N°d'ordre :

UNIVERSITE*MOHAMED BOUDIAF* DE M'SILA
FACULTE DES SCIENCES ET SCIENCES DE L'INGENIEUR
DEPARTEMENT D'INFORMATIQUE

MEMOIRE

Présenté pour l'obtention du diplôme de :

MAGISTER

Spécialité : Informatique
Option : Informatique industrielle

Par : MEHENNI Tahar

Thème :

UTILISATION DES METAHEURISTIQUES POUR RESOUDRE UN PROBLEME D'ORDONNANCEMENT SUR MACHINE A CONTRAINTE DE RESSOURCE NON RENOUVELABLE

Soutenu publiquement le : 23/09/2006 devant le jury composé de :

Mr Abdelkader KHELLADI, Professeur, USTHB	**PRESIDENT**
Mr Hocine BELOUADAH, Maître de conférence, Université de M'sila	**RAPPORTEUR**
Mr Meziane AIDER, Professeur, USTHB	**EXAMINATEUR**
Mr Rachid OUAFI, Maître de conférence, USTHB	**EXAMINATEUR**
Mr Adel MOUSSAOUI, Chargé de cours, Université de M'sila	**INVITÉ**

REMERCIEMENTS

Je tiens à témoigner ma reconnaissance à mon encadreur Monsieur Hocine BELOUADAH, Maître de conférence à l'université de M'sila, pour tous ses conseils précieux, ses critiques constructives et ses encouragements. Qu'il trouve ici l'expression de ma profonde gratitude.

Mes sincères remerciements vont à Monsieur A. KHELLADI, Professeur à USTHB, Monsieur M. AIDER, Professeur à USTHB, ainsi que Monsieur R. OUAFI, Maître de conférence à USTHB, de l'honneur qu'ils m'ont fait en acceptant d'être membres du jury de ce mémoire.

Mes remerciements vont également à Monsieur A. MOUSSAOUI, Chargé de cours à l'université de M'sila pour son acceptation d'être notre invité au jury de ce mémoire.

Je remercie gracieusement Monsieur Brahim BOUDRAH, Maître de conférence à l'université de M'sila, pour son support, son ardeur et son dévouement si remarquables durant nos études de post-graduation.

Je dois également remercier vivement Monsieur I.H. OSMAN, Professeur à l'université américaine de Beirut, de l'attention qu'il nous a accordé et des conseils qu'il nous a donné durant les journées du RAMA5.

Je tiens à remercier tous mes enseignants de la première année de Magister, ainsi que tout mes collègues, et tout le personnel du département Informatique, notamment Monsieur Rosafi, Monsieur Guesmi, Monsieur Fares et Monsieur Hamza, pour leur aide si précieux.

Evidemment, je ne peux oublier ma famille, principalement ma femme qui m'a supporté durant ces deux longues années de post-graduation.

DEDICACES

A l'âme de mon père ;
A ma mère, qui m'a réellement tout donné ;
A ma femme, qui m'a vraiment aidé ;
A Soheib et Lina, mes chers enfants.

Je dédis ce mémoire

RÉSUMÉ

L'objet de cette thèse est la résolution des problèmes d'ordonnancement sur machine en présence d'une ressource non renouvelable (ou consommable). Chaque tâche nécessite une quantité de ressource pour qu'elle soit exécutée. Le problème consiste à trouver un ordonnancement qui minimise la somme pondérée des dates de fin d'exécution des tâches en satisfaisant la contrainte de ressource. Nous proposons deux métaheuristiques afin de résoudre ce problème. La première consiste à utiliser une recherche locale basée sur un voisinage obtenu par les techniques de séparation et évaluation. Deux bornes inférieures sont proposées pour permettre une évaluation de cette méthode. La seconde métaheuristique considérée est la recherche tabou qui utilise une liste taboue dynamique ainsi qu'une nouvelle technique de diversification basée sur l'utilisation d'une mémoire à long terme contenant les premières meilleures solutions non retenues lors de la recherche. Deux voisinages sont proposés pour évaluer l'efficacité de cette méthode. Enfin, nous effectuons une étude comparative des deux métaheuristiques utilisées afin de montrer leur efficacité dans la résolution de notre problème.

Mots clefs : ressource, métaheuristique, voisinage, branch-and-bound, relaxation lagrangienne, swapping, insertion, recherche locale, k-first strategy, recherche tabou.

ABSTRACT

This thesis addresses the single machine scheduling problem with non-renewable (consumable) resource constraints where the objective is to minimize the sum of weighted completion times. A job can be processed if a sufficient quantity of resource is available. We developed two metaheuristics to resolve this problem. The first one is a local search method, which is based on a truncated branch-and-bound algorithm to find the neighbourhood. Two lower bounds are proposed in order to evaluate the best one embedded in the branch-and-bound search. The second metaheuristic is a tabu search using a dynamic tabu list and a new diversification strategy, called "k-first strategy", which is based on a list of the *k*-first best solutions not chosen in the neighbourhood search. The tabu search is evaluated with two structures of the neighbourhood. Finally we present computational experiments in order to show the efficiency of the proposed metaheuristics.

Keywords : resource, metaheuristic, neighboorhood, branch-and-bound, Lagrangian relaxation, swapping, insertion, local search, k-first strategy, tabu search

---

ملـخـص

تتمثل هذه المذكرة في طرح وحل مشكل جدولة إنجاز أشغال على ماكنة مفردة مع وجود مصدر غير قابل للتجديد (مستهلك) من أجل تصغير دالة إنهاء المواقيت المثـقـلة، حيث لا يمكن لأي شغل أن ينجز إلا بوجود كمية كافية متاحة من المصدر. من أجل حل هذا المشكل استخدمنا اثنين من الطرق التقريبية التي تدخل ضمن ما يسمى بالطرق الموجهة. الأولى هي طريقة البحث المحلي على أساس التفريع والحصر لإيجاد الجوار، وقد تم اقتراح برنامجين لحساب الحد الأدنى، يرتكز الأول على قاعدة سميث أما الثاني فيستعمل إرخاء لاقرانج، كما تم إجراء تقويم عليهما لأجل اختيار الأفضل الذي يعطي أحسن الحلول. أما الطريقة الثانية المقترحة فتسمى طريقة البحث مع العزل حيث تم استعمال قائمة ديناميكية للحلول المعزولة، كما اقترحنا تقنية جديدة للتنويع ترتكز على إيجاد قائمة بأولى الحلول القريبة من الحل المتوفر والتي لم يتم اختبارها في عملية البحث. وقد تم تقويم هذه الطريقة باستعمال نوعين من الجوار هما الجوار بالتبديل والجوار بالإدراج. وأخيرا تم إجراء اختبارات على الطريقتين المقترحتين لأجل اختيار التي تعطي أحسن الحلول للمشكل قيد الدراسة.

كلمات مفتاحية: مصدر، طرق موجهة ، جوار، التفريع والحصر، إرخاء لاقرانج، التبديل، الإدراج، بحث محلي، استراتيجية قائمة الأوائل، البحث بالعزل.

SOMMAIRE

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

INTRODUCTION

Introduction

Dans la terminologie de la recherche opérationnelle, un problème d'ordonnancement désigne tout problème dans lequel l'objectif est l'allocation de ressources au cours du temps, de façon à réaliser un ensemble d'activités. Des typologies plus précises complètent cette définition très générale [CAR88]. De fait, la notion de problème d'ordonnancement regroupe une grande variété de problèmes différents selon la nature des tâches à réaliser (morcelables ou non, répétitives, etc.), les caractéristiques des ressources (renouvelables, non renouvelables, etc.), les contraintes portant sur les tâches (dates de disponibilité, précédence, etc.) et les critères à optimiser (encours, retard, etc.).

Dans ce mémoire, nous nous sommes intéressés à un problème d'ordonnancement sur machine d'un ensemble de tâches dont l'exécution nécessite la présence d'une ressource non renouvelable. Chaque tâche, pour être exécutée, a besoin de consommer une quantité déterminée de cette ressource. Comme cette ressource est non renouvelable, la quantité totale consommée par les tâches ne doit pas dépasser la quantité totale disponible. L'objectif est la minimisation de la somme pondérée des dates de fin d'exécution des tâches.

L'intérêt de l'étude de ce problème réside sur son existence dans la réalité. Dans un atelier, la ressource peut être la matière première, un lubrifiant ou du carburant, les tâches sont les travaux à réaliser sur la machine. Dans le domaine de l'informatique, la ressource peut être la mémoire, les tâches sont dans ce cas des programmes qui s'exécutent sur un processeur. Dans les systèmes embarqués (satellite, téléphone portable, micro-ordinateur portable, etc.), la ressource peut être l'énergie stockée dans une batterie, les tâches sont les différents programmes qui s'exécutent sur le processeur du système.

Ce problème fait partie des problèmes d'optimisation combinatoire pour lesquels, dans la majorité des cas, il est très difficile de trouver la solution optimale. La seule méthode connue pour résoudre ce problème de manière exacte serait de faire une énumération de toutes les solutions possibles!. Ainsi, dans ces conditions, il est nécessaire de trouver un mode de solution qui fournisse une

solution de bonne qualité dans un laps de temps raisonnable : ce que font les méthodes heuristiques.

Bien que l'obtention d'une solution optimale ne soit pas garantie, l'utilisation d'une méthode heuristique offre de multiples avantages par rapport à une méthode exacte :

- La recherche d'une solution optimale peut être totalement inappropriée dans certaines applications pratiques en raison de la dimension du problème, de la dynamique qui caractérise l'environnement de travail, du manque de précision dans la récolte des données, de la difficulté de formuler les contraintes en termes explicites ou de la présence d'objectifs contradictoires.
- Lorsqu'elle est applicable, une méthode exacte est souvent beaucoup plus lente qu'une méthode heuristique, ce qui engendre des coûts informatiques supplémentaires et des difficultés au niveau du temps de réponse.
- Les principes de recherche qui sont à la base d'une méthode heuristique sont en général plus accessibles aux utilisateurs non expérimentés. Le manque de transparence qui caractérise certaines méthodes exactes nécessite une intervention régulière de la part d'un spécialiste voire même du concepteur de la méthode.
- Une méthode heuristique peut être facilement adaptée ou combinée avec d'autres types de méthodes. Cette flexibilité augmente considérablement les possibilités d'utilisation des méthodes heuristiques.
- Malgré l'évolution permanente des calculateurs et les progrès fulgurants de l'informatique, les méthodes exactes rencontrent généralement des difficultés face aux problèmes de taille importante, ceci est dû au temps de calcul nécessaire pour trouver une solution, qui augmente exponentiellement avec la taille du problème.

Pour toutes ces raisons, nous avons abordé le problème avec des méthodes heuristiques, plus précisément les métaheuristiques, qui sont des méthodes heuristiques appliquées à plusieurs catégories de problèmes d'optimisation combinatoire.

Nous avons utilisé deux métaheuristiques pour résoudre notre problème, la recherche locale basée sur un voisinage variable obtenu par la méthode de séparation et évaluation, et la recherche tabou qui utilise des mécanismes de mémoire et de diversification.

Pour pouvoir exposer nos recherches dans ce domaine, il nous semble important de commencer par introduire les problèmes d'ordonnancement ainsi que les notations utilisées dans la suite de ce mémoire. Le chapitre 1 a pour objet la présentation des problèmes d'ordonnancement, leur classification et leur complexité.

Ensuite, nous aborderons au chapitre 2, les différentes méthodes de résolution des problèmes d'ordonnancement. On y trouve en détail la méthode par séparation et évaluation qui sera exploitée dans la suite du mémoire, ainsi que les métaheuristiques les plus connues dans le domaine, notamment la recherche tabou et la recherche locale.

Le dernier chapitre de ce mémoire (chapitre 3), sera consacré au problème d'ordonnancement que nous avons étudié. Dans un premier temps, nous décrivons ce problème et nous passons en revue, d'une façon non exhaustive, quelques articles relatifs à la classe de ces problèmes, avant de proposer une formulation mathématique englobant les contraintes et le critère d'optimisation du problème sujet d'étude.

Le problème général est par la suite abordé, en proposant deux métaheuristiques pour le résoudre. Nous décrivons, en premier lieu, une méthode de descente basée sur un voisinage obtenu par construction d'un arbre à l'aide des techniques de séparation et évaluation. Afin de mieux évaluer cette méthode, deux bornes inférieures sont proposées, l'une obtenue par relaxation lagrangienne tandis que l'autre est une application directe d'une règle de priorité.

Ensuite, nous décrivons une deuxième métaheuristique, connue sous le nom de recherche tabou. Les paramètres de cette méthode sont définis, ainsi que deux stratégies de voisinage pour mieux comparer les résultats des tests effectués.

A la fin de ce chapitre, nous évaluons les deux métaheuristiques utilisées, en présentant et commentant les résultats de nos tests.

Finalement, ce mémoire est clôturé par une conclusion générale, où nous présentons nos suggestions pour une recherche future.

CHAPITRE 1

Généralités sur les problèmes d'ordonnancement

1 Introduction

Ordonnancer un ensemble de tâches revient à programmer leur exécution dans le temps en leur allouant les ressources requises et en fixant leur date de début, dans le respect de l'objectif fixé [CAR88]. D'après cette définition, on remarque que dans un problème d'ordonnancement interviennent deux notions fondamentales : les tâches et les ressources. Une ressource est un moyen, technique ou humain, dont la disponibilité limitée ou non est connue a priori. Une tâche est un travail élémentaire dont la réalisation nécessite un certain nombre d'unités de temps (sa durée) et d'unités de chaque ressource [SAD02].

Les problèmes d'ordonnancement sont très variés. On peut les rencontrer dans de très nombreux domaines : les systèmes industriels de production (activités des ateliers en gestion de production et problèmes de logistique), les systèmes informatiques (les tâches sont les programmes et les ressources sont les processeurs, la mémoire...), les systèmes administratifs (gestion du personnel, emplois du temps,…), les systèmes de transport, la construction, etc. (cf. *Tableau* 1.1).

Exemples de domaines d'application		
Projets	Problèmes d'affectation, de transport, d'habillage d'horaires, de tournées	Emplois du temps
Organisation des prises de vue d'un satellite		Systèmes distribués et systèmes embarqués
Organisation des visites des familles au centre de médecine préventive		Ateliers de production, lignes d'assemblage
	Problèmes de découpe	

***Tableau* 1.1** : Exemples de domaines d'applications

Dans ce chapitre, nous allons nous intéresser à la présentation des différents problèmes d'ordonnancement et de leur complexité. Pour cela nous expliquons d'abord les notations utilisées afin de nous initier au vocabulaire employé. Ensuite, nous décrivons les éléments fondamentaux d'un problème d'ordonnancement à savoir : les tâches et les ressources. Puis, nous présenterons les critères les plus utilisés ce qui nous permettra de reconnaître les principaux problèmes étudiés. Enfin, nous terminerons avec un aperçu sur la complexité de ces problèmes.

2 Notations générales

Soient $I = \{1,2,\cdots,n\}$ l'ensemble des n tâches à ordonnancer et $M = \{1,2,\cdots,m\}$ l'ensemble des m machines. Le tableau suivant résume les différentes notations utilisées dans les problèmes d'ordonnancement.

M	Nombre de machines
N	Nombre de tâches
I	Indice de tâche
J	Indice de machine
$p_{i,j}$	Durée de la tâche i sur la machine j
r_i	Date de disponibilité de la tâche i (date minimale de début d'exécution)
d_i	Date d'achèvement souhaitée de la tâche i
w_i	Poids de la tâche i
s_i	Date de début d'exécution de la tâche i
C_i	Date de fin d'exécution de la tâche i
F_i	Temps de présence dans le système de la tâche i : $F_i = C_i - r_i$
L_i	Retard algébrique de la tâche i : $L_i = C_i - d_i$
T_i	Retard vrai de la tâche i : $T_i = \max(C_i - d_i, 0)$
U_i	Indicateur de retard de la tâche i : $U_i = 1$, si $T_i > 0$, $U_i = 0$, sinon

***Tableau* 1.2** : Notations utilisées

Les données d'un problème d'ordonnancement sont: les tâches et leurs caractéristiques, les contraintes potentielles, les ressources et la fonction économique. Souvent une tâche i ne peut commencer son exécution avant une date

de disponibilité notée r_i et doit être achevée avant une date échue d_i. Les dates réelles de début et de fin d'exécution de la tâche *i* sont respectivement s_i et C_i.

3 Les tâches

Une tâche est une entité élémentaire de travail localisée dans le temps par une date de début et une date de fin d'exécution et qui consomme des ressources avec des quantités déterminées. Un coût (ou poids) est attribué à une tâche pour estimer sa priorité, son degré d'urgence, ou son coût d'immobilisation dans le système.

Une tâche qui peut être exécutée par morceaux est appelée tâche morcelable, le problème constitué de tâches morcelables est appelé problème préemptif. Si une tâche, une fois qu'elle démarre l'exécution, ne peut pas être interrompue, on dit que le problème est non préemptif [WAL96].

4 Les ressources

Une ressource est un moyen technique ou humain, destiné à être utilisé pour la réalisation d'une tâche et disponible en quantité limitée [ESQ99]. Plusieurs types de ressources sont à distinguer. Pour plus d'informations sur les ressources, on peut consulter [ESQ99], [KOL97], [WAL96] et [CAR88].

4-1 Les ressources renouvelables

Une ressource est dite renouvelable, si après avoir été utilisée par une tâche ou allouée à une tâche, elle redevient disponible pour les autres tâches en même quantité. La quantité disponible est renouvelée d'une tâche à une autre, elle peut être constante, comme elle peut varier d'une tâche à une autre. Exemples de ressources renouvelables : les machines, les hommes, l'équipement, les processeurs, les fichiers…

On distingue deux types de ressources renouvelables :

- les ressources disjonctives (ou non partageables) : qui ne peuvent exécuter qu'une tâche à la fois (machine, robot) ;
- les ressources cumulatives (ou partageables) : qui peuvent être utilisées par plusieurs tâches simultanément (équipe d'ouvriers).

4-2 Les ressources non renouvelables :

Une ressource est non renouvelable (ou consommable), si après avoir été utilisée par une tâche, elle n'est plus disponible pour les autres tâches. La consommation globale en cours du temps est limitée, on dit que la ressource est épuisée en l'utilisant. Exemples de ressources consommables : le capital (ou budget), le carburant, l'énergie dans une batterie, la matière première,…

Les ressources non renouvelables sont généralement produites dans un système indépendant de la machine sur laquelle les tâches sollicitant cette ressource sont exécutées. Toutefois, certains problèmes peuvent exister où certaines tâches produisent des ressources qui peuvent être consommées plus tard par d'autres tâches.

Les ressources non renouvelables peuvent également être stockées dans un ou plusieurs dépôts (entrepôts, magasins ou warehouses) ayant une capacité de stockage. Les ressources peuvent éventuellement avoir un stock initial dans un ou plusieurs dépôts.

Durant la consommation des ressources, la condition générale qui doit toujours être satisfaite est que la quantité totale de la ressource demandée par les tâches ne doit pas dépasser la quantité totale disponible.

La disponibilité des ressources peut être en quantité constante au démarrage de l'exécution des tâches, comme elle peut varier linéairement en fonction du temps, ou même d'une façon non linéaire. Les ressources peuvent aussi être indisponibles pendant un certain moment, dû à des interruptions, des préparations, des pannes ou d'une rupture de stock.

Enfin, les tâches peuvent consommer les ressources en quantités fixes ou variables pendant leur exécution. Elles peuvent utiliser une quantité fixe en fonction du temps (ex. 100DA pour chaque heure exécutée de la tâche). Elles peuvent également utiliser une quantité fixe pendant la durée totale de la tâche (ex. utiliser 500DA au début d'exécution de la tâche).

4-3 Les ressources doublement contraintes :

Une ressource est dite doublement contrainte lorsque son utilisation instantanée et sa consommation globale sont toutes deux limitées. Le financement d'un projet peut avoir une limitation sur chaque étape du projet ainsi que pour tout le projet. Une ressource doublement contrainte est généralement représentée par une ressource renouvelable et une ressource non renouvelable.

5 Les contraintes et les critères d'optimisation

Les contraintes et les critères sont définis durant la formulation du problème. Les contraintes définissent l'admissibilité d'un ordonnancement, tandis que les critères définissent l'optimalité d'un ordonnancement.

Les contraintes et les critères du problème ne doivent concerner que les tâches, leur localisation temporelle, et les moyens nécessaires à leur réalisation.

Les contraintes peuvent être sous différentes formes. Les contraintes temporelles constituent des impératifs liés aux tâches (délais de livraisons, dates de disponibilité, contraintes de précédence). Les contraintes de ressources sont liées à l'utilisation et à la disponibilité des ressources.

Trouver un ordonnancement admissible revient à déterminer une séquence de passage des opérations sur les machines respectant les contraintes du problème. Les facteurs importants dans l'évaluation d'un problème d'ordonnancement sont l'utilisation efficace des ressources, le délai global et le respect des contraintes du problème. Les variables intervenant le plus souvent dans l'expression de la fonction économique sont :

- La date C_i de fin d'exécution de la tâche i ; les retards $L_i = C_i - d_i$ et $T_i = \max(C_i - d_i, 0)$ de la tâche i
- L'indicateur de retard de la tâche i : $U_i = 1$, si $T_i > 0$, $U_i = 0$, sinon

Les critères, encore appelés fonctions économiques ou fonctions objectif, les plus utilisés font intervenir la durée totale, les retards et le coût des stocks des encours.

Minimiser la durée totale : La durée totale de l'ordonnancement notée C_{max} (ou makespan) est égale à la date d'achèvement de la tâche la plus tardive : $C_{max} = \max(C_i)$. Minimiser la durée totale, revient à minimiser C_{max} c'est-à-dire $\min(\max(C_i))$.

Minimiser les retards : On peut rencontrer plusieurs problèmes dans lesquels il faut respecter les délais d_i. Les critères retenus sont les retards vrais $T_{max} = \max(T_i)$ et les retards algébriques $L_{max} = \max(L_i)$. Parfois on retient le nombre de tâches en retard $\sum_{i=1}^{n} U_i$.

Minimiser les encours : Les encours sont déterminés par le temps de présence des tâches dans le système : $F_i = C_i - r_i$. Le critère essentiel est de minimiser la somme de ces temps $\sum_{i=1}^{n} F_i$. Lorsqu'on tient compte des coûts de présence des tâches dans le système ou de leur importance, on attache un poids w_i au temps de présence, le critère devient $\sum_{i=1}^{n} w_i F_i$. Puisque les dates de disponibilité r_i sont fixées (constantes), on retient seulement le critère $\sum_{i=1}^{n} C_i$ (somme de la date de fin d'exécution) et le critère $\sum_{i=1}^{n} w_i C_i$ (somme pondérée des dates de fin d'exécution).

6 Les problèmes d'ordonnancement

Les problèmes d'ordonnancement sont très différents d'un système à l'autre, et il n'existe pas de méthode universelle permettant de résoudre efficacement tous les cas.

Une classification peut exister selon le nombre des machines et l'ordre d'utilisation des machines pour réaliser un travail (par exemple fabrication d'un produit qui dépend de la nature de l'atelier). Un système (exemple atelier) se définit par le nombre de machines qu'il contient et par son type. Les différents types possibles sont les suivants :

- **Une machine :** chaque tâche est constituée d'une seule opération.
- **Machines parallèles :** elles remplissent toutes les mêmes fonctions. Selon leur vitesse d'exécution, on distingue : les machines identiques où la vitesse d'exécution est la même pour toutes les machines et toutes tâches, et les machines uniformes où chaque machine a une vitesse d'exécution propre et constante et la vitesse d'exécution est la même pour toutes les tâches d'une même machine.
- **Machines dédiées :** elles sont spécialisées à l'exécution de certaines opérations. Dans cette catégorie, chaque tâche est constituée de plusieurs opérations. En fonction du mode de passage des opérations sur les différentes machines, trois ateliers spécialisés sont différenciés, à savoir :

  - ***Flow shop*** **:** c'est un atelier dans lequel le cheminement des tâches est unique : les n tâches utilisent les m machines dans l'ordre 1,2, ..., m ;

  - ***Job shop*** **:** les n tâches doivent être exécutées sur les m machines, sous des hypothèses identiques à celles du flow shop, la seule différence est que les séquences opératoires relatives aux différentes tâches peuvent être distinctes et sont propres à chaque tâche ;

  - ***Open shop*** **:** c'est un modèle d'atelier où l'ordre d'exécution des tâches n'est pas fixé.

Il existe une très grande variété de problèmes d'ordonnancement. Pour leur identification et leur classification nous adoptons la notation constituée de trois champs $\alpha|\beta|\gamma$.

Le premier champ est constitué de deux éléments : $\alpha=\alpha_1\alpha_2$ et décrit l'environnement des machines utilisées :

- le paramètre α_1 décrit le type des machines utilisées $\alpha_1 \in \{1, P, Q, R, F, J, O\}$ avec :

$\alpha_1 = 1$: une machine ;
$\alpha_1 = P$: machines parallèles identiques ;
$\alpha_1 = Q$: machines parallèles uniformes ;
$\alpha_1 = R$: machines parallèles indépendantes ;
$\alpha_1 = F$: Flow shop;
$\alpha_1 = J$: Job shop;
$\alpha_1 = O$: Open shop.

- le paramètre α_2 indique le nombre de machines utilisées.

Le deuxième champs $\beta=\beta_1\beta_2\beta_3\beta_4\beta_5\beta_6\beta_7$ décrit les caractéristiques des tâches et des machines. Il indique si des éléments spécifiques sont à prendre en compte tels que des dates de disponibilité, des précédences entre tâches,...

le paramètre β_1 indique la possibilité de préemption ou non des tâches.
le paramètre β_2 indique la présence ou non de ressources auxiliaires $\beta_2 \in \{Ø, res\}$ avec :

$\beta_2 = Ø$: indique qu'il n'y a pas de ressources auxiliaires;
$\beta_2 = res$: il existe des ressources auxiliaires.

le paramètre β_3 indique la présence ou non de contraintes de précédence.
le paramètre β_4 décrit les dates de disponibilité des tâches
le paramètre β_5 décrit les durées d'exécution des tâches $\beta_5 \in \{Ø, p_i=p, \underline{p} \le p_i \le \overline{p}\}$ avec :

$\beta_5 = Ø$: les durées d'exécution sont arbitraires et sont propres à chaque tâche ;
$\beta_5 = (p_i = p)$: toutes les tâches ont une durée d'exécution égale à p ;
$\beta_5 = (\underline{p} \le p_i \le \overline{p})$: toutes les tâches ont une durée d'exécution comprise entre $\underline{p}$ *et* $\overline{p}$.

le paramètre β_6 décrit les dates de sortie au plus tard des tâches.
le paramètre β_7 indique si un temps d'attente entre les différentes opérations d'une tâche est autorisé.

Le dernier champ γ indique le critère d'optimisation, il peut donc prendre de nombreuses valeurs et peut être une combinaison entre plusieurs critères : ΣC_i (somme des dates de fin d'exécution des tâches), C_{max} (makespan)…

L'utilisation de la notation composée des trois champs α|β|γ permet d'identifier les problèmes d'ordonnancement. Ainsi, $1|res|\Sigma w_i C_i$ représente le problème consistant à ordonnancer n tâches sur une machine en présence d'une ressource auxiliaire, en vue de minimiser la somme pondérée des dates de fin d'exécution des tâches (cf. chapitre 3).

7 Notions de complexité

7-1 Introduction

L'expérience montre que certains problèmes sont plus faciles que d'autres à résoudre. Une théorie de la complexité a été développée et permet mathématiquement de classer les problèmes faciles et difficiles en deux classes : les classes P et NP. Nous exposons dans la section qui suit, les grands principes de la théorie de la complexité. Le lecteur intéressé par de plus amples informations pourra consulter différents ouvrages complètement ou partiellement dédiés à la complexité dont les livres de Sakarovitch [SAK84], Carlier [CAR88] ou encore French [FRE82].

On suppose que, pour chaque problème que l'on veut résoudre, l'on dispose d'une mesure de la taille du problème. Par exemple, on utilisera comme mesure le nombre de tâches lorsqu'il s'agit d'un problème d'ordonnancement de n tâches. Le nombre d'opérations élémentaires effectuées par un algorithme est donc une fonction qui dépend de n. Cette fonction s'appelle la complexité [REB99].

Déterminer la complexité d'un algorithme donné n'est pas toujours si simple. Déterminer la complexité d'un problème apparaît moins ambitieux, puisqu'il s'agit de déterminer quelle est la plus faible complexité d'un algorithme de résolution, parmi tous les algorithmes que nous pouvons imaginer - et bien sûr tous ceux que nous n'imaginons même pas [RAP02].

7-2 Les classes de problèmes

Définition 1 : La complexité d'un problème est la complexité de son meilleur algorithme connu.

On peut ainsi classer les problèmes suivant leur complexité. Cependant, on distingue deux familles de problèmes : les problèmes qui ont une complexité polynomiale, et les autres, ceux dont la complexité est exponentielle [REB99].

Définition 2 : On appelle algorithme polynomial, un algorithme dont le nombre d'opérations est un polynôme de n; par exemple $O(n^a)$ avec a une constante.

Les algorithmes polynomiaux sont considérés comme « efficaces » en ce sens que leur coût de calcul augmente de manière raisonnable avec la taille du problème [REB99]. Par contre, les algorithmes non polynomiaux seront considérés comme « inefficaces » car leur temps d'exécution augmente de façon spectaculaire avec la taille de l'instance traitée.

La classe des problèmes NP (NP pour Non déterministe Polynomial) est la classe des problèmes de décision pouvant être résolus par un algorithme polynomial non déterministe. Notons qu'un problème de décision est un problème dont la réponse est « oui » ou « non » [SCH95]. Parmi la classe des problèmes NP, on distingue deux grandes sous-classes : la classe des problèmes polynomiaux (la classe P) et la classe des problèmes NP-complets.

La classe P est la classe des problèmes résolubles par un algorithme polynomial déterministe. C'est la classe des problèmes les plus faciles de NP. Malheureusement, la classe P des problèmes que l'on peut résoudre en temps polynomial est assez limitée.

La définition de la classe NP-complet est basée sur la notion de « réduction polynomiale » [RAP02]. Un problème de décision L_1 est dit réductible à un autre problème de décision L_2 (noté $L_1 \propto L_2$) s'il existe une fonction polynomiale f qui transforme chaque énoncé I_1 de L_1 en un énoncé $f(I_1)$ de L_2 de sorte que la réponse

au problème L_1 pour l'instance I_1 est « *oui* » si et seulement si la réponse au problème L_2 pour l'instance $I_2 = f(I_1)$ est « *oui* ». Un problème Q est dit NP-complet, s'il appartient à la classe NP et si tout problème C connu pour être NP-complet se réduit à Q : C $\propto$ Q [SAD02].

La classe NP-complet est caractérisée par deux propriétés importantes :
- il n'existe pas d'algorithme polynomial connu pour la résolution d'un problème de NP-complet;
- si on pouvait résoudre un problème de NP-complet en temps polynomial, on obtiendrait automatiquement des algorithmes polynomiaux pour tous les problèmes de NP-complet [SAD02].

Quand un problème est NP-complet, il n'est pas raisonnable d'espérer construire un algorithme polynomial le résolvant. Mais dans certains cas, on peut construire des algorithmes très efficaces appelés algorithmes pseudo polynomiaux, le problème est dit NP-complet au sens faible. Dans le cas contraire, le problème est dit NP-complet au sens fort [SAD02].

7-3 Les 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 $x \in I$, S(x) représente l'ensemble des solutions réalisables pour x.
- Une fonction m à valeur entière définie sur tous les couples $x \in I$ et $y \in$ S(x). Cette mesure m représente la fonction objectif d'une solution y pour 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, $m^*(x) = \min\{m(x,y) \mid y \in S(x)\}$ et $y^*(x)$ est une solution vérifiant $m(x, y^*(x)) = m^*(x)$ [RAP02].

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 un entier K. La question est : existe-t-il une solution y vérifiant $m(x, y) \leq K$ pour le problème de `min` (resp. $m(x, y) \geq K$ pour `max`).

Les problèmes d'optimisation ne sont pas des problèmes de décision, et ne peuvent donc appartenir à NP, a fortiori à NP-complet. La notion de réduction entre ces problèmes s'impose pour définir intuitivement qu'un problème Π est au moins aussi difficile qu'un problème 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 [RAP02]:

Définition 3 : Un problème Π est dit NP-difficile ou NP-Dur, ssi pour tout problème $\Pi_0 \in$ NP, $\Pi_0 \propto \Pi$.

Pour un problème d'optimisation Π, on a clairement $\Pi_D \propto \Pi_C$. Pour cette raison on utilise en pratique la propriété suivante [RAP02]:

Propriété 1 : 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-difficiles, car ils sont des problèmes d'optimisation. Toutefois, il existe quelques cas particuliers pour lesquels on a trouvé des algorithmes polynomiaux, comme : 1| r_i, *pmtn*|ΣC_i résolu par Baker en 1974, et 1| *prec*, *pmtn,* $p_i = p$, r_i|ΣC_i résolu par Baptiste et al. en 2004 [EIN04]

Dans le domaine de l'optimisation, l'appartenance à la classe des problèmes NP-difficiles ne signifie pas du tout la fin d'étude d'un problème, mais plutôt son début. Le chapitre suivant donnera un aperçu sur les méthodes de résolution de ce type de problème.

CHAPITRE 2

Méthodes de résolution des problèmes d'ordonnancement

1 Introduction

Nous avons vu précédemment qu'il existe des problèmes d'ordonnancement de complexités différentes. Les problèmes appartenant à la classe P ont des algorithmes polynomiaux permettant de les résoudre.

Ces algorithmes sont spécifiques au problème à résoudre et nous ne les citerons pas ici. Pour les problèmes appartenant à la classe NP, l'existence d'algorithmes polynomiaux semble peu réaliste. Ainsi, différentes méthodes de résolution (méthodes exactes ou heuristiques), sont largement utilisées pour appréhender les problèmes NP-difficiles.

Nous exposons dans ce chapitre, les grandes lignes des méthodes les plus connues classées en trois catégories : les méthodes exactes, les heuristiques constructives et les métaheuristiques. Le lecteur intéressé par de plus amples détails pourra se reporter aux références données, notamment [TAI01], [CHE04], [SEV04], [HAO99], [OSM96], [SIL02], [SIA03], [WID01], [PIN01] et [KAR98].

2 Les méthodes exactes

Dans cette section, il est question de trois types de méthodes exactes : la programmation dynamique, la méthode de séparation et évaluation et la résolution à partir d'une modélisation analytique. Comme nous le verrons, le temps de calcul de ces méthodes est exponentiel ce qui explique qu'elles ne sont utilisables que sur des problèmes de petites tailles.

2-1 Modélisation analytique et résolution

La modélisation analytique d'un problème permet, non seulement de mettre en évidence l'objectif et les différentes contraintes du problème, mais également, parfois de le résoudre. L'idéal est d'obtenir un programme linéaire dont les variables sont réelles. Dans ce cas, il existe bon nombre de solveurs pouvant le résoudre. Dès que le problème comporte des coûts fixes, ou des décisions

nécessitant l'utilisation de variables entières, les modèles deviennent plus difficiles à résoudre.

Néanmoins, il est parfois surprenant de voir qu'un problème particulier de taille intéressante peut être appréhendé et résolu par la programmation mathématique. Il est donc justifié de commencer à étudier un problème en proposant une ou plusieurs modélisations analytiques. De plus, cette démarche a été simplifiée car il existe aujourd'hui plusieurs langages de modélisation, tels que AMPL, GAMS, LINGO, MPL, permettant d'écrire les programmes linéaires de façon formelle, proche de l'écriture mathématique. Ces langages de modélisation peuvent soit générer des fichiers lisibles par des solveurs, tels que CPLEX et XPRESS, soit être directement couplés à ces solveurs. Malheureusement, tous les problèmes ne peuvent être résolus par cette approche car la résolution de programmes linéaires mixtes, par exemple, demande souvent beaucoup trop de temps de calcul.

2-2 La programmation dynamique

Introduite par Bellman dans les années 50 [CAR88], la programmation dynamique décompose un problème ayant une dimension n en n problèmes de dimension 1. Le système est alors constitué de n étapes que l'on résout séquentiellement, le passage d'une étape à une autre se faisant à partir des lois d'évolution du système et d'une décision [CAR88].

Le principe d'optimalité de Bellman est basé sur l'existence d'une équation récursive permettant de décrire la valeur optimale du critère à une étape en fonction de sa valeur à l'étape précédente. Ainsi, pour appliquer la programmation dynamique à un problème combinatoire, le calcul du critère pour un sous-ensemble de taille k nécessite la connaissance de ce critère pour chaque sous-ensemble de taille k-1 et porte le nombre de sous-ensembles considérés à 2^n (où n est le nombre d'éléments considérés dans le problème).

La programmation dynamique est donc de complexité exponentielle. Pourtant pour les problèmes NP-difficiles au sens faible, c'est-à-dire pour lesquels il existe des algorithmes de résolution pseudo-polynomiaux, il est souvent possible de construire un algorithme de programmation dynamique pseudo-polynomial, pouvant alors être utilisé pour des problèmes de taille raisonnable.

La programmation dynamique travaille sur une formulation récursive du problème, pour laquelle on fait de la tabulation des résultats intermédiaires.

Prenons par exemple, le calcul de la suite de Fibonacci. Plutôt que de recalculer *fib(n-1)* et *fib(n-2)* pour obtenir *fib(n*), ce qui conduit à une complexité exponentielle, le calcul par programmation dynamique commence du cas de base en stockant *fib(0)* et *fib(1*), utilise la formule de récurrence du bas vers le haut, *mémorise* tous les résultats intermédiaires, jusqu'à obtenir le résultat désiré. Cette technique nécessite un stockage compact des résultats (pour des raisons évidentes de taille mémoire) [LAB98].

En 1962, Held et Carp ont appliqué les principes de la programmation dynamique pour résoudre le problème d'ordonnancement 1| |$\Sigma f_i(C_i)$ [FRE82]. Les équations récursives suggérées par Held et Carp sont de la forme :

$$f^*(J) = \min_{i \in J} \left\{ f^*(J - \{i\}) + f_i\left(\sum_{i \in J} p_i\right) \right\}$$

avec $J \subseteq I = \{1,2,\cdots,n\}$ un ensemble arbitraire de tâches et *f*(J)* le coût minimal d'ordonnancement des tâches de *J* sur la période $\left[0, \sum_{i \in J} p_i\right]$. Le coût *f*(J)* est initialisé par *f**(∅)=0.

2-3 La méthode de séparation et évaluation

2-3-1 Description de la méthode

La méthode de séparation et évaluation (Branch-and-Bound ou B&B) est basée sur une énumération implicite et intelligente de l'ensemble des solutions réalisables. Pour cela, la séparation consiste à décomposer le problème initial en plusieurs sous-problèmes qui sont à leur tour décomposables. Ce processus peut se visualiser sous forme d'un arbre d'énumération généré dynamiquement, qui contient initialement le nœud racine représentant le problème d'origine. Pour chaque sous-problème (noeud de l'arbre), la procédure d'évaluation calcule (dans le cas d'un problème de minimisation) une borne inférieure de la solution obtenue à partir de ce sous-problème. Au préalable, une borne supérieure de la solution optimale a été calculée et est utilisée pour éviter l'exploration de noeuds dont la valeur de la borne inférieure est supérieure à la valeur de la borne supérieure. Cette borne supérieure est réactualisée lorsqu'une solution réalisable de valeur inférieure est atteinte.

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

2-3-2 L'arbre de recherche

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

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

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

Un algorithme B&B pour un problème de minimisation est composé des éléments suivants [CLA99] :

1- Une fonction qui calcule pour chaque nœud la meilleure borne inférieure de ce nœud.
2- Une stratégie pour sélectionner le nœud adéquat pour la recherche de la solution.
3- Une règle de branchement appliquée au nœud, en le subdivisant en plusieurs sous-ensembles de nœuds à visiter.

2-3-3 Calcul de la borne inférieure

La procédure qui calcule la borne inférieure est l'élément clé de n'importe quel algorithme B&B, car une qualité médiocre de la borne inférieure ne peut pas être compensée par de bons choix de la stratégie de recherche ou de la règle de branchement [CLA99].

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

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

Les techniques de relaxation sont souvent utilisées dans le calcul d'une borne inférieure. Appliquées à un problème de minimisation, elles fournissent une évaluation par défaut de l'optimum, en relâchant les contraintes les plus difficiles à satisfaire, c'est-à-dire en les supprimant, ou en les prenant partiellement en compte [DEM03]. Voici dans ce qui suit, les principales techniques de relaxation.

- ***Relaxation des contraintes :*** Une technique simple de relaxation consiste à ignorer certaines contraintes du problème. On obtient alors un problème dont la solution optimale est plus facile à calculer. Par exemple, certains problèmes utilisent l'autorisation de la préemption comme une relaxation de la contrainte de non préemption des tâches.

- ***Relaxation lagrangienne :*** La relaxation lagrangienne s'articule sur l'idée de relâcher les contraintes difficiles, non pas en les supprimant totalement, mais en les prenant en compte dans la fonction objectif de sorte qu'elles pénalisent la valeur des solutions qui les violent [DEM03].

  Soit le problème de minimisation suivant :

$$\begin{aligned} & \textbf{Min } f(x) \\ \text{s.c.} \quad & g_k(x) \leq 0 \qquad 1 \leq k \leq n \end{aligned}$$

Si on suppose que les contraintes $g_k(x) \leq 0$ sont difficiles, la relaxation lagrangienne consiste à les intégrer dans la fonction objectif, à l'aide des multiplicateurs de Lagrange. Le problème relaxé sera écrit :

$$\textbf{Min } L(\lambda,x) \quad \text{avec} \quad L(\lambda,x) = f(x) + \sum_{k=1}^{n} \lambda_k g_k(x) \quad \text{et} \quad \lambda_k \geq 0$$

où λ_k sont appelés les multiplicateurs lagrangiens.

La valeur optimale $v(\lambda)$ du problème relaxé est une borne inférieure du problème initial. Comme nous voulons que cette borne soit la plus proche de l'optimum, autrement dit, la plus grande possible, nous chercherons les multiplicateurs λ qui fournissent la valeur maximale de $v(\lambda)$. Nous aboutissons au problème de maximisation, appelé le dual lagrangien :

$$\begin{aligned} & \textbf{Max } v(\lambda) \\ \text{s.c.} \quad & \lambda \geq 0 \end{aligned}$$

Ayant employé la relaxation lagrangienne pour notre problème, nous reviendrons sur la résolution du problème lagrangien au chapitre 3.

2-3-4 Stratégies de recherche

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

a) Sélectionner le nœud dont la borne inférieure est la plus petite, avec l'idée que ce nœud aura le plus de chance de contenir une solution optimale ;
b) Sélectionner le nœud le plus récemment créé, cette stratégie correspond à l'exploration dite « profondeur d'abord » ;
c) Sélectionner le nœud le plus anciennement créé, cette stratégie correspond à l'exploration dite « largeur d'abord ».

Dans la pratique, on utilise souvent un mélange des stratégies a et b. On utilise la stratégie de profondeur d'abord pour descendre dans l'arbre, dès qu'on trouve une solution, on se branche sur le nœud qui possède la plus petite borne inférieure [SAK84].

2-3-5 Règles de branchement

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

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

Toutefois, il existe des heuristiques qui, une fois appliquées, déterminent à chaque étape, les tâches qui doivent être affectées au début, à la fin, directement après ou directement avant une tâche [HAR81], [POT85].

2-3-6 Calcul de la borne supérieure

La borne supérieure est un élément essentiel de l'algorithme B&B. Calculée initialement par une heuristique appropriée, la borne supérieure est toujours comparée avec les bornes inférieures trouvées au niveau des nœuds. Dans le cas où

une borne inférieure d'un nœud est supérieure à la borne supérieure, ce nœud est élagué. Une fois arrivé à un nœud feuille, la borne supérieure sera remplacée par la valeur de la solution calculée au niveau de la feuille.

2-3-7 Efficacité de l'algorithme B&B

Différentes techniques sont utilisées pour rendre l'algorithme B&B plus efficace. On peut en citer quelques unes :

- ✓ **Choix d'une stratégie de recherche :** La stratégie « profondeur d'abord » mariée avec le choix du nœud de plus petite borne inférieure, est utilisée en pratique car elle nécessite peu d'espace mémoire [SAK84]. Par contre, la stratégie « largeur d'abord » trouve une solution optimale plus rapidement que la stratégie « profondeur d'abord » [FRE82]. Il est donc primordial de faire un bon choix de la stratégie de recherche, qui dépend généralement de la taille du problème et des ressources informatiques disponibles.

- ✓ **Choix de la borne inférieure :** Une borne inférieure doit être proche de l'optimum. Le choix de la borne inférieure dépend pleinement du problème à résoudre et de la technique utilisée (relaxation, heuristiques…). Les bonnes bornes sont celles qui éliminent un grand nombre de nœuds le plus rapidement possible, ce qui réduit considérablement l'arbre de recherche.

- ✓ **Règles de dominance :** Les règles de dominance sont des conditions posées sur un nœud, pour pouvoir l'éliminer si elles sont vérifiées (on dit que ce nœud est dominé), et ce avant de calculer sa borne inférieure. Les règles de faisabilité sont appliquées en premier aux nœuds, pour décider de l'admissibilité de la séquence traversant ce nœud, évidemment, le nœud est éliminé si la séquence n'est pas admissible. Les règles de dominance sont utiles pour réduire au mieux l'arbre de recherche, en éliminant les nœuds dominés.

3 Les méthodes heuristiques constructives

Pour les problèmes de grandes tailles, les méthodes exactes ne sont pas envisageables de par leur temps de calcul. Il est dans ce cas possible d'utiliser des méthodes approximatives qui donnent des solutions certes sous-optimales, mais obtenues rapidement. Ces solutions peuvent ensuite servir de solution initiale pour les méthodes amélioratrices (voir section 4).

En pratique, l'objectif n'est pas d'obtenir un optimum absolu, mais seulement une bonne solution et la garantie de l'inexistence d'une solution sensiblement meilleure. Pour atteindre cet objectif au bout d'un temps de calcul raisonnable, il est nécessaire d'avoir recours à des méthodes appelées "heuristiques".

Un grand nombre d'heuristiques, qui produisent des solutions proches de l'optimum, ont été développées pour les problèmes d'optimisation combinatoire difficiles. La plupart d'entre elles sont conçues spécifiquement pour un type de problème donné. D'autres, au contraire, désormais appelées "métaheuristiques", sont capables de s'adapter à différents types de problèmes, combinatoires ou même continus.

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

Le type de recherche qui est à la base d'une méthode constructive est représenté dans la figure 2.1. L'idée consiste à diminuer la taille du problème à chaque étape, ce qui revient à se restreindre à un sous-ensemble X^k inclus dans X toujours plus petit.

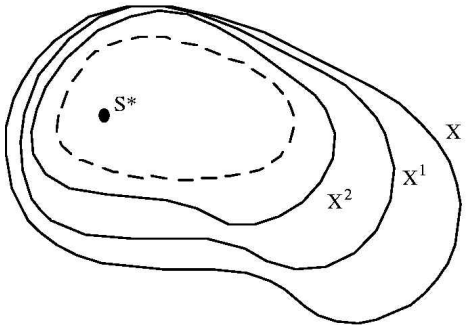

Figure 2.1. Exploration de l'espace X par approche constructive

Parmi les méthodes constructives nous en exposons deux types :

- Les algorithmes de liste : Ces algorithmes consistent, dans une première phase, à calculer une liste qui donnera l'ordre de prise en compte des éléments. Cette liste construite a priori, à partir d'un critère bien défini, n'est pas remise en cause au cours de l'ordonnancement. La deuxième phase de l'algorithme se réduit à considérer les tâches dans l'ordre de la liste pour construire l'ordonnancement.

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

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

- Parmi les règles statiques, citons les critères *SPT (shortest processing time*), et *SWPT* (*shortest weighted processing time*) appelées aussi règles de Smith [RIN76].
- Les règles dynamiques tiennent compte de l'ordonnancement construit jusqu'à présent. On peut, par exemple, chercher à placer les tâches qui génèrent le moins de temps mort dans l'ordonnancement.

Les méthodes constructives se distinguent par leur rapidité et leur grande simplicité. On obtient en effet très rapidement une solution admissible pour un problème donné sans avoir recours à des techniques hautement sophistiquées. Le principal défaut de ces méthodes réside malheureusement dans la qualité des solutions obtenues. Le fait de vouloir opérer à tout prix le meilleur choix à chaque étape est une stratégie dont les effets peuvent être catastrophiques à long terme.

La performance de telles méthodes est généralement calculée par le rapport entre la valeur de la solution calculée par l'heuristique et la valeur de la solution optimale : $R_A(I) = A(I)/OPT(I)$ pour le pire des cas. D'autres analyses de performances peuvent être menées. Par exemple, l'analyse en moyenne regarde le comportement moyen de l'heuristique en calculant $R_A(I)$, non pas seulement pour le pire des cas, mais également pour la moyenne de plusieurs instances. De plus, lorsque la solution optimale n'est pas calculable (parce que les problèmes sont de trop grande taille, par exemple), il est également possible d'étudier expérimentalement le comportement de l'heuristique en comparant ses performances soit avec les performances d'autres heuristiques, soit avec des bornes inférieures de la solution optimale.

4 Les métaheuristiques

Depuis toujours, les chercheurs ont tenté de résoudre les problèmes NP-difficiles le plus efficacement possible. Pendant longtemps, la recherche s'est orientée vers la proposition d'algorithmes exacts pour des cas particuliers polynomiaux. Ensuite, l'apparition des heuristiques a permis de trouver des solutions en général de bonne qualité pour résoudre les problèmes. En même temps, les méthodes de type "séparation et évaluation" ont aidé à résoudre des problèmes de manière optimale, mais souvent pour des instances de petite taille.

Un grand nombre d'heuristiques, qui produisent des solutions proches de l'optimum, ont été développées pour les problèmes d'optimisation combinatoire difficiles. La plupart d'entre elles sont conçues spécifiquement pour un type de problème donné. D'autres, au contraire, souvent appelées "métaheuristiques", sont

capables de s'adapter à différents types de problèmes, combinatoires ou même continus.

Une métaheuristique est constituée d'un ensemble de concepts fondamentaux (par exemple, la liste taboue et les mécanismes d'intensification et de diversification pour la métaheuristique tabou), qui permettent d'aider à la conception de méthodes heuristiques pour un problème d'optimisation. Ainsi les métaheuristiques sont adaptables et applicables à une large classe de problèmes.

Lorsque les premières métaheuristiques apparaissent, beaucoup de chercheurs se sont lancés dans l'utilisation de ces méthodes. Cela a conduit à une avancée importante pour la résolution pratique de nombreux problèmes. Cela a aussi créé un engouement pour le développement même de ces méthodes. Il existe des équipes entières qui ne travaillent qu'au développement des métaheuristiques. Il faut aussi reconnaître que c'est un formidable outil pour la résolution efficace des problèmes posés.

Les métaheuristiques sont représentées essentiellement par les méthodes de recherche locale comme le recuit simulé et la recherche tabou, et les algorithmes évolutifs comme les algorithmes génétiques et les colonies de fourmis. Grâce à ces métaheuristiques, on peut proposer aujourd'hui des solutions approchées pour des problèmes d'optimisation classiques de plus grande taille et pour de très nombreuses applications qu'il était impossible de traiter auparavant. On constate, depuis ces dernières années, que l'intérêt porté aux métaheuristiques augmente continuellement en recherche opérationnelle et en intelligence artificielle [HAO99].

4-1 Les méthodes de recherche locale

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

Le passage d'une solution admissible à une autre se fait sur la base d'un ensemble de modifications élémentaires qu'il s'agit de définir de cas en cas. Une solution x' est obtenue à partir de x en appliquant une modification élémentaire. Le voisinage $N(x)$ d'une solution $x \in X$ est défini comme l'ensemble des solutions admissibles atteignables depuis x en effectuant une modification élémentaire. Un tel processus d'exploration est interrompu lorsqu'un ou plusieurs critères d'arrêt sont satisfaits.

Le fonctionnement d'une méthode de recherche locale est illustré de manière générale dans la figure 2.2. Les passages successifs d'une solution à une solution voisine définissent un chemin au travers de l'espace des solutions admissibles. La modélisation d'un problème d'optimisation et le choix du voisinage doivent être effectués de telle sorte qu'il existe au moins un "chemin" entre chaque solution $x \in X$ et une solution optimale x^*. En effet, l'existence de tels chemins permet à la méthode de recherche locale d'atteindre une solution optimale à partir de n'importe quelle solution admissible.

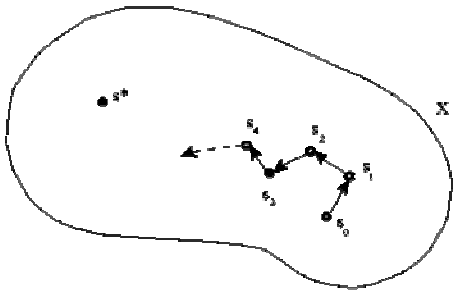

Figure 2.2. Exploration de l'espace X par une approche de recherche locale

L'utilisation de ces heuristiques itératives suppose que l'on puisse définir pour toute solution x, un voisinage de solution, $N(x)$, contenant les solutions voisines (proches dans un certain sens). En général, le voisinage d'une solution est généré en appliquant, plusieurs fois et de façon différente, à cette solution, une petite transformation (échanges, par exemple). A chaque solution x, nous associons le coût de cette solution, $f(x)$.

4-1-1 Définition du voisinage

Toutes les approches de recherche locale utilisent la notion de voisinage. Un aspect fondamental de ces approches est donc la détermination de ce voisinage.

Déterminer le voisinage consiste à caractériser tous ses éléments. Le voisinage est souvent représenté par une fonction N qui, à un point x, associe un ensemble de points N(x). Il existe une infinité de manières de choisir N, il faut adapter ce choix au problème, c'est-à-dire choisir la meilleure fonction N selon le problème considéré.

Définition [BAR03] **:** Soit X l'espace de recherche d'un problème. Une fonction de voisinage N est une association $N : X \rightarrow 2^X$, définissant, pour chaque point $x \in X$, un ensemble $N(x) \subseteq X$ de points "proches" de x.

Un autre aspect important est la taille du voisinage. En effet, certaines fonctions N peuvent calculer un ensemble si grand qu'il est impossible de le traiter efficacement avec un ordinateur (dépassement de la taille mémoire par exemple). Il convient alors soit de changer la fonction N, soit, pour un voisinage complètement différent, soit, et c'est souvent la solution retenue, pour restreindre le voisinage à un ensemble contenant moins d'éléments.

Il est clair que plusieurs points x seront visités pendant la recherche. Certaines méthodes comme le recuit simulé ne calculent pas tout le voisinage mais uniquement certains points. D'autres, comme la recherche Tabou, préconisent l'examen de tous les voisins pour réaliser un mouvement. Dans ce dernier cas de figure, il est important que le voisinage N(x) soit calculé rapidement.

L'évaluation du ou des voisins est un autre aspect important de ces méthodes. En effet, le choix de la prochaine configuration dépend en grande partie de l'évaluation des voisins. Plus le voisinage est important et plus les phases d'évaluations sont longues. Il est donc impératif que l'évaluation d'un candidat soit très rapide. Le plus souvent les méthodes de recherche locale utilisent des fonctions d'évaluation dites incrémentales. Ces fonctions se basent sur le changement local de x pour calculer de manière incrémentale la nouvelle

évaluation. En effet, elles permettent, si un ordre de parcours du voisinage est respecté, d'évaluer chaque voisin très rapidement

4-1-2 La 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 2.1 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, ($f(x') < f(x)$) 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 2.1 Descente Simple

1: *initialisation* : trouver une solution initiale x
2: répéter
3: *recherche de voisinage* : trouver une solution $x' \in N(x)$
4: si $f(x') < f(x)$ alors
5: $x' := x$
6: fin si
7: jusqu'à $f(y) \geq f(x), \forall y \in N(x)$

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 meilleure solution voisine est un problème qui peut être aussi difficile que le problème initial;

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

Pour faire face 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 [HAO03].

4-1-3 Le recuit simulé

Cette classe de méthodes d'optimisation s'inspire des méthodes de simulation de Metropolis (années 50) en mécanique statique. 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 refroidit 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 du 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 est négative ($\Delta C < 0$),
- soit le mouvement détériore la qualité de la solution précédente et la probabilité p d'accepter un tel mouvement est inférieure à une valeur dépendant 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 [HAO99] :

- 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 2.2 présente les principales caractéristiques d'un recuit simulé.

Algorithme 2.2 Recuit simulé

1: *initialisation :* trouver une solution initiale x, poser une température initiale t
2: répéter
3: *recherche de voisinage :* trouver une solution $x' \in N(x)$
4: déterminer $\Delta C = f(x') - f(x)$
5: obtenir $p \sim U(0,1)$
6: si $\Delta C < 0$ ou $e^{-\Delta C/ t} > p$ alors
7: $x' := x$
8: fin si
9: réduire la température t selon un schéma de refroidissement
10: jusqu'à un critère d'arrêt satisfait.

4-1-4 La recherche tabou

La recherche tabou est une métaheuristique originalement développée par Glover, 1986 et indépendamment par Hansen, 1986 [GLO97] et [GLO99]. Elle est basée sur des idées simples, mais elle est néanmoins très efficace. Cette méthode combine une procédure de recherche locale avec un certain nombre de règles et de mécanismes permettant à celle-ci de surmonter l'obstacle des optima locaux, tout en évitant de cycler. Elle a été appliquée avec succès pour résoudre de nombreux problèmes difficiles d'optimisation combinatoire : problèmes de routage de véhicule, problèmes d'ordonnancement, problèmes de coloration de graphes, etc.

4-1-4-1 Principe de base

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

Afin de choisir le meilleur 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 rapport aux méthodes locales, qui s'arrêtent dès 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 plus mauvais que la solution d'où l'on vient. Ce critère autorisant les dégradations de la fonction objectif évite à l'algorithme d'être piégé dans un minimum local. Mais il induit un risque de cyclage. En effet, lorsque l'algorithme a quitté un minimum quelconque par acceptation de la dégradation de la fonction objectif, il peut revenir sur ses pas, à l'itération suivante.

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

sont stockées dans une liste de longueur L donnée, appelée liste taboue. Une nouvelle solution n'est acceptée que si elle n'appartient pas à cette liste taboue. Ce critère d'acceptation d'une nouvelle solution évite le cyclage de l'algorithme, durant la visite d'un nombre de solutions au moins égal à la longueur de la liste taboue, et il dirige l'exploration de la méthode vers des régions du domaine de solutions non encore visitées. L'algorithme 2.3 donne les principes de base de la méthode tabou classique.

Algorithme 2.3 Tabou classique

1 : *Initialisation*
    x := solution initiale, f_{min} := f(x), X_{min} := x
    TABOU := liste de solutions x, de longueur L
    TABOU := VIDE
2 : répéter
3 : générer un voisinage N(x) tel que $x_i \in$ 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 : jusqu'à conditions d'arrêt satisfaites

La liste taboue est généralement gérée 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 solutions interdites par une liste de "transformations interdites", en interdisant la transformation inverse d'une transformation faite récemment.

4-1-4-2 Critère d'aspiration

Le remplacement de la liste taboue des solutions visitées par la liste des transformations élémentaires conduit non seulement à l'interdiction de revenir vers des solutions précédentes (on évite le cyclage court), mais aussi vers un ensemble de solutions dont plusieurs peuvent ne pas avoir été visitées jusqu'ici. Il est donc primordial de corriger ce défaut et de trouver un moyen de lever l'interdiction de l'acceptation d'une transformation élémentaire déjà effectuée (donc appartenant à la liste taboue), sous un certain critère, appelé critère d'aspiration. Cette correction permet aussi de revenir à une solution déjà visitée et de redémarrer la recherche dans une autre direction.

Le critère d'aspiration le plus simple et le plus couramment utilisé consiste à tester si la solution produite de statut tabou présente un coût inférieur à celui de la meilleure solution trouvée jusqu'à présent. Si cette situation se produit, le statut tabou de la solution est levé. Ce critère est évidemment très sévère, il ne devrait pas être vérifié très souvent, donc il apporte peu de changements à la méthode. D'autres critères d'aspiration plus complexes peuvent être envisagés. L'inconvénient de recourir trop souvent à l'aspiration est qu'elle peut détruire, dans une certaine mesure, la protection offerte par la liste taboue vis-à-vis du cyclage.

Notons que, dans le cas d'une liste taboue de solutions, le concept de critère d'aspiration n'est pas intéressant. Toute annulation du statut tabou d'une solution se trouvant dans la liste taboue pourrait conduire l'algorithme au cyclage.

4-1-4-3 Intensification

L'intensification consiste à approfondir la recherche dans certaines régions du domaine, identifiées comme susceptibles de contenir un optimum global. Cette intensification est appliquée périodiquement, et pour une durée limitée. Pour mieux intensifier la recherche dans une zone bien localisée, plusieurs stratégies sont proposées dans la littérature.

La plus simple consiste à retourner à l'une des meilleures solutions trouvée jusqu'à présent, puis de reprendre la recherche à partir de cette solution, en

réduisant la longueur de la liste taboue pour un nombre limité d'itérations. Dans ce cas, on adapte la procédure de recherche tabou, en élargissant le voisinage de la solution courante (en augmentant la taille de l'ensemble N(x)), tout en diminuant le pas des transformations. On peut aussi remplacer simplement l'heuristique tabou par une autre méthode plus puissante, ou mieux adaptée, pour une recherche locale.

4-1-4-4 Diversification

La diversification permet à l'algorithme de bien explorer l'espace des solutions, et d'éviter que le processus de recherche ne soit trop localisé et laisse de grandes régions du domaine totalement inexplorées. La plus simple des stratégies de diversification consiste à interrompre périodiquement l'acheminement normal de la procédure tabou, et à la faire redémarrer à partir d'une autre solution, choisie aléatoirement, ou "intelligemment". Une autre méthode consiste à biaiser la fonction d'évaluation f, en introduisant un terme qui pénalise les transformations effectuées fréquemment, afin de favoriser des transformations nouvelles. Ce type de stratégie de diversification peut être utilisé de façon continue, sans interrompre la procédure de recherche tabou.

En résumé, nous dirons que la diversification et l'intensification sont des concepts complémentaires, qui enrichissent la méthode de recherche tabou et la rendent plus robuste et plus efficace.

4-1-5 La recherche à voisinage variable

La recherche à voisinage variable (Variable Neighbourhood Search ou VNS) est une méthode récente et pourtant très simple, basée sur la performance des méthodes de descente. Introduite par Mladenovic et Hansen, 1997 [HAN01], la méthode propose simplement d'utiliser plusieurs voisinages successifs quand on se trouve bloqué dans un minimum local.

Avant tout, il est nécessaire de définir un ensemble de k_{max} voisinages, dénotés par $N_{k=1...kmax}$ (et de préférence tels que $N_k \subset N_{k+1}$). On choisit une solution de départ x par heuristique. Ensuite, à partir d'une solution initiale x' choisie dans le

premier voisinage *N(x)* de *x,* on applique une méthode de descente (ou une autre méthode de recherche locale) jusqu'à arriver dans un minimum local (ou que la recherche locale s'arrête). Si la solution trouvée *x″* est meilleure que *x* alors on recentre la recherche en repartant du premier voisinage, sinon on passe au voisinage suivant (qui a priori est plus grand). La recherche s'arrête quand tous les voisinages ne sont plus capables d'améliorer la solution.

Le point crucial dans une VNS, c'est bien évidement la constitution des voisinages de plus en plus grands et inclus les uns dans les autres. Une bonne structure de voisinage -ceci étant vrai pour toute recherche locale- conduit généralement à de bons résultats ou au moins à une recherche efficace.

Cette méthode récente intéresse de plus en plus de chercheurs et est promise à un bel avenir. Le nombre de papiers utilisant cette méthode est en très forte augmentation dans les congrès [SEV04].

Un squelette général décrivant les principales étapes de cette méthode est proposé dans l'algorithme 2.4.

Algorithme 2.4 Recherche à voisinage variable

1: *initialisation* : trouver une solution initiale *x*, *k* :=1
2: répéter
3: *voisinage* : générer une solution *x*' à partir du voisinage $N_k(x)$
4: *recherche locale* : appliquer la procédure de recherche locale à partir de la solution *x*' pour trouver une solution *x*"
5: si *x*" est meilleure que *x* alors
6: *x* := *x*" et *k* := 1 (centrer la recherche sur *x*" et rechercher encore avec un petit voisinage)
7: sinon
8: *k* :=*k* + 1 (élargir le voisinage)
9: fin si
10: jusqu'à $k = k_{max}$

4-1-6 GRASP

Introduite en 1989 par Feo et Resende et présentée dans sa forme plus définitive en 1995 [SEV04], la méthode Greedy Randomized Adaptative Search (GRASP) combine une heuristique gloutonne et une recherche aléatoire. A chaque itération, on construit une solution comme dans une heuristique gloutonne (en se servant d'une liste d'attributs comme liste de priorité). Cette solution est améliorée par l'intermédiaire d'une méthode de descente. En se basant sur la qualité générale de la solution ainsi obtenue, on met à jour l'ordre de la liste des attributs et le processus est itéré jusqu'à satisfaction d'un critère d'arrêt. Un des avantages de cette méthode est la simplicité avec laquelle on peut comprendre le processus d'optimisation. La mise en œuvre elle aussi n'est pas trop compliquée [SEV04].

Algorithme 2.5 GRASP

1: construire une liste de priorité S des attributs de la solution
2: répéter
3: répéter
4: prendre s∈ S avec une heuristique gloutonne
5: $x = x \cup s$
6: jusqu'à solution x complète
7: *recherche locale* : trouver un optimum local x' à partir de x
8: mise à jour de la liste des attributs S
9: jusqu'à critère d'arrêt satisfait

4-1-7 Iterated local search

La méthode iterated local search est une variante très simple des méthodes de descente qui pallient au problème de l'arrêt de ces dernières dans des optima locaux. Dans cette méthode, on génère une solution initiale qui servira de point de départ. Ensuite, on va répéter deux phases : une phase de perturbation aléatoire dans laquelle la solution courante va être perturbée (parfois en tenant compte d'un historique maintenu à jour) et une seconde phase de recherche locale (ou tout simplement de méthode de descente) qui va améliorer cette solution jusqu'à buter sur un optimum local. Dans cette boucle, on est aussi à même d'accepter ou non la

nouvelle solution selon que l'on souhaite donner un caractère plus ou moins agressif à la méthode.

Algorithme 2.6 Iterated local search

1: *initialisation* : trouver une solution initiale *x*
2: répéter
3: *perturbation aléatoire* : trouver une solution *x*' proche de *x*
4: *recherche locale* : à partir de *x*', trouver l'optimum local *x*"
5: si *x*" est meilleure que *x* alors
6: *x* := *x*"
7: fin si
8: jusqu'à critère d'arrêt satisfait

4-1-8 Guided local search

La guided local search (ou recherche locale guidée) est une variante assez élaborée d'une méthode de descente classique. La méthode de base est simple. Elle consiste à modifier la fonction à optimiser en ajoutant des pénalités. La recherche locale est appliquée alors sur cette fonction modifiée. La solution trouvée (qui se trouve être un optimum local) sert à calculer les nouvelles pénalités. Pour cela, on calcule l'utilité de chacun des attributs de la solution et on augmente les pénalités associées aux attributs de valeur maximale. Ces étapes successives sont répétées jusqu'à ce qu'un critère d'arrêt soit validé.

Algorithme 2.7 Guided local search

1: *initialisation* : trouver une solution initiale *x*, poser pénalités=0
2: répéter
3: *fonction coût* : augmenter le coût original de *x* avec les pénalités
4: *recherche locale* : appliquer une procédure de rechercher locale pour trouver un optimum local *x*' (basé sur la fonction coût augmentée)
5: *mise à jour des pénalités* : par calcul des expressions d'utilité
6: *x* := *x*'
7: jusqu'à critère d'arrêt satisfait

4-2 Les méthodes évolutives

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

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

Dans la phase de coopération, les solutions de la population courante sont comparées puis combinées entre elles dans le but de produire des solutions inédites et de bonne qualité à long terme. L'échange d'information qui en résulte se traduit par l'apparition de nouvelles solutions admissibles qui héritent des caractéristiques prédominantes contenues dans les solutions de la population courante. Dans la phase d'adaptation individuelle, les solutions évoluent de manière indépendante en respectant un ensemble de règles prédéfini. Les modifications subies par chacune d'entre elles se font sans aucune interaction avec les autres solutions de la population. Une nouvelle génération de solutions est créée au terme de chaque phase d'adaptation individuelle.

Le mécanisme de recherche qui est à la base d'une approche évolutive est représenté sommairement dans la figure 2.3. Le but est de repérer des solutions aussi bonnes que possible en manipulant à chaque étape un ensemble de solutions localisées dans différentes régions prometteuses de l'espace X.

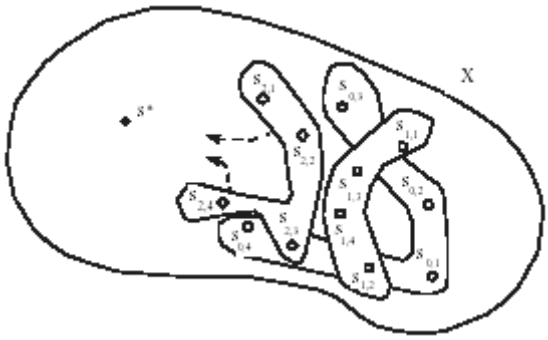

Figure 2.3. Exploration de l'espace X par une approche évolutive

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

4-2-1 Algorithmes génétiques

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

De manière générale, les algorithmes génétiques utilisent un même principe. Une population d'individus (correspondants à des solutions) évolue en même temps comme dans l'évolution naturelle en biologie. Pour chacun des individus, on mesure sa faculté d'adaptation au 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 souvent la valeur de la fonction objectif de la solution associée à l'individu).

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

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

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

Algorithme 2.8 Un algorithme génétique simple

1: *initialisation*: générer une population initiale P de solutions de taille |P|= *n*
2: répéter
3: *selection* : choisir 2 solutions *x* et *x*' par une technique de sélection
4: *croisement* : combiner les 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 remplacée dans la population
7: remplacer *y*' par *y* dans la population
8: jusqu'à critère d'arrêt satisfait

4-2-2 Algorithmes de colonies de fourmis

Cette nouvelle métaheuristique imite le comportement de fourmis cherchant de la nourriture. A chaque fois qu'une fourmi se déplace, elle laisse sur la trace de son passage une odeur (la phéromone). Comme la fourmi est rarement une exploratrice isolée, avec plusieurs de ses congénères, elle explore une région en quête de nourriture. Face à un obstacle, le groupe des fourmis explore les deux côtés de l'obstacle et se retrouvent, puis elles reviennent au nid avec de la nourriture. Les autres fourmis qui veulent obtenir de la nourriture elles aussi vont emprunter le même chemin. Si celui-ci se sépare face à l'obstacle, les fourmis vont alors emprunter préférablement le chemin sur lequel la phéromone sera la plus forte. Mais la phéromone étant une odeur elle s'évapore. Si peu de fourmis empruntent une trace, il est possible que ce chemin ne soit plus valable au bout d'un moment, il en est de même si des fourmis exploratrices empruntent un chemin plus long (pour le contournement de l'obstacle par exemple). Par contre, si le chemin est fortement emprunté, chaque nouvelle fourmi qui passe redépose un peu de phéromone et renforce ainsi la trace, donnant alors à ce chemin une plus grande probabilité d'être emprunté.

Algorithme 2.9 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 phéromone
5: mise à jour des traces de phéromone basée sur la qualité des solutions trouvées
6: fin pour
7: jusqu'à critère d'arrêt satisfait

CHAPITRE 3

Ordonnancement sur machine à contrainte de ressource non renouvelable

1 Introduction

Dans ce chapitre nous abordons l'étude du problème d'ordonnancement sur une machine sous contrainte d'une ressource non renouvelable. L'objectif est la minimisation de la somme pondérée des dates de fin d'exécution des tâches.

Nous proposons de résoudre ce problème en utilisant les métaheuristiques. Deux approches sont utilisées puis comparées pour garder celle qui permet de donner la meilleure solution. La première est une recherche locale basée sur un voisinage généré par la méthode de séparation et évaluation, tandis que la deuxième utilise la méthode tabou.

Les deux approches ont été utilisées avec différentes stratégies. La recherche locale a été testée avec deux méthodes différentes pour le calcul de la borne inférieure nécessaire à l'exploration de l'arbre par branch & bound. La méthode taboue est utilisée avec deux techniques différentes de définition du voisinage. Enfin les meilleures méthodes des deux approches ont été analysées dans le but d'en retenir la plus efficace pour notre problème.

2 Présentation et modélisation du problème

2-1 Introduction

Le problème qu'on se propose d'étudier dans ce chapitre est l'ordonnancement d'un certain nombre de tâches sur une machine unique. Chaque tâche nécessite une certaine quantité de ressource pour qu'elle soit exécutée. Cette ressource est non renouvelable (ou consommable), c'est-à-dire elle diminue à chaque utilisation par les tâches, jusqu'à une certaine limite, exemples : les lubrifiants, le carburant, la matière première, l'énergie sur batterie, les produits semi-finis, …

Une caractéristique essentielle de cette ressource est qu'elle est produite dans un système externe avec une vitesse constante en fonction du temps. Cette

production est stockée dans une zone de stockage où un stock initial peut éventuellement exister avant le lancement de la production.

Pour qu'elles soient exécutées, les tâches requièrent une certaine quantité déterminée a priori de la ressource. Pour satisfaire leurs demandes, les tâches doivent consommer directement la production instantanée de la ressource, elles ont recours au stock de la ressource, si la quantité demandée n'est pas satisfaite par la production instantanée.

Le problème est de trouver un ordonnancement des tâches pour minimiser la somme pondérée des dates de fin, en satisfaisant la contrainte de disponibilité de la ressource.

2-2 Revue de la littérature

Dans ce qui suit, nous allons présenter un aperçu de quelques travaux récents qui ont traité les problèmes d'ordonnancement à contrainte de ressources non renouvelables.

A.Janiak, C.N.Potts et T.Tautenhahn [JAN00] ont considéré le problème d'ordonnancement de n tâches pour minimiser le makespan. Les contraintes de précédences sont imposées ainsi que les dates de disponibilités des tâches. Ces dates de disponibilités ne sont pas fixes mais dépendent d'une ressource additionnelle. Plus précisément, la date de disponibilité de chaque tâche est fonction de la ressource allouée à elle. La quantité de la ressource est déterminée. Ce problème est trouvée dans les ateliers de production des feuilles métalliques, l'étape principale de ce système est le laminage à chaud, comme cet équipement est très chère, les tâches doivent être préchauffées par un gaz durant un certain moment qui dépend de l'intensité du flux du gaz, avant de passer au laminage. Si les quantités allouées aux tâches sont fixes et déterminées, le problème particulier est équivalant au $1|prec, r_i|C_{max}$ et, il est résolu par la règle de Jackson $O(n^2)$. Les auteurs ont développées différentes heuristiques à ce problème, en fixant un ensemble de valeurs possibles des dates de disponibilités. Le cas le plus simple est lorsque toutes les tâches ont des dates de disponibilités égales. Dans le cas où 2

valeurs sont allouées, le programme est à valeurs entières et devient un problème de knapsack (sac à dos). Dans le cas général où *k* valeurs sont allouées et où les contraintes de précédence sont supprimées; les auteurs ont montré qu'une LP-relaxation du programme à valeurs entière peut-être utilisée pour ordonnancer (*n*-*k*) tâches de façon optimale.

K.De Bontridder [BON01] a utilisé des méthodes de recherche locales pour résoudre des problèmes d'approvisionnement et de planification de la production en présence d'une ressource non renouvelable. Le problème de JobShop posé par l'auteur est une généralisation du problème classique. Les dates de disponibilités, les règles de précédence, les dates de début et de fin des tâches et la contrainte de ressource non renouvelable, sont imposées dans l'objectif de trouver un ordonnancement qui minimise la somme pondérée des retards. La ressource peut être produite par d'autres tâches dans le même système ou livrée par des fournisseurs avant le début d'exécution des tâches. L'auteur a proposé une heuristique basée sur l'approche "list scheduling" pour générer un ordonnancement qui doit satisfaire les contraintes de précédence des tâches sans créer un stock négatif de la ressource. L'approche de recherche tabou est ensuite utilisée pour trouver un ordonnancement de bonne qualité.

Un article publié en 2005 par A.Grigoriev, M.Holthuijsen et J.V. De Klundert [GRI05] a étudié un ensemble de problèmes d'ordonnancement sur une machine en présence d'une ou plusieurs matières premières. Les auteurs ont étudié plusieurs problèmes particuliers à contrainte de matières premières où les temps d'exécution sont unitaires ou égales.

Une première classe des problèmes étudiés a pour objectif la minimisation du plus grand retard L_{max}, elle est composée des cas suivants :

1- **La matière première est dédiée à chaque tâche :** ce problème est équivalent à celui où l'on impose des dates de disponibilités aux tâches. Des algorithmes polynomiaux sont connus pour les problèmes $1|r_j,prec,p_j=p|L_{max}$.

2- **Les temps d'exécution sont unitaires, en présence d'une seule matière première :** un algorithme basé sur la règle EDD est utilisé pour résoudre ce problème en $O(n^2)$.

3- **Les temps d'exécution sont égaux, en présence de deux matières premières :** Les auteurs ont démontré que le problème où les temps d'exécution sont unitaires, en présence de deux matières est NP-difficile. Un algorithme d'approximation est proposé.

La 2ème classe a été abordée dans un objectif commun : la minimisation du makespan, ces problèmes sont :

1- **La matière première est dédiée à chaque tâche :** ce problème est équivalent à celui où l'on impose des dates de disponibilités aux tâches. Des algorithmes polynomiaux sont connus pour les problèmes $1|r_j|C_{max}$.

2- **Les temps d'exécution sont égaux, en présence d'une seule matière première :** Ce problème est NP-difficile. Le cas particulier, où la consommation de la matière est unitaire, est étudié. Une réduction au problème du flowshop à 2 machines est proposée, dont l'algorithme de résolution est connu sous le nom de la méthode de Johnson.

3- **Plusieurs matières premières :** Le problème de 2 matières premières, où les temps d'exécution sont unitaires, est NP-difficile. Pour le cas général, un algorithme d'approximation est proposé.

Les travaux de H.Belouadah [BEL04] ont traité le problème d'ordonnancement sur machine à contrainte de ressource non renouvelable pour minimiser la somme pondérée des dates de fin. Un algorithme optimal a été proposé pour le cas particulier du problème où les temps d'exécution sont unitaires ainsi que les quantités de la ressource. Le problème général a été résolu par la méthode Branch-and-Bound, en proposant quelques règles de dominance, ainsi qu'une borne inférieure basée sur la relaxation lagrangienne.

2-3 Formulation mathématique

On considère le problème d'ordonnancement de n tâches sur une machine qui n'exécute qu'une seule tâche à la fois. Les tâches ont des temps d'exécutions p_i, des poids w_i sont allouées aux tâches selon leur importance.

Une ressource non renouvelable est sollicitée pour que les tâches soient exécutées. Cette ressource est produite d'une façon continue dans un système externe suivant une vitesse constante r par unité de temps. La production est stockée dans une zone de stockage où un stock initial l_0 existe au temps zéro. La tâche i requiert la ressource en quantité déterminée a_i .

Le problème consiste à trouver un ordonnancement qui minimise la somme pondérée des dates de fin des tâches $\sum w_i C_i$, tout en satisfaisant la contrainte de la ressource.

La contrainte de la ressource est formulée par une inégalité qui doit être satisfaite dans tout ordonnancement admissible et qui impose que la totalité des demandes de la ressource par les tâches ne doit pas dépasser la production totale (en intégrant, éventuellement, un stock initial) [BON01]. Cette condition est formulée par l'inégalité :

$$\sum_{i=1}^{n} a_i \le l_0 + \sum_{i=1}^{n} (p_i r)$$

avec : a_i la quantité demandée par la tâche i

l_0 le stock initial de la ressource au temps zéro.

$p_i r$ la quantité produite de la ressource.

Cette inégalité peut être écrite : $\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} (p_i r) \le l_0$

ou encore $\sum_{i=1}^{n} (a_i - p_i r) \le l_0$

en posant $q_i = a_i - p_i r$, i=1, …,n l'inégalité devient : $\sum_{i=1}^{n} q_i \le l_0$

Soit σ = (σ(1), …, σ(*n*)) une séquence de *n* tâches. Posons *P* l'ensemble de toutes les permutations de séquences possibles. On peut écrire une formulation mathématique du problème comme suit :

$$(\text{Prq}) \quad \begin{cases} f = \min_{\sigma \in P} \sum_{i=1}^{n} w_{\sigma(i)} C_{\sigma(i)} \\ \\ \sum_{\alpha=1}^{i} q_{\sigma(\alpha)} \le l_0 \ , \quad i = 1,\ldots,n \quad (1) \end{cases}$$

La proposition suivante concerne l'admissibilité d'une séquence pour le problème (Prq).

Proposition 1 : Soit une séquence σ = (σ(1), …, σ(*n*)). Si la séquence σ vérifiant les inégalités $q_{\sigma(1)} \le q_{\sigma(2)} \le \ldots \le q_{\sigma(n)}$ est une séquence non admissible alors il n'existe aucune séquence admissible pour le problème (Prq).

Preuve : Puisque σ n'est pas admissible, alors $\exists i_0 \in \{1,2,\ldots,n\}$ telle que $\sum_{i=1}^{i_0} q_{\sigma(i)} \ge l_0$. Considérons une séquence arbitraire σ', posons t=max{ $i \mid 1 \le \sigma'(i) \le i_0$ }. Puisque l'on a $\{1,2,\ldots,n\} \subseteq \{ \sigma'(1), \ldots, \sigma'(t) \}$, alors on a toujours la tâche i_0, telle que $\sum_{i=1}^{t} q_{\sigma(i)} \ge \sum_{i=1}^{i_0} q_{\sigma(i)} \ge l_0$ et donc σ' n'est pas admissible.

Pour assurer la possibilité d'existence d'une solution admissible, la condition suivante doit être satisfaite : pour chaque tâche *i* , la somme des quantités de la ressource relatives aux tâches antérieures jusqu'à *i*, ne doit pas dépasser la quantité initiale l_0 i.e., $\sum_{\alpha=1}^{i} q_{\sigma(\alpha)} \le l_0$. Pour plus de simplicité et sauf mention particulière, nous posons désormais $l_0 = 0$.

Le problème d'ordonnancement sur machine sous contrainte d'une ressource non renouvelable est un problème NP-difficile [GEN84] et [BEL04]. Dans la section suivante nous allons aborder la résolution de ce problème par une méthode de recherche locale basée sur les techniques de séparation et évaluation, ensuite nous proposons une métaheuristique différente qui est la recherche taboue, avant de clôturer le chapitre avec une étude expérimentale des deux méthodes.

3 Résolution du problème par une recherche locale basée B&B

3-1 Introduction

Les méthodes de recherche locale sont des algorithmes itératifs qui explorent l'espace de solutions admissibles X en se déplaçant pas à pas d'une solution à une autre. Une méthode de ce type débute à partir d'une solution $x_0 \in X$ choisie arbitrairement ou alors obtenue par le biais d'une heuristique. Une solution x' est obtenue à partir de x en appliquant une modification élémentaire. Le voisinage $N(x)$ d'une solution $x \in X$ est défini comme l'ensemble des solutions admissibles atteignables depuis x en effectuant une modification élémentaire. Un tel processus d'exploration est interrompu lorsqu'un ou plusieurs critères d'arrêt sont satisfaits.

La méthode de descente est un exemple de méthode de recherche locale. Une telle méthode progresse au travers de X en choisissant à chaque étape la meilleure solution voisine de la solution courante. Ce procédé est répété aussi longtemps que la valeur de la fonction objectif diminue. La recherche s'interrompt dès lors qu'un minimum local de f est atteint.

Nous allons présenter dans ce qui suit une résolution de notre problème par une méthode itérative de descente en utilisant l'approche par séparation et évaluation (B&B). La section 2 fournit une description du voisinage utilisé et la section 3 présente la stratégie de sélection d'une solution dans le voisinage généré.

Quelques règles de dominance sont décrites dans la section 4, avant d'aborder la section 5 en proposant deux bornes inférieures qui doivent être intégrées dans le corps de la méthode B&B, la première utilise la méthode de relaxation lagrangienne et une technique itérative pour obtenir une meilleure borne inférieure, tandis que la deuxième est une application directe de la règle de Smith.

Ensuite, la section 6 présente une heuristique de recherche d'une séquence initiale, avant de passer à la section 7 qui décrit les grandes étapes de la méthode de recherche globale.

Enfin, dans la section 8 nous effectuerons une analyse et commentaires des résultats de nos tests, puis nous donnons une conclusion à propos de la méthode utilisée.

3-2 Définition du voisinage

Dans les méthodes de recherche locale, la définition du voisinage est de loin, l'étape la plus importante qui permet de rendre efficace ou non la méthode utilisée. La technique utilisée pour définir un voisinage pour notre problème consiste à construire un arbre basé sur un algorithme de séparation et évaluation.

La construction de l'arbre est décrite comme suit : Le nœud N_0 au niveau 0 de l'arbre, s'appelle la racine. A ce nœud, aucune tâche n'est ordonnancée. Au niveau 1, il y a k nœuds fils de N_0 qui sont générés. Le paramètre k détermine le nombre de nœuds fils générés à partir d'un nœud père. Ce paramètre prend au démarrage de l'exécution une valeur initiale k_0. En général, l'arbre est ainsi construit, en générant à partir de chaque nœud, un nombre k nœuds fils de l'ensemble des tâches non ordonnancées dans l'ordre de leurs positions dans la séquence initiale σ, jusqu'à arriver au niveau n-1, qui correspond à une séquence partielle où il reste exactement une tâche non ordonnancée, d'où chaque nœud du niveau n-1 correspond à une permutation qui est en fait un voisinage de la séquence σ.

Il est évident que si $k=n$, le voisinage doit contenir toutes les permutations possibles, c'est-à-dire $n!$. et que si $k=1$, le voisinage contient seulement la séquence σ. Par ailleurs, si $1<k<<n$, la taille du voisinage reste toujours exponentielle mais elle est réduite d'une manière significative par rapport à l'ensemble de toutes les permutations possibles.

Nous donnons dans la figure 3.1, un exemple de la construction d'un tel voisinage. Soit la séquence $\sigma=1234$. Avec $k=3$, le voisinage de σ est composé de 18 séquences énumérées aux feuilles de l'arbre généré.

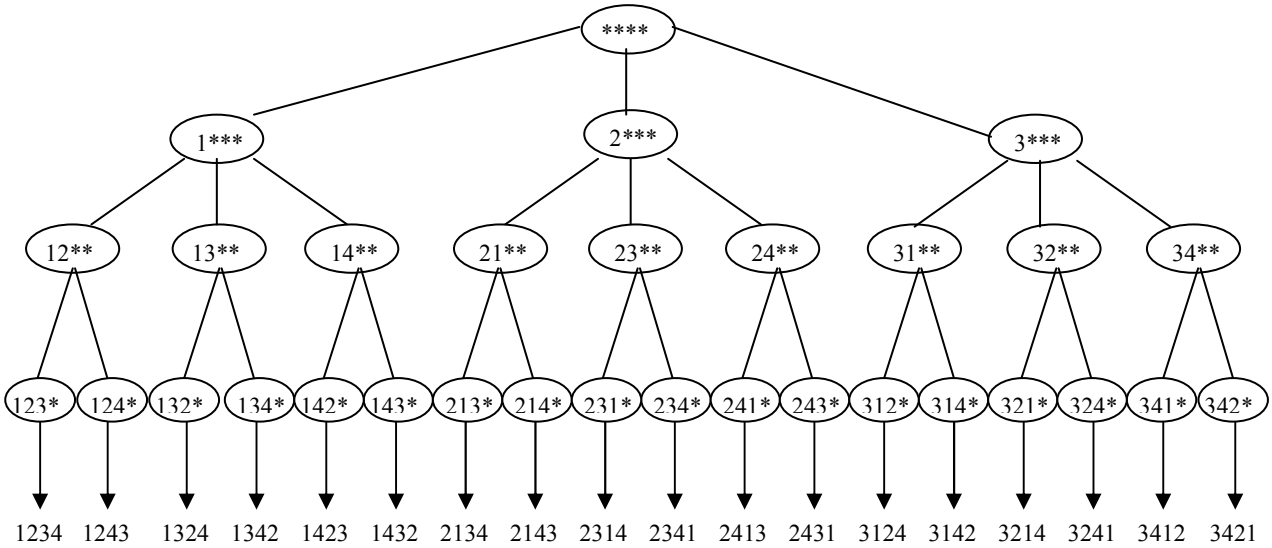

Figure 3.1 : Exemple de voisinage construit par un arbre.

La taille du voisinage généré par cette technique est déterminée comme suit :

Observation 1 : Etant donnée une séquence σ, la cardinalité de l'ensemble N(σ) voisinage de σ est $k!k^{n-k}$.

Preuve : On remarque que pour chaque nœud de niveau l (l=0,1,…,n-k-1), on peut brancher k nœuds, donc k^{n-k} nœuds sont construits jusqu'au niveau n-k-1. A partir du niveau n-k jusqu'au niveau n-1, à chaque nœud est branché exactement n-l nœuds, c'est-à-dire : $(n-(n-k))(n-(n-(k-1)))\ldots(n-(n-1))= k(k-1)\ldots1=k!$. D'où le nombre de nœuds générés aux feuilles (i.e. la taille du voisinage) égal à $k!k^{n-k}$.

3-3 Stratégie de sélection d'une solution

Il existe deux alternatives de sélection d'un voisin qui améliore la solution. La première, connue sous le nom : stratégie de meilleure amélioration, consiste à choisir dans le voisinage de σ le meilleur voisin qui améliore la solution actuelle. La deuxième, connue sous le nom : stratégie de première amélioration, consiste à choisir dans le voisinage de σ le premier voisin rencontré qui améliore la solution.

Pour notre voisinage qui est de taille exponentielle, la stratégie de meilleure amélioration est inadéquate, du fait qu'elle nécessite la définition complète de l'ensemble N(σ), ce qui prendra un temps exponentiel de le faire. Nous avons donc choisi la stratégie de première amélioration pour sélectionner un voisin de l'ensemble N(σ).

Mieux encore, nous avons utilisé les techniques de séparation et évaluation pour explorer l'arbre du voisinage. A chaque nœud généré, on calcule sa borne inférieure, si cette borne est supérieure à la solution actuelle, le nœud est supprimé, sinon on continue l'exploration de l'arbre, en mémorisant ce nœud pour un traitement ultérieur par backtracking, ou en branchant immédiatement à partir de ce nœud, et ce, jusqu'à arriver au niveau *n*-1 de l'arbre, où l'on peut déterminer un voisin de la séquence σ. La stratégie de première amélioration consiste à arrêter l'exploration de l'arbre, dès qu'un voisin rencontré améliore la solution actuelle.

La technique Branch-and-Bound que nous avons utilisé consiste à appliquer également des règles de dominance (bien choisies) pour éliminer le plus grand nombre de nœuds possibles durant l'exploration de l'arbre, ceci permet d'alléger d'une façon significative la taille de l'arbre et par conséquent accélère l'accès aux feuilles.

Un dernier élément que nous avons ajouté à l'algorithme de recherche dans le voisinage B&B consiste à ne pas fixer le nombre *k* de nœuds à brancher à partir d'un nœud père, mais plutôt de le choisir aléatoirement par une loi de distribution uniforme dans l'intervalle [2, k_{max}], avec k_{max} une valeur fixe.

L'algorithme qui détermine le premier voisin de σ dans un voisinage B&B, est décrit dans ce qui suit :

Notation

C	Ensemble des nœuds candidats
$J(N_k)$	Ensemble des tâches non ordonnancées relatif au nœud N_k
$f(\sigma)$	Valeur de la fonction objectif de σ ; $f(\sigma)=\Sigma_{i\in\sigma} w_i C_i$
$l(N_k)$	Niveau du nœud N_k
UB	Borne supérieure de la fonction objectif du meilleur voisin.

$LB(N_k)$	Borne inférieure associée au nœud N_k
σ_k	permutation qui correspond au nœud N_k, au niveau $l(N_k)=n-1$
η	Nombre de nœuds créés
k_{max}	Nombre maximum de nœuds à brancher à partir d'un nœud père
k	Nombre de nœuds à brancher prix aléatoirement entre 2 et k_{max}

***Procedure* 3.1** *Premier_Voisin*(σ)

Initialisation $C=\{N_0\}$, $UB=f(\sigma)$, $\eta=1$;

répéter jusqu'à $C=\emptyset$

Sélectionner $N_r \in C$ ayant la borne inférieure la plus petite parmi les nœuds récemment créés.

$C=C\backslash\{ N_r \}$;

Prendre k aléatoirement dans $U[2, k_{max}]$

$s=\min(k, |J(N_r)|)$

Brancher sur N_r générant s nœuds fils, numérotés $N_{\eta+1}, \ldots, N_{\eta+s}$

Pour p=1 **à** *s* **faire**

Vérifier admissibilité($N_{\eta+p}$)

Appliquer Règles de Dominance($N_{\eta+p}$)

Si $N_{\eta+p}$ éliminé **alors** aller à **FinPour**

Calculer $LB(N_{\eta+p})$

Si $LB(N_{\eta+p}) \geq UB$ **alors** Eliminer $N_{\eta+p}$; aller à **FinPour**

Si $l(N_{\eta+p})=n-1$ **alors**

si $f(\sigma_{\eta+p})<UB$ **alors**

Arrêter la recherche avec $\sigma_{\eta+p}$ voisin de première amélioration. STOP

finsi

Finsi

$C=C\cup\{ N_{\eta+p} \}$;

Finpour

Finrépéter

3-4 Règles de dominance

Pour avoir un algorithme Branch-and-Bound efficace et rapide, différentes règles de dominance sont appliquées sur les nœuds générés de l'arbre. Ces règles ont pour objectif de laisser la séquence partielle obtenue si elle peut garantir une solution optimale, sinon la séquence est abandonnée (on dit alors que le nœud est dominé).

Pour notre problème, les règles de dominance que nous avons choisi ont été utilisées par H.Belouadah [BEL04]. Nous en avons sélectionné les plus simples et les plus efficaces.

Règle de dominance N°1 :

Soit $\sigma = \sigma_1 i$ une séquence partielle et soit U l'ensemble des tâches non encore ordonnancées, alors il existe un ordonnancement optimal dans lequel la tâche i est précédée par une tâche quelconque $j \in$ U telle que $p_j \leq p_i$; $w_j \geq w_i$ et $q_j \leq q_i$.

A tout instant t, la tâche qui arrive la dernière doit être dominée par les tâches ayant une durée d'exécution plus courte, un poids plus grand et sollicitant moins de quantité de ressource.

Règle de dominance N°2 :

Cette règle est une conséquence directe de la programmation dynamique. La séquence partielle initiale est dominée, s'il est possible de permuter ses deux dernières tâches, sans faire augmenter la somme pondérée des dates de fin de ses tâches, tout en vérifiant la contrainte de ressource.

Soit $\sigma = \pi i$ une séquence partielle et soit Q_π la somme des quantités de ressource relatives à π avec $Q_\pi \leq 0$. Soit U l'ensemble des tâches non positionnées.

Théorème 1 : Si pour toute tâche $j \in$ U telle que $\frac{p_j}{w_j} \leq \frac{p_i}{w_i}$ et $Q_\pi + q_j \leq 0$ alors $\sigma = \pi i$ est dominée.

Preuve : Pour prouver ce théorème, nous devons citer une proposition inspirée de [POS81] qui, ne tenant pas compte des conditions d'admissibilité, donne le coût relatif de permutation de deux tâches dans une séquence quelconque.

Proposition 2 : Soit $\sigma = \sigma_1 i \sigma_2 j \sigma_3$ et $\sigma' = \sigma_1 j \sigma_2 i \sigma_3$.

$$f(\sigma) \leq f(\sigma') \text{ ssi } (p_i - p_j)\sum_{\sigma_2} w_k + (w_j - w_i)\sum_{\sigma_2} p_k \leq w_i p_j - w_j p_i$$

preuve : Soient $v_t = \sum_{\sigma_t} p_k$ et $u_t = \sum_{\sigma_t} w_k$. Supposons que $f(\sigma) \leq f(\sigma')$, alors

$$w_i(v_1 + p_i) + u_2(v_1 + p_i) + w_j(v_1 + v_2 + p_i + p_j) \leq$$
$$w_j(v_1 + p_j) + u_2(v_1 + p_j) + w_i(v_1 + v_2 + p_j + p_i)$$

d'où $$u_2(v_1 + p_i) + w_j(v_2 + p_i) \leq u_2(v_1 + p_j) + w_i(v_2 + p_j)$$

par conséquent

$$u_2(p_i - p_j) + (w_j - w_i)v_2 \leq w_i p_j - w_j p_i .$$

L'inverse s'obtient de la même manière.

Un résultat particulier découle de cette proposition (due à Smith) est le suivant :

Corollaire 1 : Si $\sigma_2 = \varnothing$, alors $f(\sigma) \leq f(\sigma')$ ssi $\frac{p_i}{w_i} \leq \frac{p_j}{w_j}$.

Revenons maintenant à la preuve du théorème 1; soit $\sigma = \pi i$, pour toutes tâche $j \in$ U, si les conditions d'admissibilité sont satisfaites; à savoir : $Q_\pi + q_j \leq 0$, alors en appliquant le Corollaire 1, $f(\pi ji) \leq f(\pi ij)$ ssi $\frac{p_j}{w_j} \leq \frac{p_i}{w_i}$ donc πi est dominée.

Règle de dominance N°3 :

Cette règle donne la priorité d'être ordonnancée, parmi les tâches non encore positionnées, à celle qui satisfait les conditions d'admissibilité et qui présente un rapport p_i/w_i minimal.

Soit U l'ensemble des tâches non positionnées et soit σ une séquence partielle avec Q_σ la somme des quantités de ressource relatives à σ ($Q_\sigma \leq 0$).

Théorème 2 : Pour toute tâche $j \in$ U, si $p_j / w_j = \min_{k \in U}(p_k / w_k)$ et $Q_\sigma + q_j \leq 0$ alors il existe un ordonnancement optimal dans lequel la tâche *j* est ordonnancée en première position libre parmi les tâches de U.

Preuve : Une fois encore, la preuve de ce théorème est basée sur le corollaire 1. En effet, parmi les tâches non ordonnancées, la tâche la plus prioritaire est celle qui présente un rapport p_i/w_i inférieur à celui des autres tâches et qui satisfait les conditions d'admissibilité (i.e., la quantité de ressource qu'elle nécessite ne rend pas le stock négatif).

3-5 Bornes inférieures

Pour trouver une borne inférieure pour le problème (Prq), on propose d'utiliser la relaxation lagrangienne de la contrainte de ressource (1) (cf. 2-3).

En associant à cette contrainte un multiplicateur a_i, le problème lagrangien sera de la forme :

$$L(a_i) = \min_{\sigma \in P}\left\{\sum_{i=1}^{n} w_{\sigma(i)} C_{\sigma(i)} + \sum_{i=1}^{n} a_i \sum_{\alpha=1}^{i} q_{\sigma(\alpha)}\right\} \text{ avec } a_i \geq 0, \quad i=1,\ldots,n$$

en posant $a_i = kw_i$, *avec* $k \geq 0$, on aura

$$L(k) = \min_{\sigma \in P}\left\{\sum_{i=1}^{n} w_{\sigma(i)} (C_{\sigma(i)} + k\sum_{\alpha=1}^{i} q_{\sigma(\alpha)})\right\}$$

ou encore $$L(k) = \min_{\sigma \in P}\left\{\sum_{i=1}^{n} w_{\sigma(i)} \sum_{\alpha=1}^{i} (p_{\sigma(\alpha)} + kq_{\sigma(\alpha)})\right\} \quad \text{avec } k \geq 0$$

Le problème lagrangien $L(k)$ peut être résolu en utilisant la règle SWPT de Smith, en posant $p'_i = p_i + kq_i$ *avec* $i = 1,\cdots,n$. Pour cela, il faut qu'on ait le rapport $(p_i + kq_i)/w_i \geq 0$. Puisque $w_i > 0$; on doit avoir $p_i + kq_i \geq 0$

si $q_i \geq 0$; la condition est satisfaite. si $q_i < 0$; on doit avoir $k \leq \frac{-p_i}{q_i}$

d'où $k \in [0, a]$ avec $a = \min_{q_i \prec 0} \left\{ \frac{-p_i}{q_i} \right\}$

Posons :

$$R_{\sigma^{(r)}(i)}(k) = \frac{p_{\sigma^{(r)}(i)} + kq_{\sigma^{(r)}(i)}}{w_{\sigma^{(r)}(i)}} \quad \text{et} \quad L^{(r)}(k) = A_{\sigma^{(r)}} + kB_{\sigma^{(r)}}$$

avec $A_{\sigma^{(r)}} = \sum_{i=1}^{n} w_{\sigma^{(r)}(i)} C_{\sigma^{(r)}(i)}$ et $B_{\sigma^{(r)}} = \sum_{i=1}^{n} w_{\sigma^{(r)}(i)} \sum_{\alpha=1}^{i} q_{\sigma^{(r)}(\alpha)}$

Initialement, les tâches sont ordonnées suivant la règle SWPT, générant ainsi une séquence $\sigma^{(0)} = (\sigma(1), \cdots, \sigma(n))$ ayant une borne inférieure $L^{(0)}(0) = A_{\sigma^{(0)}}$. En fait, ceci est la borne inférieure utilisée par P.C.Geng [GEN84] que nous allons la comparer dans notre méthode de recherche locale, avec la borne inférieure obtenue par relaxation lagrangienne, dont la description est comme suit :

A la première itération *r*=1, on considère la séquence $\sigma^{(1)} = \sigma^{(0)}$ et l'on choisit *k* tel que $\sigma^{(1)}$ soit une solution du problème lagrangien. En appliquant la règle de Smith, on obtient des ensembles $S_i^{(1)}$ tels que :

$$S_i^{(1)} = \left\{ h \in [0, a] \,\middle|\, R_{\sigma^{(1)}(i)}(h) \leq R_{\sigma^{(1)}(i+1)}(h) \right\}, \quad i = 1, \cdots, n-1$$

Le nouveau intervalle de *k* sera l'intersection de tous les ensembles $S_i^{(1)}$, tel que $k \in S^{(1)}$ avec $S^{(1)} = \bigcap_{i=1}^{n-1} S_i^{(1)}$. Supposons $S^{(1)} = \left[a^{(0)}, a^{(1)}\right]$ avec $a^{(0)} = 0$. Dans ce cas, la nouvelle borne inférieure sera : $L^{(1)}(k) = A_{\sigma^{(1)}} + kB_{\sigma^{(1)}}$

Si on trouve $B_{\sigma^{(1)}} \leq 0$, on arrête la procédure itérative, avec la borne inférieure maintenue : $L^{(1)}(0) = A_{\sigma^{(1)}}$, sinon on améliore la borne, en passant à une autre itération *r* =*r*+1, à partir d'une nouvelle séquence obtenue par permutation de deux tâches *i* et *i*+1, avec *i* l'indice correspondant à l'égalité :

$$R_{\sigma^{(r-1)}(i)}(a^{(r-1)}) = R_{\sigma^{(r-1)}(i+1)}(a^{(r-1)})$$

On arrête la procédure dès qu'on trouve, à l'itération *r*, la valeur $B_{\sigma^{(r)}} \le 0$, on prend dans ce cas la dernière borne inférieure trouvée $LB = L^{(r-1)}(k)$

Dans ce qui suit, nous donnons l'algorithme (noté LB_RL) qui calcule la borne inférieure par relaxation lagrangienne.

***Procedure* 3.2** LB_RL /* Borne inférieure par Relaxation Lagrangienne

obtenir $\sigma^{(0)} = (\sigma(1), \cdots, \sigma(n))$ en appliquant la règle SWPT

trouver $a = \min_{q_{\sigma^{(0)}(i)} \le 0} \left\{ \dfrac{-p_{\sigma^{(0)}(i)}}{q_{\sigma^{(0)}(i)}} \right\}$; $k \in [0, a]$

$L^{(0)}(0) = A_{\sigma^{(0)}}$; $a^{(0)} = 0$; $\sigma^{(1)} = \sigma^{(0)}$; r=1

$S_i^{(1)} = \left\{ h \in \left[a^{(0)}, a \right] \middle| R_{\sigma^{(1)}(i)}(h) \le R_{\sigma^{(1)}(i+1)}(h) \right\}, \quad i = 1, \cdots, n-1$

$S^{(1)} = \left[a^{(0)}, a^{(1)} \right] = \bigcap_{i=1}^{n-1} S_i^{(1)}$

répéter jusqu'à $B_{\sigma^{(r)}} \le 0$

$\sigma^{(r+1)} \equiv \sigma^{(r)}$ *sauf* $\sigma^{(r+1)}(i) = \sigma^{(r)}(i+1)$ *et* $\sigma^{(r+1)}(i+1) = \sigma^{(r)}(i)$

r=r+1

$S_i^{(r)} = \left\{ h \in \left[a^{(r-1)}, a \right] \middle| R_{\sigma^{(r)}(i)}(h) \le R_{\sigma^{(r)}(i+1)}(h) \right\}, \quad i = 1, \cdots, n-1$

$S^{(r)} = \left[a^{(r-1)}, a^{(r)} \right] = \bigcap_{i=1}^{n-1} S_i^{(r)}$

Fin répéter

$k^* = a^{(r-1)}$

$LB = L^{(r-1)}(k^*)$

La complexité de cet algorithme est $O(n^2 \log n)$ [BEL04].

A titre de rappel, nous présentons ci-dessous l'algorithme utilisé par P.C.Geng [GEN84], qui permet de calculer la borne inférieure en appliquant la règle de Smith.

Notation

S : ensemble des tâches non encore ordonnancées

LB: borne inférieure

h: position de la tâche dans la séquence

P: temps d'exécution des tâches de 1 à h.

Procedure 3.3 LB_SWPT /* Borne inférieure par SWPT

Initialisation

S={1,2,…,*n*} ; LB=0 ; h=0 ; P= 0;

Tant que S ≠ ∅ faire

trouver la tâche *i* telle que $p_i / w_i = \min_{k \in S}(p_k / w_k)$

effectuer h=h+1, affecter à la tâche *i* la position h ;

effectuer P=P+p_i ; LB=LB+w_iP ;

S=S-{*i*} ;

Fin tant que

La complexité de cet algorithme est $O(n \log n)$ [GEN84].

3-6 Heuristique d'une solution initiale

Pour obtenir une solution initiale, nous avons utilisé une heuristique basée sur celle proposée par H.Belouadah [BEL04], dont le principe est le suivant : « affecter à la première position libre, la tâche ayant le plus petit rapport p_i/w_i , parmi les tâches non encore ordonnancées qui vérifient la contrainte de ressource ». La complexité de cet algorithme est $O(n \log n)$.

Notation

S : ensemble des tâches à ordonnancer

J: ensemble des tâches non encore ordonnancées

f : coût de la fonction objectif

h: position de la tâche dans la séquence

P: temps d'exécution des tâches de 1 à h.

Q: quantité de ressource des tâches de 1 à h.

***Procedure* 3.4** HB ; /* Heuristique d'une solution initiale

Initialisation

S={1,2,…,n},

Q=0 ; h=0, f=0, P=0;

Tant que S ≠ ∅ faire

définir J={$j \mid j \in$ S ;Q+$q_j \leq 0$} ;

trouver la tâche i telle que $p_i / w_i = \min_{k \in J}(p_k / w_k)$

effectuer h=h+1, affecter à la tâche i la position h ;

effectuer P=P+p_i ; f=f+w_iP ; Q=Q+q_i ;

S=S-{i} ;

Fin tant que

3-7 Description de la méthode de recherche globale

La méthode de recherche globale proposée consiste à redémarrer autant de fois la procédure de recherche du premier voisin, dans le but d'améliorer la solution trouvée. Toutefois, la recherche du premier voisin qui améliore la solution par exploration de l'arbre B&B, peut ne donner aucun résultat améliorant la solution. Pour faire face à cette carence et sortir des minima locaux, nous avons utilisé une technique d'élargissement du voisinage, en augmentant le nombre (k_{max}) de nœuds à générer dans l'arbre.

La règle d'élargissement du voisinage est fixée comme suit : initialement, on pose $k_{max}=k_0$, si aucune amélioration n'est trouvée dans le voisinage de la séquence courante σ, on pose $k_{max}=k_{max}+[n/10]$, sinon on laisse $k_{max}=k_0$.

Cette technique de voisinage agrandi est appliquée à chaque fois que l'on ne peut trouver une solution améliorante dans le voisinage. Cependant, la taille de l'arbre à générer retourne à sa valeur initiale $k_{max}=k_0$ dès qu'une amélioration est effectuée.

Le processus de recherche de la solution globale est arrêté dans le cas où k_{max} atteint la taille du problème ($k_{max}=n$), ou un temps d'exécution T_{max} fixé préalablement est achevé.

Nous présentons dans l'algorithme ci-après, les principales étapes de la méthode utilisée pour une recherche globale de la solution du problème, en utilisant l'approche par séparation et évaluation dans un voisinage pour trouver une première amélioration, ainsi que la technique d'élargissement du voisinage pour échapper aux optima locaux.

Notation

n	Taille du problème.
T_{max}	Temps maximal d'exécution.
$f(\sigma)$	Valeur de la fonction objectif de σ ; $f(\sigma)=\Sigma_{i\in\sigma} w_i C_i$.
k_{max}	Nombre de nœuds générés à partir de chaque nœud père
k_0	Nombre initial de nœuds générés à partir de chaque nœud père

Algorithme 3.5 BB_LS /* Méthode de recherche globale

Obtenir une solution initiale σ par l'heuristique HB ;
Poser $k_{max}=k_0$;
Tant que ($k_{max}<n$) et (T_{max} n'est pas atteint) **faire**
σ'= premier_voisin(σ) ;
Si $f(\sigma') < f(\sigma)$ **alors** $\sigma=\sigma'$ et $k_{max}=k_0$
Sinon poser $k_{max}=k_{max}+[n/10]$
finsi
Fin tant que

3-8 Tests et résultats

Les tests ont été générés en faisant varier les données du problème. Les durées d'exécutions et les poids des tâches sont générés suivant une loi de distribution uniforme dans l'intervalle [1,100]. Les quantités de la ressource sont générées de la manière suivante : soit q_i^+ (et q_j^-) les quantités de ressource telles que $q_i>0$ (et $q_j<0$) avec $i=1,2,\ldots,\xi$ et $j=\xi+1,\ldots,n$, en posant $\xi=\left\lfloor \frac{n}{2} \right\rfloor$. Les quantités q_i^+ et q_j^- sont générées suivant une distribution uniforme dans l'intervalle [1,100] pour q_i^+ avec $i=1,2,\ldots,\xi$ et l'intervalle [-100,-1] pour q_j^- avec $j=\xi+1,\ldots,n$. Une fois les quantités générées; la quantité totale est calculée $Q=\Sigma q_i^+ + \Sigma q_j^-$. Pour maintenir la

faisabilité du problème; on doit avoir Q≤0, d'où un paramètre Δ est utilisé pour maintenir un degré de faisabilité du problème. Ce paramètre est choisi dans l'ensemble {-100,-50,0}. Les quantités générées q_i^+ et q_j^- sont donc réaménagées de la façon suivante :

$$pour\ i = 1,2,\cdots,\xi \qquad q_i^+ = \begin{cases} q_i^+ - \left\lfloor \frac{Q-\Delta}{\xi} \right\rfloor & si\ Q-\Delta \prec 0 \\ q_i^+ - \left\lceil \frac{Q-\Delta}{\xi} \right\rceil & si\ Q-\Delta \succ 0 \\ q_i^+ & autrement \end{cases}$$

$$pour\ j = \xi+1,\cdots,n \quad q_j^- = q_j^-$$

Notre programme est écrit en Visual C++ (Version 6.0) et implémenté sur un micro-ordinateur Intel Pentium IV 2.4 Ghz, de mémoire 128 Mo. On a généré 5 instances pour chacune des valeurs de $\Delta \in$ {-100,-50,0}, et chaque taille du problème $n \in$ {20,40,50,100,200,400,500}.

Le nombre de nœuds généré initialement pour construire l'arbre du voisinage est fixé à k_{max}=10. Le temps total pour exécuter la recherche globale est défini par T_{max}=1500 secondes. Par ailleurs, une durée d'exécution de 120 secondes est allouée à la recherche du premier voisin dans l'arbre du voisinage (i.e. si le temps alloué arrive à expiration sans aucune amélioration trouvée, la procédure de recherche d'un voisin est arrêtée).

Pour pouvoir évaluer cette méthode, nous l'avons testé en utilisant deux bornes inférieures. Le premier algorithme est celui qui calcule la borne inférieure par relaxation lagrangienne (noté par LB_RL), le second est celui utilisé par P.C.Geng [GEN84] basé sur la règle SWPT (noté par LB_SWPT). Pour chaque borne inférieure, nous avons exécuté la méthode de recherche globale sur les instances des problèmes générés.

Le tableau 3.1 montre les différentes valeurs de la fonction objectif obtenues en calculant la moyenne de 5 instances exécutées d'un problème de valeurs *n* et Δ fixes. Les temps reportés sur le tableau déterminent la durée moyenne d'exécution

de la méthode jusqu'à la dernière amélioration de la solution. Le coût moyen de la fonction objectif de la séquence initiale a été également reporté sur le tableau.

N	Δ	Solution Initiale (moyenne de 5 instances)	Recherche locale avec LB_RL		Recherche locale avec LB_SWPT		Meilleure Solution
			Solution moyenne	Temps moyen	Solution moyenne	Temps moyen	
20	-100	330 996	329 348	3.55	**329 188**	5.13	**329 188**
	-50	337 909	**329 483**	1.32	329 922	1.56	**329 483**
	0	331 780	329 761	0.76	**329 461**	0.26	**329 461**
40	-100	1 208 317	1 204 773	10.91	**1 204 603**	10.32	**1 204 603**
	-50	1 284 319	1 274 380	33.21	**1 273 146**	51.85	**1 273 146**
	0	1 143 227	1 142 089	24.42	**1 141 209**	1.40	**1 141 209**
50	-100	1 611 277	**1 608 050**	2.64	1 608 221	2.18	**1 608 050**
	-50	1 828 928	1 826 731	17.15	**1 826 700**	6.89	**1 826 700**
	0	1 831 765	1 829 482	13.28	**1 828 705**	11.04	**1 828 705**
100	-100	6 428 033	6 426 784	15.51	**6 426 758**	14.62	**6 426 758**
	-50	6 722 314	6 721 793	41.99	**6 720 679**	44.97	**6 720 679**
	0	6 923 042	6 918 970	32.50	**6 913 804**	44.42	**6 913 804**
200	-100	27 352 623	**27 352 441**	137.87	27 352 441	146.66	**27 352 441**
	-50	25 994 294	25 985 728	273.40	**25 984 426**	164.03	**25 984 426**
	0	27 453 800	**27 451 332**	160.10	27 453 755	120.10	**27 451 332**

Tableau 3.1 : résultats moyens des tests de la recherche locale par B&B

Les résultats du tableau 3.1 montrent clairement que l'application de la méthode en utilisant la borne inférieure SWPT, donne les meilleures solutions, comparées à celle utilisant la relaxation lagrangienne. Ceci peut être expliqué par la complexité des deux bornes utilisées. En effet, la borne inférieure par relaxation lagrangienne est de complexité $O(n^2 \log n)$, par contre celle qui utilise la règle SWPT est de complexité $O(n \log n)$. Cette dernière est plus rapide, facile à implémenter et nécessite peu de temps, ce qui favorise l'exploration de l'arbre B&B dans un temps raisonnable.

La complexité des deux bornes a été un facteur déterminant, malgré la qualité de la borne LB_RL par rapport à LB_SWPT, car le principe de génération de l'arbre est le même, seulement la borne LB_SWPT explore plus rapidement les nœuds, du fait de sa complexité, et atteint les feuilles pour déterminer les voisins dans un temps plus court. Ainsi, dans un temps de recherche fixé, l'espace des nœuds de l'arbre couvert par la borne LB_RL est plus petit que celui couvert par LB_SWPT, ce qui favorise l'utilisation de LB_SWPT à trouver une amélioration.

Une exception est notée pour les problèmes de petite taille (n<20), où les deux bornes ont donné pratiquement la même solution. Ceci est expliqué par la taille de l'arbre généré durant la recherche du voisinage. En effet, les deux bornes arrivent à explorer l'arbre dans un temps raisonnable, et par conséquent donnent les mêmes solutions. Mais, nous devons signaler que la borne LB_SWPT reste toujours la meilleure car elle donne la solution en un temps inférieur par rapport à la borne LB_RL. Encore une fois, ceci est dû à la complexité de la borne LB_RL.

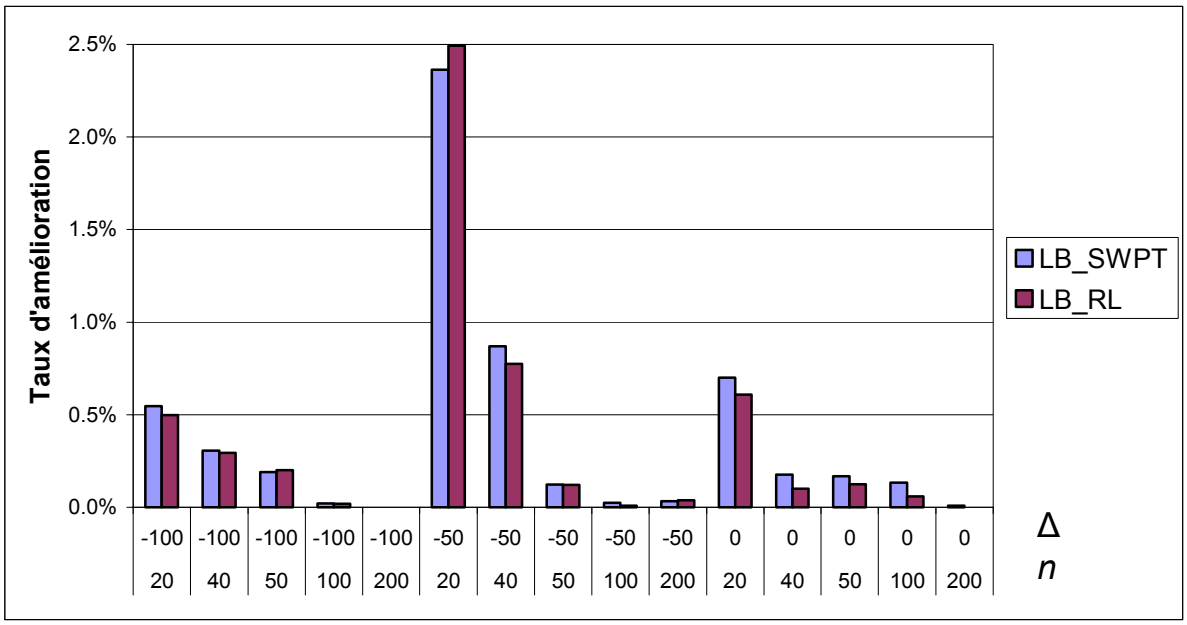

Figure 3.2 : Taux d'amélioration des solutions par la recherche locale avec B&B.

La figure 3.2 montre les différents taux d'amélioration réalisés durant l'exécution de la recherche locale basée B&B en utilisant les deux bornes inférieures LB_SWPT et LB_RL. Le taux d'amélioration est calculé sur la base de la solution initiale par la formule suivante:

T_a = (Solution initiale - Solution trouvée)/Solution initiale

Il est aisé de remarquer que la majorité des problèmes ont été améliorés par les deux bornes proposées. Cette amélioration qui se situe entre 0% et 2.5% est significative pour la borne LB_SWPT, mais les taux d'amélioration des deux bornes sont très proches. Ceci peut être expliqué par l'efficacité de la méthode utilisée (i.e., la recherche locale basée B&B) ainsi que la performance des deux bornes.

Concernant les temps d'exécution, les résultats montrent qu'en général, plus les problèmes grandissent en taille, plus ils sont difficiles, ceci se traduit par l'augmentation des temps d'exécution selon le nombre de tâches. Aussi, nous constatons qu'il n'y a pas une grande différence entre les deux méthodes, concernant les durées d'exécution. Ceci peut se traduire par la difficulté d'exploration de l'arbre qui est généralement de taille exponentielle, notamment lorsque la taille du problème dépasse les 100 tâches. On remarque d'ailleurs que la majorité des améliorations a été réalisée dans les premières itérations, car plus la taille de l'arbre augmente lorsqu'il n'y a pas d'améliorations, plus il est difficile d'explorer suffisamment l'arbre pour avoir un voisin qui améliore la solution.

Par ailleurs, nous devons signaler que pour les problèmes de taille importante (n>200), les deux bornes utilisées semblent ne donner aucun succès. Ceci est dû essentiellement à la taille trop grande de l'arbre de voisinage, ce qui rend difficile son exploration en un temps raisonnable pour avoir une solution améliorante. Cependant, la performance et la qualité des bornes implémentées jouent aussi un grand rôle dans l'exploration rapide de l'arbre et par conséquent l'accès dans un temps suffisant à une séquence qui améliore la solution.

3-9 Conclusion

Nous avons présenté une méthode de recherche locale basée sur la génération d'un voisinage en utilisant l'approche par séparation et évaluation. Pour trouver une solution globale, nous avons utilisé la technique d'élargissement du voisinage en augmentant la taille de l'arbre généré.

Différentes comparaisons ont été réalisées, en utilisant deux méthodes de calcul de la borne inférieure, nécessaire à la construction de l'arbre du voisinage. Les résultats ont montré que l'utilisation de la borne basée sur la règle SWPT donne, pour notre problème, les meilleures solutions.

L'exploration de l'arbre du voisinage semble être de plus en plus difficile lorsque la taille des problèmes augmente. De plus, la qualité de la borne inférieure, sa complexité et sa performance sont des facteurs importants dans la recherche dans l'arbre d'une solution améliorante. Pour cette raison, les deux bornes que nous avons proposées n'ont pas pu donner de meilleures solutions pour des problèmes de taille importante. Nous estimons qu'une borne inférieure rapide et de meilleure qualité peut arriver, dans un temps suffisant, à donner de meilleures solutions pour des problèmes de taille suffisamment large.

4 Résolution du problème par la recherche tabou

4-1 Introduction

La recherche tabou est une métaheuristique qui combine une procédure de recherche locale avec un certain nombre de règles et de mécanismes permettant à celle-ci de surmonter l'obstacle des optima locaux, tout en évitant de cycler (cf. chap.2, section 4-1-4).

Nous allons présenter dans les sections suivantes, la résolution du problème d'ordonnancement sur machine sous contrainte de ressource, en utilisant la recherche tabou. La section 2 fournit une description des voisinages générés. Nous avons proposé deux techniques de définition du voisinage, l'une utilise la permutation de deux tâches quelconques (voisinage par swapping), tandis que l'autre consiste à insérer une tâche dans une position de la séquence (voisinage par insertion).

La section 3 présente la structure de la liste taboue utilisée dans la méthode, et la section 4 propose une nouvelle méthode de diversification pour améliorer la recherche. Ensuite, la section 5 fournit les grandes étapes de la procédure de recherche tabou ainsi utilisée.

Enfin, dans la section 6, nous donnons les résultats et commentaires des tests effectués, avant de clôturer avec une conclusion sur la performance de cette méthode à la résolution de notre problème.

4-2 Structures du voisinage

La définition d'un voisinage constitue l'étape la plus importante dans l'élaboration des méthodes métaheuristiques. En effet, la connaissance de l'ensemble des voisins d'une séquence permet d'en sélectionner le meilleur qui apporte une amélioration de la solution.

On doit noter, par ailleurs, que plus le voisinage considéré est grand, plus la chance d'obtenir une solution optimale est grande, mais par contre, plus les calculs sont importants. L'expérience a prouvé qu'il est beaucoup plus avantageux de ne considérer que des voisinages très restreints quitte à recommencer plusieurs fois la procédure à partir de solutions initiales différentes [SAK84].

Nous allons présenter dans ce qui suit, deux méthodes de définition de voisinage pour notre problème, afin de les comparer en les intégrant dans la recherche tabou, puis d'en retenir le meilleur qui permet d'améliorer la solution.

Le premier voisinage consiste à utiliser la technique de permutation de deux tâches quelconques de la séquence. Le deuxième utilise le principe d'insertion d'une tâche dans une position de la séquence.

4-2-1 Voisinage par swapping

Définition **:** Etant donnée une séquence σ constituée de n tâches, un voisin σ' est obtenu par permutation de deux tâches i et j avec i=1..n et j=1..n. Cette opération s'appelle : permutation simple de deux tâches (en anglais : single pairwise interchange ou swapping).

L'ensemble N(σ)={σ', σ' obtenu par permutation de i et j} s'appelle le voisinage de σ. Cet ensemble est donc obtenu par permutation de toutes les tâches de σ deux à deux.

La taille du voisinage par swapping est déterminée comme suit :

Observation 2: Etant donnée une séquence σ, la cardinalité de l'ensemble N(σ) voisinage de σ est $n(n-1)/2$.

Preuve : La permutation de toutes les tâches deux à deux consiste à prendre chaque tâche de la séquence et de la permuter avec toutes les tâches restantes sans avoir deux séquences identiques. Le nombre de permutations possibles dans une séquence σ constituée de n tâches est donc : $(n-1)+(n-2)+\ldots+2+1=n(n-1)/2$.

Exemple : Soit une séquence σ=1234, le voisinage N(σ) est :

N(σ)={2134,3214,4231,1324,1432,1243}, voir tableau 3.2.

Tâche *i*	Tâche *j*	Séquence obtenue
1	2	2134
	3	3214
	4	4231
2	3	1324
	4	1432
3	4	1243

Tableau 3.2 : Exemple de construction d'un voisinage par swapping

4-2-2 Voisinage par insertion

***Définition* :** Etant donnée une séquence σ constituée de *n* tâches, un voisin σ' est obtenu par la suppression de la tâche *i* d'une position *k* puis son insertion dans une nouvelle position *k'*.

L'ensemble N(σ)={σ', σ' obtenu par insertion de *i* dans *j*} s'appelle le voisinage de σ. Cet ensemble est donc obtenu en effectuant toutes les insertions possibles des tâches de σ.

La taille du voisinage par insertion est déterminée comme suit :

Observation 3: Etant donnée une séquence σ, la cardinalité de l'ensemble N(σ) est égale à $(n\text{-}1)^2$.

Preuve : L'insertion d'une tâche *i* d'une position *k* dans une autre position *k*', permet d'avoir *n*-1 insertions possibles, donc pour *n* tâches, il y a $n(n\text{-}1)$ insertions à effectuer. Pour empêcher d'avoir des séquences identiques, les insertions des tâches adjacentes sont comptées une seule fois, par conséquent, *n*-1 insertions sont supprimées. Finalement, le nombre d'insertions effectuées est:

$n(n\text{-}1)\text{-}(n\text{-}1)=(n\text{-}1)^2$.

Exemple : Soit une séquence σ=1234, le voisinage N(σ) est :

N(σ)={2134,2314,2341,1324,1342,3124,1243,4123,1423}, voir tableau 3.3

Position / Tâche	1	2	3	4
1		2134	2314	2341
2	~~2134~~		1324	1342
3	3124	~~1324~~		1243
4	4123	1423	~~1243~~	

Tableau 3.3 : Exemple de construction d'un voisinage par insertion

4-3 Structure de la liste taboue

La méthode tabou s'appuie sur le principe qui consiste à garder en mémoire les dernières solutions visitées et à interdire le retour à celles-ci pour un nombre fixe d'itérations, le but étant de donner assez de temps à l'algorithme pour lui permettre de sortir d'un minimum local. En d'autres termes, la méthode tabou conserve à chaque étape une liste T de solutions « taboues », vers lesquelles il est interdit de se déplacer momentanément. L'espace nécessaire pour enregistrer un ensemble de solutions taboues peut s'avérer important en place mémoire.

La liste taboue que nous proposons, contient les séquences des solutions trouvées. Après plusieurs essais, nous avons conçu une liste de taille dynamique qui varie selon l'état d'amélioration de la recherche. La taille de la liste est initialement fixée à $\frac{3\sqrt{n}}{2}$, ensuite durant la recherche, lorsque 5 itérations successives sont effectuées sans amélioration de la solution, la liste est réduite, libérant ainsi des solutions taboues pour sortir d'un minimum local, la liste est ainsi réduite jusqu'à une limite inférieure égale à $\sqrt{n}$. Par contre, lorsque 5 itérations successives sont effectuées avec amélioration de la solution, la taille de la liste est augmentée, jusqu'à une limite supérieure égale à $2\sqrt{n}$. La liste taboue est donc dynamique et sa taille varie dans l'intervalle $[\sqrt{n}, 2\sqrt{n}]$, la réduction ou l'augmentation de la taille se fait toujours à la fin de la liste.

4-4 Stratégie de diversification

La diversification permet à la méthode de bien explorer l'espace des solutions, et d'éviter que le processus de recherche ne soit trop localisé et laisse de grandes régions du domaine totalement inexplorées.

La stratégie de diversification la plus simple consiste à redémarrer périodiquement le processus de recherche à partir d'une solution générée aléatoirement ou choisie judicieusement dans une région non encore visitée de l'ensemble des solutions admissibles [WID01].

Nous allons présenter dans ce qui suit une nouvelle technique de diversification basée sur le redémarrage de la recherche à partir d'une solution choisie judicieusement.

Cette technique, baptisée « stratégie des *k*-premières solutions » (en anglais : *k*-first strategy), est basée sur le principe de mémorisation dans une liste triée, des *k* premières meilleures solutions non encore visitées, qui ne sont ni taboues, ni des solutions que le processus de recherche les a retenues. Ces solutions représentent des séquences trouvées pendant la recherche dans le voisinage, mais ne sont pas retenues comme des solutions car ne sont pas améliorantes, elles appartiennent donc à des régions non encore explorées. Le principe consiste à choisir parmi ces solutions, celles qui sont juste au-dessus des solutions retenues, c'est-à-dire les premières meilleures, et de les sauvegarder dans une liste qui doit être mise à jour continuellement.

Le principe de construction de la liste *k*-first est expliqué dans ce qui suit : Initialement la liste est vide, ensuite lors de la recherche dans le voisinage, chaque solutions non taboue est comparée aux éléments de la liste. Si la solution peut être insérée, un décalage à droite est effectué à partir de la position d'insertion, sinon aucune mise à jour de la liste n'est faite. A la fin de la recherche dans le voisinage la liste contient les *k* premières meilleures solutions. La méthode taboue permet de prendre la meilleure solution du voisinage, cette solution est supprimée de la liste *k*-first si elle y existe. Ainsi, à la fin de chaque itération, la liste *k*-first contient les *k* premières meilleures solutions non encore visitées.

La stratégie de diversification que nous avons utilisée, consiste à prendre une solution de la liste *k*-first (généralement la première car la meilleure) et de redémarrer une nouvelle recherche taboue à partir de cette solution (i.e., construction de voisinage de cette solution). Cette démarche est réalisée lorsque 5 itérations successives de la recherche taboue n'ont pas amélioré la solution du problème.

La taille de la liste *k*-first doit être raisonnable, du fait de sa mise à jour en continu durant la recherche dans le voisinage. Après plusieurs essais, nous avons fixé la taille de la liste *k*-first à $\frac{\sqrt{n}}{2}$.

La procédure de mise à jour de la liste *k*-first est appelée par la procédure de génération du voisinage. Les algorithmes correspondants sont décrits comme suit :

Procedure 3.6 Genere_V (σ) ; /* procédure de génération du voisinage
$F_{\min}$ coût minimal dans tout le voisinage
répéter
Construire un voisin σ' par swapping ou insertion;
Si σ' ∉ TABOU
Calculer $F = f(\sigma')$;
Actualiser $F_{\min}$;
Update_klist(F, σ') ;
finsi
Jusqu'à conditions d'arrêt satisfaites

Procedure 3.7 Update_klist (H,π) ; /* Mise à jour de *k*_list
Pour *i*=1 à taille(klist) faire
Si $H <$ klist[*i*]
Inserer (H,π) dans la position *i* ;
STOP ;
Fin si
Fin pour

4-5 Procédure de la recherche tabou

La méthode de recherche tabou que nous proposons consiste à générer un voisinage N(σ) par l'une des techniques mentionnées plus haut (cf. 4-2) et retenir la meilleure solution σ' parmi celles de N(σ), même si σ' est plus mauvaise que σ. Ensuite, la solution σ' retenue est mémorisée dans la liste taboue. Par ailleurs, on vérifie si σ' est meilleure que σ, dans ce cas, la solution σ' devient la meilleure. Une autre itération est ensuite exécutée à partir de cette solution σ', et ce, jusqu'à une condition d'arrêt, choisie pour notre procédure par un temps global d'exécution T_{max}.

Pour démarrer avec une séquence initiale dans la recherche tabou, nous avons repris l'heuristique HB déjà utilisée dans la résolution du problème par une recherche locale (cf. 3-6).

Nous présentons dans ce qui suit, l'algorithme qui décrit les grandes étapes de la méthode de recherche tabou que nous avons conçue. Nous commençons par les principales notations utilisées, puis nous donnons les différentes instructions de l'algorithme.

Notation

T_{max}	Temps maximal d'exécution.
$f(\sigma)$	coût de la fonction objectif de σ ; $f(\sigma)=\Sigma_{i\in\sigma} w_i C_i$.
f_{min}	coût minimal.
σ_{min}	solution de coût minimal.
TABOU	Liste taboue.
LenTabu	Taille de la liste TABOU
kfirst	Liste kfirst.
MaxTabu	Taille maximale de la liste TABOU
MinTabu	Taille minimale de la liste TABOU

Algorithme 3.8 RTabou /* Méthode de recherche tabou

Obtenir une solution initiale σ par l'heuristique HB ;
$f_{min}= f(\sigma)$, $\sigma_{min}= \sigma$, TABOU=Ø;
MaxTabu=2*SQRT(*n*) ; MinTabu=SQRT(*n*) ;
Tant que (T_{max} n'est pas atteint) **faire**
Appeler Gsenere_V(σ) ;
Prendre σ' telle que $f(\sigma') = \min_{\xi \in N(\sigma)} f(\xi)$ et $\sigma' \notin$ TABOU ;
Si $f(\sigma') < f_{min}$ **alors** $f_{min}= f(\sigma')$, $\sigma_{min}= \sigma'$ **fin si ;**
Si (nombre d'itérations successives d'amélioration=5)
Taille(TABOU)=min(LenTabu+1,MaxTabu) ;
Fin si
Si (nombre d'itérations successives de non amélioration=5)
Taille(TABOU)=max(LenTabu-1,MinTabu) ;
Prendre σ = premier élément de klist ; /*Diversification
Fin si
TABOU=TABOU ∪ {σ'} ;
Fin tant que

4-6 Tests et résultats

Les tests que nous avons réalisés ont été générés en faisant varier les données du problème. Les durées d'exécutions et les poids des tâches sont générés suivant une loi de distribution uniforme dans l'intervalle [1,100]. Les quantités de la ressource sont générées de la même manière que celle utilisée dans les tests de la méthode par recherche locale (cf. 3-8).

Notre programme est écrit en Visual C++ (Version 6.0) et implémenté sur un micro-ordinateur Intel Pentium IV 2.4 Ghz, de mémoire 128 Mo. On a généré 5 instances pour chacune des valeurs de $\Delta \in$ {-100,-50,0}, et chaque taille du problème $n \in$ {20,40,100,200,400,500}.

Le temps total pour exécuter la recherche globale est défini par T_{max}=1500 (secondes). Le temps de recherche pour définir un voisinage et déterminer la solution minimale, est fixé à t = 120 secondes (i.e. si le temps alloué arrive à

expiration, la procédure de recherche d'un voisin est arrêtée). Ce temps fixé permet d'avoir un voisinage restreint si on n'arrive pas à définir tous les éléments dans cette durée. Les différents tests effectués ont montré que la taille du voisinage généré pendant cette durée et dans la configuration informatique que nous avons décrite, se situe entre 100 et 19 000 séquences admissibles, cela varie effectivement selon la taille du problème et son degré de difficulté.

Pour pouvoir évaluer cette méthode, nous l'avons testé en utilisant deux voisinages différents : le voisinage par swapping (noté V_Swap) et celui par insertion (noté V_Inser) (cf. 4-2). Pour chaque voisinage, nous avons exécuté la méthode de recherche tabou sur les instances des problèmes générés.

Le tableau 3.4 montre en moyenne les différentes valeurs de la fonction objectif obtenues en utilisant V_Swap et V_Inser. Les temps reportés sur le tableau déterminent la durée d'exécution moyenne de 5 instances jusqu'à la dernière amélioration de la solution. Le coût de la fonction objectif de la séquence initiale a été également reporté sur le tableau.

Les résultats du tableau 3.4 montrent que l'exécution de la méthode tabou en utilisant le voisinage par swapping a donné les meilleures solutions pour les 2/3 des problèmes (70 instances), comparées à celle utilisant le voisinage par insertion qui a donnée des meilleures solutions au 1/3 des problèmes (35 instances). Ceci peut être expliqué par la nature des voisinages utilisés, ainsi que les données du problème. En effet, le swapping constitue une permutation simple de deux tâches quelconques, par contre l'insertion est réalisée, non seulement par des mouvements de permutation (pour les tâches extrêmes), mais aussi par des décalages à droites et à gauches (pour les autres tâches). Le temps de génération d'un voisin est plus court par swapping que par insertion, par conséquent, pour un temps fixe, la taille du voisinage généré par swapping est généralement plus importante que celle du voisinage par insertion, ce qui favorise la recherche d'une solution améliorante.

Néanmoins, les deux voisinages utilisés ont contribué efficacement à l'amélioration de la solution initiale, et l'on ne peut affirmer la bon ou le mauvais voisinage, car ceci dépend généralement de la nature des données du problème.

N	Δ	Solution Initiale (moyenne de 5 instances)	Recherche tabou par V_Swap		Recherche tabou par V_Inser		Meilleure Solution
			Solution moyenne	Temps Moyen	Solution moyenne	Temps Moyen	
20	-100	330 996	**327 224**	1.31	330 866	0.05	**327 224**
	-50	337 909	331 961	0.32	**330 690**	1.21	**330 690**
	0	331 780	**327 450**	0.98	327 474	0.97	**327 450**
40	-100	1 208 317	**1 203 874**	7.55	1 204 375	2.10	**1 203 874**
	-50	1 284 319	**1 258 277**	11.57	1 266 016	12.89	**1 258 277**
	0	1 143 227	**1 139 804**	10.26	1 140 136	12.98	**1 139 804**
50	-100	1 611 928	**1 608 571**	1.70	1 610 027	2.52	**1 608 571**
	-50	1 828 928	1 827 347	1.99	**1 826 462**	6.31	**1 826 462**
	0	1 831 765	1 829 556	73.50	**1 828 571**	7.93	**1 828 571**
100	-100	6 428 033	**6 427 114**	78.30	6 427 728	36.32	**6 427 114**
	-50	6 722 314	**6 719 096**	173.07	6 719 914	109.04	**6 719 096**
	0	6 923 042	**6 908 702**	255.63	6 909 490	187.60	**6 908 702**
200	-100	27 352 623	27 347 372	435.53	**27 347 107**	399.64	**27 347 107**
	-50	25 994 294	**25 986 139**	296.30	25 986 259	254.54	**25 986 139**
	0	27 453 800	27 446 287	444.88	**27 444 506**	253.27	**27 444 506**
400	-100	109 427 447	109 425 155	604.40	**109 424 364**	753.14	**109 424 364**
	-50	108 521 160	**108 516 428**	387.43	108 518 019	459.86	**108 516 428**
	0	113 416 582	**113 407 963**	967.47	113 412 508	871.47	**113 407 963**
500	-100	173 892 004	173 890 994	605.49	**173 890 861**	636.09	**173 890 861**
	-50	176 251 087	**176 249 470**	909.21	176 249 639	639.61	**176 249 470**
	0	157 908 557	**157 904 808**	1277.79	157 906 545	1278.19	**157 904 808**

Tableau 3.4 : Résultats moyens des tests de la méthode tabou.

Nous remarquons par ailleurs, en examinant la figure 3.3 que le taux d'amélioration de la solution initiale en utilisant les deux voisinages n'est pas grand. En effet, ce taux varie en moyenne entre 0.01% et 2.5%. De plus, ces taux montrent que les problèmes de petites tailles (n<200) ont subi des améliorations significatives, par contre les problèmes de tailles importantes ne sont améliorés que d'une façon minime, ceci est traduit par le voisinage restreint lorsque *n* augmente, dû essentiellement au temps alloué de recherche dans le voisinage qui est fixe.

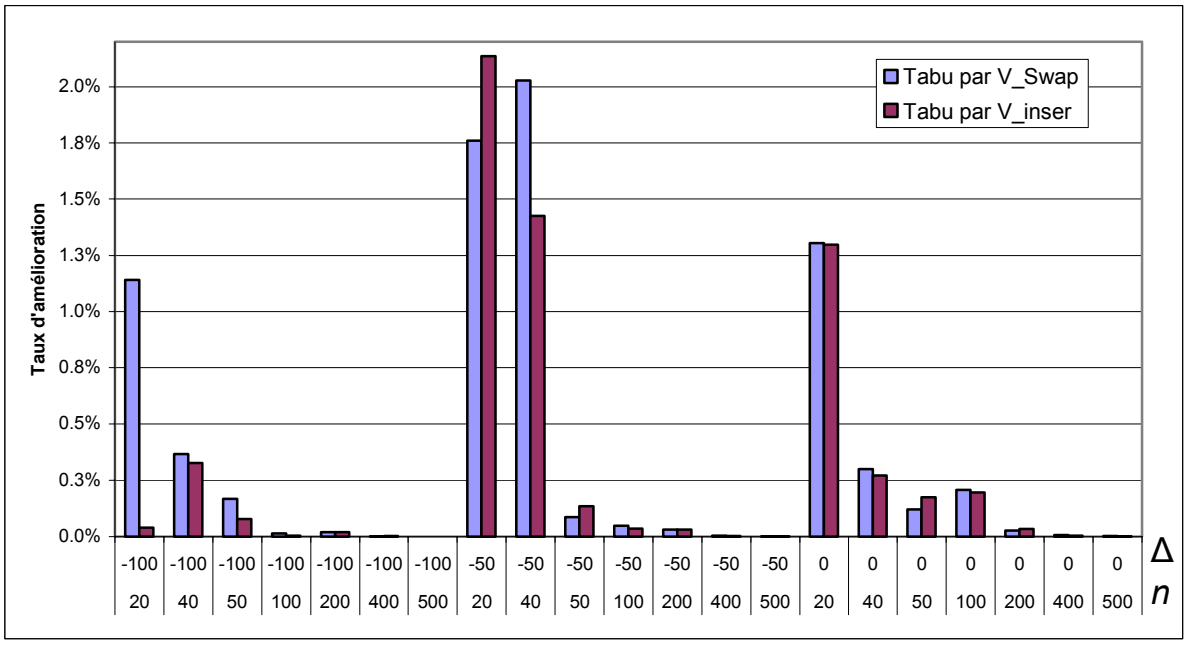

Figure 3.3 : Taux d'amélioration par la méthode tabou

Concernant les temps d'exécution, la figure 3.4 montre qu'en général, les deux méthodes achèvent leur exécution en des temps très proches. On peut remarquer également que les temps d'exécution augmentent avec la valeur de delta qui définit le degré de faisabilité des problèmes générés.

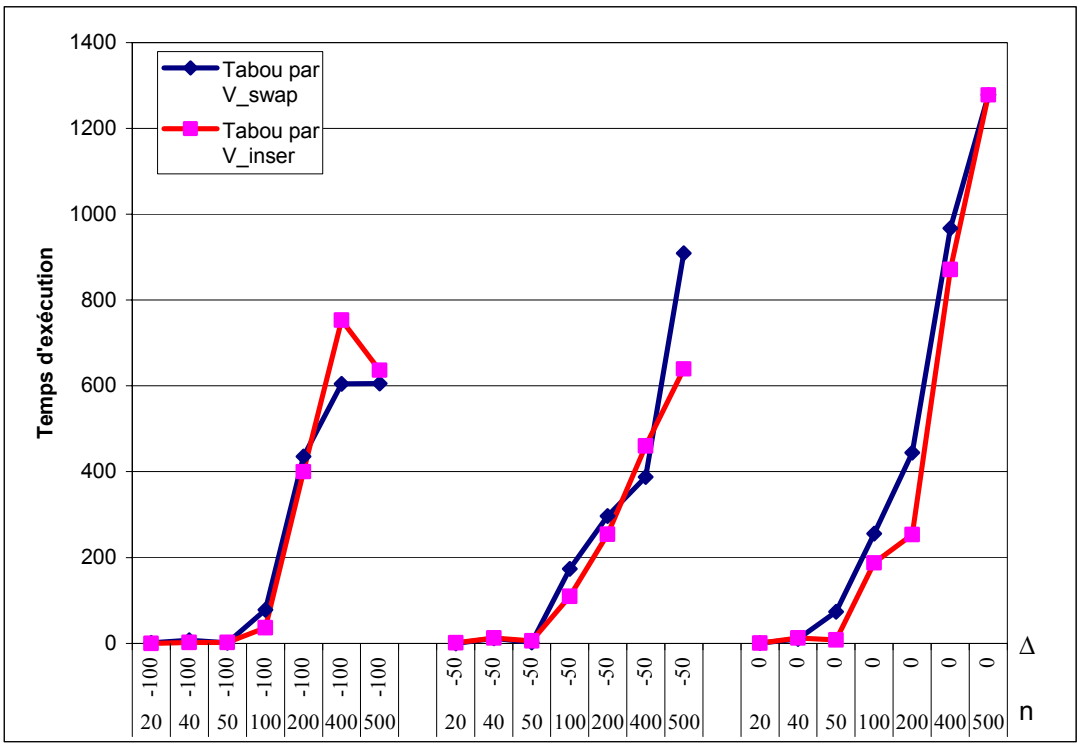

Figure 3.4 : Temps d'exécution par la méthode tabou.

4-7 Conclusion

Nous avons présenté une méthode de recherche tabou pour résoudre le problème d'ordonnancement sur machine en présence d'une ressource non renouvelable. La liste taboue que nous avons conçue est dynamique; la taille de la liste varie selon l'état d'amélioration de la solution. Une technique de diversification a été proposée, basée sur le redémarrage de la recherche à partir d'une solution choisie parmi les premières meilleures solutions trouvées au cours de la recherche et qui n'ont pas été retenues, ceci revient à visiter les régions non encore explorées.

Différents tests ont été effectués, en utilisant deux stratégies de voisinage, l'un basé sur le swapping, tandis que l'autre utilise la technique d'insertion. Les résultats ont montré que les deux voisinages contribuent conjointement à l'amélioration de la solution, mais le voisinage par swapping semble donner de meilleurs résultats pour notre problème. Néanmoins, les solutions améliorées par les deux techniques sont très proches.

Les améliorations faites sur les problèmes de grandes tailles ont été minimes pour les deux voisinages utilisés, la raison principale à cela, est la taille du problème qui a empêché la définition complète des éléments du voisinage. La recherche dans un voisinage restreint lié à un temps d'exécution fixe, a freiné l'amélioration de la solution. Nous estimons que le choix d'un bon voisinage qui permet d'avoir le plus de voisins possibles aura certainement de meilleurs résultats.

5 Etude comparative des méthodes utilisées

Nous allons présenter dans cette section, une comparaison des méthodes utilisées dans la résolution de notre problème. Plus précisément, nous allons comparer les deux meilleures approches retenues dans les tests et résultats précédents des deux méthodes, à savoir la méthode de recherche locale basée sur la technique branch-and-bound en utilisant la borne inférieure SWPT, et la méthode de recherche tabou en utilisant le voisinage par swapping.

Ayant généré les données du problème de la même façon que pour les tests précédents (cf. 3-8 et 4-6), nous avons exécuté en premier lieu la méthode de recherche locale basée B&B avec la borne inférieure SWPT, puis avec les mêmes données du problème, nous avons exécuté la méthode de recherche tabou.

Le temps total pour exécuter la recherche globale pour les deux méthodes est défini par T_{max}=1500 secondes. Par ailleurs, une durée d'exécution de 120 secondes est allouée à la recherche dans le voisinage, et ce pour les deux méthodes.

Le tableau 3.5 montre en moyenne les différentes valeurs de la fonction objectif obtenues en utilisant les deux méthodes sus-mentionnées. Les temps reportés sur le tableau déterminent la durée d'exécution de la méthode jusqu'à la dernière amélioration de la solution. Le coût de la fonction objectif de la séquence initiale a été également reporté sur le tableau.

n	Δ	Solution Initiale (moyenne de 5 instances)	Recherche locale basée B&B avec SWPT		Recherche tabou avec voisinage V_swap		Meilleure Solution
			Solution moyenne	Temps moyen	Solution moyenne	Temps moyen	
20	-100	330 996	329 188	5.13	**327 224**	1.31	**327 224**
	-50	337 909	**329 922**	1.56	331 961	0.32	**329 922**
	0	331 780	329 461	0.26	**327 450**	0.98	**327 450**
40	-100	1 208 317	1 204 603	10.32	**1 203 874**	7.55	**1 203 874**
	-50	1 284 319	1 273 146	51.85	**1 258 277**	11.57	**1 258 277**
	0	1 143 227	1 141 209	1.40	**1 139 804**	10.26	**1 139 804**
50	-100	1 611 277	**1 608 221**	2.18	1 608 571	1.70	**1 608 221**
	-50	1 828 928	**1 826 700**	6.89	1 827 347	1.99	**1 826 700**
	0	1 831 765	**1 828 705**	11.04	1 829 556	73.50	**1 828 705**
100	-100	6 428 033	**6 426 758**	14.62	6 427 114	78.30	**6 426 758**
	-50	6 722 314	6 720 679	44.97	**6 719 096**	173.07	**6 719 096**
	0	6 923 042	6 913 804	44.42	**6 908 702**	255.63	**6 908 702**
200	-100	27 352 623	27 352 441	146.66	**27 347 372**	435.53	**27 347 372**
	-50	25 994 294	**25 984 426**	164.03	25 986 139	296.30	**25 984 426**
	0	27 453 800	27 453 755	120.10	**27 446 287**	444.88	**27 446 287**

Tableau 3.5 : Résultats moyens des tests (recherche locale par B&B et recherche tabou).

Les résultats du tableau 3.5 montrent d'une façon remarquable que l'exécution de la méthode tabou en utilisant le voisinage par swapping, donne pour la majorité des problèmes (66%) les meilleures solutions, comparées à celles obtenues par la recherche locale basée B&B. La raison principale de ce résultat est la stratégie de recherche de la méthode elle-même. En effet, la méthode de recherche locale utilisée est une méthode de descente itérative qui permet de sortir des minima locaux par la technique de relance ainsi que l'élargissement du voisinage, contrairement à la recherche tabou qui utilise un mécanisme de mémorisation des solutions pour ne pas y revenir ainsi qu'une technique de diversification très efficace.

Par ailleurs, les voisinages générés dans les deux méthodes se basent sur des mécanismes très différents; le voisinage construit à partir d'un arbre régi par les règles de séparation et évaluation est sans doute plus performant mais également plus coûteux en termes de ressources informatiques et de temps. La taille de l'arbre généré occupe souvent un espace mémoire très grand, et son exploration, même partiellement, nécessite un temps plus long qui dépasse fréquemment le temps alloué à la recherche du voisinage. Par contre, le voisinage généré dans la méthode tabou est plus simple et n'occupe pratiquement que peu de mémoire, de plus, l'ensemble des voisins est déterminé presque en totalité notamment dans les problèmes de petite taille, ce qui favorise largement l'amélioration de la solution.

La figure 3.5 présente l'état des améliorations apportées par les deux méthodes. Nous pouvons constater que pour les problèmes de petite taille (jusqu'à 100 tâches), les deux méthodes participent conjointement à l'amélioration de la solution. La méthode tabou est plus performante lorsque les problèmes grandissent en taille, le voisinage utilisé dans cette méthode joue effectivement un grand rôle dans la détermination rapide d'une amélioration, contrairement à la recherche locale qui se trouve rapidement bloquée dans une solution moins bonne, principalement lorsque l'arbre généré du voisinage est élargi.

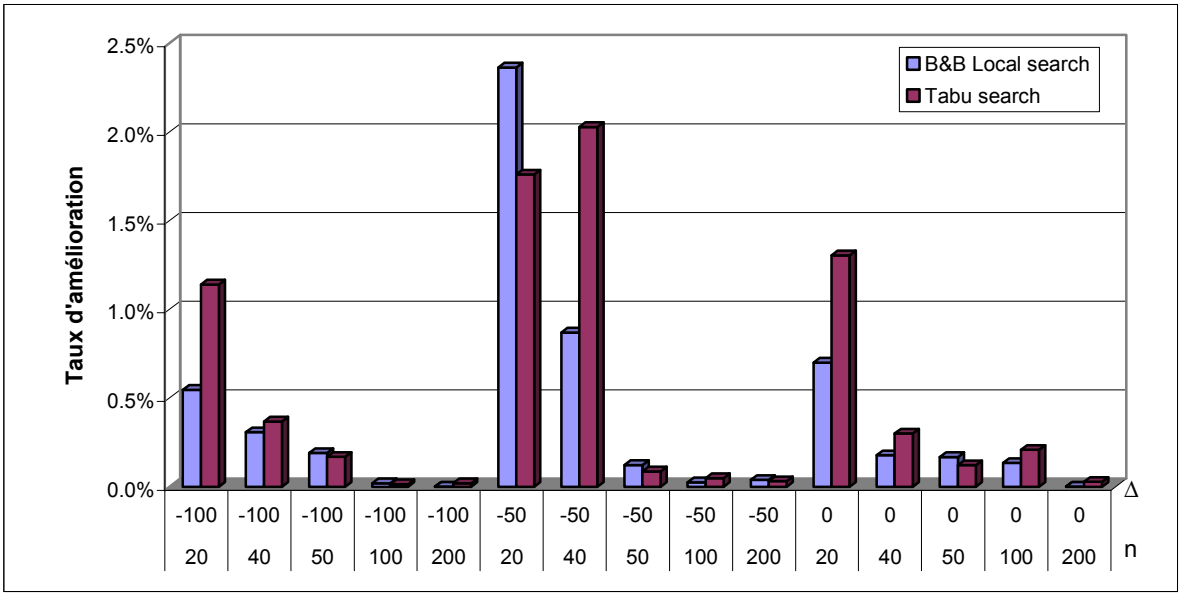

Figure 3.5 : Taux d'amélioration par les métaheuristiques utilisées.

Quant aux temps d'exécution, on remarque sur la figure 3.6 que la recherche locale est plus rapide que la méthode tabou, en effet, la technique par séparation et évaluation en utilisant une borne inférieure simple et rapide, est plus performante lorsque la taille du problème est petite, de telle sorte que la méthode arrive souvent à trouver une solution améliorante, contrairement à la recherche tabou qui doit effecteur plusieurs itérations pour trouver une amélioration. Cependant, la solution finale de la méthode tabou est presque toujours la meilleure, car celle-ci utilise d'autres stratégies d'amélioration comme la liste taboue et la diversification, alors que la recherche locale ne fait que relancer la recherche et élargir l'arbre du voisinage, ce dernier qui se trouve rapidement trop large à explorer pour en tirer une solution améliorante.

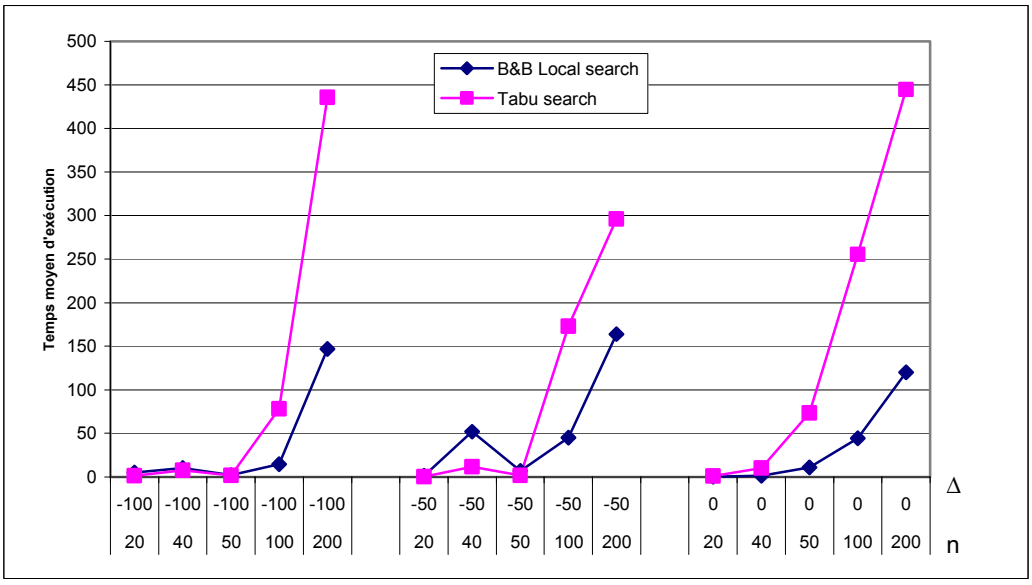

Figure 3.6 : Temps d'exécution des méthodes utilisées.

Les métaheuristiques posent un problème non encore résolu d'une manière satisfaisante : celui de la comparaison des heuristiques qu'elles engendrent. En effet, ces heuristiques sont itératives, ce qui veut dire que plus longtemps on les laisse travailler, meilleures sont les solutions qu'elles produisent. La comparaison d'heuristiques itératives doit se faire sur la qualité des solutions produites en fonction du temps d'exécution sur la machine [TAI01]. Les résultats des tests que nus avons mené ont montré la performance de la recherche tabou par rapport à la recherche locale, notamment en qualité de solutions. En fait, ces résultats ne sont pas surprenants; plusieurs expériences et résultats publiés montrent la qualité

supérieure des solutions issues de la recherche tabou par rapport à la recherche locale [HAO99].

6 Conclusion

En arrivant à la fin de ce chapitre, il est utile de noter que les métaheuristiques que nous avons utilisées pour résoudre notre problème sont des méthodes approchées, elles ne garantissent pas de trouver une solution optimale ni de prouver l'optimalité d'une solution trouvée. Toutes fois, certaines métaheuristiques sont plus performantes que d'autres, les différences principales entre ces méthodes se situent au niveau du choix de la solution voisine et du critère d'arrêt.

Le paramètre du temps utilisé dans toutes nos méthodes est un facteur très important qui peut altérer la qualité de la solution. D'autres paramètres et choix liés à chaque méthode sont également d'importants ingrédients.

La définition du voisinage pour notre problème a été le principal ingrédient et a favorisé presque totalement une méthode sur une autre, en particulier la recherche tabou en utilisant un ensemble restreint du voisinage par swapping, d'autres part, l'approche par séparation et évaluation utilisé dans le voisinage de la recherche locale dépendait exponentiellement de la taille du problème et de la qualité de la borne inférieure.

Enfin, nous devons noter qu'il est devenu possible de résoudre des problèmes d'optimisation combinatoire de taille intéressante, en utilisant les métaheuristiques. Pour notre problème, l'utilisation de la recherche tabou, en fixant une durée d'exécution totale suffisamment large et un temps raisonnable de recherche dans le voisinage, a certainement contribué à une meilleure amélioration de la solution initiale.

CONCLUSION GENERALE

CONCLUSION GENERALE

Dans ce mémoire, nous avons abordé un nouveau problème sur machine, qui consiste à ordonnancer des tâches sous la contrainte d'une ressource non renouvelable, dans l'objectif de minimiser la somme pondérée des dates de fin d'exécution.

Une première étude s'est focalisée sur la résolution de ce problème par une recherche locale à voisinage obtenu par les techniques de séparation et évaluation. Cette méthode a permis de résoudre notre problème en améliorant la solution initiale obtenue par une heuristique, toutes fois, l'arbre du voisinage généré se trouve rapidement de taille importante lorsque l'amélioration est difficile, ce qui a rendu cette méthode moins appropriée, et même ne donnant aucune amélioration pour les problèmes à très grande taille.

Ensuite, le problème est abordé en utilisant la méthode de recherche tabou. En proposant deux stratégies de voisinage pour permettre une évaluation de cette méthode et en mettant en œuvre une liste taboue dynamique ainsi qu'une technique de diversification appropriée; la recherche tabou a donné des solutions de meilleure qualité, notamment lorsque le voisinage par swapping est utilisé.

Pour conclure notre étude, une comparaison des deux métaheuristiques a été réalisée, mettant en évidence pour notre problème, l'efficacité de la recherche tabou par rapport à la recherche locale à voisinage basée B&B.

L'étude des problèmes d'ordonnancement sous les contraintes des ressources non renouvelables présente un champ vaste et étendu. Plusieurs problèmes classiques d'ordonnancement auxquels sont ajoutées les contraintes d'une ou plusieurs ressources non renouvelables peuvent être posés et ensuite abordés.

Nous suggérons pour le problème étudié le long de ce mémoire, une amélioration de la méthode de recherche locale à voisinage par B&B, en proposant une borne inférieure plus rapide et de meilleure qualité, ainsi qu'un voisinage qui

varie plus lentement pour ne pas avoir un arbre de taille importante. Cette méthode basée sur l'approche par séparation et évaluation semble avoir des facteurs de qualité qui peuvent la rendre efficace en faisant les bons choix de ses paramètres.

La recherche tabou quant à elle, a donné de meilleurs résultats pour notre problème, elle sera encore davantage si un bon voisinage basé sur les données du problème est défini, ainsi qu'une technique d'aspiration et une stratégie de diversification, ont été élaborées.

Enfin, nous estimons que la qualité des métaheuristiques utilisées dépend étroitement du compromis entre la qualité de la solution obtenue et le temps d'exécution. Cependant, quelque soit la métaheuristique utilisée, il faut constater qu'il est devenu possible de résoudre des problèmes d'optimisation combinatoire de taille intéressante.

Nous devons souligner à la fin de cette thèse, que notre travail n'a pas la prétention d'être complet et exhaustif. Il est sujet à des questionnements, des ajouts et des modifications. Nous souhaitons, par ailleurs, que notre contribution, si modeste soit-elle, permettra de donner une impulsion à la recherche scientifique.

REFERENCES

[AVA03] C.Avanthay et al., 2003, A Variable Neighbourhood Search for Graph Coloring, EJOR, 151, 379-388.

[BAR03] Vincent Barichard, 2003, Approches Hybrides pour les Problèmes Multi objectifs, Thèse Doctorat, Ecole Doctorale d'Angers.

[BEL04] Hocine Belouadah, 2004, Single Machine Scheduling Problem with non-renewable Resources, Working Paper, Université Med Boudiaf de M'sila.

[BLE99] M. Blesa and F. Whafa, 1999, A Skeleton for the Tabu Search Metaheuristic With Applications to Problems in Software Engineering, ALCOM-FT and CICYT project (Mallaba).

[BON01] K.M.J. De Bontridder, 2001, Integrating Purchase and Production Planning, Ph.D. thesis, University of Eindhoven.

[CAR88] J. Carlier, P.Chrétienne, 1988, Problèmes d'ordonnancement : modélisation, complexité, algorithmes. Masson.

[CHA96] J.B. Chambers and J.W. Barnes, 1996, New Tabu Search Results for the Job Shop Scheduling Problem, MS.

[CHE04] C. Chekuri and S. Khanna, 2004, Handbook of Scheduling Algorithms, Models and Performance Analysis. CRC Press. Chap.11: Approximation Algorithms for Minimizing Average Weighted Completion Time.

[CLA99] Jens Clausen, 1999, Branch and Bound Algorithms – Principles and Examples, Dept. of Computer Science, University of Copenhagen.

[COR02] O. Cordon et al., 2002, A Review on the Ant Colony Optimization Metaheuristic: Basis, Models and New Trends, Mathware & Soft Computing, 9.

[DIG03] L.DI Gaspero, 2003, Local Search Techniques for Scheduling Problems: Algorithmes and Software Tools, Ph.D. Thesis, Universita Degli Studi Di Udine (Italia).

[DEM03] Sophie Demassey, 2003, Méthodes Hybrides de Programmation par Contraintes et Programmation Linéaire pour le Problème d'ordonnancement de Projets à Contraintes de Ressources. Thèse Ph.D, Université d'Avignon et des Pays de Vaucluse.

[DUM03] I. Dumitresco and T.Stutzle, 2003, Combinations of Local Search and Exact Algorithms, EvoWorkShops, 211-223, Springer Verlag, Berlin

[EIN04] Single Machine Problems, http://www.mathematik.uni-osnabrueck.de/research/OR/class

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

[FRE82] S.French, 1982, Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop, John Wiley & Sons, New York.

[GEN84] P.C. Geng, 1984, Multi-Product Lot-Sizing Problems on Single Machine, Ph.D Thesis, University of Waterloo, Ontario(Canada).

[GLO97] Fred Glover and Manuel Laguna, 1997, Tabu search, Kluwer Academic Publishers

[GLO99] Fred Glover and Said hanafi, 1999, Tabu Search and Finite Convergence, in Discrete Applied Mathematics.

[GRI05] A. Grigoriev, M. Holthuijsen and J.V. De Klundert, 2005, Basic Scheduling Problems with Raw Materials Constraints, Naval Research Logistics.

[HAN01] P. Hansen and N. Mladenovic, 2001, Recherche à voisinage variable, Les cahiers du GERAD.

[HAO99] Jin-Kao Hao et al., 1999, Métaheuristiques pour l'Optimisation Combinatoire et l'Affectation sous Contraintes, Revue d'Intelligence Artificielle, vol. N°1999.

[HAO03] M.Haouari and T.Ladhari, 2003, A Branch-and-Bound-Based Local Search Method for the Flow Shop Problem, JORS, 54, 1076-1084.

[HAR81] Hariri A.M.A., 1981, Scheduling : Using Branch and Bound Techniques. Ph.D Thesis, University of Keele.

[HER03] A. Hertz and M. Widmer, 2003, Guidelines for the use of Meta-heuristics in Combinatorial Optimization, EJOR, 151, 247-252.

[JAN00] A. Janiak, C.N. Potts and T. Tautenhahn, 2000, Single Machine Scheduling with Nonlinear Resource Dependencies of Release Times, AMS Journal.

[KAR98] David Karger, Cliff Stein and Joel Wein, 1998, Scheduling Algorithms, Massachusetts Institute of Technology.

[KOL97] R. Kolisch and R. Padman, 1997, An Integrated Survey of Project Scheduling, German Science Foundation.

[LAB98] F.Laburthe, 1998, Contraintes et Algorithmes en Optimisation Combinatoire, Thèse Ph.D, Université Paris VII.

[MAL97] K. Malek, 1997, Minimisation de la Somme Pondérée des Fonctions Exponentielles des Dates de Fin sur une Machine et sous Contraintes Temporelles. Thèse de Magister, Université de Constantine.

[OSM96] Ibrahim H.Osman, James P.Kelly, 1996, Meta-heuristics: Theory & Applications, Kluwer Academic Publishers, London.

[PAN05] Y.Pan and L.Shi, 2005, Dual Constrained Single Machine Sequencing to Minimize Total Weighted Completion Time, IEEE.

[PIN01] Janos D. Pinter, 2001, Global Optimization: Software, Test Porblems and Applications, Chap.15 in Handbook of Global Optimization, Vol.2, Kluwer Academic Publishers.

[POS85] Marc E. Posner, 1985, Minimizing Weighted Completion Times with Deadlines, J. Operations Research, 33, 562-574.

[POT83] Potts C.N. and Van Wassenhove L.N., 1983, An Algorithm for Single Machine Sequencing with Deadlines to Minimize Total Weighted Completion Time. European J. of Operational Research,12,379-387.

[POT85] Potts C.N., 1985, A Lagrangean Based Branch and Bound Algorithm for Single Machine Sequencing with Precedence Constraints to Minimize Total Weighted Completion Time.
Management Science., 31, 1300-1311.

[RAP02] Christophe Rapine et denis Trystram, 2002, Théorie de la Complexité, Notes de cours, ENSGI – INP Grenoble.

[REB99] Pascal Rebreynd, 1999, Algorithmes Génétiques Hybrides en Optimisation Combinatoire, Thèse Ph.D,
Ecole Normale Supérieure de Lyon.

[RIN76] A.H.G. Rinnooy Kan, 1976, Machine problems: Classification, Complexity and Computations, Martinus Nijhoff; The Hague.

[SAD02] Cherif Sadfi, 2002, Problèmes d'ordonnancement avec Minimisation des Encours. Thèse Ph.D, Institut National Polytechnique de Grenoble.

[SAK84] M.Sakarovitch, 1984, Optimisation Combinatoire, Méthodes Mathématiques et Algorithmiques, Hermann, Paris.

[SAL01] Michel Salomon, 2001, Etude de la Parallélisation des Méthodes Heuristiques d'optimisation Combinatoire. Thèse Ph.D, Université Louis Pasteur, Strasbourg.

[SEV04] Marc Sevaux, 2004, Métaheuristiques: Stratégies pour l'Optimisation de Biens et de Services, HDR, CNRS France.

[SCH95] A. Schirmer, 1995, A Guide to Complexity Theory in Operations Research, Working Paper N°381, Universitat zu Kiel, Germany.

[SIA03] P. Siarry et al., 2003, Métaheuristiques pour l'optimisation difficile, Chap.4: Les Algorithmes de colonies de Fourmis, Eyrolles.

[SIL02] Edward A. Silver, 2002, An Overview of Heuristic Solution Methods, Working Paper, University of Calgary.

[TAI01] Eric D.Taillard, 2001, Métaheuristiques et Outils Nouveaux en RO, Chap.2 : Principes d'implémentation des Métaheuristiques.

[WAL96] Matew Bartschi Wall, 1996, A Genetic Algorithm for Resource Constrained Scheduling, Ph.D thesis, Massachusetts Institute of Technology.

[WID01] Marino Widmer, 2001, Les Métaheuristiques: des Outils Performants pour les Problèmes Industriels, MOSIM'01, France.