A-PDF MERGER DEMO

N°d'ordre :

Université*MOHAMED BOUDIAF* DE M'SILA
Faculté des sciences et sciences de l'ingénieur
Département d'informatique

MEMOIRE

Présenté pour l'obtention du diplôme de :

Magister

Spécialité : Informatique
Option : Informatique industrielle

Par :
HEMMAK Allaoua

Thème :

RESOLUTION D'UN PROBLEME D'ORDONNANCEMENT SUR UNE MACHINE AVEC DATE ECHUE COMMUNE PAR LA PROGRAMMATION DYNAMIQUE ET LA RELAXATION LAGRANGIENNE

Soutenu publiquement le : 17/01/2007 devant le jury composé de :

Dr. BOUDERAH Brahim	**M.C. UMB- M'sila**	**PRESIDENT**
Dr. BELOUADAH Hocine	**M.C. UMB- M'sila**	**RAPPORTEUR**
Dr. MIHOUBI Douadi	**M.C. UMB- M'sila**	**EXAMINATEUR**
Dr. MOUSSAOUI Abdelouahab	**M.C. UFA- Sétif**	**EXAMINATEUR**

REMERCIEMENTS

Je tiens à exprimer mes plus vifs remerciements et toute ma gratitude à :

- Mon encadreur, M. Hocine BELOUADAH pour avoir accepté de me proposer le sujet de ce mémoire, pour avoir bien su, grâce à sa compétence et son riche expérience, me guider dans la réalisation de ce modeste travail et pour ses précieux conseils ; je tiens à lui exprimer mon plus profond respect pour sa patience et pour tous les efforts qu'il a consentis et tout le temps précieux qu'il a consacré dans la réalisation de ce travail ;

- Tous les membres de jury pour avoir accepté de juger ce travail, pour leurs précieuses recommandations et pour leurs fructueux conseils qui ont nettement apprécié la finalité de ce mémoire; c'est un grand honneur et un grand plaisir que :

  Dr. BOUDERAH Brahim M.C. UMB- M'sila

  Dr. MIHOUBI Douadi M.C. UMB- M'sila

  Dr. MOUSSAOUI Abdelouahab M.C. UFA- Sétif

  aient accepté de juger mon travail ;

- Je tiens, tout particulièrement, à exprimer mon plus profond respect et ma sincère gratitude au Dr. BOUDEREH Brahim pour son soutien, pour sa générosité et pour tout ce qu'il a déployé, dès le premier jour du magistère et jusqu'à la dernière minute, afin que notre tâche s'achève dans les meilleures conditions, et qu'il sache que je n'oublierai jamais cette immense dette.

- Je tiens aussi à remercier chaleureusement Pr. OSMANE Brahim de AUB (American University Of Beirut) d'abord, pour m'avoir invité à AUB, pour son chaleureux accueil à Beyrouth, pour ses précieux conseils, pour les outils qu'il a mis à notre disposition et pour tout ce qu'il a consenti durant notre séjour à Beyrouth en dépit de ses multiples préoccupations et malgré les dures

circonstances qui ont coïncidé avec notre séjour à Beyrouth. Je vous prie, monsieur, d'agréer, ma respectueuse gratitude et mes sentiments les plus dévoués.

- Mr Osmane, thanks for all things that you have done to us, sincerely, it's a great honour to working with you. I'll never forget your generosity.

Je tiens également à exprimer mes plus vifs remerciements à :

- Tout le corps administratif du département d'informatique de l'université Mohamed Boudiaf de M'sila pour les efforts qu'ils ont consentis afin de faciliter l'achèvement de ce travail;

- Toute la promo 2006 du magister informatique de l'université Mohamed Boudiaf de M'sila pour leurs soutien et conseils lors de la réalisation de ce travail;

- Toute personne ayant, de près ou de loin, contribué à la réalisation de ce modeste travail.

PLAN DU MEMOIRE

Par le biais de ce mémoire, on se propose d'aborder une synthèse exhaustive de l'approche dite "*Programmation dynamique*" en l'appliquant à un problème d'ordonnancement : "*minimisation de la somme des coûts des avances et des retards avec date échue commune sur une seule machine*", de décrire la formulation dynamique de certains problèmes d'optimisation à caractère combinatoire ayant trait à la recherche opérationnelle, et de concevoir une approche fondée sur le procédé de la relaxation de l'espace des états. En effet, ce mémoire est principalement composé :

- ✓ d'un chapitre premier consacré à développer minutieusement la théorie de la programmation dynamique dotée d'exemples choisis pour mettre en évidence l'utilisation de cette technique ainsi que ses modalités d'application ;
- ✓ d'un second chapitre intitulé "Notions d'ordonnancement" illustrant la terminologie et les techniques de l'ordonnancement où on évoquera certains exemples parmi lesquels celui qu'on se propose d'étudier;
- ✓ d'un troisième chapitre entièrement dédié à la présentation du volet pratique de ce travail, en l'occurrence: application de la programmation dynamique à un problème d'ordonnancement de tâches indépendantes sur une seule machine et, en tant qu'alternative, conception et implémentation de la relaxation lagrangienne au problème en question, avec des commentaires qu'on jugera utiles qui consisteront en une étude comparative;

RESUME

L'objet de ce mémoire est l'implémentation de la méthode de la programmation dynamique appliquée à un problème d'ordonnancement intitulé : « minimisation de la somme des coûts des avances et des retards avec date échue commune sur une seule machine ». Vu le nombre exponentiel des états requis par cette méthode, on tente de développer une approche fondée sur la récursivité dynamique mais en tronquant certains états : la relaxation de l'espace des états pour trouver une solution approchée, et, dans certains cas, une solution optimale au problème posé.

Mots clés : Programmation Dynamique ; Ordonnancement sur une seule machine ; Date échue commune ; Coûts des avances et des retards ; Relaxation de l'espace des états.

ABSTRACT

The object of this memoir is the implementation of the dynamic programming method used to solving a scheduling problem : minimizing the sum of earliness and tardiness penalties with common due date on a single machine. Since the exponential number of states that the dynamic programming method requires, we try to develop an approach based on the dynamic recursion but by cutting some states: states space relaxation for getting near solution or, for some cases, exact solution to the problem posed.

Key words: Dynamic Programming; Single machine Scheduling; Common Due Date; Earliness Tardiness Penalties; State Space Relaxation.

ملخـــــص

موضوع هذه الرسالة هو تطبيق طريقة البرمجة الحيوية لحل مسألة ترتيبية: تخفيض مجموع تكاليف التقديم والتأخير بتاريخ مستحق مشترك على آلة واحدة. نظرا للعدد الأسي للحالات التي تقتضيها هذه الطريقة، نحاول إنجاز مقاربة مؤسسة على التراجع الحيوي ولكن بقطع بعض الحالات: اختزال فضاء الحالات لإيجاد حل مقرب وفي بعض الحالات حل مضبوط للمسألة المطروحة.

كلمات مفتاحية: البرمجة الحيوية؛ الترتيب على آلة واحدة؛ تاريخ مستحق مشترك؛ تكاليف التأخير و التقديم؛ اختزال فضاء الحالات.

INTRODUCTION

CHAPITRE PREMIER

THEORIE DE LA PROGRAMMATION DYNAMIQUE

1. Introduction

2. Historique

3. Généralités

- **3.1. Processus de décisions séquentiel**
- **3.2. Sous-politique / Politique**
- **3.3. Sous politique optimale / Politique optimale**
- **3.4. Hypothèse de Markov**
- **3.5. Principe d'optimalité / Théorème d'optimalité**
- **3.6. Récurrence avant. Récurrence arrière**

4. Programmation dynamique en horizon illimité

5. Cas stochastique

6. Exemples

- **6.1. Un exemple de gestion des stocks**
- **6.2. Un deuxième exemple : problème du sac-à-dos**

7. Conclusion

CHAPITRE II

NOTIONS D'ORDONNANCEMNT

1. Introduction
2. Généralités
    2.1. Tâches
    2.2. Les types de problèmes d'ordonnancement
    2.3. Les contraintes
    2.4. Les ressources
    2.5. Les critères
    2.6. Notation
3. Les méthodes de résolutions
    3.1. Enumération complète
    3.2. Méthode par séparation et évaluation
    3.3. Méthode de la programmation dynamique
    3.4. Méthodes approchées ou heuristiques (Heuristic Methods)
    3.5. Méthodes métaheuristiques (Metaheuristic Methods)
4. Complexité des algorithmes de résolution
5. Conclusion

CHAPITRE III

MINIMISATION DE LA SOMME DES COUTS DES AVANCES ET DES RETARDS AVEC DATE ECHUE COMMUNE SUR UNE SEULE MACHINE PROGRAMMATION DYNAMIQUE ET RELAXATION DE L'ESPACE DES ETATS

1. Introduction

2. Présentation et historique

3. Formulation du problème

4. Résolution par la programmation dynamique

- **4.1. Récurrence avant**
- **4.2. Récurrence arrière**
- **4.3. Algorithme dynamique approché**
- **4.4. Règles de dominance**

5. Heuristique de calcul d'une solution approchée

- **5.1. Présentation**
- **5.2. Algorithme**
- **5.3. Commentaires et discussions**
- **5.4. Tests et résultats**

6. Relaxation de l'espace des états

- **6.1. Relaxation lagrangienne**
- **6.2. Relaxation de l'espace des états**
- **6.3. Algorithme de relaxation de l'espace des états**
- **6.4. Commentaires et discussions**
- **6.5. Tests et résultats**

7. Conclusion

CONCLUSION GENERALE

BIBLIOGRAPHIE

SOMMAIRE

INTRODUCTION 2

CHAPITRE PREMIER

THEORIE DE LA PROGRAMMATION DYNAMIQUE

1. Introduction 7

2. Historique 7

3. Généralités 8

3.1. Processus de décisions séquentiel 10

3.2. Sous-politique / Politique 10

3.3. Sous politique optimale / Politique optimale 11

3.4. Hypothèse de Markov 12

3.5. Principe d'optimalité / Théorème d'optimalité 13

3.6. Récurrence avant. Récurrence arrière 14

4. Programmation dynamique en horizon illimité 17

5. Cas stochastique 18

6. Exemples 20

6.1. Un exemple de gestion des stocks 20

6.2. Un deuxième exemple : problème du sac-à-dos 25

7. Conclusion 27

CHAPITRE II

NOTIONS D'ORDONNANCEMNT

1. Introduction 30

2. Généralités 30

2.1. Tâches 31

2.2. Les types de problèmes d'ordonnancement 32

2.3. Les contraintes 33

2.4. Les ressources 34

2.5. Les critères 34

2.6. Notation 37

3. Les méthodes de résolutions 38

3.1. Enumération complète 38

3.2. Méthode par séparation et évaluation ……39
3.3. Méthode de la programmation dynamique ……39
3.4. Méthodes approchées ou heuristiques ……42
3.5. Méthodes métaheuristiques ……43
4. Complexité des algorithmes de résolution ……44
5. Conclusion ……47

CHAPITRE III

MINIMISATION DE LA SOMME DES COUTS
DES AVANCES ET DES RETARDS
AVEC DATE ECHUE COMMUNE SUR UNE SEULE MACHINE
PROGRAMMATION DYNAMIQUE
ET RELAXATION DE L'ESPACE DES ETATS

1. Introduction ……49
2. Présentation et historique ……50
3. Formulation du problème ……53
4. Résolution par la programmation dynamique ……53
4.1. Récurrence avant ……53
4.2. Récurrence arrière ……55
4.3. Algorithme dynamique approché……59
4.4. Règles de dominance……61
5. Heuristique de calcul d'une solution approchée ……65
5.1. Présentation ……65
5.2. Algorithme ……66
5.3. Commentaires et discussions ……67
5.4. Tests et résultats ……67
6. Relaxation de l'espace des états ……68
6.1. Relaxation lagrangienne ……68
6.2. Relaxation de l'espace des états ……68
6.3. Algorithme de relaxation de l'espace des états ……71
6.4. Commentaires et discussions ……71
6.5. Tests et résultats ……72
7. Conclusion ……75
CONCLUSION GENERALE……77
Bibliographie ……80

INTRODUCTION

Les exigences de notre vie quotidienne ne cessent guère de nous faire face à de véritables problèmes qui sont de plus en plus complexes et qui nécessitent des outils puissants capables de nous acheminer vers les solutions de ces derniers. Cependant la diversité et le nombre important de ces problèmes font l'objet d'obstacles majeurs pour aboutir à de telle fin. Ce qui préconise l'identification, la classification et la modélisation de chacun d'eux pour pouvoir déterminer quelle méthode est adéquate et quels outils sont nécessaires pour mener à bien la résolution du problème en question.

En dépit des méthodes inventées au fil des années, les systèmes (économiques, industriels, informatiques, …) et leurs évolutions ont toujours imposé l'introduction de nouvelles techniques permettant de remédier aux handicaps éventuels des méthodes existantes et ont ainsi donné naissance à d'autres méthodes chacune pouvant faire face à une situation donnée.

La recherche opérationnelle est la discipline qui traite en particulier les problèmes d'optimisation combinatoires qui se manifestent principalement par le nombre important de solutions possibles qui doivent être soumises aux différents tests pour en choisir celles qui sont qualifiées de "solutions optimales", en d'autres termes : "meilleures solutions", s'il en existe bien sûr. En effet, elle a nécessité l'élaboration d'algorithmes permettant la résolution des problèmes posés avec le moindre coût en termes de temps et de ressources humaines et/ou matérielles. Malheureusement, ces algorithmes ne permettent pas toujours de confirmer que tel problème admet telles solutions et, dans la plupart des cas, ne peuvent même pas affirmer l'existence de celles-ci. En bref : ils ne sont pas toujours efficaces. En termes de complexité, on dit qu'ils ne sont pas toujours polynomiaux. En effet, ceci est devenu le souci majeur des chercheurs et spécialistes de la recherche opérationnelle ces dernières décennies et s'y sont penchés laborieusement en tentant de combler les lacunes générées par les algorithmes existants.

La programmation dynamique est l'une des méthodes d'optimisation de problèmes combinatoires qui a vu le jour il y a quelque décennies, et qui, son domaine d'application est de plus en plus vaste, vu son importance et sa simplicité en matière de recherche opérationnelle. C'est au mathématicien américain Richard Bellman que revient le mérite d'avoir conçu et introduit en 1957 [FRE82] cette méthode et c'est lui qui l'a baptisée ainsi. En fait, peu avant, en 1944, le français Pierre Massé [SAK84] appliqua cette même technique, sans utiliser le terme, pour planifier les investissements d'électricité de France et c'est ainsi que naquit l'idée de la programmation dynamique pour la première fois. A la différence de la programmation linéaire qui est définie comme l'étude d'une classe de problèmes posés indépendamment de la méthode de solution dont ils sont justiciables, la programmation dynamique se caractérise d'abord par une méthode de solution : l'application du principe d'optimalité, ou encore le théorème d'optimalité de Bellman : "*Toute politique optimale ne peut être formée que de sous politiques optimales*"[ROB79]. Bien que naturel, ce principe n'est pas toujours applicable! Prenons le cas des chemins dans un graphe, le principe fonctionne si on cherche le plus court chemin entre deux points, par contre, il ne marche pas si on cherche le plus long chemin sans boucle.

En général, la programmation dynamique concerne l'évolution dans le temps d'un système économique, informatique, industriel, … La transition d'un état à un autre peut être partiellement aléatoire (*intervention du hasard*) et partiellement contrôlée (*intervention de l'homme*). On peut distinguer les évolutions de première et seconde espèces, selon que, à chaque phase, l'intervention du hasard précède la décision humaine, ou au contraire, la suit. Evidemment, les cas limites sont ceux opposés :

- où le hasard disparaît (cas déterministe) ;
- où la décision humaine ne se manifeste pas (processus stochastique).

La programmation dynamique consiste alors à décomposer le problème initialement posé qualifié d'ordre n (*n entier naturel dit horizon*), en sous problèmes d'ordres respectifs 1,2,…,n-1,n ; puis, en résolvant ces derniers récursivement, on

déterminera une solution (*dite sous politique optimale*) [ROB79] à chacun d'eux, on aboutira en dernière phase, comme on peut le remarquer d'ailleurs, à en déduire la *politique optimale* (ou *stratégie optimale dans le cas stochastique*) [ROB79] du problème tout entier. C'est en quelque sorte, le "*diviser pour régner*". Cette décomposition est, dans la plupart des cas, temporelle, (c'est pour cette raison qu'on parle de phases au lieu de sous problèmes), ceci concerne les problèmes où le facteur temps intervient explicitement dans l'évolution du système en question, comme dans le cas des problèmes de gestion de stock, processus industriels, la planification économique,… En revanche, dans d'autres cas, où le facteur temps n'intervient pas explicitement, elle est artificielle, comme dans le cas des problèmes d'ordonnancement, du plus court chemin, du sac-à-dos, … Mais cette technique n'est applicable que si la solution de chacun de ces sous problèmes ne dépend que de celles de ses voisins directs (les plus proches) : l'antérieur et le postérieur, cette hypothèse dite *hypothèse de Markov*, n'est pas toujours satisfaite. En d'autres termes : Le problème à résoudre doit donc justifier que chaque état du système en question ne dépend du passé qu'à travers son état prédécesseur (*on parle de récurrence avant : forward dynamic programming*) [FRE82], et ne dépend du futur qu'à travers son état successeur (*on parle alors de récurrence arrière: backward dynamic programming*) [FRE82]. Pour certains problèmes, ces deux approches sont applicables (cas des problèmes d'ordonnancement), pour d'autres, uniquement la récurrence avant est possible vu la nature de l'évolution du système, comme le cas de gestion des stocks ou de processus industriels où l'horizon est limité à droite et illimité à gauche. En termes de recherche opérationnelle, cette hypothèse se traduit par : en tout état du système, la valeur de la fonction objectif (si elle est additive, ce qui est le cas pour la plupart des problèmes d'optimisation) est la somme de deux termes : l'optimum du passé (ou du futur en cas du *backward dynamic programming*) et celui de la décision à prendre en passant de l'état actuel de la phase k à un état de la phase k+1 (k-1 s'il s'agit de l'approche arrière).

Il faut noter aussi que le nombre de sous problèmes (ou de phases) peut être fini ou infini selon la nature du problème et la fonction objectif à optimiser, c'est

également le nombre de décisions prises dans une séquence formant une politique optimale. En cas où ce nombre est fini, il est dit : "horizon du problème"[ROB79]. A noter enfin, que la programmation dynamique est une méthode exacte avec des modalités d'application bien spécifiques et, bien entendu, des avantages et inconvénients dont on reviendra minutieusement dans ce mémoire. Elle n'a pas un schéma algorithmique précis mais une démarche générale dont le gros du travail à faire est primo : la décomposition du problème, et secundo : l'établissement de l'équation récursive qui permet d'exprimer la fonction objectif du sous problème d'ordre k en fonction de celle du sous problème d'ordre k-1 (ou k+1).

D'après tous les documents que nous avons pu consulter, nous nous permettons de soulever que cette méthode existe depuis bien longtemps, elle est praticable sous ses formes mathématiques et économiques, mais elle n'est pas encore exploitée pour tous les cas, car il n'y a pas d'algorithmes spécifiques à informatiser. C'est dans le but d'enrichir le domaine de la recherche opérationnelle, qu'on tente par le biais de ce travail, à mettre en évidence les outils que met la programmation dynamique à notre disposition pour remédier à certains handicaps des problèmes d'optimisation. En effet, on se propose, en premier lieu, de présenter cette méthode avec tous les détails, puis en second lieu, l'appliquer à un problème d'ordonnancement : "minimisation de la somme des coûts des avances et des retards avec date échue commune sur une seule machine". On estime que cette application est absolument pertinente vu l'importance croissante ces dernières années de l'ordonnancement qui ne cesse de s'étendre sur pas mal de domaines économiques, industriels, informatiques,… Enfin, pour aborder des problèmes de grandes tailles, on tentera de concevoir une approche fondée sur la même idée en tronquant l'espace des états générés par la méthode de la programmation dynamique en procédant par la relaxation lagrangienne. L'objectif primordial est donc de présenter une alternative avec un coût minimum permettant de donner une solution approchée dans un temps raisonnable qui s'avère très utile pour certaines situations des problèmes d'optimisation.

CHAPITRE PREMIER

THEORIE DE LA PROGRAMMATION DYNAMIQUE

1. Introduction

L'objet de ce chapitre est de présenter en toute généralité la théorie de la programmation dynamique et de donner une formulation abstraite et générale de cette technique en évoquant tous ses éléments moteurs. On tentera alors de faire comprendre, en premier lieu, la décomposition des problèmes et les équations récursives permettant d'évaluer la solution optimale, puis, en second lieu, définir certains termes ayant trait au sujet à l'aide d'exemples illustratifs. En fin, on vise à mettre en évidence l'utilité et l'apport de cette technique dans le domaine de la recherche opérationnelle.

2. Historique

En dépit des travaux de Pierre Massé en 1944 pour planifier les investissements d'électricité de France [SAK84], qui avait pourtant, une idée de ce genre, la programmation dynamique est considérée comme le fruit des recherches menées dans les années 50 par le mathématicien américain Richard Bellman, chercheur à la Rand Corporation, pour résoudre certains problèmes d'affectation et d'optimisation [TIS05]. C'est grâce à son fameux théorème d'optimalité que la programmation dynamique a vu le jour en 1957 et c'est lui qui l'a baptisée ainsi [SAK84]. En fait, les travaux de Bellman sont intervenus peu après que la découverte de l'algorithme du simplexe par G. B. Dantzig en 1947 ait donné un grand élan à la programmation linéaire [SAK84]. C'est certainement pour des raisons de mode que la programmation dynamique a été dénommée ainsi. L'adjectif « *dynamique* », outre qu'il offre une image avantageuse de cette technique, s'explique par le fait qu'il s'agit d'une méthode d'optimisation séquentielle. Pour certains spécialistes en la matière, le terme « optimisation

récursive » eût été plus approprié car il résume excellemment l'idée de l'approche. Pour d'autres, le terme « programmation dynamique » peut paraître un peu étrange et est maintenant parfois utilisé dans un autre sens et que Bellman l'a choisi dans un souci de communication : son supérieur ne supportait ni le mot « recherches » ni le mot « mathématiques », il lui a semblé que l'utilisation du terme « programmation dynamique » donnait à son travail une apparence qui plaisait à son supérieur [TIS05]. En 1962, Held et Karp [HEL62] suivirent les pas de Bellman et appliquèrent l'idée de la programmation dynamique à certains problèmes d'ordonnancement à une machine. Par la suite, White, en 1969, élabora une formulation générale de cette technique en étendant son application à plusieurs domaines [FRE82].

3. Généralités

Le théorème d'optimalité de Bellman, considéré comme le point de départ de la programmation dynamique, est si simple qu'il paraît presque trivial lorsqu'on l'a bien compris. Son importance et l'efficacité des méthodes d'optimisation séquentielles auxquelles il a donné naissance s'accentuent au fur et à mesure que l'on s'aperçoit que la vraie nature de nombreux problèmes est séquentielle : ils peuvent être décomposés en phases, chacune d'elles ne dépendant que de ses voisines les plus proches, et dans les cas favorables, seulement de l'antérieure ou de la postérieure. Autrement dit, Si la fonction objectif à optimiser (appelée aussi *critère*) satisfait cette hypothèse, alors le problème est décomposable en phases ou *périodes*. A chacune d'elles correspond une solution dite *sous-politique*. Il s'agit donc d'un système qui évolue dans le temps d'un état initial e_i jusqu'à un état final e_f, en transitant par des états intermédiaires :

e_{11}, e_{12},… à la phase 1

e_{21}, e_{22},… à la phase 2

………………………

e_{k1}, e_{k2},… à la phase k

…………..……………

$e_{(n-1)1}$, $e_{(n-1)2}$,… à la phase n-1

Le nombre d'états diffère d'une phase à une autre selon le problème en question. Cette décomposition est dans la plupart des cas naturelle ou temporelle, c'est-à-dire que le temps est effectivement une variable de l'évolution du système, par contre, dans d'autres, elle est artificielle, c'est-à-dire que le temps n'intervient pas explicitement dans l'évolution du système. La transition d'un état de l'étape k à un état de l'étape k+1 est appelée *décision*.

A chaque décision u correspond une valeur de la fonction objectif qui représente un coût à minimiser ou un profit à maximiser (selon la fonction objectif). (fig. 1.1).

Très grossièrement, c'est cette décomposition en sous-problèmes qui permet de la grande économie de calcul que procure la programmation dynamique.

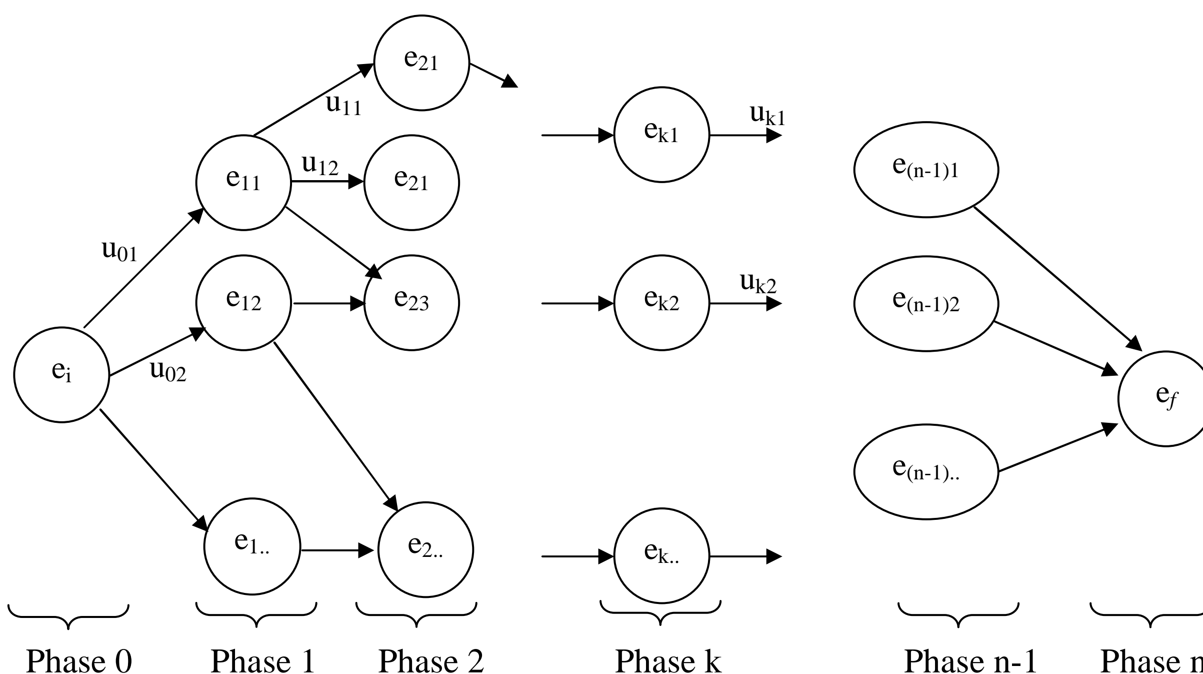

Fig. 1.1. Schéma général de la décomposition d'un problème d'optimisation.

La résolution du problème consiste alors à trouver une séquence de décisions $(u_0,u_1,u_2,\ldots,u_n)$ dite *politique optimale* réalisant la valeur optimale de la fonction objectif. Si la valeur de cette dernière, en se trouvant à un donné, ne dépend du passé qu'à travers son état prédécesseur, cela veut dire que si la prise d'une décision à une étape donnée ne remet pas en cause les décisions des étapes antérieures, alors la

décomposition du problème est possible. Cette condition est appelée *hypothèse de Markov* [CAR88]. Cette hypothèse, qui est généralement satisfaite pour les fonctions additives, peut l'être également pour certaines fonctions multiplicatives ou autres.

3.1. Processus de décisions séquentiel

Si la fonction objectif satisfait l'hypothèse de Markov, on peut schématiser cette décomposition par un modèle mathématique : *le processus de décisions séquentiel.*

Un processus de décisions séquentiel est un triplet (X,U,g) où [CAR88] :

- X : un ensemble appelé espace des états ;
- U : un ensemble appelé espace des décisions ;
- g : une application de $X \times U \times N$ dans $X \cup \{\varepsilon\}$;

N : ensemble des entiers naturels ;

ε est l'état « null » ou « vide » et g est la fonction de transition du graphe ainsi formé.

L'ensemble des états de la phase k est noté : x(k) ;

$x(0)= \{e_i\}$ et pour k>0 : $x(k)= \{e \in X : \exists s \in X, u \in U / e=g(s,u,k-1)\}$;

L'ensemble des décisions qui peuvent être prises à la phase k pour l'état e est noté $\Delta(e,k)$; $\Delta(e,k) = \{ u \in U: g(e,u,k) \neq \varepsilon\}$;

Le nombre de phases est appelé *horizon* [CAR88] [ROB79] du processus de décisions séquentiel. Il peut être fini ou infini, selon le problème à traiter. Au cas où l'horizon est fini, le processus de décisions séquentiels est noté par le quadruplet (X,U,g,n). où n est l'horizon.

Le processus de décisions séquentiel est un modèle mathématique par lequel on schématisera les problèmes d'optimisation évolutifs. En fait, il s'agit d'un outil puissant permettant de représenter les problèmes décomposables.

3.2. Sous-politique / Politique

Etant donné un processus de décisions séquentiel (X,U,g,n).

On appelle sous politique d'ordre k pour l'état e de l'ensemble x(n-k) l'arrangement $(e,u_0,u_1,\dots,u_{k-1})$ tel que : $u_0,u_1,u_2,\dots,u_{k-1}$ des décisions :

$u_0 \in \Delta(e,n-k)$

$u_1 \in \Delta(e_1,n-k+1)$ avec $e_1 = g(e,u_0,n-k)$

$u_2 \in \Delta(e_2,n-k+2)$ avec $e_2 = g(e_1,u_1,n-k+1)$

……………………………………..

$u_{k-1} \in \Delta(e_{k-1},k+1)$ avec $e_{k-1} = g(e_{k-2},u_{k-2},k-2)$ [CAR88]

D'une manière littéraire simple, une sous-politique pour l'état e du système est la séquence de décisions menant à l'état final à partir de l'état e.

Une politique est une sous politique d'ordre n [CAR88]. c'est donc une sous politique pour l'état initial e_i. Il faut remarquer ici que le terme « *politique* » en programmation dynamique substitut le terme « *solution* » en programmation linéaire. Ceci reflète l'aspect évolutif du problème et le caractère dynamique des solutions [ROB79].

3.3. Sous politique optimale / Politique optimale

L'objectif primordial des politiques admissibles d'un système est de pouvoir les évaluer et, par conséquent, pouvoir choisir celle qui est qualifiée de meilleure. Ainsi, chaque problème d'optimisation est caractérisé par une fonction à optimiser (minimiser ou maximiser selon les cas) dite « *fonction objectif* » ou encore « *critère* ». En général, il s'agit d'une fonction réelle F de l'ensemble $\mathcal{P}$ des sous politiques dans R^+.

Une sous politique p* d'ordre k est dite optimale si :

$\forall p \in \mathcal{P}_k$: $F(p^*) \leq F(p)$ i.e. : $F(p^*) = \min_{p \in \mathcal{P}_k}\{ F(p)\}$ pour une fonction à minimiser;

$\forall p \in \mathcal{P}_k$: $F(p^*) \geq F(p)$ i.e. : $F(p^*) = \max_{p \in \mathcal{P}_k}\{ F(p)\}$ pour une fonction à maximiser;

En général : $F(p^*) = \mathrm{Opt}_{p \in \mathcal{P}_k}\{ F(p)\}$;

Où P_k est l'ensemble des sous politiques d'ordre k. ($P_k \subset P$).

Une politique optimale est une sous politique optimale d'ordre n.

3.4. Hypothèse de Markov

Soient (X,U,g,n) un processus de décisions séquentiel, F la fonction objectif associée. $(u_0,u_1,\ldots,u_{k-1}, u_k,u_{k+1},\ldots,u_{n-1})$ et $(u_0,u_1,\ldots,u_{k-1}, v_k,v_{k+1},\ldots,v_{n-1})$ deux politiques ayant le même préfixe $(u_0,u_1,\ldots,u_{k-1})$ jusqu'à un état e de la phase k.

Si $F(e, u_k,u_{k+1},\ldots,u_{n-1})$ est meilleure que $F(e, v_k,v_{k+1},\ldots,v_{n-1})$

alors $F(u_0,u_1,\ldots,u_{k-1}, u_k,u_{k+1},\ldots,u_{n-1})$ est meilleure que $F(u_0,u_1,\ldots,u_{k-1}, v_k,v_{k+1},\ldots,v_{n-1})$.

Le terme « est meilleure que » désigne ici :

- « ≤ » dans le cas de minimisation ;
- « ≥ » dans le cas de maximisation ;

Plus concrètement, cette propriété implique que si deux politiques u et v ont le même préfixe d'ordre k (mêmes k premières décisions) et si à partir de l'état résultant e de ces k premières décisions, la sous politique $(e, u_k,u_{k+1},\ldots,u_{n-1})$ d'ordre n-k pour l'état e est meilleure que la sous politique $(e, v_k,v_{k+1},\ldots,v_{n-1})$ d'ordre n-k pour l'état e alors u est meilleure que v.(fig. 1.2.).

La quasi-totalité des fonctions objectifs satisfait cette propriété dite hypothèse de Markov [CAR88] et cette hypothèse qui autorise la décomposition du problème et, par la suite, l'application de programmation dynamique.

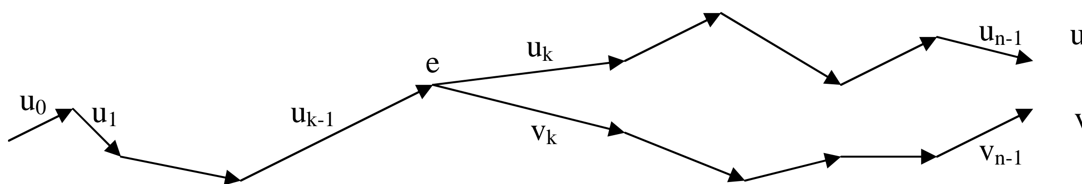

Fig. 1.2. Deux politiques ayant même préfixe illustrant la propriété de Markov.

Bien qu'elle parait naturelle, cette hypothèse n'est pas justifiée pour tous les problèmes. Dans un graphe, par exemple, lorsqu'on cherche le plus court chemin entre deux points : si le chemin le plus court entre A et B passe par C, alors le tronçon de A à C est le chemin le plus court entre A et C, i. e. l'hypothèse est tout à fait satisfaite.

Par contre, elle ne l'est pas si on cherche le plus long chemin sans boucle entre deux points. En fait, si le chemin le plus long entre A et B passe par C, alors le tronçon de A à C n'est pas forcément le chemin le plus long entre A et C.

3.5. Principe d'optimalité / Théorème d'optimalité

En bref, Bellman exprime ceci [ROB79]:

« *Toute politique optimale ne peut être formée que de sous politiques optimales.* »

Ceci s'exprime aussi par :

« *Dans une séquence optimale de décisions, quelque soit la première décision prise, les décisions subséquentes forment une sous-séquence optimale, compte tenu des résultats de la première décision.*» [SAK84]

En d'autres termes [ROB79] :

« *Tout chemin optimal est constitué de portions de chemins elles mêmes optimales.*»

Pour donner une démonstration par l'absurde simple et brève, on peut s'exprimer ainsi :

S'il n'en était pas ainsi pour une portion quelconque, celle-ci pourrait être remplacée par une autre meilleure, et, par conséquent, le chemin ne serait pas optimal, contrairement à l'hypothèse.

Théorème

Soit (X,U,g,n) un processus de décisions séquentiel associé à une fonction objectif satisfaisant l'hypothèse de Markov.

« Si la politique $(u_0,u_1,\ldots,u_{k-1}, u_k,u_{k+1},\ldots,u_{n-1})$ est optimale, alors toute sous-politique $(e, u_k,u_{k+1},\ldots,u_{n-1})$ est optimale. » [CAR88]

Preuve

Supposons que la politique $(u_0,u_1,\ldots,u_{k-1}, u_k,u_{k+1},\ldots,u_{n-1})$ est optimale, et que la sous-politique $(e, u_k,u_{k+1},\ldots,u_{n-1})$ d'ordre n-k pour l'état e ne l'est pas. i. e. : il existe

une autre sous-politique $(e, v_k,v_{k+1},\dots,v_{n-1})$ d'ordre n-k pour l'état e qui est strictement meilleure.

D'où : la politique $(u_0,u_1,\dots,u_{k-1}, v_k,v_{k+1},\dots,v_{n-1})$ est strictement meilleure que la politique $(u_0,u_1,\dots,u_{k-1}, u_k,u_{k+1},\dots,u_{n-1})$.

Par conséquent, la politique $(u_0,u_1,\dots,u_{k-1}, u_k,u_{k+1},\dots,u_{n-1})$ n'est pas optimale. Ce qui contredit l'hypothèse. Ceci entraîne que la sous-politique $(e, v_k,v_{k+1},\dots,v_{n-1})$ n'existe pas. Or la sous-politique $(e, u_k,u_{k+1},\dots,u_{n-1})$ est optimale. Ce principe paraît à première vue simple et évident, mais une étude approfondie fait apparaître qu'il s'agit d'un outil puissant et d'une idée subtile. Dans le cadre général, ce principe est appelé « *principe d'optimalité* », dans le cadre d'un horizon fini, ce principe devient « *théorème d'optimalité.* »

3.6. Récurrence avant. Récurrence arrière

Dans la résolution de certains problèmes, quand on procède par la méthode de la programmation dynamique, deux approches sont applicables [FRE82] :

Une approche descendante dite « *récurrence avant* » (forward dynamic programming) dans laquelle on procède comme suit : en partant de l'état initial du système, on résout les sous-problèmes d'ordres 1,2,…,n-1,n ; on déterminera ainsi les sous-politiques optimales respectives d'ordres 1,2,…,n-1,n.
Ainsi, on choisira la première décision de la politique optimale, puis la seconde, et ainsi de suite jusqu'à la n-ième décision.

Une approche ascendante dite « *récurrence arrière* » (backward dynamic programming) dans laquelle on procède comme suit : en partant de l'état final du système, on résout les sous-problèmes d'ordres 1,2,…,n-1,n ; on déterminera ainsi les sous-politiques optimales respectives d'ordres 1,2,…,n-1,n ; Ainsi, on choisira la dernière décision de la politique optimale, puis l'avant dernière, et ainsi de suite jusqu'à la première décision.

La première approche est toujours applicable et plus simple à modéliser, tandis que la seconde n'est pas applicable dans les problèmes en horizon infini et plus complexe dans les cas stochastiques. On peut procéder indifféremment par les deux

approches dans les problèmes « non ordonnés », néanmoins dans les problèmes ordonnés ou partiellement ordonnés, seule la première approche est faisable. Dans le cadre de ces conditions, on peut formuler les deux algorithmes généraux suivants dans lesquels on tentera de donner une démarche abstraite à appliquer dans la plupart des cas :

***Algorithme AVANT**((X,U,g,n),F,p*,Opt)*

***Entrées :** un PDS (X,U,g,n), une fonction objectif F*

***Variables :** Min : un réel, u : une décision, e : un état, p : une sous politique.*

***Sorties :** une politique optimale p* =(u_0,u_1,….,u_{n-1}), une solution optimale Opt=F(p*).*

Dédut

p*=∅

Opt=0

Pour k=0 à n-1

Pour tout e∈ X(k)

Min=∞

Pour tout u∈ Δ(e,k)

p=p*∪{u}

Si F(p) < Min alors

Min=F(p)

u_k=u

fin si

Fin pour

Fin pour

p*= p*∪{u_k}

Opt= Opt+ Min

Fin pour

Fin

Remarque

Comme on peut le remarquer, cet algorithme s'applique aux problèmes de minimisation. Dans un problème de maximisation, l'initialisation « Min = ∞ » devrait être remplacée par : « Max=0 » et la comparaison : « Si F(p) < Min » deviendrait : « Si F(p) > Max ».

Algorithme ARRIERE((X,U,g,n),F , p*,Opt)

***Entrées :* un PDS (X,U,g,n), une fonction objectif F**

***Variables :* Min : un réel, u : une décision, e : un état, p : une sous politique.**

***Sorties :* une politique optimale p* =(u_0,u_1,….,u_{n-1}), une solution optimale Opt=F(p*).**

Dédut

p*=∅

Opt=0

Pour k=n-1 à 1 pas -1

Pour tout e∈ X(n-k)

Min=∞

Pour tout u∈ Δ(e,n-k)

p=p*∪{u}

Si F(p) < Min alors

Min=F(p)

u_k=u

fin si

Fin pour

Fin pour

p*= p*∪{u_k}

Opt= Opt+ Min

Fin pour

Fin

Ces algorithmes constituent un modèle général pour une certaine classe de problèmes, pour sa mise au point, on aura recours aux spécifications du problème concerné (par exemple Opt peut être un entier ou un réel selon les problèmes), et aux structures de données adéquates pour représenter les décisions, les états et les sous politiques.

Il est clair que ces deux algorithmes sont itératifs. Une forme récursive pourrait également être tentée, elle aurait été plus appropriée, on l'a évitée ici dans un souci de simplification, on l'utilisera au moment de l'implémentation des algorithmes. La forme récursive est beaucoup plus simple à programmer, mais en terme de complexité en temps, elle est plus lente que la forme itérative.

4. Programmation dynamique en horizon illimité

Dans un problème d'optimisation, même en avenir déterminé, l'horizon peut être limité ou, au contraire, illimité. Dans le cas d'un horizon fini, on dit que l'horizon est fermé dans les deux sens, à gauche et à droite ; dans le cas d'un horizon infini, on distingue :

- Un horizon ouvert dans les deux sens ;
- Un horizon ouvert à gauche et fermé à droite ;
- Un horizon fermé à gauche et ouvert à droite.

Le cas le plus important, économiquement parlant, est évidemment, ce dernier, où : $0 \leq n \leq +\infty$, ce qui revient à faire tendre le nombre n de phases vers l'infini [ROB79].

Dans ce cas, une politique est, bien entendu, une suite infinie de décisions, et la valeur de la fonction objectif correspondante est alors la somme d'un nombre infini de termes (pour les fonctions additives). Il est évident que c'est rare que cette fonction converge (par exemple, la somme d'un nombre infini de coûts est un coût qui tend, théoriquement, vers l'infini). D'où la difficulté de déterminer la politique optimale en l'absence d'un moyen de comparaison. On a recours à plusieurs remèdes à cette situation :

- Le plus souvent, on a recours à une actualisation des coûts (ou des profits) par période. Ainsi, si p_n est une politique et $w_0, w_1, \ldots, w_{n-1}$ les valeurs respectives des décisions de p_n, on pose [ROB79] :

$$F(p_n) = w_0 + \frac{w_1}{1+\tau} + \frac{w_2}{(1+\tau)^2} + \frac{w_3}{(1+\tau)^3} + \ldots + \frac{w_{n-1}}{(1+\tau)^{n-1}} \quad \text{Et on calcule : } \lim_{n \to +\infty} F(p_n)$$

Où τ est le taux d'actualisation[1]. ($\tau \in R_+^*$). Il est tout à fait clair que cette série converge, ce qui permet de déterminer aisément la politique optimale.

- Si l'actualisation ne se justifie pas, on considère [ROB79] :

$$F(p_n) = w_0 + w_1 + \ldots + w_{n-1} \quad \text{Et on calcule : } \lim_{n \to +\infty} \frac{F(p_n)}{n}$$

Cette valeur moyenne prend particulièrement un sens lorsqu'on a affaires à des sous politiques cycliques, comme le cas de renouvellement d'équipements. Pour certains problèmes, on peut combiner entre valeur moyenne et actualisation.

- Dans certaines situations, on considère que le nombre de phases est fini mais la durée de chaque phase élémentaire est infinitésimale. Ainsi, les fonctions à optimiser deviennent des équations différentielles (au lieu des équations aux différences) [SAK84]. Il s'agit, donc, de calcul de variations. La transposition du principe d'optimalité dans ce contexte est désignée sous le nom de « principe du maximum ».

- Dans le cas général, on prend :

$F(p_n) = \sum_{i=0}^{\infty} \alpha_i w_i$; où les coefficients α_i sont tels que la série précédente converge ; par exemple $\alpha_i = \tau^i$ avec $0 < \tau < 1$.

5. Cas stochastique

Dans un processus de décisions séquentiel, si la transition d'un état à un autre est, partiellement ou entièrement soumise au hasard, on dit qu'il s'agit d'un processus aléatoire ou « stochastique »[ROB79]. Comme dans le cas déterministe, à chaque

(1) Le taux d'actualisation est un terme utilisé en économie pour évaluer les politiques. Il s'agit du rapport de la valeur d'une unité du coût (1 D. A. par exemple) d'une période à celle de la période suivante.

décision u_k (transition) est associée une valeur w_k (un coût ou un profit), mais aussi, pour les transitions aléatoires, est associée une probabilité de transition p_k. Dans un tel processus, le terme « politique » est remplacé par le terme « stratégie », car il s'agit d'un ensemble de décisions face à plusieurs situations [ROB79].

Le théorème d'optimalité s'énonce alors : « Toute stratégie optimale ne peut être formée que de sous-stratégies optimales. » La résolution du problème consiste alors à déterminer une stratégie optimale. Si on considère le cas où le processus est partiellement contrôlé par l'homme, et que la décision humaine précède l'intervention du hasard, (évolution de première espèce), alors chaque phase est composée de deux périodes : la première comporte les décisions certaines (décisions humaines), la seconde comporte les décisions incertaines (décisions aléatoires) (fig. 1.3.).

L'hypothèse de Markov demeure là aussi indispensable : l'état dans lequel on se trouve ne dépend du passé qu'à travers ses prédécesseurs immédiats ; en clair : concernant l'évolution future, toute l'information sur le passé est contenu dans l'état actuel [FRE82], et c'est pour cette raison qu'a toujours, dans de telles situations, recours aux chaînes de Markov. Il ne faudra donc pas s'étonner si la programmation dynamique est largement utilisée en conjonction avec les chaînes de Markov.

	Phase 1				Phase k					Phase n-1			
		a_{11}		b_{11}		a_{k1}		b_{k1}			$a_{(n-1)1}$		$b_{(n-1)1}$
		a_{12}		b_{12}		a_{k2}		b_{k2}			$a_{(n-1)2}$		$b_{(n-1)2}$
e_0		a_{13}		b_{13}		a_{k3}		b_{k3}			$a_{(n-1)3}$		$b_{(n-1)3}$
	Décisions humaines		Décisions aléatoires		Décisions humaines		Décisions aléatoires			Décisions humaines		Décisions aléatoires	

Où : a_{ij} : l'état certain j de la phase i et b_{ij} : l'état incertain j de la phase i.

Fig. 1.3. Schéma général d'un processus dynamique aléatoire de première espèce.

Bien entendu, pour un programme dynamique stochastique de seconde espèce l'ordre des décisions humaines et aléatoires serait inversé. Et si l'intervention humaine disparaît, on aura, certes, un autre aspect.

La fonction objectif considérée prend la forme suivante :

$F(p_n) = w_0+z_0p_0+ w_1+z_1p_1+ w_2+z_2p_2+\ldots+ w_{n-1}+z_{n-1}p_{n-1}$

Où :

- ✓ w_k : la valeur de la décision humaine u_k.
- ✓ z_k : la valeur de la décision aléatoire v_k.
- ✓ p_k : la probabilité de la décision aléatoire v_k.

Ainsi, la fonction objectif est la somme de deux termes : un terme représentant la somme des valeurs décisions certaines et un terme représentant l'espérance mathématique de la variable aléatoire définie par les décisions aléatoires. Elle deviendrait une espérance mathématique dans le cas extrême où la décision humaine ne se manifeste pas.

6. Exemples

Dans cette section, on se propose de revenir sur les notions et les termes qu'on a évoqués à travers ce chapitre à l'aide d'exemples illustratifs comportant les divers aspects de l'application de la technique de la programmation dynamique. En effet, on traitera d'abord un premier exemple de gestion de stock, où la décomposition est temporelle, puis un second exemple du sac à dos, où la décomposition est artificielle. Un troisième exemple concernant l'ordonnancement de tâches indépendantes sera traité en détails dans le chapitre trois y compris son implémentation informatique.

6.1. Un exemple de gestion des stocks

Enoncé

Le tableau suivant donne, pour les quatre périodes qui font l'objet de l'étude, les quantités d'un certain bien qu'un marchand aura à revendre, ainsi que les prix d'achat :

Période i	1	2	3	4
Demande d_i	5	4	3	4
Prix p_i	10	20	15	12

Sachant que ce marchand dispose d'une capacité de stockage de 6 unités (le coût de stock est supposé nul), le problème consiste alors à déterminer une politique optimale d'achat, c'est-à-dire qu'on cherche à déterminer les quantités à acheter à chaque période de manière à minimiser le coût total d'achat pour les quatre périodes. On considère de plus que :

i. les achats se font en début de période ;
ii. on doit stocker les ventes de la période en cours ;
iii. on commence et on termine avec un stock nul ;

<u>Résolution du problème</u>

La décomposition de ce problème est temporelle en quatre périodes, elle est explicite dans l'énoncé, de plus, elle est, en quelque sorte, imposée.

Posons : e_k : la quantité restant en stock à la fin de la $k^{ème}$ période ;

u_k : la quantité achetée au début de la $k^{ème}$ période ;

La valeur de e_k décrit parfaitement l'état du système ; e_k est dite « variable d'état ».

La valeur de u_k caractérise la décision prise ; u_k est dite « variable de décision ».

L'état initial est $e_0 = 0$, l'état final est $e_4 = 0$;

Et pour : $1 \leq k \leq 4$ on a : $0 \leq e_k \leq 6 - d_k$; et $e_k = e_{k-1}+u_k-d_k$;

Ainsi,

$\Rightarrow$ à la première période on a deux états: 0 , 1 , ;

$\Rightarrow$ à la deuxième période on a trois états: 0 , 1, 2 ;

$\Rightarrow$ à la troisième période on a quatre états: 0 , 1 , 2 , 3 ;

$\Rightarrow$ à la quatrième période on a trois états: 0 , 1 , 2 ;

Dans ce cadre, une politique est une séquence de 4 décisions (u_0, u_1, u_2, u_3) représentant les quantités achetées aux différentes périodes.

A chaque quadruplet (u_0, u_1, u_2, u_3) est associé une valeur de la fonction objectif représentant le coût total d'achat.

On est donc tenu à déterminer la ou les politiques optimales encourrant un coût total d'achat minimal.

Le processus de décisions séquentiel ainsi défini est schématisé par le réseau suivant qui illustre l'évolution de ce système :

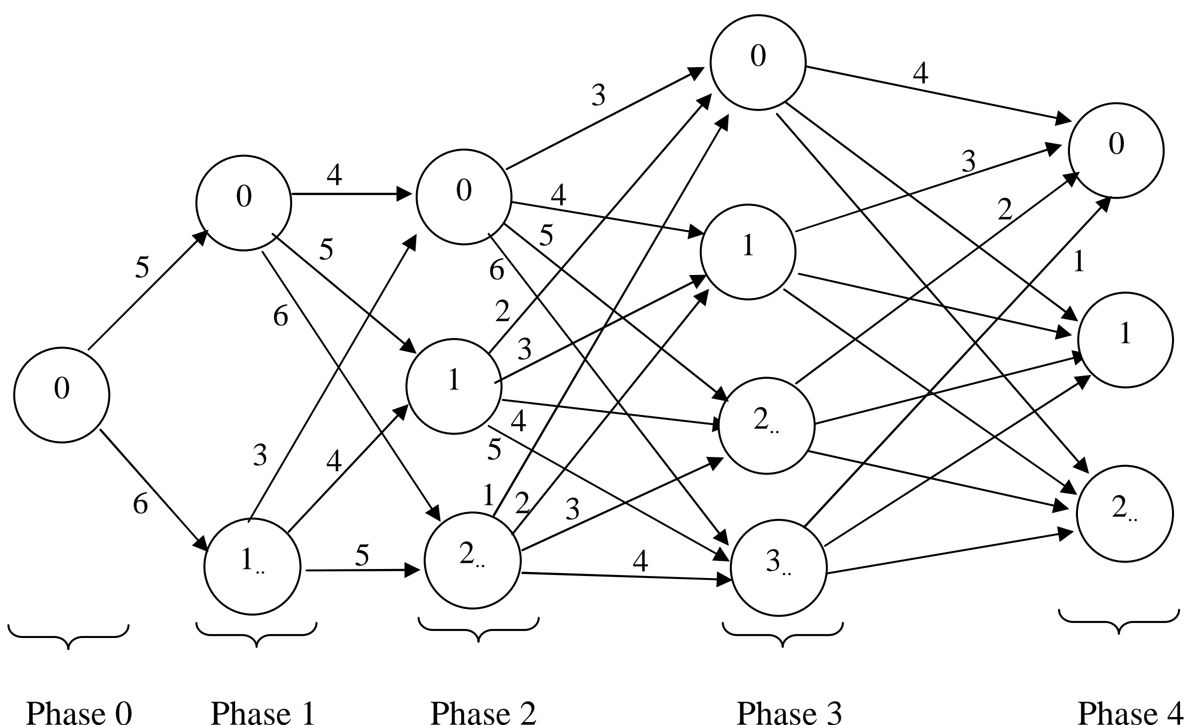

Fig. 1.4. Représentation du problème dynamique de gestion de stock.

A la première période « phase 0 », le stock est nul, la demande est 5, la capacité de stockage étant 6, pour satisfaire cette demande, on peut alors acheter 5 ou 6 unités, ce qui termine la première phase avec un stock de 0 ou 1 unité respectivement. Donc il y a deux décisions possibles : $u_{01} = 5$; $u_{02} = 6$ qui mènent le système aux deux états respectifs $e_{01} = 0$; $e_{02} = 1$.

Le tableau suivant récapitule la démarche adoptée pour les quatre périodes. Les lignes marquées d'un * sont celles qui correspondent aux décisions optimales :

Période k	Demande d_k	Décision u_k	Etat e_k	Coûts cumulés
1	5	5	0	50 *
		6	1	60 *
2	4	4	0	50+80=130
		3		60+60=120 *
		5	1	50+100=150
		4		60+80=140 *
		6	2	50+120=170
		5		60+100=160 *
3	3	3	0	130+45=175 *
		2		150+30=180
		1		170+15=185
		4	1	120+60=185 *
		3		140+45=185 *
		2		160+30=190
		5	2	130+75=205 *
		4		150+60=210
		3		170+45=215
		6	3	120+90=210 *
		5		140+75=215
		4		160+60=220
4	4	4	0	175+48=223
		3		185+36=221 *
		2		205+24=229
		1		210+12=222

Ce tableau illustre clairement la procédure de calcul dans chaque phase du problème qui représente explicitement une période d'achat. En se trouvant dans un état e_k, la prise d'une décision u_k, mène le système économique ainsi conçu, à un état e_{k+1} avec une valeur $f(u_0,u_1,\ldots,u_{k+1})$ de la fonction objectif.

En lisant ce tableau du bas en haut, on déduit que la solution optimale est 221 qui correspond à $u_4 = 3$, ceci implique : $e_3 = 1$, avec $u_3 = 4$ ou $u_3 = 3$.

Ceci entraîne : $e_2 = 0$, avec $u_2 = 3$

D'où : $e_1 = 1$, avec $u_1 = 6$.

La politique optimale recherchée est donc : (6,3,3,3) ou bien (6,3,4,3) dont la valeur est 221.

Remarques

- ✓ Le tableau précédent permet non seulement de résoudre le problème donné comportant les quatre périodes, mais aussi tous ses sous problèmes issus des quatre périodes. Par exemple, la solution optimale de la période 2 est 120 avec la sous politique (6,3).
- ✓ Le tableau comporte 24 lignes, autrement dit, on a résolu 24 sous problèmes pour parvenir à la politique optimale recherchée. En fait, il s'agit des 24 chemins possibles menant de e_0 à e_4. C'est aussi le nombre de quadruplets (a,b,c,d) de décisions qu'on a effectivement explorés. En revanche, dans une méthode énumérative, on aurait exploré 7^4 cas possibles. (on parle ici du nombre d'arrangements de 4 éléments (a,b,c,d) d'un ensemble de 7 éléments {0,1,2,3,4,5,6}). En terme de complexité, l'écart est, évidemment, fatal. (ceci pour, uniquement, quatre périodes. Quel serait cet écart pour un nombre plus grand ?).
- ✓ On peut étendre ce problème à un horizon infini, voire même considérer des demandes d'achats stochastiques en faisant intervenir les probabilités. Mais, là, il faut faire appel aux outils appropriés, et tenir compte, surtout, de l'actualisation des prix.

6.2. Un deuxième exemple : problème du sac-à-dos

Un problème standard du sac-à-dos prend la formulation mathématique suivante : Max $\sum_{j=1}^{n} c_j x_j$

$$\sum_{j=1}^{n} w_j x_j \leq b \quad ; \; x_j \text{ entier positif ; où b est la capacité du sac-à-dos.}$$

On demande de déterminer les valeurs des x_j réalisant une solution maximale tout en satisfaisant la contrainte donnée, c'est-à-dire déterminer le n-uplet $(x_1,x_2,\ldots,x_n)$ de N^n représentant une politique optimale du problème.

Pour cela, on considère b sous problèmes du même type avec un sac-à-dos de taille k , k=1,2,….,b-1,b. Ainsi, ce système n'évolue pas dans le temps, mais c'est la capacité du sac-à-dos qui le fait évoluer.

Un état de l'étape k du système correspond à un n-uplet $(x_1,x_2,\ldots,x_n)$ satisfaisant la contrainte : $\sum_{j=1}^{n} w_j x_j \leq k$.

L'état initial étant le n-uplet (0,0,…,0).

L'état final est le n-uplet $(x_1,x_2,\ldots,x_n)$ satisfaisant cette contrainte pour k=b.

La décision correspond à une augmentation d'une unité de la variable x_j satisfaisant : $w_j \leq k$.

L'équation récursive permettant d'évaluer la fonction objectif F s'écrit alors :

$F(k) = 0$ si $k \pi \min_{j} \{w_j\}$

$F(k) = \max_{j:\, w_j \leq k} \{c_j + F(k - w_j)\}$

Appliquons cette approche à l'exemple suivant :

Max $2x_1 + 3x_2 + 7x_3$

$3x_1 + 4x_2 + 8x_3 \leq 14$

x_1 , x_2 , x_3 entiers positifs.

<u>Résolution du problème</u>

Le tableau suivant résume les quatorze itérations de la procédure adoptée :

Période k	***Décision u_k***	***Etat e_k***	***F(k)***
0	/	/	0
1	/	/	0
2	/	/	0
3	1	(1,0,0)	2
4	1	(1,0,0)	2
	2	(0,1,0)	3 *
5	1	(1,0,0)	2
	2	(0,1,0)	3 *
6	1	(1,0,0)	4 *
	2	(0,1,0)	3
7	1	(1,1,0)	5 *
	2	(1,1,0)	5 *
8	1	(2,0,0)	4
	2	(1,1,0)	5
	3	(0,0,1)	7 *
9	1	(3,0,0)	6
	2	(0,2,0)	6
	3	(0,0,1)	7 *
10	1	(2,1,0)	7 *
	2	(2,1,0)	7 *
	3	(0,0,1)	7 *
11	1	(1,0,1)	9 *
	2	(1,2,0)	8
	3	(1,0,1)	9 *
12	1	(1,0,1)	9
	2	(2,2,0)	10 *
	3	(0,1,1)	10 *
13	1	(1,0,1)	9
	2	(2,2,0)	10 *
	3	(0,1,1)	10 *
14	1	(2,0,1)	11 *
	2	(0,1,1)	10
	3	(2,0,1)	11 *

D'où la solution optimale est 11 qui correspond à la politique optimale (2,0,1).

7. Conclusion

A travers l'étude théorique et les exemples abordés dans ce chapitre, on peut conclure que, malgré que la programmation dynamique soit une méthode exacte, elle reste trop énumérative, surtout lorsqu'il s'agit pratiquement de résoudre des problèmes de grande taille (auquel cas les algorithmes approchés s'avèrent plus adaptés). Elle n'est alors préconisée que dans un cadre bien déterminé. Mais lorsqu'on peut tirer profit de ses aspects fonctionnels, on se rend compte qu'il s'agit d'un véritable outil pour aborder certaines situations des problèmes d'optimisation. Pour cela, nous nous permettons de donner quelques éléments de réponses concernant des points que je juge très intéressants :

Avantages

- ✓ Très flexible. Il est facile d'ajouter des contraintes, d'incorporer des fonctions.
- ✓ Fournit une solution globalement optimale.
- ✓ Fournit les solutions optimales de tous les sous problèmes.
- ✓ Permet de traiter des problèmes stochastiques.
- ✓ S'adapte à des situations où le temps, les états et les décisions varient continûment.

Inconvénients

- ✓ Ne s'applique que sous les hypothèses d'additivité et d'amnésie.
- ✓ Parfois lente et donc inefficace.
- ✓ Ne se prête pas à la réoptimisation.

Programmation dynamique et programmation linéaire

En dépit de la similitude des noms et du fait que certains problèmes peuvent être résolus par les deux méthodes, elles sont tout à fait différentes à la fois par leur esprit et par les problèmes auxquels elles sont destinées. Le tableau suivant récapitule les divergences qui permettent d'identifier aisément les deux méthodes :

	Programmation dynamique	*Programmation linéaire*
Problèmes	Justifiant le principe d'optimalité	Linéaires
Variables	Entières ou discrétisées	Même continues
Contraintes	Peu nombreuses mais quelconques	Nombreuses mais linéaires
Fonction objectif	Quelconque mais séparée	Linéaire

Enfin, on espère que l'application de cette technique à un problème d'ordonnancement qu'on développera au chapitre trois puisse combler d'autres lacunes et éclaircir l'aspect informatique de cette technique.

CHAPITRE II

NOTIONS D'ORDONNANCEMNT

1. Introduction

Dans le but d'appliquer la méthode de la programmation dynamique à un problème d'ordonnancement, il m'est avéré très utile d'aborder dans ce chapitre quelques notions de base relatives aux problèmes d'ordonnancement. Cette présentation de la théorie de l'ordonnancement comportera d'abord les généralités et la terminologie des problèmes d'ordonnancement, puis, on s'intéressera aux différents types de ces derniers ainsi qu'aux différentes méthodes de résolution. Le choix de ce type de problèmes s'émane de son importance et son extension dans les différents systèmes économiques, industriels, informatiques,…d'une part, et de la diversité des méthodes de résolution qui se caractérisent par l'apparition des algorithmes approchés qui font l'objet d'actualité ces dernières années.

2. Généralités

Les problèmes d'ordonnancement sont apparus au départ dans la planification de grands projets. Le but était de gagner du temps sur leur réalisation. De tels projets sont composés de nombreuses étapes appelées également tâches. La résolution de tels problèmes est restée empirique, en utilisant les diagrammes de Gantt, jusqu'au 1958 quand les deux célèbres méthodes de résolutions [CAR88], fondées sur les graphes, ont vu le jour : la méthode américaine CPM (Critical Path Method) avec sa variante PERT (Program Evaluation and Review Technique) et la méthode française MPM (Méthode des Potentiels) de Bernard Roy. Depuis, les problèmes d'ordonnancement ne cessaient de se multiplier et ont envahi ainsi pas mal de domaines. Cette résolution consiste à organiser dans le temps la réalisation de tâches compte tenu de certaines contraintes.

La formulation mathématique des problèmes d'ordonnancement et l'implémentation informatique des algorithmes de résolution étaient l'objectif principal visé par les spécialistes du domaine. Conway et al. [CON67] et Baker [BAK74] ont introduit les formulations de programmation en nombres entiers des problèmes d'ordonnancement [LIA67].

On dit qu'on a affaire à un problème d'ordonnancement, lorsque, en vue de la réalisation d'un objectif quelconque, il faut accomplir de multiples tâches, elles mêmes soumises à un ensemble de contraintes et auxquelles sont attribuées des ressources; il s'agit de trouver un ordre d'exécution de ces tâches.

2.1. Tâches

Une tâche est une entité élémentaire d'un certain travail (une opération ou un ensemble d'opérations).

Exemples :

- ✓ En construction : pose des dalles, peinture, plomberie, …
- ✓ En informatique : exécution d'un programme, impression d'un document,…
- ✓ En transport ferroviaire : occupation d'une portion de voie par un train,…

Donc, il n'y a pas de définition formelle d'une tâche, mais toute considération dépend de la nature du domaine et du problème en question.

Dans un problème d'ordonnancement, l'ensemble des tâches est généralement noté I, le nombre de tâches par n et chaque tâche est notée par i et elle est décrite par les caractéristiques suivantes :

- ✓ r_i : la date de disponibilité de tâche i (date de début au plus tôt) ;
- ✓ t_i : date de début d'exécution de la tâche i ;
- ✓ C_i : date de fin d'exécution de la tâche i ;
- ✓ p_i : durée d'exécution de la tâche i ;
- ✓ d_i : date échue de la tâche i (date de fin au plus tard : deadline) ;
- ✓ w_i : poids de la tâche i si elle est achevée après sa date échue ($C_i > d_i$) ;
- ✓ T_i : retard d'exécution de la tâche i : $T_i = \max\{C_i - d_i, 0\}$ (Tardiness)
- ✓ E_i : avance de la tâche i : $E_i = \max\{d_i - C_i, 0\}$ (Earliness)

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

On peut également, au lieu de w_i, définir une fonction γ_i de R^+ vers R^+ qui, à toute date d'achèvement C_i, fait correspondre un coût unitaire $\gamma\,(C_i)$ si $C_i > d_i$; ou un profit unitaire $\gamma\,(C_i)$ si $C_i \leq d_i$; cette fonction est généralement choisie croissante au sens large.

Exemples

- ✓ $\gamma_i\ (Ci) = w_iC_i$
- ✓ $\gamma_i\ (Ci) = w_iT_i$
- ✓ $\gamma_i\ (Ci) = C_i$
- ✓ $\gamma_i\ (Ci) = \alpha_iE_i + \beta_iT_i$

Relations entre les tâches

Deux tâches sont dites :

- ✓ successives, lorsqu'elles s'exécutent l'une après l'autre ;
- ✓ simultanées, lorsqu'elles doivent s'exécuter simultanément ;
- ✓ convergentes, si elles doivent précéder une tâche donnée ;

Dans ces trois cas, on dit que les tâches sont *dépendantes*, autrement, on dit qu'elles sont *indépendantes*. Pour les tâches simultanées, on parle des problèmes d'ordonnancement sur plusieurs machines.

2.2. Les types de problèmes d'ordonnancement

i) Problème statique / Problème dynamique

On dit qu'on a affaire à un *problème statique*, lorsque l'on connaît à l'avance quelles seront les tâches à exécuter et à partir de quelle date pourra débuter chaque tâche. En revanche, si l'ensemble des tâches évolue avec le temps de façon non déterministe, on dit qu'on a affaire à un *problème dynamique*.

ii) Problème classique / Problème répétitif

Dans un *problème d'ordonnancement classique*, chaque tâche ne s'exécute qu'une seule fois. Par contre, si chaque tâche peut s'exécuter plusieurs fois, on dit qu'il s'agit d'un *problème d'ordonnancement répétitif.*

iii) Problème préemptif / Problème non préemptif

Si les tâches sont exécutées par morceaux avec interruption, le problème est dit *préemptif.* Au contraire, si elles sont exécutées sans interruption, le problème est dit *non préemptif.*

iv) Problème d'atelier

Dans un *problème d'atelier* à m machines, les tâches sont dites travaux (jobs). Chaque travail est un ensemble de m opérations élémentaires, chacune d'elles devant s'exécuter sur une machine différente. On distingue trois types de problèmes d'ateliers :

- ✓ Problème « open-shop » : si les opérations élémentaires sont indépendantes ;
- ✓ Problème « job-shop » : si les opérations élémentaires d'un même travail sont liées par un ordre total, non nécessairement identique pour tous les travaux ;
- ✓ Problème « flow-shop » : si les opérations élémentaires de chaque travail sont liées par un ordre total;

2.3. Les contraintes

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

i) Les contraintes potentielles

Ce sont les contraintes de localisation temporelle. Par exemple : «La tâche i doit précéder la tâche j» ou «La tâche i doit être achevée avant telle date.». Avec ce type de contraintes, le problème est dit "*problème central d'ordonnancement"*.

ii) Les contraintes disjonctives

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

iii) Les contraintes cumulatives

Elles concernent l'évolution dans le temps du volume total des moyens humains ou matériels consacrés à l'exécution des tâches. Par exemple :
« cinq tâches à exécuter sur quatre machines, quelle tâche devrait être retardée ? ».

2.4. Les ressources

Les ressources sont les moyens requis pour l'exécution des tâches. Ce sont de deux types : *les ressources renouvelables et les ressources consommables*.

Une ressource est renouvelable si, après avoir été allouée à une tâche, elle redevient disponible à l'achèvement de cette pour les autres tâches. Comme le cas d'une machine, un processeur, une imprimante,.. ;

Une ressource est consommable si, après avoir été allouée à une tâche, elle n'est plus disponible pour les autres tâches. Comme le cas d'une matière première, de l'argent,…

2.5. Les critères

Le critère d'un problème d'ordonnancement est la fonction économique qu'on vise à optimiser. On l'appelle également dans un autre sens : « fonction objectif ». C'est grâce à l'évaluation de celui-ci qu'on parvient à distinguer entre les différents ordonnancements ou solutions candidates, car il s'agit d'un outil numérique permettant d'apprécier les qualités des solutions. En général, le critère est considéré comme une application F de l'ensemble $\sum$ des permutations des tâches du problème posé vers R^+, qui, à chaque permutation $(j_1,j_2,\ldots,j_n)$, fait associer un réel positif $F(j_1,j_2,\ldots,j_n)$ et qui exprime, en outre, l'utilisation efficace des ressources, le délai global d'exécution et le respect du plus grand nombre de contraintes.

La fonction objectif associée à un problème d'ordonnancement est définie par : $FObj = \underset{\sigma \in \Sigma}{Opt} F(\sigma)$. Les critères les plus utilisés font intervenir la durée totale, le délai d'exécution pondérée, les retards et les coûts des retards et/ou des avances d'exécutions des tâches [CAR88].

Ainsi, la résolution d'un problème d'ordonnancement consiste à déterminer une séquence (un ordre temporel) d'exécution des tâches qui :

- ✓ satisfait toutes les contraintes du problème ;
- ✓ réalise une valeur optimale du critère ;
- ✓ assure une utilisation efficace des ressources ;

<u>Exemples de critères</u>

Durée totale : C_{max}

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

Où : I est l'ensemble des tâches

i est une tâche et C_i la date d'achèvement de la tâche i ; $C_i = t_i + p_i$

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

La fonction objectif s'écrit : $FObj = \min_{\sigma \in \Sigma} \{C_{max}(\sigma)\}$

Retard global : T_{max}

Dans problèmes d'ordonnancement où on vise à minimiser le retard global d'exécution des tâches, le critère est défini par $T_{max}(\sigma) = \max_{i \in I}\{T_i\}$

Où : σ est une permutation des tâches

i est une tâche et T_i le retard d'exécution de la tâche i ; $T_i = \max\{0, C_i - d_i\}$

La fonction objectif s'écrit : $FObj = \min_{\sigma \in \Sigma} \{T_{max}(\sigma)\}$

Somme des retards : $\sum T_i$

La fonction objectif s'écrit : $FObj = \min_{i \in I} \{\sum T_i\}$

Somme pondérée des retards : $\sum w_i T_i$

La fonction objectif s'écrit : $FObj = \min_{i \in I} \{\sum w_i T_i\}$,

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

Nombre de tâches en retard : $\sum u_i$

La fonction objectif s'écrit : $FObj = \min_{i \in I} \{\sum u_i\}$

$u_i = 1$ si la tâche i est achevée en retard (i. e. $C_i > d_i$) ;

$u_i = 0$ sinon .

Nombre pondéré de tâches en retard : $\sum w_i u_i$

$FObj = \min_{i \in I} \{\sum w_i u_i\}$

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

Le retard moyen : $\bar{T} = \sum T_i / n$

$FObj = \min_{i \in I} \{\sum T_i / n\}$ où n est le nombre de tâches.

La somme des coûts des avances et des retards

$FObj = \min_{i \in I} \{\sum (\alpha_i E_i + \beta_i T_i)\}$

- ✓ E_i avance de la tâche i (Earliness) ;
- ✓ T_i retard de la tâche i (Tardiness) ;
- ✓ α_i coût unitaire d'avance associé à la tâche i ;
- ✓ β_i coût unitaire de retard associé à la tâche i ;

Il y a deux variantes pour ce critère : celle avec des dates échues d_i différentes des tâches et une variante où les tâches possèdent une date échue commune d (*Common Due Date*). Ce problème fera l'objet de l'étude qu'on se propose de développer en détails dans le chapitre trois.

2.6. Notation

Afin d'identifier d'une manière synthétique et précise le type du problème d'ordonnancement abordé, on utilise une notation à trois champs $\alpha|\beta|\gamma$ où :

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

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

ƀ: ordonnancement sur une seule machine ;

P : ordonnancement sur plusieurs machines parallèles et identiques ;

q : ordonnancement sur plusieurs machines parallèles et uniformes ;

R : ordonnancement sur plusieurs machines parallèles et différentes ;

O : il s'agit d'un problème « *open shop* » ;

F : il s'agit d'un problème « *flow shop* » ;

J : il s'agit d'un problème « *job shop* » ;

$\alpha_2 \in \{ƀ, k\}$

ƀ: le nombre de machines est variable ;

k : le nombre de machine est k (k entier positif) ;

β : l'ensemble des contraintes ;

γ : le critère à optimiser.

Exemples

- ✓ $1||C_w$: représente les problèmes de minimisation de la somme pondérée des dates de fin des tâches indépendantes sur une seule machine.
- ✓ $F3|r_i|\sum T_i$: représente les problèmes de minimisation de la somme des retards en flow shop à 3 machines avec contrainte des dates de disponibilité.

3. Les méthodes de résolutions

Aborder un problème d'ordonnancement est avant tout pouvoir le résoudre, c'est en fait, déterminer les séquences (les permutations) d'exécution des tâches qui réalisent l'optimalité du critère et satisfont les contraintes du problème avec le moindre coût d'utilisation des ressources. Cela a nécessité l'élaboration de procédés spécifiques à chaque classe de problèmes permettant d'une part, de prouver l'existence des solutions, d'autre part, de déterminer celles qui sont qualifiées de "meilleures solutions". Ces procédés sont, dans la plus part des cas, des modèles d'optimisation programmables : des algorithmes qui peuvent être par la suite implémentés sous formes de programmes, et ont ainsi mené à un véritable progrès dans le domaine de l'ordonnancement avec l'évolution croissante des ordinateurs. La complexité et la diversité des problèmes d'ordonnancement ont imposé au fil des années l'amélioration et l'extension de ces procédés qui constituent de véritables méthodes de résolutions des problèmes d'ordonnancement, qui peuvent acheminer, sous certaines conditions spécifiques, à donner les solutions de ces derniers. Ces méthodes sont, généralement, caractérisées par [ESQ99]:

- ✓ L'existence d'un modèle d'optimisation ;
- ✓ L'exactitude des solutions ;
- ✓ Le mode de résolution (automatique / interactif) ;
- ✓ Le coût de résolution.

Une méthode est dite « exacte » si elle garantit l'optimalité des solutions trouvées. Elle est dite « approchée » ou « heuristique » si on observe qu'elle fournit empiriquement de « bonnes » solutions [ESQ99].

Parmi les méthodes exactes les plus célèbres on cite :

3.1. L'énumération complète (Complete Enumeration)

Elle consiste à explorer toutes les séquences possibles et choisir celles qui sont optimales. Il s'agit d'une énumération exhaustive de tous les cas (les permutations) possibles. Or, pour un problème non préemptif sur une seule machine, on a (n!) permutations possibles, sur m machines, on en a $(n!)^m$. En termes d'analyse

combinatoire, (n!) est le nombre de permutations d'un ensemble contenant n éléments, $(n!)^m$ est le nombre d'arrangements de m éléments de l'ensemble de permutations qui contient (n!) éléments. Il est tout à fait clair que le nombre astronomique de cas possibles justifie nettement le coût considérable de cette méthode, même pour les problèmes de petite taille. Elle est, évidemment, à écarter.

3.2. Méthode par séparation et évaluation (Branch and Bound Method)

Cette méthode consiste à représenter les solutions possibles par un « arbre de recherche» qu'il faut explorer du haut en bas. En se servant d'une borne inférieure (Lawer Bound), de la solution optimale, déterminée au préalable, on évalue, à chaque nœud de l'arbre, la valeur du critère pour éliminer les branches qui dérivent d'un nœud plus « mauvais » que la borne inférieure et n'explorer que celles qui améliorent cette borne inférieure. Cette méthode permet donc de réduire le nombre de cas à explorer. Ce nombre dépend essentiellement de la qualité de la borne inférieure adoptée au départ. L'utilisation de cette méthode ne cesse de s'étendre sur pas mal de domaines. Elle a été utilisée pour la première fois dans la résolution du problème du voyageur de commerce par Eastman [EAS59], dans la programmation en nombres entiers par Land et Doig [LAN60] et dans les problèmes d'ordonnancement par Lomnocki [LOM65] et Ignall et Scharge [IGN65].

3.3. Méthode de la programmation dynamique (Dynamic Programming Method)

La méthode de la programmation dynamique est une méthode générique pour les problèmes d'optimisation, et, en particulier, il a été prouvé que c'est un outil à la fois souple et puissant pour traiter les problèmes d'ordonnancement [ABD87]. Elle a été appliquée pour la première fois aux problèmes d'ordonnancement par Held et Carp [HEL62], et ensuite, par Lawer [LAW64]. Cette méthode consiste à décomposer le problème posé en sous-problèmes (ou phases), puis à établir une équation récursive exprimant la solution du sous-problème d'ordre k en fonction de celle du sous-problème d'ordre (k-1). Ainsi, le dernier sous-problème serait d'ordre n, en l'occurrence, c'est le problème à résoudre. Rappelons qu'on a minutieusement

présenté cette méthode dans le chapitre premier, c'est pourquoi on se contente ici de donner un exemple d'un problème d'ordonnancement de tâches sur une seule machine.

Exemple

Enoncé (problème : $1||\overline{T}$)

n tâches indépendantes doivent être exécutées sans interruption chacune une seule fois sur une seule machine et toutes disponibles à la date 0. Etant données les durées d'exécution p_i et les dates échues d_i des tâches, on demande de déterminer une solution optimale minimisant le retard moyen d'exécution des tâches.

Résolution

La fonction objectif s'écrit : $FObj = \min_{i\in I} \{\sum g_i(C_i)\}$; $I= \{1,2,\ldots,n\}$

$g_i(C_i) = T_i / n$; $T_i = \max \{C_i-d_i,0\}$

La formulation dynamique du problème consiste à considérer un sous ensemble S de I des tâches déjà ordonnancées et, en partant de $S=\emptyset$, on résout récursivement les sous-problèmes d'ordres 1, 2,… , n relatifs aux sous ensembles comprenant respectivement 1,2,…,n tâches, par l'application de l'une des équations récursives suivantes:

Dans le cas de la récurrence avant :

$$F^*(\emptyset) = 0 \; ; \quad F^*(S) = \min_{i\in S}\{ F^*(S-\{i\}) + g_i(p_S) \} \quad \text{si} \quad S \neq \emptyset;$$

La solution du problème est $F^*(I)$

Dans le cas de la récurrence arrière :

$$F^*(I) = 0 \; ; \quad F^*(S) = \min_{i\in S}\{ F^*(S+\{i\}) + g_i(p_S + p_i) \} \text{ si } S \neq I;$$

La solution du problème est $F^*(\emptyset)$; où $F^*(S)$ est la solution optimale du sous-problème associé au sous ensemble S. et $p_S = \sum_{i\in S} p_i$

On parvient ainsi, en dernière phase, à la solution optimale du problème. Le nombre de cas à explorer ainsi est 2^n. Or, lorsque n croît, $2^n << n!$, il est clair que cette méthode est nettement meilleure que l'énumération complète.

Exemple numérique

i	1	2	3	4	5
p_i	2	8	9	1	3
d_i	13	14	19	19	17

Pour n = 5 et les données générées aléatoirement du tableau ci-dessus, et en implémentant un petit programme écrit en visuel basic 6 traduisant la formulation dynamique sus détaillée sur un PC, on a trouvé F* = 0,8 avec la séquence (5,4,1,2,3) en récurrence avant et (1,2,4,5,3) en récurrence arrière. Ces deux séquences optimales prouvent qu'un tel problème peut avoir plusieurs séquences ou politiques optimales mais elles réalisent la même valeur du critère : la solution optimale du problème.

Quant à la résolution manuelle de ce problème, on peut procéder exactement de la même manière que pour le problème de gestion de stock ou celui du sac-à-dos évoqués dans le chapitre précédent. Une phase k (ou un sous-problème d'ordre k, $k\in[0,5]$) correspond à un sous ensemble S de I des tâches déjà ordonnancées. D'où : à la phase k, il y a C_5^k états accessibles S pour chacune d'elles if faut évaluer la solution optimale F*(S). Chaque tâche i du complément I-S de S est une décision qui peut être prise à la phase k+1. Ainsi :

- ✓ l'état initial est ∅ avec $F^*(\emptyset)=0$;
- ✓ à la phase 1 : on a 5 états possibles {1},{2},{3},{4},{5} avec $F^*(\{i\}) = g_i(p_i)$;
- ✓ à la phase 2 : on a 10 états possibles {1,2},{1,3},{1,4},{1,5},{2,3},…
  avec par exemple $F^*\{1,2\}= \min \{F^*\{1\} + g_2(p_{\{1,2\}}), F^*\{2\} + g_1(p_{\{1,2\}})\}$
- ✓ à la phase 3 : on a aussi 10 états possibles {1,2,3},{1,2,4},{1,2,5},{1,3,4},…
  avec par exemple :
  $F^*\{1,2,3\}=\min\{F^*\{1,2\}+g_3(p_{\{1,2,3\}}),F^*\{1,3\}+g_2(p_{\{1,2,3\}}),F^*\{2,3\}+ g_1(p_{\{1,2,3\}})\}$

- ✓ l'état final est I ;
- ✓ chaque permutation de S est une sous-politique d'ordre k ;

Une politique optimale du problème considéré est évidemment une permutation de I réalisant le minimum de la fonction objectif : $FObj = \min_{i \in I} \{\sum T_i / 5\}$

Rappelons encore une fois que :

$$g_i\{p_i\} = \max\{p_i - d_i, 0\} \quad ; \quad g_i\{p_s\} = \max\{p_S - d_i, 0\} / 5 \quad ; \quad p_S = \sum_{i \in S} p_i$$

Ce procédé peut être détaillé à l'aide d'une approche par tabulation.

Remarque

En dépit de l'évolution spectaculaire des PC au fil des années, l'inconvénient impeccable des méthodes exactes demeure leur coût en termes de temps d'exécution et d'espace mémoire requis pour leur résolution, surtout pour des problèmes de taille plus élevée, où elles deviennent très coûteuses, lentes et voire même pratiquement impossibles à partir d'un seuil donné (théoriquement, il y a des problèmes d'ordonnancement qui nécessitent des siècles pour être résolus par une méthode exacte!). Les méthodes exactes ne sont donc préconisées que pour des problèmes de taille assez faible. Cela ne néglige guère l'utilité des méthodes exactes, mais, au contraire, il accentue leur importance du moment que l'on s'aperçoit qu'il s'agit de la « jauge » des méthodes approchées qu'on ne peut jamais s'en passer.

3.4. Méthodes approchées ou heuristiques (Heuristic Methods)

On appelle méthode approchée, toute méthode construisant une solution dont on ne peut garantir l'optimalité [CAR88], mais où on peut évaluer l'écart par rapport à l'optimum et qui permet d'apprécier empiriquement la qualité de la solution [CAR88]. On ne peut pas donner une règle globale où on ait recours aux méthodes approchées, cependant, d'une manière générale, on peut récapituler que les méthodes approchées s'avèrent très utiles et révèlent leur importance lorsque :

- ✓ on a affaire à un problème de taille relativement importante ;
- ✓ les méthodes exactes sont pratiquement inapplicables ou coûteuses ;

- ✓ on désire résoudre le problème dans un temps raisonnable ;
- ✓ l'optimalité de la solution n'est pas primordiale ;
- ✓ une méthode exacte nécessite une solution approchée (telle que la méthode par séparation et évaluation pour le calcul de la borne inférieure).

Ces méthodes ne sont pas efficaces pour toutes les données, c'est pourquoi, il est souvent indispensable de construire plusieurs solutions, et ce en faisant introduire le hasard ou améliorer les solutions par des techniques de voisinage. Elles sont connues actuellement sous plusieurs variantes : méthode de voisinage, la relaxation lagrangienne, le recuit simulé, …

Une autre façon d'étudier l'efficacité d'une méthode heuristique consiste à examiner son comportement au mauvais cas [MAL97]. Les premiers résultats concernant la performance des heuristiques dans le plus mauvais cas sont dus à Graham [GRA66] [GRA69].

3.5. Méthodes métaheuristiques (Metaheuristic Methods)

Rappelons qu'une heuristique constitue une méthode approchée de résolution d'un certain type bien déterminé de problèmes. Elle est, en clair, spécifique à un problème donné. Cependant, une métaheuristique est une méthode générale, une démarche générique, applicable à tout type de problèmes d'optimisation combinatoire. Elle peut être projetée sur n'importe quel problème et souvent inspirée d'un phénomène naturel comme :

- ✓ *le recuit simulé* inspiré de la métallurgie et la thermodynamique ;
- ✓ *les colonies* de fourmis inspirées des colonies de fourmis
- ✓ *les algorithmes génétiques* inspirés de la génétique.

Il s'agit, en fait, de toute une philosophie abstraite qui constitue un recueil référentiel pour générer des heuristiques traitant certaines problématiques de la recherche opérationnelle et l'intelligence artificielle ce qui a conduit à une avancée importante pour la résolution de nombreux problèmes. C'est une branche très récente de ces domaines qui a vu le jour ces dernières décennies pour faire face à certaines situations.

Pour terminer avec ce volet, on se contente de reprendre une définition de Osman I. (AUB) traduite de l'anglais ci-dessous :

« Une métaheuristique est un processus itératif qui gère une heuristique en combinant intelligemment différents concepts pour explorer et exploiter l'espace de recherches en utilisant des stratégies d'approximation de structurer les informations dans un ordre permettant de trouver efficacement des solutions approchées .»[Osm95]

Signalons, enfin, que dans le cadre de la résolution du problème sujet de ce mémoire, on présentera dans le chapitre trois une première heuristique pour évaluer une borne supérieure de la solution optimale dite SEA (Sequential Exchange Approach) et une deuxième heuristique dite relaxation de l'espace des états pour évaluer un borne inférieure de la solution optimale. On tentera alors de "rapprocher" ces deux bornes pour améliorer la solution approchée. Cette deuxième heuristique est inspirée de la relaxation lagrangienne qui peut être considérée comme une métaheuristique qui consiste à relaxer une partie des contraintes et à les intégrer dans la fonction objectif à l'aide des muliplicateurs de Lagrange.

4. Complexité des algorithmes de résolution

En général, on mesure l'efficacité d'un algorithme par une expression mathématique C(n) qui exprime le nombre d'opérations élémentaires indispensables à l'exécution de l'algorithme en fonction de la taille n des données en entrée tout en considérant le pire des cas [BRU01]. C'est le nombre maximum d'étapes de calcul –en fonction de n- nécessaires pour aboutir à une solution optimale [ESQ99]. Cette expression mathématique s'appelle « complexité de l'algorithme ». Si la complexité d'un algorithme est C(n), on dit que cet algorithme est en O(C(n)). On parle ici de la complexité temporelle (celle exprimant le temps d'exécution en fonction de n) de l'algorithme. On aurait pu donner une définition similaire de la complexité spatiale (celle exprimant l'espace mémoire requis pour l'exécution en fonction de n) de l'algorithme.

Exemples

- ✓ Si la complexité d'algorithme résolvant un problème de taille n est $3n^2+4n+5$, on dit que cet algorithme est en $O(3n^2+4n+5)$, évidemment cette complexité est polynomiale, l'algorithme est donc dit « *polynomial* »
- ✓ Si la complexité d'un algorithme résolvant un problème de taille n est 3^n, on dit que cet algorithme est en $O(3^n)$, évidemment cette complexité est exponentielle, l'algorithme est donc dit « *exponentiel*»

Pour évaluer la complexité d'un algorithme, on associe à chaque problème d'optimisation, un problème de décision dont la réponse est oui ou non. Ainsi, à un problème d'ordonnancement, on associe un problème de décision concernant l'existence d'une solution inférieure ou égale à une valeur donnée. La complexité d'un problème d'optimisation est celle du problème de décision associé.

Remarque

Notons qu'il ne faut pas se soucier des détails de la notion d'«opération élémentaire» évoquée dans la définition ci-dessus. En fait, qu'il s'agisse d'une addition, d'une affectation, d'un test ou autre, ce qui importe n'est pas laquelle des opérations est plus vite ou bien quel langage, quel compilateur ou quelle machine sont appropriés, mais c'est le comportement général de l'algorithme qui vaut, car, lorsque n croît, c'est le terme en n qui prime.

Exemple

Pour le problème $1||\overline{T}$, la complexité de la méthode énumérative complète est O(n!), celle de la programmation dynamique est $O(n2^n)$ et celle de la relaxation de l'espace des états, qu'on verra dans le chapitre trois, est en O(nT) où : n est le nombre de tâches et $T = \sum p_i$, i=1,2,…,n.

Algorithme efficace

Un algorithme est dit *efficace* ou *polynomial* si sa complexité est bornée par un polynôme ayant la taille des données comme variable [BRU01].

Exemple

Un algorithme de complexité $3n^2$ est efficace.

Afin d'aborder les problèmes d'optimisation, les spécialistes du domaine se sont servis de la notion de la complexité des algorithmes pour dresser une classification de ces problèmes.

La classe NP

Un problème appartient à la classe NP si on peut déterminer sa solution en un temps polynomial. On dit qu'elle regroupe les problèmes faciles (la classe P) et les problèmes difficiles (la classe NP-Complet et les problèmes ouverts).

La classe P

Elle regroupe les problèmes les plus faciles de la classe NP. Ce sont les problèmes pour lesquels on connaît au moins un algorithme polynomial pour les résoudre.

Réduction polynomiale

Soient deux problèmes P1 et P2. On dit que P1 est réduit à P2 si on peut résoudre P1 en utilisant un algorithme de P2 comme sous-routine. Cette réduction est dite polynomiale si, en considérant l'appel à la sous-routine de P2 comme une opération élémentaire, l'algorithme de P1 est polynomial [BRU01].

La classe NP-Complet

Elle regroupe les problèmes les plus difficiles de la classe NP. Ce sont les problèmes tels que n'importe quel problème de la classe NP leur est polynomialement réductible.

La classe NP-Difficile

Elle regroupe les problèmes (pas forcément de la classe NP) tels que n'importe quel problème de la classe NP leur est polynomialement réductible.

5. Conclusion

L'ordonnancement demeure un domaine très vaste en plein développement qui est souvent décrit comme une fonction particulière de décision au sein d'un système de gestion du travail concernant la production de biens, d'ouvrages ou de services. L'objectif qu'on a visé dans ce chapitre n'est pas une présentation exhaustive et minutieuse de ce domaine, mais plutôt, on a tenté de donner un bref aperçu des notions de base de l'ordonnancement dans le but de faciliter leur introduction dans le chapitre trois. En dépit de la complexité et la diversité des problèmes d'ordonnancement, il existe, dans la plupart des temps, des modèles et des méthodes pour aborder ces problèmes. A l'heure actuelle, le contexte dans lequel la fonction ordonnancement s'insère est le siège de changements profonds. C'est la tendance vers les méthodes approchées dont la volonté de substituer les méthodes exactes est de plus en plus attrayante du moment qu'elle remet en cause les acteurs influant sur les coûts, les délais et la qualité des produits de projets. Cependant, face aux situations rencontrées en pratique, le recours aux méthodes exactes demeure incontestable même pour adopter une méthode approchée. Ce sont, en bref, les arguments qui ont motivé le choix du problème et l'adoption de la programmation dynamique et la relaxation lagrangienne qu'on entamera dans le chapitre suivant.

CHAPITRE III

MINIMISATION DE LA SOMME DES COUTS DES AVANCES ET DES RETARDS AVEC DATE ECHUE COMMUNE SUR UNE SEULE MACHINE PROGRAMMATION DYNAMIQUE ET RELAXATION DE L'ESPACE DES ETATS

1. Introduction

L'optimisation combinatoire occupe une place très importante en recherche opérationnelle, en mathématiques discrètes et en informatique. Son importance se justifie d'une part par la grande difficulté des problèmes d'optimisation et d'autre part par de nombreuses applications pratiques pouvant être formulées sous la forme d'un problème d'optimisation combinatoire. Bien que les problèmes d'optimisation combinatoire soient souvent faciles à définir, ils sont généralement difficiles à résoudre. En effet, la plupart de ces problèmes appartiennent à la classe des problèmes NP-hard et ne possèdent pas, à ce jour, de solution algorithmique efficace valable pour toutes les données.

L'objet principal de ce chapitre est la résolution d'un problème d'ordonnancement de tâches indépendantes intitulé : « minimisation de la somme des coûts des avances et des retards avec date échue commune sur une seule machine » par une méthode exacte, la programmation dynamique, en faisant introduire une approche de tabulation pour sauvegarder les solutions optimales des sous-problèmes résolus récursivement afin de réduire le temps d'exécution, ensuite par une méthode approchée fondée sur l'approche SEA (Sequential Exchange Approach), mais qui est spécifique au problème étudié. Cette approche fournira une solution approchée au problème qui servira comme borne supérieure. Enfin, on appliquera la relaxation

lagrangienne pour réduire l'espace des états, ce qui découle à une autre solution approchée du problème, mais cette fois ci, il s'agira d'une borne inférieure. Cette relaxation est également propre à ce problème. Auparavant, dans les problèmes d'ordonnancement relatifs aux délais d'exécution, on ne s'intéressait qu'à minimiser les coûts des retards, récemment, on s'est rendu compte que, dans plusieurs situations rencontrées en pratique, on devrait tenir compte des coûts des avances d'exécution des tâches par rapport à leur(s) date(s) échue(s) (date commune ou dates distinctes). Le choix de ce type de problèmes s'émane essentiellement de cette tendance, qui est en plein développement, pour tenter de contribuer au progrès si rapide de ce domaine surtout avec l'avènement des algorithmes approchés et les métaheuristiques dont l'importance est de plus en plus considérable et ne cessent de s'étendre pour les différents types de problèmes d'ordonnancement, et ce, dans le contexte de faire face à des problèmes « difficiles » avec un niveau raisonnable de ressources.

2. Présentation et historique

Les problèmes d'ordonnancement traitant les avances et les retards sont connus sous plusieurs variantes selon la forme de la fonction objectif et la date échue commune ou des dates échues différentes. Ils ont été abordés ces dernières années par plusieurs chercheurs du domaine. Certains cas de ces problèmes sont faciles et peuvent être résolus par des algorithmes polynomiaux, d'autres, suivant les données, sont NP-complet ou NP-dur. Mais la classification exacte de ces problèmes reste une question ouverte.

Ces problèmes, connus souvent sous la notation E/T : Ealiness / Tardiness en anglais (A/V : Avance / Retard en français), ont fait, dans plusieurs occasions, l'objet de publications internationales, notamment celui qu'on se propose d'aborder, en l'occurrence [LIN06] [FEL03] [IBA94] [HIN05] :

- ✓ Baker et Scudder (1990) : établirent un compte rendu sur les modèles d'ordonnancements des problèmes des coûts de A/V.
- ✓ Ibaraki et Nakaruma (1991) : application de la relaxation de l'espace des états.
- ✓ Hall et al. (1991) : prouve que ce problème est NP-hard.

- ✓ Lee et Kim (1995) : développèrent un algorithme génétique au problème.
- ✓ James (1997) : utilisation de la recherche tabou.
- ✓ Gordon et al. (2002) : étude de l'ordonnancement de tâches avec date échue commune.
- ✓ Feldmann et Biskup (2003) : élaboration de métaheuristiques : single machine scheduling for minimizing earliness and tardiness penalties by meta-heuristic approaches.
- ✓ Hino, Ronconi et Mendes (2005) : proposent une heuristique et une méta-heuristique.
- ✓ Lin, Chou, Ying (2005) : heuristiques de résolutions, brève publication : A sequential exchange approach for minimizing earliness-tardiness penalties of single machine scheduling with a common due date.

Ibaraki et al. [IBA92] a publié en 1992 une étude traitant l'application de la relaxation de l'espace des états pour un problème pareil, cependant, à la différence de celui qu'on se propose d'aborder où les tâches ont une date échue commune, il a pris le cas des dates échues distinctes. Deux autres différences qu'il faut absolument souligner ici:

i) Dans l'heuristique utilisée pour évaluer une borne supérieure de la solution : en effet, Ibaraki, a utilisé l'approche SSDP (Successive Sublimation Dynamic Programming) [IBA92], par contre, pour notre cas, on utilisera une approche SEA (Sequential Exchange Approach) adaptée au problème qu'on détaillera plus loin dans ce chapitre.

ii) Dans la procédure itérative de mise à jour de la borne supérieure, Ibaraki et al. prend l'équation : $UB(k) = [\ LB(k-1) + UB(k-1)\ ] / 2$, il 'agit, en fait, du centre de l'intervalle $[\ LB(k-1)\ ,\ UB(k-1)\ ]$ alors qu'en cette étude, on prendra l'équation : $UB(k) = \mu * LB(k-1) + (1-\mu) * UB(k-1)$ où μ est un paramètre que l'on choisira de l'intervalle $]0\ ,\ 1[$, on étudiera les cas $\mu \in \{0,25\ ;\ 0,50\ ;\ 0,75\}$. Il s'agit du « barycentre » de LB(k-1) et UB(k-1) affectés des coefficients μ , $1-\mu$ respectivement. (On reviendra plus loin dans ce chapitre sur les détails de ces points.)

Quant aux travaux relatifs aux cas généraux des problèmes de type A/V, on les a énumérés dans le tableau 3.1.

Fonction objectif	*Dates échues*	*Références*
$F(S) = \sum \| C_i - d \|$	égales	Sundararaghavan et Ahmed 1984 Bagchi, Sullivan et Chang 1986, Kanet 1981 Baker et Chadowitz 1989, Szware 1989 Hall, Kubiak et Sethi 1989, Emmons 1987
$F(S) = \sum\sum \| C_i - C_j \|$	/	Kanet 1981
$F(S) = \alpha\sum E_i + \beta\sum T_i$	égales	Panwalker, Smith and Serdmann 1982 Bagchi, Sullivan et Chang 1987 Baker and Chadowitz 1989, Emmons 1987
$F(S) = \sum (C_i - d)^2$	égales	Bagchi, Sullivan et Chang 1987 De Ghosh et Wells 1989
$F(S) = \sum (C_i - C)^2$	/	Raghavachan et Vani 1987, Kanet 1981 Eilon and Chow dahury 1977
$F(S) = \sum\sum (C_i - C_j)^2$	/	Kanet 1981
$F(S) = \alpha\sum E_i^2 + \beta\sum T_i^2$	égales	Bagchi, Sullivan and Chang 1987
$F(S) = \sum \alpha_i E_i + \sum \beta_i T_i$	égales	Bagchi 1985, Cheng 1987, Emmons 1987 Bector, Gupta et Gupta 1988, Quaddus 1987 Baker et Skuder 1989, Hall et Posner 1989
$F(S) = \sum \alpha_i E_i + \sum \beta_i T_i$	distinctes	Ahmadi et Baghi 1986 , Davis et Kanet 1987 Fry, Armstrong et Blackstone 1987 Fry et Leong 1987, Yano et Kim 1987 Fry , Leong et Rakes 1987 Chand et Schneeberger 1988 Abderrazak et Potts 1988 Ow et Morton 1988,1989
$F(S) = \sum\alpha_i E_i^2 + \sum\beta_i T_i^2$	distinctes	Gupta et Sen 1983 , Cheng 1984

Tableau 3.1. Recueil des références des problèmes de type A/V [KEN90].

3. Formulation du problème

n tâches indépendantes doivent s'exécuter chacune une et une seule fois sans interruption sur une seule machine. On demande de déterminer une séquence (un ordre d'exécution) des n tâches de telle façon que la somme des coûts des avances et des retards soit minimum.

Notation : n : nombre de tâches.

I : un ensemble de n tâches $I = \{ 1 , 2 , \ldots , n \}$;

d : la date échue commune de toutes les n tâches ;

C_i : date d'achèvement de la tâche i ;

p_i : durée d'exécution de la tâche i ;

E_i : avance de la tâche i : $E_i = \max\{d-C_i , 0\}$ (Earliness)

T_i : retard de la tâche i : $T_i = \max \{C_i-d , 0\}$ (Tardiness)

α_i : coût unitaire d'avance associé à la tâche i ;

β_i : coût unitaire de retard associé à la tâche i ;

h : coefficient de la date échue commune : $d = h * T$ où $T = \sum_{i \in I} p_i$

On prendra : $h \in \{ 0.2 , 0.4 , 0.6 , 0.8 \}$

La fonction objectif à optimiser s'écrit : $FObj = \min_{i \in I} \{\sum (\alpha_i E_i + \beta_i T_i)\}$

4. Résolution par la programmation dynamique

Avant d'entamer ce volet, on doit remarquer que le critère à optimiser est la somme de coûts immédiats. La propriété de Markov est donc vérifiée. Il reste à associer au problème posé un processus de décisions séquentiel dont la conception dépend de la récurrence adoptée, avant ou arrière.

4.1. Récurrence avant

Comme on l'a vue au chapitre premier, cette approche consiste à décomposer le problème posé en sous problèmes (ou en phases) d'ordre 1, 2 , …n , puis établir une équation récursive exprimant la solution du sous-problème d'ordre k en fonction de celle du sous-problème d'ordre k-1. Ainsi, la solution du sous-problème d'ordre n

(dernière phase) est celle recherchée. Chaque phase est un ensemble d'états du système ainsi conçu. Chaque état correspond à un sous-ensemble S de I des tâches déjà ordonnancées. Une décision consiste à choisir la tâche de S qui sera ordonnancée en dernière position dans une permutation de S. D'où : le nombre de phases est 2^n. L'état initial est ∅, l'état final est I.

Si F* est la solution optimale pour l'état S de I, alors :

$$F^*(\emptyset) = 0 \ ; \quad F^*(S) = \min_{i \in S} \{ F^*(S-\{i\}) + g_i(C_S) \} \text{ si } S \neq \emptyset \ ;$$

où : $C_S = \sum_{i \in S} p_i$

et $g_i(C_i) = \alpha_i (d - C_i)$ si $C_i \leq d$; $g_i(C_i) = \beta_i (C_i - d)$ si $C_i > d$;

La solution du problème est, par conséquent, F*(I).

Remarquons que la valeur du critère pour le sous-problème associé à S est la somme de deux termes X et Y : $X = g_i(C_i)$ est le coût de la décision prise i et $Y = F^*(S-\{i\})$ est le coût cumulé du « passé » précédant la décision i; c'est en fait l'application explicite du théorème de Bellman.

Algorithme de la récurrence avant
ALGORITHME DPFwd
Entrées : I, n , p_i , α_i ,β_i pour i = 1,n;
Variables intermédiaires : S un sous-ensemble de I ; min , k , i , X , Y : entiers ;
Sorties: F*(I) ;

```
Début
  F*(∅) = 0
  Pour k = 1 à n
     Pour tout sous-ensemble S contenant k éléments de I
          F*(S) = ∞
        Pour élément i de S
              Cs = ∑pi pour i ∈ S.
              X = gi(Cs)
              Y = F*( S-{i})
              Min= X + Y
             Si min < F*(S) alors F*(S) = min
        Fin pour
     Fin pour
  Fin pour
Fin
```

Enfin, pour pouvoir déterminer la ou les séquences optimales du problème, on doit garder, pour chaque sous-problème résolu, le dernier élément de la séquence qui est la tâche i réalisant le minimum $\min_{i \in S} \{ F^*(S-\{i\}) + g_i(C_i) \}$. Puis, on doit procéder à rebours en montant du bas en haut. Parallèlement à F(S), on construit une table Last(j), $j = 1,2,\ldots, 2^n$. Ainsi, la dernière tâche ordonnancée serait $i_n = Last(2^n)$; et si k est la dernière tâche ordonnancée dans I- $\{i_n\}$, alors $i_{n-1} = k$, c'est-à-dire k est l'avant dernière tâche de la séquence optimale. Et ainsi de suite.

4.2. Récurrence arrière

Dans cette approche on doit établir une équation récursive exprimant la solution du sous problème d'ordre k en fonction du sous problème d'ordre k+1. Ainsi, la solution du sous problème d'ordre 0 (dernière phase) est celle recherchée. Chaque phase est un ensemble d'états du système ainsi conçu. Chaque état correspond à un sous-ensemble S de I des tâches non ordonnancées. Une décision consiste à choisir la tâche de I-S qui sera ordonnancée en première position dans une permutation de S. D'où : le nombre de phases est 2^n. L'état initial est I, l'état final est $\varnothing$.

Si F* est la solution optimale pour l'état S de I, alors :

$$F^*(I) = 0 \ ; \quad F^*(S) = \min_{i \in S} \{ F^*(S+\{i\}) + g_i(p_i) \} \text{ si } S \neq I \ ;$$

$$\text{et} \quad g_i(p_i) = \alpha_i\,(d - p_i) \text{ si } p_i \leq d \ ; \quad g_i(p_i) = \beta_i\,(p_i - d) \text{ si } p_i > d \ ;$$

La solution du problème est, par conséquent, $F^*(\varnothing)$.

Remarquons que la valeur du critère pour le sous problème associé à S est la somme de deux termes X et Y : $X = g_i(p_i)$ est le coût de la décision prise i et $Y = F^*(S+\{i\})$ est le coût cumulé du « futur » succédant la décision i; c'est aussi l'application explicite du théorème de Bellman.

Enfin, pour pouvoir déterminer la ou les séquences optimales du problème, on doit garder, pour chaque sous problème résolu, le premier élément de la séquence qui est la tâche i réalisant le minimum $\min_{i \in S} \{ F^*(S+\{i\}) + g_i(p_i) \}$, puis on procède de la même manière que dans la récurrence avant.

Algorithme de la récurrence arrière
ALGORITHME DPBwd
Entrées : I, n , p_i , α_i ,β_i pour i = 1,n;
Variables intermédiaires : S un sous-ensemble de I ;min , k , i , X , Y : entiers ;
Sorties: F*(∅) ;

```
Début
  F*(I) = 0
  Pour k = n-1  à 0 pas -1
      Pour tout sous-ensemble S contenant k éléments de I
          F*(S)  = ∞
          Pour tout élément i de I-S
              X =  gi(pi)
              Y = F*( S+{i})
              Min=  X + Y
              Si min < F*(S) alors F*(S) = min
          Fin pour
      Fin pour
  Fin pour
Fin
```

Commentaires et Discussions

- ✓ Cet algorithme est, évidemment, rédigé sous forme itérative. Une forme récursive pourrait être également tentée, elle est très simple à programmer mais, en revanche, elle est plus lente vu le nombre exponentiel d'appels récursifs engendrés, ce qui peut saturer la pile des appels.
- ✓ Le nombre de sous-ensembles d'un ensemble contenant n éléments est 2^n. Pour chaque sous-ensemble contenant k éléments, on a une boucle de k itérations. Dans chaque itération, il faut effectuer une addition et une comparaison. Le nombre total d'opérations à effectuer est donc : $\sum_{k=1}^{n} k C_n^K$, soit $n2^{n-1}$ ($k C_n^K = n C_{n-1}^{k-1}$ et $\sum_{k=1}^{n} C_{n-1}^{k-1} = 2^{n-1}$) La complexité de cet algorithme est alors $O(n2^{n-1})$.

  Si les p_i, α_i, β_i sont bornés, la taille d'un énoncé du problème sera une fonction linéaire en n et la complexité de l'algorithme est donc exponentielle.
- ✓ Dans cette méthode, on doit énumérer 2^n états possibles, cela est évidemment coûteux en temps d'exécution et en espace mémoire.

Tests et résultats

Les tableaux suivants donnent les résultats moyens de 10 instances pour chacun des cas : n = 10 , 15 , 20 , 25 et illustrent la complexité exponentielle en temps d'exécutions des deux algorithmes DPFwd et DPBwd.

Les p_i, α_i, β_i sont générés aléatoirement entre 1 et 9.

h = 0,2

n	Solution	Temps	
		DPfwd	DPBwd
10	9232	142 ms	145 ms
15	11307	738 ms	754 ms
20	17187	17,27 s	19,53 s
25	22097	10,77 mn	11,34 mn

h = 0,4

n	Solution	Temps	
		DPfwd	DPBwd
10	8769	134 ms	139 ms
15	10943	712 ms	740 ms
20	17091	17,10 s	18,78 s
25	20597	10,21 mn	11,24 mn

h = 0,6

n	Solution	Temps	
		DPfwd	DPBwd
10	8769	132 ms	135 ms
15	10943	712 ms	725 ms
20	16461	16,78 s	17,80 s
25	20597	10,17 mn	11,07 mn

h = 0,8

n	Solution	Temps	
		DPfwd	DPBwd
10	8769	109 ms	125 ms
15	10943	650 ms	678 ms
20	16461	16,50 s	17,45 s
25	20597	10,02 mn	10,54 mn

D'après ces tableaux, on conclut que la programmation dynamique devient très lente, voire impraticable dès que n atteigne un seuil déterminé selon la machine utilisée. La récurrence avant est sensiblement plus rapide que la récurrence arrière. C'est pourquoi, dans tout ce qui suit, on ne s'intéressera qu'à la récurrence avant.

Il faut signaler qu'on a utilisé une approche par tabulation : la solution optimale de chaque sous-problème résolu est sauvegardée dans une table pour être utilisée par la suite dans la résolution de sous-problèmes engendrant ce dernier ; on évite ainsi de résoudre le même sous-problème plusieurs fois. En effet, avant d'entamer l'ordonnancement des tâches d'un sous-ensemble S de I, on vérifie dans cette table si F*(S) y figure, on récupère cette valeur pour l'utiliser ultérieurement, par exemple pour l'ordonnancement des tâches de $S \cup \{j\}$ (si $|S|<n$ et $j \notin S$). D'où le terme tabulation introduit ici.

Exemple

Pour ordonnancer les deux sous-ensembles de tâches {1,2,3} et {1,2,4}, on aura besoin de F*{1,2}. On devra donc, lors de l'ordonnancement de {1,2,3}, ordonnancer l'ensemble {1,2} et sauvegarder F*{1,2}. Puis, lors de l'ordonnancement de {1,2,4} on récupère cette valeur de la table.

Pour mettre en évidence la complexité exponentielle de ces algorithmes, on a fixé h à 0,4 et on a représenté graphiquement le temps d'exécution de l'algorithme avant en fonction de n dans la figure 3.1. ci-dessous :

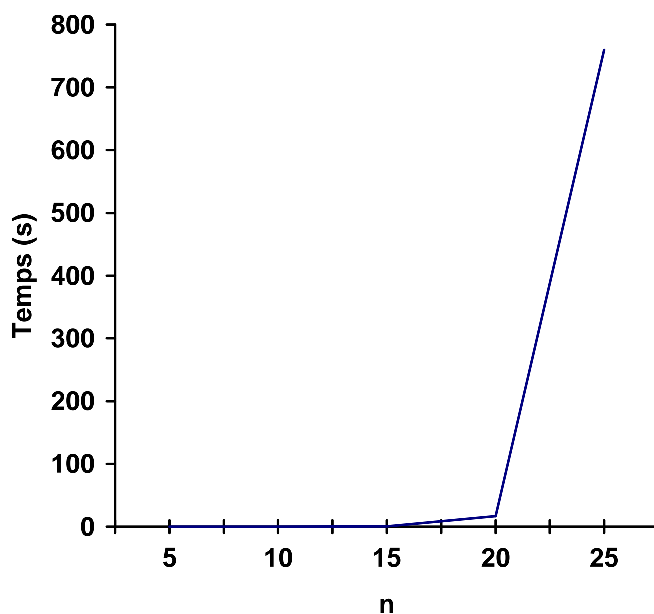

Fig. 3.1. Représentation graphique du temps d'exécution en fonction de n.

4.3. Algorithme dynamique approché

On se propose dans ce paragraphe de présenter un algorithme dynamique approché en considérant l'algorithme avant de la programmation dynamique sus détaillé, mais, pour chaque état S de cardinal k, on n'explorera qu'un nombre aléatoire k_1 ($k_1 \leq k$) de décisions prises aussi aléatoirement de I-S. Ceci se fait par l'ajout d'une ligne calculant k_1 (k_1 = Int(Rnd*k)+1 et la modification de la portée de la boucle : « pour j=1 à k » qui devient « pour j=1 à k_1 » dans l'algorithme en question.

D'une part, la solution trouvée ainsi n'est pas forcément optimale, d'autre part, l'exécution de cet algorithme peut durer aussi longtemps que celle de l'algorithme exact. On a testé ce programme pour plusieurs instances de différentes tailles et on a trouvé que cette tentative est infructueuse dans la plupart des cas. Ceci est principalement dû à la nature de la programmation dynamique où ce n'est pas le nombre de décisions éligibles à chaque phase qui, certes, « explose », mais c'est le nombre d'états qui prime. Malgré le gain en temps d'exécution, l'obstacle majeur demeure l'incapacité d'aborder des problèmes de taille plus importante.

Algorithme Dynamique Approché
Entrées : I, n , p_i , α_i , β_i pour i = 1,n;
Variables intermédiaires : S un sous-ensemble de I ; min , k,k_1 , i , X , Y : entiers ;
Sorties: F*(I) ;
<u>Début</u>
  F*(∅) = 0
  Pour k = 1 à n
    Pour tout sous-ensemble S contenant k éléments de I
      F*(S) = ∞
      k_1=Int(Rnd*k)+1
      Pour k_1 élément i de S
        $C_s = \sum p_i$ pour $i \in S$.
        X = $g_i(C_s)$
        Y = F*(S-{i})
        Min= X + Y
        Si min < F*(S) alors F*(S) = min
      Fin pour
    Fin pour
  Fin pour
<u>Fin</u>

Voici les résultats qu'on a obtenus en comparant cet algorithme avec l'algorithme exact : n = 20 , h= 0,4.

Instances	***Algorithme dynamique exact***		***Algorithme dynamique approché***	
	Solution	***Temps***	***Solution***	***Temps***
01	17 326	17,34	17 430	15,00
02	12 094	18,08	12 408	12,65
03	15 895	18,12	17 120	15,84
04	15 833	16,80	15 945	12,65
05	12 098	17,54	12 273	10,45
06	7 830	17,86	8 126	14,33
07	16 723	18,00	18 100	15,32
08	11 649	17,39	12 544	15,77
09	17 538	16,90	17 652	11,10
10	15 896	16,50	16 053	12,00
Moyennes	14 288	17,45	14 765	13,51

Conclusion : La qualité des solutions approchées trouvées n'est pas tellement bonne. Résultat : cette approche n'est pas opportune.

4.4. Règles de dominance

Les règles de dominance sont des propriétés spécifiques à un problème d'ordonnancement qui permettent, lors de la recherche d'une solution optimale ou approchée, de réduire le nombre de cas à explorer.

Un sous-ensemble P de l'ensemble de solutions E d'un problème est dit dominant pour un critère donné s'il contient au moins une solution optimale du problème [ESQ99]. On dit, alors, que E est dominé par P.

Les solutions des problèmes de minimisation des avances et des retards qu'on se propose de traiter ici possèdent trois propriétés qui peuvent être considérées comme des règles de dominance [LIN06] [FEL03] [KEN90] [HIN05] :

Propriété 1

Dans une séquence optimale, il n'y aura pas de temps mort de la machine séparant l'exécution de tâches successives. C'est-à-dire : si, dans un ordonnancement, la tâche j suit immédiatement la tâche i, alors : $C_j = C_i + p_j$. [KAH93] [CHE91]

On vise par « temps mort » de la machine, la durée pendant laquelle la machine reste inoccupée. Il n'y a pas de tâches en exécution.

Preuve

Supposons qu'il existe une séquence optimale σ avec un temps mort $T_{idle} > 0$ pendant lequel la machine reste inoccupée entre les deux tâches i et j.

D'où : $C_j = C_i + T_{idle} + p_j$.

Soit σ' la séquence obtenue de σ mais où j suit immédiatement i. $C_j = C_i + p_j$.
D'où : $F^*(\sigma) \geq F^*(\sigma') + T_{idle} * \beta_i$; or $T_{idle} > 0$ et $\beta_i > 0$, d'où : $F^*(\sigma) > F^*(\sigma')$. Par conséquent σ n'est pas optimale, ce qui contredit l'hypothèse. Ceci justifie par l'absurde la propriété 1.

Il est clair que cette démonstration reste valable quelque la position de ce temps mort dans l'intervalle [0,T].

Propriété 2

Toute séquence optimale du problème est en forme de V (V-shaped) autour de d : les tâches achevées avant ou exactement à la date d, sont ordonnées suivant l'ordre décroissant du rapport p_i/α_i ; tandis que celles commençant après ou exactement à la date d, sont ordonnées suivant l'ordre croissant du rapport p_i/β_i. Ainsi, une séquence optimale peut éventuellement contenir une tâche « enjambée » (Strapped job) n'appartenant ni au premier sous-ensemble ni au second. C'est-à-dire : elle débute avant d et s'achève après d.[BAK90] [BIS01] [HOO91].

Preuve

Soit σ une séquence des n tâches où i est ordonnancée avant j et toutes les deux avant d; et σ' une séquence des n tâches où j est ordonnancée avant i et toutes les deux avant d.

On alors : $F^*(\sigma) = \alpha i * pi + \alpha j * pj + \alpha i * pj + F(t)$ puisque i précède j ;

$F^*(\sigma') = \alpha i * pi + \alpha j * pj + \alpha j * pi + F(t)$ puisque j précède i ;

D'où : $F^*(\sigma) - F^*(\sigma') = \alpha_i * p_j - \alpha_j * p_i$

i précède j si σ est meilleure que σ', i. e. $F^*(\sigma) - F^*(\sigma') < 0$,

c'est-à-dire : si : $\alpha_i * p_j - \alpha_j * p_i < 0$ i. e. : $p_i / \alpha_i > p_j / \alpha_j$

Donc, dans une séquence optimale, les tâches achevées avant d sont ordonnées suivant l'ordre décroissant du rapport p_i / α_i. On peut prouver de la même manière que : dans une séquence optimale, les tâches débutant après d sont ordonnées suivant l'ordre croissant du rapport p_i / β_i.

Propriété 3

Dans une séquence optimale, la première tâche débute à la date 0 ou bien l'une des tâches s'achève à la date d. [HOO91] [BIS01].

Autrement dit, pour qu'une séquence soit optimale, elle doit impérativement, justifier au moins l'une de ces deux situations.

Preuve

Supposons qu'il existe une séquence optimale σ dont la première tâche débute à la date $t_0 > 0$ et qu'aucune des autres ne s'achève à la date d. ceci implique que σ comprend une tâche i qui débute avant d et s'achève après d.

Soit σ' la séquence obtenue de σ mais où la première tâche débute à la date 0 et σ'' la séquence obtenue de σ mais où la tâche i s'achève à la date d.

Il en résulte que $F^*(\sigma)$ est compris entre $F^*(\sigma')$ et $F^*(\sigma'')$. Ceci implique que σ n'est optimale, ce qui contredit l'hypothèse. Ceci justifie par l'absurde la propriété 3.

Dans la méthode de la programmation dynamique qu'on a utilisée, les propriétés 1 et 3 sont implicitement appliquées, puisque, les séquences recherchées débutent toutes à la date 0 (propriété 3) et si j suit immédiatement i, on a : $C_j = C_i + p_j$ selon l'équation récursive de la formulation dynamique (propriété 1). Pour appliquer la propriété 2, il faut sauvegarder la dernière tâche i ordonnancée dans le sous-ensemble S, et avant d'examiner la tâche j de I-S, il faut d'abord vérifier si la propriété 2 est satisfaite, pour éliminer ainsi une des k itérations. Cet examen se fait de la manière suivante :

Si les tâches i et j sont ordonnancées (s'achèvent) avant d, c'est-à-dire :

si ($C_i < d$ et $C_i + p_j \leq d$) alors : on compare entre p_i/α_i et p_j/α_j :

si $p_i/\alpha_i \geq p_j/\alpha_j$ alors j ne peut pas succéder i (éviter cette itération) ;

Si les tâches i et j sont ordonnancées (débutent) après d, c'est-à-dire :

si ($C_i > d$ et $C_i - p_i \geq d$) alors : on compare entre p_i/β_i et p_j/β_j :

si $p_i/\alpha_i \leq p_j/\alpha_j$ alors j ne peut succéder i (éviter cette itération) ;

L'introduction de cette propriété exige donc, pour chaque itération, d'effectuer 5 tests et calculer 2 rapports, pour, en contre partie, éliminer une itération, ce qui est, évidemment, n'est pas certain : il se peut que cette action soit inopportune, car si les tests sont positifs, après quoi on sera amené au cas sans propriété 2.

Bref. Cette propriété s'applique conjointement avec les heuristiques où elle est de grande rentabilité, comme on le verra dans le paragraphe suivant.

Tests et résultats

Les tableaux suivants illustrent l'effet de l'introduction de la propriété V-shaped sur le temps d'exécution.

h = 0,2

n	DP Sans V-shaped	DP Avec V-shaped
10	139 ms	159 ms
15	765 ms	790 ms
20	18,15 s	19,34 s
25	10,81 mn	12,50 mn

h = 0,4

n	DP Sans V-shaped	DP Avec V-shaped
10	139 ms	159 ms
15	765 ms	790 ms
20	18,10 s	19,34 s
25	10,23 mn	11,66 mn

h = 0,6

n	DP Sans V-shaped	DP Avec V-shaped
10	132 ms	159 ms
15	712 ms	790 ms
20	16,78 s	19,34 s
25	10,17 mn	11,50 mn

h = 0,8

n	DP Sans V-shaped	DP Avec V-shaped
10	109 ms	159 ms
15	650 ms	790 ms
20	16,50 s	19,34 s
25	10,00 mn	10,96 mn

D'après ces tableaux, on conclut qu'il est inutile d'incorporer la propriété V-shaped dans les problèmes de taille assez faible.

5. Heuristique de calcul d'une solution approchée SEA (Sequential Exchange Approach) [LIN05]

5.1. Présentation

On présente dans ce paragraphe une heuristique utilisant les trois propriétés précédentes pour évaluer une solution approchée au problème qui servira de borne supérieure dans la relaxation de l'espace des états qui sera présentée plus loin dans ce chapitre.

Cette heuristique consiste à partitionner l'ensemble I des tâches en deux sous-ensembles A et B : A est l'ensemble des tâches débutant au plus tôt à d, B est l'ensemble des tâches s'achevant au plus tard à d, une tâche transitoire s (straped job) peut éventuellement révéler : celle débutant avant d et s'achevant après d, puis, les tâches de A et B sont ordonnancées suivant la propriété V-shaped. La valeur de la fonction objectif de la séquence ainsi obtenue fournit une première solution approchée qui sera améliorée par permutations de tâches entre A et B.

La solution approchée est notée F* telle que :

$F^* = Fobj(\sigma)$ où σ est une permutation de $A \cup \{s\} \cup B$ obtenue suivant la propriété V-shaped.

où : $Fobj(\sigma) = \sum_{i \in \sigma} g_i(C_i)$ et $g_i(C_i) = \alpha_i\,(d - C_i)$ si $C_i \leq d$;
$g_i(C_i) = \beta_i\,(C_i - d)$ si $C_i > d$;

Remarques

- Si la dernière tâche de B s'achève en d alors s = null ($\{s\} = \varnothing$) (il n'y aura pas de tâche « enjambée ») et σ est donc une permutation de $B \cup A$.
- Si s existe, elle est ordonnancée entre les tâches de B et celles de A..
- Pour évaluer une solution approchée au problème, on doit trier les deux sous ensembles A et B selon la propriété 2, puis évaluer la valeur du critère pour la séquence ainsi trouvée.

5.2. Algorithme

Etape 1.
Trier les tâches de I suivant l'ordre croissant des p_i ; $A = \varnothing$; $B = \varnothing$; $C_B = 0$;
Pour tout i de I
Si ($p_i/\alpha_i < p_i/\beta_i$ et $C_B < d$) alors
$B = B \cup \{i\}$
$C_B = C_B + p_i$
Si non
$A = A \cup \{i\}$
Fin si
Fin pour
Trier les tâches de A et B suivant la propriété V-shaped.
$f^* = Fobj(A \cup B)$

Etape 2.
Pour toute tâche i de B
Pour toute tâche j de A
$S = A \cup B$;
Permuter i et j dans S ;
Si $Fobj(S) < f^*$ alors
$f^* = Fobj(S)$
$B = B \cup \{j\}$; $A = A \cup \{i\}$; $B = B - \{i\}$; $A = A - \{j\}$
Fin si
Fin pour
Fin pour
Trier les tâches de A et B suivant la propriété V-shaped.
$f^* = Fobj(A \cup B)$

Etape 3.
Pour toute tâche i de A
Pour toute tâche j de B
$S = A \cup B$;
Permuter i et j dans S
Si $Fobj(S) < f^*$ alors
$f^* = Fobj(S)$
$B = B \cup \{j\}$; $A = A \cup \{i\}$; $B = B - \{i\}$; $A = A - \{j\}$
Fin si
Fin pour
Fin pour
Trier les tâches de A et B suivant la propriété V-shaped. ;
$f^* = Fobj(A \cup B)$

5.3. Commentaires et discussions

- Cette approche permet de déterminer une séquence des tâches réalisant une solution approchée du problème en question en se basant principalement sur la propriété 2 et la permutation de tâches entre les ensembles A et B.
- La séquence initiale est obtenue de l'ordre SPT (Short Processing Times) pour remédier au cas d'équité lors de l'application de la propriété V-shaped.
- Dans l'algorithme précédent, la tâche s (straped job), si elle existe, est considérée comme dernière tâche de B, ce qui n'altère pas la démarche générale de la méthode.
- L'amélioration que peut éventuellement apporter la permutation de tâches entre A et B découle de leur positionnement dans la nouvelle séquence, c'est pour cette raison qu'on refait l'ordre par la propriété V-shaped.
- La solution approchée obtenue en sortie constituera une borne supérieure initiale de la solution du problème dans la relaxation de l'espace des états qui fera l'objet du paragraphe suivant.
- La complexité de l'étape 1 est O(4n + n(n-1)/2 + 1), celle de chacune de 2 et 3 2 et 3 est $O(9 * n^2/2 + n(n-1)/2 + 1)$. L'algorithme est donc polynomial.

5.4. Tests et résultats

Le tableau suivant illustre les résultats obtenus pour 10 instances
en prenant n = 25 h=0,4.

Instance	Optimum	UB	Erreur Relative
01	32061	33432	0,04
02	12762	12900	0,01
03	7615	8238	0,08
04	12890	14066	0,09
05	23845	24438	0,02
06	21300	21641	0,02
07	30012	31650	0,05
08	9389	10299	0,10
09	17909	19322	0,08
10	15713	17002	0,08
Moyenne	18349,6	19298,8	0,05

A partir de ce tableau, on peut conclure que dans le contexte d'améliorer cette solution approchée ultérieurement par la relaxation lagrangienne, ces résultats sont admissibles, du moment qu'on vise à réduire l'erreur relative.

6. Relaxation de l'espace des états

6.1. Relaxation lagrangienne

La relaxation lagrangienne est une méthode surtout utilisée pour calculer les bornes inférieures (Lower Bound) des solutions de certains problèmes d'ordonnancement pour les intégrer par la suite dans des heuristiques évaluant une solution approchée qui devient, par ce principe, une borne supérieure (Upper Bound) de la solution approchée évaluée.

Munie par ces deux bornes, la solution approchée pourrait s'améliorer itérativement en les faisant rapprocher. Cette approche peut, dans certains cas, converger vers la solution exacte du problème, suivant les données et le nombre d'itérations effectuées.

La relaxation lagrangienne est également très utile pour les méthodes exactes exigeant une borne inférieure pour explorer un arbre de recherche telle que la méthode par séparation et évaluation (Branch and Bound Method).

Elle s'avère très importante du moment qu'on s'aperçoit que l'efficacité de cette méthode dépend essentiellement de la qualité de la borne supérieure générée par la relaxation lagrangienne.

6.2. Relaxation de l'espace des états

En général, la relaxation lagrangienne consiste à relaxer une partie des contraintes et à les intégrer dans la fonction objectif à l'aide des multiplicateurs de Lagrange [CHU96]. On parle ici de la relaxation lagrangienne dans un cadre général pour différents problèmes d'optimisation et différentes méthodes de résolutions. Cependant, dans ce volet, on ne s'intéressera qu'à la relaxation de l'espace des états générés par la méthode de la programmation dynamique appliquée à notre problème. Or, on a vu que la programmation dynamique génère 2^n états (n est le nombre de

tâches), sachant que $T << 2^n$, $(T = \sum_{i \in I} p_i)$, l'approche qu'on se propose de présenter consiste à relaxer cet espace d'états dans un ensemble de temps discrets {0,1, 2 , . , T} où en introduisant un vecteur $\lambda = (\lambda_1, \lambda_2, , , \lambda_n)$ associé aux tâches et en définissant une fonction f associée au vecteur λ de la manière suivante :

Pour $t \in Z$:

- $F(t, \lambda) = \infty$ si $t < 0$;
- $F(0, \lambda) = 0$;
- $F(t, \lambda) = \min_{i \in I} \{F(t - p_i, \lambda) + g_i(t) + \lambda_i\}$ si $t = 1,2,\dots, T$;

$F(T, \lambda)$ est une somme de n termes, qui ne sont pas forcément distincts, de la forme : $g_i(C_i) + \lambda_i$; ceci veut dire que : de cette approche découle une séquence de n tâches qui peut éventuellement contenir des tâches répétées (successivement ou non) ; donc d'autres tâches n'apparaissent pas dans cette séquence.

En termes mathématiques, cette séquence est un arrangement de n éléments de I et ne serait une permutation de I que si chaque tâche de I y apparaît une et une seule fois. Posons donc : $F(T, \lambda) = g_a(C_a) + \lambda_a + g_b(C_b) + \lambda_b + \dots$

On sait que dans l'expression du critère ne doit apparaître que les termes de la forme $g_i(C_i)$; il faut alors retrancher les composantes λ_i rajoutée dans la formulation dynamique ci-dessus. La résolution récursive du problème ainsi formulé aboutit alors en dernière itération à la borne inférieure initiale $LB(\lambda) = F(T, \lambda) - \sum \lambda_i$.

Remarquons qu'on n'a pas certainement rajouté tous les λ_i mais on les a retranchés tous, ceci justifie le qualificatif «inférieure» de la solution approchée trouvée ainsi.

Si, en revanche, si chaque tâche apparaît une et une seule fois dans la séquence, cette dernière est certes, optimale puisqu'elle réalise le minimum (voir l'équation récursive).

Or, quelque soit le vecteur λ, on a : $LB(\lambda) \leq F^*(I)$. ($F^*(I)$ est la solution optimale du problème) ; par conséquent, l'amélioration ou éventuellement l'exactitude de $LB(\lambda)$ dépend du choix du vecteur λ.

C'est pourquoi on introduit ici une approche itérative pour trouver le meilleur vecteur λ.

On commence par $\lambda^{(0)} = (0 , 0 , \dots , 0)$ et pour tout entier non nul k :

$$\lambda_i^{(k)} = \lambda_i^{(k-1)} + \frac{(f^{(k-1)} - LB(\lambda^{(k-1)}))(n_i^{(k-1)} - 1)}{\sum_{j \in I}(n_j - 1)^2}$$

Où : $\lambda_i^{(k)}$ est la $i^{ème}$ composante du $k^{ème}$ vecteur λ.

$n_i^{(k-1)}$ est le nombre d'occurrences de la tâche i dans la séquence obtenue dans le calcul de $LB(\lambda^{(k-1)})$. Si, à l'itération k, on a $\forall i \in I : n_i^{(k)} = 1$, alors la séquence est optimale.

$f^{(k-1)}$: est une estimation de la solution approchée du problème définie par :

$f^{(0)} = f(I)$: solution approchée du problème obtenue par l'heuristique SEA détaillée dans le paragraphe précédent.

Si $LB(\lambda^{(k-1)}) \geq f^{(k-1)}$ à l'itération k-1 alors :

$$f^{(k)} = LB(\lambda^{(k-1)}) + | f^{(k-1)} - f^{(k-2)} | / 2$$

Si durant k itérations successives, la borne inférieure ne s'améliore pas, alors :

$$f^{(k)} = (\mu f^{(k-1)} + (1-\mu) \max_{i \in [1,k-1]} \{LB(\lambda^{(i)}) \}) \;;\; \mu \in]0,1[$$

En appliquant cette récursivité pour quelques centaines d'itérations, on parvient progressivement à améliorer le vecteur λ, ce qui conduit à rapprocher $LB(\lambda^{(k)})$ de $F^*(I)$. Selon les tests qu'on a effectués, on a obtenu empiriquement que pour k = 500 on puisse trouver des résultats satisfaisants. Un nombre plus grand d'itérations peut éventuellement donner des résultats meilleurs. Cependant, il se peut que, pour certains problèmes, on se retrouve dans une situation où on perd du temps dans des itérations pratiquement inutiles, c'est-à-dire que la borne inférieure ne s'améliore pas durant ces itérations. C'est pour cette raison que cette approche n'est pas toujours convergente.

Traduisons cette approche par un algorithme à trois étapes, qui représentera la relaxation de l'espace des états qu'engendre la méthode de la programmation dynamique.

6.3. Algorithme de relaxation de l'espace des états

Algorithme Relaxation()
Entrées : $f^*(I)$, n, p_i, α_i, β_i ***pour*** $i = 1,n$;
Variables intermédiaires : $\lambda(n)$, ***LBmax ; min , k , i , t : entiers ;***
Sorties: $LB^{(k)}$;
Début
$T = \sum_{i=1}^{n} p_i$; $f^{(k-1)} = f^*(I)$; $\lambda^{(k-1)} = (0, 0, \ldots, 0)$; $LBmax = 0$; $LB^{(k)} = 0$;
Pour k = 1 à 7
  Pour t = 1 à T
    $F(t) = \infty$;
    Pour i=1 à n
      Si $t - p_i < 0$: $min = \infty$;
      Si $t - p_i = 0$: $min = g_i(t) + \lambda_i$;
      Si $t - p_i > 0$: $min = F(t - p_i) + g_i(t) + \lambda_i$;
      Si $min < F(t)$ ***Then*** $F(t) = min$;
    Fin pour
  Fin pour
    $LB^{(k-1)} = LB^{(k)}$; $LB^{(k)} = F(T) - \sum \lambda_i$;
    Si $LB^{(k)} > LBmax$: $LBmax = LB^{(k)}$;
    Si $LB^{(k-1)} \geq f^{(k-1)}$ ***alors***
      $f^{(k-1)} = LB^{(k-1)} + Abs(f(k-1) - f^{(k-2)}) / 2$
    Si non
      $f^{(k-2)} = f^{(k-1)}$
      $f^{(k-1)} = f^{(k)}$
      $f^{(k)} = \mu * f^{(k-1)} + (1-\mu) * LBmax$
    Fin si
    Pour i=1 à n : $\lambda_i^{(k)} = \lambda_i^{(k-1)} + \dfrac{(f^{(k-1)} - LB(\lambda^{(k-1)}))(n_i^{(k-1)} - 1)}{\sum_{j \in I} (n_j - 1)^2}$
  Fin pour
Fin

6.4. Commentaires et discussions

- On a vu que la complexité des algorithmes de la programmation dynamique est $O(n2^{n-1})$, en revanche, celle de l'algorithme de relaxation est $O(nT)$, ce qui souligne l'écart considérable des temps d'exécutions et soutient le choix de cette approche.

- Cette approche peut converger, dans certains cas, après un certain nombre d'itérations, vers la solution exacte du problème ce qui peut être utilisée pour évaluer l'efficacité d'une heuristique. Ceci ait lieu si à une itération donnée k on aurait : $\forall i \in I$: $n_i^{(k)} = 1$. ce nombre d'itérations k est fonction de plusieurs facteurs : les données du problème, l'heuristique fournissant la borne supérieure initiale, le paramètre μ et, bien entendu le hasard. Mais, attention, ce n'est pas toujours le cas.
- L'effet du paramètre μ révèle une grande importance car il permet de choir un positionnement de $f^{(k)}$ dans l'intervalle $[LB^{(k-1)}\ ,\ f^{(k-1)}]$. C'est en d'autres termes, le barycentre au lieu du centre de ces deux valeurs pour permettre ainsi plusieurs éventualités d'amélioration.

6.5. Tests et résultats

Les tableaux suivant illustre les résultats obtenus pour 10 instances
avec n = 25 , k = 7, h = 0.4

μ = 0.25

Instance	Optimum	LB	Erreur Relative
01	32061	30562	0,05
02	12762	11897	0,07
03	7615	7256	0,05
04	12890	12374	0,04
05	23845	23348	0,02
06	21300	20238	0,05
07	30012	28130	0,06
08	9389	8761	0,07
09	17909	17207	0,04
10	15713	15249	0,03
Moyenne	18349,6	17502,2	0,05

$\mu = 0.50$

Instance	Optimum	LB	Erreur Relative
01	32061	31400	0,02
02	12762	12108	0,05
03	7615	7350	0,03
04	12890	12374	0,04
05	23845	23560	0,01
06	21300	20600	0,03
07	30012	28325	0,06
08	9389	8806	0,06
09	17909	17283	0,03
10	15713	15340	0,02
Moyenne	18349,6	17714,6	0,04

$\mu = 0.75$

Instance	Optimum	LB	Erreur Relative
01	32061	31736	0,01
02	12762	12482	0,02
03	7615	7498	0,02
04	12890	12490	0,03
05	23845	23603	0,01
06	21300	20850	0,02
07	30012	28780	0,04
08	9389	8976	0,04
09	17909	17540	0,02
10	15713	15532	0,01
Moyenne	18349,6	17948,7	0,02

Les résultats ci-dessus montrent l'effet du paramètre μ sur la qualité de la solution approchée et que le choix de ce paramètre dépend de l'heuristique adoptée pour le calcul de la borne supérieure initiale. Pour l'heuristique utilisée ici, il s'avère que le cas μ = 0,75 réalise le bon résultat.

Pour montrer l'effet du paramètre μ sur la qualité de la solution approchée, on a représenté graphiquement l'erreur relative en fonction de μ en prenant : h=0,4 ; n=25. (fig. 3.2).

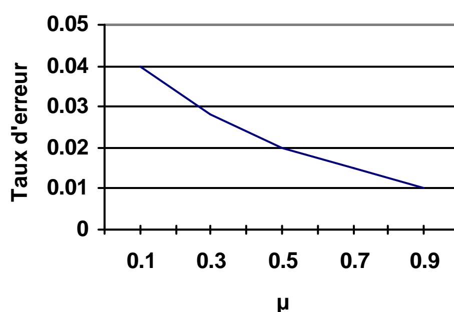

Fig. 3.2. Représentation graphique du taux d'erreur en fonction de μ.

Ce graphe montre clairement que le taux d'erreur est inversement proportionnel au paramètre μ.

Résultat : pour μ ≈ 1 , on obtient des résultats meilleurs.

Le tableau suivant illustre le temps d'exécution de l'algorithme de relaxation pour des problèmes de tailles plus grandes.(k=7, μ=0.75).

h	n=50	n=100	n=200	n=500	n=1000
0.2	2,47	7,60	18,01	132,06	441,43
0.4	2,47	7,53	17,89	122,43	423,50
0.6	2,45	7,50	17,76	115,50	412,67
0.8	2,44	7,14	17,68	109,82	408,50
Moyenne	2,45	7,44	17,83	119,95	421,52

Ce tableau reflète nettement l'efficacité de la relaxation de l'espace des états surtout en temps d'exécution.

7. Conclusion

On a traité dans ce chapitre le problème d'ordonnancement de minimisation de la somme des coûts des avances et des retards sur une seule machine et avec une date échue commune en trois principaux volets :

D'abord, par une méthode exacte, la programmation dynamique, où on a introduit un procédé de tabulation qui a permis de résoudre des problèmes de ce type dans un temps raisonnable.

En deuxième lieu, on a élaboré un algorithme approché fondé sur l'approche SEA qui consiste en la permutation de tâches et l'application de la propriété V-shaped, cet algorithme a donné des résultats qui peuvent considérés comme bornes supérieures initiales dans la relaxation de l'espace des états.

Enfin, on a appliqué la relaxation de l'espace des états au problème en introduisant un paramètre pour déterminer la meilleure stratégie à adopter pour améliorer la borne inférieure de la solution.

On estime que cette démarche puisse être un outil pour aborder ce type de problèmes et qu'il puisse être généralisé et adapté à d'autres types de problèmes.

CONCLUSION GENERALE

Par le biais du travail qu'on a abordé ici, on a visé plusieurs objectifs, qu'il s'avère très intéressant de les évaluer ici en évoquant les principaux concepts qu'on a étudiés pour donner quelques éléments de réponse aux questions qu'on considère pertinentes:

- En dépit de sa lenteur et de son besoin excessif en espace mémoire, la méthode de la programmation dynamique demeure un outil très puissant et souple pour aborder des problèmes d'optimisation combinatoires en général de tailles assez faibles et pour tester et apprécier les méthodes approchées et l'évolution tellement accélérée des ordinateurs pourra, certes, réduire considérablement l'effet de ses handicaps.
- En particulier, l'application de la programmation dynamique à la résolution de problèmes d'ordonnancement fournit une alternative aux autres méthodes exactes puisqu'elle s'adapte à toutes les fonctions objectifs et à tous types de contraintes où on n'ait pas besoin d'une heuristique pour déterminer une borne inférieure de la solution comme dans la méthode par séparation et évaluation.
- La relaxation lagrangienne de l'espace des états qu'engendre la programmation dynamique fournit un moyen très efficace pour donner une solution approchée de bonne qualité à un problème d'ordonnancement, voire même, dans certains cas, pour déterminer la solution exacte du problème, et ce, suivant les données et le nombre d'itérations effectués.
- L'approche SEA (Sequential Exchange Approach) spécifique au problème qu'on a étudié ici, représente un procédé très simple qui peut être généralisé à d'autres types de problèmes et peut participer ainsi aux sujets d'actualités.

Dans le cadre d'un bref constat des points essentiels de la contribution apportée par ce modeste travail, on doit citer les éléments suivants :

- ✓ Une conception abstraite et une formulation généralisée de la programmation dynamique qu'on a projetées sur deux exemples illustratifs.

- ✓ La résolution d'un problème d'ordonnancement de type A/R par la méthode de la programmation dynamique en faisant intervenir une procédure de tabulation ce qui a manifesté un gain considérable en temps d'exécution.
- ✓ La démonstration mathématique des trois propriétés du problème en question qu'on n'a pas pu, malheureusement, trouver ailleurs.
- ✓ L'application, pour la première fois, de la relaxation de l'espace des états au problème avec une approche qui est propre à notre étude et qui a donné des résultats appréciables dans le contexte de la recherche de solutions approchées.
- ✓ L'introduction d'une version spécifique de l'approche SEA pour rechercher la borne supérieure de la solution optimale fondée sur les trois propriétés du problème et la permutation de tâches.

En fin, dans le cadre de situer cette contribution dans les sujets ayant trait au domaine de ce travail, je me permets de soulever les éléments suivants :

- ➢ L'évolution de la technologie pourra donner un véritable soutien à la programmation dynamique pour remédier à ses inconvénients et fournira ainsi un grand apport aux problèmes d'optimisation.
- ➢ La démarche générale appliquée qui se base essentiellement sur trois volets : la programmation dynamique, l'approche SEA pour déterminer une solution approchée qui servira de borne supérieure et la relaxation de l'espace des états pour déterminer une borne inférieure de la solution du problème, peut être améliorée empiriquement selon le nombre et la forme des itérations.
- ➢ Quant aux problèmes d'ordonnancement traitant les coûts des avances et des retards, malgré le progrès considérable des techniques de l'ordonnancement ces dernières années, ils demeurent un espace très vaste de diverses situations qu'on peut rencontrer en pratique et tellement variés qu'on ne puisse estimer la complexité. C'est dans de le but de fournir une contribution à de tels sujets et de participer à l'évolution si rapide de ce domaine qu'on a abordé ce sujet. On espère avoir donné au moins une synthèse d'une démarche qui peut servir d'une plate-forme pour faire face à d'autres situations et ouvre ainsi de nouveaux horizons dans le domaine.

REFERENCES BIBLIOGRAPHIQUES

[ABD87] Abdul-Rasak T. S. 1987, Machine Scheduling Problems : A Branch and Bound Approach. PHD Thesis. Department of Mathematics. University of Keel.

[BAK74] Baker. K.R 1974. Introduction to Sequencing and Scheduling. John Wiley & Sons.

[BAK78] Baker. K.R and Schrage L.E. 1978. Finding an optimal sequence by Dynamic Programming an Extension to precedence-related tasks. Oper. Res. 26, 111-120.

[BAK88] Baker. K.R and Scudder G. D. 1988. sequencing with earliness and tardiness penalties. A review. Oper. Res. 38, 22-36.

[BEL85] Belouadah H. 1985. Scheduling to minimize Total Cost. MSC. Thesis University of Keele. U.K.

[BRU01] Bruno B. 2001. informatique/recherche opérationnelle. Bruno.bachelet.net.

[CAR88] Carlier J. et Chrétienne P. 1988. Problèmes d'ordonnancement Modélisation complexité/algorithmes. Masson.

[CHU96] Chu C. and Proth J. M. 1996. l'ordonnancement et ses applications. MASSON. Paris.

[CON67] Conway R. W. Maxwell W. L. and Miller L. W. 1967. Theory of Scheduling. Addison Wesley, Reading, MA.

[CON67] Conway R. W. Maxwell W. L. and Miller L. W. 1967. Theory of Scheduling. Addison Wesley, Reading, MA.

[EAS59] Eastman W. L. 1959. A solution to the Travelling Salesman Problem Econometrica. 27, 282.

[ESQ99] Esquirol P. Lopez P. 1999. L'ordonnancement, série:production et techniques quantitatives appliqués à la gestion, ECONOMICA, Paris.

[FAU79] Faure R. 1979. Précis de Recherche Opérationnelle. Dunod , Paris.

[FEL02] Feldmann M. and Biskup D. 2002. Single Machine Scheduling For Minimizing Earliness And Tardiness Penalties By Meta-Heuristic Approaches. Computers & Industrial Engineering 44 (2003) 307-323.

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

[GRA66] Graham R. L. 1966. Bounds for Certain Multiprocessing Anomalies. Bell Sys. Tech. J. 45, 1563-1581.

[GRA69] Graham R. L. 1969. Bounds on Multiprocessing Anomalies. SIAM. J. Appl. Math. 17, 416-429.

[HEL62] Helde H. and Karp R.M. 1962. A Dynamic Programming Approach to Sequencing Problems. SIAM. J. 10, 196-210.

[HIN04] Hino C. M., Ronconi D. P. and Mendes A. B. 2004. minimizing earliness and tardiness penalties in a single machine with a common due date. European Journal. O. R. 160 (2005) 190-201.

[HOO91] J. A. Hoogeveen, S. L. Van, De Velde, Scheduling around a small common due date. European Journal of O. R. 55 (1991) 237-242.

[IBA92] Ibaraki T. and Nakamura Y. 1992. A Dynamic Programming Method For Single Machine Scheduling. European. Journal. O. R. 76 (1994) 72-82.

[IGN65] Ignall. E. and Schrage. L. 1965. Application of The Branch and Bound Technique to some Flow-Shop Scheduling Problems. Oper. Res. 13, 400-412.

[KAH93] H. G. Kahlbacher. Scheduling With Monotonous Earliness And Tardiness, European journal of O. R. 64 (2) (1993) 258-277.

[KAN76] Rinnooy Kan A. H. G. 1976. Machine Problems : Classification, Complexity and Computations. *Martinus Nijhoff, The Hague, Holland.*

[KAR72] Karp. R. M. Reducibility AMONG Combinatorial Problems. In : R. E. Miller & J. W. Thatcher (eds) Complexity of Computer Computations. Plenum Press, New York. 85-103.

[KEN88] Kenneth R. Baker and Gardy D. Scudder. 1988. Sequencing with Earliness and Tardiness Penalties : A Review.

[LAN60] Land A. H. and Doig. A. G. 1960. An Automatic Method for Solving Discrete Programming Problems. Econometrica 28, 497-520.

[LAW64] Lawler E. L. 1964. On Scheduling Problems with Defferal Costs. Management Sci. 11, 280-520.

[LAW77] Lawler E.L. 1977. A pseudoplynomial algorithm for sequecing jobs to minimise total tardiness. Annals of Discrete Mathematics, 1,331-342.

[LIA92] Liao C-J and You C-T 1992. An Improved Formulation on the Job-Shop Scheduling Problem. J Oper. Res. 43, 1047-1054.

[LIN05] Lin S. W., Chou S. Y. and Ying K. C. A Sequential Exchange Approach For Minimizing Earliness-Tardiness Penalties Of Single-Machine Scheduling With A Common Due Date. European Journal O. P. (2006).

[LIA03] Liao C. J. Chen W. J. 2003. Single machine scheduling with periodic maintenance and nonresumable jobs.Computers and O. R. 30,1335-1347.

[LOM65] Lomnicki Z. A. 1965. A Branch and Bound Algorithm for the Exact Solution of the Three-machine Scheduling Problem. Oper. Res. Quart. 16, 89-100.

[MAL57] Posner M. E. 1985. Minimizing Weighted Completion Times with Deadlines. Oper. Res. 33, 562-574.

[MAR02] Posner M. E. Nicholas G. H. 2000. Generating experemental data for computational testing with machine scheduling applications. Operations Research , pp 854-865.

[SAK84] Sakarovitch M. 1984. Optimisation Combinatoire. Méthodes mathématiques et algorithmiques, Programmation discrets.HERMANN. Paris.

[TIS05] Tison. S. 2004-2005. Programmation Dynamique. Cours master informatique. UFR IEEA.