UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences

Département de Mathématiques

MEMOIRE DE MAGISTER

Spécialité : Modélisation Mathématique pour l'Aide à la Décision

LES PROBLEMES D' ORDONNANCEMENT A MACHINES PARALLELES, DE TACHES DEPENDANTES

Par

BOUMEDIENE MEROUANE Hocine

Devant le jury composé de

M. BLIDIA	Professeur, U.de Blida	Président
A. DERBALA	Maître de conférences, U. de Blida	Promoteur
M. BENDRAOUCHE	Chargé de cours, U. de Blida	Examinateur
M. BOUDHAR	Maître de conférences, USTHB	Examinateur
S. BOUROUBI	Maître de conférences, USTHB	Examinateur
M. CHELLALI	Maître de conférences, U. de Blida	Examinateur

Blida, Juillet 2006

UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences
Département de Mathématiques

MEMOIRE DE MAGISTER

Spécialité : Modélisation Mathématique pour l'Aide à la Décision

LES PROBLEMES D' ORDONNANCEMENT A MACHINES PARALLELES, DE TACHES DEPENDANTES

Par

BOUMEDIENE MEROUANE Hocine

Devant le jury composé de

M. BLIDIA	Professeur, U.de Blida	Président
A. DERBALA	Maître de conférences, U. de Blida	Promoteur
M. BENDRAOUCHE	Chargé de cours, U. de Blida	Examinateur
M. BOUDHAR	Maître de conférences, USTHB	Examinateur
S. BOUROUBI	Maître de conférences, USTHB	Examinateur
M. CHELLALI	Maître de conférence, U. de Blida	Examinateur

Blida, Juillet 2006

RESUME

Nous considérons les problèmes d'ordonnancement NP-difficiles, de tâches dépendantes sur des machines parallèles identiques, afin de minimiser la longueur d'ordonnancement appelée *makespan*. Six listes de priorités ont été définies, implémentées et comparées. Elles sont basées sur le chemin critique, la durée d'exécution la plus courte d'abord, le nombre de successeurs immédiats d'une tâche le plus grand d'abord, l'ordre aléatoire et une variante du plus court chemin. La performance des méthodes est reproduite graphiquement. Nous mesurons la qualité et l'efficacité de chaque liste. Un algorithme génétique, noté AG, basé sur deux types de croisement, est implémenté et comparé avec les listes. Après de nombreuses expériences numériques et suite à des tests, les meilleurs résultats sont obtenus avec des variantes utilisant les listes comme population initiale de l'AG. Un second graphe comparatif avec les listes est obtenu. Les temps de calculs des six listes et de quatre variantes de l'AG sont donnés sous forme d'un tableau récapitulatif.

Mots-clés : Longueur d'ordonnancement, méthode de liste, algorithme génétique, rapport du plus mauvais cas.

ABSTRACT

We deal with the NP-hard problems of scheduling precedence constrained tasks on identical parallel processors expecting to minimize the schedule length or the makespan. Six lists of priority are defined, implemented and compared. They are based around the critical path, short processing time first, most immediate successors first, random order and a variant of the critical path. The lists performance is reproduced graphically. We measure a quality and efficiency of every list. A genetic algorithm, for short GA, based on two crossover is implemented and compared with the lists. After numerical experiments, with some tests, the best results are obtained for the GA's variant which uses the lists as initial population. A second comparative graphic with the lists is obtained. The computation times of the six lists and of four variants of GA are given with a table.

Key-words : Schedule length, list method, genetic algorithm, worst case ratio.

LISTE DES ILLUSTRATIONS, GRAPHIQUES ET TABLEAUX

Figure	Page
Figure 1. 1 : Diagramme de Gantt.	18
Figure 1. 2 : Ordonnancement d'un P_2 |prec |c_{max}.	20
Figure 1. 3 : Diagramme de Gantt pour la liste L'.	21
Figure 1. 4 : Les durées d'exécution diminuent, $p_j' = p_j - 1$ (j = 1, ..., 8).	21
Figure 1. 5 : Le nombre de machines augmente, m' = 3.	21
Figure 1. 6 : a- Contraintes de précédence affaiblies. b- Ordonnancement obtenu.	22
Figure 1. 7 : Illustration de la réduction Partition α P_2 | |C_{max}.	24
Figure 2. 1 : Une méthode SEP pour 1 | d_j | $\sum w_j (c_j - d_j)$. a- Données du problème. b- Arborescence de séparations.	28
Figure 3. 1 : Relation entre les classes d'approximation.	39
Figure 3. 2 : Diagramme de résolution de P par A.	40
Figure 3. 3 : Technique d'intervalle.	47
Figure 4. 1 : Exemple de P_2 |prec |C_{max}.	52
Figure 4. 2 : Diagramme de Gantt utilisant la règle avec délai.	54
Figure 4. 3 : Diagramme de Gantt utilisant la règle sans délai.	54
Figure 4. 4 : Exemple de P |s_{ij} |C_{max}. a- Les durées d'exécution. b- Les temps d'installation. c- Une solution optimale.	55
Figure 4. 5 : Exemple où la borne $\rho_{Ls} = 2 - \frac{1}{m}$ est atteinte.	57

a- Ordonnancement optimal (L). b- Ordonnancement de L'.	
Figure 5.1 : La fonction w (ord) a- w = cte b-w ↑ c- w ↑↑ d-w ↑↑↑	77
Figure 5. 2 : Performance des listes, durées d'exécution unitaires.	81
Figure 5. 3 : Performance des listes, durées d'exécution uniformément distribuées.	81
figure 5. 4 : Exemple de graphe de précédence. a- Le graphe de précédence. b,c,d- Respectivement les listes PLC, Max et SOM.	82
Figure 5. 5 : Rapport d'approximation moyen , durées d'exécution uniformément distribuées.	83
Figure 5. 6 : Nombre d'améliorations en fonction du nombre de générations.	90
Tableau 5. 1 : Exemple avec la fitness $\frac{1}{C_{max}}$.	85
Tableau 5. 2 : Exemple avec la fitness définie.	86
Tableau 5. 3 : Temps de calculs des six listes et des variantes de l'AG.	90

TABLE DES MATIERES

RESUME 1
REMERCIEMENTS 4
TABLE DES MATIERES 5
LISTE DES ILLUSTRATIONS GRAPHIQUES ET TABLEAUX 8
INTRODUCTION 10
1. LES PROBLEMES D' ORDONNANCEMENT A MACHINES PARALLELES 12
  1.1 Généralités et définitions 12
    1.1.1 Les caractéristiques des machines 13
    1.1.2 Les caractéristiques des tâches 14
    1.1.3 Les fonctions objectifs d'un problème d'ordonnancement 15
    1.1.4 Classification des problèmes d'ordonnancement 1[illegible]
    1.1.5 Diagramme de GANTT 1[illegible]
  1.2 Les problème d'ordonnancement à machines parallèles 1[illegible]
  1.3 Les anomalies de Graham 20
  1.4 Complexité des POCs 22
  1.5 Difficulté du problème P |prec |C_{max} 24
2. METHODES DE RESOLUTION DES PROBLEMES
D' ORDONNANCEMENT NP-DIFFICILES 26
  2.1 Les méthodes exactes 26
    2.1.1 Les méthodes par séparation et évaluation 27
    2.1.2 La programmation dynamique 28
  2.2 Les méthodes approchées 31
    2.2.1 Les méta heuristiques 32
      2.2.1.1 Recuit simulé 33
      2.2.1.2 Recherche tabou 33
    2.2.2 Evaluation des performances des méthodes approchées 33
      2.2.2.1 Analyse du plus mauvais cas 34

2.2.2.2 Analyse en moyenne 34

2.2.2.3 Analyse empirique 35

3. L'APPROXIMATION POLYNOMIALE 36

3.1 Généralités et définitions 36

3.2 La conception d'algorithme ρ-approximatif 39

3.3 La conception de schéma d'approximation polynomiale 4

3.3.1 Arrondir les données d'entrée 40

3.3.2 Arrondir les données de sortie 43

3.3.3 Modifier l'exécution d'algorithme 44

3.4 Les résultats négatifs 46

3.4.1 La non existence d'un FPAS 47

3.4.2 La technique de l'intervalle (*gap technique*) 47

3.4.3 L'utilisation de l'APX-difficulté

4. OUTILS DE RESOLUTION DU PROBLEME $P|prec|C_{max}$:

LES METHODES DE LISTES ET LES ALGORITHMES GENETIQUES 51

4.1 Les méthodes de listes 51

4.1.1 Règle de priorité 52

4.1.2 Règle d'affectation 53

4.1.3 Ensemble de listes dominant 54

4.1.4 Une liste dynamique 55

1.1.5 Utilisation des listes pour résoudre $P|prec|C_{max}$ 56

4.1.5.1 Algorithme de liste quelconque 56

4.1.5.2 Algorithme de Hu (1961) 58

4.1.5.3 Algorithme de CAUFFMAN et GRAHAM (1972) 59

4.1.5.4 Recherche d'une liste optimale par une méthode SEP 60

4.2 Les algorithmes génétiques 60

4.2.1 Un algorithme génétique 61

4.2.1.1 Codage 61

4.2.1.2 Population initiale 62

4.2.1.3 Evaluation 62

4.2.1.4 Sélection 63

4.2.1.5 Croisement 64

4.2.1.6 Mutation 65

4.2.1.7 Condition d'arrêt 65
4.2.2 La convergence des AGs 66
4.2.3 Les particularités des AGs 68
4.2.4 Des extensions pour P |prec |C_{max} 69
4.2.4.1 Codage des ordonnancements 69
4.2.4.2 Prise en compte des contraintes de précédence 71
4.2.2.3 Les anti-clones 73
5. IMPLEMENTATION, EXPERIMENTATION ET EVALUATION DE SIX LISTES DE PRIORITES ET D' UN AG 75
5.1 Un générateur de jeux d'essai 75
5.2 Critères de performance 77
5.3 Les expérimentations 79
5.4 Résultats des listes 79
5.5 Conception et résultats de l'AG 84
5.5.1 Le codage 84
5.5.2 La population initiale 84
5.5.3 L'évaluation et la sélection 85
5.5.4 Le croisement 87
5.5.5 La mutation 87
5.5.6 Un mécanisme anti-clones 87
5.5.7 Le critère d'arrêt 88
5.5.8 Résultats des AGs 88
5.6 Les temps de calculs 90
CONCLUSION ET PERSPECTIVES 92
REFERENCES 93

INTRODUCTION

Un domaine bien connu de la théorie de l'ordonnancement déterministe concerne l'attribution de tâches à un système multiprocesseurs afin de minimiser la longueur de l'ordonnancement. Ce critère est important pour les problèmes d'ordonnancement à machines parallèles où l'ordonnanceur assure l'équilibre de la charge des machines, qui est la somme de durées d'exécution des tâches affectées sur la dite machine. Les machines sont supposées identiques, le temps requis pour exécuter une tâche donnée est connu d'avance et ne dépend pas de la machine utilisée. Des relations de précédence existent entre les tâches.

Nous considérons les ordonnancements non- préemptifs où une fois une tâche débute son exécution, elle ne peut être interrompue. Le problème d'ordonnancement à contraintes de précédence sur des machines parallèles, noté $P|prec|C_{max}$, sont souvent difficiles et une résolution exacte n'est pas efficace. La résolution approchée est pratique pour des applications d'ingénieries, qui ont un besoin urgent en estimation de ressources de calculs [1][2][3], tels le calcul parallèle, la synthèse d'un système digital et la compilation de haute performance.

Les méthodes approchées sont évaluées à l'aide de bornes inférieures. Nous rappelons qu'un algorithme qui donne des solutions presque optimales est appelé *algorithme d'approximation*. S'il le fait en un temps polynomial, il est appelé *algorithme d'approximation polynomial*. Un algorithme d'approximation qui fournit toujours une solution presque optimale avec un coût au plus un facteur multiplicatif ρ de la solution optimale (où $\rho > 1$ est un nombre fixé), est appelé un ρ-algorithme d'approximation, et la valeur ρ est appelé le *facteur de garantie du plus mauvais cas.*

SCHURMAN et WOEGINGER [4] ont évoqué, pour les problèmes d'ordonnancement déterministes NP-difficiles, ce qu'ils appellent les questions ouvertes les plus vexantes, au sens où elles restent ouvertes depuis fort long temps. Nous avons voulu répondre à celle des machines parallèles de tâches dépendantes, respectivement $P|prec|C_{max}$ et $P|prec, pj=1|C_{max}$. Les problèmes sont respectivement de fournir un algorithme

d'approximation polynomial et un algorithme polynomial avec un facteur de garantie du plus mauvais cas de $2 - \delta$, $\delta > 0$.

En ordonnancement, les algorithmes les plus utilisés sont basés sur les listes. Ils déterminent pour un ordre de tâches donné par une liste, l'ordonnancement correspondant. Ils considèrent les tâches une par une et prennent la décision d'ordonnancer sur la base d'un ordonnancement partiel de tâches ordonnancées auparavant.

Pour beaucoup de problèmes difficiles dans une grande variété de domaines, les méta heuristiques ont reçu un intérêt considérable et se sont avérées efficaces pour les problèmes difficiles de l'optimisation combinatoire apparaissant dans des domaines variés: industriels, économique, logistique, ingénierie, commerce, domaines scientifiques, etc. Des exemples de méta heuristiques sont les algorithmes évolutionnistes de type génétique.

PHELIPEAU-GELINEAU [2] a développé un algorithme de recherche Taboue pour résoudre le problème $P|prec|C_{max}$. Dans un article récent, AYTUG et al. [5] fournissent un état de l'art sur l'utilisation des algorithmes génétiques dans la résolution des problèmes d'ordonnancement. Les articles sont référencées par type de problème qu'ils résolvent. Pour le problème $P||C_{max}$, CHIU et al. [6] ont été incapables de comparer leur algorithme génétique avec d'autres heuristiques.

Au lieu d'affronter les problèmes ouverts proposés par SCHURMAN et WOEGINGER, nous avons préféré et nous étions motivé , d'une part, à étudier et comparer six méthodes de listes définis de la littérature [2]. La performance des méthodes est reproduite graphiquement où nous mesurons la qualité et l'efficacité de chaque liste. D'autre part, un algorithme génétique, basé sur deux types de croisement, est implémenté et comparé avec les listes. Un second graphe comparatif avec les listes est obtenu. Les temps de calculs des six listes et de quatre variantes de l'AG sont comparés.

Dans la suite de ce mémoire, les problèmes d'ordonnancement à machines parallèles sont introduits au chapitre un. Notre intérêt est porté sur ceux classés NP-difficiles. Leurs méthodes de résolution, exactes et approximatives, sont exposées au chapitre deux. La notion d'approximation polynomiale, qui conçoit et analyse des algorithmes polynomiaux avec une garantie de performance, est largement introduite au chapitre suivant. Les outils de résolution du problème $P|prec|C_{max}$, qui sont les méthodes de type listes et les algorithmes génétiques, sont proposés en détail au chapitre quatre. Au chapitre final, on définit six listes de priorités et

on conçoit un algorithme génétique. Les méthodes ont été testées, implémentées et évaluées sur beaucoup d'instances générées aléatoirement. Une comparaison fait que les méthodes de listes sont meilleures que l'algorithme génétique. Nous concluons ce mémoire en donnant des perspectives de recherches.

CHAPITRE 1

LES PROBLEMES D' ORDONNANCEMENT

A MACHINES PARALLELES

Dans un problème d'ordonnancement interviennent deux notions fondamentales : les tâches et les ressources. Une tâche est un travail dont la réalisation nécessite un certain nombre d'unités de temps, sa durée, et d'unités de chaque ressource. Une ressource est un moyen, technique ou humain, dont la disponibilité limitée ou non est connue à priori.

Nous présentons les problèmes d'ordonnancement déterministes et leur classification et on illustre la représentation graphique d'une solution. Notre intérêt porte sur les problèmes à machines parallèles identiques. Ils modélisent plusieurs problèmes pratiques et théoriques.

1.1 Généralités et définitions

Un problème d'ordonnancement consiste à organiser dans le temps la réalisation d'un ensemble de tâches, compte tenu de contraintes temporelles (délais, contraintes d'enchaînement, etc.), et de contraintes portant sur l'utilisation et la disponibilité des ressources requises pour les tâches, et visant à minimiser (resp. maximiser) un certain critère d'optimalité [7].

Un problème d'ordonnancement est caractérisé par deux ensembles : un ensemble de n tâches $T = \{T_1, T_2, ..., T_n\}$ et un ensemble de m machines $M = \{M_1, M_2, ..., M_m\}$. Dans ce cas, une machine est considérée comme une ressource. Ordonnancer c'est assigner les tâches de l'ensemble T aux machines de l'ensemble M, dans l'ordre de compléter toutes les tâches sous des contraintes imposées.

De nombreuses contraintes peuvent être imposées. On s'intéresse essentiellement aux contraintes dues à une relation de dépendance entre les tâches. On note 'i prec j' et on dit "i précède j" si la tâche j ne peut commencer son exécution que si la tâche i ait terminé. Cette

relation est donnée par un graphe de précédence G (V,E) où V est l'ensemble des tâches et un arc (i, j) est dans E si et seulement si 'i prec j'.

Un problème est dit *préemptif* si la préemption est autorisée, l'exécution d'une tâche peut être interrompue et reprise plutard, avec ou *sans résume*. Une tâche est reprise à partir du point où elle a était interrompue ou elle est reprise du début. Un problème est *statique* ou *dynamique* (*on line* ou *offline*), si ses paramètres (les durées d'exécution, les dates de disponibilités, etc.) sont, respectivement, connus à priori ou à posteriori. Il est *déterministe* ou *stochastique* si, respectivement, ses paramètres sont connus et donnés ou suivent une distribution de probabilité.

Dans un problème classique d'ordonnancement, une machine ne peut exécuter qu'une tâche à la fois et une tâche ne peut être exécutée que sur une machine à la fois. Une affectation des tâches qui respecte les contraintes liées à l'environnement des machines et aux caractéristiques des tâches est dite *ordonnancement réalisable* ou *ordonnancement*. Si l'ordonnancement optimise le critère d'optimalité, il est dit *optimal*.

Un problème d'ordonnancement est défini par les caractéristiques des machines, celles des tâches et par le critère d'optimalité.

1.1.1 Les caractéristiques des machines

Les machines peuvent être *en parallèles*, faisant la même fonction, ou *spécialisées* dans l'exécution de certaines opérations (une tâche est constituée de plusieurs opérations), dans ce cas, les machines sont disposées en séries.

On distingue trois types de machines parallèles dépendant de leurs vitesses. Si elles ont la même vitesse d'exécution des tâches, elles sont dites *identiques*. Si elles diffèrent par leurs vitesses d'exécution et la vitesse d'une machine est constante et ne dépend pas des tâches, elles sont dites *uniformes*. Si les vitesses d'exécution des machines dépendent des tâches et sont différentes alors elles sont dites *quelconques*.

Dans le cas de machines spécialisées, il y a trois modèles ou types d'exécution de tâches: le *flow shop*, l'*open shop* et le *job shop*. Pour plus de détails, voir [8].

1.1.2 Les caractéristiques des tâches

A chaque tâche $T_j \in T$, on associe les paramètres suivants :

1. La *durée d'exécution* p_{ij} de la tâche T_j sur la machine M_i. Dans le cas de machines identiques, le temps d'exécution d'une tâche ne dépend pas de la machine, d'où $p_{ij} = p_j$, $i = 1..m$. Si les machines sont uniformes alors $p_{ij} = \frac{p_j}{b_i}$, $i = 1..m$ où p_j est une durée d'exécution de référence, mesurée usuellement sur la machine la moins rapide et b_i est le facteur vitesse d'exécution de la machine M_i .

2. La *date d'arrivée* ou *date de disponibilité* (*release date*) r_j. La date où la tâche T_j est prête pour l'exécution. Si les dates d'arrivée sont les mêmes pour toutes les tâches, on suppose que $r_j = 0$, $j = 1..n$.

3. La *date de fin d'exécution au plutard* (*due date*) d_j. Appelée aussi *date de fin échue* ou *date de fin souhaitée*. Si la tâche termine son exécution après cette date, elle encourt une pénalité.

4. La *date de fin impérative* $\tilde{d}_j$ (*deadline*). Si la tâche termine son exécution après cette date, elle ne risque pas seulement une pénalité mais des problèmes surgiront, soit l'atelier est bloqué ou la machine tombe en panne, etc.

5. Le *poids* ou la *priorité* w_j (*weight*). Il exprime une urgence dans l'exécution de la tâche T_j.

6. La *date de fin d'exécution* C_j. En général, c'est une variable à déterminer.

7. Le *décalage* $L_j = C_j - d_j$ (*lateness*). Le temps total durant lequel, la tâche est autorisée à rester dans l'atelier.

8. Le *retard* $D_j = \max \{C_j - d_j, 0\}$ (*tardiness*).

9. L'*avance* $E_i = \max (0 , -L_i)$ (*earliness*).

10. L'*indicateur de retard* $U_i = 0$ si $c_i \leq d_i$ et $U_i = 1$ sinon.

1.1.3 Les fonctions objectifs d'un problème d'ordonnancement

Les critères d'optimalité les plus utilisés font intervenir la durée totale de l'ordonnancement, le délai d'exécution, les retards de l'ordonnancement, etc. La durée totale d'ordonnancement, notée C_{max}, est égale à la date d'achèvement de la tâche la plus tardive: $C_{max} = \max_j C_j$. C'est la *longueur d'ordonnancement* appelée aussi *makespan* (*schedule length*).

Le *flow time pondéré* $\sum_{j=1}^{n} w_j C_j$ permet d'estimer le coût des stocks d'encours. En effet, une tâche T_j est présente dans l'atelier entre les instants r_j et C_j. Les stocks dont elle a besoin doivent être disponibles entre ces deux dates; d'où le coût $\sum_{j=1}^{n} w_j (C_j - r_j)$ égale, à une constante prés, à $\sum_{j=1}^{n} w_j C_j$.

Dans beaucoup de problèmes, il faut respecter les délais et les dates au plutard d_j. Minimiser le *plus grand retard* $D_{max} = \max_j D_j$, la *somme des retards* $\sum_{j=1}^{n} D_j$ ou la *somme pondérée des tâches en retard* $\sum_{j=1}^{n} w_j D_j$ peuvent être d'un intérêt.

D'autres critères peuvent être définis et utilisés.
Le *décalage maximum* $L_{max} = \max_j L_j$. Le *nombre de tâches en retard* $\sum_{j=1}^{n} U_j$. Le *nombre de tâches en retard pondéré* $\sum_{j=1}^{n} w_j U_j$.

1.1.4 Classification des problèmes d'ordonnancement

Il existe un grand nombre de problèmes d'ordonnancement, une notation et une classification s'imposent. En 1979, GRAHAM et al. [9] ont classifié ces problèmes en utilisant une notation à trois champs $\alpha|\beta|\gamma$. Dans la littérature, de nouvelles extensions peuvent être définies permettant de représenter des problèmes nouveaux liés aux systèmes informatiques, aux modèles économiques, etc.

Le champ $\alpha = \alpha_1 \ \alpha_2$ décrit l'environnement des machines.

Le paramètre $\alpha_1 \in$ { 1 ou $\varnothing$, P, Q, R, O, F, J, FH } caractérise le type de machines utilisées :

α_1 = 1 : auquel cas, nous avons un problème à une seule machine.

α_1 = P : machines identiques parallèles.

α_1 = Q : machines parallèles uniformes.

α_1 = R : machines parallèles quelconques.

α_1 = O : système open shop.

α_1 = F : système Flow shop.

α_1 = J : système Job shop.

α_1 = FH : système flow shop *hybride*.

$\alpha_2 \in \{\varnothing, k\}$ est un entier qui représente le nombre de machines dans le problème.

$\alpha_2 = \varnothing$: le nombre de machines est supposé être variable.

α_2 = k : le nombre de machines est égal à k (k entier positif).

Le second champ $\beta = \beta_1 \beta_2 \beta_3 \beta_4 \beta_5 \beta_6 \beta_7 \beta_8$ décrit les tâches et les caractéristiques des ressources.

Le paramètre $\beta_1 \in$ {$\varnothing$, pmtn} indique la possibilité de la préemption de la tâche: resp. la préemption n'est pas autorisée ou autorisée.

$\beta_2 \in$ {$\varnothing$, res} caractérise les ressources additionnelles: resp. aucune ressource additionnelle n'existe ou des contraintes de ressources spécifiées.

$\beta_3 \in$ {$\varnothing$, prec, tree, chains} reflète les contraintes de précédence : resp. les tâches sont indépendantes, de contraintes de précédence quelconques, formant un arbre ou formant union de chaînes.

$\beta_4 \in \{\emptyset, r_j\}$: resp. toutes les instants au plutôt sont nuls ou sont différents.

$\beta_5 \in \{\emptyset, p_j = p, \underline{p} \le p_i \le \bar{p}\}$: resp. les durées d'exécution sont arbitraires, égaux ou dans l'intervalle défini.

$\beta_6 \in \{\emptyset, \tilde{d}\}$ décrit les deadlines : resp. aucune date de fin impérative ou des dates sont imposées.

$\beta_7 \in \{\emptyset, n_j \le k\}$ décrit le nombre maximum d'opérations constituant une tâche : resp. aucune contrainte ou le nombre est inférieur ou égal k.

$\beta_8 \in \{\emptyset$, no-wait$\}$ décrit, pour un problème à machines spécialisées, une propriété d'attente: resp. la région tampon est de capacité illimitée ou sans attente, les capacités d'attente sont nulles, une tâche qui termine son exécution sur une machine passe instantanément sur une autre machine.

Le troisième champs γ désigne le critère à utiliser.

$$\gamma \in \{ C_{max} ; \sum_{j=1}^{n} C_j ; L_{max} ; \sum_{j=1}^{n} D_j ; \sum_{j=1}^{n} w_j D_j / \sum_{j=1}^{n} w ; \sum_{j=1}^{n} U_j ; \sum_{j=1}^{n} w_j U_j , \text{etc.} \}.$$

Exemple 1.1 : - $P \,||\, C_{max}$ désigne le problème de minimiser la longueur d'ordonnancement de tâches indépendantes, arrivant aux instants 0, à machines parallèles identiques. La préemption n'est pas autorisée.

- $P_3 \,|pmtn, prec|\, L_{max}$ désigne le problème de minimiser le retard maximum de tâches liées par des contraintes de précédence arbitraires, à trois machines parallèles identiques. La préemption est autorisée.

1.1.5 Diagramme de GANTT

Pour représenter un ordonnancement, on utilise couramment une représentation dite *diagramme de Gantt*. Celle-ci utilise un repère orthogonal dans un plan. L'axe des abscisses représente le temps et l'axe des ordonnées représente l'ensemble des machines. Pour chaque machine, on représente la séquence de tâches effectuées dans le temps. Les parties en gris représentent les périodes de oisiveté d'une machine, dues aux éventuelles indisponibilités des tâches.

La figure 1.2 représente un ordonnancement à deux machines M_1 et M_2 et quatre tâches T_1,T_2,T_3,T_4 de durées d'exécutions respectives 1,2,4,5 .

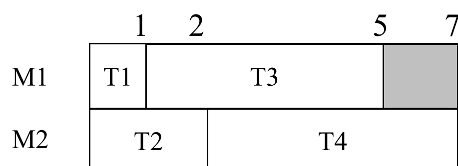


Figure 1.1 :Diagramme de Gantt

- La machine M_1 exécute T_1 ensuite T_3, et M_2 exécute T_2 puis T_4.
- Le makespan C_{max} est de 7 unités de temps. La somme des date de fin d'exécution $\sum_{j=1}^{4} C_j$ est de 15.

1.2 Les problèmes d'ordonnancement à machines parallèles

Ces problèmes sont une généralisation de ceux à une seule machine, plusieurs machines sont disponibles pour l'exécution d'une tâche.

L'utilisation du parallélisme pour la réalisation de projets de grande taille et pour le traitement d'applications réclamant une puissance de calcul de plus en plus importante est aujourd'hui une réalité. Il est aussi intéressant de considérer le parallélisme pour des raisons de fiabilité. D'autres part, le parallélisme pose des problèmes non rencontrés dans l'exécution séquentielle, comme l'extraction du parallélisme d'un projet et sa décomposition en tâches, la détection des contraintes chronologiques (délais de communication)[10] et les dépendances entre les tâches (contraintes de précédence) ou entre les tâches et les machines (contraintes de placement) [2]. Le processus d'ordonnancement consiste en l'affectation des tâches aux machines et leur séquencement sur chacune d'elles.

On appelle *charge* d'une machine la somme des durées d'exécution des tâches qui lui sont affectées. Dans la suite de ce mémoire, nous ne considérons que les problèmes d'ordonnancement statiques et déterministes, à machines parallèles identiques, non préemptifs et à contraintes de précédence quelconques, le critère d'optimalité est la longueur d'ordonnancement, soit $P|prec|C_{max}$. Il s'interprète comme l'équilibre de la charge entre les machines. Sans perte de généralité, les problèmes sont à minimiser.

Les deux exemples suivants modélisent ce type de problème.

i) Le problème des deux voleurs : [11] Deux voleurs sortent chaque nuit pour dérober des villas. Ils partagent les objets volés juste avant le levé du jour. Chaque objet j est indivisible et a une certaine valeur C_j. Ils veulent partager équitablement les gains. Le problème peut se formuler comme un $P_2 | | C_{max}$, en considérant les deux voleurs comme deux machines identiques M_1 et M_2 et les objets volés comme des tâches $T_1,T_2,,...,T_n$. Minimiser le makespan signifie que le partage du butin est équitable. Pour ce faire, les voleurs doivent utiliser un algorithme rapide pour leur éviter d'être capturés.

Ce problème est utilisé pour illustrer les différents concepts et méthodes introduits dans ce mémoire. Une généralisation à m voleurs se formule comme un $P_m | | C_{max}$.

ii) L'architecture VLIW : [12] Dans une architecture langage VLIW (*very long instruction word*), un processeur contient plusieurs unités fonctionnelles capable d'exécuter des opérations de base en parallèle durant un même cycle d'horloge. Le compilateur combine ces opérations dans des méta opérations. Il doit prendre en compte les dépendances entre les opérations de base dues aux dépendances des données. Le problème peut se formuler comme celui d'ordonnancement, en associant aux unités fonctionnelles de même vitesse 'm' machines identiques en parallèles et aux opérations de base des tâches de durées égales au cycle d'horloge. Les dépendances entre les opérations sont supposées arbitraires. Minimiser la durée de l'application correspond au problème d'ordonnancement $P_m | prec, p_j = p | C_{max}$.

Le compilateur de VLIW est plus important que dans le cas des architectures traditionnelles séquentielles. Il comporte un algorithme de résolution du problème $P_m | prec, p_j=1 | C_{max}$.

Un problème d'ordonnancement est un problème d'optimisation combinatoire (POC), qui consiste à chercher le minimum s* d'une application f, définie d'un ensemble S dans un ensemble R le plus souvent à valeurs entières.

Dans le problème $1 | prec | C_{max}$, l'ensemble S est constitué des séquences de tâches qui satisfont les contraintes de précédence. L'application f associe à chaque séquence la date de fin d'exécution de la tâche la plus tardive.

Nous considérons qu'un *problème* est une forme générique ou modèle et une *instance* est son application numérique. Un *algorithme* de résolution d'un problème donné est une procédure décomposable en opérations élémentaires, qui transforme les données en résultats.

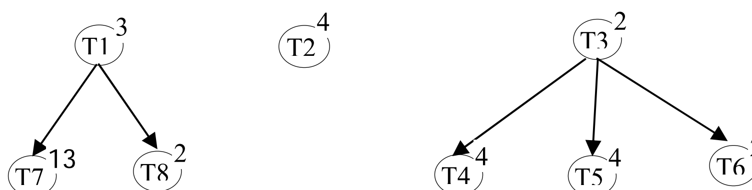
1.3 Les anomalies de GRAHAM (1966)

Une méthode simple pour déterminer un ordonnancement réalisable consiste à établir une liste de priorité des tâches et affecter les tâches disponibles, l'une après l'autre, suivant cette liste, à une machine disponible ou celle qui se libère en premier. On présente ces méthodes au quatrième chapitre.

Exemple 1.2 : Soit $P_2 | prec | C_{max}$,

Tâche	T1	T2	T3	T4	T5	T6	T7	T8
p_i temps d'exécution	3	4	2	4	4	2	13	2

Les contraintes de précédence sont données ci-dessous :



Sur la figure 1.2, la liste de priorité L, L = $T_1\ T_2\ T_3\ T_7\ T_4\ T_5\ T_6\ T_8$, est utilisée pour déterminer un ordonnancement au problème $P_2 | prec | C_{max}$.

	0–3	3–5	5–9	9–13	13–15	15–17
M1	T1	T3	T4	T5	T6	T8
M2	T2 (0–4)	T7 (4–17)				

Figure 1.2 : Ordonnancement d'un $P_2 | prec | c_{max}$

GRAHAM (1966) [13] a met en évidence des anomalies dans le comportement de ces méthodes. Pour le problème $P | prec | C_{max}$, le makespan d'un ordonnancement de liste peut augmenter si l'un des changements suivants a lieu:

- Le nombre de machines augmente,
- Les durées d'exécution des tâches diminuent,
- Les contraintes de précédence sont affaiblies,
- La liste de priorité change.

Les figures 1.3, 1.4, 1.5, 1.6, montrent ces anomalies sur l'exemple précédent.

Si on change la liste L en L', L'= T_1 T_2 T_3 T_4 T_5 T_6 T_8 T_7, la longueur d'ordonnancement augmente de 17 à 23 unités.

0 3 4 5 6 9 10 11 23

M1	T1	T3	T4	T6	
M2	T2	T8	T5	T7	

Figure 1.3 : Ordonnancement de la liste L'

Même si on réduit les temps d'exécution de chaque tâche d'une unité de temps,
$p_j' = p_j - 1$ (j = 1..8), la longueur d'ordonnancement augmente aussi de 17 à 18 unités.

0 2 3 6 7 8 18

M1	T1	T3	T4	T6	T8
M2	T2	T5	T7		

Figure 1.4 : Les durées d'exécution diminuent, $p_j' = p_j - 1$ (j = 1, ..., 8)

Même si on dispose d'une machine supplémentaire qu'on affecte à l'atelier, la longueur d'ordonnancement augmente aussi de 17 à 19 unités.

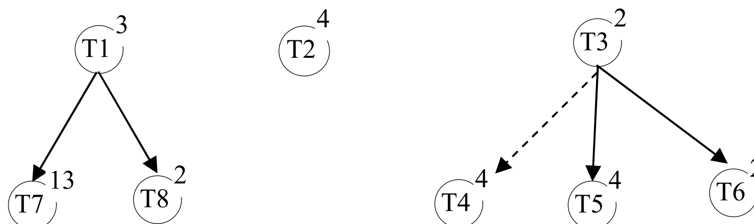
0 2 3 4 6 7 8 19

M1	T1	T5	
M2	T2	T6	T7
M3	T3	T4	T8

Figure 1.5 : Le nombre de machines augmente, m' = 3

Même si on affaiblit les contraintes de précédence en retirant un arc (T_3, T_4) du graphe de précédence, la longueur d'ordonnancement augmente aussi de 17 à 22 unités.

a)



b)

0 3 4 5 8 9 10 12 22

M1	T1	T3	T5	T7	
M2	T2	T4	T6	T8	

Figure 1.6 : a- Contraintes de précédence affaiblies
b- Ordonnancement obtenu

Remarque : Si une liste de priorité est optimale pour un problème donné, on ne peut rien en déduire pour un problème similaire où des paramètres changent. D'où, la difficulté à établir un algorithme de résolution optimale de type liste pour $P|prec|C_{max}$.

1.4 Complexité des POCs

Il est clair que parmi les POCs, des problèmes sont plus faciles à résoudre que d'autres. La complexité d'un problème est spatiale (les algorithmes connus demandent une place en mémoire exagérée) ou temporelle (le temps nécessaire à leur résolution est démesuré). Nous nous intéresserons à la complexité temporelle.

Cook [14] a élaboré une théorie de la complexité en mesurant le temps de calcul en fonction de la taille du problème. Sa théorie distingue entre les algorithmes polynomiaux, de complexité de l'ordre d'un polynôme, et les autres dits exponentiels. La puissance de calcul ne résout rien pour des complexités exponentielles, il est faux de croire qu'un ordinateur peut résoudre tous les problèmes combinatoires.

Une représentation standard des données du problème A, appelée *codage* prend un espace mémoire appelé *taille* du problème et notée $|A|$. S'il existe un algorithme qui donne

une solution optimale en un temps $T \leq p(|A|)$ où p est un polynôme, on dit que notre problème est *facile* ou *polynomial*. Les problèmes polynomiaux forment la *classe P*.

Les problèmes possédant une propriété plus générale : "ayant une solution x du problème A, tel que $|x|$ est bornée par un polynôme en $|A|$, on peut vérifier en un temps polynomial que x est une solution réalisable" constituent la *classe NP*. Il est clair que $P \subseteq NP$.

La notion de *réduction polynomiale* a été introduite dans le but de comparer la complexité des problèmes. Un problème A peut être transformé en un autre problème B tel que : à partir d'une solution de B, on peut calculer une solution de A. Si on peut faire la transformation et le calcul en un temps polynomial, on dit que A *se réduit polynomialement* à B et on note $A \alpha B$. Par conséquence, un algorithme polynomial pour résoudre B induit un algorithme polynomial pour résoudre A.

La classe NP-C est définie, elle contient les problèmes NP vers lesquels on peut réduire polynomialement tout autre problème NP. S'il existe un algorithme polynomial pour résoudre un problème NP-C alors P = NP. Les problèmes de NP-C sont dits *NP-Complets* ou *difficiles*.

Le choix du codage des données peut influencer la complexité d'un problème. Le codage utilisé par défaut est le *codage binaire*. Si un problème A est NP-Complet et devient, avec un *codage unaire*, polynomial, on dit que A est NP-Complet *au sens faible* et l'algorithme est dit *pseudo polynomial*. Par contre si A reste NP-Complet même avec un codage unaire, on dit qu'il est NP-Complet *au sens fort*.

Les travaux de Cook sont basés à l'origine sur les *problèmes de décision* qui sont exprimés sous forme d'une question. La résolution du problème consiste à apporter une réponse *positive* ou *négative* (oui ou non). Il est possible, pour certains problèmes d'optimisation, d'associer une *version décision*. On associe une question portant sur l'existence ou non d'une solution de valeur meilleure qu'un seuil arbitraire. C'est par ce biais qu'on fait le lien entre les problèmes d'optimisation et les problèmes de décision.

Un problème d'optimisation est au moins aussi difficile que sa version décision, puisque pour résoudre cette dernière, il suffit de résoudre le problème d'origine. Lorsque sa version décision est NP-complète, le problème d'optimisation est dit NP-difficile, ou par abus de langage NP-complet.

1.5 Difficulté du problème $P|prec|C_{max}$

Les problèmes d'ordonnancement intéressants se montrent difficiles. $P||C_{max}$, le problème de tâches indépendantes et de durées d'exécution arbitraires, est montré NP-difficile même pour le cas simple à deux machines ($P_2||C_{max}$).

Théorème 1.1 : [15] Le problème d'ordonnancement $P_2||C_{max}$ est NP-difficile.

Preuve : Un POC est NP-difficile si le problème décisionnel lui correspondant est NP-complet. Le problème$P_2||C_{max}$. « Existe-t-il une affectation des tâches, tel que la durée totale des tâches affectées à chaque machine, est au plus k ? » est la version décision du problème $P_2||C_{max}$, où k est un entier.

Considérons le problème de partition connu comme étant NP-complet [15].

Le problème Partition : Soit un ensemble fini E d'entiers positifs a_j. Existe t-il un sous ensemble E' de E tel que : $\sum_{a_j \in E'} a_j = \sum_{a_j \in E-E'} a_j$.

Etant donnée une instance quelconque du problème de partition, on peut créer une instance de la version décision de $P_2||C_{max}$, de la manière suivante :

i. $n = |E|$, où n est le nombre de tâches

ii. $p_j = a_j$, $j = 1 .. n$

iii. $k = \frac{1}{2}\sum_{j=1}^{n} p_j$.

Il existe un sous-ensemble E' pour l'instance de partition si et seulement si il existe un ordonnancement de longueur inférieur ou égale à $k = \frac{1}{2}\sum_{j=1}^{n} p_j$ pour $P_2||C_{max}$.

M1	E'
M2	E - E'

Figure 1.7 : Illustration de la réduction Partition α $P_2||C_{max}$

Remarques : 1) $P_2 | | C_{max}$ et $P | | C_{max}$ sont NP-complets au sens faible [11][16] tandis que $P_m | | C_{max}$ est NP-complet au sens fort [17].

2) Si la préemption est autorisée, le problème $P | pmtn | C_{max}$ est polynomial [18].

3) En présence de contraintes de précédence, même avec des temps d'exécution unitaires, $P | prec, p_j = 1 | C_{max}$ est NP-complet au sens fort [19].

4) Dans le cas de deux machines, $P_2 | prec, p_j = 1 | C_{max}$ est polynomial [20].

5) La complexité de $P_3 | prec, p_j = 1 | C_{max}$ reste un problème ouvert.

6) Les problèmes $P | chains, p_j = 1 | C_{max}$ et $P | tree, p_j = 1 | C_{max}$ [21] sont polynomiaux.

CHAPITRE 2

METHODES DE RESOLUTION DES PROBLEMES D' ORDONNANCEMENT NP-DIFFICILES

Pour les problèmes NP-difficiles, on ne connaît pas d'algorithmes polynomiaux pour les résoudre, et la conjecture $P \neq NP$ fait qu'il est peu probable qu'il en existe. En pratique, on résout un tel problème en tirant parti de :

• La taille des données. L'énumération complète peut être valable sur des instances de petite taille.

• La complexité moyenne : un algorithme exponentiel sur son pire cas, peut être assez rapide en moyenne, comme l'algorithme du simplexe.

• La nature des données : le problème peut devenir facile sur des cas particuliers.

• Méthodes diminuant la combinatoire : méthodes par séparation et évaluation, programmation dynamique, etc.

• Méthodes approchées ou heuristiques. On accepte des solutions de bonne qualité sans garantie d'optimalité.

Dans le reste de ce chapitre, on présente les méthodes de résolution exacte et approchée.

2.1 Les méthodes exactes

Les techniques de résolution exacte sont définies pour les problèmes combinatoires en général. L'énumération exhaustive est la méthode la plus simple. Les méthodes par séparation et évaluation et la programmation dynamique font une énumération intelligente des solutions

possibles. Pour des problèmes difficiles de grande taille, la durée d'exécution est démesurée et il faut envisager une résolution approchée. Ils sont exposées dans la suite.

2.1.1 Les méthodes par séparation et évaluation (SEP)

Elles sont aussi appelées méthodes arborescentes. Ce sont des méthodes exactes qui pratiquent une énumération complète et améliorée des solutions. Elles partagent l'espace des solutions en sous ensembles de plus en plus petits, la plupart étant éliminés par des calculs de bornes. Appliquées à des problèmes NP-difficiles, ces méthodes restent bien sur exponentielles, mais leur complexité est bien plus faible que pour une énumération exhaustive. Elles peuvent pallier le manque d'algorithmes polynomiaux pour des problèmes de taille moyenne.

Pour un POC, on peut inventer plusieurs méthodes par séparation et évaluation. Cependant elles auront trois composantes communes :

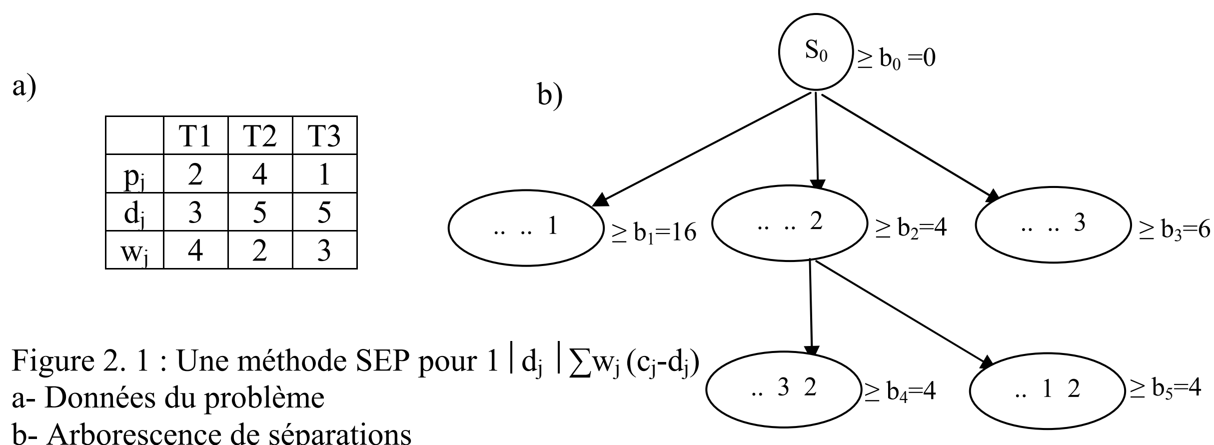
- Une règle de séparation, permettant de partitionner un ensemble de solutions en sous ensembles.
- Une fonction d'évaluation, permettant le calcul d'une borne pour un ensemble de solutions.
- Une stratégie d'exploration de l'arborescence de recherche.

Exemple 2.1 : Une méthode SEP pour un problème à une seule machine.

Soit le problème $1 | d_j | \sum w_j (c_j - d_j)$ qui consiste à exécuter n tâches sur une machine. Une tâche de durée p_j doit terminer son exécution avant l'instant d_j, sinon il faut payer une pénalité de retard w_j par unité de temps. Le critère à minimiser est la somme des pénalités de retard.

L'ensemble des solutions réalisables S_0 est formé des n! permutations possibles des n tâches.

La séparation se fera sur la tâche que l'on fixe en dernier, puis en avant dernier et ainsi de suite. Pour le calcul des bornes ou minorants, on prend en considération la somme des pénalités des tâches placées en queue. L'exploration est en profondeur en suivant le sommet ayant le plus petit minorant. Le problème suivant est à 1 machine et 3 tâches.



a)

	T1	T2	T3
p_j	2	4	1
d_j	3	5	5
w_j	4	2	3

Figure 2. 1 : Une méthode SEP pour $1 \mid d_j \mid \sum w_j (c_j - d_j)$
a- Données du problème
b- Arborescence de séparations

Initialisation : Au pire des cas, aucune tâche n'est en retard, d'où $b_0 = 0$.

On a séparé par rapport à une tâche placée en dernière position. Pour calculer b_1, quelque soit la séquence (2, 3, 1) ou (3, 2, 1), la tâche 1 ne peut commencer son exécution qu'à la date $p_2 + p_3 = 5$. Elle sera en retard de 4 unités. Le coût de retard sera d'au moins 4 x 4 = 16 d'où $b_1 = 16$. Ainsi de suite $b_2 = 2 \times 2$ car la tâche 2 ne sera pas en retard si elle est placée la dernière. Si la tâche 3 est placée la dernière, elle sera en retard de deux unités, $b_3 = 3 \times 2 = 6$.

On explore suivant le sommet de b_2. Les séquences (1, 3, 2) et (3, 1, 2) sont optimales et ont des pénalités de 4 unités. Les sommets de $b_1 = 16$ et $b_3 = 6$ sont alors éliminés.

En ordonnancement, cette approche est utilisée dans la résolution des problèmes à une seule machine. Lageweg et al [22] ont proposé un algorithme polynomial pour résoudre les problèmes $1 \mid prec, r_j = 1 \mid L_{max}$ et $1 \mid prec, pmtn, r_j = 1 \mid L_{max}$. C'est une généralisation de l'algorithme et Backer et Su [23] et McMahon et Florian [24] qui résout $1 \mid r_j \mid L_{max}$.

Pour résoudre les problèmes à plusieurs machines, les méthodes par séparation et évaluation sont combinées avec celles de listes, qu'on étudiera au quatrième chapitre.

2.1.2 La programmation dynamique

La programmation dynamique est une technique mathématique qui aide à prendre des décisions séquentielles indépendantes les unes des autres. Il n'y a pas de formalisme mathématique standard, c'est une approche de résolution où les équations doivent être spécifiées selon le problème à résoudre.

Les propriétés connues de la programmation dynamique sont :

i) le problème peut être décomposé en étapes et une décision doit être prise à chaque étape. Ces décisions sont interdépendantes et séquentielles.

ii) A chaque étape correspond un certain nombre d'états qui peut être infini ($x_n \in \angle$) ou continu ($x_n \in 3$).

iii) A chaque étape, la décision prise transforme l'état actuel en un état associé à l'étape suivante.

iv) Etant donné un état, une stratégie optimale pour les étapes restantes est indépendante des décisions prises aux étapes précédentes. L'état actuel contient toute l'information nécessaire aux décisions futures. Cette propriété est dite *principe d'optimalité*.

v) L'algorithme de recherche de la solution optimale commence par trouver la stratégie optimale pour tous les états de la dernière étape.

vi) Une relation de récurrence identifie la stratégie optimale dans chaque état de l'étape n à partir de la stratégie optimale dans chaque état de l'étape n + 1. La relation est de la forme : $f_n^* (S) = \min_{x_n} \{ C_S x_n + f^*_{n+1} (x_n) \}$ (resp. max). f_n dépend de S et x_n.

vii) Utilisant cette relation de récurrence, l'algorithme procède en reculons étape par étape. Il détermine la stratégie optimale pour chaque état de chaque étape.

Un programme dynamique peut être décomposé en quatre phases [25]. La première phase concerne la définition d'une étape et d'un état, la deuxième définie la relation de récurrence qui est utilisée dans la troisième phase pour déterminer les ensembles L_k des états de chaque étape k. Dans la dernière phase, on construit une solution optimale par une procédure retour arrière (*backtracking*).

Exemple 2.2 : Un programme dynamique pour résoudre $P_2||C_{max}$ [11].

Soit le problème $P_2||C_{max}$ qui consiste à minimiser la longueur d'ordonnancement de n tâches T1, T_2, ..., T_n sur deux machines parallèles identiques M_1 et M_2.

On note S_1 la charge de M_1. L'algorithme se base sur l'extension de l'ensemble des valeurs possibles de S_1.

Phase 1 : Soit L_k, l'ensemble des valeurs possibles de S_1, en considérant seulement les tâches $T_1, T_2, \ldots, T_k$, ce qui s'écrit :

$$L_k = \{ \sum_{j \in S} p_j, S \subset \{1,2,\ldots,k\} \} \quad \text{et} \quad L_0 = \{0\}$$

Phase 2 : A chaque étape k, on doit décider d'assigner la tâche T_k à M_1 ou à M_2. Cela conduit à la relation de récurrence suivante:

$$L_k = L_{k-1} \cup \{ x + p_k / x \in L_{k-1} \}, k = 1..n.$$

Phase 3 : On calcule récursivement les ensembles $L_1, L_2, \ldots, L_n$. Les éléments de $L_0, L_1, \ldots, L_n$ peuvent être vus comme les états du programme dynamique. On calcul OPT la valeur optimale du makespan par la formule :

$$OPT = \min_x (x - \frac{1}{2} \sum_{j=1}^{n} p_j) \geq 0,$$

sachant que $\frac{1}{2} \sum_{j=1}^{n} p_j$ est une borne inférieure triviale.

Phase 4 : On détermine une solution optimale de coût OPT comme suit : pour k : n, …, 1, on vérifie si OPT est dans L_{k-1}, on affecte la tâche T_k à M_2, sinon on l'affecte à M_1 et on remplace OPT par OPT - p_k.

Le programme dynamique est en $o(\sum_{k=1}^{n} |L_k|)$. Pour un codage unaire des données, il est polynomial et $P_2||C_{max}$.

Exemple 2.3 : Un programme dynamique pour résoudre $P||C_{max}$ [16].

Le problème $P||C_{max}$ consiste à minimiser la longueur d'ordonnancement de n tâches T1, T_2, .., T_n sur m machines parallèles identiques $M_1, M_2, \ldots, M_m$, m est arbitraire.

Le temps est discret, $t_i = 0, 1, \ldots, c$, où c est une borne supérieure du makespan. L'algorithme se base sur des variables booléennes.

Phase 1: Les variables $x_k(t_1, t_2, \ldots, t_m)$, k = 1..n sont définies par les expressions :

$$x_k(t_1, t_2, ..., t_m) = \begin{cases} \text{vrai} & \text{si les stâchs } T_1, T_2, ..., T_k \text{ peuvent être affectées aux machines} \\ & M_1, M_2, ..., M_m \text{ tel que } M_i \text{ est occupée dans } [0, t_i], i = 1,2,...,m \\ \text{faux} & \text{si non} \end{cases}$$

Les x_k $(t_1, t_2, ..., t_m)$ sont les états du programme dynamique.

Phase 2 : Les x_k $(t_1, t_2, ..., t_m)$ sont calculées par la relation de récurrence

$$x_k (t_1, t_2, ..., t_m) := \bigvee x_{k-1} (t_1, t_2, ..., t_{i-1}, t_i\text{-}p_k, t_{i+1}, \ldots, t_m), (i = 1..m); (*)$$

Où $\bigvee$ est le ou logique.

Phase 3 : On calcule la valeur d'une solution optimale par :

$$OPT := \min \{ \max \{t_1, t_2, ..., t_m\} / x_n (t_1, t_2, ..., t_m) = \text{vrai} ;$$

Phase 4 : En utilisant la formule (*), on détermine une solution optimale de valeur OPT comme suit :

Pour k = n .. 1, si x_{k-1} $(t_1, t_2, ..., t_{i-1}, t_i\text{-}p_j, t_{i+1}, \ldots, t_m)$ = vrai alors T_k est affectée à M_i .

L'algorithme est en $O(nc^m)$. Pour m arbitraire et des données codées en unaire, il est pseudo polynomial et $P_m||C_{max}$ est NP-complet au sens faible.

Cet algorithme s'applique aussi pour les fonctions objectifs de type L_{max}, Σw_jC_j et Σw_jU_j [26].

Remarques : - Les méthodes exactes sont utilisées pour des instances de taille moyenne, dès que la taille est grande, elles deviennent gourmandes en temps de calcul et non pratiques.

- Une solution proche de l'optimum mais obtenue en temps raisonnable, vaut mieux parfois qu'une solution optimale nécessitant un temps trop élevé.

2.2 Les méthodes approchées

Le but d'une méthode approchée est de trouver une solution réalisable, tenant compte de la fonction objectif mais sans garantie d'optimalité. Son utilisation offre de multiples avantages par rapport à une méthode exacte, citons :

- Elles sont plus simples et plus rapides à mettre en œuvre quand la qualité de la solution n'est pas trop importante.

- Elles sont plus souples dans la résolution des problèmes réels. En pratique, il arrive souvent que l'on doive prendre en considération de nouvelles contraintes qu'on ne pouvait pas formuler dès le départ. Ceci peut être fatal pour une méthode exacte si les nouvelles contraintes changent les propriétés sur lesquelles s'appuyait la méthode.

- Elles fournissent des solutions et des bornes qui peuvent être utiles dans la conception de méthodes exactes.

LEE et SHAW (2000) [27] divisent les méthodes approchées en celles de *recherche par voisinage* et celles *constructives*. Pour les premières, on détermine une solution initiale qu'on améliore à chaque itération, alors que pour les secondes, à chaque itération, on complète une solution partielle pour obtenir une solution finale.

Pour les problèmes d'ordonnancement, si on cherche une permutation optimale des tâches, l'espace des solutions admissibles comporte n! permutations. Un exemple de voisinage d'une solution est défini par toutes les permutations obtenues en échangeant deux tâches consécutives de la permutation.

Les algorithmes de listes sont un exemple type de méthodes approchées constructives. Ces algorithmes établissent une liste de priorité. L'ordonnancement est construit en exécutant les tâches l'une après l'autre, tout en respectant l'ordre de la liste.

Les méthodes approchées diffèrent aussi, les unes des autres, par le type de compromis qualité / complexité qu'elles offrent. Nous en distinguons deux classes : les *heuristiques* dont la qualité de la solution qu'elles déterminent ne peut être estimée qu'à posteriori et de façon expérimentale. Et celles appelées *algorithmes polynomiaux à garantie de performances* dont on peut estimer à priori la qualité de la solution qu'elles fournissent.

2.2.1 Les méta heuristiques

L'inconvénient des méthodes de voisinage est qu'elles restent parfois bloquées dans un minimum local, les méta heuristiques dépassent cet inconvénient. L'idée consiste à autoriser une détérioration temporaire de l'objectif, permettant ainsi de quitter un minimum local.

On décrit brièvement deux grandes idées de méta heuristiques : le recuit simulé et la recherche tabou. Une grande partie du cinquième chapitre est consacrée aux algorithmes génétiques.

2.2.1.1 Recuit simulé

L'idée de base de cette heuristique provient de l'opération de recuit, courante en métallurgie et dans l'industrie du verre. Après avoir fait subir des déformations au métal (par exemple après avoir mis en bobine une tôle d'acier laminé), on réchauffe celui-ci à une certaine température, de manière à faire disparaître les tensions internes causées par les déformations, puis on laisse refroidir lentement. L'énergie fournie par le réchauffement permet aux atomes de se déplacer légèrement et le refroidissement lent fige peu à peu le système dans une structure d'énergie minimale.

Cette idée se transpose assez naturellement en optimisation combinatoire pour modifier un algorithme de recherche par voisinage. On autorise des augmentations de la valeur objectif même importantes. A mesure que le temps passe, on autorise les augmentations de plus en plus rarement.

2.2.1.2 Recherche tabou

Comme le recuit simulé, il s'agit au moins dans sa version de base, d'une variante de l'algorithme de recherche par voisinage. Tant que l'on se trouve pas dans un optimum local, la recherche tabou se comporte comme toute méthode de voisinage et améliore à chaque étape la valeur objectif. Lorsque l'on atteint par contre un minimum local, on se permis de passer au moins mauvais des voisins.

L'inconvénient de cette dernière démarche est que parfois, une itération ne suffit pas pour s'en sortir d'un minimum local et un déplacement peut provoquer un déplacement inverse à une étape ultérieure, ce qui provoquera un cycle autour du minimum local. C'est pour cette raison que RT garde en mémoire les dernières solutions visitées pour éviter d'y revenir, c'est ce qu'on appelle *liste tabou*.

2.2.2 Evaluation des performances des méthodes approchées

Les performances d'une méthode approchée sont liées à la qualité de la solution produite et au ressources de calculs (temps et espace mémoire) nécessaires pour l'obtenir. Ces deux critères étant opposés, il s'agit de trouver un compromis au cas par cas en considérant les spécificités du problème.

Suivant la qualité des solutions qu'elles donnent, différentes approches ont été introduites pour apprécier la performance des méthodes approchées et pour les comparer entre elles.

2.2.2.1 Analyse du plus mauvais cas

Dans des applications temps réel, le comportement d'une méthode dans le pire des cas ou l'évaluation de la plus grande erreur possible par rapport à l'optimum est utile. La solution approchée est évaluée à l'aide d'une mesure d'approximation définie pour chaque instance du problème. La plus utilisée est le *rapport d'approximation*.

Soit un algorithme A fournissant, pour toute instance I, une solution réalisable de valeur $C^A(I)$. Le rapport d'approximation, noté $\rho(I)$, vérifie : $C^A(I) = \rho(I).OPT(I)$, où OPT(I) est la valeur de la solution optimale. On dit alors que A est *ρ-approximatif*, si $\rho(I) \leq \rho \ \forall I$.

GRAHAM [28] fut le pionnier de ce type d'analyse. Ses résultats sont présentés, pour le problème $P|prec|C_{max}$, au quatrième chapitre.

2.2.2.2 Analyse en moyenne

Si des paramètres du problème sont des variables aléatoires de distribution de probabilité connue, une étude en moyenne est faite. On calcule l'espérance de l'écart entre la valeur de la solution approchée et celle de la solution optimale.

Exemple 2.4 : L'algorithme LPT (*longuest procesing time first*), utilisé pour résoudre $P_2||C_{max}$, consiste à établir une liste de priorité des tâches dans l'ordre croissant des durées d'exécution et affecter les tâches l'une après l'autre selon cette liste. COFFMAN et al [29] ont analysé, en moyenne, les résultats de cet algorithme. Les durées d'exécution des tâches sont uniformes sur [0,1]. On a :

$$\frac{1}{4}+\frac{1}{(n+1)} \leq E(C_{max}^{LPT}) \leq \frac{n}{4}+\frac{e}{2(n+1)} \qquad [29]$$

Où e est la base du logarithme naturel.

Sachant que $E(C_{max}^{*}) \leq \frac{n}{4}$, on obtient alors que:

$$\frac{E(C_{\max}^{LPT})}{E(C_{\max}^{*})} \leq 1 + O(\frac{1}{n^2}) \qquad [29]$$

Cette borne peut être généralisée au problème $P_m || C_{max}$ et devient:

$$E(C_{\max}^{LPT}) \leq \frac{n}{2m} + O(\frac{m}{n}) \qquad [30]$$

2.2.2.3 Analyse empirique

Généralement, le plus mauvais cas est une instance rare à déterminer et à réaliser. Les résultats de ce type d'analyse ne donnent pas un bon aperçu sur le comportement de la méthode avec des instances plus représentatives du problème. L'analyse en moyenne essaye de surpasser cet inconvénient mais nécessite la connaissance des distributions de probabilité sur les instances du problème. Les méthodes approchées sont souvent évaluées expérimentalement. La méthode sera testée sur un échantillon d'instances du problème mais il n'y a aucune garantie sur les performances de la méthode avec d'autres instances. C'est ce type d'analyse qu'on utilise dans notre étude au cinquième chapitre.

Dans le cas où il est difficile d'obtenir une solution optimale, la solution approchée peut être comparée à une borne inférieure connue, à condition que cette borne soit proche de l'optimum.

CHAPITRE 3

L' APPROXIMATION POLYNOMIALE

Beaucoup de problèmes d'optimisation sont NP-difficiles. Une façon de les résoudre est parfois de se contenter d'une solution presque optimale. On mesure la qualité d'une solution par l'analyse du plus mauvais cas (*worst case analysis*), ça nous garantit l'obtention d'une solution d'une certaine qualité. L'approximation polynomiale conçoit et analyse les algorithmes polynomiaux avec une garantie de performance.

3.1 Généralités et définitions

Définition 3.1 : Pour un problème P de minimisation, un algorithme A est dit *ρ-approximatif* s'il donne une solution avec une valeur au plus ρ.OPT pour toute instance I de P, OPT étant la valeur de la solution optimale. ρ est appelé *garantie de performance* ou *rapport du plus mauvais cas* de l'algorithme (*performance guarentee or worst case ratio*).

Ce qui se traduit par l'inéquation : $C^A(I) \leq \rho.OPT(I), \forall I$ où $C^A(I)$ est la valeur de la solution donnée par l'algorithme A.

ρ doit être supérieur ou égal à 1 (dans le cas d'une maximisation, ρ est inférieure ou égal à 1). Dans ce cas, il existe ε, $\varepsilon \geq 0$ tel que ρ est représenté par $1+\varepsilon$. Plus ρ tend vers 1, plus l'algorithme est meilleur. On s'intéressera aux algorithmes approximatifs qui s'exécutent en temps polynomial.

Supposons que pour tout ρ, $\rho > 1$, il existe un algorithme ρ-approximatif pour résoudre un problème donné. Une famille de $(1+\varepsilon)$-algorithmes approximatifs polynomiaux est appelée *schéma d'approximation en temps polynomial* (*polynomial approximation scheme*) ou PAS. Si la complexité d'un PAS est aussi bornée polynomialement en $\frac{1}{\varepsilon}$, le schéma est dit *complet* (*fully polynomial time approximation scheme*) noté FPAS.

Les problèmes qui admettent respectivement un PAS ou un FPAS constituent des classes appelées respectivement *classe PAS* ou *classe FPAS*. Pour éviter toute confusion, ces classes seront appelées *PAS* ou *FPAS*.

Remarques : - Un algorithme approximatif est dit *bon* s'il admet un ρ constant. Les FPAS sont les meilleures.

- Les algorithmes polynomiaux exactes sont trop bons pour être vrais pour un problème NP-difficile, sauf si P = NP, le meilleur qu'on peut espérer pour un problème NP-difficile est un FPAS.

- Le résultat le plus puissant pour un problème NP-difficile au sens fort est un PAS [31].

Pour le problème $P_2||C_{max}$, illustrons par l'exemple de l'algorithme *tie break*, qui est $\frac{3}{2}$-approximatif [11].

Exemple 3. 1 : Les tâches sont supposées ordonnées dans l'ordre décroissant des durées d'exécution, $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n$. L'affectation des tâches aux machines est comme suit: la première machine reçoit la première tâche, ayant la plus grande durée d'exécution, après quoi la seconde machine reçoit à son tour les deux tâches suivantes, en recommençant avec la première machine, et ainsi de suite. Cet algorithme essaye d'équilibrer les charges des deux machines.

La première machine reçoit donc $T_1, T_4, T_5, T_8, T_9, \ldots$

La seconde machine reçoit $T_2, T_3, T_6, T_7, T_{10}, T_{11}, \ldots$

Proposition 3.1 : [11] Cet algorithme a un rapport du plus mauvais cas égale à $\frac{3}{2}$ et cette borne est atteinte.

Preuve : $p_1 \leq OPT$ et $\frac{1}{m}\sum_{j=1}^{n} p_j \leq OPT$ sont deux bornes triviales. Soient L_1 et L_2 les charges respectives de M_1 et M_2 et C^{tb} la valeur de la solution de l'algorithme tie break. Il est utile de voir que :

$$L_2 + L_2 + L_2 = 3L_2 \leq \frac{\Sigma p_j}{2} + \frac{\Sigma p_j}{2} + \frac{\Sigma p_j}{2} \leq 2\Sigma p_j$$

d'où $$L_2 \le \frac{2}{3}\sum_{j=1}^{n} p_j \le \frac{4}{3}\text{OPT}$$

et $$L_1 \le p_1 + \frac{1}{2}\sum_{j=2}^{n} p_j \le \frac{p_1}{2} + \frac{1}{2}\sum_{j=1}^{n} p_j \le \frac{3}{2}\text{OPT}$$

donc, $$C^{tb} = \max\{L_1, L_2\} \le \frac{3}{2}\text{OPT}.$$

Cette borne est atteinte en posons $n = 4k + 1$ où k est un entier et $p_1 = n - 1$, $p_2 = p_3 = \ldots = p_n = 1$. Une solution optimale est celle qui affecte T_1 à l'une des machines et le reste des tâches à l'autre.

$$\text{OPT} = n - 1 = 4k$$

$$C^{tb} = p_1 + \frac{4k}{2} = 4k + 2k = 6k = \frac{3}{2}\text{OPT} \qquad \square$$

Une autre mesure d'approximation est employée, le *rapport différentiel du plus mauvais cas* [38] noté γ. Il est définit par :

$$\omega(I) - C^A(I) \le \gamma.(\omega(I) - \text{OPT}(I)), \forall I$$

où $\omega(I)$ désigne la plus mauvaise valeur objectif de l'instance I.

Le rapport différentiel γ mesure la position de la valeur objectif donnée par l'algorithme entre la plus mauvaise et la meilleure des valeurs.

Le rapport du plus mauvais cas ρ reste le plus utilisé. On a au total cinq classes de problèmes NP-difficiles :

PAS et FPAS : Déjà définies auparavant.

ABS : Contient les problèmes absolument approximables. L'écart entre les valeurs des solutions approchée et optimale est, au plus, une valeur fixée indépendamment de la valeur optimale.

$$\exists c, c \ge 0 : C^A(I) - \text{OPT}(I) \le c \ \forall I$$

APX : Contient les problèmes avec un rapport du plus mauvais cas.

f(n)-APX : Contient les problèmes avec un rapport du plus mauvais cas dépendant de la taille du problème.

$$\rho \leq f(n), \text{ avec } n = |I| \text{ et f est une fonction définie.}$$

P étant la classe des problèmes polynomiaux et NP celle des problèmes faciles, difficiles ou non encore classés. Une propriété est que ces sous classes sont ordonnées par inclusion.

Lemme 3.1 : $P \subseteq ABS$ et $P \subseteq FPAS \subseteq PAS \subseteq APX \subseteq$ f(n)-APX $\subseteq NP$.

Sous l'hypothèse $P \neq NP$, toutes les inclusions sont strictes.

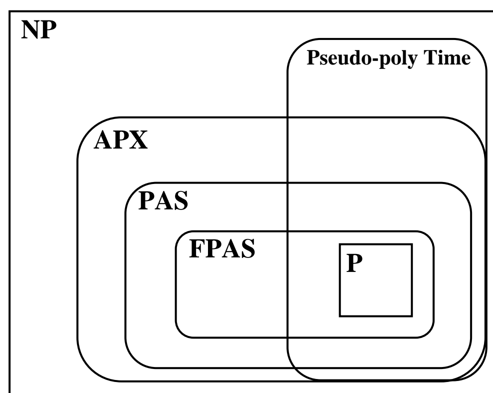


Figure 3.1 : Relation entre les classes d'approximation

Nous appelons *résultat positif* l'établissement de l'existence d'un algorithme d'approximation. Sinon il est dit *résultat négatif*, il désapprouve l'existence des résultats d'approximation.

3.2 La conception d'algorithme ρ-approximatif

Le paramètre principal d'un algorithme approximatif A est le rapport du plus mauvais cas ρ vérifiant :

$$C^A \leq \rho.OPT$$

Une façon d'obtenir le ρ de cette formule est de déterminer une borne inférieure LB et de la comparer avec la valeur de la solution de l'algorithme A. Ce qui devient :

1) $LB \leq OPT$

2) $C^{A} \leq \rho.LB$

Une borne inférieure de OPT est obtenue en relaxant le problème d'origine en relâchant quelques contraintes.

3.3 La conception de schéma d'approximation polynomiale

Soient P un problème NP-difficile et A son algorithme de résolution exacte. Au trois parties du diagramme de la figure 3.2, à gauche l'entrée ou l'instance I du problème, à droite la sortie ou la solution A(I) et au milieu l'exécution de l'algorithme A, correspondent trois approches de conception d'un schéma d'approximation polynomial : *arrondir les données d'entrée*, *arrondir les données de sortie* et *modifier l'exécution d'algorithme*.

Figure 3.2: Diagramme de résolution de P par A

3.3.1 Arrondir les données d'entrée

On transforme l'instance difficile I en une instance simplifiée, notée IS, facile à résoudre. Par une procédure retour arrière (*backtracking*), on déduit, à partir de la solution optimale de l'instance simplifiée, une solution de l'instance d'origine. Cette approche est décrite par les étapes de simplification, de résolution et de '*backtracking*'.

L'instance simplifiée est obtenue en simplifiant les données d'entrées et se fait en:

- Arrondissant les données du problème qui peuvent être les durées d'exécution des tâches.
- Représentant des tâches de l'instance d'origine par une seule tâche de l'instance simplifiée.
- Eliminant les tâches qui causent la difficulté de l'instance.
- , Etc.

Donnons une construction d'un PAS pour le problème $P_2||C_{max}$.

Exemple 3.3 : Un PAS pour $P2||C_{max}$ [32].

On note Ps la somme des durées d'exécution de toutes les tâches, $P_s = \sum_{j=1}^{n} p_j$ et P_m la durée d'exécution de la plus longue tâche, $P_m = \max_{j=1..n} p_j$. $\frac{1}{2}P_s$ et P_m sont des bornes inférieures triviales de la valeur de la solution optimale OPT. L = max $\{\frac{1}{2}P_s, P_m\}$ représente aussi une borne inférieure de OPT, $L \leq OPT$.

Construction de l'instance simplifiée IS : On classe les tâches en 'courtes' et 'longues'. Une tâche T_j est dite *longue* si $p_j > \varepsilon L$, $\varepsilon \geq 0$ et *courte* si $p_j \leq \varepsilon L$.

IS contient les tâches longues de I et des tâches dites *auxiliaires* de durées εL. Le nombre de ces dernières est de $\lfloor \frac{S}{\varepsilon L} \rfloor$ où $\lfloor . \rfloor$ est l'arrondit inférieur d'un nombre et S, la somme des durées des tâches courtes dans I.

Où se situe la valeur optimale de IS, notée OPTS, par rapport à celle de I? Dans un ordonnancement optimal σ^* de I, on note S_i (i = 1, 2) la somme des durées des tâches courtes sur la machine M_i. On construit un ordonnancement réalisable de IS en laissant les tâches longues dans σ^* et en remplaçant les tâches courtes par $\lceil \frac{S_i}{\varepsilon L} \rceil$ tâches de durées εL, où $\lceil . \rceil$ est l'arrondit supérieur d'un nombre. Sachant que :

$$\lceil \frac{S_1}{\varepsilon L} \rceil + \lceil \frac{S_2}{\varepsilon L} \rceil \geq \lfloor \frac{S_1}{\varepsilon L} + \frac{S_2}{\varepsilon L} \rfloor = \lfloor \frac{S}{\varepsilon L} \rfloor,$$

La charge de M_i augmente d'au plus :

$$\lceil \frac{S_i}{\varepsilon L} \rceil \varepsilon L - S_i \leq (\frac{S_i}{\varepsilon L} + 1)\varepsilon L - S_i = \varepsilon L$$

l'ordonnancement obtenu est réalisable, et :

$$OPTS \leq OPT + \varepsilon L \leq (1 + \varepsilon)OPT$$

Résolution de l'instance simplifiée : En remplaçant les tâches courtes par les tâches auxiliaires, la somme des durées d'exécution P_s reste inchangée. Puisque les tâches dans IS sont de durées au moins εL, on a au plus $\frac{P_s}{\varepsilon L}$ tâches, et :

$$\frac{P_s}{\varepsilon L} \leq \frac{2L}{\varepsilon L} = \frac{2}{\varepsilon}$$

Pour ε fixé, le nombre de tâches dans IS est constant et indépendant de celui dans I. On a qu'à énumérer toutes les solutions possibles. Chacune des $\frac{2}{\varepsilon}$ tâches est affectée à l'une des deux machines. Il y a au plus $2^{\frac{2}{\varepsilon}}$ ordonnancements possibles, qu'on peut évaluer en $o(\frac{2}{\varepsilon})$. On peut résoudre IS en temps borné par une constante.

'Backtracking' : Reste à obtenir un ordonnancement réalisable σ pour I à partir d'un ordonnancement optimale σ_s de IS . Dans σ_s et pour la machine M_i, on note : LS_i la charge, BS_i et SS_i sont, respectivement, la somme des durées des tâches longues et auxiliaires. Il est clair que :

$$LS_i = BS_i + SS_i \quad \text{et} \quad SS_1 + SS_2 = \varepsilon L \left\lfloor \frac{S}{\varepsilon L} \right\rfloor > S - \varepsilon L \qquad (2)$$

On construit un ordonnancement σ pour I comme suit : Les tâches longues gardent la même machine que dans σ_s. Pour les tâches courtes, on réserve deux intervalles de temps sur M_1 et M_2, de durées respectives $SS_1 + 2\varepsilon L$ et SS_2. On commence par M_1, on affecte les tâches l'une après l'autre dans l'intervalle, et on arrête lorsqu'on rencontre une tâche qui ne tient pas. La somme des durées des tâches courtes affectées à M_1 est au moins $SS_1 + \varepsilon L$. La somme des durées des tâches restantes est donc $S – SS_1 - \varepsilon L$. D'après (2), le deuxième intervalle de durée SS_2 peut contenir les tâches restantes. Ce qui termine la construction de σ.

On compare les charges de M_1 et M_2 dans σ avec celles dans σ_s. Puisque la somme des durées des tâches courtes dans M_i est au plus $LS_i + 2\varepsilon L$ alors :

$$L_i \leq BS_i + (SS_i + 2\varepsilon L) = LS_i + 2\varepsilon L \leq (1 + \varepsilon)OPT + 2\varepsilon.OPT = (1 + 3\varepsilon).OPT$$

On peut rapprocher autant qu'on le veut 3ε de 0, on a alors un PAS pour $P_2 | | C_{max}$.

3.3.2 Arrondir les données de sortie

L'idée est de partitionner l'ensemble des solutions réalisables en plusieurs sous ensembles pour lesquels on sait déterminer facilement une solution approchée. Cette approche est décrite en trois étapes :

- Partitionner l'ensemble des solutions F en d sous ensembles F_1, F_2, …,F_d tel que $F = \bigcup_{k=1}^{d} F_k$. Cette partition dépend de la précision voulue ε. Le nombre d des sous ensembles doit être borné polynomialement en fonction de l'entrée.
- Déterminer en temps polynomial, pour chaque sous ensemble F_k, une solution de valeur C_k.
- Choisir la meilleure des solutions comme une solution approchée de I, de valeur C.

$$C = \min C_k \ (k = 1..d)$$

Pour illustrer la technique, on construit un deuxième PAS pour $P_2||C_{max}$.

Exemple 3.4 : Un PAS pour $P_2||C_{max}$ [32].

On garde les notations précédentes de P_s, P_m et $L = \max\{P_s,P_m\}$, on a toujours :

$$L \leq OPT \qquad (1)$$

La partition : Soit une instance I de $P_2||C_{max}$ et le paramètre de précision ε>0. On utilise les définitions précédentes de tâches courtes et longues. Deux ordonnancement σ_1 et σ_2 sont dans le même sous ensemble s'ils ont la même affectation des tâches longues aux machines.

De même que pour dans l'exemple précédent, il y a au plus $\frac{2}{\varepsilon}$ tâches longues, et on a $2^{\frac{2}{\varepsilon}}$ affectations possibles. Donc, le nombre de sous ensembles est polynomial, mieux encore ce nombre est borné par une constante.

Une solution de chaque sous ensemble : On considère un sous ensemble F_k, et on note OPT(k) la valeur de sa meilleure solution. L'affectation des tâches longues est fixée, on note $B_i(k)$, i = 1, 2, la somme des durées des tâches longues affectées à M_i. On a :

$$T = \max \{ B_1(k), B_2(k) \} \leq OPT(k) \quad (2)$$

On affecte les tâches courtes l'une après l'autre à la machine de charge minimale. On obtient l'ordonnancement σ(k) de valeur APP(k).

Où se situe APP(k) par rapport à OPT(k)? D'après (2), si APP(k) = T alors σ(k) est une solution optimale. Sinon, APP(l) > T. Considérons la machine M de charge maximale dans σ(l). La dernière tâche affectée à M est courte (de durée inférieure à εL). Lorsque cette tâche a été affectée à M, la charge de M était d'au plus $\frac{1}{2}$Ps (la plus petite charge était de M). En utilisant (1), on a :

$$APP(k) \leq \frac{1}{2} P_s + \varepsilon L \leq (1 + \varepsilon)L \leq (1 + \varepsilon)OPT \leq (1 + \varepsilon)OPT(k)$$

Dans les deux cas (APP(k) = T ou APP(k) > T), la valeur de la solution déterminée est au plus (1 + ε) facteur multiplicatif de la meilleure valeur de F_k.

Une solution pour I : On choisit parmi toute les solutions σ(k), la meilleure solution.

3.3.3 Modifier l'exécution d'algorithme

L'algorithme de résolution exacte A est lent. L'idée principale est d'agir sur les données d'exécution intermédiaires. Ignorer une partie de ces données, accélère l'algorithme, voire, le rend polynomial.

Dans l'exemple suivant, un troisième PAS pour $P_2||C_{max}$ est construit en modifiant l'exécution d'un programme dynamique.

Exemple 3.5 : Un PAS pour $P_2||C_{max}$ [11].

L'algorithme de résolution exacte : Reprenant le programme dynamique du deuxième chapitre désigné pour résoudre $P_2||C_{max}$. Il calcule, récursivement, la liste des valeurs possibles de la charge de M_1. Il choisit la meilleure entre ces valeurs et détermine la solution optimale correspondante par une procédure retour arrière (*backtracking*).

Algorithme 3.1 : Un programme dynamique pour $P_2 \mid\mid C_{max}$.

//Initialisation

$L_k = \{0\}$;

//Calculer les ensembles L_k récursivement

Pour k allant de 1 à n faire

$L_k = L_{k-1} \cup \{x + p_k / x \in L_{k-1}\}$;

//Calculer la valeur de la solution optimale

$$C = \min_{x \in L_n} \{ x - \frac{1}{2}\sum_{j=1}^{n} p_j \} \} ;$$

//Calculer la solution optimale par 'backtracking'

Pour k allant de 1 à n faire

Si C est dans L_{k-1} alors T_k est affectée à M_2

Sinon T_k est affectée à M_1 et $C := C - p_k$;

Modifier l'exécution de l'algorithme : Les données intermédiaires du programme dynamique sont les éléments des listes L_k. Imposons directement que la cardinalité de L_k, k = 1..n soit un polynôme de l'entrée. Pour cela, on utilise une procédure ROUND qui arrondit inférieurement les éléments de la liste au multiple suivant de $1 + \delta$, avec $\delta = \frac{\varepsilon}{2n}$. La relation de récurrence devient :

$$L_k = L_{k-1} \cup \text{ROUND } \{x + p_k \ / x \in L_{k-1}\}$$

On calcule une solution de valeur C_{max}. Dans une liste L_k , le nombre d'éléments est celui des différentes puissances. Il est, au plus, en O(l), où :

$$l = \left\lceil \frac{2n}{\varepsilon} \log(np_{max}) \right\rceil, \text{ puisque}$$

$$(1+\varepsilon)^1 \geq ((1+\frac{\varepsilon}{2n})^{\frac{2n}{\varepsilon}})^{\log(np_{max})}$$

$$\geq 2^{\log(np_{max})}$$

$$\geq n.p_{\max}$$

$$\geq \sum_{j=1}^{n} p_j$$

Le temps d'exécution de cet algorithme est polynomial en entrée et en $\frac{1}{\varepsilon}$. De plus, ROUND fait à chaque étape k, (k = 1..n), une erreur qui nous éloigne de la solution optimale d'au plus un facteur multiplicatif $1 + \delta$.

$$(1+\delta)^n = (1+\frac{\varepsilon}{2n})^n \leq e^{\frac{\varepsilon}{2}} \leq e^{\ln(1+\varepsilon)} = 1+\varepsilon,$$

L'erreur finale est dans un facteur multiplicatif $1 + \varepsilon$.

$$C_{max} \leq (1 + \varepsilon)\ OPT$$

Remarquer que l'algorithme est polynomial en $\frac{1}{\varepsilon}$ donc on a un FPAS.

3.4 Les résultats négatifs

Il est parfois possible, pour des problèmes difficiles, de prouver les limites de l'approximation polynomiale. Pour trouver de tels résultats, on montre que l'existence d'un algorithme ρ-approximatif permet de résoudre un problème NP-Complet en temps polynomial. et donc P = NP.

Pour illustrer ceci, considérons le *problème du voyageur de commerce* (PVC).

Problème PVC : Soit un graphe complet avec des coûts positifs sur les arcs. On cherche la tournée de longueur minimale, visitant chaque sommet une et une seule fois.

Ce problème n'admet pas un algorithme ρ-approximatif $\forall\rho$, sinon le problème de l'existence du circuit Hamiltonien dans un graphe, connu NP-Complet, est résolu en temps polynomial.

Problème du circuit Hamiltonien : Soit un graphe G(V,U). Existe-t-il un circuit passant par tous les sommets, une et une seule fois ?

Preuve : [33] On résout PVC en prenant des coûts 1 si l'arc est dans U et $+\infty$ sinon. Si la valeur de la solution approchée est $+\infty$, la réponse au problème du circuit Hamiltonien est non, sinon la réponse est oui. Le problème du circuit Hamiltonien est alors résolu.

3.4.1 La non existence d'un FPAS

Théorème 3. 1 : [31] Un problème NP-difficile au sens fort n'admet pas un FPAS .

Preuve : Pour une preuve facile, voir [32].

On se base sur ce théorème, montrer qu'un problème n'admet pas un FPAS revient à montrer qu'il est NP-difficile au sens fort.

3.4.2 La technique de l'intervalle (*gap technique*)

Pour montrer la 'non existence' d'un algorithme approximatif pour un problème d'ordonnancement donné, la *technique de l'intervalle* est utilisée [34][35][36].

Elle est basée sur la construction d'une transformation, en temps polynomial, entre un problème décisionnel NP-complet où la solution est une réponse positive ou négative (*yes* ou *no*) et le problème d'ordonnancement. Les instances positives (*yes instances*) sont transformées en des instances à valeurs objectifs au plus 'a', et les instances négatives (*no instances*) en des instances à valeurs objectifs au moins 'b', avec b > a. Un algorithme d'approximation polynomiale du problème d'ordonnancement, avec une garantie de performance strictement inférieure à $\frac{b}{a}$, permet d'identifier, en temps polynomial, les instances positives et négatives du problème décisionnel. Par conséquence, le problème d'ordonnancement n'admet pas un algorithme d'approximation polynomiale avec $\rho < \frac{b}{a}$ à moins que P = NP.

Cette transformation est appelée *réduction polynomiale*. Elle crée un intervalle qui sépare, d'une part, les valeurs objectifs du problème d'ordonnancement correspondantes aux instances positives, et d'autre part, celles correspondantes aux instances négatives. Voir figure 3.3. Cet intervalle est exploité pour minorer le rapport du plus mauvais cas.

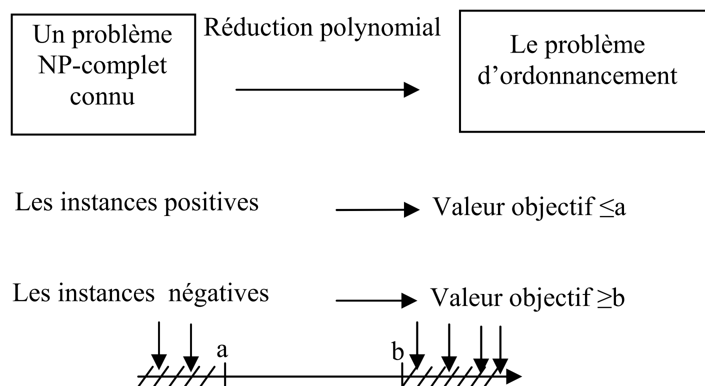


Figure 3.3: Technique d'intervalle

Théorème 3. 2 : [32] Soient X un problème décisionnel NP-complet, Y un problème de minimisation et 'r' une réduction polynomiale des instances de X vers les instances de Y, qui vérifie pour a < b les conditions :

i) Chaque instance positive de X est transformée en une instance de Y de valeur objectif au plus a.

ii) chaque instance négative de X est transformée en une instance de Y de valeur objectif au moins b.

Le problème Y n'admet pas un algorithme ρ-approximatif avec $\rho < \frac{b}{a}$, sauf si P = NP.

Preuve : Par contraposé. Supposons qu'il existe un algorithme ρ-approximatif pour résoudre Y avec $\rho < \frac{b}{a}$. Soit I une instance de X et r(I) l'instance de Y obtenue en réduisant I. On résout r(I) par l'algorithme approximatif.

- Si I est une instance positive alors la valeur optimale de r(I) est au plus a, et donc la valeur de la solution approximative est strictement inférieur à $\frac{b}{a}a = b$.
- Si I est une instance négative alors la valeur optimale de r(I) est au moins b, et la valeur de la solution approximative est au moins b.

En résumé, on distingue, en temps polynomial, entre les instances positives (le coût d'une solution approximative de r(I) est au moins b) et négatives (le coût d'une solution approximative de r(I) est strictement inférieure à b), et donc P = NP. □

3.4.3 L'utilisation de l'APX-difficulté

En théorie de la complexité, la réduction polynomiale permet de montrer qu'un problème est au moins aussi difficile qu'un autre problème. Par analogie, dans la théorie de l'approximation polynomiale, une réduction qui préserve l'approximation dans un facteur multiplicatif constant, appelée *réduction préservant l'approximation*, permet de montrer qu'un problème est au moins aussi difficile à approximer qu'un autre problème.

Des réductions préservant l'approximation ont été définies dans la littérature, citons : *A-Réduction*, *F-Réduction*, *L-Réduction*, *P-Réduction*, *R-Réduction*. On présente ici la L-réduction pour son caractère pratique [32].

Définition 3.2 : [32] Soient X et Y deux problèmes d'optimisation. Une L-Réduction de X à Y est un couple de fonctions (R, S). R une fonction de l'ensemble des instances de X vers celui de Y. S une fonction de l'ensemble des solutions des instances de Y vers celui de X. R et S sont calculées en temps polynomial et vérifient :

i) Pour toute instance I de X de coût optimal OPT(I), R(I) est une instance de Y de coût optimal OPT (R(I)), tel que $\exists\ \alpha > 0 : OPT(R(I)) \leq \alpha OPT(I)$.

ii) Pour toute solution réalisable s de R(I), S(s) est une solution réalisable de I tel que : $\exists\ \beta > 0 : | c(S(s)) - OPT(I) | \leq \beta | c(s) - OPT(R(I)) |$
avec c(S(s)) et c(s) les valeurs objectifs des solutions S(s) et s respectivement.

Les fonctions R et S assurent une liaison étroite entre l'approximation du problème X et celle de Y.

Théorème 3.3 : [39] On suppose qu'il existe une L-réduction d'un problème X vers un problème Y de paramètres α et β . S' il existe un algorithme $(1 + \varepsilon)$-approximatif pour Y alors il existe un algorithme $(1 + \alpha\beta\varepsilon)$-approximatif pour X.

Preuve : Voir [32].

Théorème 3.4 : [31] Soit une L-réduction d'un problème X vers un problème Y. Si Y admet un PAS alors X admet un PAS.

Preuve : Voir [32].

La L-réduction maintient la non-existence d'un PAS comme le fait la réduction polynomiale avec la non-existence d'un algorithme polynomial. On rappelle que APX est la classe des problèmes possédant un algorithme ρ-approximatif. Elle joue, avec la L-réduction, le même rôle joué par la classe NP avec la réduction polynomiale. Un problème X est dit *APX-difficile* si pour tout problème de APX, il existe une L-réduction de ce problème vers X. Plusieurs problèmes d'optimisation ont été montré APX-difficiles. Si on trouve un PAS pour l'un d'eux, on a un PAS pour tous les autres.

Théorème 3.4 : [37] Si un problème APX-difficile possède un PAS alors P = NP.

Preuve : Voir [32].

Par Conséquence, pour montrer qu'un problème Y ne possède pas un PAS à moins que P = NP, il suffit de donner une L-réduction d'un problème APX-difficile vers Y.

Remarque : Dans la pratique, le recours aux heuristiques avec aucune garantie de performance, reste majoritaire. Ceci peut être attribué, non seulement, à leur bon comportement mais aussi à une relative méconnaissance des particularités de l'approximation polynomiale. L'autre raison est le pessimisme de certains résultats négatifs qui provient essentiellement des restrictions que ce cadre impose. Soulignons, néanmoins, que les méthodes polynomiales peuvent avoir dans la pratique, des résultats bien meilleurs que les bornes garanties théoriquement. D'autant plus qu'en combinant une heuristique et un algorithme polynomial à garantie de performance, on peut bénéficier du bon comportement en pratique de l'heuristique et des garanties d'approximation du second. C'est ce qu'on fait au chapitre cinq.

CHAPITRE 4
OUTILS DE RESOLUTION DU PROBLEME $P|prec|C_{max}$: LES METHODES DE LISTES ET LES ALGORITHMES GENETIQUES

Deux types de méthodes approchées sont présentés dans ce chapitre: les méthodes de listes, déterministes et très rapides, et qui garantissent une certaine qualité de la solution obtenue. Et les algorithmes génétiques, qui ont un caractère stochastique, et qui n'offrent aucune garantie sur la qualité de la solution.

4.1 Les méthodes de listes

Dans le cas de problèmes d'ordonnancement NP-difficiles, les algorithmes les plus utilisés sont basés sur les listes. Ils déterminent pour un ordre de tâches donné par une liste, l'ordonnancement correspondant. Ils considèrent les tâches une par une et prennent la décision d'ordonnancer sur la base d'un ordonnancement partiel de tâches ordonnancées auparavant. Les décisions pour les premières tâches affectées restent inchangées. L'ordonnancement obtenu est le résultat de l'algorithme de type liste. En général il n'est pas optimal. De telles approches sont aussi appelées règles de priorité statique.

Un algorithme de liste se développe en deux étapes:

- Selon une *règle de priorité* on calcule, pour chacune des tâches, une valeur mesurant une urgence dans son exécution. Cela permet de construire une liste qui commence par la tâche la plus prioritaire, suivie d'une tâche moins prioritaire et ainsi de suite jusqu'à la tâche la moins prioritaire. En cas d'égalité de priorités, on choisit la tâche arbitrairement.

- Ayant la liste de priorité et selon une *règle d'affectation* prenant en compte cette liste, on affecte les tâches aux machines disponibles, l'une après l'autre jusqu'à l'épuisement de l'ensemble des tâches.

Dans un problème d'ordonnancement avec des contraintes de précédence, une tâche est dite *prête* si tous ses prédécesseurs ont terminé leurs exécutions. Si l'ordre total donné par

la liste de priorité respecte l'ordre partiel donné par les contraintes de précédence, on dit que la liste est *réalisable*. Si de plus la liste donne un ordonnancement optimal, elle est dite *optimale*.

4.1.1 Règle de priorité

Plusieurs règles de priorité sont définies en littérature. Présentons quelques unes :

- Les tâches les plus courtes d'abord (*shortest processing time*). SPT.
- Les tâches les plus longues d'abord (*longuest processing time*). LPT.
- Les tâches avec des dates au plus tard les plus courtes d'abord (*shortest due date*). SDD.

Exemple 4. 1 : Soit $P_2||C_{max}$ le problème à deux machines identiques M_1, M_2 et huit tâches T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8 de durées respectives 3, 4, 2, 4, 4, 2, 13, 2.

Selon les deux premières règles, les listes obtenues sont :

$$L_{SPT} = T_3\ T_6\ T_8\ T_1\ T_2\ T_4\ T_5\ T_7$$

$$L_{LPT} = T_7\ T_2\ T_4\ T_5\ T_1\ T_3\ T_6\ T_8$$

En cas de contraintes de précédence, il existe des règles de priorités qui prennent en compte le graphe de précédence. Celles dites *de chemin critique* calculent la priorité d'une tâche en fonction du plus long chemin entre cette tâche et une tâche sans prédécesseur. Pour le problème $P|prec|C_{max}$, citons quelques priorités :

SUCC :

En fonction du nombre de successeurs immédiats .

PLC :

En fonction du plus long chemin $PLC(i) = \begin{cases} p_i & si\ i\ n'a\ pas\ de\ successeurs \\ PLC(j) + p_i & si\ j\ successeur\ de\ i \end{cases}$

SOMPLC :

En fonction d'une variante de PLC, $VPLC(i) = \begin{cases} p_i & \text{si i n'a pas de successeurs} \\ \sum_{\text{j successuer de i}} \{ VPLC(j) + p_i \} \end{cases}$

Max{,} :

En fonction de l'expression : $\max \left\{ PLC(i), \frac{\sum_{\text{j descendant de i}} p_j}{m} + p_i \right\}$

Pour l'exemple précédent et avec des contraintes de précédence représentées sur la figure 4.1.

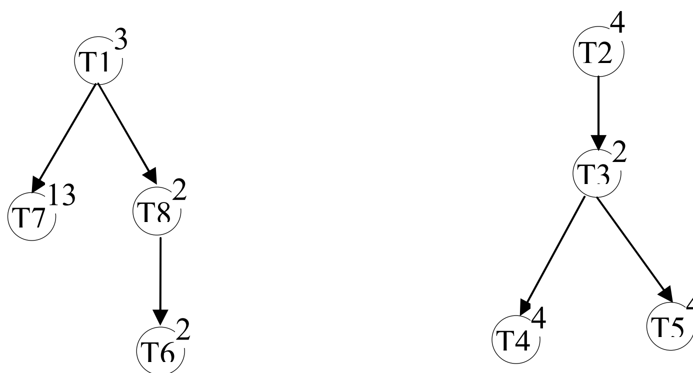


Figure 4.1 : Exemple de $P_2 | prec | C_{max}$

On obtient les listes de priorités :

$L_{Succ} = T_1\ T_2\ T_3\ T_8\ T_4\ T_5\ T_6\ T_7$

$L_{PLC} = T_1\ T_7\ T_2\ T_3\ T_4\ T_5\ T_6\ T_8$

$L_{SOMPLC} = T_1\ T_2\ T_7\ T_3\ T_4\ T_5\ T_8\ T_6$

$L_{Max\{,\}} = T_1\ T_7\ T_2\ T_3\ T_4\ T_5\ T_8\ T_6$

Ces règles sont étudiées en détail au chapitre cinq.

4.1.2 Règle d'affectation

Différentes règles d'affectation peuvent être utilisées. La règle d'affectation dite *avec délai* respecte strictement l'ordre des tâches dans la liste. Elle peut laisser des machines oisives même si des tâches sont prêtes à s'exécuter.

La règle d'affectation la plus utilisée est dite *sans délai*. Elle ne laisse pas des machines oisives si des tâches sont prêtes à s'exécuter.

Sur l'exemple précédent, on applique les deux règles avec la liste de priorité L ; $L = T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7\ T_8$.

Si on respecte strictement l'ordre de la liste en utilisant la règle avec délai, on obtient l'ordonnancement représenté sur la figure 4.2 de longueur vingt deux unités de temps.

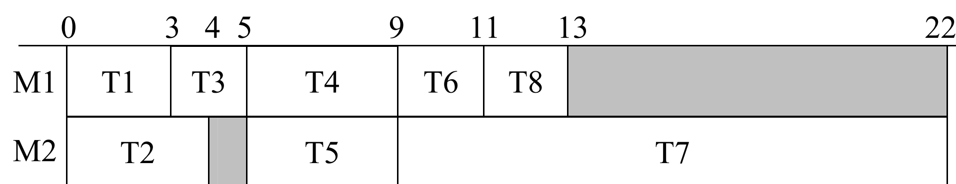


Figure 4.2 : Diagramme de Gantt utilisant la règle avec délai

Si on cherche à occuper les machines en utilisant la règle sans délai, on obtient l'ordonnancement représenté sur la figure 4.3 de longueur dix sept unités de temps.

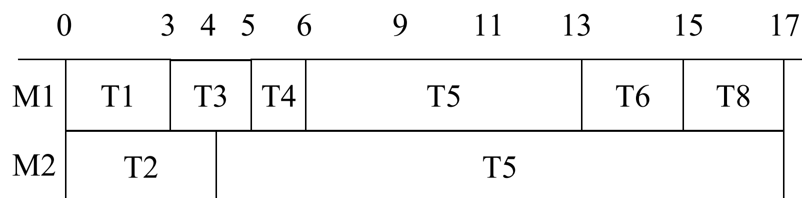


Figure 4.3 : Diagramme de Gant utilisant la règle sans délai

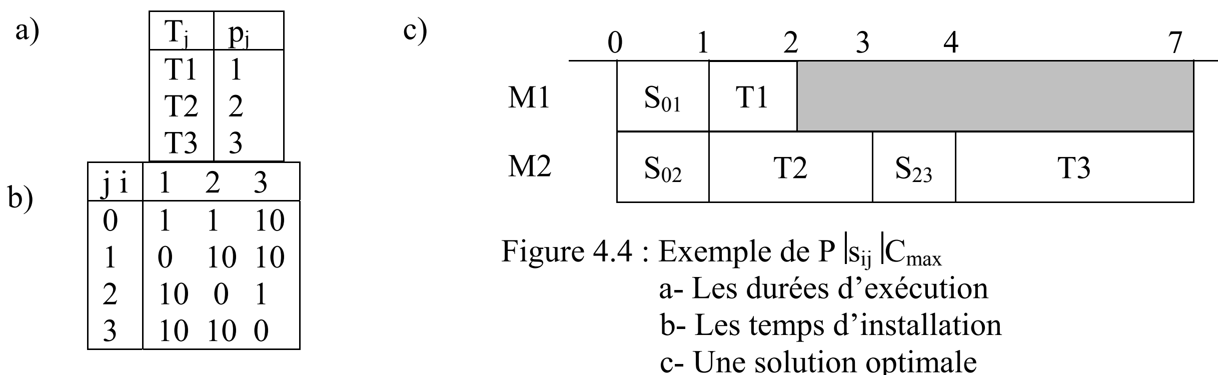
C'est un ordonnancement optimal, L est une liste optimale.

4.1.3 Ensemble de listes dominant

Un ensemble de listes est dit *dominant* s'il contient au moins une liste optimale. La résolution exacte des problèmes possédant cet ensemble revient à chercher la liste optimale dans un ensemble de listes dominant.

SOY [40] donne un exemple du problème $P|s_{ij}|C_{max}$ présentée dans la figure 4.4 où l'ensemble dominant n'existe pas.

Problème $P|s_{ij}|C_{max}$: Minimiser la longueur d'ordonnancement à machines parallèles identiques et des temps d'installation (*setup times*), une tâche 'i' nécessite s_{ij} unités de temps supplémentaires si elle suit une tâche 'j' sur la même machine.



a)

T_j	p_j
T1	1
T2	2
T3	3

b)

j i	1	2	3
0	1	1	10
1	0	10	10
2	10	0	1
3	10	10	0

c)

Figure 4.4 : Exemple de P $|s_{ij}|C_{max}$
a- Les durées d'exécution
b- Les temps d'installation
c- Une solution optimale

SOY [40] proposa une autre règle d'affectation avec laquelle, la liste optimale et l'ensemble dominant existent. Elle prend en compte les temps d'installation et affecte la tâche en tête de liste à la machine où cette tâche se termine rapidement. Dans l'exemple précédent, la liste T_1 T_2 T_3 est optimale.

Remarques :

- Un ensemble de listes peut être dominant avec une règle d'affectation et ne pas l'être avec une autre.

- Les problèmes avec des machines uniformes ou différentes n'admettent pas forcément un ensemble de listes dominant [40]. Pour ceux à machines identiques, l'ensemble dominant existe toujours [36]. Pour le problème P $|prec|C_{max}$, la règle proposée de SOY est équivalente à la règle avec délai.

Sauf mention du contraire, la règle d'affectation utilisée est sans délai.

4.1.4 Une liste dynamique

Des algorithmes d'ordonnancement utilisant une liste dynamique ont été proposés dans la littérature [41], [42], [43]. Dans les algorithmes de listes classiques, la liste est construite avant le début des affectations et reste inchangée. Dans ceux de listes dynamiques, après chaque affectation, les priorités sont recalculées.

Les listes dynamiques donnent des solutions meilleures que celles données par les listes statiques mais la complexité de l'algorithme augmente considérablement.

4.1.5 Utilisation des listes pour résoudre $P|prec|C_{max}$

Depuis les travaux de GRAHAM (1966), les listes sont toujours d'actualité. Au risque de se répéter, on a vu au premier chapitre que pour $P|prec|C_{max}$, la longueur d'un ordonnancement de liste peut augmenter si l'un des changements suivants a lieu:

- Le nombre de machines augmente,
- Les durées d'exécution diminuent,
- Les contraintes de précédence sont affaiblies,
- La liste de priorité change.

GRAHAM [13] a évalué le changement maximal de la longueur d'ordonnancement induit par ces changements.

Soit l'ensemble des contraintes de précédence noté PR et les durées d'exécution données par un vecteur P. La longueur d'ordonnancement obtenu en utilisant une liste L est C_{max}.

Changeons les paramètres comme suit : les durées deviennent $P' \leq P$, l'ensemble des contraintes de précédence devient PR', avec $PR' \subseteq PR$ et le nombre de machines diminue, $m' < m$. En utilisant une autre liste de priorité L', on obtient un ordonnancement de longueur notée C'_{max}.

Théorème 4.1 : [13] Sous les hypothèses précédentes, on a :

$$\frac{C'_{max}}{C_{max}} \leq 1 + \frac{m-1}{m'}$$

Preuve : Voir [15].

4.1.5.1 Algorithme de liste quelconque

On peut du théorème précédent, en déduire le rapport du plus mauvais cas d'une liste quelconque.

Corollaire 4.1 : [13] Le rapport ρ_{LS} d'un algorithme de liste quelconque LS, est :

$$\rho_{LS} = 2 - \frac{1}{m}$$

Preuve : On utilise le résultat du théorème 4.1.

$$\frac{C'_{max}}{C_{max}} \leq 1 + \frac{m-1}{m'}$$

On prend L' comme étant une liste optimale et m' = m.

$$\frac{C'_{max}}{C_{max}} = \rho_{LS} \quad \leq \quad 1 + \frac{m-1}{m} = 2 - \frac{1}{m}$$

La borne est atteinte en considérons l'exemple : n = (m-1).m, p = (1, 1, …,1, m), PR = Ø, L = T_n T_1 T_2 …T_{n-1} et L' = T_1 T_2 …T_n. Sur la figure 4.5 on présente une instance avec m = 4.□

a)

0 – 1	1 – 2	2 – 3	3 – 4	
T13				M1
T10	T7	T4	T1	M2
T11	T8	T5	T2	M3
T12	T9	T6	T3	M4

b)

	0 – 1	1 – 2	2 – 3	3 – 7
M1	T1	T5	T9	T13
M2	T2	T6	T10	
M3	T3	T7	T11	
M4	T4	T8	T12	

Figure 4.5 : Exemple où la borne $\rho_{Ls} = 2 - \frac{1}{m}$ est atteinte.
a- Ordonnancement optimal (L).
b- Ordonnancement de L'.

Pour $P||C_{max}$, on peut montrer le corollaire 4.1 directement.

Preuve : On associe à une tâche T_i, ses dates de début et de fin notées respectivement S_i et C_i. Soit T_k la tâche qui termine son exécution en dernier. On a :

$$C_{max} = S_k + p_k \text{ , } T_k \text{ termine à } C_{max}$$

$$S_k \leq \frac{1}{m} \sum_{j \neq k} p_j \text{, aucune machine n'est libre avant } S_k$$

$$p_k \leq OPT \quad \text{et} \quad \frac{1}{m} \sum_j p_j \leq OPT \text{, des bornes triviales}$$

on a donc :

$$C_{max} = S_k + p_k \leq \frac{1}{m} \sum_{j \neq k} p_j + p_k$$

$$= \frac{1}{m} \sum_j p_j + (1 - \frac{1}{m}) p_k$$

$$\leq OPT + (1 - \frac{1}{m}) OPT$$

$$= (2 - \frac{1}{m}) OPT \quad \square$$

Pour $P|prec|C_{max}$, une preuve du corollaire est dans [44].

Une liste de priorité quelconque est 2-approximatif. Ordonner la liste selon la règle de priorité LPT (les tâches de durée grande en premier) donne une meilleure approximation du problème $P||C_{max}$.

Théorème 4.2 : [13] Une algorithme de liste LPT est $\frac{4}{3}$-approximatif pour $P||C_{max}$.

$$\rho_{LPT} = \frac{4}{3} - \frac{1}{3m}$$

Preuve : Voir [28].

4.1.5.2 Algorithme de Hu (1961)

L'algorithme de Hu [21] résout exactement $P|intree, p_j = 1|C_{max}$, le problème avec des tâches de durées égales et liées par un graphe de précédence sous forme d'un arbre entrant. On appelle *niveau* d'une tâche dans un graphe, le nombre maximum de tâches dans un chemin

reliant cette tâche et une tâche sans successeur. L'algorithme de Hu est un algorithme de liste où le niveau d'une tâche détermine sa priorité.

Algorithme 4.1 : [21] $P \mid intree, p_j = 1 \mid C_{max}$

Calculer le niveau de chacune des tâches par la formule récursive :

$$N(T_i) = \begin{cases} 0 \quad \text{si } T_i \text{ est sans successeurs} \\ \max\limits_{T_j \text{ succeseur de } T_i} N(T_j) + 1 \end{cases} \quad ;$$

Sortir la liste des tâches dans l'ordre décroissant des niveaux ;

Affecter les tâches dans l'ordre de la liste ;

Il est aussi appelé algorithme de niveau ou algorithme de chemin critique. Il est en O(n).

Remarques :

- En cas de graphe de précédence formant un arbre sortant (*out tree*), on inverse les arcs et l'ordonnancement obtenu est lu de droite à gauche.
- Si le graphe forme une collection d'arbres entrant ou bien d'arbres sortant (*tree*), on ajoute un sommet de durée nulle, respectivement successeur de tous les sommets finaux ou prédécesseur de tous les sommets racines.

Si le graphe de précédence est arbitraire ($P \mid prec, pj{=}1 \mid C_{max}$), l'algorithme de Hu donne le rapport du plus mauvais cas :

$$\rho_{Hu} = \begin{cases} \frac{4}{3}, \quad \text{pour } m = 2 \\ 2 - \frac{1}{m-1}, \quad \text{pour } m \geq 3 \end{cases} \qquad [45]$$

4.1.5.3 Algorithme de CAUFFMAN et GRAHAM (1972)

L'algorithme de CAUFFMAN et GRAHAM [20] résout exactement $P_2 \mid prec, p_j{=}1 \mid C_{max}$, le problème à deux machines identiques, à durées d'exécutions égales et à contraintes de précédence arbitraires. C'est aussi un algorithme de liste. Une procédure d'*étiquetage* des tâches prend en compte à la fois le niveau et le nombre de successeurs

immédiats d'une tâche. La liste est construite dans l'*ordre lexicographique* décroissant des étiquettes.

Algorithme 4.2 : [20] $P_2 | prec, p_j = 1 | C_{max}$

Calculer une étiquette à chaque tâche T_i par la formule récursive :

$$E(T_i) = \begin{cases} 0 & \text{si } T_i \text{ est sans successeurs} \\ \max_{T_j \text{ succeseur de } T_i} E(T_j) + 1 & \end{cases} ;$$

Sortir la liste des tâche dans l'ordre lexicographique des étiquettes ;

affecter les tâches dans l'ordre de la liste ;

Cet algorithme est en $O(n^2)$. Si le nombre de machines est arbitraire ($P_m | prec, p_j = 1 | C_{max}$), l'algorithme de CAUFFMAN et GRAHAM donne le rapport du plus mauvais cas :

$$\rho_{CG} = 2 - \frac{2}{m} \qquad [46]$$

4.1.5.4 Recherche d'une liste optimale par une méthode SEP

Le problème $P | prec | C_{max}$ admet une liste optimale qu'on peut déterminer en utilisant une méthode de recherche SEP sur l'espace des listes réalisables. La complexité d'un tel algorithme est liée au nombre de feuilles maximum de l'arborescence de recherche, il est en o(n!). Il diminue considérablement en présence de contraintes de précédence et en utilisant une évaluation efficace au moyen d'algorithmes donnant de bonnes bornes inférieures [42].

Une telle approche est utilisé par SCHUTTEN et LENSINK [47] pour résoudre $P | r_j | L_{max}$ et par STERN [48] pour résoudre $R | | C_{max}$, le problème avec des contraintes de placement, une tâche ne peut s'exécuter que sur certaines machines.

4.2 Les algorithmes génétiques

Un algorithme génétique noté AG reproduit l'évolution naturelle d'organismes vivants, génération après génération, en respectant les phénomènes d'hérédité et la loi de

survie énoncés par Darwin. Dans une population, ce sont les individus les mieux adaptés au milieu qui survivront et pourront donner une descendance.

Dés les années cinquante, plusieurs biologistes ont simulé des structures biologiques sur ordinateur. Dans la décennie soixante, John Holland [49] détermina une analogie entre un individu dans une population d'individus et une solution d'un POC dans un ensemble de solutions. L'ouvrage de Goldberg (1989) [50] introduit l'application des AGs sur des problèmes concrets.

Un individu est caractérisé par une structure de données qui représente une solution. La force d'un individu, encore appelée *fitness* exprime la qualité de la solution correspondante. Les opérateurs génétiques de croisement et de mutation agissent sur les structures de données associées aux individus. Ils permettent de parcourir l'espace de solutions du problème. La population correspond à l'ensemble courant de solutions. Le renouvellement de cette dernière, autrement dit la création d'une nouvelle génération, est obtenu par itération de l'AG, créant de nouveaux individus et détruisant d'autres. L'exécution d'un tel algorithme doit conduire, à partir d'une population initiale, après de nombreuses générations, à une population où les individus sont tous forts (un ensemble de 'bonnes' solutions).

4.2.1 Un algorithme génétique

Pour mettre en œuvre un AG, il est nécessaire de disposer de :

- Un codage approprié des solutions,
- Un moyen pour obtenir la population initiale,
- Une fonction d'évaluation mesurant la fitness d'un individu,
- Un mode de sélection des individus à reproduire,
- Des opérateurs génétiques adaptés au problème,
- Des valeurs pour les paramètres de l'algorithme.

4.2.1.1 Codage

Le codage s'inspire des chromosomes rencontrés dans la nature, qui est une chaîne de gênes. Le codage classique sous forme de chaîne de bits, appelé *codage binaire*, est souvent utilisé. Les gênes sont représentés par des 0 et des 1.

On peut représenter :

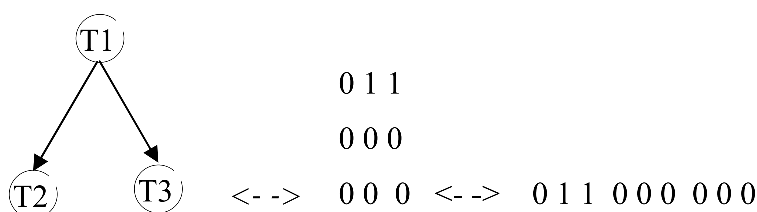
- Un entier appartenant à [0,31] par son code dans le système binaire.

3 <--> 00011 ; 6 <--> 00110 ; 9 <--> 01001

- Un vecteur de trois variables entières dans [0, 31] par la concaténation de leurs codes binaires.

(3, 6, 9) <- -> 00011 00110 01001

- Un graphe par la concaténation des lignes de sa matrice d'incidence.



4.2.1.2 Population initiale

Dans un algorithme de recherche par voisinage, le choix des solutions initiales de bonne qualité accélère la convergence vers l'optimum.

On peut obtenir la population initiale par :

- Génération aléatoire, permettant d'avoir des solutions de diverses caractéristiques.
- Des heuristiques, donnant des individus 'forts' mais qui risquent de faire converger l'algorithme vers des optimums locaux [51][52].
- Un mélange de solutions heuristiques et aléatoires, permettant d'avoir à la fois des individus 'forts' et une diversité.

- Duplication et évolution d'une seule solution, lorsque le problème est fortement contraint et l'obtention de plusieurs solutions est difficile. On prend une solution et on la modifie par mutation et croisement.

4.2.1.3 Evaluation

L'évaluation consiste à mesurer la 'force' de chaque individu (fitness) de la population. Si le problème est de maximisation, la fonction objectif peut être utilisée comme mesure. En cas de minimisation, on utilise une fonction qui varie inversement avec la fonction objectif. Soit le problème,

$$\min_{x} C(x)$$

On peut mesurer la fitness d'un individu correspondant à une solution x_i par :

- $f(x_i) = UB - C(x_i)$ telle que UB est une borne supérieure de $C(x)$.
- $f(x_i) = \frac{1}{C(x_i)}$.
- $f(x_i) = \frac{mx - C(x_i)}{mx - mn}$.

  tels que mx et mn sont respectivement le mauvais et le meilleur des coût de la population courante.

Les deux dernières mesures sont étudiées au chapitre cinq.

Aucune condition particulière n'est requise pour la fonction objectif, il suffit qu'elle retourne des valeurs numériques comparables. La performance de l'AG peut être sensible au choix de la mesure de la fitness comme on le verra au chapitre cinq.

4.2.1.4 Sélection

La sélection aussi bien celle des individus de 'bonne qualité' que celle des individus de 'mauvaise qualité', comporte généralement un aspect aléatoire. Chaque individu x_i de la population parmi laquelle se fait la sélection se voit attribuer une probabilité p_i d'être choisi d'autant plus grande que son évaluation est haute (basse dans le cas de 'mauvais individus'). On tire un nombre r_i au hasard (uniformément sur [0, 1]). La probabilité que x_k soit choisi est aussi bien égale à p_k. L'individu k est choisi si :

$$\sum_{j=1}^{k-1} p_j < r_i \le \sum_{j=1}^{k} p_j$$

Cette procédure est appelée *roue de la fortune*. La procédure est itérée jusqu'à ce que l'on choisit un nombre fixé d'individus.

La diversité de la population doit être entretenue au cours des générations afin de parcourir le plus largement possible l'espace des solutions. C'est le rôle des opérateurs de croisement et de mutation. Les premiers sont des opérateurs sexués, deux parents sont nécessaires pour générer deux enfants, les seconds sont des opérateurs asexués qui n'ont besoin que d'un parent pour générer un enfant.

4.2.1.5 Croisement

Bien que de nombreuses variantes de ces opérateurs existent, nous présentons ici celles qui sont, le plus couramment, utilisées.

Le *croisement à un point* : C'est le plus simple. Si n est la longueur du chromosome, on choisit aléatoirement une position s entre 1 et n-1. On coupe les deux parents à cette position et on les recolle en les croisant.

parent 1 = B_1 B_2 B_3 ....B_s B_{s+1} ... B_n
parent 2 = C_1 C_2 C_3 ... C_s C_{s+1} ... C_n

fils 1 = B_1 B_2 B_3 ....B_s C_{s+1} ... C_n
fils 2 = C_1 C_2 C_3 ... C_s B_{s+1} ... B_n

Le *croisement à k points* : C'est une généralisation du croisement à un point. On choisit aléatoirement k positions entre 1 et n, créant ainsi k+1 morceaux. Le 1er fils (respectivement le 2ème fils) est la concaténation des morceaux d'ordre pair (impair) du 1er parent et des morceaux d'ordre impair (pair) du 2ème parent.

Soient k = 2 et i, j les points de croisement générés,

parent 1 = B_1 B_2 B_3 ....B_i B_{i+1} ... B_j B_{j+1}... B_n
parent 2 = C_1 C_2 C_3 ... C_i C_{i+1} ... C_j C_{j+1}... C_n

fils 1 $= B_1 B_2 B_3 \ldots B_i C_{i+1} \ldots C_j B_{j+1} \ldots B_n$

fils 2 $= C_1 C_2 C_3 \ldots C_i B_{i+1} \ldots B_j C_{j+1} \ldots C_n$

Le *croisement uniforme* : Chaque gêne de chaque fils est celui de l'un des deux parents avec la même probabilité.

parent 1 $= B_1 B_2 B_3 B_4 B_5$

parent 2 $= C_1 C_2 C_3 C_4 C_5$

On génère deux fois cinq valeurs aléatoires dans {0,1}, soient (0,1,1,0,1) et (1,0,0,1,1). Le 0 indique qu'on prend le gêne du 1er parent et le 1 indique qu'on prend celui du 2ème parent. On obtient :

fils 1 $= B_1 C_2 C_3 B_4 C_5$

fils 2 $= C_1 B_2 B_3 C_4 C_5$

4.2.1.6 Mutation

Une mutation est une perturbation introduite pour modifier une solution individuellement, par exemple, en cas de codage binaire, la transformation d'un 0 en un 1 ou inversement. En général, on décide de muter une solution avec une probabilité assez faible. Le but de la mutation est d'introduire un élément de diversification et d'innovation.

La mutation peut créer des chromosomes difficiles à avoir avec le croisement. Elle correspond à une petite perturbation de la solution. On cite les trois variantes suivantes :

Modifier la valeur d'un gêne : Le gêne de position 3 est modifié de B_3 en B_5.

parent $= B_1 B_2 B_3 B_4 B_5 \rightarrow$ fils$= B_1 B_2 B_5 B_4 B_5$

Permuter deux gênes consécutifs : Les gênes de positions 3 et 4 sont échangés.

parent $= B_1 B_2 B_3 B_4 B_5 \rightarrow$ fils $= B_1 B_2 B_4 B_3 B_5$

Permuter deux gênes quelconques : Les gênes de positions 1 et 4 sont échangés.

parent $= B_1 B_2 B_3 B_4 B_5 \rightarrow$ fils $= B_4 B_2 B_3 B_1 B_5$

Inverser l'ordre d'une sous chaîne du chromosome : La sous chaîne B_2 B_3 B_4 B_5 est inversée.

parent = $B_1\ B_2\ B_3\ B_4\ B_5$ → fils = $B_1\ B_5\ B_4\ B_3\ B_2$

4.2.1.7 Condition d'arrêt

Comme toute procédure itérative, un AG s'arrête si une certaine condition d'arrêt est vérifiée. Plusieurs conditions peuvent être définies, citons par exemple :

Arrêt après un nombre de générations fixé à priori : C'est la plus utilisée lorsqu'un impératif de temps de calcul est imposé ou lorsque la performance de l'algorithme est à tester. Un nombre de générations est produit signifie qu'un pourcentage de l'espace des solutions est exploré. Dans nos expérimentations ultérieures, c'est cette stratégie qui est adoptée.

Arrêt lorsque la population cesse d'évoluer en solutions ou n'évolue plus rapidement : Cela s'interprète par l'atteinte d'un minimum local ou la population est homogène. On peut penser que la population se situe à proximité de solutions optimums. Ce critère permet d'arrêter l'algorithme quand la plupart des gains ont été obtenus. L'inconvénient de cette stratégie est qu'il n'est pas possible de prévoir la durée d'exécution.

Une combinaison des deux conditions : On arrête lorsqu'on évolue plus dans la qualité de la solution ou bien lorsqu'on atteint un nombre maximum de générations.

4.2.2 Convergence des AGs

Les AGs ont montré leur efficacité pratique bien avant que les résultats de convergence théorique ne soient établis. Puisque ces algorithmes sont destinés à fournir de bonnes solutions, non nécessairement optimales, les résultats théoriques sont des preuves de convergence asymptotique vers un optimum. Ces résultats ne sont apparus que tardivement, une démonstration complète et rigoureuse de convergence stochastique est établie par CERF [53] en 1993.

Nous disposons de trois approches théoriques différentes permettant de mieux comprendre le fonctionnement des AGs :

La théorie des schémas : Elle s'applique sur des chaînes de bits et utilise l'opérateur de croisement à un point et la roue de la fortune comme procédure de sélection. HOLLAND (1975) [102] montra que les individus avec une force supérieure à la moyenne ont tendance à apparaître plus fréquemment dans une nouvelle génération.

L'approche du recuit simulé : Elle se base sur les résultats de convergence stochastique de l'algorithme du recuit simulé développés par LAARHOVEN et AARTS [62]. Sous certaines hypothèses, une convergence asymptotique est obtenue par l'opérateur de mutation.

L'approche du processus Markovien : C'est la plus récente, elle est proposée par CERF [53]. Il utilise une modélisation de l'AG en processus markovien. Les résultats de convergence asymptotique sont obtenus grâce à la théorie de FREIDLIN et WENTZELL [54].

Cette dernière approche est la plus satisfaisante tant sur le plan mathématique que sur celui de la modélisation. Les différents opérateurs étant présentés comme 'perturbant' d'un processus Markovien représentant la population à chaque étape. Ici, comme dans la deuxième approche, on démontre l'importance de l'opérateur de mutation, le croisement pouvant être omis. Ce résultat est intuitif puisque seul cet opérateur permet réellement l'exploration aléatoire de l'espace de recherche.

Les étapes d'un AG se résume selon le schéma suivant :

Etape 1 : générer une population initiale d'individus

Etape 2 : répéter jusqu'à un critère d'arrêt

- Mesurer la fitness des individus.
- Sélectionner les individus pour la population suivante.
- Diversifier la population (croisement et mutation).

Il existe, suivant ce schéma, de nombreuses variantes d'algorithmes. La diversification de la population peut se faire en deux étapes : le croisement génère de nouveaux individus à partir des anciens et la mutation modifie un individu indépendamment des autres individus de la population. C'est un AG classique qui s'écrit en pseudo code, comme suit :

Algorithme 4.3 : AG.

Etape 0 : Définir un codage du problème ;

Etape 1 : t = 0, créer une population initiale $P(0) = x_1,x_2,x_3,...,x_p$;

Etape 2 : Calculer la force $f(x_i)$ de chaque individu x_i, i : 1..p ;

Etape 3 : Sélectionner ‘p’ individus de P(t) ; les ranger dans S(t) ;

Etape 4 : Pour chaque paire d’individus de S(t):

- Appliquer l’opérateur de croisement avec une probabilité p_c ;
- Appliquer l’opérateur de mutation avec une probabilité p_m ;

Etape 5 : t = t+1, P(t) = S(t), aller à l’étape 2 ;

4.2.3 Les particularités des AGs

Les AGs ont retenu l’attention de nombreux chercheurs et praticiens de l’optimisation en raison de leurs particularités attrayantes ci-dessous mentionnées.

- Ils travaillent sur un codage des paramètres du problème et non sur les paramètres eux-mêmes.
- Ils effectuent la recherche d’un optimum à partir d’une population et non d’un point unique.
- Ils utilisent :
  - Les informations apportées par la fonction objectif elle-même et non par ses fonctions dérivées ou son gradient. La fonction objectif peut être le résultat d’une simulation.
  - Des règles de transition probabilistes qui leur permettent de sortir d’un optimum local .
- Le peu d’hypothèses requises leur permet de traiter des problèmes très complexes.
- Ils sont attrayants pour l’utilisateur car ils ne réclament pas en général des connaissances très pointues en optimisation.
- Leur parallélisme intrinsèque augmente leurs possibilités pratiques.

Dans la suite de ce chapitre, on détaille les principales notions énoncées plus haut et dans le cinquième chapitre on conçoit et étudie un AG pour résoudre le problème qui nous intéresse, à savoir $P|prec|C_{max}$.

4.2.4 Des extensions pour $P|prec|C_{max}$

La structure classique d'un AG et le codage binaire sont souvent mal adaptés aux problèmes d'optimisation complexes. Des extensions ont été proposées utilisant d'autres types de codages ou d'autres opérateurs génétiques ou même des structures nouvelles de l'algorithme.

On appelle *algorithme à stratégie d'évolution* (ASE) [55] une extension des algorithmes génétiques où l'on modifie un AG classique par :

- Utilisation de codages et d'opérateurs génétiques non classiques,
- Modification de la structure de l'algorithme classique.

Les ASE sont mieux adaptés à la description des problèmes d'ordonnancement où l'on manipule des ordres (permutations) ou des affectations (matrices), etc.

4.2.4.1 Codage des ordonnancements

Dans un problème d'ordonnancement, une solution est une affectation des tâches sur les machines en respectant les contraintes du problème. On distingue trois types de codages [56] :

La représentation directe : On découpe un chromosome en sous chromosomes, représentant chacun un ordre des tâches sur une machine. Un gêne est un couple représentant une opération et sa date de début d'exécution. Les opérateurs génétiques doivent être définis de telle façon que les contraintes soient respectées et que la structure des chromosomes reste correcte.

La représentation indirecte : C'est un codage en permutations. On utilise les algorithmes de listes. Les sous chromosomes ne représentent plus des ordonnancements mais des listes de priorités sur une machine. On détermine un ordonnancement selon une règle d'affectation. L'AG ne cherche que dans l'espace des listes. Pour certains problèmes n'admettant pas un ensemble de listes dominant, les solutions obtenues sont localement optimales [56].

La représentation spécifique au problème : Cette représentation reprend les deux principes précédents mais elle est adaptée à un problème spécifique. Elle prend en compte des informations telles que l'ensemble des machines pouvant effectuer une opération ou des

contraintes autres que celles de précédence. LAWTON (1992) [57] utilise des chromosomes qui sont des listes de n-uplets. Chacun représente une tâche et ses contraintes de précédence.

Dans ces différents types, on utilise le *codage en permutations* où un chromosome est une permutation. Pour ce codage ‘non classique’, ils existent des opérateurs génétiques spéciaux car les opérateurs ‘classiques’ ne sont plus valides, par exemple, soient deux individus choisis pour le croisement:

Parent 1 = A B C D E F G H

Parent 2 = B E F H A D G C

un croisant à deux points ‘2’ et ‘5’, donne:

fils 1 = A B | F H A | F G H

fils 2 = B E | C D E | D G C

Les fils ne sont pas des permutations, le croisement classique à deux points n’est pas valide pour le codage en permutations.

Le croisement OX : L’idée de ce croisement est de ‘préparer’ les deux parents avant l’échange des séquences.

Dans l’exemple précédent, la zone d’échange de parent 1 est préparée à accueillir la séquence des villes F, H, A du parent 2. Pour ce faire, on remplace chacune des tâches F, H et A dans le parent 1 par une place vide symbolisée par une *, soit * B|CDE|*G* et en commençant à la droite de la zone d’échange, on tasse les tâches restantes de la permutation dans l’ordre du parent 1 en oubliant les *, ce qui donne : DE|***|GBC. Les * se retrouvent dès lors dans la zone d’échange, alors que l’ordre de parcours des autres tâches n’a pas été changé. On procède de même pour le parent 2 : B*|FHA|*G* devient HA|***|GBF.

On procède alors à l’échange des séquences, ce qui donne deux permutations enfants ‘fils 1’ et ‘fils 2’:

fils 1 = DE|FHA|GBC

fils2 = HA|CDE|GBF

4.2.4.2 Prise en compte des contraintes de précédence

Les opérateurs génétiques, de par leur fonctionnement, ne permettent pas la prise en compte de contraintes de précédence entre les tâches. On distingue trois types de solutions :

Une fonction d'évaluation à pénalité : Si les opérateurs génétiques donnent des solutions non réalisables, la fonction d'évaluation leur affecte des pénalités de sélection. Les probabilités de sélection sont ainsi fortement diminuées et par conséquent ces solutions sont éliminées [58].

Des algorithmes de réparation : Comme leur nom l'indique, ils rendent admissible une solution non admissible générée par l'opérateur génétique.

Des opérateurs génétiques adaptés au problème : Plutôt que de créer des solutions non réalisables puis les réparer, on utilise des opérateurs génétiques qui génèrent directement des solutions réalisables. On inclut dans ces opérateurs la connaissance du problème.

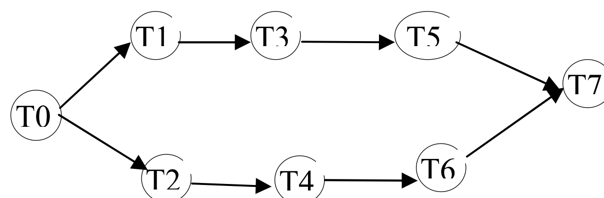
Pour $P|prec|C_{max}$, on présente trois opérateurs de croisement adaptés au codage indirecte (en permutations) et qui prennent en compte les contraintes de précédence [59].

On note les individus parents $P^1 = (P_1^1, P_2^1, ..., P_n^1)$, $P^2 = (P_1^2, P_2^2, ..., P_n^2)$ et leurs fils $F^1 = (F_1^1, F_2^1, ..., F_n^1)$, $F^2 = (F_1^2, F_2^2, ..., F_n^2)$. Un individu est dit *réalisable* s'il correspond à une liste réalisable.

i) Opérateur de croisement à un point : On génère un entier s avec $1 \leq s \prec n$. Pour F^1, les positions i = 1, …, s sont prises de $P^1, F_i^1 := P_i^1$. Les positions i = s+1, …, n sont prises de P^2 sans considérer les tâches déjà prises, $F_i^1 := P_k^2$ où k est le plus petit indice tel que $P_k^2 \notin \{ F_1^1, F_2^1, F_3^1, ..., F_{i-1}^1 \}$.

On construit F^2 d'une manière analogue, $F_i^2 := P_i^2$ pour i = 1, …, s et pour i = s+1, …, n, $F_i^2 := P_k^1$ où k est le plus petit indice tel que $P_k^1 \notin \{F_1^2, F_2^2, F_3^2, ..., F_{i-1}^2 \}$.

Exemple 4. 2 :



Soient $P^1 = T_1\ T_3\ T_2\ T_5\ T_4\ T_6$ et $P^2 = T_2\ T_4\ T_6\ T_1\ T_3\ T_5$, le croisement à un point s = 3 donne :

$$F^1 = T_1\ T_3\ T_2\ T_4\ T_6\ T_5 \text{ et } F^2 = T_2\ T_4\ T_6\ T_1\ T_3\ T_5$$

Théorème 4.3 : [59] Le croisement à un point défini, donne deux fils réalisables à partir de deux parents réalisables.

Preuve : Soient deux parents réalisables et q le point de croisement. De la définition, une tâche n'apparaît qu'une et une seule fois dans un fils. Il reste à vérifier qu'il est réalisable. Par contraposé, supposons que la permutation du premier fils n'est pas réalisable, ils existent deux tâches F_i et F_k avec $1 \le i < k \le n$ et la contrainte de précédence $F_k < F_i$. Trois cas se présentent :

- $i \le q$ et $k \le q$: F_i précède F_k dans la séquence du premier parent. Contradiction avec le fait que le premier parent soit réalisable.

- $i \succ q$ et $k \succ q$: Comme les positions relatives sont maintenues par l'opérateur de croisement, la tâche F_i précède F_k dans la séquence du deuxième parent. Contradiction avec le fait que le deuxième parent soit réalisable.

- $i \le q$ et $k > q$. F_i précède F_k dans la séquence du 1er parent. Contradiction avec le fait que le premier parent soit réalisable. □

Une preuve similaire est valable pour le deuxième fils.

ii) Opérateur de croisement à deux points : C'est une extension du croisement à un point. On génère deux entiers s_1 et s_2 tel que : $1 \le s_1 \prec s_2 \le n$. F^1 est construit en prenant la séquence des tâches de positions $i = 1,\ldots,s_1$ de P^1, $F_i^1 := P_i^1$. Les positions $i = s_1+1, \ldots, s_2$ sont prises de P^2, $F_i^1 := P_k^2$ où k est le plus petit indice tel que $P_k^2 \notin \{F_1^1, F_2^1, F_3^1, \ldots, F_{i-1}^1\}$. Les positions restantes $i = s_2+1, \ldots, n$ sont prises de P^1, $F_i^1 := P_k^1$ où k est le plus petit indice tel que $P_k^1 \notin \{F_1^1, F_2^1, F_3^1, \ldots, F_{i-1}^1\}$.

F^2 est construit comme suit : $F_i^2 := P_i^2$, pour i = 1, …, s_1. $F_i^2 := P_k^1$ où k est le plus petit indice tel que $P_k^1 \notin \{F_1^2, F_2^2, F_3^2, ..., F_{i-1}^2\}$ pour i = s_1+1, …., s_2. $F_i^2 := P_k^2$ où k est le plus petit indice tel que $P_k^2 \notin \{F_1^2, F_2^2, F_3^2, ..., F_{i-1}^2\}$ pour i = s_2+1, …, n.

Le croisement à deux points, $s_1 = 1$ et $s_2 = 3$, donne pour l'exemple précédent :

$$F^1 = T_1\ T_2\ T_4\ T_3\ T_5\ T_6 \text{ et } F^2 = T_2\ T_1\ T_3\ T_4\ T_6\ T_5$$

iii) Opérateur de croisement uniforme : F^1 est construit comme suit : on tire une séquence de nombres aléatoires $p_i \in \{0,1\}$, i = 1,…,n. On complète les positions i = 1, …, n de F^1 successivement. Si $p_i = 1$, on prend la tâche de F^1 avec l'indice le plus petit parmi les tâches non encore prises, $F_i^1 := P_k^1$ où k est le plus petit indice tel que $P_k^1 \notin \{F_1^1, F_2^1, F_3^1, ..., F_{i-1}^1\}$. Sinon $p_i = 0$, on prend de la même façon une tâche de P^2, $F_i^1 := P_{k'}^2$ où k' est le plus petit indice tel que $P_{k'}^2 \notin \{F_1^1, F_2^1, F_3^1, ..., F_{i-1}^1\}$.

D'une manière analogue, on construit F^2 en prenant la i^{ème} tâche de P^2 si $p_i = 1$, sinon celle de P^1.

Pour l'exemple précédent et avec la séquence de nombres aléatoires (0,1,1,0,1,1), on obtient :

$$F^1 = T_2\ T_1\ T_3\ T_4\ T_5\ T_6 \text{ et } F^2 = T_1\ T_2\ T_4\ T_3\ T_6\ T_5$$

Notez que le croisement uniforme généralise le croisement à deux points. En effet, en prenant $p_i = 1$ pour i ∈ {1, …, s_1, s_2, …, n} et $p_i = 0$ pour i ∈ {s_1+1, …, s_2}, on se ramène à la définition de F^1 dans le croisement à deux points.

En utilisant des arguments similaires que ceux utilisés pour le croisement à un point, on peut démontrer le théorème précédent pour les croisements à deux points et uniforme.

Comme résultat des définitions de ces opérateurs, les positions relatives dans les parents sont conservées et donc les deux parents contribuent dans la fitness de leurs fils.

4.2.2.3 Les anti-clones

Le croisement peut produire des individus identiques à des individus existant déjà dans la population, appelés clones. Ces derniers diminuent l'efficacité de l'algorithme en

augmentant le temps à produire des individus existant déjà et diminuant la variété de la population. La mutation peut échouer à diversifier la population surtout si la probabilité de mutation est faible. Pour faire face à ce problème, on peut mettre au point plusieurs mécanismes. Par exemple, si le pourcentage des clones dans la population courante dépasse un certain seuil fixé, on remplace les individus clones par des individus générés aléatoirement.

CHAPITRE 5

IMPLEMENTATION, EXPERIMENTATIONS ET EVALUATION DE SIX LISTES DE PRIORITES ET D' UN AG

Une façon de mesurer la performance d'une méthode approchée consiste à faire une analyse expérimentale. Ce type d'analyse devient incontournable en l'absence des deux types d'analyses théoriques : du plus mauvais cas et en moyenne. Rappelons qu'une instance est une application numérique d'un problème. On teste la méthode sur un grand nombre d'exemples qui représentent au mieux les instances du problème.

Une variante d'un algorithme peut donner un 'bon' résultat avec une première instance et un 'mauvais' résultat avec une seconde. Une question se pose: quelle variante est bonne pour quelle instance ?

On définit un ensemble fini de variantes qu'on détermine sur la base de connaissances préalables et durant les tests. On étudie les performances de ces variantes avec des ensembles de tests ayant des caractéristiques en commun.

Dans ce chapitre, on fait une analyse expérimentale de six listes de priorités et de quelques variantes d'un AG, utilisés pour résoudre le problème $P|prec|C_{max}$.

5.1 Un générateur de jeux d'essai

Nous avons réalisé un générateur de jeux d'essais du problème $P|prec|C_{max}$. Il comporte des paramètres qu'on peut régler pour avoir des jeux d'essais à caractéristiques voulues.

On génère des problèmes à durées d'exécution unitaires ou distribuées uniformément entre 0 et une constante b, entières ou réelles.

Sachant que les sommets d'un graphe G(V, E) sans circuits peuvent être ordonnés comme suit:

$$\text{Pour } i \in V : \text{ord}(i) = \begin{cases} 0 & \text{si i n'a pas de prédécesseurs} \\ \max\limits_{\text{j prédécesseur de i}} \{\text{ord}(j) + 1\} & \text{sinon} \end{cases}$$

Un graphe de précédence G (V, U) est alors généré en deux étapes :

Etape 1 : On génère aléatoirement une distribution d'ordre pour les n sommets, selon l'algorithme suivant :

Algorithme 5.1 : Donner un ordre à chaque sommet.

ord entier (ordre); ord = 0 ; //initialisation

Répéter

Pour chaque sommet i non encore muni d'un ordre faire

Avec une probabilité w = w (ord), affecter l'ordre ord à i ;

S'il y a des sommets qui ont été affectés de l'ordre ord, alors ord = ord + 1 ;

Jusqu'à l'affectation d'un ordre à chaque sommet ;

Etape 2 : Pour chaque sommet possédant un ordre, on génère les arcs entre les sommets. Chaque sommet précède un sommet d'ordre supérieur avec une probabilité 'd'. Voici l'algorithme correspondant :

Algorithme 5.2 : Générer les arcs entre les sommets ordonnés.

Pour tout sommet i, faire

Pour tout sommet j, faire

Si ord(i) > ord(j) alors

Tirer au hasard un nombre $d^* \in [0, 1]$,
Si $d^* \leq d$, $ij \in U$ **;**

Ayant une distribution d'ordre sur les sommets, on peut mesurer la densité du graphe par le rapport entre le nombre d'arcs générés et le nombre d'arcs possibles. La moyenne de la densité des graphes générés est égale à d. Ce résultat est vérifié expérimentalement et peut être démontré.

On a donc, deux nouveaux paramètres de contrôle du générateur :

- **La probabilité d :** Elle mesure la densité du graphe de précédence en moyenne.

- **La fonction w :** Elle donne la forme du graphe de précédence en moyenne. Voir figure 5.1.

    - w = cte ∈ [0, 1] : En allant dans le sens des ordres croissants, le graphe se rétrécit.

    - w est croissante en ord. le graphe a tendance à se redresser pour avoir la même hauteur. Si cette croissance est suffisamment rapide, le graphe est en moyenne rectangulaire. Si la croissance est encore plus rapide, il s'élargit.

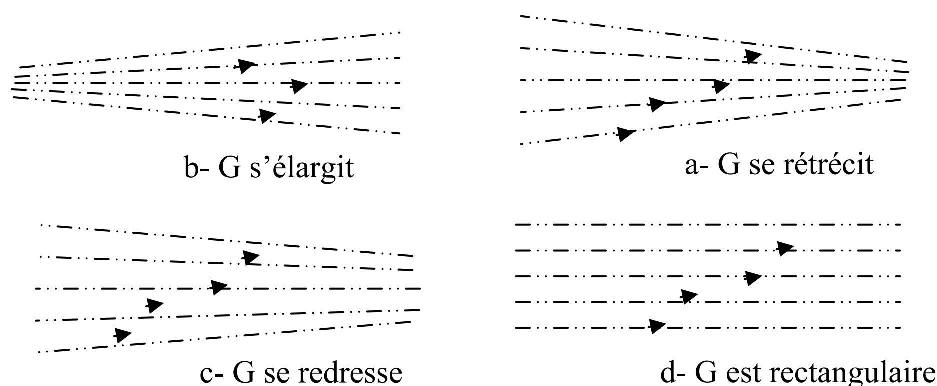


Figure 5.1 : La fonction w (ord)

a- w = cte
b- w ↑
c- w ↑↑
d- w ↑↑↑

5.2 Critères de performance

Pour les méthodes approchées, deux critères sont à considérer : le temps pour calculer une solution approchée et l'éloignement de la valeur de cette solution par rapport à la valeur optimale. L'idéal serait de pouvoir comparer les résultats avec la longueur d'un ordonnancement optimal, ce qui est bien sûr coûteux en temps de calcul.

Selon le deuxième critère, on peut étudier les performances des méthodes de différentes manières :

- Classement des différentes méthodes. On obtient le nombre de jeux d'essais pour lesquels la méthode donne le meilleur résultat parmi les autres méthodes.

- La *performance relative* d'une méthode pour un jeu d'essai donné est le rapport entre le makespan calculé par cette méthode et celui calculé par une méthode arbitraire. La *performance relative moyenne* est la moyenne des performances relatives sur un ensemble de tests.

- Comparaison entre les valeurs objectifs des solutions données par les méthodes et une borne inférieure de la valeur optimale. C'est cette approche qu'on utilise. Une mesure de performance expérimentale, appelée *rapport d'approximation expérimental*, est défini par :

$$\rho(I) = \frac{C_{max}^{A}(I)}{OPT(I)} \leq \frac{C_{max}^{A}(I)}{LB(I)} = \rho_{exp}(I),$$

Où : I, le jeu d'essai. $C_{max}{}^{A}(I)$, la valeur de la solution obtenue par la méthode A. OPT(I), la valeur optimale. LB(I), une borne inférieure de la valeur optimale.

On a utilisé $LB = \max(LB_1, LB_2)$, avec :

$$LB_1 = \max_{\text{i sans prédécesseur}} PLC(i)$$

$$LB_2 = \frac{\sum_i p_i}{m}$$

Pour un échantillon de 100 jeux d'essai, on mesure le rapport d'approximation expérimental en moyenne: $\rho_{moy} = \frac{\sum_I \rho_{exp}(I)}{100}$.

et au maximum : $\rho_{max} = \max_I \rho_{exp}(I)$.

Des détails sur les techniques d'évaluation expérimentale des heuristiques, incluant le plan expérimental, les sources de tests, les instances, les mesures de performance, l'analyse et la présentation des résultats sont dans [63].

5.3 Les expérimentations

Un nombre important d'expérimentations a été réalisé, sur six listes et sur différentes variantes de l'AG. Des paramètres sont à fixer pour le générateur de jeux d'essai.

Si le nombre de machines est égal à 1, m = 1, une solution triviale est obtenue par séquencement arbitraire des tâches en respectant les contraintes de précédence et sans temps d'oisiveté. Si m = 2, on a des instances faciles à résoudre de type $P_2 | prec, p_j = 1 | C_{max}$. Il est résolu exactement par l'algorithme de CAUFFMAN et GRAHAM (1972) [20]. Le nombre de machines ne doit pas être trop grand, sinon le retardement de tâches prioritaires n'apparaît pas et les listes ont tendance à donner des ordonnancements de même longueur.

Le nombre de tâches doit être suffisamment grand. On a choisit n = 100 et m = 3.

On a utilisé la fonction w définie par :

$$w(ord) = \frac{ord + 1}{n + 1}$$

Elle donne des graphes presque rectangulaires en moyenne.

5.4 Résultats des listes

Une méthode de liste se caractérise par la règle de priorité adoptée pour construire la liste et la règle d'affectation. Le but de nos expérimentations a été de tester, d'évaluer et de comparer six règles de priorités. En plus des quatre listes présentées au quatrième chapitre PLC, SUCC, SOMPLC, Max{,} qu'on rappelle leurs définitions ci-dessous, deux autres règles sont introduites, RAND et SPT.

PLC :

En fonction du plus long chemin $PLC(i) = \begin{cases} p_i & \text{si i n'a pas de successeurs} \\ \sum_{\text{j successuer de i}} PLC(j) + p_i & \end{cases}$

Ou règle du chemin critique.

SPT :

En fonction de la durée d'exécution P_i.

SUCC :

En fonction du nombre de successeurs immédiats .

RAND :

Un ordre aléatoire .

SOMPLC :

En fonction d'une variante de PLC, $VPLC(i) = \begin{cases} p_i & \text{si i n'a pas de successeurs} \\ \sum_{\text{j successuer de i}} \{ VPLC(j) + p_i \} \end{cases}$

Max{,} :

En fonction de l'expression : $\max \left\{ PLC(i), \frac{\sum_{\text{j descendant de i}} p_j}{m} + p_i \right\}$

La règle d'affectation utilisée est celle sans délai.

Respectivement, les figures 5.2 et 5.3 représentent le comportement des six listes en fonction de la densité moyenne des graphes de précédence. Pour des instances à durées d'exécution unitaires et uniformément distribuées sur]0, 1]. Pour chacune des listes, on donne, en pourcentage, le nombre de fois où elle est la meilleure.

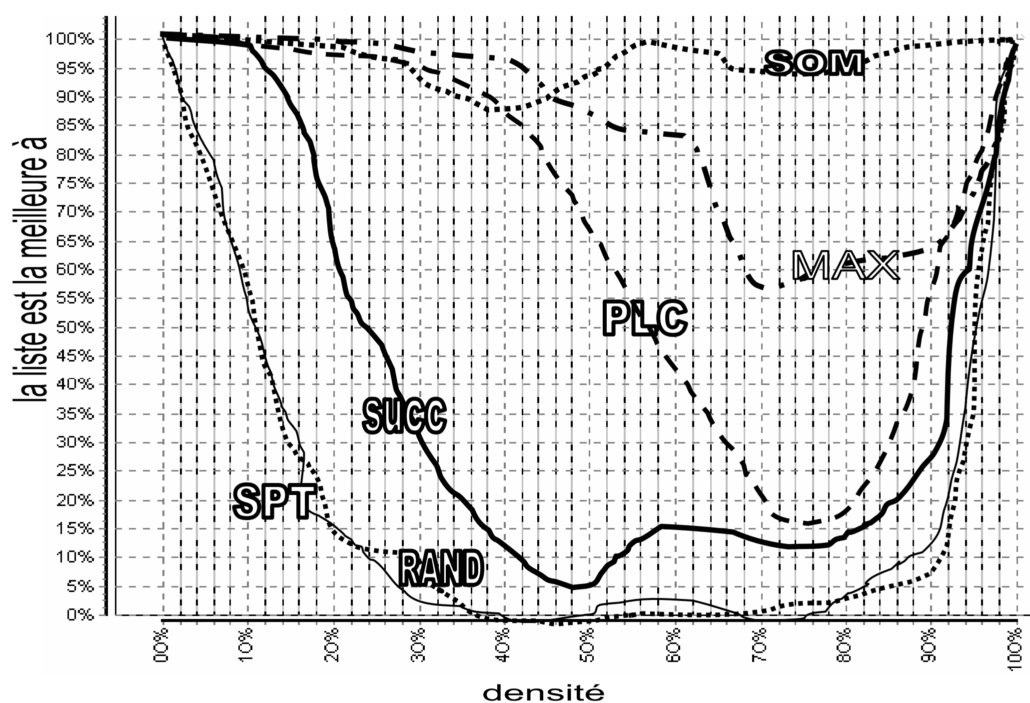


Figure 5.2 : Performance des listes, durées d'exécution unitaires.

Les listes PLC, SOM et Max sont les meilleures. Pour des graphes de forte densité, SOM est meilleure que PLC et Max. Pour des graphes de faible densité, les trois listes sont proches les unes des autres.

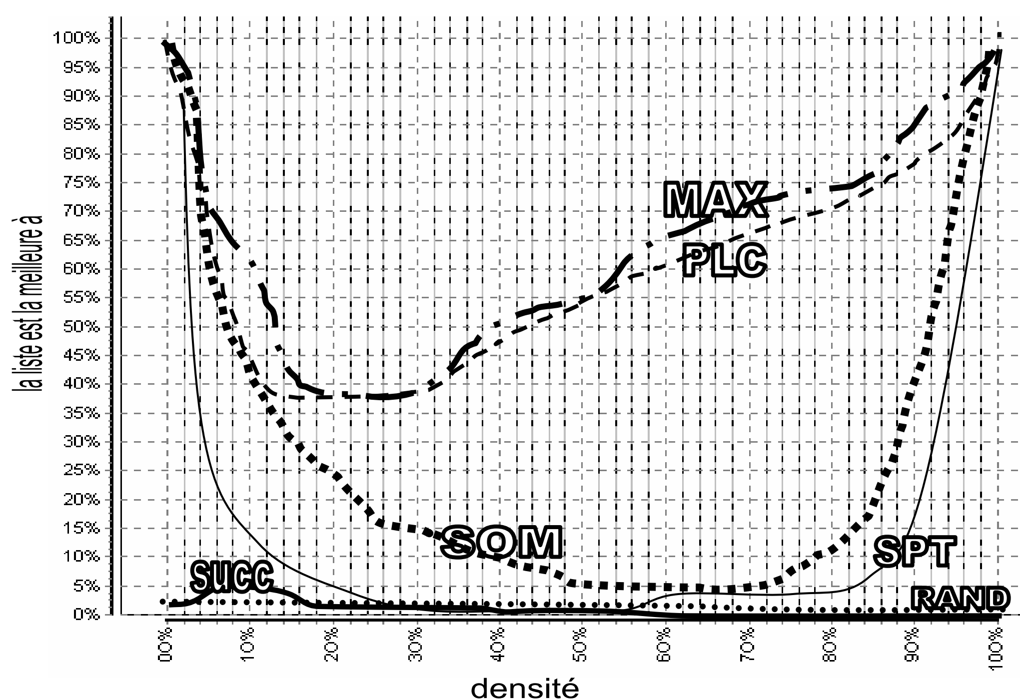


Figure 5.3 : Performance des listes, durées d'exécution uniformément

Pour des durées d'exécution distribuées uniformément, les listes PLC, SOM et Max sont encore, les meilleures. Pour des graphes de moyenne et forte densité, PLC et Max sont meilleures que SOM, la liste Max est légèrement meilleure que PLC.

Les trois listes PLC, SOM et Max sont basées sur la notion du chemin critique. PLC favorise les tâches à la tête d'un chemin critique car ces tâches risquent d'augmenter la longueur d'ordonnancement si elles sont retardées.

Illustrons le principe des listes Max et SOM sur l'exemple de contraintes de précédence de la figure 5.4 avec des durées unitaires.

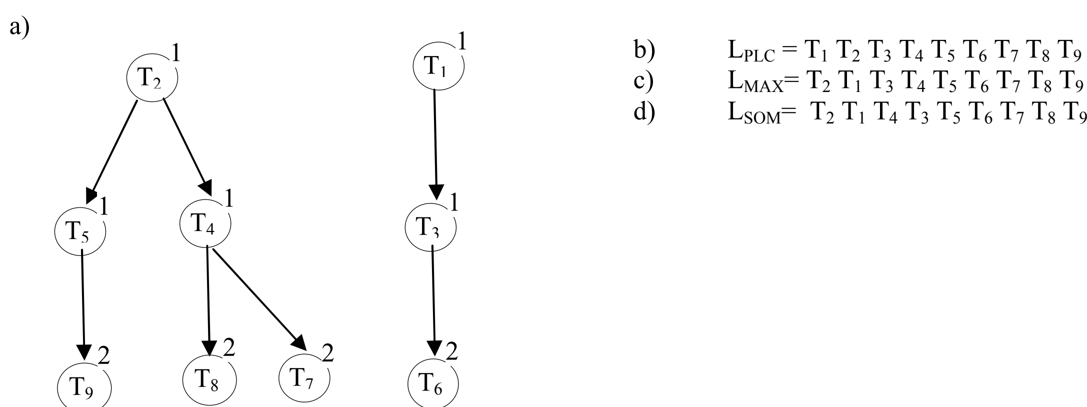


figure 5.4 : Exemple de graphe de précédence
a- Le graphe de précédence
b,c,d- Respectivement les listes PLC, Max et SOM

PLC donne une priorité identique pour les tâches T_1 et T_2 alors que T_2 est à la tête de plusieurs tâches pouvant s'exécuter en parallèles. Les listes Max et SOM cherchent un compromis entre la priorité des tâches d'un chemin critique et la priorité des tâches permettant l'exploitation du parallélisme. Elles favorisent la tâche T_2.

En plus, Donner la même priorité à deux tâches d'importances différentes, conduit à un choix arbitraire pendant l'étape d'affectation et cela risque de nous éloigner du bon choix. Une liste de la forme Max{.,.} permet d'éliminer ces inégalités par rapport à L_1 ou L_2. En effet, on a :

$L = \max \{L_1, L_2\}$ où L_1 et L_2 sont deux listes de priorités.

$L(T_1) = \max \{ L_1(T_1), L_2(T_1) \}$ et $L(T_2) = \max \{L_1(T_2), L_2(T_2)\}$

Si L_1 donne une priorité identique aux tâches T_1 et T_2, les situations possibles sont:

- $L_2(T_1) < L_1(T_1) = L_1(T_2) < L_2(T_2)$ alors $L(T_2) > L(T_1)$ et T_2 est prioritaire à T_1.

- $L_2(T_1) > L_1(T_1) = L_1(T_2) > L_2(T_2)$ alors $L(T_2) < L(T_1)$ et T_1 est prioritaire à T_2.

- $L_2(T_1) < L_1(T_1) = L_1(T_2) > L_2(T_2)$ alors $L(T_2) = L(T_1)$ et le choix reste arbitraire.

- $L_2(T_1) > L_1(T_1) = L_1(T_2) < L_2(T_2)$ alors $L(T_i) = L_2(T_i)$, i=1, 2 et le choix est selon L_2.

Le nombre de fois, où un choix arbitraire entre deux tâches est à faire, est diminué.

Des résultats des graphes, on peut en déduire une meilleure liste de priorité. En effet, Max est meilleure pour des graphes de densité faible inférieure à 45% et SOM est meilleure pour des densités fortes supérieure à 45%, la liste les combinant, définie par :

MS = Max si $d \leq 0.45$ et MS = SOM si $d > 0.45$, est meilleure que Max et SOM .

Dans ce qui a précédé, on a pu comparé les listes entre elles en les classant. Maintenant mesurons l'écart entre les valeurs des solutions des listes et une borne inférieure de la solution optimale. Le rapport d'approximation expérimental moyen défini précédemment est mesuré en fonction de la densité moyenne. Pour le même ensemble de jeux d'essai que précédemment ($p_j \in]0,1]$), on obtient le graphe de la figure 5.5.

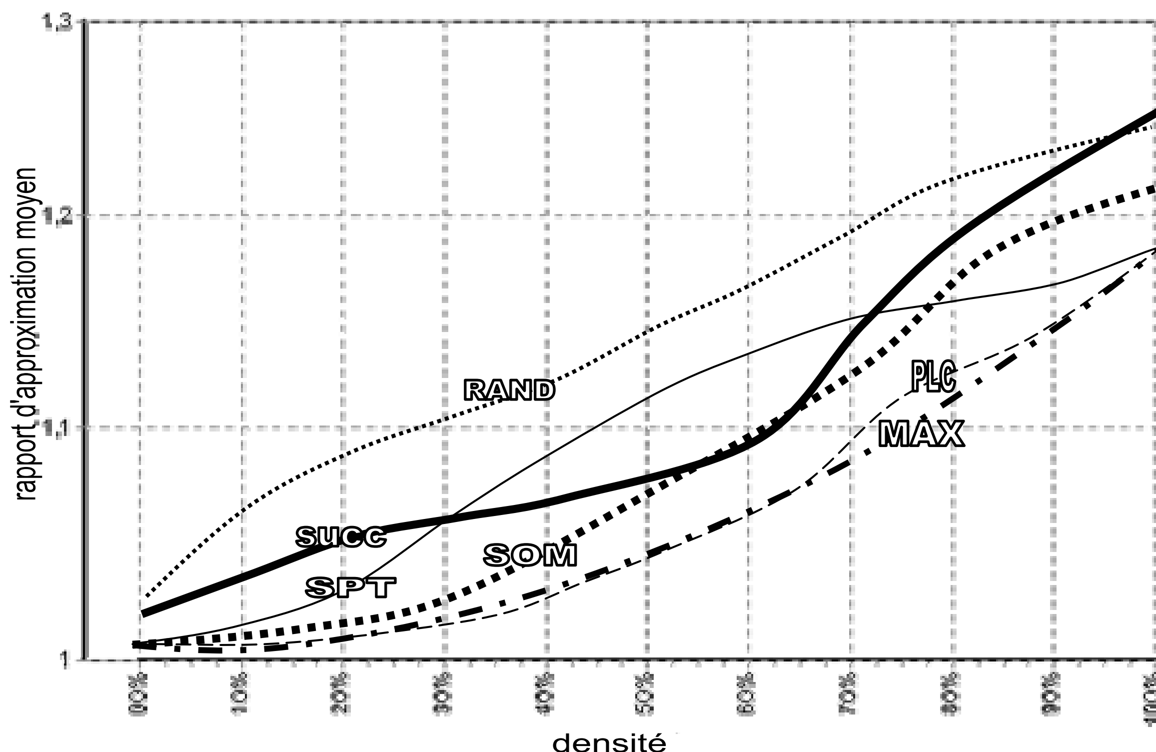


Figure 5.5 : Rapport d'approximation moyen , durées d'exécution uniformément distribuées

On remarque que, pour toutes les densités, les listes PLC et Max sont les meilleures en moyenne.

Remarque : Les résultats des listes sont proches car une liste quelconque est 2-approximatif. $1 \leq \rho_L \leq 2$ où L est une liste quelconque. Le rapport d'approximation différentiel devra représenter mieux graphiquement les performances des liste puisque il prend en compte le dispersement des valeurs objectifs.

5.5 Conception et résultats de l'AG

Afin d'étendre le champ d'application des AGs à un problème particulier comme celui de $P|prec|C_{max}$, il faut prendre en compte les spécificités du problème, en particulier les contraintes de précédence entre les tâches.

5.5.1 Le codage

On a choisit de coder indirectement une solution (ordonnancement) par la liste de priorité qui donne l'ordonnancement. La permutation des tâches formant la liste est le chromosome représentant l'individu ou la solution. Une population est formée d'un ensemble de permutations de n tâches.

Dans un problème à 8 tâches T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, les permutations suivantes constituent une population de quatre individus.

$T_1\ T_3\ T_8\ T_5\ T_7\ T_6\ T_2\ T_4$

$T_8\ T_3\ T_1\ T_6\ T_2\ T_4\ T_5\ T_7$

$T_3\ T_4\ T_1\ T_7\ T_2\ T_5\ T_8\ T_6$

$T_3\ T_7\ T_4\ T_1\ T_5\ T_2\ T_6\ T_8$

Une solution est obtenue à partir d'un individu (liste) en affectant les tâches l'une après l'autre selon la règle d'affectation sans délai.

5.5.2 La population initiale

Pour construire la population initiale, on a étudié deux approches :

- Génération aléatoire de permutations réalisables (respectant la relation de précédence).

- Prendre les permutations construites en utilisant les six règles de priorités vues auparavant : PLC, SPT, SUCC, RAND, SOM, Max{.,.}.

5. 5. 3 L'évaluation et la sélection

L'évaluation permet de calculer la fitness d'un individu. En cas de problème de minimisation, la finesse croit inversement avec la fonction objectif. La finesse d'un individu x_i est mesurée par $f(x_i) = \frac{1}{C(x_i)}$ où $C(x_i)$ est la longueur d'ordonnancement obtenu avec x_i. La probabilité de sélection d'un individu x_i, notée $prob(x_i)$, est proportionnelle à sa finesse $f(x_i)$ et telle que $\sum f(x_i) = 1$. Elle est définie par :

$$prob(x_i) = \frac{f(x_i)}{\sum_{j=1}^{j=6} f(x_j)}$$

Par exemple, pour une population de six individus générés aléatoirement, on obtient :

Tableau 5.1 : Exemple avec la fitness $\frac{1}{C_{max}}$

C_{max}	$fit = \frac{1}{C_{max}}$	prob
2.67 E+2	3.74 E-3	1.65 E-1
2.75 E+2	3.63 E-3	1.60 E-1
2.56 E+2	3.90 E-3	1.72 E-1
2.63 E+2	3.80 E-3	1.68 E-1
2.61 E+2	3.83 E-3	1.69 E-1
2.7 0E+2	3.70 E-3	1.63 E-1

On remarque que les probabilités de sélection sont proches, un individu fort a une probabilité de sélection proche de celle d'un faible individu. Ça vient du fait que les valeurs objectifs le sont aussi car pour le problème $P|prec|C_{max}$, on a :

$$OPT \leq C(x) \leq 2.OPT \text{ pour une liste x quelconque.}$$

On a utilisé une deuxième mesure de fitness qui évite cet inconvénient. Elle prend en compte le dispersement des valeurs objectifs de toute la population.

$$f(x_i) = \frac{mx - C(x_i)}{mx - mn} + \frac{1}{T}$$

Où : $C(x_i)$ la longueur d'ordonnancement obtenu avec x_i. mx = max { C(x), x individu } et mn = min{C(x), x individu }. T est la taille de la population.

$\frac{1}{T}$ représente, pour un individu, un minimum de chance d'être sélectionné. Pour la même population du tableau précédent, on obtient :

Tableau 5.2 : Exemple avec la fitness définie

C_{max}	fit = .	prob
2.67 E+2	6.21 E-1	1.46 E-1
2.75 E+2	2.00 E-1	4.70 E-1
2.56 E+2	1.20 E 0	2.82 E-1
2.63 E+2	8.32 E-1	1.96 E-1
2.61 E+2	9.37 E-1	2.20 E-1
2.70 E+2	4.63 E-1	1.09 E-1

On remarque que les probabilités de sélection sont dispersées, tout en étant proportionnelles aux fitness des individus.

La méthode de sélection utilisée est basée sur le principe de la roue de la fortune [50].

Algorithme 5. 3 : Sélection.

// On affecte à chaque individu x_i une probabilité cumulée//

Pour tout individu x_i de la population, calculer :

$$q_i = \frac{[f(x_1) + f(x_2) + ... + f(x_i)]}{[f(x_1) + f(x_2) + ... + f(x_p)]} \quad (p = 6);$$

//On sélectionne les individus suivant leurs probabilités cumulées//

Répéter

Générer un nombre r au hasard entre 0 et 1;

Si $r \leq q_1$ on choisit x_1, si $q_{i-1} < r \leq q_i$ on choisit x_i ;

Jusqu'à la sélection de tous les parents ;

Un individu fort peut être sélectionné plusieurs fois et un individu faible peut ne pas l'être du tout.

5. 5. 4 Le croisement

On a testé les deux opérateurs de croisement présentés au quatrième chapitre : Le croisement simple à un point et le croisement uniforme. Ils préservent la faisabilité des parents. Si les deux parents sont réalisables alors leurs fils l'est aussi.

5. 5. 5 La mutation

Pour un chromosome, la mutation transpose, un certain nombre de fois, deux tâches consécutives choisies aléatoirement.

Algorithme 5. 4 : Mutation.

Soit un chromosome, //une permutation de tâches//

Pour toutes les tâches de la permutation, sauf la dernière,

Faire

Générer un nombre r au hasard entre 0 et 1;

Si $r \leq p_m$ on permute entre la tâche courante et sa voisine à droite ;

5. 5. 6 Un mécanisme anti-clones

L'opérateur de croisement, ainsi que la méthode de sélection, peuvent produire beaucoup de clones qui ralentissent l'exploration de nouvelles solutions. On a choisit d'agir si

la population est identique, on remplace la moitié de la population par des solutions générées aléatoirement.

Ce mécanisme est efficace lorsque la taille de la population est petite, elle est de six dans nos tests. Comme le nombre de clones augmente rapidement, on a rapidement une population identique. Quand la taille de la population est grande, les tests montrent que le mécanisme fonctionne mal. Les clones envahissent la population à travers les générations sans être toute la population, à cause de l'opérateur de mutation, le mécanisme anti-clones est alors retardé.

Algorithme 5. 5 : Anti-clones.

A chaque génération

Si $\sum_{i=1}^{T} f(x_i) = T.f(x_1)$ alors

Remplacer la moitié des individus de la population par des individus

générés aléatoirement ;

5. 5. 7 Le critère d'arrêt

On arrête le déroulement de l'algorithme si on atteint un certain nombre de générations qui ne doit pas être trop grand, sinon l'AG génère une grande partie de l'espace des solutions, ce qui diminue son efficacité. Aussi, ce nombre ne doit pas être trop petit, sinon on ne donne pas suffisamment de temps à l'algorithme pour améliorer la population.

L'idéal est d'avoir une fonction qui calcule ou estime, dans un temps raisonnable, le nombre de solutions possibles. On arrête l'exécution de l'algorithme si on explore un certain pourcentage de ces solutions. Dans l'absence d'une telle fonction et pour une population de 6 individus, on a fixé le nombre d'itération à 30, donc 180 solutions. Ce choix est efficace car avec 100 tâches et une densité forte (nombre de solutions réalisables faible), le nombre de solutions est de l'ordre de 10^3, donc 180 est inférieur à 20% de 10^3.

5. 5. 8 Résultats des AGs

Les variantes suivantes sont étudiées :

- Croisement simple, sans mutation
- Croisement uniforme, sans mutation
- Pas de croisement, mutation
- Croisement simple et mutation.

Les tests ont montré que les différentes variantes de l'AG donnent des performances proches les unes des autres, en ajustant convenablement les paramètres de l'algorithme. Ainsi on peut avoir des résultats de la variante 'croisement simple et une mutation à $p_m = 0.2$' proches de ceux de la variante 'croisement uniforme et une mutation à $p_m=0.1$'.

Une variante de l'algorithme génétique est notée par G (a, b, c) où :

$$a = \begin{cases} 0 : \text{population initiale aléatoire} \\ 1 : \text{population initiale à partir des listes} \end{cases} \qquad b = \begin{cases} 0 : \text{pas de croisement} \\ 1 : \text{croisement à un point} \\ 2 : \text{croiement uniforme} \end{cases}$$

$$c = \begin{cases} 0 : \text{pas de mutation} \\ p_m : \text{mutation avec une probabilité } p_m \end{cases}$$

Les différentes variantes étudiées améliorent la population initiale aléatoire lentement (après un nombre d'itérations considérable). Il est préférable de prendre la population initiale construite à partir des six listes et essayer de l'améliorer par l'algorithme génétique.

Pour des instances de durées d'exécutions unitaires, les AGs n'apportent pas d'améliorations, ces instances sont en général des cas faciles, PLC les résout exactement.

Le graphique de la figure 5.6 donne le pourcentage de fois où la variante G(1,1,0.1) améliore les résultats des listes à travers les générations. Pour des instances de 100 tâches et 3 machines, les cas de graphes de faible, moyenne et forte densité sont représentés dans le même graphique. Après 30 générations, une fois sur quatre, l'AG améliore les résultats des listes.

Pour les instances de faible densité, les graphes de précédence sont, en général, sous formes d'arbres. PLC donne un résultat optimal ou proche de l'optimum. L'AG obtient rapidement l'optimum.

Pour ceux de forte densité, l'espace des solutions réalisable est réduit. L'AG explore rapidement presque la totalité des solutions et l'optimum est, souvent, atteint.

Pour les instances de moyenne densité, les graphes sont de formes variées et le nombre de solutions réalisables est suffisamment grand pour permettre à l'AG d'améliorer les résultats des listes à travers les générations.

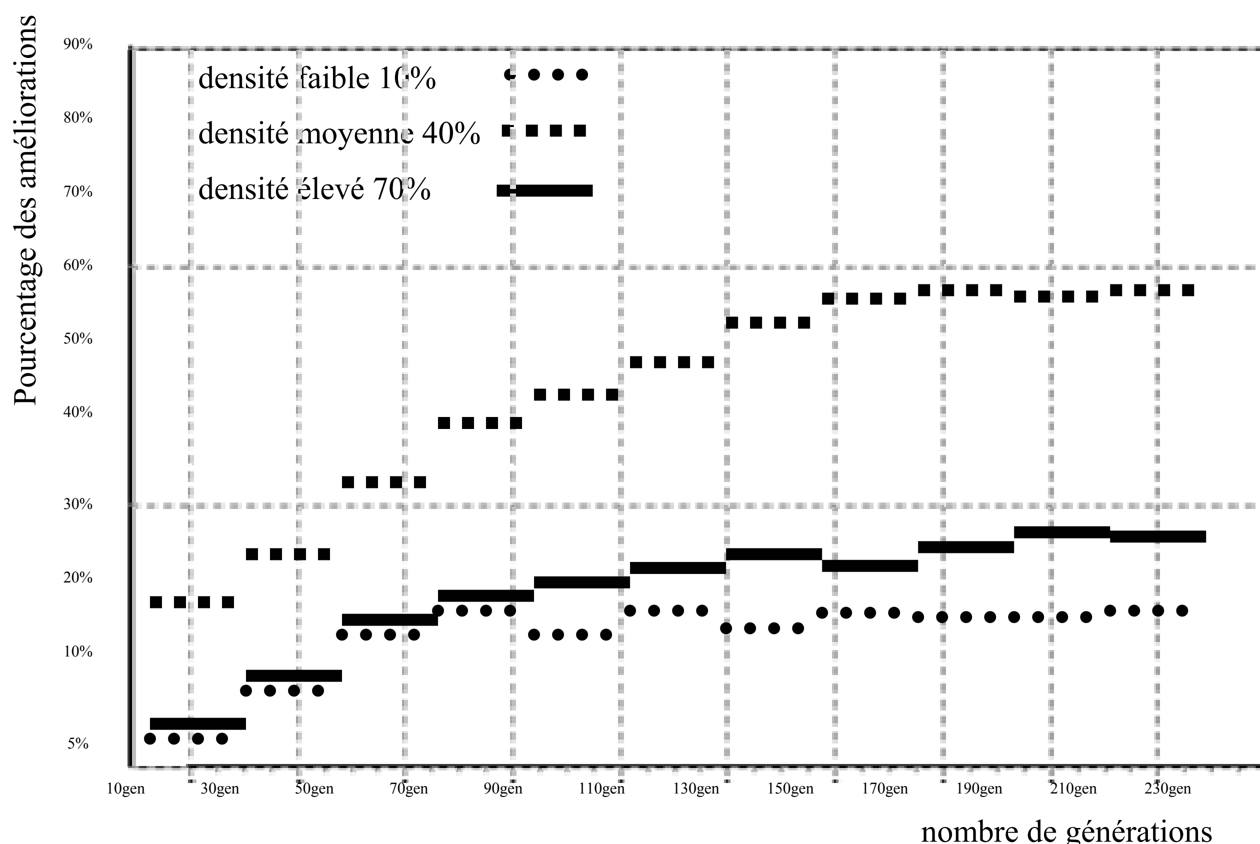


Figure 5.6 : Nombre d'améliorations en fonction du nombre de générations

5.6 Les temps de calcul

Les temps de calcul des six listes est petit par rapport à celui des quatre variantes de l'AG. Des jeux d'essais avec m = 3 et une densité de l'ordre de 50%, ont donnés les temps (en seconde) représentés dans le tableau suivant:

Tableau 5.3 : Temps de calcul des six listes et des variantes de l'AG

	PLC	SPT	SUCC	RAND	SOM	MAX	G(0,1,0)	G(0,0,0.1)	G(0,1,0.1)	G(0,2,0)
100tâches	<1	<1	<1	<1	<1	<1	0.44	0.82	0.77	0.44
200tâcjes	<1	<1	0.05	<1	<1	0.17	1.76	3.07	3.07	1.92
300tâches	0.06	0.05	<1	0.06	0.05	0.55	6.31	10.71	10.87	5.98
400tâches	0.05	0.11	0.06	0.06	0.11	1.3	15.54	27.46	27.57	15.82
500tâches	0.16	0.11	0.17	0.11	0.16	2.64	29.99	55.69	54.49	28.4
800tâches	0.5	0.55	0.49	0.5	0.55	11.37	97.55	175.04	198.82	105.96

Les résultats montrent que, d 'un point de vu pratique, les méthodes de listes sont meilleures que les AGs, puisqu'elles donnent des solutions satisfaisantes dans un temps raisonnable. Si on dispose de suffisamment de temps, les AGs peuvent être utilisés pour améliorer les résultats des listes.

SCHUURMAN et WOEGINGER [4] ont proposé le problème ouvert suivant :

Problème Ouvert :

- Concevoir un algorithme d'approximation polynomial pour le problème $P|prec|C_{max}$ ou $P|prec, p_j=1|C_{max}$, avec un rapport du plus mauvais cas 2-δ (même un algorithme exponentiel en m est intéressant).
- Prouver la non existence d'un algorithme $(\frac{4}{3}+\delta)$-approximatif pour $P|prec|C_{max}$.
- Concevoir un schéma d'approximation polynomial (PAS) pour le problème $P_2|prec|C_{max}$.
- Concevoir un PAS (ou même un FPAS) pour $P_3|prec, p_j=1|C_{max}$.

CONCLUSION GENERALE ET PERSPECTIVES

Face aux problèmes NP-difficiles, on se contente, souvent d'une résolution approchée et rapide au lieu d'une résolution exacte mais lente. On a étudié deux principaux types de méthodes approchées: les AGs, qui ont un caractère stochastique et n'offrent aucune garantie sur la qualité de la solution obtenue, et les méthodes de liste, déterministes, très rapides et garantissent une certaine qualité de la solution.

Les algorithmes de type listes sont des algorithmes gloutons qui prennent une décision en se basant sur l'information locale. Ne pas décider optimalement à une étape, affectera sûrement les décisions ultérieures et nous éloignera d'avantages de la solution optimale. Une solution consiste à concevoir des algorithmes de liste dynamique, qui prennent en considération les informations nouvelles dues aux affectations. Ces informations, pour des raisons du temps de calcul, sont limitées. Pour le problème $P|prec|C_{max}$, l'utilisation d'informations sur les tâches successeurs d'une tâche (PLC, SOM, Max) peut améliorer les performances de la méthode de liste.

Nous avons essayer d'améliorer les résultats des listes par un AG (hybridation des algorithmes génétiques avec les méthodes de listes) qui considère les listes de priorités comme population initiale. Un chromosome est une liste de priorité des tâches (un gêne est une tâche). Une autre façon de le faire, consiste à considérer un chromosome comme une liste de règles d'affectation utilisées à chaque affectation (un gêne est une règle d'affectation).

Il serait intéressant de comparer ces résultats avec d'autres méthodes d'amélioration par voisinage qui réduisent la population à un seul individu, comme le recuit simulé ou la méthode tabou, en utilisant le même codage et les mêmes techniques de construction de solutions que pour les AGs. La mutation peut jouer le rôle de la fonction de voisinage.

REFERENCES

[1]: Hurink J., "Solving optimization problems by local search", Habilitationsschrift, Osnabrük, (1998).

[2]: Phelippeau-Gelineau L., "Etude de problèmes d'ordonnancement multiprocesseur avec communication par diffusion", PhD thesis, Université Paris 6, (1996).

[3]: Yu-Kwong K. et Ishfaq A., "Static scheduling algorithms for allocating directed task graphs to multiprocessors", ACM Computing Surveys, vol. 31, no. 4, (1999).

[4]: Schuurman P. et Woeginger G.J., "Polynomial time approximation algorithms for machine scheduling: Ten open problems", Journal of Scheduling 2, (1999), 203-213.

[5]: Aytug H., Khouja M. Et Vergara E., "Use of genetic algorithms to solve production and operations management problems: a review", Int. J. Prod. Res. 41 no. 17, (2003), 3955-4009.

[6]: Chiu N., Fang S. et Lee Y., "Sequencing parallel machining operations by genetic algorithms", Comuter and Industrial Engineering 36, (1999), 259-280.

[7]: Lopez P. et Roubellat F., "Ordonnancement de la production", Hermes Science Publications, Paris 2001.

[8] : Pinedo M., "Scheduling theory, algorithms and systems", Prentice Hall, (1995).

[9]: Graham R.L, Lawler E.L. et Lenstra J.K. et Rinnoy Kan A.HG., "Optimisation and approximation in deterministic sequencing and scheduling: asurvey", Discrete Mathematics 5, (1979), 287-326.

[10]: Giroudeau R., "Cours d'introduction à l'ordonnanement", D.E.A. (2003) V1.0, Montpelier, France.

[11]: Schuurman P., "Approximating schedules", PROEFSCHRIFT, Eindhoven, The Netherlands 2000.

[12]: Engels D.W., Feldman J., Karger D.R. et Ruhl M., “Parallel processor scheduling with delay constraints”, MIT Laborator for Computer Science, Cambridge, MA 02139, USA.

[13]: Graham R.L., “Bounds of certain multiprocessing anomalies”, Bell System Tech. J. 45, (1966), 1563-1581.

[14]: Cook S.A., “The complexity of theorem-proving procedures”, Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC), (1971), 151-158.

[15]: Karp R.M., “reducibility among combinatorial problems”, dans: Miller R.E. et Thatcher J.W. (eds.), “Complexity of computer computations”, Plenum Press, New York (1972), 85-104.

[16]: Rothkopt M.H., ”scheduling independant tasks on parallel processors”, Management Science, vol. 12, No. 5, (Jan. 1966).

[17]: Garey M.R. et Johnson D.S., “Complexity results for multiprocessor scheduling under resource constraints», SIAM Journal on Computing 4, (1978), 397-411.

[18]: McNaughton R., «scheduling with deadlines and loss functions», Management Sci. 6, (1959), 1-12.

[19]: Ullman J.D. «Complexity of sequencing problems», In Bruno J.I, Coffman F.G., Graham R.I., Kohler W.H, Sethi R., Steiglitz K. et ullman J.D., editors, «Computer and Job/Shop Scheduling Theory», John Wiley & Sons Inc., New York, (1976).

[20]: Coffman E.G. et Graham Jr. R.L., ”optimal scheduling for two-processor systems”, acta Inform. 1, (1972), 200-213.

[21]: Hu T.C., «Parallel sequencing and assembly line problems», Oper. Res. 9, (1961), 841-848.

[22]: Lageweg B.J. et Lenstra J.K. et Rinnoy Kan A.H.G., ”Minimizing maximum Lateness on one Machine: computatuional experience and some appilacations”, Statistica Nederlandeeca, vol. 30, no. 1, (1976), 25-41.

[23]: Backer K.R. et Su Z.-S., «Sequencing with due-dates and early start times to minimize maximum tardiness”, Naval Research Logistics Quarterly, vol. 21, no. 1, (Mar. 1974), 171-176.

[24]: McMahon G. et Florian M., «On scheduling with ready times and due dates to minimize maximum lateness”, Oper. Res., vol. 23, no. 3, (May/June 1975), 475-482.

[25]: Cormen T., Leiserson C. et Rivest R., «Introduction to algorithms», The MIT Press, Cambridge, Massachusetts (1990).

[26]: Lawler E.L. et Moore J.M., «A functional equation and its application to resource allocation and sequencing problems”, Management Sci. 16, 77-84.

[27]: Silver E., “An overview of heuristic solution methods”, Journal of the operational research society, 55(9), (2004), 936–956.

[28]: Graham R.L.,»Bounds on multiprocessing anomalies», SIAM Journal of Applied Mathematics 17, (1969), 263-269.

[29]: Coffman E.G., Frederickson Jr. G.N. et Luecker G.S., ”A note on axpected makespans for largest-first sequences of independent task on two processors”, Math. Oper. Res. 9, (1984), 260-266.

[30]: Coffman, Jr. E.G., Frederickson G.N. et Luecker G.S., “Probabilistic analysis of the LPT processor scheduling heuristic”, (article non publié), (1983).

[31]: Garey M.R. et Johnson D.S., ”Strong NP-completeness results: Motivation, examples, and applications” Journal of the ACM 25, (1978), 499-508.

[32]: Schuurman P. et Woeginger G.J., “Approximation schemes-a tutorial“, Preliminary version of a chapter of the book “Lectures on Scheduling”, to appear around 2007 A.D.

[33]: Goemans M.X., “Approximation algorithms”, Advanced Algorithms, (1994), 18.415 / 6.854.

[34]: Sanhi S. Et Gonzalez T.F., ”P-complete approximation problems”, journal of the ACM 23, (1976), 555-565.

[35]: Garey M.R. et Johnson D.S., ”The complexity of near-optimal graph coloring”, journal of the ACM 23, (1976), 43-49.

[36]: Lenstra J.K. et Rinnoy Kan A.H.G., ”complexity of scheduling under precedence constraints”, Oper. Res. 26, (1978), 22-35.

[37]: Arora S., Lund C., Motwani R., Sudan M. et Szegedy M., ”Proof verification and hardness of approximation problems”, Proceedings of the 33rd IEEE Symposium on the Foundations of Computer Science (FOCS), (1992), 14-23.

[38]: Demange M. et Paschos V., «Autour de nouvelles notions pour l’analyses des algorithmes d’approximation : Formalisme unifié et classes d’approximation», RAIRO Oper. Res. 36 (2002) 237-277.

[39]: Papadimitriou C.H. et Yannakakis, “Optimization, approximation and complexity classes”, Journal of Computer and System Sciences 43, (1991), 425-440.

[40]: Ovacik I.M. et Uzsoy R., “Worst-case error bounds for parallel machine scheduling problems with bounded sequence-dependant setup times”, Oper. Res. Lett. 14, (1993), 251-256.

[41]: Kwok Y.K. et Ahmad I., “static scheduling algorithms for allocating directed tasks graphs to multiprocessors”, ACM Computing Surveys, Vol. 31, No. 4 (Dec. 1999).

[42]: Sih G. C. et LEE E. A., «A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures”, IEEE Trans. Parallel Distrib. Syst. 4, 2 (Feb.), 75–87.

[43]: Gerasoulis A. et Yang T., «On the granularity and clustering of directed acyclic task graphs”, IEEE Trans. Parallel Distrib. Syst. 4, 6 (Jun. 1993), 686–701.

[44]: Rinnoy Kan A.H.G., “An introduction to the analysis of approximation algorithms”, Discr. Appl. Math, vol. 14 (1986), 171-185.

[45]: Chen N.F. et Liu C.L., »On a class of scheduling algorithms for multiprocessors computing systems”, dans: Feng T.-Y. (ed.), «Parallel processing, lectures notes», Computer Science 24, Springeer Verlag, Berlin, (1975), 1-16.

[46]: Lam S. et Sethi R., “worst case analysis of two scheduling algorithms”, SIAM J. Comput. 6, (1977), 518-536.

[47]: Schutten J.M.J., “List scheduling revisited”, Oper. Res. Lett. 18, (1996), 167-170.

[48]: Stern H.E., ”Minimizing makespan for independent jobs on non identical parallel machines-an optimal procedure”, Working Paper 2/75, Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva.

[49]: Holland J.H., “Adaptation in natural and artificial systems”, Cambridge, Mass: MIT press (1975).

[50]: Goldberg D.E., “Genetic Algorithms in Search, Optimization and Machine Learning”, Addison-Wesley Publishing Company, Inc. (1989).

[51]: Della Croce F., Tadei R. et Volta G., “A genetic algorithm for the job-shop problem”, 3rd International Workshop on PMS, Como, Italy, Comput. Oper. Res., to appear (1992).

[52]: Lash S., “Genetic Algorithms for weighted tardiness scheduling on parallel machines”, Technical report 93-01, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois, USA (1993).

[53]: Cerf R., «Une théorie asymptotique des algorithmes génétiques’’, These de Doctorat, Université de Montpellier II, (1994).

[54]: Freidlin M. I. and Wentzell, «Random perturbations of dynamical systems’’, Springer Verlag, New-York, (1983).

[55]: Michalewicz Z., «Genetic Algorithms + Data Structures = Evolution programs», Springer Verlag, New-York, (1991).

[56]: Uckun S., Bagchi S. Et Kawamuara K., «Managing genetic search in job shop scheduling», IEEE Expert, 8 (5), (1993), 15-24.

[57]: Lawton G., “Genetic Algorithms for Schedule Optimization”, AI Expert, (1992), 23-27.

[58]: Homaifar A., Qi C.X et Lai S.H., «Constrained optimization via genetic algorithms», Simulation 62(4), (1994), 242-253.

[59]: Hartmann S., “A competitive genetic algorithm for resource-constrained project scheduling”, Naval Research Logistics 45, (1998), 733-750.

[60]: Woerlee A.P., “Decision support systems for production scheduling”, Ph. D. Thesis, Erasmus University, Rotterdam, (1991).

[61]: Michalewicz Z., “ Evolutionary computation techniques for non linear programming“, International Transactions on Operational Research, 1(2), (1994), 223-240.

[62]: Van Laarhoven P. et Aarts E., “Simulated Annealing : Theory and Applications”, Kluwer, 1987.

[63]: RARDIN R.L. et UZSOY R., “Experimental Evaluation of Heuristic Optimization

Algorithms: A Tutorial“, Journal of Heuristics, 2, (2001), 261–304.