_i / 4$
  - début de la deuxième période = $2 \sum p_i / 4$
  - début de la troisième période = $3 \sum p_i / 4$ (durée de la période = 10)

2-5 Expérimentation et résultats:

Nombre de tâches	f (swaping)	f (par block)	f (s. initiale)
n = 10	9615	11161	11740
	15567	16482	18917
	7434	6077	9616
n = 30	81564	93955	96735
	59598	96403	98680
	92705	10620	11620
n = 50	210370	276280	280970
	208340	269140	283690
n = 100	Inutile	1024100	1058500
	Inutile	1127300	1149000

Tableau (03) - (période d'indisponibilité = 2)

Nous avons choisi le **MATLAB** comme outil de programmation et de test .Et nous avons pris les valeurs des poids et des temps d'exécution successivement dans les lois uniformes $w \in [1,20]$ *et* $p \in [1,20]$

Dans cette partie nous illustrons une comparaison entre la méthode Taboue par SwaPing (permutation de deux tâches adjacentes) et notre proposition qui est la permutation par block dont la taille est de 5 tâches. (le choix de la séquence initiale est aléatoire)

D'après nos tests, nous avons remarqué les points suivants:

a- Si le nombre de tâches **n** est inférieur à 10, alors les méthodes exactes sont envisageables.

b- Si le nombre de tâches **n** est entre 10 et 50, la méthode Tabou par SwaPing donne meilleurs résultats que la notre.

c- Si le nombre de tâches excède 50, dans ce cas là, la méthode Tabou par swaPing dont la complexité est de **O ($n^2 \times (n-1)$)** devient inutile pratiquement alors que notre proposition dont la complexité est de **O ($n^2 \times (n-1)/5$)** c'est-à-dire Tabou par permutation de blocks est meilleur pratiquement.

Conclusion1:

Nous avons présenté une méthode de recherche Taboue pour résoudre le problème d'ordonnancement sur une seule machine comme un cas particulier avec périodes d'indisponibilité et nous avons présenté les différents types de voisinage. Nous avons, chaque fois, remarqué l'amélioration des solutions pour notre problème, les meilleurs solutions basées sur le voisinage par SwaPing.

Donc la stratégie d'amélioration par voisinage (permutation de deux tâches adjacentes) est meilleur que la permutation par blocks dans le cas où le nombre de tâches est inférieur à 50 mais pour le nombre de tâches dépassant 50 tâches la permutation par block donne de meilleurs résultats avec un temps d'exécution minimal.

3- Ordonnancement sur machines parallèles:

- **Introduction:**

Un modèle basique dans la théorie d'ordonnancement suppose que toutes les machines sont disponibles durant tout l'horizon d'ordonnancement. Ce n'est pas souvent le cas dans la pratique industrielle. Certaines machines ont des périodes d'indisponibilité qui impliquent l'impossibilité d'exécuter des travaux liés au processus de production. Généralement on planifie des périodes d'intervention préventives pour concevoir les équipements en tant que bien et assurer une meilleure disponibilité globale.

Etat de l'ART:

- Sans contraintes d'indisponibilité le problème d'ordonnancement sur une machine avec minimisation de la somme pondérée des dates de fin des tâches est résolue optimalement avec la règle WSPT.

Lee et Chen (1999) ont étudié l'ordonnancement simultané des travaux de la production et des activités de la maintenance sur machines parallèles pour minimiser la somme pondérée des dates de fin des tâches. Deux cas sont étudiés, le premier où i l y a suffisamment de ressources pour que plusieurs machines puissent être maintenues simultanément (chevauchement dans la période d'indisponibilité). Dans le second cas une seule machine peut être maintenue à la fois (chevauchement non autorisé), il est démontré que même si les travaux ont tous le même poids, le problème est NP-difficiles. Ils ont proposé une méthode de Branch an Bound basée sur l'approche de la génération de colonnes pour résoudre les deux cas. Une étude expérimentale à été faite sur des instances de taille moyenne.

- Schmidt (1984) a étudié le problème d'ordonnancement sur machines parallèles avec différents intervalles d'indisponibilité et des dates de livraison des travaux (deadlines). Il a également construit un ordonnancement préemptif admissible de complexité 0 (n.m log n) où n le nombre de tâches et m le nombre de machines.

Kacem et al (2005) ont étudié le Problème 1/N-C// $\sum w_i C_i$ et ont comparé deux méthodes exactes: La méthode Branch and bound et la méthode de programmation en nombres entier. Ils ont conclu à la bonne performance de la branch and bound qui permet de résoudre des instances de plus de 1000 tâches.

• Présentation du problème sur machines parallèles:

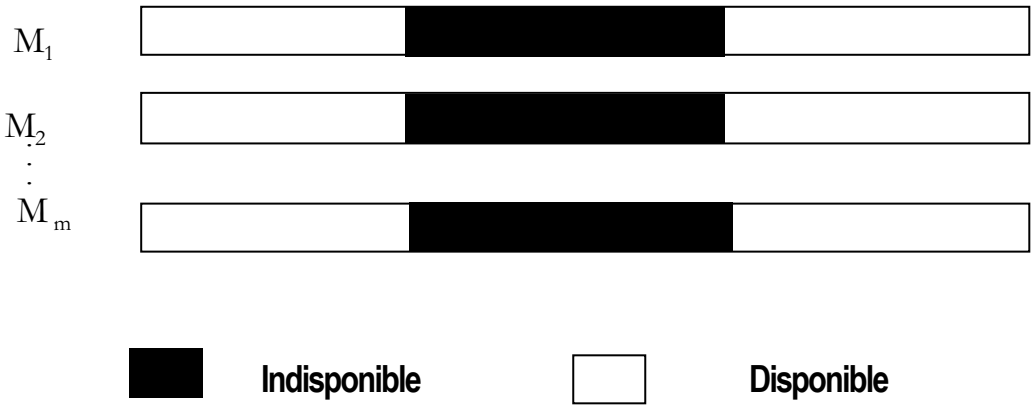
Le problème consiste à ordonnancer n jobs sur M machines parallèles identiques $\{M_1, M_2, ...M_m\}$ ($n >> m \geq 2$).

Présentant les périodes d'indisponibilité pour minimiser la somme pondérée des dates de fin des tâches: $\sum w_i C_i$

Nous supposons que les travaux $\{J_i, J_2, ...J_n\}$ sont tous disponibles à t = 0 et que leurs temps opératoires sont indépendants du choix des machines auxquelles ces travaux seront affectés. Dans le cas générique du problème, chacune des m machines peut présenter plusieurs périodes d'indisponibilité durant l'horizon d'ordonnancement et chaque tâche doit être exécutée une seule fois.

Ce problème noté P_m/NC // $\sum w_i C_i$

Affecter les travaux $\{j_i, j_2, ... j_n\}$ aux machines sur des intervalles de disponibilité



.

Proposition:

Dans chaque période de disponibilité les tâches doivent être ordonnancé selon la règle WSPT généralisée :

Algorithme SWPT Généralisé :

Etape 0 :

$j = \{1,..,n\}, j = \phi$

$f = 0$

$f^2 = 0, i = \overline{1,m}$

$C^j = 0, j = \overline{1,m}$

Etape 1 :

Tant que $j \neq \phi$ faire

Déterminer l'ensemble $h \in J$ tel que

$$\frac{p_h}{w_h} = \min\left\{\frac{p_k}{w_k}\right\}$$

Fin

Etape 2 :

Déterminer la machine $j \in \{\overline{1,n}\}$ tel que

$C^j = \min\limits_{\alpha=1,m}\{C^\alpha\}$

Si $C^j + p_h > t_{2i_0}$ alors poser $C^j = t_{2i_0} + \Delta t$

Ordonnancer la tache h sur la machine j

Poser $\sigma = \sigma \cup \{h\}, j = j/\{h\}, C^j = C^j + p_h$

$f^j = f^i + w_h C^j$

Fin

Méthode Taboue:

•Choix des Paramètres du problème sur machines parallèles identiques:

a/ Choix de la séquence initiale σ_o:

La séquence initiale est choisi d'après la règle (heuristique) WSPT, Generaliise puis générée pour l'étude d'amélioration des solutions.

b/ La taille du block:

Nous avons choisi le nombre des tâches n = 100 tâches et $L = 10$ j et Le nombre de machines est $m \in \{2, 3, 4, 5, 6, 7\}$

dl Chaque machine possède {1, 2, 3} périodes d'indisponibilité.les durée est même avec le cas particulier d'une seul machine

e/ Liste Tabou:

La taille de la liste égale à 10 est une valeur suffisante pour éviter le blokage et le cyclage.

Expérimentation et résultats :

Problème 1

Periode indi = 1						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	402760	274784	209430	170644	145315	126370
Fs	402517	273171	208566	169797	143981	125626
Fb	402760	274340	209430	170644	145078	126237

Periode indi = 2						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	782962	519184	388942	313472	262344	632440
Fs	761308	511888	388270	310771	259637	626454
Fb	763723	518894	388942	313472	261938	632440

Periode indi = 3						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	2194500	1513200	1108400	894569	750126	632440
Fs	2152800	1432908	1088400	875931	728650	626454
Fb	2186715	1469500	1108400	894569	737123	632440

Problème 2

Periode indi = 1						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	360789	244193	187067	151816	129076	112893
Fs	359530	234911	186152	151514	128473	111979
Fb	360329	244193	186780	151816	128862	112844

Periode indi = 2						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	693484	467153	347636	279472	233218	200782
Fs	687410	458370	344920	277490	231950	198954
Fb	688650	463520	347636	278261	232905	200684

Periode indi = 3						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	1955153	1341663	1016774	790386	663418	572274
Fs	1929700	1287805	983000	786578	655240	563183
Fb	1955153	1302100	992933	790386	663418	572274

Problème 3

Periode indi = 1						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	437789	299350	226543	184622	157872	136635
Fs	437709	296762	226367	184117	155955	135889
Fb	437789	298627	226543	184622	157215	136424

Periode indi = 2						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	854119	565753	422076	340798	287084	702646
Fs	830549	557458	421724	338668	282668	684846
Fb	834911	565435	422076	340798	285962	693050

Periode indi = 3						
Nombre tache =100						
machine	2	3	4	5	6	7
Fini	2391249	1649100	1243018	974909	824661	702646
Fs	2329400	1581944	1196330	957875	796080	684846
Fb	2391249	1626700	1206880	974909	814880	693050

Notre problème consiste à minimiser la somme pondérée des dates de fin des tâches sur machines parallèles identiques avec périodes d'indisponibilité. Pour étudier ce problème nous avons adopté la méthode Tabou et en utilisant les deux techniques: permutation par swaping et la deuxième la permutation par block. Pour générer les instances du test nous avons pris les valeurs des poids w $\in$ [1, 10] et les valeurs du temps d'exécution.,P$\in$ [1, 100] en basant sur la loi uniforme.

Sur les tableaux précédents nous avons illustré les résultats des 3 problèmes de 100 tâches pour chacune. Pour faire ce test nous avons utilisé un ordinateur Pentium IV de 3GHZ de vitesse et de 256 Mo de mémoire et comme outil de programmation nous avons utilisé le MATLAB. Pour chaque exemple nous avons pris 1, 2 et 3 périodes d'indisponibilité et 2, 3, ...7 machines parallèles identiques. Les tableaux nous montrent les valeurs de la fonction objective de la séquence initiale de la technique de permutation par swaping et l'autre technique c'est-à-dire la technique de permutation par block, sur ces exemples nous avons constitué le point suivant selon le matériel que nous avons utilisé et le temps d'exécution pour calculer la fonction objective, qui a une durée de 20 minutes, par la technique de permutation par swaping et de 2 minutes pour la technique de la permutation par block.

Relativement au temps d'exécution, la technique de permutation par block est meilleure par rapport à la technique de permutation par swap

Conclusion 2:

Nous avons représenté une méthode de recherche métaheuristique recherche tabou pour résoudre le problème d'ordonnancement des taches sur des machine parallèles identiques avec périodes d'indisponibilité et nous avons représenté les différents types de voisinage, nous avons remarqué à chaque fois l'amélioration des solutions pour chaque stratégie. On conclus que la permutation par block améliore la fonction objectif par apport au temps d'exécution que la permutation par swaPing, ce qui fait que cette dernière est plus rentable

Conclusion générale

Dans ce sujet nous avons abordé le problème d'ordonnancement sur des machine parallèles avec des périodes d'indisponibilité afin de minimiser la somme pondérée des dates de fin des tâches ($\sum \mathbf{w}_i \mathbf{C}_i$) et nous avons étudies un cas particulier sur une seul machine puisque le problème est rest NP-difficile.

Nous avons représenté les méthodes exacte et les méthodes approchées avec les avantages et les inconvénients de chaque méthode, puisque la majorité des problémes d'ordonnancents sur des machines paralléles NP-difficiles ainsi un algorithme polynomial pour une solution exacte est impossible d'étre établis .

La métaheristique recheche tabou en été développé pour une classe trés large pour donner des solutions approchées pour les problémes d'ordonnancements de grand taille avec instance considérable dans les machines paralléles avec des périodes d'indisponibilité.la recherche tabou est trés efficace pour les problémes de grande taille ,notre proposition pour la résolution du probléme en question est basée sur deux choix dont le premier est le voisinage par swaPing qui est plus rentable et la deuxiéme est la permutation par blocks qui améliore la fonction objectif par apport le temps ,puis le teste des résultats ,finalement la comparaison entre les deux choix proposés.

```
% Génération de la structure de données

n=100; % Nombre de taches

a=1;
b=20;
x= a + (b-a)*rand(n);

pwr = x(1:2,1:n);
%pwr = [7 2 3 8 4;2 5 6 7 1]% ********
clear x;

sequence = randperm(n);

% Génération des periodes d'indisponibilité

casindi = 2
perindi = 10;  % perindi : periode d'indisponibilté
per = sum (pwr(1,1:n)) + perindi;

switch casindi
   case 1
      debindi = per/2;  % debindi : debut de periode d'indisponiblité
      %debindi = 7 %**************
      %perindi = 2 %***************
      pperindi(1,1) = debindi;
      pperindi(2,1) = perindi;

     f = objectif(sequence,pwr,n,pperindi,1);
   case 2
      debindi = per/3;  % debindi : debut de periode d'indisponiblité
      pperindi(1,1) = debindi;
      pperindi(2,1) = perindi;

      pperindi(1,2) = 2*debindi;
      pperindi(2,2) = perindi;

     f = objectif(sequence,pwr,n,pperindi,1);
  case 1
      debindi = per/4;  % debindi : debut de periode d'indisponiblité
      pperindi(1,1) = debindi;
      pperindi(2,1) = perindi;
```

```
      pperindi(1,2) = 2*debindi;
      pperindi(2,2) = perindi;

      pperindi(1,3) = 3*debindi;
      pperindi(2,3) = perindi;
     f = objectif(sequence,pwr,n,pperindi,1);
end
```

function [seq,seqs,f] = generer_voisinage(sequence,pw,n,perindi,casindi,choix)

```
seq = voisinage (sequence,1,choix);
f = objectif(seq,pw,n,perindi,casindi);

switch choix
   case 's'
      nbr_seq = length(sequence)-1;
   case 'b'
      nbr_seq = length(sequence)/5-1;
end

for i=1:nbr_seq
    seqs (i,1:n) = voisinage (sequence,i,choix);
    seqs (i,n+1) = objectif(voisinage (sequence,i,choix),pw,n,perindi,casindi);
    if f >  objectif(voisinage (sequence,i,choix),pw,n,perindi,casindi)
      f = objectif(voisinage (sequence,i,choix),pw,n,perindi,casindi);
      seq = voisinage (sequence,i,choix);
    end
end

for i=1:nbr_seq-1
  for j = i:nbr_seq
    if seqs(i,n+1) > seqs(j,n+1)
      inter = seqs(j,1:n+1);
      seqs(j,1:n+1) = seqs(i,1:n+1);
      seqs(i,1:n+1) = inter;
    end
  end

end
```

```
function f = objetif (seq,pw,nn,perindi,casindi);
f=0;
t =0;
c =0;
switch casindi
   case 1
      tt = 0;
      for i=1:nn
         j = seq(i);
         c = pw(1,j) + t;
         if (c > perindi(1,1)) & (tt == 0)
            t = perindi(1,1) + perindi(2,1);
            c = pw(1,j) + t;
            tt = 1;
            t=c;
         else
            t = t + pw(1,j);
         end

        f = f + pw(2,j)*c;
      end
   case 2
      tt1 = 0;
      tt2 =0;
      for i=1:nn
         j = seq(i);
         c = pw(1,j) + t;
         if (c > perindi(1,1)) & (tt1 == 0)
            t = perindi(1,1) + perindi(2,1);
            c = pw(1,j) + t;
            tt1 = 1;
            t=c;
         else
            t = t + pw(1,j);
         end

         if (c > perindi(1,2)) & (tt2 == 0)
            t = perindi(1,2) + perindi(2,2);
            c = pw(1,j) + t;
            tt2 = 1;
            t=c;
         else
            t = t + pw(1,j);
```

```
        end

       f = f + pw(2,j)*c;
    end
  case 3
    tt1 = 0;
    tt2 =0;
    tt3 =0;
    for i=1:nn
       j = seq(i);
       c = pw(1,j) + t;
       if (c > perindi(1,1)) & (tt1 == 0)
          t = perindi(1,1) + perindi(2,1);
          c = pw(1,j) + t;
          tt1 = 1;
          t=c;
       else
          t = t + pw(1,j);
       end

       if (c > perindi(1,2)) & (tt2 == 0)
          t = perindi(1,2) + perindi(2,2);
          c = pw(1,j) + t;
          tt2 = 1;
          t=c;
       else
          t = t + pw(1,j);
       end

      f = f + pw(2,j)*c;

       if (c > perindi(1,3)) & (tt3 == 0)
          t = perindi(1,3) + perindi(2,3);
          c = pw(1,j) + t;
          tt3 = 1;
          t=c;
       else
          t = t + pw(1,j);
       end
         f = f + pw(2,j)*c;
    end

end % end switch
```

```
function [fopt,seqopt] = rech_tab(pw,perindi,casindi,seq,n,choix)

taille_tabou = fix(sqrt(n));

[seq,seqs,f] = generer_voisinage(seq,pw,n,perindi,casindi,choix);


fopt = f;
seqopt =seq;

seq_cour = seq;


liste_tabou(1,1:n) = seq;
liste_tabou(1,n+1) = f;

for i = 1:n*n

 [seq,seqs,f] = generer_voisinage(seq_cour,pw,n,perindi,casindi,choix);
 e = true;
 ind_seqs = 1;

 i

 if f<fopt
    fopt = f;
    seqopt = seq;
 end

 while e
   j = 1;
   sze = size(liste_tabou);
   e1 = true;
   while j <= sze(1,1) & e1
      if f == liste_tabou(j,n+1)
         e1=false;
      else
         j= j+1;
   end

   if e1 == true
      if taille_tabou == sze(1,1)
         jj =sze(1,1);
```

```
        for ii = 1:sze(1,1)-1
            liste_tabou(jj-ii+1,1:n+1) = liste_tabou(jj-ii,1:n+1);
        end
      else
        jj =sze(1,1)+1;
        for ii = 1:sze(1,1)-1
            liste_tabou(jj-ii+1,1:n+1) = liste_tabou(jj-ii,1:n+1);
        end
      end
      liste_tabou(1,1:n) = seq;
      liste_tabou(1,n+1) =f;
      e=false;
   else
      ind_seqs = ind_seqs +1;
      seq = seqs(ind_seqs,1:n);
      f= seqs(ind_seqs,n+1);
   end
 end

 seq_cour = seq;

end
end
```

function seq = voisinage(sequence,n,choix)

```
% n appartient à [0,1,2,...,N-1] , N: c'est la taille de la sequence
% initiale
switch choix
    case 's'
       inter = sequence(n);
       sequence(n) = sequence(n+1);
       sequence(n+1) = inter;
       seq = sequence;
   case 'b'
       inter = sequence(n:n+4);
       sequence(n:n+4) = sequence(n+5:n+9);
       sequence(n+5:n+9) = inter;
       seq = sequence;
end
```

Le programme :

```
% Génération de la structure de données

n=100; % Nombre de taches

a=1;
b=20;
%x= a + (b-a)*rand(n);

%pwr = x(1:2,1:n);
%pwr = [7 2 3 8 4;2 5 6 7 1]% ********
clear x;
%**********************pi/wi*************************************
**********
pw1(1,1:n) = 1:n;
pw1(2:3,1:n) = pwr;

for i =1:n-1
   for j =i+1:n
     if pw1(2,i)/pw1(3,i) > pw1(2,j)/pw1(3,j)
        v = pw1(1:3,j);
        pw1(1:3,j)=pw1(1:3,i);
        pw1(1:3,i)=v;
     end
   end
end
%************************************************************
***********

sequence = pw1(1,1:n);

% Génération des periodes d'indisponibilité
n_machine = 7;
casindi = 3
perindi = 10;  % perindi : periode d'indisponibilté
per = sum (pwr(1,1:n)) + perindi;

switch casindi
   case 1
     debindi = per/2;  % debindi : debut de periode d'indisponiblité
     %debindi = 7 %*************
     %perindi = 2 %***************
```

```
      pperindi(1,1) = debindi;
      pperindi(2,1) = perindi;

     f = objectif(sequence,pwr,n,pperindi,casindi,n_machine);
   case 2
      debindi = per/3;  % debindi : debut de periode d'indisponiblité
      pperindi(1,1) = debindi;
      pperindi(2,1) = perindi;

      pperindi(1,2) = 2*debindi;
      pperindi(2,2) = perindi;

     f = objectif(sequence,pwr,n,pperindi,casindi,n_machine);
  case 3
      debindi = per/4;  % debindi : debut de periode d'indisponiblité
      pperindi(1,1) = debindi;
      pperindi(2,1) = perindi;

      pperindi(1,2) = 2*debindi;
      pperindi(2,2) = perindi;

      pperindi(1,3) = 3*debindi;
      pperindi(2,3) = perindi;

     f = objectif(sequence,pwr,n,pperindi,casindi,n_machine);

end
```

function seq = voisinage(sequence,n,choix)

```
% n appartient à [0,1,2,...,N-1] , N: c'est la taille de la sequence
% initiale
switch choix
   case 's'
      inter = sequence(n);
      sequence(n) = sequence(n+1);
      sequence(n+1) = inter;
      seq = sequence;
  case 'b'
      inter = sequence(n:n+4);
      sequence(n:n+4) = sequence(n+5:n+9);
      sequence(n+5:n+9) = inter;
      seq = sequence;
end
```

```
function [fopt,seqopt] =
rech_tab(pw,perindi,casindi,seq,n,choix,n_machine)

taille_tabou = fix(sqrt(n));

[seq,seqs,f] = generer_voisinage(seq,pw,n,perindi,casindi,choix,n_machine);


fopt = f;
seqopt =seq;

seq_cour = seq;


liste_tabou(1,1:n) = seq;
liste_tabou(1,n+1) = f;

for i = 1:n*n
[seq,seqs,f] =
generer_voisinage(seq_cour,pw,n,perindi,casindi,choix,n_machine);
 e = true;
 ind_seqs = 1;

 i

 if f<fopt
    fopt = f;
    seqopt = seq;
 end

 while e
   j = 1;
   sze = size(liste_tabou);
   e1 = true;
   while j <= sze(1,1) & e1
      if f == liste_tabou(j,n+1)
         e1=false;
      else
         j= j+1;
   end

   if e1 == true
      if taille_tabou == sze(1,1)
```

```
            jj =sze(1,1);
            for ii = 1:sze(1,1)-1
               liste_tabou(jj-ii+1,1:n+1) = liste_tabou(jj-ii,1:n+1);
            end
         else
            jj =sze(1,1)+1;
            for ii = 1:sze(1,1)-1
               liste_tabou(jj-ii+1,1:n+1) = liste_tabou(jj-ii,1:n+1);
            end
         end
         liste_tabou(1,1:n) = seq;
         liste_tabou(1,n+1) =f;
         e=false;
      else
         ind_seqs = ind_seqs +1;
         seq = seqs(ind_seqs,1:n);
         f= seqs(ind_seqs,n+1);
      end
   end

   seq_cour = seq;




end
end
```

```
function f = objetif (seq,pw,nn,perindi,casindi,n_machine);
seq1 = seq;
f=0;
c =0;
%***********affectation parallele**************
ii =1;
j =1;
x=1;
while j<=nn

   if ii > n_machine
      ii = 1;
      x = x+1;
   end
   machine_paral(ii,x) = seq1(j);
   j=j+1;
   ii=ii+1;


end

%***********************************************
for n_m =1:n_machine
 if n_m<ii
   seq1 = machine_paral(n_m,:);
   n_par = x;
 else
   seq1 = machine_paral(n_m,1:x-1);
   n_par = x-1;
 end
 t =0;
switch casindi
   case 1
      tt = 0;
      for i=1:n_par
         j = seq1(i);
         c = pw(1,j) + t;
         if (c > perindi(1,1)) & (tt == 0)
            t = perindi(1,1) + perindi(2,1);
            c = pw(1,j) + t;
            tt = 1;
            t=c;
         else
```

```
            t = t + pw(1,j);
        end

       f = f + pw(2,j)*c;
    end
case 2
    tt1 = 0;
    tt2 =0;
    for i=1:n_par
        j = seq1(i);
        c = pw(1,j) + t;
        if (c > perindi(1,1)) & (tt1 == 0)
            t = perindi(1,1) + perindi(2,1);
            c = pw(1,j) + t;
            tt1 = 1;
            t=c;
        else
            t = t + pw(1,j);
        end

        if (c > perindi(1,2)) & (tt2 == 0)
            t = perindi(1,2) + perindi(2,2);
            c = pw(1,j) + t;
            tt2 = 1;
            t=c;
        else
            t = t + pw(1,j);
        end

       f = f + pw(2,j)*c;
    end

case 3
    tt1 = 0;
    tt2 =0;
    tt3 =0;
    for i=1:n_par
        j = seq1(i);
        c = pw(1,j) + t;
        if (c > perindi(1,1)) & (tt1 == 0)
            t = perindi(1,1) + perindi(2,1);
            c = pw(1,j) + t;
            tt1 = 1;
```

```
            t=c;
         else
            t = t + pw(1,j);
         end

         if (c > perindi(1,2)) & (tt2 == 0)
            t = perindi(1,2) + perindi(2,2);
            c = pw(1,j) + t;
            tt2 = 1;
            t=c;
         else
            t = t + pw(1,j);
         end

        f = f + pw(2,j)*c;

        if (c > perindi(1,3)) & (tt3 == 0)
            t = perindi(1,3) + perindi(2,3);
            c = pw(1,j) + t;
            tt3 = 1;
            t=c;
         else
            t = t + pw(1,j);
         end

        f = f + pw(2,j)*c;
      end


end % end switch
end
```

```
function [seq,seqs,f] =
generer_voisinage(sequence,pw,n,perindi,casindi,choix,n_machine)

seq = voisinage (sequence,1,choix);
f = objectif(seq,pw,n,perindi,casindi,n_machine);

switch choix
   case 's'
      nbr_seq = length(sequence)-1;
   case 'b'
      nbr_seq = length(sequence)/5-1;
end

for i=1:nbr_seq
    seqs (i,1:n) = voisinage (sequence,i,choix);
    seqs (i,n+1) = objectif(voisinage
(sequence,i,choix),pw,n,perindi,casindi,n_machine);
    if f >  objectif(voisinage (sequence,i,choix),pw,n,perindi,casindi,n_machine)
       f = objectif(voisinage (sequence,i,choix),pw,n,perindi,casindi,n_machine);
       seq = voisinage (sequence,i,choix);
    end
end

for i=1:nbr_seq-1
  for j = i:nbr_seq
    if seqs(i,n+1) > seqs(j,n+1)
       inter = seqs(j,1:n+1);
       seqs(j,1:n+1) = seqs(i,1:n+1);
       seqs(i,1:n+1) = inter;
    end
  end

end
```

Références bibliographiques:

[AK T00] Akturk M. S And Ozdemir D. 2000. An exact approach to menimizing total weighted tardiness with unequal release dates. HE transaction 32, 1091 – 1101

[AK T01] Akturk M. S And Ozdemir D. 2001. A new dominance rule to minimize total weighted tardiness with unequal release dates. European Journal of Oper. Res . 135 – 394 – 412.

[AR T84] Ari P. J. V. epsalainen . 1984. state dependent priority rules for scheduling. Air force office of scientific Research.

[AY 099] Ayten T. 1999 Machine Scheduling with availibility constraints, departement of Industrial Engenering, Bilkent University ANKARA.

[BAK 74] Baker. K. R. 1974 Introduction to sequencing and scheduling John Wily & sons.

[BAK 72] Baker. K. R. and share L. E. 1978. Finding an optimal sequence by dynamic programming and extention to precedence related tasks. Oper. Res 26 – 111 – 120.

[BEL 04] Belouadah. H. 2004 Single machine scheduling with periods of machine unavilibity working paper, faculté des sciences, département d'informatique Algérie.

[BEL 85] Belouadah. H. 1985 . Sheduling to minimize total cost M. Thesis. University of keele UK.

[BEL 88] Belouadah. H. 1988. Scheduling and sequencing, branch and bound based on job splitting and Lagrangian Relaxation . Ph. D Thesis. University of Southampton, UK.

[BEL 92] Belouadah. H. Posner M. E. and Potts C.N. 1992 . Scheduling with relates dates on a single machine to minimize Total weighted Completion time.

[CAR 88] Carlier J. et Chrétienne P. 1988. Problème d'ordonnancement, modélisation, complexité / algorithmes. Masson.

[CHR 05] Cherif S. et Bernard P. Christophe R. Blazewicz J. Formanowicz P. 2005 an improved aproximation algorithm for the single machine total completion time scheduling problem with availability constraints. European journal of operational research 161 – 3 – 10.

[CHE 99] Chen. T. Qi X. 1999. Scheduling the maintenance of a single machine. Journal operational research society 50 – 1071 – 1078.

[CHU 96] Chu. C, proth J. M. 1996. l'ordonnancement et ses applications. Masson. Paris.

Références bibliographiques:

[DUM 03] I.Dumitresco and T. Stutzle, 2003. Combinations of Local Search and Exact Algorithms, Evo Workshops, 211 – 223, Springer Verlag, Berlin.

[EIN 04] Single Machine Problems.
http:// www. Mathematic. Uni – osnabrueck.de/research/OR/class.

[ESQ99] P. Esquerol et P. Lopez, 1999, L'ordonnancemen, Economica

[FRE82] S.French, 1982, Sequencing and Scheduling: An Introuction to the Mathematicx of the job-shop, John wiley & Sons, New York.

[**GEN 84]** P. C. Geng, 1984, Multi-Product Lot-Sizing Problems on single machine, Ph.D Thesis, University of Waterloo, Ontario (Canada).

[GLO 97] Fred Glover and Manuel Laguna, 1997, Tabou Search, Kluwer Academic Publishers.

[GLO 99] Fred Glove and Said Hanafi, 1999, Tabou Search and Finite Convergence, in Discrete Applied Mathematics.

[GRI 05] A. Grigoriev, M. Holthuijsen and J. V. De Klundertm 2005, Basic Scheduling Problems with Raw Materials Constraints, Naval Research Logistics.

[HAN 01] P. Hansen and N. Mladenovic, 2001, Rechrche à voisinage variable, Les cahiers De GERARD.

[HAO99] Jin-Kao Hao et al., 1999, Métaheuristique pour l'optimisation Combinatoire et l'affectation sous contraintes, Revue d'Intelligence Artificielle vol. N° 1999

[HAO 03] M. Haouari and T. Ladhari, 2003, A Branch – and – Bound – Based Local Search Method for the Flow Shop Probem, JORS, 54, 1076 – 1084.

[HAR 81] Hariri A. M., 1981, Scheduling: Using Branch and Bound Techniques. Ph.D Thesis, University of Keele.

[HER 03] A. Hertz and M. Widmer, 2003, Guidelines for the use of Metaheuristics in Combinatorial Optimization, EJOR? 151 – 247 – 252.

[JAN 00] A. Janiak, C. N. Potts and T. Tautenhahn, 2000, Single Machine Scheduling with Nonlinear Ressource Dependencies of Release Times, AMS Journal.

[KAR 98] David Karger, Cliffs and Joel Wein 1998.

Références bibliographiques:

[RAB 99] Rachmadugu R. V. and morton T.E 1983 . Myopic Heuristics for the weighted tardiness problem on identical parallel machines. The robotics institute, Carnegie Mellon University, Pittsburgh.

[RAC 83] Rachmadugu R. V. and Morton T. E. 1983. Myopie Heuristics for the weighted tardiness problem on identical parallel machines. The robotics institute, Carnegie-Mellon University, Pitsburgh.

[SCB 90] Schmidt G. 2000. Scheduling with limited machine availability. European Journal of Operational Research 121, 1 – 15.