UNIVERSITE DE BLIDA, FACULTE DES SCIENCES
DEPARTEMENT DE MATHEMATIQUES

Post-Graduation : Recherche Operationel / Modélisation Stochastiques

Année Universitaire 2020 - 2021

Module : Ordonnancement

Cours donnés par : Dr. Redouane Boudjemaa

Rédiger et conçu par: Dr. Ali Derbala

Université de Blida1, Faculté des sciences, Département de Mathématiques,

BP 270, Route de Soumaa, Blida, Algérie.

Tel & Fax : (+) 025- 43 36 42

Email : boudjemaa.redouane@univ-blida.dz

Résumé :

Ce cours est une introduction à la théorie de l’ordonnancement. Il a pour objet de donner aux étudiants une culture large et récente en ordonnancement dans les ateliers ou en informatique. Après une introduction générale aux problèmes d'ordonnancement (typologie et notation), ils aborderont les problèmes à une machine, à machines parallèles, les problèmes de type "flowshop", "jobshop" et l’open shop. Ce cours aborde les principaux algorithmes d’ordonnancement utilisés en informatique et plus particulièrement dans l'optimisation. Après un rappel sur les propriétés fondamentales des problèmes classiques (l'optimisation numérique et combinatoire, avec ou sans contraintes) sont traités plus particulièrement des approches les méthodes évolutionnistes (algorithmes génétiques, recuit simulé, recherche tabou et les processus bandits).

Mots clés : ordonnancement, machine, machines parallèles, flowshop, jobshop et affectation.

Programme d’enseignement

Introduction

1. Les problèmes d'ordonnancement d'ateliers

2. Formulation des problèmes d’ordonnancement

2.1. Définition des problèmes d’ordonnancement

2.2. Analyse des problèmes d’ordonnancement et des algorithmes

2.3. Classification des problèmes d’ordonnancement.

3.Complexité des algorithmes et des problèmes d’ordonnancement

4. Ordonnancement à une seule machine

4.1 Minimisation de la longueur d’un ordonnancement

4.2 Minimisation du flow time pondéré moyen

4.3 Minimisation des critères incluant des dates de fin au plus tard ou due-date

4.3.1 décalage maximum

4.3.2 nombre de tâches en retard

4.3.3 les problèmes de retard et d’avance

4.3.4 Autres critères

5. Résolution des problèmes difficiles par des méthodes énumératives

5.1.1. Méthodes par séparation et évaluation

5.1.2. Programmation dynamique

6. Ordonnancement de type shop static

6.1 Ordonnancement flow shop

6.2 Ordonnancement open shop et job shop

6.3. Ordonnancement flow shop hybride

7. Ordonnancement de machines parallèles

7.1 Minimisation de la longueur d’ordonnancement ou makespan

7.1.1 machines identiques

7.1.2 machines uniformes et quelconques (non-related)

7.1.2 Minimisation du flow time

Autres modèles

8. Résolutions approchées des problèmes difficiles d’ordonnancement

Pour l'évaluation des étudiants, un mini-projet est proposé. Il consiste en la lecture, la présentation et l'implémentation d’algorithmes parus dans la littérature. Un examen écrit de fin d'année, d'une durée de deux heures, est aussi à prévoir.

Introduction

Ce module a pour objet de donner aux étudiants une culture large et récente en ordonnancement dans les ateliers (et en informatique). Après une introduction générale aux problèmes d'ordonnancement (typologie et notation), ils aborderont les problèmes à une machine, à machines parallèles, les problèmes de type "flowshop", "jobshop" et l'open shop. Ce cours aborde les principaux algorithmes d'ordonnancement utilisés et plus particulièrement dans l'optimisation. Après un rappel sur les propriétés fondamentales des problèmes classiques (l'optimisation numérique et combinatoire, avec ou sans contraintes) sont traités plus particulièrement des approches les méthodes évolutionnistes (algorithmes génétiques, recuit simulé, recherche tabou et les processus bandits).

Ordonnancer un ensemble de tâches, c'est programmer leur exécution en leur allouant les " ressources " requises et en fixant leur dates de début d'exécution. La théorie de l'ordonnancement traite de modèles mathématiques mais analyse également des situations réelles fort complexes. L'ordonnancement est un domaine très vaste et varié. Il existe même un journal dédié spécifiquement aux problèmes d'ordonnancement : "Journal of scheduling". Ce cours a pour but de vous familiariser avec les notations, et de vous donner quelques résultats obtenus dans le domaine de l'ordonnancement. De nouveaux modèles et résultats sont régulièrement présenter dans les conférences et journaux.

En effet, *l'ordonnancement* est lié à des secteurs de recherches et d'activités très variés. La conception et la réalisation de grands projets de constructions navales ou aéronautiques, de grands chantiers routiers ou ferroviaires et les constructions de travaux publics se modélisent comme des problèmes d'ordonnancement. En informatique, le partage de la mémoire, le choix des tâches à exécuter sur des processeurs font également appel à cette théorie. Nous citons aussi l'organisation de la production et la détermination des cycles de fabrication dans les ateliers manufacturiers. La théorie de l'ordonnancement est une branche de la recherche opérationnelle. Elle consiste en l'analyse de situations réelles évoquées ci-dessus, la recherche de modèles mathématiques et la mise au point de méthodes de résolution. Les problèmes traités sont à une seule machine, à machines parallèles et en séries.

Pour un fort long temps et spécialement dans les travaux publics, pour représenter une solution d'un problème d'ordonnancement, on utilise couramment une représentation dite *diagramme de Gantt*.

Celle-ci utilise un repère orthogonal à deux dimensions. L'axe des abscisses représente le temps et l'axe des ordonnées représente l'ensemble des machines. Pour chaque machine, on représente la séquence de tâches effectuées dans le temps.

Dés 1958, de nouvelles méthodes ont été développées, à savoir, les méthodes *P.E.R.T* (*Program Evaluation and Review Technique*), *C.P.M* (*Critical Path Method*) et la méthode des *« Potentiels tâches »*. Elles sont fondées sur le calcul des chemins ou tensions de valeur maximum dans un graphe valué.

Depuis quelques années, l'informatique est présente et rythme notre quotidien. En effet, il suffit de constater le taux d'équipement des ménages en ordinateur, l'évolution du monde automobile, tant au niveau de la conception par l'intermédiaire des outils de C.A.O., que de la fabrication avec la robotisation, de l'aide à la conduite avec les ordinateurs de bord, du pilotage automatique des avions, de la connaissance en temps réel des bouchons, etc.

Si le dernier siècle était celui des machines, le siècle qui s'ouvre à nous sera celui de la communication, des échanges de toutes sortes via les réseaux à haut-débits. Mais avec cette expansion, de nouveaux problèmes apparaissent liés à la discipline elle-même (où le parallélisme, l'internet et la globalisation de l'environnement de l'information en sont la face apparente), et également avec l'apparition de problèmes calculatoires importants issus d'autres disciplines (analyse de séquences en biologie, simulations numériques en physique, etc.). L'utilisation du parallélisme, l'emploi de plusieurs processeurs, pour le traitement des applications de grande taille qui réclament une puissance de calcul de plus en plus importante est aujourd'hui une réalité. En effet, il n'est plus à démontrer l'intérêt du parallélisme pour le traitement des grandes applications issues de le physique (simulations en physique nucléaire) ou le traitement des séquences de nucléotides en biologie. Des applications de ce type ne peuvent pas être traitées en un temps raisonnable sur une machine séquentielle. Dans le but de les traiter le plus rapidement possible, les solutions techniques qui ont été développées pour le parallélisme sont des architectures parallèles avec divers choix architecturaux : des architectures parallèles avec mémoire partagée (les processeurs se partagent une mémoire centrale), à mémoire distribuée (chaque processeur dispose d'une mémoire), des machines vectorielles, etc. Ces dernières années, il existe un regain d'intérêt pour le parallélisme avec l'apparition et l'utilisation de plus en plus croissante des grappes de stations de travail comme machine parallèle.

Néanmoins, les puissances de calcul théoriques des machines parallèles ne sont, en pratique, jamais atteintes. Ceci est dû principalement aux difficultés liées à la gestion des ressources et des contraintes de fonctionnement des architectures multiprocesseurs. Parmi les difficultés

que l'on rencontre, on peut citer l'étape d'extraction du parallélisme intrinsèque d'une application ou les problèmes de routage et d'ordonnancement des communications entre les différents processeurs de l'architecture.

Pour tenter de pallier à la première difficulté, une phase de partitionnement est nécessaire. Elle correspond à la première étape fondamentale de la parallèlisation et consiste en l'extraction du parallélisme potentiel d'une application. L'extraction du parallélisme détermine les dépendances fonctionnelles entre les différentes parties d'une application. Ainsi, une application parallèle peut être représentée sous forme d'un graphe orienté sans circuit (graphe de précédence) où chaque sommet du graphe représente une partie (instruction ou ensemble d'instructions) du programme et les arcs, les contraintes chronologiques entre ces parties (dites aussi contraintes de précédence). Dans le cadre des modèles idéalisés où les communications sont considérées comme instantanées (par exemple le modèle PRAM), la mesure cruciale de la complexité parallèle est la profondeur du graphe
de précédence. Or il s'avère que dans la pratique, on doit tenir compte du délai de communication entre l'instant où une information est produite par un processeur, et l'instant à partir duquel cette information peut être utilisée par un autre processeur. Ceci induit un surcoût lié aux communications qui dépend, entre autres, de la quantité d'information échangée. L'ordonnancement des tâches de l'application va dépendre dans ce cas non seulement des durées d'exécution des tâches mais aussi des temps de communications entre celles-ci.

Plusieurs modèles prennent en compte les délais de communications entre les différentes parties d'une application dont le plus populaire est celui qui est connu comme le modèle à *communications homogènes*. Dans ce modèle nous supposons que tous les processeurs de la machine parallèle sont totalement connectés : si deux tâches communicantes s'exécutent sur deux processeurs différents alors la communication potentielle, entre ces deux tâches, est indépendante des processeurs sur lesquels elles s'exécutent. La localisation de l'exécution des tâches n'influe pas sur la communication. Il est important de noter que la prise en compte de la topologie de la machine parallèle induit un degré de difficulté supplémentaire rendant les problèmes d'optimisation très difficiles car le placement des tâches sur les processeurs devient une composante fondamentale dans le but d'obtenir un ordonnancement réalisable et de "bonne qualité". Dans le modèle avec *communications homogènes*, nous pouvons distinguer deux types de communications :

– les *communications intra-processeurs* : ce sont les communications entre deux tâches adjacentes dans le graphe de précédence, qui s'exécutent sur le même processeur (en respectant les contraintes de précédence). Nous supposerons ces délais négligeables.
– les *communications inter-processeurs* : ce sont les communications entre deux tâches adjacentes dans le graphe de précédence qui s'exécutent sur des processeurs différents.
Formellement, nous considérons un graphe orienté sans cycle avec un ensemble des sommets et un ensemble des arcs. Chaque tâche a une durée d'exécution et à chaque arc est associée une valeur représentant le délai de communication potentiel entre les tâches.
Ce modèle a été très largement étudié ces dernières années (recherche de solutions approchées, résultats de complexité, algorithmes optimaux ...), comme en témoignent les nombreux articles publiés sur le domaine.

Chapitre1 : Les problèmes d'ordonnancement d'ateliers

1. Introduction

En général, il y a trois types de problèmes d'ordonnancement. L'ordonnancement de grand projets, dans les ateliers et l'ordonnancement dans les systèmes informatiques appelé aussi ordonnancement en temps réel où la réponse au problème doit être instantanée et donnée en tout instant. En Anglais, on utilise deux appellations différentes pour l'ordonnancement. S'il y a une machine, on parle de " sequencing " ou séquencement et s'il y a plusieurs machines on parle de "scheduling" qui est la notion " d'allocation et de séquencement " .

Dans ce cours, nous considérons l'ordonnancement dans les ateliers industriels.

Un atelier est formé de machines reliées par des moyens de transport (de façon analogue, un ordinateur est constitué de composants : mémoires, processeurs…, reliés par des connexions). Il est destiné à fabriquer des pièces (dont le modèle théorique est appelé job), elles mêmes se décomposant en diverses tâches, et ce travail nécessite des ressources (machines, moyens de transport, matière première, ou bien, si l'atelier est un ordinateur : processeurs, bus, mémoires). Des questions se posent à propos de la conception puis de la gestion d'un atelier.

- *Conception* d'un futur atelier : outre bien sûr, les types de machines et leurs caractéristiques liées au produit à fabriquer. Pour optimiser l'atelier, il faut répondre principalement aux deux questions suivantes :
  1) quel doit être l'agencement global, la topologie de l'atelier (en tenant compte des caractéristiques du bâtiment, des impératifs de sécurité, des impératifs sociaux…)? Ceci ne concerne pas seulement la topologie des machines et des stockages, mais encore et surtout la nature et la topologie des moyens de transport entre ces machines, ainsi qu'à l'entrée et à la sortie de l'atelier.
  2) quelles doivent être les caractéristiques technologiques des machines, comme celles des moyens de transport, compte tenu des débits, des besoins envisagés ou prévisibles?

  Ces questions sont fort complexes et interférent fortement.
- *Modification* d'un atelier existant, que l'on veut améliorer, en le réorganisant : on se heurte à des problèmes avec des contraintes liées au passé, et qui se résument souvent en la question : que peut-on faire pour apporter l'amélioration souhaitée au moindre prix, ou en apportant un minimum de bouleversements, en particuliers sociaux ?

- *Gestion* d'un atelier en fonctionnement : lorsque les pièces à fabriquer sont de diverses natures, on est amené à se poser la question : comment gérer " au mieux " la fabrication, compte tenu :

  des capacités, de l'organisation, et des caractéristiques de l'atelier,

  des ressources disponibles en matière première,

  des besoins en nature, quantité, date de fin de fabrication à respecter,

  des gammes d'usinages des pièces considérées,

  des dates de disponibilités des pièces à usiner,

  des contraintes légales et sociales,

  …….

Ces questions qui constituent le problème de *l'ordonnancement prévisionnel* sont complexes à résoudre. On se heurte à des problèmes *d'ordonnancements exceptionnels* où la solution doit en être trouvée dans un délai beaucoup plus court, lorsque l'on se demande :
" que faire en cas de panne ? " ou bien " comment et quand introduire une pièce urgente?"
Une fois ces questions résolues, il faut encore *piloter*, en temps réel, l'atelier en tenant compte, non pas des comportements prévisibles, mais des comportements réels observés et mesurés, pour ajuster l'ordonnancement prévisionnel, sans parler des décisions à prendre, à tout moment, devant une modification inopinée(éventuellement non prévue) des pièces (pièces urgentes) ou des caractéristiques de l'atelier internes(panne), ou externes(rupture de stock). Tout ceci constitue le problème de la *conduite de l'atelier*. Il n'est pas plus simple que celui des ordonnancements prévisionnels ou de traitements d'exceptions, et il s'y ajoute, en plus, un impératif : la réaction du pilote doit avoir lieu dans un laps de temps très court, éventuellement en temps réel.
La plupart de ces questions peuvent se ramener à une optimisation sur un ensemble fini.
Notre but étant de mettre au point et de tester de nouvelles méthodes. Nous envisageons d'étudier quelques cas particuliers d'ateliers théoriques, simplifiés et très étudiés.

2. Des problèmes d'ateliers classiques

Un atelier, formé de machines toutes différentes, est destiné à fabriquer des pièces qui doivent toutes passer une fois sur chaque machine pour y être usinées pendant une durée connue. L'ordre d'utilisation des machines est le même pour toutes les pièces. Une machine ne peut pas usiner plusieurs pièces à la fois et lorsqu'une pièce est sur une machine, elle n'en repart qu'après avoir été complètement usinée. Les moyens de transport sont tels que, dès que le traitement d'une pièce p sur une machine m est terminé, la pièce p peut instantanément entrer

sur la machine suivante (si elle est libre) ou, si la machine suivante est au travail, rester en attente, et la machine m peut commencer instantanément le traitement de la pièce suivante si celle-ci est prête. Le problème classique appelé flow-shop est le suivant : quel doit être l'ordre d'introduction des pièces pour que le travail total soit terminé le plus tôt possible ? En cas d'absence ou de limitation en capacités de stockage entre les machines, on obtient des variantes de ce problème. Nous faisons également intervenir, entre les machines, des temps de transport dont les performances sont soit déterministes soit stochastiques.

Nous envisageons aussi une autre classe d'ateliers théoriques, où l'on s'intéresse à des pièces à fabriquer disponibles après une date connue, devant ressortir de l'atelier avant une date également connue, après avoir été traitées dans l'atelier pendant une durée également connue. On cherche un ordre d'entrée des pièces permettant de les livre avec le moins de retard possible, ou bien en minimisant le temps de fonctionnement de l'atelier… Ces problèmes sont appelés *problèmes à une machine*. Nous nous intéresserons à des modèles d'ateliers plus proches de la réalité.

3. Des problèmes d'ateliers réels

Un atelier est constitué de machines, mais l'unique moyen de transport est une navette permettant de transporter les pièces de l'entrée de l'atelier sur une machine, d'une machine sur une autre, ou d'une machine à la sortie de l'atelier. Chaque pièce doit passer, durant un temps connu, sur des machines (pas nécessairement sur toutes), dans un ordre connu(fixé pour chaque pièce, mais pas nécessairement le même pour toutes, et une pièce peut repasser plusieurs fois sur la même machine). Il peut y avoir plusieurs machines susceptibles d'effectuer un même travail sur une pièce donnée et ce, pas nécessairement dans le même temps. La navette ne peut pas prendre en charge plus d'une pièce à la fois et chaque machine ne peut usiner plus d'une pièce à la fois. Enfin, une pièce, entrée dans l'atelier, ne peut pas en ressortir avant d'être totalement usinée, mais doit ressortir de la machine où elle se trouve, lorsqu'elle est usinée, dans un délai maximum connu (tolérance) et, si elle est prise en charge par la navette, elle doit être transportée à destination, la navette ne pouvant pas être un moyen de stockage puisqu'elle constitue le seul moyen de transport possible. La navette met, pour se déplacer, un temps connu. Cet atelier est susceptible de voir son fonctionnement bloqué, du fait de l'unicité du moyen de transport (une pièce ne doit pas demeurer sur une machine au delà du délai de tolérance, sans que la navette n'ait eu le temps de la prendre en charge), ou du manque de stockage intermédiaire (une pièce, sur la navette, ne peut pas être placée sur une machine qui est encore occupée).

On recherche quelles doivent être les dates d'introduction des pièces et la politique de gestion de la navette pour éviter tout blocage de l'atelier et effectuer le travail en un temps minimal. Ce problème s'appelle le problème du *job shop avec navette et sans stockage intermédiaire*. Nous citons une variante industrielle à stockage intermédiaire limité.

4. L'atelier flexible

Les petites et moyennes entreprises, dans leur ensemble et dans leur état actuel, ont peu de perspectives d'augmenter leur productivité, c'est à dire leur compétitivité dans les délais. Sauf à prendre un raccourci possible et récent qui métamorphoserait l'entreprise d'une manière globale, et pourrait la faire décoller d'un bloc, pour entrer tout droit dans l'univers du futur : la productique intégrée. Elle recouvre et fait bénéficier l'entreprise, même petite, de tout ce qu'apporte l'informatisation, de la conception d'un produit jusqu'à sa vente, en intégrant aussi bien la fabrication des différentes pièces, la facturation, la gestion du personnel, le stockage, le transport, etc. dans le processus informatique.

Tout commence avec la commande numérique au milieu des années soixante, dans la mécanique, plus précisément dans la machine-outil qui a joué le détonateur pour l'explosion de techniques nouvelles. Perceuses, fraiseuses, aléseuses, rectifieuses, les machines outils créent les pièces de notre univers industriel. A partir de blocs d'aciers ou débauches de fonte, d'aluminium, de titane, de cuivre…elles donnent forme aux écrous, aux vis, aux blocs-moteurs, aux arbres de transmission, engrenages, soupapes, carcasses de moteurs électriques.

La machine-outil, productrice des choses du monde contemporain, mère de tous les biens industriels, a été un pôle d'innovation. Elle a su réussir le mariage de la mécanique et de l'informatique. Elle s'est donné un cerveau, non pas pour faire des calculs, mais pour fabriquer des pièces.

L'atelier est dit "flexible" parce que, sans opérateurs, cet ensemble de centres d'usinage et de chariots transporteurs intelligents possèdent la souplesse, la " flexibilité" d'un artisan capable de passer d'une activité à une autre, de fabriquer une pièce unique aussi bien qu'une série de pièces et de s'assurer, sans moyens humains, que son travail est bien fait. Son but est de produire sans homme. On confie à l'informatique l'exécution de tâches banales.

Dans l'entreprise intégrée, informations, ordres et communication sont véhiculés par l'informatique en flot continu. De la conception du produit à son exécution par l'atelier et à son expédition, la mémoire de l'ordinateur se charge de remplacer plans et papiers. Toutefois l'homme garde la main et reste le pilote du réseau informatisé.

Une forme réduite de l'atelier flexible est la " cellule flexible" qui comporte une ou deux machines à commande numérique équipées d'un transporteur de pièces et d'un ou deux robots de chargements. Ces cellules autonomes travaillent 24 heures sur 24 sans opérateur (avec une surveillance intermittente, toutefois, et un réapprovisionnement en matière première).

La productique apporte à l'usine du futur l'abandon définitif de l'organigramme en forme de pyramide : les qualifications des travailleurs sont peu différentes les unes des autres, leurs salaires très voisins et les tâches répétitives et harassantes disparues.

Il n'y aura que deux niveaux hiérarchiques: plus de directeur, de chef d'atelier, de contremaître, d'ouvriers spécialisés, de manœuvres.

Certes , il y a beaucoup à faire avant d'arriver à la productique intégrée dans nos usines. Cet assemblage intelligent de l'informatique utilisé à divers postes, va demander réflexion, investissements, heures de travail pour créer les bonnes liaisons. Cela va bouleverser les habitudes, apporter des gênes, coûter du chiffre d'affaires pendant la transformation. Organisation et mentalités vont changer par la force des choses. Pour beaucoup, l'intégration de la productique reste un projet futuriste.

5. Les problèmes d'ordonnancement

Les problèmes d'ordonnancements fréquents en milieu industriel, se ramènent à rechercher les dates d'introduction de pièces dans un atelier, connaissant les ressources disponibles, les temps de transport, les dates de disponibilité, les dates échues pour les pièces, leur gamme d'usinage, la topologie de l'atelier…Le but est de minimiser une fonction coût qui peut être le temps moyen de séjour dans l'atelier, le temps total de séjour des pièces dans l'atelier, le total des retards, le retard moyen, l'avance-retard (totale ou moyenne). Dans le cas du multicritères, on désire optimiser simultanément plusieurs de ces fonctions. On peut également rechercher quand introduire, dans une production déjà ordonnancée, une pièce prioritaire en créant le moins possible de perturbations de l'une des fonctions à optimiser ci-dessus.

Nous nous intéresserons de préférence, aux ateliers ayant une gestion interne précise et où le problème d'ordonnancement se résume en le chois des dates d'entrée des pièces dans l'ateliers.

Le problème est de minimiser une fonction f, telle que pour toute donnée x, on sache calculer f(x), le domaine de définition X de f est un ensemble, celui des diverses dates d'introduction possibles des J pièces (ce domaine est a priori $\mathbb{R}^J$).

En se bornant aux dates vraisemblables (ce qui donne un intervalle, pour chaque date d'introduction, et conduit généralement à $[0, a]^J$), que l'on exprime dans une unité

correspondant aux délais de réaction de l'atelier et en particuliers de ses moyens de transport (la seconde, la minute…). Ce domaine, ensemble de recherche, se ramène finalement à un domaine inclus dans [0, 1, 2, …, n]^{J}, qui est donc fini. Souvent chaque pièce est introduite à partir de la date 0. Seul alors, l'ordre d'introduction des pièces importe.

En informatique, l'atelier peut être un ordinateur et la pièce un ensemble de programmes à exécuter, les ressources brutes (fixant la date à partir duquel un programme peut être exécuté) étant constituées de données en mémoire. Ces problèmes recouvrent un domaine plus vaste que celui de l'ordonnancement d'ateliers industriels qui se résume en la minimisation, sur un ensemble fini, d'une fonction (généralement compliquée et sans les propriétés générales permettant l'usage des théorèmes mathématiques, puis d'algorithmes efficaces).

Notre but est de disposer d'outils permettant d'automatiser le calcul de ces ordonnancements, car les méthodes manuelles risquent, fort vite, de se révéler trop lentes devant le raccourcissement, toujours en progrès, des temps de fabrication et de transport à l'intérieur de l'atelier, sans parler de ce que sont les problèmes appliqués à l'informatique. De plus s'il survient un impondérable (panne, rupture de stock, demande exceptionnelle, ou urgente), il faut pouvoir réagir suffisamment vite. Ceci nous conduit à rechercher des algorithmes permettant de minimiser sur une ensemble fini, une fonction dont, peu de choses sont connues, si ce n'est un algorithme de calcul pour chaque donnée. Nous aurons une exigence : pouvoir, en un temps précis, fournir un "bon" résultat.

Sauf dans les cas théoriques, la fonction à minimiser n'est pas connue exactement : personne ne sait précisément quel peut être le coût financier d'un retard (ce peut être aussi bien insignifiant que causer un préjudice énorme par la perte d'un important client…), ni d'une rupture de stock (qui peut aller jusqu'à nécessiter l'utilisation des moyens onéreux de réapprovisionnement : affrètement d'un avion ou au contraire, être sans conséquence immédiate) etc.

Nous nous attacherons à développer des outils de minimisation approchée, mais sûrs et assez rapides, plutôt que des outils exacts qui sont de toutes façons, généralement inapplicables en pratique.

Chapitre 2 : Formulation des problèmes d'ordonnancement

1. Les problèmes d'ordonnancement

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

Les problèmes d'ordonnancement sont caractérisés par trois ensembles.

L'ensemble de "n" tâches $T = \{T_1, T_2, ..., T_n\}$, l'ensemble de "m" machines $P = \{P_1, P_2, ..., P_m\}$ et l'ensemble de "s" ressources $R = \{R_1, R_2, ..., R_s\}$.

Ordonnancer c'est assigner les machines de l'ensemble P et des ressources de l'ensemble R aux tâches de l'ensemble T dans l'ordre de compléter toutes les tâches sous des contraintes imposées.

Il y a deux types de contraintes classiques. En tout instant chaque tâche est à exécuter au plus sur une machine et chaque machine n'est capable d'exécuter qu'une tâche à la fois.

1.1. Caractérisation des machines

Elles peuvent être en *" parallèles "*, faisant la même fonction, ou spécialisées dans l'exécution de certaines tâches (*"dédiées "*, dedicated). Dans ce cas, les machines sont disposées, en général, en séries. On distingue trois types de machines parallèles dépendant de leur vitesse.

Si toutes les machines ont la même vitesse d'exécution des tâches, les machines sont appelées *" identiques "* est le problème est noté (P).

Si les machines différent par leur vitesse d'exécution et la vitesse b_i de chaque machine est constante et ne dépend pas de l'ensemble des tâches, elles sont dites *" uniformes "* (Q).

Si les vitesses d'exécution des machines dépendent des tâches et sont différentes alors elles sont dites *" quelconques "* ou " différentes " (unrelated, R).

Dans le cas de machines ″ dédiées ″, le plus souvent sont en série, il y a trois modèles ou types d'exécution de tâches: le flow shop, l'open shop et le job shop.

Si dans un atelier donné il y a "M" machines et N travaux, si les opérations élémentaires ou tâches ne sont pas liées par un ordre particulier, on parle de problème d'open shop.

Dans ces problèmes, le nombre d'opérations de chaque tâche est égal au nombre de machines. Chaque opération utilisant une machine différente. Les opérations d'une même tâche ne sont

pas soumises à des contraintes de précèdence, par contre, il ne peut y avoir de chevauchements entre ces opérations.

Dans un flow shop, le nombre d'opérations de chaque tâche est égal au nombre de machines. Seulement il y a des contraintes de précédence de type chaîne entre les opérations d'une tâche: l'opération (i, j + 1) ne peut commencer avant la fin de l'opération (i, j).

Dans le cas du Job shop, il existe des contraintes de précédence entre les opérations d'une même tâche. Par contre, l'ordre d'utilisation de machines n'est pas le même pour toutes les tâches. En général le nombre d'opérations est plus grand que le nombre de machines.

Un autre type d'ordonnancement existe, le flow shop hybride où des machines parallèles sont en série. C'est une organisation du processus de production en série. Les produits fabriqués passent dans un premier temps dans une première cellule (des machines en parallèles), puis dans une seconde, etc.

En général dans de tel système les espaces tampons ou Buffers entre les machines sont supposés de capacité illimitée et un job après sa fin d'exécution sur une machine doit attendre avant que son exécution ne commence sur une autre machine. Si les espaces tampons sont de capacité nulle, les jobs ne peuvent pas attendre entre deux machines consécutives et la propriété de ″ sans attente ″ est admise.

Le diagramme de la figure ci-dessous schématise les différents types de machines.

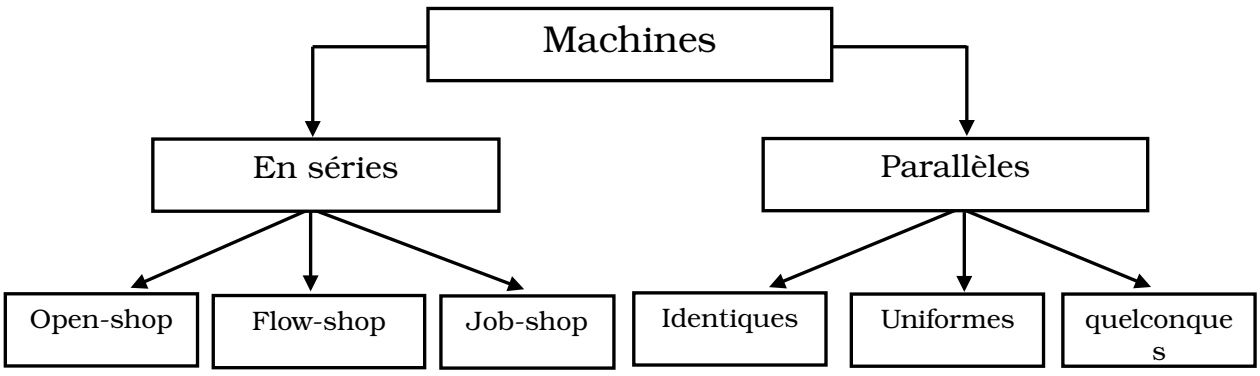


Figure 1. Types de machines.

1.2. Les Tâches

En général une tâche $T_j \in T$ est caractérisée par les données suivantes.

1. Le vecteur des temps d'exécution $p_j = [\, p_{1j}, p_{2j}, ..., p_{mj} \,]^T$, où p_{ij} est le temps qu'il faut pour la machine P_i pour exécuter la tâche T_j.

Dans le cas de machines ″ identiques ″ on a $p_{ij} = p_j$, i = 1,2,.., m.

Si les machines de P sont uniformes alors $p_{ij} = p_j / b_i$, i = 1, 2,.., m où p_j est un temps d'exécution standard (usuellement mesuré sur la machine la moins rapide) et b_i est le facteur vitesse d'exécution de la machine P_i .

Dans le cas de l'ordonnancement shop, le vecteur des temps d'exécution décrit les demandes en exécution de chaque tâche comprenant une ″ tâche ″. Pour une tâche $J_j = [p_{1j} , p_{2j}, ..., p_{nj}]^T$ où p_{ij} est le temps d'exécution de la tâche T_{ij} sur la machine correspondante.

Pour fixer les idées, supposons qu'un travail n'est formé que d'une seule tâche ou opération.

Alors à chaque tâche T_j on associe :

1. son temps d'exécution p_j

2. son temps d'arrivée ou temps de début d'exécution au plus tôt(ou ready time) r_j, le temps où la tâche T_j est prête pour l'exécution. Si les temps d'arrivée sont les mêmes pour toutes les tâches de T, alors on supposera que $r_j = 0$ pour tout j.

3. sa date de fin d'exécution au plus tard (ou due date) d_j . Si la tâche termine son exécution après cette date, elle encourt une pénalité. Elle est appelée "date de fin souhaitée".

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

5. un poids ou une priorité de la tâche(weight) w_j, qui exprime une urgence dans l'exécution de la tâche T_j.

6. Une date de fin d'exécution Cj (qui est en général une variable).

7. de_j : la date de début d'exécution de la tâche (qui est en général une variable).

8. son temps d'attente dans l'atelier W_j (a_j)

On définit le flow time de la tâche j, l'expression $F_j = c_j - r_j$. Il est aussi appelé " intervalle manufacturier ou temps d'atelier ".

Une relation entre ces paramètres existe. $r_j \leq c_j = p_j + W_j \leq d_j \leq \tilde{d}_j$.

On suppose qu'à toutes les tâches sont assignées les ressources nécessaires pour leurs exécutions, quand elles commencent à être exécuter, ou qu'elles attendent pour passer sur une autre machine ou quand elles sont interrompues ou terminent leur exécution.

Le décalage $L_j = C_j - d_j$ (Lateness). Le temps total durant lequel la tâche j est autorisée à rester dans l'atelier après son exécution.

Le retard $D_j = T_j = \max \{ C_j - d_j , 0 \}$ (Tardiness)

L'avance $E_i = \max (0 , - L_i)$ (Earliness)

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

Pour évaluer un ordonnancement on utilisera trois mesures de performance importante ou de critères d'optimalité.

1.3. Fonctions objectives d'un problème d'ordonnancement

Les fonctions économiques ou critères d'optimalité les plus utilisées font intervenir la durée totale de l'ordonnancement, le délai d'exécution, les retards de l'ordonnancement et le coût des stocks d'encours. La durée totale de l'ordonnancement notée C_{max} est égale à la date d'achèvement de la tâche la plus tardive : $C_{max} = \max c_j$. C'est la longueur de l'ordonnancement (schedule length ou makespan).

Le Flow time moyen $\overline{F} = \frac{1}{n} \sum_{j=1}^{n} F_j$ ou le flow time moyen pondéré $\overline{F}_w = \sum_{j=1}^{n} w_j F_j / \sum_{j=1}^{n} w_j$

Le critère, flow time, $\sum_{i=1}^{n} w_i C_i$ permet d'estimer le coût des stocks d'encours.

En effet la tâche " i " est présente dans l'atelier entre les instants r_i et C_i , et donc les stocks dont elle a besoin doivent être disponibles entre ces deux dates; d'où le coût $\sum_{i=1}^{n} w_i (C_i - r_i)$ est égale à une constante prés à $\sum_{i=1}^{n} w_i C_i$.

Dans beaucoup de problèmes, il faut respecter les délais, donc les dates au plus tard d_i ; on peut chercher à minimiser le plus grand retard $T_{max} = \max T_i$, ou bien la somme des retards $\sum_{i=1}^{n} T_i$, ou encore la somme pondérée des tâches en retard $\sum_{i=1}^{n} w_i T_i$.

Le décalage maximum $L_{max} = \max \{ L_j \}$.

D'autres critères peuvent être utilisés

Le retard moyen $D_{moy} = \frac{1}{n} \sum_{j=1}^{n} D_j$. Le retard moyen pondéré $D_w = \sum_{j=1}^{n} w_j D_j / \sum_{j=1}^{n} w_j$.

Le nombre de tâches en retard $U = \sum_{j=1}^{n} U_j$, où $U_j = 1$ si $C_j > d_j$ et 0 sinon.

Le nombre de tâches en retard pondéré $U_w = \sum_{j=1}^{n} w_j U_j$.

1.4. Quelques définitions

Un ordonnancement pour lequel la valeur d'une mesure de performance particulière γ est à son minimum sera appelé optimal, et la valeur correspondante de γ sera notée γ^*.
Un ″ algorithme d'ordonnancement ″ est un algorithme qui construit un ordre pour un problème donné Π.

Un ordonnancement est dit ″ préemptif ″ si l'exécution des tâches peut être interrompue en tout instant et peut être reprise plus tard, éventuellement sur une autre machine.
Si la préemption de toutes les tâches n'est pas permise l'ordonnancement est dit ″ non-préemptif ″.

Sur l'ensemble des tâches T, une relation d'ordre ou de précèdence apparaît souvent. Cette relation sera traduite dans un graphe orienté ou les sommets correspondent aux tâches et les arcs aux contraintes de précédence (appelé a task-on-node graph).
Une tâche $T_i < T_j$ signifie que l'exécution de la T_i doit se faire avant que T_j ne commence à s'exécuter. T peut être partiellement ordonner par une relation d'ordre <. Les tâches dans T sont dites liées ou dépendantes sinon les tâches sont appelées indépendantes.

Une tâche T_j est dite ″ disponible ″ à l'instant t si $r_j \leq t$ et tous ses prédécesseurs (en respectant les contraintes de précédence) ont terminé leur exécution à l'instant t.

Une mesure régulière est une fonction objectif à minimiser qui peut être exprimée en fonction des temps de fin d'exécution des travaux et qui croit si au moins un des instants de fin d'exécution croit. La moyenne ou le maximum des temps de fin d'exécution, le flow time, le décalage et le retard sont des mesures régulières.
Par contre, "la moyenne ou le maximum Earliness" ne sont pas régulières. La différence entre le plus grand et le second plus grand temps de fin d'exécution n'est pas une mesure régulière.
On a les résultats suivants : si $L_i = c_i - d_i = F_i - a_i$ où a_i est le temps d'attente de la tâche i alors:

$$\overline{L} = \frac{1}{n}\sum_{i=1}^{n} L_i = \frac{1}{n}\sum_{i=1}^{n} F_i - \frac{1}{n}\sum_{i=1}^{n} a_i = \frac{1}{n}\sum_{i=1}^{n} c_i - \frac{1}{n}\sum_{i=1}^{n} d_i = \overline{F} - \overline{a} = \overline{c} - \overline{d}$$

où $\overline{a}$, $\overline{r}$ et $\overline{d}$ sont des constantes.
Un ordonnancement qui est optimal pour $\overline{F}$ l'est aussi = $\overline{c} + \overline{L}$

De la formule $c_i = W_i + r_i + p_i$, on aura en moyenne $\overline{c} = \frac{1}{n}\sum_{i=1}^{n} W_i + \overline{r} + \frac{1}{n}\sum_{i=1}^{n} p_i$

Pour un problème donné, $\overline{r}$ et $\frac{1}{n}\sum_{i=1}^{n} p_i$ sont constantes.

Le temps d'attente moyen est aussi minimisé par l'ordonnancement qui minimise les temps de fin d'exécutions moyens $\overline{C}$.

On peut aussi écrire de la formule $L_i = T_i - E_i$ que $\overline{L} = \overline{T} - \overline{E}$.

On ne peut pas conclure qu'une règle qui minimise $\overline{L}$ minimise $\overline{T}$. Quelquefois cette assertion est vraie.

Si tous les jobs sont en retard, leur due-dates sont voisines tel que la valeur de Lateness soit positive, alors Lateness et Tardiness sont équivalentes.

Si tous les jobs ne sont pas en retard et sont terminés à temps, Earliness est la négation de la Lateness.

2. Classification des problèmes d'ordonnancement déterministes_

Les problèmes d'atelier sont représentés selon une classification (de Graham et al.[Gra66] et Blazewicz et al. [Bla94]) à trois champs $\alpha / \beta / \gamma$.

Le champ $\alpha = \alpha_1 \ \alpha_2$ décrit l'environnement de la machine.

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

$\alpha_1 = 1$, auquel cas nous avons un problème à une seule machine.

$\alpha_1 = P$: machines identiques parallèles.

$\alpha_1 = Q$: machines parallèles uniformes.

$\alpha_1 = R$: machines parallèles quelconques (unrelated).

$\alpha_1 = O$: il s'agit d'un open shop.

$\alpha_1 = F$: machines dédiées : système Flow shop.

$\alpha_1 = J$: machines dédiées : système Job shop.

$\alpha_1 = FH$: flow shop hybride

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

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

$\alpha_2 = k$: le nombre de machines est égal à k (k entier positif).

Le second champ $\beta = \beta_1 \beta_2 \beta_3 \beta_4 \beta_5 \beta_6 \beta_7 \beta_8$ décrit la tâche et les caractéristiques de la ressource.

Le paramètre $\beta_1 \in \{\varnothing, pmtn \}$ indique la possibilité de la préemption de la tâche.

$\beta_1 = \varnothing$: la préemption n'est pas autorisée.

$\beta_1 = pmtn$: la préemption est autorisée.

Le paramètre $\beta_2 \in \{\varnothing, res \}$ caractérise les ressources additionnelles.

$\beta_2 = \varnothing$: aucune ressource additionnelle n'existe.

β_2 = res : il y a des contraintes de ressources spécifiées.

Le paramètre $\beta_3 \in \{\varnothing$, prec, uan, tree, chains $\}$ reflète les contraintes de précédence.

β_3 = $\varnothing$, prec, uan, tree, chains : respectivement tâches indépendantes, contraintes de précédence générales, réseau d'activité uniconnexe, contraintes de précédence formant un arbre ou une chaîne.

$\beta_4 \in \{\varnothing, r_j\}$: les temps au plus tôt sont zéro, soit les temps au plus tôt différent par tâche.

$\beta_5 \in \{\varnothing, p_j = p, \underline{p} \leq p_i \leq \bar{p}\}$: les tâches ont des temps d'exécution arbitraires, le temps d'exécution dans l'intervalle défini.

$\beta_6 \in \{\varnothing, \tilde{d}\}$ décrit les deadlines: aucune deadline n'est supposée dans le système (date au plus tard peuvent être définie si le critère utilisé pour évaluer l'ordonnancement est celui de date au plus tard), deadlines sont imposées sur la performance de l'ensemble des tâches.

$\beta_7 \in \{\varnothing, n_j \leq k\}$: décrit le nombre maximum de tâches constituant un job dans le cas du système job shop: le nombre est arbitraire ou le problème d'ordonnancement n'est pas du job shop; le nombre de tâches pour chaque job n'est pas plus grand que k.

$\beta_8 \in \{\varnothing$, no-wait $\}$ décrit une propriété sans attente de l'ordonnancement sur des machines dédiées : la région tampon est de capacité illimitée; sans attente les capacités d'attente sont nulles. Une tâche qui termine son exécution sur une machine passe instantanément sur une autre machine.

Le troisième champs γ signifie le critère à utiliser.

$\gamma \in \{ C_{max} ; \sum_{j=1}^{n} C_j ; \frac{1}{n} \sum_{j=1}^{n} F_j ; \sum_{j=1}^{n} w_j F_j / \sum_{j=1}^{n} w_j ; L_{max} ; \frac{1}{n} \sum_{j=1}^{n} D_j ; \sum_{j=1}^{n} w_j D_j / \sum_{j=1}^{n} w_j ; \sum_{j=1}^{n} U_j ;$

$\sum_{j=1}^{n} w_j U_j$,…, etc. $\}$.

Exemples : P // C_{max} signifie : ordonnancement non-préemptif de tâches indépendantes de temps de service arbitraire , arrivant aux instants 0, sur des machines parallèles, identiques dans l'ordre de minimiser la longueur de l'ordonnancement.

O3 / pmtn , r_j / $\sum_{j=1}^{n} C_j$: signifie ordonnancement préemptif de longueurs de tâches arbitraires dans un open shop à trois machines, les tâches arrivent en des instants différents et l'objectif est de minimiser le flow time moyen.

3. Les méthodes de résolution d'un problème d'ordonnancement [3]

Le problème d'ordonnancement étant combinatoire, il est improbable de trouver un algorithme polynomial pour le résoudre dans le cas général. Pour des problèmes de grandes tailles, on ne cherche que des solutions approchées. Les algorithmes qui fournissent des solutions optimales ne s'appliquent qu'aux problèmes de tailles raisonnable. Les méthodes de résolution sont stochastiques et les méthodes déterministes qui sont les plus étudiées jusqu'à présent.

- *Les méthodes par construction :* complètent une solution partielle à chaque itération et génèrent généralement des ordonnancements qui ne sont pas optimaux. Les algorithmes de liste en sont partie. Ils consistent à construire une liste ordonnée de travaux selon un critère donné (durée d'exécution la plus faible d'abord (SPT), date de fin souhaitée la plus petite d'abord, …). Un des algorithmes les plus connus est la règle EDD [Jackson, 1955] qui permet de résoudre de façon optimale $1 / d_i / T_{max}$. C'est une construction progressive.

Plusieurs techniques sont utilisés pour résoudre le problème d'ordonnancement de s .

Nous présentons les techniques les plus employées.

- *Les méthodes par voisinage* travaillent sur des solutions complètes contrairement aux méthodes par construction . Il existe plusieurs méthodes de voisinage (méthodes de plus forte pente, les méthodes de descente, …)

- *Les méthodes par décomposition* décomposent le problème en un ensemble de sous ensembles. La définition du critère de décomposition permet de mettre en évidence plusieurs méthodes structurelles (décomposer l'ensemble des variables de décision en S_1, et S_2. Résoudre avec S_1 sachant S_2 et S_2 sachant S_1) temporelle ou spatiale.
- *Les méthodes par modification de contraintes* modifient les contraintes du problème, afin de restreindre l'ensemble des solutions faisables et de résoudre plus facilement le problème modifié. On peut aussi hiérarchiser les contraintes et ne retenir que celles qui semblent fondamentales de sorte que le problème soit " plus simple ", relaxation des contraintes d'intégrité, variables de décision.

- *Les méthodes employant des techniques issue de l'intelligence artificielle* à base de connaissance ou de l'apprentissage (learning). Elles sont basées sur la représentation des connaissances au moyen d'un ensemble de règles de production. Un moteur d'inférence contenant la stratégie de contrôle (comparable à un algorithme de résolution) est nécessaire.

4. Méthodes exactes

Elles recherchent un ordonnancement optimal minimisant un des critères présentés ou une combinaison de plusieurs critères. Les techniques utilisées sont les méthodes par séparation et évaluation, la programmation dynamique (voir chapitre 6), la déduction mathématique, la théorie des jeux, la théorie des graphes qui s'applique surtout à l'ordonnancement global avec placement statique des tâches appelé aussi hors-ligne où on cherche des partitions de coupe minimale pour minimiser les coûts de communication entre les sommets. Cette technique est coûteuse en temps de calcul, etc.

5. Les heuristiques

Elles sont souvent dynamique ou dites en-ligne. Leur inconvénient est qu'elle ne garantisse pas de solution optimale. Elles permettent de réduire considérablement le temps de recherche pour trouver une solution acceptable. Trois principales heuristiques dites Méta-heuristiques sont introduites au chapitre 8.

Le recuit simulé : Cette technique provient de l'observation de la formation d'une structure cristalline quand un métal refroidit. Le recuit simulé est une technique permettant de trouver, au bout d'un temps raisonnable, une solution pour des problèmes de complexité élevée. Elle est utilisée pour la détermination de la répartition statique des tâches. Elle est considérée comme technique stochastique et ne garantit pas de solution optimale.

Les algorithmes génétiques : Ils sont fondés sur les principes de l'évolution biologique des espèces. Comme le recuit simulé, ils permettent de trouver au bout d'un temps raisonnable des solutions pour des problèmes de complexité élevée. Ils sont facilement parallèlisables. Les algorithmes génétiques sont basés sur une notion étendue de voisinage. Plusieurs solutions sont maintenues simultanément et évoluent par des techniques de recombinaison et de mutation. Une des difficultés rencontrée est le codage des solutions.

Les réseaux de neurones : Ils ont été appliqués à la résolution approchée des problèmes d'optimisation en général et à d'ordonnancement en particulier. Leur avantage est la possibilité d'utiliser des circuits analogiques dont l'évolution converge vers des états stables correspondant à une solution approchée du problème d'ordonnancement. Grâce à leur vitesse

élevée de convergence, leur utilisation en-ligne peut être envisageable. Leur inconvénient est le blocage à la rencontre d'un minimum local.

La logique floue : Elle apporte des solutions pour les problèmes qu'on ne sait pas spécifier complètement et quantitativement. On associe un degré de satisfaction à la terminaison de chaque tâche si elle se termine avant sa date critique, la satisfaction est totale. Le degré d'insatisfaction augmente au fur et à mesure que l'on s'éloigne de cette date critique.

Pour l'ordonnancement stochastique, quand les temps d'exécution des tâches sont aléatoires, deux approches sont souvent utilisées : La théorie des files d'attentes et Les processus Bandits. On modélise les jobs comme étant des processus à décisions (en tout instant une tâche est soit en attente ou en exécution) semi-Markovien. Gittins et Jones (1972) étaient les pionniers à les appliquer aux problèmes d'ordonnancement. A chaque tâche est associée une fonction rang ou une priorité ou un indice d'allocation dynamique. En tout instant, on exécute la tâche avec le plus grand indice. On détermine même les instants d'interruption de tâches.

6. Propriétés des ordonnancements

Les ordonnancements peuvent être classés en au moins trois catégories : les semi-actifs, les actifs et les sans délai.

Ordonnancement semi-actif : si aucun travail, ne peut avancé sur la ressource où il se trouve, compte tenu des contraintes de gamme et de précédence. La longueur de l'ordonnancement correspond au chemin le plus long dans le graphe potentiel-tâche.

Ordonnancement actif : si aucune opération d'un travail i ne peut débuter son exécution plus tôt sans déplacer au minimum une autre opération.

Ordonnancement sans délai : lorsqu'aucune machine n'est laissée inoccupée, ceci alors qu'une file d'attente contient au moins un travail susceptible de débuter son exécution sur cette machine. Lorsqu'une méthode parcourt l'ensemble des ordonnancements sans délai, elle n'est pas capable de trouver la solution optimale, on ne laisse pas une machine inoccupée alors qu'une file contient des tâches.

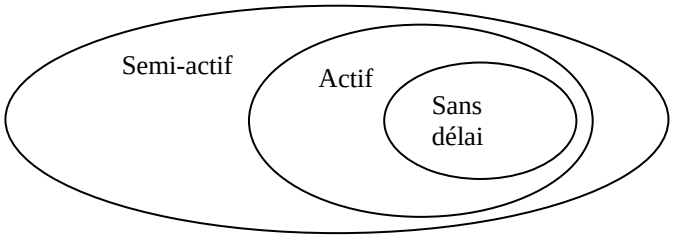


Figure 2. les catégories d'ordonnancement

Chapitre 3 : Complexité des algorithmes et des problèmes d'optimisation

1. Introduction

Les notions de complexité des problèmes sont très importantes, car si un problème est identifié comme complexe, il sera difficile d'en spécifier un modèle, on pourra même perdre espoir de trouver un algorithme pour le résoudre. Dans ce cas on se contentera d'exigences plus limitées : résolution approchée du problème posé, résolution d'un problème voisin plus simple, …Un problème d'optimisation combinatoire en abrégé POC, consiste à chercher le minimum s* d'une application "f" d'un ensemble S dans R le plus souvent à valeurs entières ou réelles, tel que f(s*) = $\underset{s\in S}{\text{Min}}\ f(s)$ (une même définition peut se donner en utilisant le maximum sachant que $\underset{s\in S}{\text{Max}}\ f(s) = -\ \underset{s\in S}{\text{Min}}\ -(f(s))$).

Les problèmes d'optimisation combinatoire se répartissent en deux catégories : ceux qui sont résolus optimalement par des algorithmes efficaces et rapides et ceux dont la résolution optimale peut prendre un temps exponentiel. Les problèmes de chemins, de flots et d'arbres disposent d'algorithmes efficaces, même certains problèmes d'ordonnancement qu'on verra dans les chapitres ultérieurs. Certains problèmes d'ordonnancement où les due-date ou les release-date apparaissent font partie de la catégorie des problèmes difficiles.

Pour évaluer et classer les divers algorithmes disponibles pour un problème, il nous faut une mesure de performance indépendante du langage et de l'ordinateur utilisé.

2. Diverses mesures de complexité

Nous présentons les critères permettant de définir la complexité d'un système. Les problèmes informatiques de l'industrie sont généralement difficiles à résoudre. Leur complexité peut être définie en deux catégories. Dans la première on s'intéresse au système en soi, et l'on mesure la complexité d'un problème, d'un système par le nombre de ses composants, par le degré d'interdépendance de ses composants. Les systèmes indécidables sont plus complexes que les systèmes déterministes. On peut prendre en compte la quantité d'information interne au système. Un système ayant plusieurs composants identiques est plus simples par rapport à un système ayant le même nombre de composants mais tous différents. On peut tenir compte de la géométrie du système en distinguant entre système discret et système continu. La complexité d'un problème peut se mesurer par celle de son domaine et des méthodes nécessaires à sa résolution ce qui conduit à :

a) tenir compte du nombre de paramètre du système
b) la complexité computationnelle, qui prend en compte le temps et la place nécessaire pour résoudre ce problème sur un ordinateur
c) on peut tenir compte de la complexité des démonstrations proposées, mesurées par les caractéristiques structurelles de leurs graphes.

Notons que certains systèmes complexes lorsque leur description est exacte, peuvent devenir moins complexes à un certain degré d'approximation. Souvent les problèmes intéressants se montrent difficiles. La complexité d'un problème peut être **spatiale** (les algorithmes connus demandent une place en mémoire exagérée) ou **temporelle** (le temps nécessaire à leur résolution qui est démesurée). Nous nous intéresserons à la complexité temporelle.

2. Comment mesurer l'efficacité d'un algorithme

Un algorithme de résolution d'un problème P donné est une procédure, décomposable en opérations élémentaires, qui transforment les données en résultats. La première idée pour comparer et évaluer des algorithmes est de les programmer, puis de mesurer leurs durées d'exécution. Le temps de calcul dépend trop de la machine, du langage de programmation, du compilateur utilisé pour le langage et les données. En pratique on compte le " nombre d'opérations caractéristiques " de l'algorithme à évaluer. Ce nombre ne dépend ni de la machine, ni du langage et peut s'évaluer sur du papier. L'évaluation dans certaines conditions s'appellent " complexité de l'algorithme ". Ce n'est pas la complexité de structure ou la difficulté de programmation. Nous établissons une relation entre la durée d'exécution de l'algorithme exprimée en termes du nombre d'opérations élémentaires, et la taille, exprimée en termes de caractères nécessaires pour coder les données. Quelques questions peuvent se poser: quelles sont les opérations élémentaires que nous allons prendre en compte ? Quel codage des données nous permettra de mesurer la taille du problème? Quelle est la nature de la relation liant la taille des données et le nombre d'opérations élémentaires ?

La taille d'une donnée est la quantité de mémoire pour la stocker. Il faudrait la mesurer en nombre de bits. La complexité d'un algorithme A est une fonction $C_A(N)$ donnant le nombre d'instructions caractéristiques exécutées par A **dans le pire des cas,** pour une donnée de taille N. Les opérations élémentaires sont en général, l'addition, la soustraction, la multiplication, la division, la comparaison, l'incrémentation etc. La complexité est basée sur la donnée qui va demander le plus du travail à l'algorithme. Une complexité moyenne se calcule en effectuant par exemples, des statistiques de temps de calculs sur un grand nombre de problèmes_tests.

La connaissance du pire des cas est critique pour des applications temps réel tel le contrôle aérien, le robotique, l'automatisme, les systèmes d'alarmes qui exigent un temps de réponse borné.

Une donnée d'un algorithme de chemin optimal dans un réseau ou graphe valué comprendra le nombre N de sommets, la liste des sommets, le nombre M d'arcs et la liste des arcs avec chacun d'eux, la valeur numérique de l'arc associée.

3. Ordre d'une fonction

Considérons deux algorithmes A_1 et A_2 pour un même problème de taille N, avec $C_{A1}(N) = N^2/2$ et $C_{A2}(N) = 5\ N$. L'algorithme A_2 fait plus d'opérations que A_1 pour N égal à une valeur moins de dix strictement mais moins dès que N est supérieur à dix. A_2 ira toujours plus vite à partir d'une certaine taille du problème. Cette idée de croissance s'exprime par l'ordre d'une fonction.

Définition 1. Soient f, g deux fonctions de dans . On dit que f est d'ordre inférieur ou égal à g, ou d'ordre au plus g, si on peut trouver un réel x_0 et un réel c tels que :

$$\forall\ x \geq x_0, \quad f(x) \leq c.\ g(x).$$

g devient plus grande que f à partir de x_0, à un facteur c près.

On l'écrit O(g), $f \in O(g)$ ou $f = O(g)$. f est un grand O de g.

Exemple 1 : $5\ N = O(0.5\ N^2)$ car $5\ N \leq 0.5\ N^2$ pour $N \geq 10$.

Les complexités des algorithmes sont souvent exprimées à l'ordre près.

On dit que f est polynomiale s'il existe un k réel, tel que f est un $O(N^k)$.

Exemple 2 : Soit à chercher si un entier x est présent dans un tableau $T = \{ T_1, T_2, \ldots, T_n \}$ de n entiers selon l'algorithme suivant :

```
i : = 1
Tant que ( i ≤ n ) et ( Ti ≠ x )
I : = i + 1
Ft
```

On peut trouver x bien sûr dans T_1, mais au pire x n'est pas dans T et l'algorithme va exécuter n fois le corps de la boucle. Soit n tests et n incrémentations de i. Un dernier concluant i > n

va provoquer la sortie. Le test $T_i \neq x$ est exécuté n + 1 fois au pire. On dira que cette recherche est en O(n).

Définition 2 : Un codage est raisonnable si le nombre d'emplacements nécessaires pour coder un entier N est égal à [log(N+1)] dans une base.

Exemple 3 : addition de deux nombres A et B. Ce n'est même pas un POC. En codage raisonnable, la taille des données est [log(A+1)] + [log(B+1)] + 1. En codage décimal le nombre d'opérations est borné par 3 n.

On peut proposer une façon d'additionner qui n'est pas polynomiale. Le codage est le même mais l'algorithme est : tant que A > 0 faire

A : = A - 1 et B : = B + 1;

Fin tant que.

A la fin de l'algorithme, B contient la somme cherchée. Le nombre d'opérations N est égale à A et dans le plus mauvais cas, si $A = 10^k$ et B = 0, on a $N = 10^k$. Cette algorithme n'est pas polynomial.

4. Bons et mauvais algorithmes

Une classification distingue les algorithmes polynomiaux, de complexité de l'ordre d'un polynôme, et les autres dits exponentiels.

Des exemples de complexités polynomiales sont : logn, $n^{0.5}$, n logn, n^2, …, n^5.

Une complexité exponentielle est une fonction e^n, 2^n, ou n^{logn} dite sous exponentielle, n! et n^n.

Un bon algorithme est polynomial.

Pour appréhender ce qu'est une croissance exponentielle, on dresse le tableau suivant. Les temps de calculs supposent une micro- seconde par instruction de haut niveau du CPU.

Taille → Complexité ↓	20	50	100	200	500	1000
$10^3 n$	0.02s	0.05s	0.1s	0.2s	0.5s	1s
$10^3 n \log_2 n$	0.09s	0.3s	0.6s	1.5 s	4.5s	10s
$100\ n^2$	0.04	0.25	1	4s	25s	2 mn
$10\ n^3$	0.02s	1s	10 s	1 mn	21 mn	27 h
N^{logn}	0.4s	1.1 h	220 jours	12500 ans	$5.\ 10^{10}$ ans	-

$2^{n/3}$	0.0001s	0.1s	2.7 h	$3 . 10^6$ ans	-	-
2^n	1s	36 ans	-	-	-	-
3^n	58 mn	2.10^{11}ans	-	-	-	-
n!	77.100ans	-	-	-	-	-

Les cases non remplies correspondent à des durées supérieures à 1000 milliards d'années.

La puissance de la machine ne résout rien pour des complexités exponentielles. Pour traiter un problème de taille 100 dans le même temps qu'un problème de taille 50, il suffit d'un ordinateur 10 fois plus puissant pour une complexité de 10. n^3 , 4800 fois plus puissant pour n^{logn} , 100.000 fois plus pour $2^{n/3}$. Ce qui est impossible actuellement.

Ces calculs montrent qu'il est faux de croire qu'un ordinateur peut résoudre tous les problèmes combinatoires. Il faut se méfier des méthodes énumératives consistant à examiner tous les cas possibles. Elles conduisent à des algorithmes exponentiels. L'énumération des parties d'un ensemble de n objets est en $O(2^n)$, celle des affectations de n objets à n autres est en O(n!), et n décisions successives avec k possibilités à chaque fois génèrent k^n possibilités.

Problème d'optimisation combinatoire

Nous rappelons qu'un problème d'optimisation combinatoire consiste à chercher le minimum s* d'une application "f " d'un ensemble S dans R tel que $f(s^*) = \min_{s \in S} f(s)$. Par exemple, mesurer des durées de tâches avec des nombres entiers de secondes offre une précision largement suffisante pour un problème d'ordonnancement industriel. " f " est appelée " fonction économique ou objectif ". Parfois on s'intéresse aux éléments de S vérifiant certaines contraintes, ce sont des solutions réalisables. Une donnée (S, f) est appelée un cas du POC ou instance.

Un problème d'existence (PE) appelé aussi problème de décision ou de reconnaissance consiste à chercher dans un ensemble fini S s'il existe un élément "s" vérifiant une certaine propriété P. Il peut être formulé par un énoncé et une question à réponse oui-non. On peut les considérer comme des problèmes d'optimisation particuliers en définissant comme fonction objectif f, f(s) = 0 si et seulement si s vérifie P.

Un grand nombre de problèmes peuvent être abordés par l'optimisation combinatoire. Il existe un " problème d'existence associé" à tout POC. Il suffit d'ajouter à la donnée S et f, un entier k et de définir la propriété P par " $f(s) \leq k$? " .

Les problèmes de la classe P et de la classe NP

Définition 3 : Un problème est dit de la classe P s'il peut être résolu par un algorithme polynomial. Il est dit faciles.

Sinon les problèmes sont dits difficiles.

Si le problème de reconnaissance associé à un problème POC donné est difficile alors le POC est dit difficile.

Définition 4 : Un algorithme non déterministe est muni (à l'inverse des déterministes) d'une instruction qui permette, chaque fois qu'elle est appliquée, de faire le bon choix.

Définition 5 : Un problème Π appartient à la classe NP si Π peut être résolu par un algorithme polynomial non déterministe. NP signifie non déterministe polynomial.

Supposons qu'on sait que la réponse à un problème de reconnaissance est vraie. Si on peut faire partager notre conviction à une autre personne en temps polynomial, alors le problème appartient à la classe NP, même si on ne sait trouver en temps polynomial une solution pour laquelle la réponse est vraie.

Un problème Π appartient à la classe NP est un problème qu'on peut résoudre par une exploration arborescente dont la profondeur, la longueur du plus long chemin de la racine à un sommet pendant, soit une fonction polynomiale de la taille des données.

On conjecture que P ≠ NP . La classe NP contient des problèmes qui sont plus difficiles que ceux de la classe P si toutefois P ≠ NP.

Exemple 3 : a) Le problème de satisfaisabilité ou SAT est dans la classe NP

Soi X = { x_1, x_2, …, x_n } un ensemble où $x_i \in$ { 0, 1 }, ∀ i = 1, …, n et soit l'expression booléene formée de " m " clauses C_j (j = 1, …, n). Une clause et une conjonction et disjonction des variables et de leur complémentaires. Soit $E = C_1 \cap C_2 \cap \ldots \cap C_n$.

Le problème de satisfaisabilité consiste à trouver une affectation des variables x_k, k = 1,…, n tel que E (x_1 , …, x_k) = 1 ou vraie.

Il y a des problèmes qu'on ne peut même pas prouver que c'est dans la classe NP. C'est le problème du kième sous ensemble.

Soit E, un ensemble fini de cardinal n. Une application c de E dans , un entier k, $k \leq 2^n$

Et un entier b quelconque.

On associe à chaque sous-ensemble E' de E, une valeur c(E') = $\sum_{e \in E'} c(e)$.

Le problème consiste à déterminer au moins " k " sous-ensembles E' distincts tel que c(E) ≤ b.

b) Le problème du 3-partition

Soit A = { a_1, …, a_{3t}} à 3t éléments. Une borne B appartient à et une taille s(a) de tel que

(B/4) < s(a) < B/2 pour tout a ∈ A et $\sum_{a \in S_i} s(a) = \sum_{j=1}^{3t} a_j = t\,B$.

Peut-on partitionner A en 3 sous ensembles disjoints S_1, S_2 et S_3 tel que ∀i, 1≤ i ≤ 3,

$$\sum_{a \in S_i} s(a) = B \ ?$$

De la même façon , il peut être défini le problème de la k-partition, k > 3.

Les problèmes NP complets

Pour la plupart des problèmes de la classe NP, on ne sait pas dire s'ils peuvent ou n peuvent pas être résolus par un algorithme polynomial.

Définition 6 : Soient P_1 et P_2 deux problèmes de reconnaissance. On dit que P_1 se réduit en temps polynomial à P_2 s'il existe un algorithme pour P_1 qui fait appel comme un sous programme, à un algorithme de résolution de P_2 et si cet algorithme de résolution de P_1 est polynomial lorsque la résolution de P_2 est comptabilisée comme une opération élémentaire.

Définition 7 : Un problème de reconnaissance (PE) est dit " NP-complet " si tout problème de la classe NP peut se réduire polynomialement à lui.

Théorème 1 : Si un problème P se réduit en temps polynomial à P' et si P' peut être résolu par un algorithme polynomial, il en est de même de P.

La propriété de NP-complet signifie que si on avait un algorithme polynomial pour un problème NP-complet alors on aurait un algorithme polynomial pour tous les problèmes de la classe et dans ce cas P = NP.

Cook (1970) a jeté les fondements de la complexité des algorithmes.

Théorème 2 : Si un problème P est NP-complet et si on peut mettre en évidence une réuction polynomial de P à P', alors P' est NP-complet.

Un problème P quelconque est dit " NP-dur " s'il existe une réduction polynomial du problème de satisfaisabilité à P. Les problèmes NP-durs sont des problèmes au moins aussi difficile que les problèmes NP-complets. Le problème du kième sous ensemble est NP-dur. Les POC tel que le voyageur de commerce ou le sac à dos dont le problème de reconnaissance associé est NP-complets sont NP-durs.

Exemples de problèmes NP-complets

Les problèmes faciles sont les problèmes d'exploration de graphes (O(M)), de chemin de coût minimal (O(NM) Bellman, Dijsktra $O(N^2)$), du flot maximal (Ford & Fulkerson, $O(NM^2)$), de l'arbre recouvrant de poids minimal (Prim, $O(N^2)$, Kruskal, O(MlogN)), les couplages de cardinal maximal, les parcours eulériens ou chinois (O(M), le test de planarité (O(N), Hopcroft & Tarjan), le test du bipartisme, la recherche d'une information parmi N, la programmation linéaire (Kachian (1979), Karmakar (1984), $O(n^{3.5}L)$, L le nombre de bits pour coder A, b et c du programme)).

Les problèmes sans algorithmes polynomiaux sont celui du stable maximal, le transversal minimal, la clique maximal, la coloration minimale, les problèmes hamiltoniens, le problème d'empilement, du Bin Packing etc.

En pratique, on résout un problème en tirant parti :

- de la taille des données. L'énumération complète peut être valable sur des petits cas.
- de la complexité moyenne : un algorithme exponentiel sur son pire cas peut être assez rapide en moyenne, comme l'algorithme du simplexe.
- de la nature des données : le problème peut devenir plus facile sur les cas particuliers.
- des méthodes diminuant la combinatoire : énumération intelligente, programmation dynamique.
- des méthodes approchées ou heuristiques : si on accepte des solutions de bonnes qualité sans garantie d'optimalité.

Définition 8 : On se donne un hypergraphe $H = (S, \Im)$ où $S = \{ s_1, \ldots, s_m \}$ un ensemble de m éléments et une famille $\Im = \{ E_1, E_2, \ldots, E_n \}$ de n sous-ensembles non vides de S recouvrant S, $\bigcup_{j=1}^{n} E_j = S$. Un coût $c_j = c(E_j)$ est associé à chaque sous ensemble E_j de $\Im$.

On appelle **problème de recouvrement** le problème de la recherche d'une sous famille de $\Im$ qui recouvre l'ensemble S et minimisant le coût total.

Pour le cas du problème de partition, on impose de plus que cette famille forme une partition de S(les sous ensembles E_j de sont disjoints deux à deux).

Soit A = (a_{ij}) la matrice d'incidence de l'hypergraphe H définie par $a_{ij} = \begin{cases} 1 \text{ si } s_i \in E_j \\ 0 \quad \text{sinon} \end{cases}$

Si on représente un sous ensemble de $\Im$ par son vecteur caractéristique X = (x_1, …, x_n)

Où $x_j = \begin{cases} 1 \text{ si } E_j \in \Im \\ 0 \quad \text{sinon} \end{cases}$

X représentera : un recouvrement si et seulement si

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1 \quad (i = 1, \ldots, m)$$

$x_j \in \{ 0, 1\}$ (j = 1, …, n)

un partitionnement si et seulement si

$$\sum_{j=1}^{n} a_{ij} x_j = 1 \quad (i = 1, \ldots, m)$$

$x_j \in \{ 0, 1\}$ (j = 1, …, n)

Le problème de recouvrement de coût minimum se formalise comme un PLNE

$$\min z = \sum_{j=1}^{n} c_j x_j$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1 \quad (i = 1, \ldots, m)$$

$x_j \in \{ 0, 1\}$ (j = 1, …, n)

Exemple 4 : Affectation produits-machines

Un atelier contenant m machines (i = 1, …, m) doit fabriquer n produits (j = 1, …, n).

Chaque produit j peut ou bien être entièrement fabriqué par la machine i ou bien il ne peut pas l'être. On appelle a_{ij} le temps pris par la machine i pour fabrique le produit j.

b_i : le temps total de travail de la machine i.

c_{ij} : le coût de fabrication du produit j par la machine i.

On cherche l'affectation des produits aux machines minimisant le coût total.

Ce problème se modélise comme celui de l'affectation généralisé qui s'écrit sous la forme suivante:

$$\min z = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}\, x_{ij}$$

$$\sum_{j=1}^{n} a_{ij} x_{ij} \le b_i , \qquad i = 1,\ldots, m$$

$$\sum_{j=1}^{n} x_{ij} = 1 , \quad j = 1,\ldots, n$$

$x_{ij} \in \{ 0, 1\}$, i = 1, …, m et j = 1, …, n ; $a_{ij} \ge 0$ et $b_i \ge 0$.

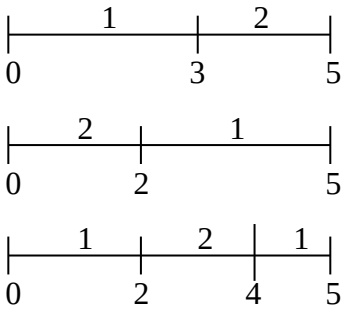
Chapitre 4 : Les problèmes d'ordonnancement à une seule machine
a) cas de tâches indépendantes

Les résultats sur les problèmes d'ordonnancement sont trop nombreux pour pouvoir être tous décrits. Nous nous limitons à quelques cas particuliers en donnant des idées des méthodes employées et des résultats que l'on peut obtenir. C'est les problèmes qu'on sait résoudre polynomialement et où les tâches sont indépendantes. En général les problèmes deviennent difficiles si aux tâches sont associés des release-date ou des due-date ou des deadlines.

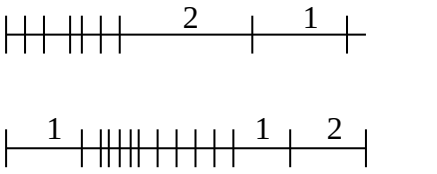
1. Les ordonnancements de permutation

Deux techniques d'ordonnancement qui ont une utilité sont la preemption où l'exécution de la tâche est interrompue (avec resume ou sans resume) et qu'une tâche quitte la machine sans avoir terminée son exécution et l'insertion du temps d'oisiveté où la machine est laissée oisive bien qu'un travail prêt est en attente d'être exécuté.

Exemple 1 (de preemption et du temps d'oisiveté. Deux jobs sur une machine):



preemption



Insertion d'un temps d'oisiveté

Nous énonçons quelques théorèmes.

Théorème 1 : Dans un problème n / 1 / - avec une mesure de performance régulière, il n'est pas nécessaire de considérer les ordonnancements qui impliquent aussi la preemption ou l'insertion du temps d'oisiveté.

Preuve : évidente.

Théorème 2 : Soit le problème n / 1 / $\frac{1}{n}\sum_{i=1}^{n} F_i = \frac{1}{n}\sum_{i=1}^{n} c_i - r_i$. Le mean flow time est minimisé en ordonnançant les tâches dans l'ordre croissant des temps d'exécution $P_{[1]} \leq P_{[2]} \leq .... \leq P_{[n]}$ appelé aussi selon la règle SPT (short processing time first) et maximisé en ordonnançant les tâches dans l'ordre décroissant des temps d'exécution $P_{[1]} \geq P_{[2]} \geq .... \geq P_{[n]}$ appelé aussi la règle LPT (longest processing time first)

Preuve : $\overline{F} = \frac{1}{n}\sum_{i=1}^{n} F_i = \frac{1}{n}\sum_{i=1}^{n} c_i - r_i = \frac{1}{n}\sum_{i=1}^{n} (w_i + P_i) = \overline{W} + \frac{1}{n}\sum_{i=1}^{n} P_i$

Minimiser $\overline{F}$, c'est minimiser $\overline{W}$ puisque $\frac{1}{n}\sum_{i=1}^{n} P_i$ est un terme constant quelque soit l'ordre d'exécution des tâches. Il suffit de choisir les plus courts temps d'exécution les premiers. La démonstration se poursuivra en résonnant par absurde et en considérant un autre ordonnancement qui ne suit pas SPT pour au moins deux tâches consécutives et en les comparant. On montre qu'on peut améliorer l'ordo non SPT en permutant les deux tâches qu'on a supposé non SPT.

Généralisation :

La règle SPT minimise $\frac{1}{n}\sum_{i=1}^{n} c_i$, $\max_i w_i$ et $\overline{T} = \frac{1}{n}\sum_{i=1}^{n} T_i = \frac{1}{n}\sum_{i=1}^{n} \max(c_i - d_i, 0)$.

LPT maximise tout ce que SPT minimise.

La règle SPT ne minimise pas toutes les fonctions objectifs. Elle ne minimise pas l'ordonnancement à due-date.

Exemple 2 :

Job	P_i	d_i
A	**1**	**3**
B	**2**	**2**

On considère soit AB ou BA. SPT sélectionne AB mais BA a un maximum lateness et un maximum tardiness, mean tardiness plus petit ($\max L_i = c_i - d_i), \max T_i = \max(c_i - d_i, 0)$, $\overline{T} = \frac{1}{n}\sum_{i=1}^{n} T_i = \frac{1}{n}\sum_{i=1}^{n} \max(c_i - d_i, 0)$).

Remarque 1 : Si les due-dates sont proches tel que tous les jobs en retard SPT minimise le mean tardiness, le mean lateness.

Si $d_1 = d_2 = \ldots = d_n$, SPT minimise le nombre de jobs en retard.

Par contre, montrer sur le même exemple que SPT ne minimise pas la variance du flowtime de jobs définie par var F = $\frac{1}{n}\sum_{i=1}^{n}(F_i - \bar{F})^2 = \frac{1}{n}\sum_{i=1}^{n}F_i^2 - \bar{F}^2$. SPT minimise chacun des termes de cette expression différence mais ne minimise pas la différence.

Exemple 3: Soit le problème 7 / 1 / $\overline{F}$ dont les données sont:

Job	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$\mathbf{P_i}$	6	4	8	3	2	7	1

L'ordonnancement SPT est (7, 5, 4, 2, 1, 6, 3)

Pour calculer $\overline{F}$ on remarque que

$F_{i(1)} = 1$

$F_{i(2)} = 1 + 2$

$F_{i(3)} = 1 + 2 + 3$

$F_{i(4)} = 1 + 2 + 3 + 4$

$F_{i(5)} = 1 + 2 + 3 + 4 + 6$

$F_{i(6)} = 1 + 2 + 3 + 4 + 6 + 7$

$F_{i(7)} = 1 + 2 + 3 + 4 + 6 + 7 + 8$

$$\frac{1}{7}\sum_{k=1}^{7}F_{i(k)} = \frac{1}{7}(7x1 + 6x2 + 5x3 + 4x4 + 3x6 + 2x7 + 8) = 12 \text{ x } \frac{6}{7}$$

En général pour un ordonnancement SPT, $\overline{F} = \frac{1}{n}\sum_{k=1}^{n}(n - k + 1)P_{i(k)}$

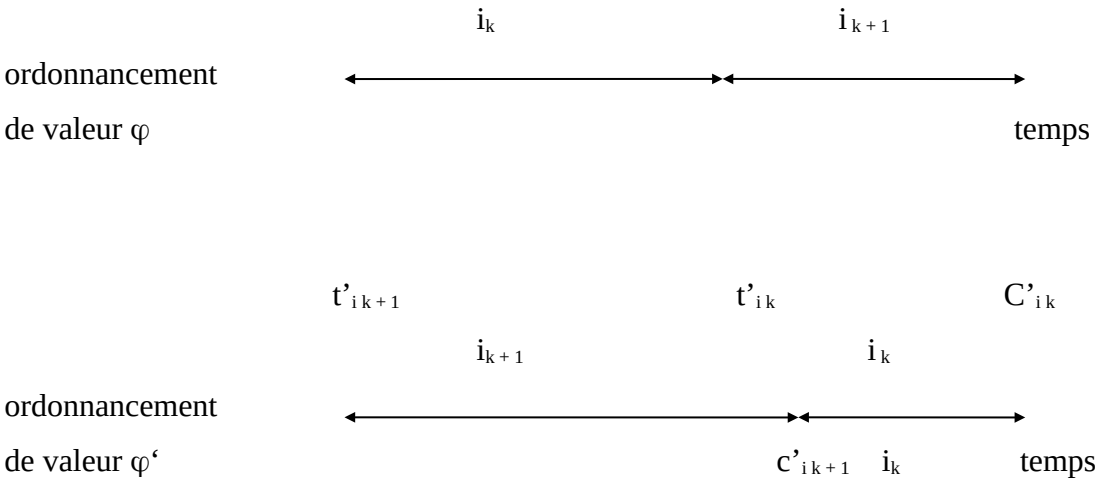
Proposition 1 : le problème 1 // $\sum_{i=1}^{n} w_i C_i$ est facile.

Soit (i1 , i2 , .., in) une permutation de T et intervenant deux tâches consécutives ik et ik+1 .

Cette méthode est appelé " Adjacent pairwise interchange property ".

Notons φ' la valeur du critère obtenue

$t_{i\,k}$ $C_{i\,k}$ $C_{i\,k+1}$



$$\varphi' - \varphi = w_{ik}\,(c'_{ik} - c_{ik}) + w_{ik+1}\,(c'_{ik+1} - c_{ik+1}) = w_{ik}\,p_{ik+1} - w_{ik+1}\,p_{ik}$$

$$\varphi' < \varphi \quad \Leftrightarrow \quad \frac{w_{ik}}{p_{ik}} < \frac{w_{ik+1}}{p_{ik+1}}.$$

Si on note $\rho_i = \frac{w_i}{p_i}$, appelé indice de priorité de la tâche i ,

alors : $\varphi' < \varphi \quad \Leftrightarrow \quad \rho_{ik+1} > \rho_{ik}$. Plus ρ_i est grand, plus la tâche i est prioritaire.

Règle de Smith

En l'absence de contrainte d'antériorité, nous ordonnons les tâches dans l'ordre des ρ_i non croissants et on obtiendra une permutation (i^*_1 , i^*_2 , .., i^*_n) tel que $\rho^*_{i1} \geq \rho^*_{i2} \geq .. \geq \rho^*_{in}$,

où $\rho_i = \frac{w_i}{p_i}$. Cette permutation est optimale.

Proposition 2 (Rothkopt(1966)) :

Si les temps de service sont aléatoires de moyennes connues,

E ($\sum_{i=1}^{n} w_i C_i$) est minimisée par un ordonnancement non-préemptif dans l'ordre décroissant des w_i / E C_i.

Preuve : identique à celle de Smith en passant à l'espérance.

2. Les problèmes avec les due-dates. Résultats élémentaires

Le critère considéré est celui du décalage (Lateness) $L_i = c_i - d_i$, la différence entre la date de fin d'exécution et la date de fin d'exécution au plus tard.

Théorème 3 : Le décalage moyen $\overline{L} = \frac{1}{n}\sum_{i=1}^{n} L_i$ est minimisé par la règle SPT.

Preuve : $\overline{L} = \frac{1}{n}\sum_{i=1}^{n} L_i = \frac{1}{n}\sum_{i=1}^{n} (c_i - d_i) = \overline{F} - \overline{d}$ où $\overline{d} = \frac{1}{n}\sum_{i=1}^{n} d_i$ qui est une constante.

Or SPT minimise $\overline{F}$ donc elle minimise $\overline{L}$.

Supposons qu'on peut ordonner les jobs selon les dates de fin au plus tard dans le sens croissant ce qui est $d_{[1]} \leq d_{[2]} \leq .... \leq d_{[n]}$

Théorème 4 (Jackson (1955)) :

Soit les problèmes n / 1 / L_{max} et n / 1 / T_{max}. Le maximum job lateness et la maximum job tardiness sont minimisés en ordonnançant les tâches dans l'ordre croissant des due-dates (EDD sequencing : Earlest due date).

Preuve : en utilisant l'échange de deux tâches et en comparant les ordonnancements obtenus.

Ce théorème peu s'énoncer comme l'algorithme suivant :

1) Commencer par une séquence de tâches non EDD
2) Déterminer une paire de tâches adjacentes i et j tel que $d_i > d_j$ où j suit i
3) Permuter i et j
4) Répéter l'étape 2 jusqu'à obtenir une séquence EDD.

En chaque étape L_{max} diminue ou reste inchangé.

De même pour le $T_{max}(S) = \max \{ 0, L_{max}(S)\}$

Exemple 4 : résoudre n / 1 / T_{max}

Job	**1**	**2**	**3**	**4**	**5**	**6**
Due date d_i	7	3	8	12	9	3
Processing time P_i	1	1	2	4	1	3

Une séquence EDD est (6, 2, 1, 3, 5, 4). On peut calculer T_{max} par une forme tabulaire.

Job $J_{i(k)}$	Completion time	Lateness	Tardiness

	$C_{i(k)} = \sum_{l=1}^{k} P_{i(l)}$	$L_{i(k)} = c_{i(k)} - d_{i(k)}$	$T_{i(k)} = \max (0, L_{i(k)})$
6	3	0	0
2	4	1	1
1	5	- 2	0
3	7	- 1	0
5	8	- 1	0
4	12	0	0

La permutation (2, 6, 1, 3, 5, 4) est aussi optimale.

On définit le temps d'écart (slack time) du job i à l'instant t par $d_i - P_i - t$. C'est le temps qu'il reste avant que le job i ne commencera son exécution s'il ne veut pas être en retard. La tâche avec le minimum de " slack time " offre le plus grand risque d'être en retard et devra être exécuter le plus tôt. L'instant de décision t est le même pour toutes les tâches, on doit ordonnancer les tâches dans l'ordre des $d_{[1]} - P_{[1]} \leq d_{[2]} - P_{[2]} \leq .... \leq d_{[n]} - P_{[n]}$

Théorème 5 : Dans le problème n / 1 / ---, le minimum job lateness L_{min} et le minimum job tardiness T_{min} sont maximisés en ordonnançant les tâches selon l'ordre croissant des " temps d'écarts ".

L_{min} n'est pas une mesure régulière.

		Job				$\overline{T}$	T_{max}	T_{min}
		1	2	3	4			
Due date	d_i	1	2	4	3			
Processing time	P_i	2	4	3	1			
SPT sequence	c_i	3	10	6	1			
4, 1, 3, 2	T_i	2	8	2	0	3.0	8	0
Due date sequence	c_i	2	6	9	10			
1, 2, 3, 4	T_i	1	4	5	4	3.5	5	1
Slack time sequence	c_i	6	4	9	10			
2, 1, 3, 4	T_i	5	2	5	4	4	5	2
Un autre ordonnancement	c_i	2	10	5	6			
1, 3, 4 , 2	T_i	1	8	1	0	2.5	8	0

Remarque 2 : On peut minimiser par SPT le mean tardiness, minimise par EDD le maximum tardiness et maximise par le slack-time le minimum tardiness mais aucune de ces règles ne minimise le mean tardiness.

Si le critère est le mean flow time alors on a :

Théorème 6 : (de Smith 1956)

S'il existe un ordonnancement tel que le maximum job tardiness est zéro, alors il existe un ordre sur les jobs avec le job k placé dans la dernière position lequel minimise le mean flow time si et seulement si : a) $d_k \geq \sum_{i=1}^{n} P_i$

b) $P_k \geq P_i \quad \forall\, i \text{ avec } d_i \geq \sum_{i=1}^{n} P_i$

Ce théorème stipule qu'une tâche "k" peut être mise à la dernière position d'un ordonnancement si cela ne la fait pas retardée et si la tâche a le plus grand temps d'exécution parmi l'ensemble de toutes les tâches qui peuvent être dans la dernière position sans être en retard. C'est une règle SPT soumise aux due dates.

Exemple 5 :

Job	**1**	**2**	**3**	**4**	**5**	**6**
Due date d_i	24	21	8	5	10	23
Processing time P_i	4	7	1	3	2	5

L'ordonnancement due -date (4 3 5 2 6 1) donne un maximum tardiness nul.

$\sum_{i=1}^{6} P_i = 22$. Les jobs 1 et 6 vérifient les conditions d'application du théorème. d_1 et d_6 sont supérieures à 22. Une des tâches est à mettre en dernière position. Le job 6 est sélectionné car $P_6 > P_1$. On exclue le job 6 et on recalcule $\sum_{i=1}^{5} P_i$ pour le reste des tâches à exécuter.

$\sum_{i=1}^{5} P_i = 17$. Les jobs 1 et 2 sont candidats. Le job 2 sera sélectionné, etc.

On obtiendra l'ordonnancement optimal (3 4 5 1 2 6).

3. Minimisation du nombre de tâches en retard N_T

Si la règle EDD donne un nombre nul de tâches en retard ou donne une seule tâche en retard alors elle est une règle optimale pour n / 1 / N_T . Si elle donne plus d'une tâche en retard, EDD peut ne pas être optimale. Un algorithme efficace pour le cas général est celui de Hodgson et Moore. La méthode suppose qu'une suite optimale est de la forme

a) un ensemble (E*) de tâches en avance dans l'ordre EDD
b) un ensemble L* de tâches en retard, ordonnancées dans n'importe quel ordre.

Algorithme 1 : (de Moore et d'Hodgson (1968))

Etape 1 : ordonnance les tâches suivant la règle EDD et soit ($J_{i(1)}, J_{i(2)}, \ldots, J_{i(n)}$) la permutation de tâches ainsi obtenue tel que $d_{i(k)} \leq d_{i(k+1)}$, k = 1, 2, …, n-1.

Etape 2 : Déterminer la première tâche en retard, soit $J_{i(l)}$ dans la suite actuelle.

Si aucune tâche n'est en retard aller à l'étape 4.

Etape 3 : Déterminer la tâche dans la sous permutation ($J_{i(1)}, J_{i(2)}, \ldots, J_{i(l)}$) avec le plus grand temps d'exécution et la rejeter de la séquence. Aller à l'étape 2 avec une séquence actuelle qui a moins d'une tâche qu'auparavant.

Etape 4 : former un ordonnancement optimale en prenant la sous séquence actuelle et en rajoutant les tâches rejetées qui peuvent être ordonnancer dans n'importe que ordre.

Exemple 6 :

Job	**1**	**2**	**3**	**4**	**5**	**6**
Due date d_i	15	6	9	23	20	30
Processing time P_i	10	3	4	8	10	6

On forme la séquence EDD et on calcule le temps de fin d'exécution des tâches jusqu'à ce qu'une tâche soit trouvée en retard (étape 1 et 2 de l'algorithme)

La séquence actuelle	**2**	**3**	**1**	**5**	**4**	**6**
Due date d_i	6	9	15	20	23	30
Processing time P_i	3	4	10	10	8	6
Temps de fin d'exécution	3	7	17			

La tâche 1 est la première en retard et a le plus grand temps d'exécution de la sous séquence (2 3 1). On rejette la tâche 1 (étape 3). On retourne et on répète l'étape 2 avec la nouvelle séquence actuelle.

Nouvelle séquence actuelle	**2**	**3**	**5**	**4**	**6**	**Tâches rejetées**
Due date d_i	6	9	20	23	30	**1**
Processing time P_i	3	4	10	8	6	
Temps de fin d'exécution	3	7	17	25		

La tâche 4 est la première tâche en retard dans cette séquence (2, 3 , 5, 4). La tâche 5 a le temps d'exécution le plus grand. La tâche 5 est rejetée (étape 3). On revient à l'étape 2, et on trouve plus de tâches en retard.

Nouvelle séquence actuelle	**2**	**3**	**4**	**6**	**Tâches rejetées**
Due date d_i	6	9	23	30	**1, 5**
Processing time P_i	3	4	8	6	
Temps de fin d'exécution	3	7	15	21	

D'où on passe à l'étape 4 et on forme la séquence optimale (2 3 4 6 1 5) ou (2 3 4 6 5 1).

On pouvait présenter les calculs dans un autre tableau plus compacte.

Séquence EDD	**2**	**3**	**1**	**5**	**4**	**6**	**Tâches rejetées**
Due date d_i	6	9	15	20	23	30	
Processing time P_i	3	4	10	10	8	6	
Temps de fin d'exécution	3	7	17				1
Temps de fin d'exécution	3	7	*	17	25		5
Temps de fin d'exécution	3	7	*	*	15	21	

Théorème 7 : L'algorithme de Moore et Hodgson fournit un ordonnancement optimal pour le problème n / 1 / N_T.

Preuve: à faire.

4. Minimisation du retard moyen $\overline{T}$ (mean tardiness)

L'algorithme de Wilkerson-Irwin utilise une règle simple. Il utilise deux listes (procédure heuristique)

Liste 1 de tâches ordonnancées : qu'on peut changer

Liste 2 de tâches non ordonnancées : ordonnées selon EDD.

A chaque étape, la première tâche sur la liste 2 des non ordonnançables est appelée " tâche pivot" qui est à déplacer.

Soit α : l'indice de la dernière tâche de la liste1 des ordonnançable

β : l'indice de la tâche pivot

γ : l'indice de la première tâche sur la liste des non ordonnançables.

Il utilise la permutation de deux tâches consécutives i et j. A chaque étape, l'algorithme applique la règle ci dessous aux tâches β et γ où $d_\beta \leq d_\gamma$.

Algorithme 2 : (de Wilkerson-Irwin, une heuristique)

Etape 1 : (Initialisation) Placer toutes les tâches dans une liste non ordonnançable selon EDD. Soient a et b les deux premières tâches sur la liste.

Si max { P_a, P_b} $\leq$ max { d_β, d_γ }, alors attribue la première position dans la séquence à la tâche avec le " earlier due date " sinon attribuez la première position à la tâche qui a le plus petit temps d'exécution. La tâche attribuée devient α et l'autre tâche β, la tâche pivot.

Etape 2 : si F_α + max { P_β, P_γ }$\leq$ max { d_β, d_γ }, ou si $P_\beta \leq P_\gamma$ alors ajouter la tâcheβ à la liste des ordonnançables. La tâche β devient α;

La tâche γ est déplacée de la liste non ordonnançable et devient β. La tâche suivante dans la liste non ordonnançable devienne γ. Répéter l'étape 2 sans si la liste non ordonnançable est vide; Dans ce cas ajouter la tâche β à la liste ordonnançable et arrêter.

Sinon F_α + max { P_β, P_γ }> max { d_β, d_γ } et si $P_\beta > P_\alpha$, retourner la tâche β à la liste non_ordonnançable et le job γ devient le jobβ. Aller à l'étape 3.

Etape 3 : si F_α - P_α + max { P_α, P_β }$\leq$ max { d_α , d_β }ou si $P_\alpha \leq P_\beta$, alors ajouter le job β à la liste_ordonnançable. Le jobβ devient le jobα; le premier job de la liste _ nonordonnable devient le privot et le job suivant de la liste non_ordonnançable devient le job γ. Aller à l'étape 2.

Sinon si F_α - P_α + max { P_α, P_β }> max { d_α , d_β }et $P_\alpha > P_\beta$, aller à l'étape 4 (de saut).

Etape 4 (de saut) :

Déplacer le job α de la liste_ordonnançable et le remettre dans la liste non_ordonnançable dans l'ordre EDD. Si des tâches restent dans list_ordonnançable, le dernier job restant devient le job α. Aller à l'étape 3.

Si list_ordonnançable est vide, le job β est attribué à la première position sur list-ordonnançable et devient le job α; le premier job de list non_ordonnançable devient le pivot; Le suivant de list non_ordonnançable devient γ. Aller à l'étape 2.

Cet algorithme peut fournir un ordonnancement optimal si $I_j = \begin{cases} \Phi & \text{si } c_j \leq d_j \\ [d_j, c_j] & \text{si } c_j > d_j \end{cases}$

appelé intervalle de retard.

Théorème 7 : La séquence produite par l'algorithme de Wilkerson-Irwin minimise $\overline{T}$ s'il n'existe pas un instant t tel que $t \in I_j$ et $t \in I_i$ pour tout couple (i, j) de tâches et d'intervalles de retards.

5. Les problèmes difficiles à une seule machine

Le problème $1 / r_j , \widetilde{d}_j / C_{max}$ est NP-difficile au sens fort même si r_j et $\widetilde{d}_j$ sont entiers. Il se réduit à un problème de 3-partition.

Proposition 3 : Le problème $1 / r_j / L_{max}$ est fortement NP-difficile.
Preuve: Le nombre de n jobs est égal à 4t - 1 et

$r_j = jb + (j - 1)$ $\quad p_j = 1$ $\quad d_j = jb + j$ $\quad j = 1, \ldots, t - 1$

$r_j = 0$ $\quad p_j = a_{j-t+1}$ $\quad d_j = tb + (t-1)$ $\quad j = t, \ldots, 4t - 1$

Un ordonnancement avec $L_{max} \leq 0$ existe si et seulement si chaque job j, j = 1, …, t - 1 peut être exécuté entre r_j et $d_j = r_j + P_j$. Cela est possible que si le reste des tâches peut être partitionné sur les t intervalles de longueur b, qui peut se faire si et seulement si le 3-partition a une solution.

Seuls les trois problèmes particuliers du problème $1 / r_j / L_{max}$ sont solvables :

a) si $r_j = r$, pour tout j = 1,…, n. Un ordonnancement optimal est obtenu par la règle de Jackson. Ordonnancer dans l'ordre croissant des due-dates (EDD).
b) Si $d_j = d$ pour tout j = 1,…, n. Ordonnancer dans l'ordre croissant des release-dates
c) Si $p_j = 1$, pour tout j = 1,…, n. En tout instant, une tâche disponible avec la plus petite due date. C'est en O(nlogn) (Horn 1974).

b) cas de tâches dépendantes

On suppose que les tâches sont liées par un graphe de précédence Γ. i < j signifie que la tâche i ne peut s'exécuter que si la tâche j a terminé son exécution.

6. Le problème 1 / prec, r_j, $\widetilde{d}_j$ / C_{max}

Il est classé NP-difficile au sens fort car déjà 1 / r_j , $\widetilde{d}_j$ / C_{max} l'est. Seulement si $P_i = 1$ et si r_i et d_i sont des entiers multiples d'une certaine unité de temps, une version modifiée de la règle EDD résolve le problème optimalement en temps polynomial [Bla94].

7. Le problème 1 / prec / $\sum w_i c_i$

Pour les contraintes de précèdence générales, Lawler et al(1978) montrent qu'il est NP-difficile. Sidney(1975) a présenté une approche de décomposition du graphe qui donne un ordonnancement optimal. Potts (1985) utilise une procédure par séparation et évaluation.

Si le graphe de précédence est un arbre, Horn(1972) a présenté un algorithme d'optimisation polynomial.

Soit G = (X, U) un graphe quelconque.

Un ensemble $U \subset T$ est dit ayant des précédences sur $V \subset T$ s'il existe une tâche $T_i \in U$ et $T_j \in V$ tel que T_j est un successeur de T_i. On écrira $U \rightarrow V$.

Un ensemble $U \subset T$ est dit "initial" dans (T, <) si (T-U) $\xrightarrow{\text{non}}$ U

$\Leftrightarrow$ (T - U) $\rightarrow$U n'est pas vraie. Aucune tâche de (T - U) n'a de successeurs dans U.

Pour toute tâche de U, tous ces prédécesseurs sont dans U.

Pour tout $U \subset T$, $U \neq \varnothing$, définissons P(U) = $\sum_{T_i \in U} P_i$ et W(U) = $\sum_{T_i \in U} w_i$ et $\rho(U) = \frac{P(U)}{W(U)}$.

Un sous ensemble $U \subset T$ est dit ρ^*-minimal pour (T, <) si

i) U est initial dans (T, <)

ii) $\rho(U) \leq \rho(V)$ $\forall$ V, V initial dans (T, <) et

iii) $\rho(U) < \rho(V)$ $\forall$ V, $V \subset U$.

Dans ces conditions, on a l'algorithme de Sidney.

Algorithme 4 (de Sidney(1975) pour le problème 1 / prec / $\sum w_i c_i$

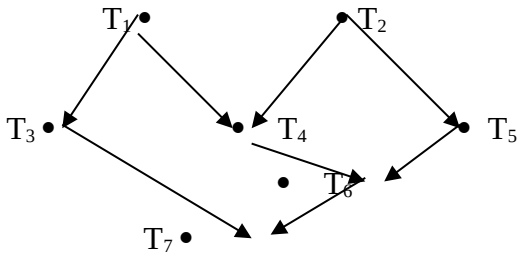
```
Begin
While T ≠ ∅ do
        Begin
        Determine task set U that is ρ*-minimal for ( T, < );
        Schedule the members of task set U optimally;
        T : = T - U
        End;
```

End.

Si $\Gamma = \varnothing$, l'algorithme de Sidney n'est autre que la règle de Smith.

Exemple 7 : Soit T = { T_1, …, T_7 }un ensemble de 7 tâches dans les temps d'exécution et les poids sont données par P = [5, 8, 3, 5, 3, 7, 6] et W = [1, 1, …, 1].

Les contraintes de précédence sont :



Le sous ensemble U = { T_1, T_3} est initial et P(U) = 8, W(U) = 2 et ρ(U) = 4.

Il n'existe pas de V, V initial tel que ρ(V) < ρ(U). Seul { T_1} est le sous ensemble propre de U avec ρ(T_1) = P_1 = 5 > ρ(U). D'où U est ρ* minimal.

On considère le sous ensemble T - U = { T_2, T_4, T_5, T_6, T_7 }sur qui on refait le même travail.

Le sous ensemble ρ* minimal est { T_2, T_4, T_5}.

L'ordonnancement optimal sera { T_1, T_3, T_2, T_5, T_6, T_7 }.

Ce problème peut être formulé comme un programme linéaire à variables bivalentes (0,1)

Soient $x_{ij} = \begin{cases} 1 & \text{si le job i est exécuté avant le job j} \\ 0 & \text{sinon} \end{cases}$ pour (i, j = 1,……,n),

$e_{ij} = 1$ si les contraintes de précédence spécifient que le job *i* est un prédécesseur du job *j* et

$e_{ij} = 0$ sinon. La date de fin d'exécution du job *j* survient à l'instant $\sum p_i x_{ij} + p_j$,

le problème s'écrit: Minimiser $\sum_i \sum_j p_i x_{ij} w_j + \sum_j p_j w_j$

Sous les contraintes: $x_{ij} \geq e_{ij}$ (i, j =1, ..., n; i ≠ j) (1)

$x_{ij} + x_{ji} = 1$ (i, j =1, ..., n; i ≠ j) (2)

$x_{ij} + x_{jk} + x_{ki} \geq 1$ (i, j, k =1, ..., n; i ≠ j, i ≠ k, j ≠ k) (3)

$x_{ij} \in \{0,1\}$ (i, j =1, ..., n) (4)

$x_{ii} = 0$(i = 1, ..., n) (5)

$e_{ij} \in \{0,1\}$ (i, j =1, ..., n) (6)

Les contraintes (1) assurent que $x_{ij} = 1$ dans le cas où la contrainte de précédence spécifie que la tâche i est un prédécesseur de j.

Une tâche T_i est ordonnancée avant ou après T_j dans (2).

(2) et (3) impliquent que le graphe G_X où $X = (x_{ij})$ comme une matrice d'adjacenses, ne contenant aucun cycles (Si un graphe contient un cycle, il contiendrait forcément un cycle de longueur trois).

8. Difficulté des problèmes

Le problème 1 / prec, r_j / L_{max} est NP-difficile.

1 / pmtn, prec, r_j / L_{max} et 1 / prec, r_j, $P_j = 1$/ L_{max} sont faciles (Sid78, Mon82).

Les problèmes 1 / prec, r_j/ $\sum w_i c_i$ et 1 / prec, $\tilde{d}_j$ / $\sum w_i c_i$ sont difficiles.

Même si le graphe de précédence est simple de la forme d'une chaîne, et les temps d'exécution unitaires, les problèmes 1 / chains, r_j, $P_j = 1$/ $\sum w_i c_i$; 1 / chains, $\tilde{d}_j$, $P_j = 1$/ $\sum w_i c_i$ sont difficiles (Lenstra & Rinnooy Kan (1980))

1 / r_j / L_{max} est NP-difficile.

Si les temps d'exécution sont unitaires, 1 / r_j, $P_j = 1$ / L_{max} ; 1/ pmtn, r_j/ L_{max} sont résolus optimalement par Horn (1974).

1/ pmtn, prec, r_j/ L_{max} es facile. Lawler a donné un algorithme en $O(n^2)$.

Les problèmes 1/ prec/ $\sum w_i T_i$; 1 / prec, $P_j = 1$/ $\sum T_i$; 1 / chains, $P_j = 1$/ $\sum w_i T_i$ sont NP-difficiles même si les poids $w_i = w$ sont de même valeur.

9. Généralisations

9.1. de la règle EDD

Définissons $d_j' = \min \{ d_j, \min \{ d_i / j < i, i$ un successeur de $j\}\}$

Théorème 8 : Les problèmes 1/ prec / L_{max} et 1/ prec / T_{max} sont minimisés par la séquence

$$d'_{[1]} \leq d'_{[2]} \leq .... \leq d'_{[n]}$$

Preuve : (adjacent pairwise interchange).

9.2. de la règle SPT

Si la mesure de performance est $\overline{F} = \frac{1}{n}\sum_{i=1}^{n} F_i = \frac{1}{n}\sum_{i=1}^{n} c_i - r_i = \frac{1}{n}\sum_{i=1}^{n}(w_i + P_i) = \overline{W} + \frac{1}{n}\sum_{i=1}^{n} P_i$

Considérons le problème 1 / prec / $\overline{F}$.

Un "string" (anneau) est un ensemble de tâches qu'il faut exécuter dans un ordre fixe sans interruption. Soit un graphe de précédence donné par " s " strings (un caractère, un ensemble d'anneaux).

N_k : le nombre de tâches dans le caractère k ($1 \leq k \leq s$)

P_{kj} : temps d'exécution de la tâche j du caractère k ($1 \leq j \leq n_k$)

$P_k = \sum_{j=1}^{n_k} P_{kj}$: temps d'exécution du caractère k

F_{kj} : flow time de la tâche j dans le caractère k

$F_k = F_{k,nk}$: date de fin du caractère k

L'objectif est de minimiser $\frac{1}{s}\sum_{k=1}^{s} F_k$. Chaque caractère est considéré comme une tâche et une séquence optimale peut être de la forme : $P_{[1]} \leq P_{[2]} \leq .... \leq P_{[s]}$

Soit un nouvel objectif noté $\overline{F} = \frac{1}{n}\sum_{k=1}^{s}\sum_{j=1}^{n_k} F_{kj}$ alors

Théorème 9 : Le problème 1 / string / $\overline{F}$ est minimisé par la séquence des caractères dans l'ordre $\frac{P_{[1]}}{n_{[1]}} \leq \frac{P_{[2]}}{n_{[2]}} \leq ... \leq \frac{P_{[s]}}{n_{[s]}}$.

10. Problèmes dans l'objectif est de la forme $\max_{i \in \{1,...,n\}} \{\gamma_i(c_i)\}$

L'algorithme de Lawler minimise les problèmes de la forme n/ 1/ prec/ $\max_{i \in \{1,...,n\}} \{\gamma_i(c_i)\}$

où $\gamma_i(c_i)$ est une fonction croissante en c_i ou régulière.

Si $\gamma_i(c_i) = c_i - d_i = L_i$, l'objectif sera le L_{max}

Si $\gamma_i(c_i) = \max (c_i - d_i , 0)$, l'objectif sera le T_{max}

Théorème 9 : considérons le problème n/ 1/ prec/ $\max_{i \in \{1,...,n\}} \{\gamma_i(c_i)\}$.

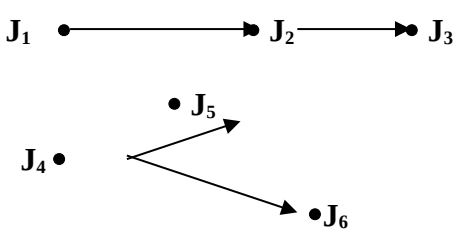
Soit V ⊆ T un sous ensemble de tâches qui peuvent être exécutées en dernier, ces tâches peuvent ne pas précéder aucune tâche.

La dernière tâche dans l'ordonnancement se termine à l'instant $\tau = \sum_{i=1}^{n} P_i$

Soit J_k la tâche dans V tel que $\gamma_k(\tau) = \min_{J_i \in V} \gamma_i(c_i)$, la dernière tâche qui coûte moins parmi celles qu'on peut exécuter la dernière, alors il existe un ordonnancement optimal où J_k est ordonnancée la dernière.

Preuve: à faire.

Exemple 8 : Soit le problème 6 / 1 / L_{max} avec des contraintes de précédence ci dessous



Job		**1**	**2**	**3**	**4**	**5**	**6**
Processing time	P_i	2	3	4	3	2	1
Due date	d_i	3	6	9	7	11	7

Trouvons la tâche qui s'exécutera la dernière, la sixième. $\tau = 2 + 3 + 4 + 3 + 2 + 1 = 15.3$

Les tâches 3, 5 et 6 peuvent être exécutée les dernières. V = { 3, 5, 6 }

Le minimum de décalage sur V est : min { (15 - 9), (15 - 11), (15 - 7)} = 4

C'est la tâche 5, J_5 qui s'exécutera la sixième.

On efface la tâche J_5 de la liste et sa date de fin.

Des cinq tâches restantes le nouveau $\tau = 15 - 2 = 13$.

J_3 ou J_6 peuvent être exécutée la dernière. On calcule le min ((13-9, 13-7) = 4

J_3 sera placée à la cinquième position.

On déterminera la tâche qui s'exécutera la quatrième.

On efface J_3 et J_5. J_2 devient disponible pour s'exécuter en dernier.

$\tau = 13 - 4 = 9$. Min ((9 - 6, 9 - 7)) = 2, la tâche J_6 réalise ce minimum et sera placée à la quatrième position.

On déterminera à la troisième position parmi V = { J_2, J_4} et $\tau = 9 - 1 = 8$.

Min (8 - 6, 8 - 7) = 1, J_4 sera exécutée la troisième. Ainsi de suite on trouve

L'ordonnancement $(J_1, J_2, J_4, J_6, J_3, J_5\}$ et $L_{max} = 4$.

Cette méthode ne donne pas directement la valeur du L_{max} qu'il faut déterminer.

On peut la retrouver en dressant un tableau typique ci dessous.

τ	J_1	J_2	J_4	J_6	J_3	J_5	Job ordonnancé
15	*	*	6	*	$\overline{4}$	8	J_5
13	*	*	$\overline{4}$	*	S	6	J_3
9	*	3	S	*	S	$\overline{2}$	J_6
8	*	2	S	$\overline{1}$	S	S	J_4
5	*	$\overline{-1}$	S	S	S	S	J_2
2	$\overline{-1}$	S	S	S	S	S	J_1

D'où $L_{max} = 4$, valeur en rond.

Chapitre 5 : Méthodes exactes de résolution des problèmes d'ordonnancement NP-difficiles

1. Introduction aux méthodes par séparation et évaluation

Les méthodes arborescentes (Branch and Bound Methods) sont des méthodes exactes d'optimisation qui pratiquent une énumération intelligente de l'espace des solutions. Il s'agit d'énumération complètes améliorées. Elles partagent l'espace des solutions en sous ensembles de plus en plus petits, la plupart étant éliminés par des calculs de bornes avant d'être construits explicitement. Appliquées à des problèmes NP-difficiles, ces méthodes restent bien sur exponentielles, mais leur complexité en moyenne est bien plus faible que pour une énumération complète. Elles peuvent pallier le manque d'algorithmes polynomiaux pour des problèmes de taille moyenne. Pour des problèmes difficile de grande taille, leur durée d'exécution est très grande et il faut se retourner vers des heuristiques ou solutions approchées.

Pour un POC, on peut inventer plusieurs méthodes arborescentes. Cependant elles auront trois composantes communes :

- Une règle de séparation
- une fonction d'évaluation
- une stratégie d'exploration

2. Principe des méthodes par séparation et évaluation

Soit (S, f) la donnée d'un POC, plus particulièrement un problème d'ordonnancement.

Définitions 1 :

- On dit qu'on sait évaluer le sous ensemble $S' \subset S$, si on sait déterminer un réel g(S') tel que $g(S') \leq f(s) \quad \forall s \in S'$ (respectivement $g(S') \geq f(s) \ \forall s \in S'$).
- Une évaluation g(S') de S' est dite exacte s'il existe $s_0 \in S'$ tel que $g(S') = f(s_0)$.
- Une évaluation h(S') de S' est dite meilleure qu'une évaluation g(S') de S' si h(S') > g(S') (resp. h(S') < g(S')).
- Si g(S') est une évaluation exacte, il ne peut exister une évaluation meilleure.

Définition 2 : Un sous ensemble $S' \subset S$, ensemble de solutions d'un POC est dit " stérile " si on connaît une solution $s_0 \in S$ tel que $f(s_0) \leq g(S')$ (resp. $f(s_0) \geq g(S')$)

S'$\subset$ S peut être stérilisé dans deux cas,

- On est sur que S' ne peut contenir de solutions optimales (S' = $\oslash$ ou $\exists$ $s_0 \in S$, $f(s_0) < g(S')$)
- on dispose d'une évaluation exacte de S'.

Définition 3 :

Un sous ensemble S'$\subset$ S est dit "séparé" en S'_1 , S'_2,…, S'_k si S' = $S'_1 \cup S'_2 \cup \ldots \cup S'_k$
avec $S'_i \subset S'$ $\forall$ i = 1, …, k.
La séparation n'est pas propre si et seulement si $\exists$ k / $S'_k = \oslash$

Théorème 1 : Soit S = $\bigcup_{S' \in F} S'$ alors si tous les éléments de la famille F sont stériles on a soit :

- Aucun ensemble S' n'a une évaluation exacte et S = $\oslash$,
- la meilleure évaluation exacte des ensembles S' correspond à une solution optimale.

3. Schéma général des méthodes par séparation et évaluation dites SE

$\hat{s}$ désignera la meilleure solution connue (non forcément optimale) et v = f($\hat{s}$).
Si $\hat{s}$ est encore indéterminée on posera v = + ∞ (resp. pour le max on pose v = - ∞)
F une famille de sous-ensembles de S recouvrant S.

3.1. Procédure d'initialisation

Si on ne connaît pas de solution réalisable $\hat{s}$, indéterminée et v = + ∞. La famille F contient l'unique élément S. S est déclaré non stérile. Aller à la procédure de choix.

3.2. Procédure de choix

a) si tous les éléments de F sont stériles, $\hat{s}$ est solution optimale. Si $\hat{s}$ est indéterminée, le problème posé n'a pas de solutions et S = $\oslash$.
b) Sinon choisir S' $\in$F, non stérile.
   b1) si S' n'a pas été évalué, aller à la procédure évaluation 3.
   b2) si S' a été évalué, aller à la procédure séparation 4.

3.3. Procédure d'évaluation

Calculer g(S') l'évaluation de S.
a) si g(S') $\geq$ v alors S' est déclaré stérile. Aller à la procédure choix 2.

b) Si g(S') < v et si l'évaluation est exacte (g(S') = f(s'), s'∈ S') alors poser $\hat{s}$ = s' et v = g(S') = f(s'). S' est déclaré stérile. Aller à la procédure choix 2.

c) Si on n'est pas dans le cas a) ou b) aller à la procédure de choix 2.

3.4. Procédure de séparation

Séparer S' en créant les sous-ensemble S'_1 , S'_2,…, S'_k où $S' = S'_1 \cup S'_2 \cup \ldots \cup S'_k$.

Les S'_i ne sont pas déclarés stériles. Posons $F = (F \setminus \{S'\}) \cup \{ S'_1, S'_2, \ldots, S'_k \}$ et aller à la procédure du choix 2.

Remarques :

Dans le cas général, l'algorithme de ce schéma est fini si $S'_i \subset S'$, $\forall$ i = 1, …, k.

Les stratégies possibles pour la procédure du choix 2 sont les suivantes,

a) sélectionner S' dont l'évaluation est la plus faible (resp. la plus grande).

b) Sélectionner S' le plus récemment crée, cette manière de faire correspond à une exploration dite " profondeur d'abord ". La famille F fonctionne comme une pile, le dernier arrivé est le premier servi.

c) Sélectionner S' le plus anciennement crée, elle correspond à une exploration " largeur d'abord ". F est géré comme une pile FIFO.

d) On pratique on combine a), b) et c).

Exemple 1 : un problème d'ordonnancement à une seule machine

De nombreux problèmes d'ordonnancement classés NP-difficiles peuvent être résolus par des procédures par séparation et évaluation. Nous présentons un problème à une machine et problème de type job shop.

Un atelier comporte une seule machine toujours disponible. "n " tâches doivent y être exécutées les unes à la suite des autres. Elles sont toutes disponibles dès l'instant zéro. La tâche i dure p_i unités de temps et doit être terminée avant l'instant d_i, sinon il faut payer une pénalité de retard w_i par unité de temps de retard. Il s'agit de trouver le meilleur ordre de passage des tâches sur la machine de manière à minimiser la somme des pénalités de retard à payer.

Ensemble des solutions réalisables S_0 est formé des n! permutations possibles des n tâches.

La séparation se fera sur la tâche que l'on fixe en dernier, puis en avant dernier, …

Pour le calcul des bornes ou minorant, on prend en considération la somme des pénalités des tâches placées en queue. L'élimination des sous ensembles se fera sur le sommet ayant le plus petit minorant. Le problème suivant est à 1 machine et 3 tâches.

	Tâche 1	Tâche 2	Tâche 3
p_i	2	4	1
d_i	3	5	5
w_i	4	2	3

Etape 0 : initialisation

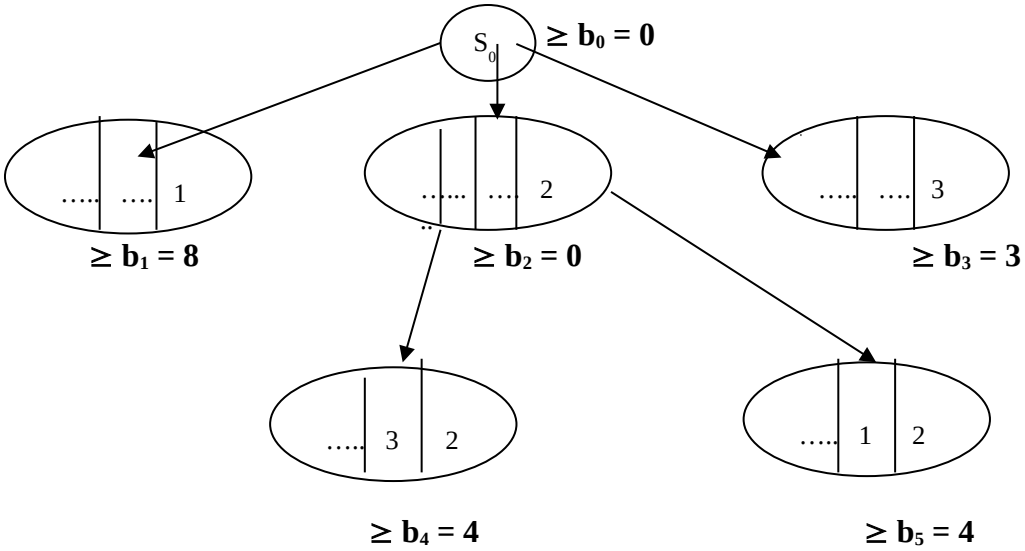


Fig 1. Schéma récapitulatifs des séparations.

On a séparé par rapport à une tâche placée en dernière position.

Pour calculer b_1, quelque soit la séquence (2, 3, 1) ou (3, 2, 1), la tâche 1 ne peut commencer son exécution qu'à la date $p_2 + p_3 = 5$. Elle sera en retard de 2 unités.

Le coût de retard sera d'au moins 2 x 4 = 8 d'où $b_1 = 8$.

Ainsi de suite $b_2 = 0$ car la tâche 2 ne sera pas en retard si elle est placée la dernière.

Si la tâche 3 est placée a dernière, elle sera en retard de 1 unité, $b_3 = 3 \times 1 = 3$.

On sépare par rapport au sommet de b_2. On trouve que pour les séquences (1, 3, 2) ou (3, 1, 2) les pénalités sont respectivement de 4 unités.

Il faut encore séparer le sommet où la tâche 3 est placée la dernière. On trouve 2 sous ensembles de coût 8 et 18. D'où (1,3,2) et (3,1,2) sont optimaux.

Exemple 2 : un problème du job shop

Pour réaliser la fabrication de trois pièces, on dispose de leur gamme de fabrication.

La pièce i demande q_i opérations et la jième opération de la pièce i occupe τ_{ij} unités de temps de la machine M_{ij}. Les durées d'usinage sont très longues par rapport aux durées de transport et de réglage et on négligera ces dernières. Cette méthode ne permet de résoudre que des problèmes de taille raisonnable. Le problème qui consiste à placer les opérations sur les machines de telle sorte que :

- L'opération j du produit i soit terminée avant le début j+1.
- Une machine ne traite qu'une seule opération à la fois
- On minimise la durée totale (l'opération qui se termine la dernière se termine le plus tôt possible)

Les gammes sont fournies dans le tableau suivant:

n° opération	1	2	3	4
Pièce 1				
durées	3	1	2	1
machines	(1)	(2)	(3)	(3)
Pièce 2				
durées	1	2	1	3
machines	(2)	(1)	(3)	(2)
Pièce 3				
durées	2	1	./.	./.
machines	(3)	(2)	./.	./.

Les bornes minorantes sont calculées par relaxation en ignorant, initialement, les contraintes dues au fait qu'une machine ne peut effectuer qu'une seule et unique opération à la fois. Ces contraintes seront intégrées progressivement au fur et à mesure des séparations jusqu'à l'obtention de solutions réalisables qui seront optimales pour le sous-ensemble de solutions auquel elles appartiennent. En fin d'exploration, la meilleure solution réalisable trouvée sera optimale. L'ensemble des solutions candidates A_0 est l'ensemble de tous les ordres possibles de toutes les opérations à exécuter sur les différentes machines.

Pour trouver un minorant à la meilleure solution possible de cet ensemble, on ignore les contraintes dues aux machines en faisant comme si on avait autant d'exemplaires de chaque machine que nécessaire pour réaliser la solution "au plus tôt" pour chaque pièce, ce qui donne

l'ordonnancement non réalisable suivant représenté sur un diagramme où chaque ligne correspond à une pièce et chaque case à la j-ème opération de la tâche i sur la machine (k).

1	2	3	4	5	6	7	8	9	10	11	12

Pièce 1

1,1(1)	1,2(2)	1,3(3)	1,4(3)

Pièce 2

2,1(2)	2,2(1)	2,3(3)	2,4(2)

Pièce 3

3,1(3)	3,2(2)

Fig 2. Diagramme " pièces " calé au plus tôt correspondant au sous ensemble A_0.

La plus longue gamme ayant une durée totale de sept (7) unités de temps, 7 est un minorant pour la durée totale pour l'ensemble des solutions du problème d'où $b_0 = 7$. La solution ci dessus serait optimale si elle était réalisable. Elle comporte des conflits sur l'utilisation des machines. A l'instant 1, un conflit survient entre 1,1 (1) (la première opération du produit un) et 2,2(1) (la deuxième opération du produit deux) sur l'utilisation de la machine 1.

Nous devons séparer l'ensemble de solutions A_0 en deux sous ensembles de solutions de telle sorte que toute solution réalisable se retrouve obligatoirement dans l'un des deux et que les solutions non réalisables qui contenaient ce premier conflit ne s'y retrouvent plus.

Pour cela, on partitionne A_0 en deux sous ensembles complémentaires

A_1 où 1,1(1) est placée sur la machine 1 avant 2,2 et

A_2 où 2,2 est placée sur la machine 1 avant 1,1.

Avant de séparer A_1 ou A_2 , nous allons calculer les nouvelles bornes les concern ant de manière à pouvoir appliquer une procédure par séparation et évaluation progressive en séparant toujours le sous ensemble de meilleur borne.

Placer 1,1 avant 2,2 revient à déplacer le début de 2,2 de 2 unités et donc à déplacer la fin du produit deux de 7 à 9 unités. D'où $b_1 = 9$.

Placer 2,2 avant 1,1 revient à déplacer le début de 1,1 de 3 unités et donc de déplacer la fin du produit un de 7 à 10 unités. D'où $b_2 = 10$.

La meilleure borne est fournie par A_1 que nous allons séparer en cherchant le premier conflit-machine sur son diagramme pièces calé au plus tôt.

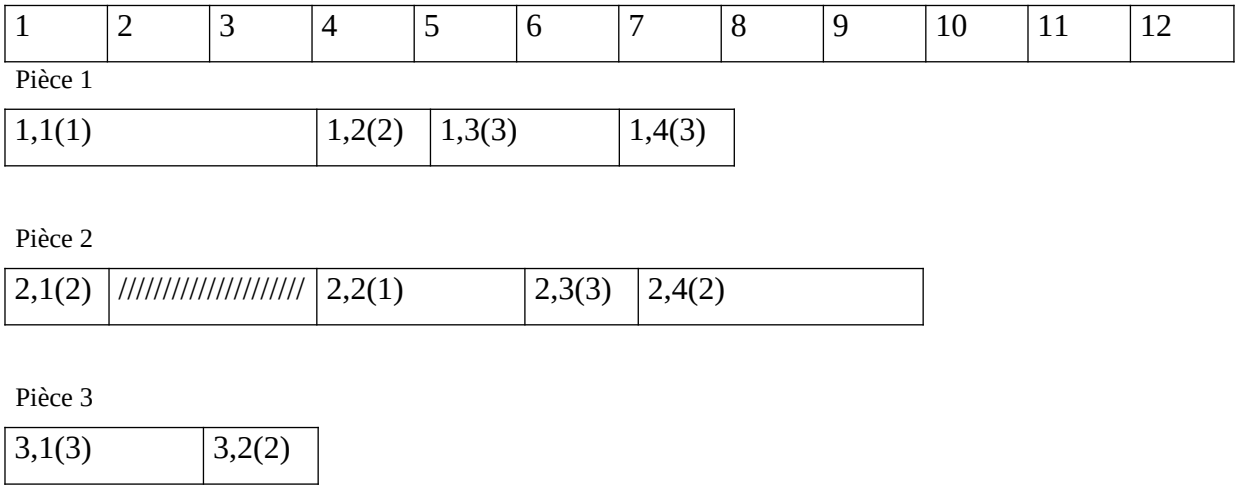


Fig 3. Diagramme " pièces " calé au plus tôt correspondant au sous ensemble A_1**.**

Ici, le diagramme de A_2 n'est pas représenté.

Un conflit survient à l'instant 5 entre 1,3(3) et 2,3(3) pour l'utilisation de la machine 3.

A_1 sera séparé en deux sous ensembles A_3, où 1,3 (3) est placée avant 2,3(3) et A_4, où 2,3(3) est placée avant 2, 3(3).

Placer 1,3(3) avant 2,3(3) revient à déplacer 2,3(3) d'une unité de temps et donc de déplacer la fin du produit 2 de l'instant 9 à 10.

Placer 2,3(3) avant 1,3(3) revient à déplacer 1,3(3) de deux unités de temps et donc de déplacer la fin du produit 1 de l'instant 7 à 9.

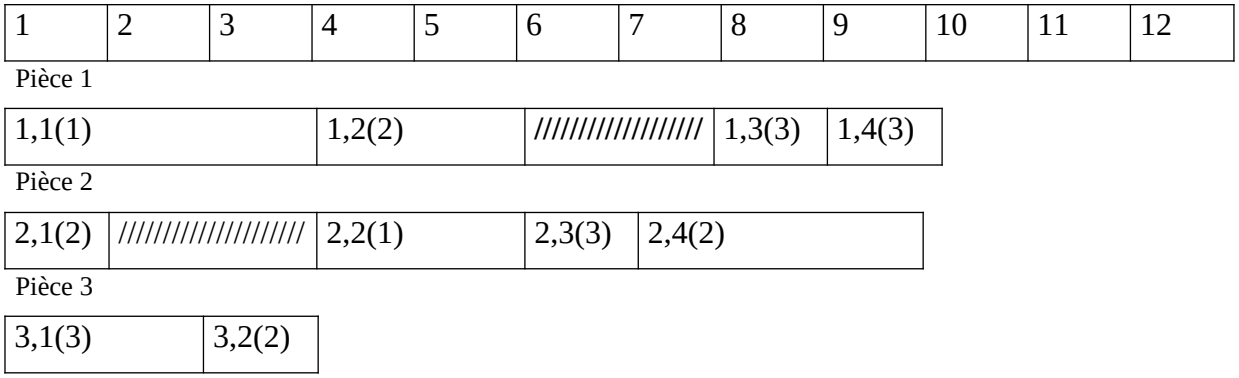


Fig 4. Diagramme " pièces " calé au plus tôt correspondant au sous ensemble A_4.

Ce diagramme ne comporte plus de conflit pour l'utilisation des machines. C'est une solution réalisable pour le problème posé, comme sa durée totale est strictement inférieure à toutes les

bornes des sous ensembles que nous n'avons pas encore séparés. C'est l'unique solution optimale parmi les solutions calées au plus tôt.

Remarque : Si on programme cette méthode, on n'utilise pas les diagrammes " pièces " mais les dates au plus tôt de toutes les opérations sur le graphe potentiel-tâche comportant les contraintes de précédence dues aux gammes puis celles ajoutées progressivement à chaque séparation pour tenir compte des conflits dus aux machines. La borne étant la date au plus tôt de la tâche fictive de fin de réalisation.

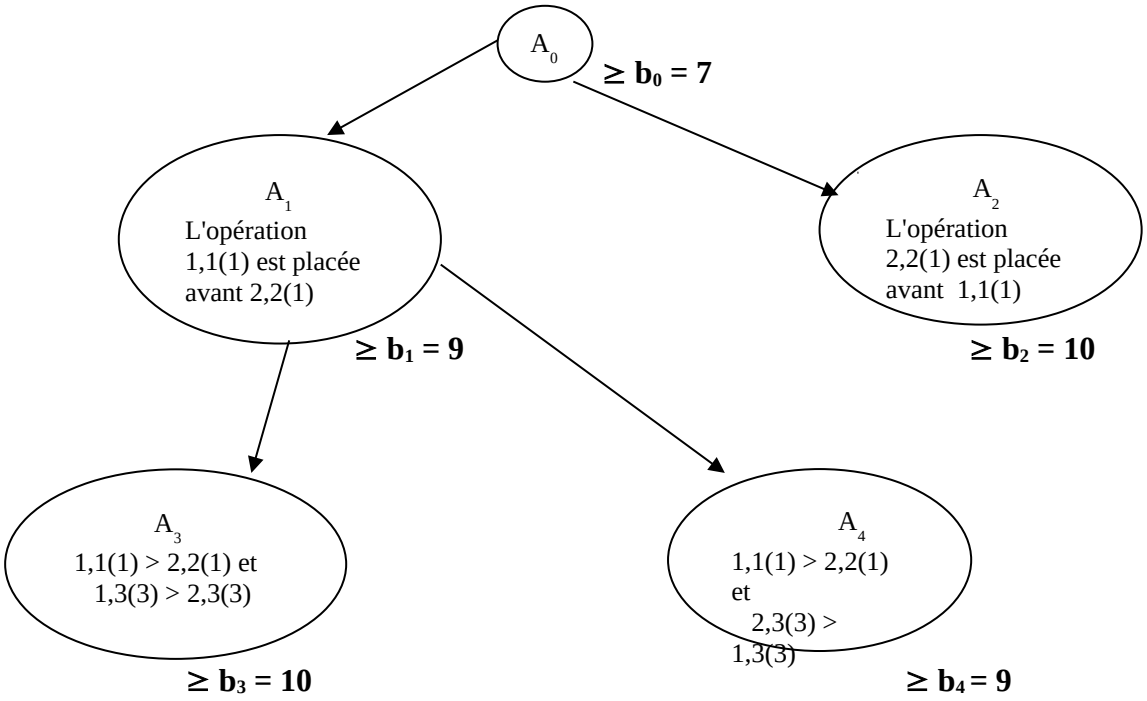


Fig 5. Arborescence des séparations.

La programmation dynamique

La programmation dynamique est une technique mathématique à implémenter qui a pour objet d'aider à prendre des décisions séquentielles indépendantes les unes des autres. Il n'y a pas de formalisme mathématique standard. C'est une approche de résolution où les équations doivent être spécifiées selon le problème à résoudre.
Ce cours se base sur des exemples de l'ordonnancement qui illustrent cette technique.

1. Caractéristiques d'un problème de programmation dynamique

Les propriétés connues de la programmation dynamique sont :

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

ii) A chaque étape correspond un certain nombre d'états qui peut être infini ($x_n \in \mathbb{N}$) ou continu ($x_n \in \mathbb{R}$).

iii) A chaque étape, la décision prise transforme l'état actuel en un état associé à l'étape suivante (dans certains cas avec une distribution de probabilité).

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

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

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

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

Tout problème de programmation dynamique peut être défini par un processus de décision séquentielle. Un processus de décision séquentielle est formé :

- d'un système dynamique à temps discret évoluant pendant un nombre fini de périodes.
- d'une fonction coût additive au fil des périodes.

2. Système dynamique à temps discret

On s'intéresse au problème dit déterministe où la connaissance de l'état et de la décision à prendre suffisent pour savoir l'état à l'étape suivante. L'évolution de l'état x_k du système est régie par le processus $x_{k+1} = f(x_k , u_k , w_k)$ $k = 1, \ldots, N$.

Le système débute son évolution dans l'état initial x_1 et s'arrête après N périodes ou étapes dans l'état final x_{n+1}.

Variables

Symbole	Définition	Domaine de définition
X_k	Etat du système au début de l'étape k avant la prise de décision	$\in S_k$, k = 1, …, N+1
U_k	Décision prise pendant l'étape k	$\in C_k$, k = 1, …, N
W_k	Perturbation aléatoire de l'étape k (non connue lors de la décision)	$\in \Omega_k$, k = 1, …, N

Les w_k sont supposées indépendantes les unes des autres.

Domaines de définition

Symbole	Définition
S_k	Ensemble des états possibles au début de l'étape k, k = 1,…, N+1
C_k	Ensemble de toutes les décisions possibles pendant l'étape k, k = 1, …, N
$U_k(x_k)$	Ensemble de décisions admissibles dans l'état x_k au début de l'étape, k = 1, …, N. $\varnothing \neq U_k(x_n) \subseteq C_k$
Ω_k	Ensemble des valeurs possibles pour les perturbations aléatoires de l'étape k, k = 1, …, N.

Fonctions

Symbole	Définition	
$F_k,(x_k, u_k, w_k)$	Fonction de transfert de l'étape k, k =1,…, N	$f_k : S_k \times C_k \times \Omega_k \rightarrow S_{k+1}$ $(x_k \times c_k \times w_k) \mid\rightarrow x_{k+1}$
$H_k(w_k)$	Loi de probabilité des perturbations aléatoires w_k, de l'étape k, peut dépendre de l'état x_k et ou de la décision u_k.	$H_k : \Omega_k \rightarrow$ □

Fonction coût

Symbole	Définition	
$G_k,(x_k, u_k, w_k)$	Coût de l'étape k	$g_k : S_k \times C_k \times \Omega_k \rightarrow$
$G_{N+1}(x_{N+1})$	Coût terminal	$g_{N+1} : S_{N+1} \rightarrow$

Fonctions de décisions et politiques

Symbole	Définition	
$u_k,(x_k)$	Fonction de décision à l'étape k	$u_k : S_k \rightarrow C_k$
π	Politique de décisions	$\pi = \{ u_1, \ldots, u_N\}$
$\pi^{(k)}$	Politique partielle ou résiduelle	$\pi^{(k)} = \{ u_k, \ldots, u_N\}$

- La fonction u_k est admissible si $u_k(x_k) \in U_k(x_k), \forall x_k \in S_k$.
- La politique π est admissible si chacune des fonctions u_k la formant l'est. L'ensemble de toutes les politiques admissibles est noté Π.
- La performance de la politique π pour l'état initial x_1 est :

$$J_\pi = \underset{w_1, w_2, \ldots, w_n}{E} \left[\sum_{k=1}^{N} g_k(x_k, u_k(x_k), w_k) + g_{N+1}(x_{N+1}) \right]$$

Où $x_{k+1} = f [x_k, u_k(x_k), w_k]$, $k = 1, \ldots, N$.

- La politique π^* est optimale si quelque soit l'état initial $x_1 \in S_1$, on a :

$$J_{\pi^*(x1)} = \underset{\pi \in \Pi}{\text{Min}} J_\pi(x_1)$$

Algorithme 1 (de la programmation dynamique)

Données : Un processus de décisions séquentielles.

Résultats : Les tables contenant la politique optimale $\pi^* = \{ u_1^*, \ldots, u_N^*\}$ et les espérances minimales J_k.

1) Initialisation $J_{N+1}(x_{N+1}) = g_{N+1}(x_{N+1}) \quad \forall x_{N+1} \in S_{N+1}$.

2) Construction des tables optimales

pour k = N jusqu'à 1 faire

Pour tout $x_k \in S_k$ calculer

$$J_k(x_k) = \underset{u_k \in U_k(x_k)}{\text{Min}} \underset{w_k}{E} \left[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right]$$

Et stocker les valeurs $J_k(x_k)$ et les arguments $u^*_k(x_k) = u^*_k$ pour lesquels les minima sont atteints.

3. Deux exemples de problèmes d'ordonnancement

Exemple 1 : L'approche de Held et Karp

Supposons que la fonction objectif est de la forme $\sum_{i=1}^{n} \gamma_i(c_i)$.

$\gamma_i(c_i)$ sont des fonctions croissantes des temps de fin d'exécution. D'où la fonction objectif est régulière. $\overline{C}$, $\overline{F}$ et $\overline{T}$ peuvent se formuler ainsi ($\gamma_i(c_i)$ est respectivement $\frac{c_i}{n}$, $\frac{c_i - r_i}{n}$ et max { $c_i - d_i$, 0}).

On veut résoudre n / 1 / $\overline{C}$; n / 1 / $\overline{F}$ et n / 1 / $\overline{T}$.

On sait que SPT est optimale pour les deux problèmes. S'il faut le faire c'est pour n/ 1/ $\overline{T}$.

L'approche peut être étendue aux fonctions ($\max_{i \in \{1,\ldots,n\}} \{\gamma_i(c_i)\}$ **).** La programmation dynamique n'est pas la meilleure, l'algorithme de Lawler l'est.

La méthode de Held et Karp se base sur la structure optimale de la solution. Dans une solution optimale, les k premiers jobs doivent formés une sous-solution optimale, k = 1, …, n, pour le problème réduit à k jobs.

Soit $\{J_{i(1)}, J_{i(2)},\ldots, J_{i(n)}\}$ un ordonnancement optimal pour le problème à n jobs $\forall k = 1, 2,\ldots, n$.

$$\sum_{k=1}^{n} \gamma_{i(k)}\, c_{i(k)} = \sum_{k=1}^{K} \gamma_{i(k)}\, c_{i(k)} + \sum_{k=k+1}^{n} \gamma_{i(k)}\, c_{i(k)} = A + B \qquad (1)$$

S'il y a un ordonnancement qui peut réduire la valeur de A, il peut réduire tout l'ordonnancement.

Soit Q un ensemble de jobs contenant le job J, alors Q - { J }ne contient pas J.

A tout Q, on associe $C_Q = \sum_{J_i \in Q} p_i$ est la date de fin du dernier job de Q.

$\Gamma(Q)$: le coût minimum en ordonnançant les jobs dans Q optimalement.

Si Q = { J_i } un singleton alors $\Gamma(Q) = \Gamma(\{J_i\}) = \gamma_i(p_i)$, (2)

Supposons que Q contient k jobs, k > 1.

Le dernier job termine à C_Q et les (k - 1) jobs sont ordonnancés optimalement d'où

$$\Gamma(Q) = \min_{J_i \in Q} \{\Gamma(Q - \{J_i\}) + \gamma_i(C_Q)\} \qquad (3)$$

Des formules (2) et (3), on peut trouver le minimum quand Q contient 1 job, 2 job, puis … n jobs.

Soit à résoudre 4 / 1 / $\overline{T}$ avec

Job		J_1	J_2	J_3	J_4
Processing time	P_i	8	6	10	7
Due date	d_i	14	9	16	16

$\gamma_i(c_i)$ est $\frac{1}{4}\max\{ci - di, 0\}$. Il faut calculer $\Gamma(Q)$ quand Q sont des singletons.

De l'équation (2) on aura : $\Gamma(\{J_1\}) = \gamma_1(c_1) = \frac{1}{4}\max\{c_1 - d_1, 0\} = \frac{1}{4}\max\{p_1 - d_1, 0\} =$

$\frac{1}{4}\max\{-6, 0\} = 0$ et on génère les autres valeurs.

Q	$\{J_1\}$	$\{J_2\}$	$\{J_3\}$	$\{J_4\}$
$c_i - d_i$	-6	-3	-6	-9
$\Gamma(Q)$	0	0	0	0

• Il faut calculer $\Gamma(Q\}$ pour les 6 sous ensembles à deux éléments qu'on peut former à partir de 4 jobs. Si $Q = \{J_1, J_2\}$, on a $C_Q = p_1 + p_2 = 14$ et de l'équation (3),

$\Gamma(Q) = \min\{ \Gamma(\{J_1\}) + \gamma_2(14), \Gamma(\{J_2\}) + \gamma_1(14) \} = \min\{ 0 + 5/4, 0 + 0\} = 0.$

D'où pour les autres sous ensembles à deux éléments, on aura le tableau suivant.

Q	$\{J_1, J_2\}$	$\{J_1, J_3\}$	$\{J_1, J_4\}$	$\{J_2, J_3\}$	$\{J_2, J_4\}$	$\{J_3, J_4\}$
C_Q	14	18	15	16	13	17
J_i le dernier job dans la séquence	J_1 J_2	J_1 J_3	J_1 J_4	J_2 J_3	J_2 J_4	J_3 J_4
$\gamma_i(C_Q)$	0 5/4	1 1/2	1/4 0	7/4 0	1 0	1/4 1/4
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$	0 5/4	1 1/2	1/4 0	7/4 0	1 0	1/4 1/4
Minimum	*	*	*	*	*	* *
$\Gamma(Q)$	0	1/2	0	0	0	1/4

• Il faut calculer $\Gamma(Q)$ pour les 4 sous ensembles à trois éléments qu'on peut former à partir de 4 jobs. Si $Q = \{J_1, J_2, J_3\}$, on a $C_Q = p_1 + p_2 + p_3 = 24$ et de l'équation (3),

$\Gamma(Q) = \min \{ \Gamma(\{J_1, J_2\}) + \gamma_3(24), \Gamma(\{J_1, J_3\}) + \gamma_2(24), \Gamma(\{J_2, J_3\}) + \gamma_1(24) \}$

$= \min \{ 0 + 2, 1/2 + 15/4, 0 + 10/4\} = 2.$

Par une table on donne les résultats suivants

Q	$\{J_1, J_2, J_3\}$	$\{J_1, J_2, J_4\}$	$\{J_1, J_3, J_4\}$	$\{J_2, J_3, J_4\}$
C_Q	24	21	25	23
J_i le dernier job dans la séquence	J_1 J_2 J_3	J_1 J_2 J_4	J_1 J_3 J_4	J_2 J_3 J_4
$\gamma_i(C_Q)$	10/4 15/4 2	7/4 3 5/4	11/4 9/4 9/4	14/4 7/4 7/4
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$	10/4 17/4 2	7/4 3 5/4	3 9/4 11/4	15/4 7/4 7/4
Minimum	*	*	*	*
$\Gamma(Q)$	2	5/4	9/4	7/4

• Finalement $\Gamma(Q)$ pour l'ensemble des quatre jobs.

$Q = \{J_1, J_2, J_3, J_4\}$, on a $C_Q = p_1 + p_2 + p_3 + p_4 = 31$ et de l'équation (3),

$\Gamma(Q) = \min \{ \Gamma(\{J_1, J_2, J_4\}) + \gamma_3(24), \Gamma(\{J_1, J_3, J_4\}) + \gamma_2(24), \Gamma(\{J_2, J_3, J_4\}) + \gamma_1(24) \}$

$= \min \{ 0 + 2, 1/2 + 15/4, 0 + 10/4\} = 2.$

Q	$\{J_1, J_2, J_3, J_4\}$			
C_Q	31			
J_i le dernier job dans la séquence	J_1	J_2	J_3	J_4
$\gamma_i(C_Q)$	17/4	22/4	15/4	15/4
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$	6	31/4	5	23/4
Minimum			*	
$\Gamma(Q)$	5			

On a l'ordonnancement optimal de mean tardiness égal à 5 où J_3 est la dernière tâche.

$\{J_1, J_2, J_4\}$ sont les premières tâches à exécuter.

De la table à trois tâches, la colonne $\{J_1, J_2, J_4\}$, on trouve qu'il est optimal que J_4 soit la dernière tâche à exécuter. Alors $\{J_1, J_2\}$ sont les premières tâches à exécuter. De la table à deux tâches, J_1, est la dernière tâche. D'où (J_2, J_1, J_4, J_3) est l'ordonnancement optimal.

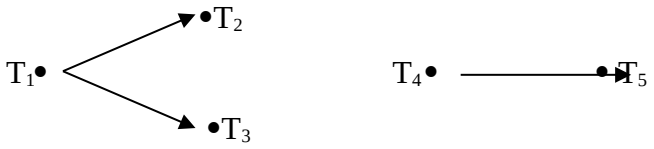
Exemple 2 : Problème d'ordonnancement sous des contraintes de précédence sur une seule machine afin de minimiser la moyenne des retards des tâches.

Soit le problème " 5 / 1 / $\overline{T} = \frac{1}{5}\sum_{i=1}^{5} T_i$ " où $T_i = \max(0, c_i - d_i)$, c_i : date de fin d'exécution de la tâche i et d_i : date de fin d'exécution au plus tard pour ne pas risquer une pénalité.

Les durées d'exécution et les dates de fin d'exécution au plus tard sont donnés par le tableau suivant :

Tâche T_i	T_1	T_2	T_3	T_4	T_5
Durée d'exécution p_i	4	2	6	3	5
Date de fin d'exécution au plus tard d_i	7	9	9	6	12

Des contraintes de précédence sont de la forme:



La même méthode s'appliquerait. On débutera avec les singletons. Il n'y a pas lieu de considérer les sous-ensembles $\{J_2\}$, $\{J_3\}$ et $\{J_5\}$. Deux seulement sont compatibles avec les contraintes.

Q	$\{J_1\}$	$\{J_4\}$
$P_i - d_i$	-3	-3
$\Gamma(Q)$	0	0

On considère après les 2-sous ensembles. Quatre seulement sont compatibles avec les contraintes. $\{J_1, J_5\}$ ne peut être les deux premiers jobs dans aucun ordonnancement.

Q	$\{J_1, J_2\}$	$\{J_1, J_3\}$	$\{J_1, J_4\}$	$\{J_4, J_5\}$
C_Q	6	10	7	8
J_i le dernier job dans la séquence	J_1 J_2	J_1 J_3	J_1 J_4	J_4 J_5
$\gamma_i(C_Q)$	Impossible 0	Impossible 1/5	0 1/5	Impossible 0
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$	0	1/5	0 1/5	0

Minimum	*	*	*	*
$\Gamma(Q)$	0	1/5	0	0

On aura aussi quatre sous ensembles réalisables à trois éléments.

Q	$\{J_1, J_2, J_3\}$	$\{J_1, J_2, J_4\}$	$\{J_1, J_3, J_4\}$	$\{J_1, J_4, J_5\}$
C_Q	12	9	13	12
J_i le dernier job dans la séquence	J_1 J_2 J_3	J_1 J_2 J_4	J_1 J_3 J_4	J_1 J_4 J_5
$\gamma_i(C_Q)$	imp 3/5 3/5	imp 0 3/5	imp 4/5 7/5	1 imp 0
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$	4/5 3/5	0 3/5	4/5 8/5	1 0
Minimum	*	*	*	*
$\Gamma(Q)$	3/5	0	4/5	0

Seulement trois sous ensembles à quatre éléments sont réalisables.

Q	$\{J_1, J_2, J_3, J_4\}$	$\{J_1, J_2, J_4, J_5\}$	$\{J_1, J_3, J_4, J_5\}$
C_Q	15	14	18
J_i le dernier job dans la séquence	J_1 J_2 J_3 J_4	J_1 J_2 J_4 J_5	J_1 J_3 J_4 J_5
$\gamma_i(C_Q)$	imp 6/5 6/5 9/5	imp 5/5 imp 2/5	imp 9/5 imp 6/5
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$	10/5 6/5 12/5	5/5 2/5	9/5 10/5
Minimum	*	*	*
$\Gamma(Q)$	6/5	2/5	9/5

Pour le seul sous ensemble à cinq éléments on aura par des calculs:

Q	$\{J_1, J_2, J_3, J_4, J_5\}$				
C_Q	20				
J_i le dernier job dans la séquence	J_1	J_2	J_3	J_4	J_5
$\gamma_i(C_Q)$	Impossible	11/5	11/5	impossible	8/5
$\Gamma(Q) - \{J_i\} + \gamma_i(C_Q)$		20/5	13/5		14/5
			*		

Minimum Γ(Q)	13/5

Pour déterminer l'ordonnancement optimal on prend l'optimum de chaque tableau.
L'ordonnancement optimal est { J_4, J_1, J_2, J_5, J_3 } de min tardiness égal à 13/5.

Remarques :

La justification de la programmation dynamique dépend de la façon dont on est capable de décomposer l'équation (1) soit :

$$\sum_{k=1}^{n} \gamma_{i(k)}\, c_{i(k)} = \sum_{k=1}^{K} \gamma_{i(k)}\, c_{i(k)} + \sum_{k=k+1}^{n} \gamma_{i(k)}\, c_{i(k)} = A + B \qquad (1)$$

1) Dans une séquence optimale ($J_{i(1)}$, $J_{i(2)}$,…, $J_{i(n)}$), la sous séquence ($J_{i(1)}$, $J_{i(2)}$,…, $J_{i(k)}$) doit être optimale pour le sous problème à k tâches ($J_{i(1)}$, $J_{i(2)}$,…, $J_{i(k)}$); sinon il pourrait être possible de réduire A et donc la somme A + B.

Cela veut dire que réordonnancer ($J_{i(1)}$, $J_{i(2)}$,…, $J_{i(k)}$) n'affecte pas le terme B

$= \sum_{k=k+1}^{n} \gamma_{i(k)}\, c_{i(k)}$.

Quand les ordonnancements sont semi-actifs et les r_i sont nuls, cela est vrai. Ils termineront à l'instant $\sum_{k=1}^{K} p_{i(k)}$. D'où $C_{i(k)}$ pour les jobs dans la séquence finale ($J_{i(1)}$, $J_{i(2)}$,…, $J_{i(n)}$) sont indépendants de la séquence ($J_{i(1)}$, $J_{i(2)}$,…, $J_{i(k)}$). On peut minimiser le terme A indépendamment de B. Si cette indépendance n'est pas suivie, l'approche programmation dynamique échoue et n'est pas optimale. L'introduction des release-date détruit l'indépendance de A et B. Un algorithme modifié de Held et Karp pour le cas des r_i non nuls existe dans la littérature [Fre82].

2) de l'indépendance des termes A et B, un ordonnancement optimal minimise A. Les k premiers jobs sont aussi considérées comme ceux du sous problème à k tâches. On peut dire que forcément l'ordonnancement optimal minimise B.

Les (n-k) derniers jobs forme un ordonnancement optimal pour le problème à (n-k) tâches.
Le terme B est minimisé tel que aucun des (n - k) tâches ne peut commencer son exécution avant $\sum_{k=1}^{K} p_{i(k)}$, avant que les k premières tâches n'eut terminé.

On peut développer une autre solution par la programmation dynamique du problème

N / 1 / $\sum_{i=1}^{n} \gamma_i(c_i)$. Pour tout ensemble de tâches Q définissons $S_Q = \sum_{J_i \notin Q} p_i$

S_Q est le " décalage" (earliest) que toute tâche de Q peut commencer son exécution si toutes les tâches qui ne sont pas dans Q doivent être exécutées les premières.

Δ(Q) : le coût minimum contracté de l'ordonnancement des tâches de Q soumis à la condition qu'aucune tâches ne commence avant S_Q. Aucune influence n'est subie des tâches qui ne sont pas dans Q sur Δ(Q).

Pour tout sous-ensemble singleton Q = { J_i }, Δ(Q) = Δ({ J_i }) = $\gamma_i(S_{\{Ji\}} + p_i)$.

La tâche seule se termine à $S_{\{Ji\}} + p_i = \sum_{k=1}^{n} p_i$,

on aura : $\Delta(\{ J_i \}) = \gamma_i(\sum_{k=1}^{n} p_i)$ (4).

En généralisant, on trouve que $\Delta(Q) = \min_{J_i \in Q} \left| \Delta(Q - \{J_i\}) + \gamma_i(S_Q + p_i) \right|$ (5)

En utilisant les formules (4) et (5), on construit Δ(Q) pour tous les sous-ensembles Q. Les approches (2), (3) pour Γ(Q), et (4) et (5) sont dites respectivement " forward dynamic programming" et " backward dynamic programming ".

4. Aspects calculatoires de la programmation dynamique

On remarque que le nombre d'opérations fait sur l'exemple soit le problème " 4 / 1 / $\overline{T}$ " est grand. Une énumération de toutes les solutions serait meilleure.

Comparons le nombre d'opérations dans une énumération complète et dans un programme dynamique.

Comptons le nombre d'opérations élémentaires qu'un ordinateur utilisera pour résoudre le problème " n / 1 / $\overline{T}$ " pour chaque méthode. En général, le temps d'exécution d'un problème est proportionnel au nombre d'opérations élémentaires.

L'énumération complète donne n! ordonnancement possibles. Pour chaque ordonnancement on calculera soit le problème $\overline{T}$ ou $\overline{nT}$ pour déterminer le minimum.

Pour calculer le " total tardiness $\overline{nT}$", pour n'importe quel ordonnancement, on doit utiliser un algorithme du type suivant :

Algorithme 2

Step1. Set k = 0, $C_{i(0)} = 0$, $\sum$ = 0.

Step2. Increment k by 1.

Step3. $C_{i(k)} = C_{i(k-1)} + p_{i(k)}$; $L_{i(k)} = C_{i(k)} - d_{i(k)}$; $T_{i(k)} = \max \{0, L_{i(k)}\}$.

Step4. Add $T_{i(k)}$ to $\sum$;

Step5. Is k = n ? If so, stop. If not go to step2.

Cet algorithme utilise :

N additions pour incrémenter k = 1,…n

N opérations pour calculer les temps d'exécutions

N soustractions pour calculer le lateness

N comparaisons pour calculer le tardiness

N additions pour $\sum$ total

N comparaisons pour voir si k = n

D'où 6n opérations sont nécessaires ou requises pour calculer le totat tardiness pour chaque ordonnancement.

Pour calculer maintenant le minimum total tardiness.

On utilisera un algorithme de type 3 suivant. Supposons que l'ensemble des n! ordonnancements possibles sont indexés par j.

Algorithme 3

Step1. Set $\sum_{\min} = \sum_{1}$, u = 1, j = 1.

Step2. Increment j by 1.

Step3. Is $\sum_{\min} \leq \sum_{j}$? If so, go to step5.

Step4. Reset $\sum_{\min}$ to $\sum_{j}$ and reset u to j.

Step5. Is j = (n!) ? If so, stop. If not, go to step 2.

A la fin de l'application de l'algorithme, u est l'indice de l'ordre optimal et $\sum_{\min}$ est le min total tardiness. On utilisera:

(n! - 1) additions pour incrémenter l'indice j, j = 1, …, n!;

(n! - 1) comparaisons pour localiser le min total tardiness;

(n! - 1) comparaisons pour voir si j = n!;

Pour résoudre le problème " n / 1 / $\overline{T}$ ", l'énumération complète nécessite

$$6\ n\ (\ n!) + 3\ (\ n! - 1\ ) \text{ opérations.} \qquad (6)$$

Considérons l'approche programmation dynamique.

On calcule Γ(Q) pour

C_n^1 sous ensembles Q de cardinalité 1. C_n^2 sous ensembles Q de cardinalité 2.

…… C_n^n sous ensembles Q de cardinalité n.

Pour contrôler une boucle avec un cycle de calculs N fois nécessite 2N opérations, 1 incrémentations et 1 comparaison pour chaque cycle.

Pour chaque sous ensemble Q, on calcule C_Q et on cycle k fois en considérant la possibilité pour chaque job d'être placé le dernier.

1 addition pour trouver C_Q

k soustractions pour chaque dernier job $L_i = C_Q - d_i$;

k comparaisons pour trouver T_i

k additions pour former pour chaque somme $\Gamma(Q - |J_i|) + \gamma_i(C_Q)$

k comparaisons du minimum des sommes, Γ(Q).

2 k opérations de contrôle de boucles pour chaque sous ensemble Q le dernier.

Soit un total de 6 k + 1 opérations. Il y a une autre boucle pour passer des sous ensembles à 1, 2, …, à n éléments. Avec 2 opérations pour chaque sous ensemble.

La programmation dynamique nécessite (6k + 3) opérations pour chaque sous ensemble à k éléments. Au total ça nécessite :

$$C_n^1 \ (6.1 + 3) + C_n^2 \ (6.2 + 3) + \ldots + C_n^n \ (6.n + 3) = 6n \cdot 2^{n-1} + 3 \cdot (2^n - 1). \qquad (7)$$

Pour résoudre le problème " n / 1 / $\overline{T}$ ",

l'énumération complète nécessite 6 n (n!) + 3 (n! - 1) opérations.

La programmation dynamique nécessite $6n\ 2^{n-1} + 3\ (2^n - 1)$ opérations.

Pour appréhender la différence, dressons le tableau suivant.

Nombre d'opérations requis

N	**Enumération complète**	**Programmation dynamique**
4	647	237
10	2286×10^8	33789
20	2.992×10^{20}	6.396×10^7
40	1.983×10^{50}	1.352×10^{14}

Chapitre 6 : Le flow shop déterministe

1. Introduction

Dans un atelier, un ensemble de machines est dit constituant un « flow-shop » si elles sont disposées en série ou numérotées de tel façon que, pour chaque job considéré, une opération k est exécutée sur une machine de rang supérieur que l'opération j si j > k. Le nombre de machines est de deux ou plus de deux. Un exemple de tel atelier est la disposition en ligne où les travailleurs ou les stations de travail représentent les machines. N'importe quel groupe de machines servis par un convoyeur unidirectionnellement et non-cycliquement sera considéré comme un flow-shop [Con67]. Il n'est pas requis que chaque job a une opération sur chaque machine. Dans ce cas, le temps d'exécution du job sur la dite machine est nul. La seule condition est que tout mouvement entre les machines dans l'atelier soit dans une direction uniforme. On suppose que { 1, 2,..., n } est un ensemble de n jobs à exécuter sur "m" machines ($m \geq 2$) disposées en série ou flow shop. Les machines sont installées dans l'ordre 1, 2, .. , m. Les jobs sont supposés être dans l'atelier à l'instant zéro.

Dans un flow shop, il est suffisant de considérer que les ordonnancements de permutation, appelés aussi ordonnancement en liste. Ils sont complètement décrit par une permutation particulière des numéros d'identification de jobs. C'est le cas où il y a une préservation de l'ordre des jobs sur les machines initiales et terminales.

Les résultats évoqués ci dessous sont cités dans la majorité des livres d'ordonnancement, voire [Con67], [Bak74], [Cof76], [Fre82], [Bla92], [Tan94] et [Pin95].

2. Le Flow-shop avec capacité de stockage intermédiaire entre les machines non limitée: Quelques résultats.

Les contraintes technologiques sans un flow shop ont la forme suivante

Job	Ordre d'exécution				
J_1	M_1	M_2	M_3	…	M_m
J_2	M_1	M_2	M_3	…	M_m
…			….		
J_n	M_1	M_2	M_3	…	M_m

Le premier résultat montré et que le problème d'ordonnancement de longueur minimale dans un flow shop m-machines ($m \geq 3$) est NP-complet. Pour m = 2, il existe un algorithme

efficace pour trouver de tel ordonnancements. L'algorithme de Johnson nécessite au plus un temps proportionnel à n logn. Le second résultat montre que le problème de la détermination d'un flow time pondéré minimale dans un flow shop m-machines est NP-complet pour $m \geq 2$.

Le problème d'un flow shop à 3-machines est NP-complet aussi bien si la longueur des entrées est mesurée par la somme des longueurs de tâches. Il se formule comme un problème de 3-partition.

C'est le problème noté F m // C_{max}. Soit $j_1, j_2, ..., j_n$ une permutation de n jobs sur m machines flow shop. Le temps de fin d'exécution du job j_k sur la machine i peut être facilement calculé par les équations de récurrence:

$$C_{i,j1} = \sum_{l=1}^{i} p_{l, j1}, \qquad i = 1,..., m;$$

$$C_{,ijk} = \sum_{l=1}^{k} p_{l, jl}, \qquad k = 1,..., n$$

$$C_{i,jk} = \text{Max} (C_{i-1,jk}, C_{i,jk-1}) + p_{i, jk}, \quad i = 2,.., m; \quad k = 1, ..., n.$$

Lemme 1 : Dans un flow shop où on désire minimiser le Makespan, ordonnancer les jobs selon la permutation $j_1, ..., j_n$ donne le même makespan que celui de la permutation $j_n, ..., j_1$.

Théorème 1 : Pour minimiser n'importe quelle fonction objective, il est *suffisant* de considérer que les ordonnancements où le même ordre de jobs est prescrit sur les deux premières machines.

Théorème 2 :[Con67]

Dans le cas du problème n / m / F / C_{max} , il est suffisant de considérer que les ordonnancements où le même ordre d'exécution des jobs est prescrit sur la machine 1 et 2, et le même ordre des jobs est prescrit sur les machines m - 1 et m.

Remarque1 : C'est un résultat non nécessaire. On ne doit pas exclure la possibilité qu'il peut y avoir des ordonnancements optimaux qui ont des ordres différents sur les deux premières machines. Il peut aussi exister un ordonnancement optimal où l'ordre optimale sur la première machine et la deuxième machine est différent que celui de la machine m - 1 et m.

Pour toute mesure de performance, et en particulier pour le flow-time pondéré, le maximum flow-time, le flow-shop à deux machines est le seul cas où il est suffisant de considérer que les ordonnancements de permutation.

Mais un contre exemple indique aussi que pour le maximum flow-time, dans les flow-shop à quatre ou plus de machines un type général d'ordonnancement doit être considéré.

3. Le flow shop à 2 - machines. L'algorithme de Johnson

Chaque job consiste en la donnée d'un couple (A_i, B_i) où :

A_i est le temps d'exécution du job i sur la machine1

B_i représente le temps d'exécution du job i sur la seconde machine.

Cet ordre sur les machines est le même pour chacun des n jobs, aussi il est permis qu'un A_i ou un B_i soit nul, puisque certain jobs peuvent avoir qu'une seule opération. Les contraintes normales d'un atelier simple sont supposés réalisées.

1) Chaque machine j ne peut exécuter qu'un job i à la fois et
2) Chaque job i ne peut être exécuté que par une machine en tout instant. Le job i ne s'exécutera sur la machine 2 que s'il a terminé son exécution sur la machine 1.

Le problème alors est : Soit 2 n valeurs A_1, A_2, ... , A_n, B_1, B_2, ..., B_n. Trouver un ordre d'exécution des jobs sur chacune des deux machines tel que ni les contraintes de précédences ou d'occupation ne soient violés et le maximum des $F_i = C_i - r_i$ est rendu aussi petit soit-il.

La préemption et l'oisiveté des machines ne sont pas autorisées.

La makespan d'une séquence de jobs peut être représenté par le temps de fin d'exécution d'un réseau PERT.

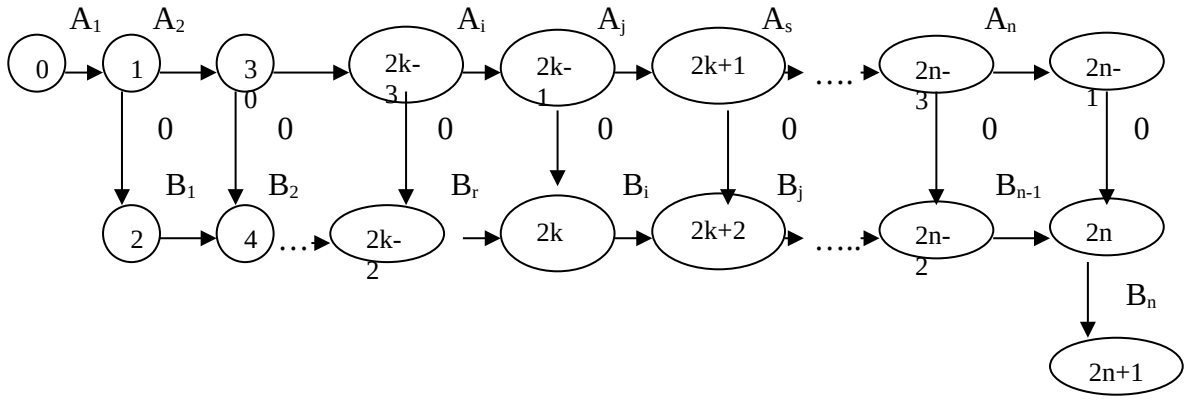


Fig. 1: Réseau pour la suite des jobs (j_1, j_2, …, r, i, j, s,…., j_n)

Où 0 est le temps de transport de la machine 1 à 2.

Les nœuds et représentent respectivement le début et la fon d'exécution de l'ordonnancement. Les arcs (2k - 3, 2k -1), (2k, 2k+2) décrivent l'exécution du job i sur la machine 1 et 2 respectivement.

Les arcs verticaux représentent des activités de durées nulles. La makespan est la longueur du plus long chemin de 0 à 2n +1.

On notera par :

$A_{[i]}$: le temps d'exécution sur la machine 1 du job qui est à la ième position dans la séquence d'ordonnancement. Le dernier job ne peut terminer son exécution plutôt que le temps requis pour exécuter tous les jobs sur la 1ère machine plus le temps qu'il faut pour l'exécuter sur la machine 2, soit $C_{max} \geq A_{[1]} + \sum_{i=1}^{n} B_{[i]}$. Aussi $C_{max} \geq \sum_{i=1}^{n} A_{[i]} + B_{[n]}$

$\sum_{i=1}^{n} A_{[i]}$ et $\sum_{i=1}^{n} B_{[i]}$ sont directement données par l'information et entièrement indépendant de l'ordre des jobs. Alors pour réduire ses bornes on ne peut influencer que sur $B_{[n]}$ $A_{[1]}$ par le choix de la suite. On choisira la plus petite des 2n valeurs, si c'est un A_i on mettra le job i en premier. Si c'est un B_i on le mettra à la fin de l'ordonnancement. On peut répéter essentiellement le même argument sur les (n-1) jobs restants. Si cette borne est atteinte alors cette construction donne un bon ordonnancement. Cet argument de bornes est précisément la base de la procédure de Johnson et il offre la preuve de son optimalité.

Soit $X_{[i]}$ le temps d'oisiveté sur la machine 2 précédent $B_{[i]}$.

Un ordonnancement type ressemble à ce qu'il suit :

Les valeurs de $X_{[i]}$ peuvent être données en fonctions des $A_{[i]}$ et $B_{[i-1]}$ par les relations

$X_{[1]} = A_{[1]}$

$X_{[2]} = \max (A_{[1]} + A_{[2]} - B_{[1]} - X_{[1]}, 0)$

$X_{[3]} = \max (A_{[1]} + A_{[2]} + A_{[3]} - B_{[1]} - B_{[2]} - X_{[1]} - X_{[2]} , 0)$

et en général on peut écrire

$$X_{[j]} = \max \left(\sum_{i=1}^{j} A_{[i]} \right) - \sum_{i=1}^{j-1} B_{[i]} - \sum_{i=1}^{j-1} X_{[i]} , 0).$$

Les sommes partielles des X peuvent être obtenues des expressions

$X_{[1]} = A_{[1]}$

$X_{[1]} + X_{[2]} = \max (A_{[1]} + A_{[2]} - B_{[1]} , A_{[1]})$

$X_{[1]} + X_{[2]} + X_{[3]} = \max (A_{[1]} + A_{[2]} + A_{[3]} - B_{[1]} - B_{[2]} , X_{[1]} + X_{[2]})$

$$= \max \left(\sum_{i=1}^{3} A_{[i]} \right) - \sum_{i=1}^{2} B_{[i]} , \sum_{i=1}^{2} A_{[i]} \right) - B_{[1]}, A_{[1]}).$$

En général on aura :

$$\sum_{i=1}^{j} X_{[i]} = \max\left(\sum_{i=1}^{j} A_{[i]} - \sum_{i=1}^{j-1} B_{[i]}, \sum_{i=1}^{j-1} A_{[i]} - \sum_{i=1}^{j-2} B_{[i]}, \ldots, \sum_{i=1}^{2} A_{[i]} - B_{(1)}, A_{(1)} \right)$$

$$\text{Si } Y_j = \sum_{i=1}^{j} A_{[i]} - \sum_{i=1}^{j-1} B_{[i]} \text{ alors } \sum_{i=1}^{j} X_{[i]} = \max (Y_1 , Y_2 , .., Y_j).$$

Si $F_{max}(S)$ désigne le maximum flow-time d'un ordonnancement particulier S alors

$$C_{max}(S) = F_{max}(S) = \sum_{i=1}^{n} B_{[i]} + \sum_{i=1}^{n} X_{[i]} = \sum_{i=1}^{n} B_{[i]} + \max (Y_1 , Y_2 , .., Y_n)$$

Puisque la somme des B_i est indépendante de la séquence, le flow-time maximum dépend entièrement de la somme des intervalles des temps d'oisiveté sur la seconde machine et c'est équivalent de maxY_j. Notre but est de trouver S* tel que $F_{max}(S^*) \leq F_{max}(S)$ pour tout S.

La règle qui minimise le makespan pour le problème de deux machines est celle de Johnson.

Théorème 3 : [Joh 54]

Dans un flow-shop n/2/F/C_{max} où tous les jobs sont disponibles à la même date,
le job j précédera le job j+1 si Min (A_j,B_{j+1}) < Min (A_{j+1},B_j).

Algorithme 1 (de Johnson)

Step1. Set k = 1, l = n

Step2. Set current list of unscheduled jobs { J_1, J_2, …, J_n }.

Step3. Find the smallest of all the a_i and b_i times for the jobs currently unscheduled.

Step4. If the smallest time is for J_i on first machine, i.e. a_i is smallest, then

i) Schedule J_i in kth position of processing sequence

ii) Delete J_i from current list of unscheduled jobs

iii) Increment k to k + 1

iv) Go to step6.

Step5. If the smallest time is for J_i on second machine, i.e. b_i is smallest, then

i) Schedule J_i in the lth position of processing sequence

ii) Delete J_i from current list of unscheduled jobs

iii) Redule l to(l - 1)

iv) Go to step6.

Step6. If there are any jobs still unscheduled, go to stpe3. Otherxise, stop.

If the smallest time occurs for more than one job in step3, the pick J_i arbitrary.

Exemple 2 : Ordonnancer les tâches dans le problème 7 / 2 / F / C_{max}

Job	M_1	M_2
1	6	3
2	2	9
3	3	
4		3
5	1	8
6	7	1
7	4	5
	7	6

Job 4 ordonnancé : 4 – – – – – –

Job 5 ordonnancé : 4 – – – – – 5

Job 2 ordonnancé : 4 2 – – – – 5

Job 3 ordonnancé : 4 2 – – – 3 5

Job 1 ordonnancé : 4 2 – – 1 3 5

Job 6 ordonnancé : 4 2 6 – 1 3 5

Job 7 ordonnancé : 4 2 6 7 1 3 5

La procédure décrite auparavant découle directement de cette inégalité. L'ordonnancement optimal résultant est unique s'il y a des inégalités entre chaque couple de jobs, mais il peut y avoir un ensemble d'ordonnancement optimales s'il y a égalité entre quelques couples de jobs. La preuve se fait en deux parties (Voir Conway et al.[Con67]).

Théorème 4 : pour le problème n / 2 / F / C_{max} avec $p_{i1} = a_i$ et $p_{i2} = b_i$, i = 1, …, n.

i) si $a_k = \min \{ a_1, \ldots, a_n, b_1, \ldots, b_n \}$ alors il y a un ordonnancement optimal où le job J_k est placé à la première position de l'ordonnancement.

ii) si $b_k = \min \{ a_1, \ldots, a_n, b_1, \ldots, b_n \}$ alors il y a un ordonnancement optimal où le job J_k est placé à la dernière position de l'ordonnancement.

La règle de Johnson peut se formuler d'une autre manière. Divisons les jobs en deux sous ensembles. L'ensemble 1 contient tous les jobs tel que $p_{1\,j} < p_{2\,j}$ et l'ensemble deux contient tous les jobs tel que $p_{1\,j} > p_{2\,j}$. S'il y a égalité le job est dans un des deux sous ensembles. On

exécute les jobs du premier ensemble dans l'ordre croissant des p_{1j} (SPT) et suivent les jobs du second ensemble dans l'ordre décroissant des p_{2j} (LPT). L'ordonnancement peut ne pas être unique .

Théorème 5 : Tout ordonnancement SPT(1)-LPT(2) est optimal pour F2//C_{max}.

4. Le flow shop général sur 2-machines, n / 2 / G / C_{max}. Algorithme de Johnson

On a supposé que les tâches doivent s'exécuter sur toutes les machines. Relaxant cette contrainte. Supposons qu'on a un ensemble de jobs { J_1, …, J_n }qu'on peut partitionner en quatre sous ensembles de jobs.

Type A : les jobs qui ne sont exécutés que sur la machine M_1

Type B : les jobs qui ne sont exécutés que sur la machine M_2

Type C : les jobs qui sont exécutés sur la machine M_1 puis sur M_2

Type D : les jobs qui sont exécutés sur la machine M_2 puis sur M_1

La solution optimale est de la forme :

1) ordonnancer les jobs de type A dans un ordre quelconque pour obtenir une séquence S_A,
2) ordonnancer les jobs de type B dans un ordre quelconque pour donner une séquence S_B
3) ordonnancer les tâches de type C selon la règle de Johnson pour donner S_C
4) ordonnancer les jobs de type D selon la règle de Johnson pour donner S_D (en permutant le rôle des machines, la machine 2 devient machine 1' et machine 1 devient 2').

Un ordonnancement optimal est alors

Machine	Ordre d'exécution
M_1	(S_C, S_A, S_D)
M_2	(S_D, S_B, S_C)

Exemple 3 :

Soit 9 / 2 / G / C_{max} dont les données sont l'ordre de passage sur les machines et les temps d'exécution des tâches.

Tâche	**Première machine**		**Seconde machine**	
1	M_1	8	M_2	2

2	M_1	7	M_2	5
3	M_1	9	M_2	8
4	M_1	4	M_2	7
5	M_2	6	M_1	4
6	M_2	5	M_1	3
7	M_1	9	-----	
8	M_2	1	-----	
9	M_2	5	-----	

Pour déterminer un ordonnancement optimal.

Les jobs de type A : job 7 sur la machine M_1

Les jobs de type B : job 8 et 9. Choisissons l'ordre (8, 9).

Les jobs de type C : job 1, 2, 3 et 4. La règle de Johnson donne (4, 3, 2, 1).

Les jobs de type D : job 5 et 6. La séquence selon Johnson est (5, 6).

L'ordonnancement optimal est:

Machine M_1	(4, 3, 2, 1, 7, 5, 6)
Machine M_2	(5, 6, 8, 9, 4, 3, 2, 1)

Par un diagramme de Gantt, F_{max} vaut 44 unités.

5. Minimiser le flow-time moyen dans le flow-shop à deux machines (n / 2/ F / $\overline{F}$)

Aussi pour le flow-shop, la minimisation du flow-time moyen est un problème très difficile. Quoique le théorème de Johnson reste applicable, il est nécessaire de considérer seulement les ordonnancements où la séquence des jobs est la même sur chaque machine. Aucun algorithme constructif comparable à celui de Johnson n'est connu. La procédure de Johnson n'est pas optimale pour le critère et en général elle n'est pas assez bonne. Des exemples existent où SPT est meilleure que celle de Johnson. Une expression générale pour le flow-time pondéré de n jobs peut être obtenu par un argument qui suit directement la règle de Johnson.

1) Soit $X_{[i]}$ le temps d'oisiveté qui précède l'opération $B_{[i]}$ sur la seconde machine.

2) Soit $Y_j = \sum_{i=1}^{j} A_{[i]} - \sum_{i=1}^{j-1} B_{[i]}$, le flow-time de chaque job est:

$F_{[1]} = A_{[1]} + B_{[1]} = X_{[1]} + B_{[1]}$

$F_{[2]} = X_{[1]} + B_{[1]} + X_{[2]} + B_{[2]}$

$F_j = \sum_{i=1}^{j} B_{[i]} + \sum_{i=1}^{j} X_{[i]} = \sum_{i=1}^{j} B_{[i]} + \max (Y_1 , Y_2 , .., Y_J)$

$\overline{F} = 1/n \left(\sum_{j=1}^{n} \sum_{i=1}^{j} B_{[i]} + \sum_{j=1}^{n} \max (Y_1 , Y_2 , .., Y_J) \right)$.

On peut énoncer que si deux jobs i et j sont adjacents et si $A_j \geq A_i$ et $B_j \geq B_i$ alors le job i précède le job j.

6. Cas de trois machines n / 3 / F / C_{max}

La règle de Johnson peut être étendue au cas de trois machines. On aura besoin des conditions dites de de dominance, $\min_i a_i \geq \max_i b_i$ ou $\min_i c_i \geq \max_i b_i$.

Dans ce cas, on regroupe deux machines en une seule et le problème devient celui à deux machines en posant $a'_i = a_i + b_i$ et $b'_i = b_i + c_i$.

Exemple 4: Soit 6 / 3/ F / C_{max}. Les valeurs données sont celles des temps d'exécution.

Job	M_1	M_2	M_3	1ère machine	2nd machine
1	4	1	3	5	4
2	6	2	9	8	11
3	3	1	2	4	3
4	5	3	7	9	10
5	8	2	6	10	8
6	4	1	1	5	2

$\min_i (a_i) = 3$, $\max_i (b_i) = 3$ et $\min_i (c_i) = 1$. D'où les formules de dominance sont vérifiées

M_1	J_2	J_4	J_5	J_1	J_3	J_6
	6	11	19	23	26	30

M_2	J_2	J_4	J_5	J_1	J_6
	6 8	14	21	27	31

M_3		J_2	J_4	J_5	J_1	J_3	J_6
	8	17	24	30	33		35

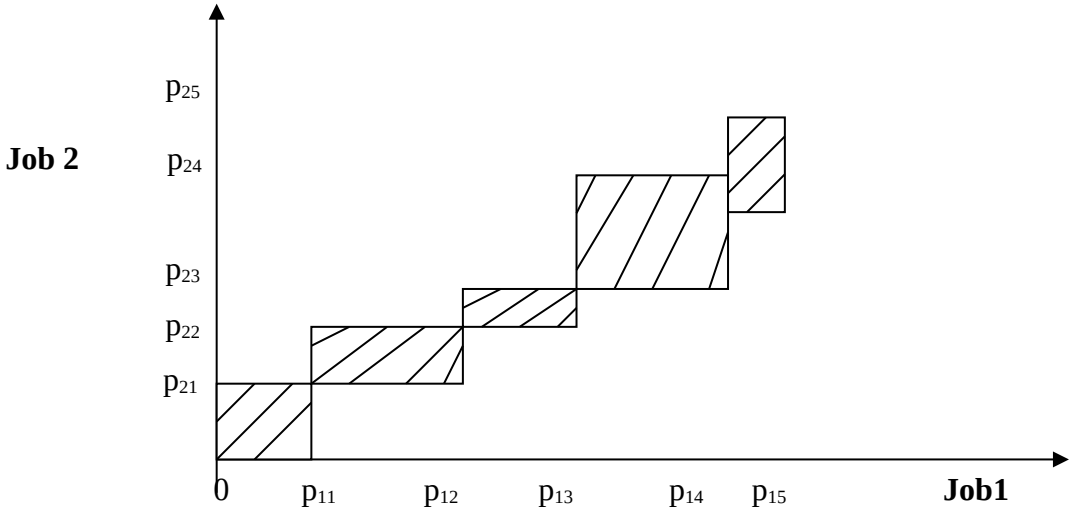
Fig 3 : Diagramme de Gantt

(2, 4, 5, 1, 3, 6) est l'ordonnancement optimal.

7. Solution graphique d'Akers pour 2 / m / F / C_{max}

C'est le cas de 2 jobs et " m " machines, m = 5.

L'axe des abscisses est représenté par le job 1(p_{11}, p_{12}, p_{13}, p_{14} et p_{15}) et des ordonnées par le job 2 ((p_{21}, p_{22}, p_{23}, p_{24} et p_{25}).



Les zones hachurées est une zone d'interdiction. La même machine ne peut pas exécuter deux tâches en même temps. Un ordonnancement est représenté par le diagramme en traçant une ligne de 0 (0, 0) à ($\sum_{j=1}^{5} p_{1j}$, $\sum_{j=1}^{5} p_{2j}$).

Un ordonnancement peut être composé de segments de droites.

i) horizontalement, représentant l'exécution sur la tâche 1 seulement.

ii) Verticalement, représentant l'exécution de la tâche 2 seulement

iii) A 45° oblique, représentant l'exécution sur les deux machines.

Le maximum flowtime est : $C_{max} = \sum_{j=1}^{5} p_{1j}$ + la somme des longueurs de segments verticaux

$= \sum_{j=1}^{5} p_{2j}$ + la somme des longueurs de segments horizontaux.

Définition 1 : Une machine « i » domine la machine « i + 1 » si

$$\min_{j \in \{1,\ldots,n\}} p_{ij} \geq \max_{j \in \{1,\ldots,n\}} p_{i+1,j} .$$

On représente cette forme par $\overline{p_i} >_d \overline{p_{i+1}}$.

Théorème 6 : Si $\overline{p_i} >_d \overline{p_{i+1}} >_d \ldots >_d \overline{p_{i+l}}$ alors la séquence optimale ne change pas si les machines i+1, .., i + l sont remplacés par une seule machine sur laquelle le temps d'exécution est la somme des temps d'exécution des l machines.

Théorème 7 : Pour Fm/ Prmu , $p_{ij} = p_j / C_{max}$, le makespan C_{max} est égale à

$\sum_{j=1}^{n} p_j$ + (m-1) max(p_1,..., p_n) et il est indépendant de tout ordonnancement.

Si la vitesse d'une machine i est v_i alors le temps d'exécution du job j sur la machine i est $p_{ij} = p_j / v_i = \alpha_i p_j$.

Définition 2 : La machine avec la plus petite vitesse est appelée *"machine goulot" ou "Bottleneck machine".*
Un ordonnancement est proportionné si $p_{1j} = p_{2j} = \ldots = p_{mj} = p_j$.

Théorème 8 : Si la première (respectivement la dernière) machine est de goulot dans un flow shop proportionné avec des vitesses différentes alors LPT (resp. SPT) minimise le makespan.

8. Flow shops avec des capacités intermédiaire de stockage limité [Pin 95]

On suppose que les capacités de stockage sont nulles entre les machines successives. Le phénomène de " blocage " peut se produire. Si une machine termine l'exécution d'un job quelconque, le job ne peut pas aller à la prochaine machine si elle est occupée mais reste sur la même machine, d'où la notion de blocage de jobs. Le problème se note Fm/ block /C_{max} .

Lemme 2 : Si $p^{(1)}_{ij} = p^{(2)}_{m+1-i,\,j}$ alors la séquence $j_1, j_2,..., j_n$ dans le premier ordonnancement flow shop donne le même makespan que la séquence $j_n, j_{n-1},..., j_1$ dans le second ordonnancement flow shop.

L'algorithme pour résoudre $1 / S_{jk} / C_{max}$ avec $S_{jk} = | a_k - b_j |$ s'utilise aussi pour
$F2/ block /C_{max}$ et est de $o(n^2)$.

Pour le problème $Fm/ block , p_{ij} = p_j /C_{max}$.
Théorème 9 : Un ordonnancement est optimal pour $F2/ block, p_{ij} = p_j / C_{max}$ si et seulement si c'est un ordonnancement SPT-LPT.
Pour le cas de $Fm/ block /C_{max}$ l'heuristique PF (Profile fitting) est la plus usitée .
* $Fm/ nwt /C_{max}$.(nwt : no-wait désigne sans attente)

Théorème 10 : $F3//C_{max}$ est fortement NP-difficile.
Aussi $F m//C_{max}$ est fortement NP-difficile ($m \geq 3$).

Chapter 07 : Parallel processor scheduling

1. Introduction

The process of scheduling in general requires both sequencing and resource allocation decisions. When there is only a single resource, as in the single machine model, the allocation of that resource is completely determined by sequencing decisions. Consequently there is no distinction between these two decision problems in the models covered in this chapter.

Any schedule can be improved by taking advantage of parallel operations on identical machines. One could view m machines working simultaneously on a single job as a single machine with m times the power of the basic machine. It is better to provide required capacity with a single machine than with an equivalent number of separate machines but considerations of reliability work is opposite. Jobs are supposed independent, that they arrive simultaneously at the workshop and that there were more than one machine to perform the processing. The capabilities of m machines in this situation might be described by giving a n by m matrix of processing-times p_{ij}, the time to perform the single operation of job i on machine j, assuming that machine j performed the entire operation. The simple case occurs when the machines are identical, indicated by having all the elements of a given row of the matrix and the second subscript on p can be omitted.

In general one can consider machines that are different by having different values in a particular row of the matrix, and admit the possibility that a particular machine might be unable to perform a particular task by making the corresponding p_{ij} prohibitively large.

It is, of course, necessary only to consider matrices that cannot be partitioned into independent sub-matrix, since the machines and jobs of an independent sub-matrix would be treated as a separate problem. The key question is whether or not individual jobs may be divided among two or more machines. If this is not possible, then the problem is one of partitioning the n jobs into m distinct subsets and determining sequence within each subset.

If some operation of a job is allowed, then better schedules are possible, but the determination of the schedule is usually more difficult. The three main criteria to be analyzed are schedule length or makespan, mean flow time and lateness.

When dealing with machines in parallel, the makespan becomes an objective of interest. Balancing the load on machines in parallel and by minimizing the makespan the scheduler ensures a good balance. We may consider scheduling parallel machines as a two-step process. First, one has to determine which jobs are to be allocated to which machines ; second, one has

to determine the sequence of the jobs allocated to each machine. With the makespan criteria only the allocation process is important.

We will characterize the machines or processors. They may be either *parallel*, i.e. performing the same functions or *dedicated*, i.e. specialized for the execution of certain tasks. Three types of parallel processors are distinguished depending on their speeds. If all processors have equal task processing speed, then we call them *identical*. If the processors differ in their speeds, but the speed of each processor is constant and does not depend on the tasks then they are called *uniform*. Finally, if the speeds of the processors depend on the particular task processed, then they are called *unrelated.*

More generally, suppose that there are m identical machines, and m jobs to be performed, each with processing-time p then $\overline{F}$ = p. There is a total of m x p works to be done, and if this is divided equally among the m machines, they will all finish simultaneously after p time units, regardless of how the jobs are assigned to individual machines. However, this assignment does affect the times at which the job finish. If each job is divided among all m machines, then the first job will finish at time p/m, the second at time 2p/m, the third 3p/m, until finally the last job is finished at time p.

The average flow-time is given by : $\overline{F} = \frac{p}{m}\left(\frac{1}{m}+\frac{2}{m}+\frac{3}{m}+\ldots+\frac{m}{m}\right) = \frac{m+1}{2m}p < p.$

In scheduling problems it is often possible to take advantage of parallelism in resource structure. Applications may envisioned in the area of repetitive processing sequences, or at the systems architecture level, when the cycle time of every operation is known in advance.

There are many examples of parallel machines when jobs may not be divided among machines. Teller's window at a bank, checkout counters at a supermarket, reservation desks at an air terminal, and toll booths on a highway.

2. Minimizing Makespan

21. Parallel identical processors and independent jobs

211. The Makespan without preemptions

This part is addressed to « parallel machine systems ». This section treats parallel machine involving identical processors and independent jobs. The time required to execute a given task does not depend on the processor used. The execution of a task, once started, cannot be interrupted.

Suppose that jobs are formed of single operation available at time zero. There is n jobs and there are m identical machines available for processing, and that a job can be processed by at most one machine at a time. We will analyze the schedule length criterion, called makespan.

Problem P2 // C_{max}

The problem considered is P// C_{max} where a set of independent tasks is to be scheduled on identical processors in order to minimize schedule length. The problem is not easy to solve since even simple cases such as scheduling on two processors is proved to be NP-hard [Kar72]. The problem P2 // C_{max} is NP-hard.

In the single machine model, there is no makespan problem. The makespan is equal to a constant for any sequence of n given jobs. In the multiple-processor case, the makespan problem is no longer trivial. If job preemption is prohibited, the problem of minimizing makespan is somewhat more difficult. No direct algorithm has been developed for calculating the optimal makespan or for constructing an optimal schedule.

One of the most often used general approximation strategies for solving scheduling problems is *List Scheduling*, whereby a priority list of the tasks is given, and at each step the first available processor is selected to process the first available task on the list[Gra66]. The accuracy of a given list scheduling algorithm depends on the order in which tasks appear on the list. Unfortunately, this strategy may result in an unexpected behavior of constructed schedules, since the schedule length for problem P / prec/ C_{max} (with arbitrary precedence constraints) may increase if :

- the number of processors increases
- task processing time decrease,
- precedence constraints are weakened, or
- the priority list changes.

These list scheduling anomalies have been discovered by Graham[Gra66], who has also evaluated the maximum change in schedule length that may be induced by varying one or more problems parameters. Let there defined a task set T together with precedence constraints <. Let the processing times of the tasks be given as a vector p, Let T be scheduled on m processors using list L, and let the obtained value of schedule length be equal to C_{max} .

Let the above parameters be changed : a vector of processing times $p' \leq p$ (for all the components), relaxed precedence constraints $<' \subseteq <$, priority list L' and the number of processors m'. Let the new value of schedule length be C'_{max}.

Then the following theorem holds.

Theorem 1 [Gra66] : Under the above assumptions,

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

From the above theorem, the *absolute performance ratio* for an arbitrary list scheduling algorithm solving problem P // C_{max} can be derived.

Let P = { P_1, P_2,…, P_M } denote a set of M identical processors. We use P_i to represent both the ith processor and the set of tasks assigned to it by an algorithm. We define P_i* to be the set of tasks assigned to the ith processor by an optimal schedule.
Let T = { T_1, T_2, …, T_N } denote a set of N independent tasks. T_{Last} represents any task finishing last according to a schedule of T.
We use $w_{opt}(T)$ to indicate the optimal finish time for T and define $w_{LPT}(T)$ and $W_{alg}(T)$ correspondingly. For a given M, let R_{algo} denote the least real number such that:

$R_{algo} \geq \frac{\omega_{algo}(T)}{\omega_{opt}(T)}$ for all task sets T or the *absolute performance ratio* or the *worst-case performance ratio*.

We seek an upper bound on the length of the schedule produced expressed relative to an optimal assignment of tasks to processors. The algorithm may not discover the best schedule, but will always provide results close to the optimum.

Corollary 1 [Gra66]

For an arbitrary list scheduling algorithm LS for P :// C_{max} we have $R_{LS} = 2 - 1/ m$.

A simple yet effective heuristic procedure for constructing a schedule involve the use of LPT rule. The *longest processing time first*.
An arbitrary list scheduling algorithm can produce schedules almost twice as long as optimal ones. One of the simplest algorithms is the LPT algorithm in which the tasks are arranged in order of non-increasing p_j.

Algorithm 1. *LPT Algorithm for P // C_{max}.*

begin
Order tasks on a list in non-increasing order of their processing times ;
-- i.e. $p_1 \geq ... \geq p_n$
for i = 1 step 1 until m do $s_i = 0$;
-- processors P_i are assumed to be idle from time $s_i = 0$ on, i = 1,..., m
j : = 1
repeat
s_k : = min { s_i } ;
Assign task T_j to processor P_k at time s_k ;
-- the first non-assigned task from the list is scheduled on the first processor
-- that becomes free
s_k : = s_k + p_j ; j : = j + 1 ;
until j = n ; -- all tasks have been scheduled
end ;

The time complexity of this algorithm is O(nlogn). The LPT heuristic procedure cannot guarantee an optimal makespan but is efficient. The worst case behavior of the LPT rule is analyzed in below.

Theorem 2 [Gra66]

If the LPT algorithm is used to solve P // C_{max}, then $R_{LPT} = \frac{4}{3} - \frac{1}{3m}$.

This means that the length of an LPT schedule can be at most about 33% greater than optimal.

A better performance from the LPT algorithm is expected when the number of tasks becomes large.

In [Cof76] another absolute performance ratio for the LPT rule was proved, taking into account the number k of tasks assigned to a processor whose last task terminates the schedules ;

Theorem 3 [Cof76b]

For the assumptions stated above, we have R_{LPT} (k) = 1 + 1/k - 1 / km.

*) The algorithm of Langston[Lan82] starts from LPT algorithm and then determines whether the input may be “bad” for LPT and iteratively exchanges tasks between processors to produce a better schedule. It reduces the worst-case performance ratio of LPT from

$\frac{4}{3} - \frac{1}{3m}$ to $\frac{5}{4} + \frac{1}{12}(2)^{-k}$ in O(kNlogM) time, where M is the number of processors, N the number of tasks and k a user-specified parameter of the algorithm.

It would be of interest to know how good the LPT algorithm is on the average.
A result was obtained by [Cof84], where the relative error was found for two processors on the assumption the task processing times are independent uniformly distributed on [0,1].

Theorem 4 [Cof84] : Under the assumptions already stated, we have the following bounds for the mean value of schedule length for the LPT algorithm, $E(C_{max}^{LPT})$, for P2 // C_{max}.

$$n/4 + 1 / 4(n+1) \leq E(C_{max}^{LPT}) \leq n/ 4 + e/ 2(n+1)$$

where e = 2,7... is the base of natural logarithm.

The existence of an optimization polynomial time for solving P 2 // C_{max} is negative. We may try to find a pseudo-polynomial algorithm . It appears that, based on a dynamic programming approach, such an algorithm can be constructed using ideas presented by Rothkopf [Rot66]. It uses Boolean variables $x_j (t_1, t_2, .., t_m)$, j = 1, 2, .., n, t_i = 0, 1, .., C, i = 1, 2, .., m, where C denotes an upper bound on the optimal schedule length C^*_{max} . The meaning of the variable is the following

$$x_j (t_1, t_2, .., t_m) = \begin{cases} \text{true if tasks } T_1, T_2, ..., T_j \text{ can be scheduled on processors} \\ P_1, P_2, ..., P_j \text{ in such a way that } P_i \text{ is busy in time} \\ \text{interval} [0, t_i], i = 1, 2, ..., m. \\ \text{false otherwise} \end{cases}$$

Now, we are able to present the algorithm. It solves problem P // C_{max} in $O(nC^m)$ time ; thus for fixed m it is a pseudo-polynomial time algorithm.

Algorithm 2. *Dynamic programming for P// C_{max} [Rot66]*

begin
for all $(t_1, t_2, ..., t_m) \in \{0, 1, ..., C\}^m$ do $x_0 (t_1, t_2, ..., t_m) := false$;
$x_0 (0, 0, ..., 0) := true$;
-- initial values for Boolean variables are now assigned
for j = 1 step 1 until n do
for all $(t_1, t_2, ..., t_m) \in \{0, 1, ..., C\}^m$ do

$$x_j(t_1, t_2, ..., t_m) = \bigcup_{i=1}^{m} x_{j-1} (t_1, t_2, ..., t_{i-1}, t_i - p_j, t_{i+1}, ..., t_m) \quad *)$$

$C^*_{max} := \min \{ \max \{ t_1, t_2, ..., t_m\} / x_n (t_1, t_2, ..., t_m) = true\}$;
-- optimal schedule length has been calculated

Starting from the value C^*_{max} assign tasks T_n, T_{n-1}, ..., T_1 to appropriate processors using formula *) backwards ;
end ;

The same problem is formulated as an integer programming problem.

Let y represent schedule makespan and let x_{ij} be an indicator variable defined as follows :

$x_{ij} = 1$ if job i is assigned to machine j and $x_{ij} = 0$ otherwise. Then a feasible assignment will assign job i to only one machine, so that $\sum_{j=1}^{m} x_{ij} = 1$, $1 \le i \le n$.

The makespan must be at least as large as the processing time assigned to any machine, so that : $y \ge \sum_{i=1}^{n} t_i x_{ij}$, $1 \le j \le m$.

The complete formulation is :

$$\text{minimize} \quad y$$

$$\text{subject to} \quad y - \sum_{i=1}^{n} t_i x_{ij} \ge 0 , \quad 1 \le j \le m.$$

$$\sum_{j=1}^{m} x_{ij} = 1 , \quad 1 \le i \le n.$$

$$x_{ij} \ge 0 \text{ and integer.}$$

This formulation contains (m + n) constraints in m n + 1 variables, but none of its computational properties have been investigated by researchers.

Problem $P / p_i = 1 ; M_j / C_{max}$

Another generalization of the $P / / C_{max}$ problem that is of practical interest arises when job j is only allowed to be processed on subset M_j of the m machines in parallel is $P / p_i = 1 ; M_j / C_{max}$ problem. Assume that the set M_j is *nested*, that is, one and only one of the following four conditions holds for jobs j and k.

i. M_j is equal to M_k ($M_j = M_k$)

ii. M_j is a subset of M_k ($M_j \subset M_k$)

iii. M_k is a subset of M_j ($M_k \subset M_j$)

iv. M_j and M_k do not overlap ($M_j \cap M_k = \oslash$)

Under these conditions, a well known dispatching rule, the *least flexible job first* (LFJ) rule is of importance. The LFJ rule selects, every time a machine is freed, the job that can be

processed on the smallest number of machines, that is the least flexible job. Ties may be broken arbitrary.

The LFJ rule is optimal for $P / p_i = 1 ; M_j / C_{max}$ when the M_j sets are nested [Pin95]

Problem $P / p_i = 1 ; r_i ; d_i / C_{max}$

Consider the following decision problem : Given some threshold value L does there exist a schedule such that $\max_{i=1,...,n} L_i = \max_{i=1,...,n} (C_i - d_i) \leq L$.

We want to find such a schedule if it exists.

This inequality holds if and only if $C_i \leq d_i^L = L + d_i$ for all i = 1,..., n. All jobs i must finish before the modified due dates d_i^L and cannot start before the release times r_i, i.e. each job J_i must be processed in an interval $[r_i , d_i^L]$ associated with J_i.

We call these intervals *time windows.*

A list is a permutation $\pi : \pi(1), \pi(2), ..., \pi(n)$ of all jobs. A corresponding list schedule is constructed by the following algorithm. It is convenient for this algorithm to number the machines from 0 to m-1. The starting time of job i in the schedule is denoted by x(i), h(i) denotes the machine on which i is to be processed, and t(j) is the finish time of the last job on machine j.

Algorithm 3. *Cyclic list schedule*

```
For j : = 1 to m  do t(j) : = 0 ;
For i : = 1 to n  do
      Begin
      Schedule job i on machine h(i) : = i ( mod m ) at time
      x(i) : = max { t(h(i), r_i  } ;
      t(h(i)) : = x(i) + 1
     End
```

We need to consider only cyclic schedule if we want to solve a problem of the form

$P / p_i = 1 ; r_i ; d_i / f$ with regular objective function f, i.e. $f = C_{max}$.

Lemma [Bru95] : Let f be a regular objective function and assume that $P / p_i = 1 ; r_i ; d_i / f$ has a feasible solution. Then there always exists a cyclic schedule which is optimal.

We will present an algorithm which constructs a schedule (x(i), h(i) respecting all time windows $[r_i , d_i]$ (i.e. $r_i \leq x(i) < x(i) + 1 \leq d(i)$ for i = 1,.., n) or finds that such a schedule does not exist. If such a schedule exists, then the schedule constructed by the algorithm minimizes C_{max} as well $\sum C_i$.

The idea of the algorithm is to construct an optimal list $\pi(1), \pi(2),..., \pi(n)$. We try to schedule at the current time tan available job i with smallest due date.

We have to define a *crisis subroutine* for *crisis job i.* If $d_i < t + 1$ (i.e. the job i is late), we call the crisis subroutine. It backtracks over the current partial schedule searching for a job $\pi(j)$ with a highest position j that has a deadline greater than that of crisis job i. If such a job $\pi(j)$ does not exist, the subroutine concludes that there is no feasible schedule and halts. Otherwise, we call $\pi(j)$ a *pull job* and the set of all jobs in the partial schedule with positions greater than j a *restricted set.* The subroutine determines the minimum release time r of all jobs in the restricted set and creates *barrier* (j, r). This barrier is a restriction on the starting time of jobs scheduled in positions $k \geq j$.

It is added to a barrier list which is used to calculate the current scheduling time. Finally, j and all jobs in the restricted set are eliminated from the partial schedule and we continue with the partial schedule $\pi(1),..., \pi(j-1)$. If $d_i \geq t + 1$ the job i is scheduled at time t on machine $h(i) = i \pmod m$.

Let U be the set of unscheduled jobs, barrier list contains all barriers, and $\pi(j)$ denotes the j-th job currently scheduled for j = 1, 2, ..., k-1.

Algorithm 4. *Algorithm P / $p_i = 1; r_i; d_i$ / C_{max} or $\sum C_i$ [in Bru95]*

Initialize
barrier list : = $\oslash$; k : = 1; U : = { 1,..., n }
WHILE there are unscheduled jobs do
BEGIN
Calculate current time t ;
IF $1 \leq k \leq m$ THEN t_1 : = t (k (mod m));
t_2 : = min { r_j / j is an unscheduled job };
t_3 : = max { r / (j, r) is a barrier ; $1 \leq j \leq k$ } $\cup$ { 0 };
t : = max { t_1, t_2, t_3 };
Find unscheduled job i available at time t with smallest due-date ;
If $d_i \geq t + 1$ **THEN schedule job i**
x(i) : = t;
h(i) : = k (mod m);
U : = U \ {i};
t(h(i)) : = t + 1;
$\pi(k)$: = i;
k : = k + 1;
Else
Crisis(i)
If there exists an index $1 \leq \nu \leq k - 1$ with $d_{\pi(\nu)} > d_i$ THEN
BEGIN
Calculate largest index $1 \leq j \leq k - 1$ with $d_{\pi(\nu)} > d_i$;
r : = min ({ $r_{\pi(\nu)}$ / ν = j + 1, ..., k - 1 } $\cup$ { r_i });
Add (j, r) to barrier list;

Add jobs π(j), π(j+1), ..., π(k-1) to U;
k : = j
END
ELSE HALT (There exists no feasible schedule)
End

This algorithm is called barrier algorithm. If a feasible exists, then it produces a feasible schedule minimizing C_{max} as well $\sum C_i$ in at most O (n^3 log log n) time.

212. The Makespan with preemption

Problem P / pmtn / C_{max}

An elementary result for the makespan problem was presented by McNaughton[McN59] when the jobs are independent and preemption is permitted. The minimum makespan is given by :

$$C^*_{max} = \max \left\{ \frac{1}{m} \sum_{j=1}^{n} p_j, \max_j p_j \right\} \quad (1).$$

A method of constructing an optimal schedule whose length is equal C^*_{max} is given by MacNaughton[McN59], to minimize schedule length of M with m Parallel, identical Machines. Its time complexity is O (n).

Algorithm 5. *McNaughton's rule for P /pmtn / C_{max}*[McN59]

begin

$C^*_{max} = \max \left\{ \frac{1}{m} \sum_{j=1}^{n} p_j, \max_j p_j \right\}$; -- minimum schedule length

t : = 0; i : = 1; j : = 1;
repeat
if $t + p_j < C^*_{max}$
then
begin
Assign task T_j to processor P_i , starting at time t ;
$t : = t + p_j$; j : = j + 1
- task T_j can be fully assigned to processor P_i
- assignment of the next task will continue at time $t + p_j$

end
else
begin
starting at time t, assign task T_j for C^*_{max} - t units to processor P_i ;
-- task T_j is preempted at time C^*_{max},
-- processor P_i is now busy until C^*_{max},
-- assignment of T_j will continue on the next processor at time 0
$p_j : = p_j - (C^*_{max} - t)$; t : = 0; i : = I + 1;

```
    end;
until j = n;     -- all task have been scheduled
end;
```

This problem does not usually have a unique solution and the construction method produces only one of many optimal schedules. By allowing preemptions we made the problem easy to solve. The method makes no attempt to minimize the number of preemptions.

Consider the following *Linear programming* (LP) formulation of this problem :

$$\text{minimize} \quad C_{max}$$

$$\text{subject to} \quad \sum_{i=1}^{m} x_{ij} = p_j, \qquad 1 \le j \le n.$$

$$C_{max} - \sum_{i=1}^{m} x_{ij} \ge 0, \quad 1 \le j \le n.$$

$$C_{max} - \sum_{j=1}^{n} x_{ij} \ge 0, \quad 1 \le i \le m.$$

$$x_{ij} \ge 0, \quad 1 \le i \le m, \; 1 \le j \le n..$$

x_{ij} is the total time spent by job j on machine i. This LP can be solved in polynomial time, but the solution of the LP does not prescribe an actual schedule; it merely specifies the amount of time job j should spend on machine i. However, with this information a schedule can easily be constructed.

Another schedule that may appear appealing for P /pmtn / C_{max} is the *Longest remaining processing time first (LRPT)* schedule. This scheduling is the preemptive version of the (nonpreemptive) LPT schedule. An interesting result is given in [Pin95] for this rule.

The LRPT rule yields an optimal schedule for P /pmtn / C_{max} in discrete time, that is when all processing times are assumed to be integers and the decision maker is allowed to preempt any machine only at integer times 1, 2, ... The proof of this result is based on dynamic programming induction technique.

Problem P / pmtn ; r_i / C_{max}

Associated with each task T_j there is a release time r_i and a due date d_i with $r_i \le d_i$.

This problem can be solved in time O(n log n + m n)[Bru95].

22. Parallel identical processors and dependent jobs

221. Cases where the preemption is not authorized

When the jobs set is dependent, the problem of minimizing makespan may be considerably more difficult. The fundamental results for this situation are based on the work of Hu[Hu61], who presented a labeling mechanism and an associated scheduling algorithm for the makespan problem. Hu's algorithm to minimize makespan for dependent jobs with equals t_j. The jobs form an assembly tree, no job has more than one direct successor. It is denoted P / prec/ C_{max}.

The first algorithm has been given for scheduling *forests*, consisting either of *in-trees* or of *out-trees*[Hu61]. We will first present Hu's algorithm for the case of an in-tree, i.e. for **the problem P / in-tree, $p_j = 1$ / C_{max}.** The algorithm is based on the notion of a task-level in an in-tree which is defined as the number of tasks in the path to the root of the graph. The algorithm by Hu, which is also called *level algorithm* or *critical path algorithm* is as follows.

Algorithm 6. *Hu's algorithm for P / in-tree, $p_j = 1$ / C_{max}* [Hu61]

```
begin
Calculate levels of tasks ;
t : = 0 ;
repeat
    construct list L_t consisting of all the tasks without predecessors at time t;
        -- all these tasks either have no predecessors
        -- or their predecessors have been assigned in time interval [ 0, t - 1 ]
   Order L_t in nonincresaing order of task levels ;
   Assign m tasks ( if any ) to processors at time t from the beginning of list L_t ;
   Remove the assigned tasks from the graph and from the list ;
   t : = t + 1 ;
until all tasks have been scheduled ;
end ;
```

The algorithm can be implemented to run in O(n) time.

A first consisting of in-trees can be scheduled by adding task that is an immediate successor of only the roots of in-trees, and then by applying the above algorithm. A schedule of an out-tree is constructed by changing the orientation of the arcs, applying the algorithm to the obtained in-tree and then reading the schedule backwards. If jobs are not preemptable, the algorithm cannot guarantee an optimal solution, but it will usually produce very good solutions.

To deal with unequal processing times and with precedence structures more general than a tree, the labeling scheme would have to be generalized.

It is interesting to note that the problem of scheduling opposing forest (that id, combinations of in-trees and out-trees) on an arbitrary number of processors is NP-hard [Gar83]

However, if the number of processors is limited to two, the problem is easily solvable even for arbitrary precedence graphs. We present the algorithm given by Coffman and Graham[Cof72] since it can be further extended to the preemptive case. The algorithm uses labels assigned to tasks, which take into account the levels of the tasks and the number of their immediate successors. The goal is to find the shortest schedule for the problem **P2 / prec, $p_j = 1$ / C_{max}.**

Algorithm 7. *Algorithm by Coffman and Graham for P2 / prec, $p_j = 1$ / C_{max} [Cof72]*

begin

Assign label 1 to any task T_0 for which isucc(T_0) = ⊘

-- recall that isucc(T) denotes the set of all immediate successors of T

j : = 1 ;

repeat

Construct set S consisting of all unlabeled tasks whose successors are labeled ;

for all T ∈ S do

begin

Construct list L(T) consisting of labels of tasks belonging to isucc(T) ;

Order L(T) in decreasing order of the labels ;

end ;

Order these lists in increasing lexicographic order $L(T_{[1]}) <\bullet \ldots <\bullet L(T_{[|S|]})$;

Assign label j + 1 to task $T_{[1]}$;

j : = j + 1 ;

until j = n ; -- all tasks have been assigned labels

call algorithm 4 of Hu ;

-- here the above algorithm uses labels instead of levels when scheduling tasks

end ;

This algorithm its time complexity is almost $O(n^2)$.

Remark : [Blaz94]

The complexity of problem P m / prec, $p_j = 1/ C_{max}$ with a fixed number m of processors is still open despite the fact that many papers have been devoted to solving various sub-cases of precedence constraints. If tasks of unit processing times are considered, the following results are available.

Problems P 3 / opposing forest, $p_j = 1/ C_{max}$ and P k / opposing forest, $p_j = 1/ C_{max}$ are solving in time O(n) and $O(n^{2k-2} \log n)$ respectively.

If the number of available processors is variable, this problem becomes NP-hard.

Some results are also available for the sub-cases in which task processing times may take only two values. Problems P 2 / prec, p_j = 1 or 2/ C_{max} P 2 / prec, p_j = 1 or k/ C_{max} are NP-hard, while problems P 2 / tree, p_j = 1 or 2/ C_{max} and P 2 / tree, p_j = 1 or 3/ C_{max} are solvable in time O(nlogn) and $O(n^2 \log n)$respectively. Arbitrary processing times result in strong NP-hardness even for the case of chains scheduled on two processors (P 2 / chains / C_{max}).

Several papers deal with approximation algorithms for P / prec, p_j = 1/ C_{max} and more general problems. P / prec, p_j = 1/ C_{max} has been analyzed by Chen and Liu [Che75] and Kunde[Kun76].

The following bound has been proved with application of the level algorithm [Hu61].

$$R_{level} = \begin{cases} 4/3 & \text{for } m = 2 \\ 2 - 1/(m-1) & \text{for } m \geq 3 \end{cases}$$

Algorithm of Coffman and Graham [Cof72] is slightly better, its bound is R = 2 - 2/m for $m \geq 2$.

222. Cases where the job are dependent and the preemption is authorized

A special result concerns a dependent job set with preemptable jobs for arbitrary precedence structures. It is denoted **P / pmtn, prec / C_{max}.** The analysis showed that preemption can be profitable from the viewpoint of two factors. They can make problems easier to solve, and second, they can shorten the schedule. Coffman and Garey [Cof91] proved that for problem P 2 / prec / C_{max} the least schedule length achievable by nonpreemptive schedule is no more than 4/3 the least schedule length achievable when preemptions are allowed. An optimal algorithm has been developed by Muntz and Coffman[Mun69] for the case of two identical machines. Unfortunately this algorithm may not produce optimal schedules for m > 2. Muntz and Coffman[Mun70] have presented an optimizing algorithm for this problem under the condition that the precedence structure is an assembly tree.

They use the concept of *processor sharing*, in which a task receives some fraction β ($\beta \leq 1$) of the processing capacity of a processor.

Algorithm 8. *Algorithm by Muntz and Coffman for P2 / pmtn, prec / C_{max} and P / pmtn, forest/ C_{max} [Mun69, Mun70]*

begin
for all $T \in \Im$ do compute the level of task T ;
t : = 0 ; h : = m ;
repeat

construct set Z of tasks without predecessors at time t ;

while h > 0 and | Z | > 0 do

begin

Construct subsets S of Z consisting of tasks at the highest level ;

if |S| > h

then

begin

Assign β : = h / | S | of a processing capacity to each of the tasks from S ;

h : = 0 ; -- a processor shared partial schedule is constructed

end

else

begin

Assign one processor to each of the tasks from S ;

h : = h - | S | ; --- a « normal » partial schedule is constructed

end ;

Z : = Z - S ;

end ; -- the most « urgent » tasks have been assigned at time t

Calculate time τ at which either one of the assigned tasks is finished or a point is reached at which continuing with the present partial assignment means that a task at a lower level will be executed at a faster rate β than a task at a higher level ;

Decrease levels of the assigned tasks by $(\tau - t)\ \beta$;

t : = τ ; h : = m ;

-- a portion of each assigned task equal to $(\tau - t)\ \beta$ has been processed

until all tasks are finished ;

call algorithm 5 (Algorithm of Coffman and Graham) to re-schedule portions of the processor shared schedule to get a normal one ;

end ;

The above algorithm can be implemented to run in $O(n^2)$ time.

An another class of the precedence graphs for which the scheduling problem can be solved in polynomial time is considered. It is the case of precedence constraints in the form of an activity network (task-on-arc precedence graph) whose nodes (events) are ordered in such a way that the occurrence of node i is not later than the occurrence of node j, if i < j. Now, let S_I denote the set of all the tasks which may be performed between the occurrence of event (node) I and I + 1. Such sets will be called *main sets*. Let us consider processor *feasible sets*, i.e. those main sets and those subsets of the main sets whose sizes are not greater than m, and number these sets from 1 to some K. Now let Qj denote the set of indices of processor feasible sets in which tasks T_j may be performed, and let x_i denote the duration of the ith feasible set. Then , a linear programming problem can be formulated in the following way [Bla94].

$$\text{Minimize } C_{max} = \sum_{i=1}^{K} x_i$$

$$\text{subject to } \sum_{i \in Q_j}^{K} x_i = p_j \ , \ j = 1,2,..., n$$

$$x_i \geq 0, \qquad i = 1,..., K.$$

The solution of the LP problem depends on the order of nodes in the activity network, hence an optimal solution is found when this order is unique. Such a situation takes place for a *Uniconnected activity network(uan)*, i.e. one in which any nodes are connected by a directed path in only one direction. We may use any polynomial algorithm which solves any LP problem (i.e. algorithm of Khachian or Karmakar). Hence P m / pmtn, uan / C_{max} is solved in polynomial time.

For general precedence graphs, the problem is NP-hard[Ull76]. The worst case behavior of algorithm 6 (Muntz and Coffman) applied in the case of P / pmtn, prec / C_{max} has been analyzed by Lam and Sethi[Lam77]. $R_{alg\ Muntz\&Coffman} = 2 - 2/m$, $m \geq 2$.

23. Uniform and Unrelated Processors

Using Blazewicz et all [Bla94] formulation of scheduling problems, ***Problem Q / p_j = 1 / C_{max}*** means that the processors are uniform.

Since the problem with arbitrary processing times is already NP-hard for identical processors, all we can hope to find is a polynomial time optimization algorithm for tasks with unit standard processing times only. Such an approach has been given by Graham et al [Gra79] where a transportation network formulation has been presented for problem Q / p_j = 1 / C_{max} .

Let there be n sources j, j = 1,2,...,n and m n sinks (i, k), i = 1,...,m and k = 1,...,n. sources corresponds to tasks and sinks to processors and positions of tasks on them. If b_i denotes the *speed* of processor i, let $c_{ijk} = k / b_i$ be the cost of arc (j, (i,k)), this value corresponds to the completion time of task T_j processed on P_i in the k th position. The arc flow x_{ijk} has the following interpretation : $x_{ijk} = \begin{cases} 1 & \text{if } T_j \textit{ is} \text{ processed in the kth position on } P_i \\ 0 & \text{otherwise} \end{cases}$

The min-max transportation problem can be formulated by :

$$\text{Minimize} \quad \max_{i,j,k} \{ c_{ijk} x_{ijk} \}$$

$$\text{subject to} \sum_{i=1}^{m} \sum_{k=1}^{n} x_{ijk} = 1 \quad \text{for all j,}$$

$$\sum_{j=1}^{n} x_{ijk} \leq 1 \quad \text{for all i, k ,}$$

$$x_{ijk} \geq 0 \quad \text{for all i, j, k.}$$

This problem can be solved by a standard transportation procedure which results in O (n^3) time complexity. The minimum schedule length is given as $C^*_{max} = \sup \{ t / \sum_{i=1}^{m} \lfloor t b_i \rfloor < n \}$. A lower bound for the above problem is $C' = n / \sum_{i=1}^{m} b_i \leq C^*_{max}$.

Bound C' can be achieved e.g. by a preemptive schedule.

Other problems of non preemptive scheduling of independent tasks are NP-hard, one may be interested by applying heuristics. One heuristic algorithm which is a list scheduling algorithm, has been presented by Liu and Liu [Liu74]. Tasks are ordered on the list in non increasing order of their processing times and processors are ordered in non increasing order of their speeds. Whenever a machine becomes free, it gets the first non assigned task of the list ; if there are two or more free processors, the fastest is chosen. The worst case behavior of the algorithm for the m + 1 processor system, m of which have processing speed factor equal to one and the remaining processor has processing speed factor b is given by the ratio

$$R = \begin{cases} \dfrac{2(m + b)}{b + 2} & \text{for } b \leq 2 \\ \dfrac{m + b}{2} & \text{for } b > 2 \end{cases} .$$

Problem Q / pmtn / C_{max}

We consider n jobs to be processed on m parallel uniform machines. The machines have different speeds s_j (j = 1, .., m). Each job J_i has a processing requirement p_i (i = 1, .., n). Execution of job J_i on machine M_j requires p_i / s_j time units. If we set $s_j = 1$ for j = 1,..., m, we have m parallel identical machines. The NP-hard problems with identical machines keep NP-hard with uniform machines. Therefore, we first consider problems with preemptions. By allowing preemptions, one can find optimal schedules in polynomial time. We will present an algorithm given by Horvath et al[Hor77] despite that the fact that there is a more efficient one by Gonzalez and Sahni[Gon78]. The first algorithm covers more general precedence constraints than the second. The algorithm is based on two concepts : the task *level*, defined as processing requirements of the unexpected portion of the task, expressed in terms of a standard processing time, and *processor sharing*, the possibility of assigning only a fraction β($0 \leq \beta \leq \max \{ b_i \}$) of processing capacity to some task. Let us assume that tasks are indexed in order of non-increasing

p_j 's and processor are in order of non-increasing values of b_i . The minimum schedule length

can be estimated by $C^*_{max} \geq C = \max \left\{ \max_{1 \leq k \leq m} \left\{ \frac{X_k}{B_k} \right\}, \left\{ \frac{X_n}{B_m} \right\} \right\}$, where X_k is the sum of processing requirements (i.e. standard processing times p_j) of the first k tasks, and B_k is the collective processing capacity (i.e. the sum of processing speed factors b_i) of the first k processors. The algorithm presented below constructs a schedule length equal to C for the problem Q / pmtn / C_{max}.

Algorithm 9. *Algorithm by Horvah, Lam and Sethi for Q/pmtn.C_{max}[Hor77]*

begin
for all T ∈ ℑ do compute level of task T ;
t : = 0 ; h : = m ;
repeat
  while h > 0 do
    begin
    Construct subset S of ℑ consisting of tasks at highest level ;
      -- the most « urgent » tasks are chosen
    if | S | > h
    then
      begin
      Assign the tasks of set S to the h remaining processors to be processed at
        the same rate $\beta = \sum_{i=m-h+1}^{m} b_i / |S|$;
      h : = 0 ; -- tasks from set S share the h lowest processors
      end

```
        else
            begin
            Assign tasks from set S to be processed at the same rate β on the fastest
               |S| processors ;
            h : = h - |S| ;          -- tasks from set S share the fastest |S| processors

            end ;
      end ;      -- the most urgent tasks have been assigned at time t
   Calculate time moment τ at which either one of the assigned tasks is finished or a
      point is reached at which continuing with the present partial assignment causes
      that a task at a lower level will be executed at a faster rate β than a higher level task ;
          -- note, that the levels of the assigned tasks decrease during task execution
    Decrease levels of the assigned tasks by ( τ - t) β ;
    t : =  τ ;  h : = m ;   -- a portion of each assigned task equal to ( τ - t) β has been processed
until  all tasks are finished ;
    -- the schedule constructed so far consists of a sequence of intervals during each
    -- of  which certain tasks are assigned to the processors in a shared mode.
    -- In the next loop task assignment in each of these intervals is determined
for each interval of the processor shared schedule  do
   begin
   let y be the length of the interval ;
  if g tasks share g processors
  then Assign each task to each processor for y/g time units
  else
      begin
      Let p be the processing requirement of each of the g tasks in the interval ;
      Let b be the processing speed factor of the slowest processor ;
      if p/b < y
      then call algorithm 4  of Coffman and Graham[Cof72]
       -- tasks can be assigned as in McNaughton's rule, ignoring different processor speeds
       else
      begin
      Divide the intervals into g subintervals of equal lengths ;
      Assign the g tasks so that each task occurs in exactly h intervals, each
            time on a different processor ;
      end ;
    end ;
 end ;
  -- a normal preemptive schedule has now been constructed
end ;
```

The time complexity of this algorithm is $O(m\ n^2)$.

A similar algorithm called algorithm level, is given by Brucker [Bru95].

In [Pin95], the processor sharing is called LRPT-FM rule. It is the *Longest remaining processing time on the fastest machine first* rule. This rule assigns at any time the job with the largest remaining processing time among the remaining jobs to the fastest machine ; the job

with the second largest remaining processing time to the second fastest machine, and so on. It usually requires an infinite number of preemptions.

LRPT-FM rule yields an optimal schedule for Q / pmtn / C_{max} in discrete time[Pin95].

Problem Q/ pmtn, prec/C_{max}

When considering dependent tasks, only preemptive polynomial time optimization algorithms are known. Algorithm of Horvath et al[Hor77] also solves problem **Q 2 / pmtn, prec/C_{max}** if the level of a task is understood as in algorithm of Muntz and Coffman[Mun69, Mun70] where standard processing times for all the tasks sere assumed. When considering this problem one should also take into account the possibility of solving it for uni-connected activity networks via the slightly modified linear programming approach. It is also possible to solve the problem by using another LP formulation which is described below.

It is also possible to solve to solve problem Q/ pmtn, prec, C_{max} approximately by the two machine aggregation approach, developed in the framework of flow shop scheduling. In this case the two fastest processors are used only, and the worst case bound is :

$$\frac{C_{\max}}{C^*_{\max}} \leq \begin{cases} \sum_{i=1}^{m/2} \max\left(b_{2i-1}/b_1, b_{2i}/b_2 \right) & \text{if m is even} \\ \sum_{i=1}^{\lfloor m/2 \rfloor} \max\left(b_{2i-1}/b_1, b_{2i}/b_2 \right) + b_m/b_1 & \text{if m is odd} \end{cases}$$

Problem R / pmtn / C_{max}

This case of unrelated processors is the most difficult. The unit-length tasks would reduce to the case of identical or uniform processors. No polynomial time optimization algorithms are known for the problems other than preemptive ones. Very little is known about approximation algorithms for this case. Some results have been obtained by Ibarra and Kim[Iba77], but the obtained bounds are not very encouraging. Thus, we will pass to the preemptive scheduling model. Problem R / pmtn / C_{max} can be solved by a *two-phase method* . The first phase consists in solving a linear programming problem formulated independently by Blazewicz et al[Bla76a, Bla77] and by Lawler and Labetoulle [Law78]. The second phase uses the solution of this LP problem and produces an optimal preemptive schedule.

Let $x_{ij} \in [\ 0, 1\ ]$ denote a part of T_j processed on P_i . The LP formulation is as follows :

Minimize C_{max}

$$\text{subject to} \qquad C_{max} - \sum_{j=1}^{n} p_{ij}\, x_{ij} \geq 0, \quad 1 \leq i \leq m.$$

$$C_{max} - \sum_{i=1}^{m} p_{ij}\, x_{ij} \geq 0, \quad 1 \leq j \leq n.$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad 1 \leq j \leq n.$$

Solving the above problem, we get $C_{max} = C^*_{max}$ and optimal values x^*_{ij} . However, we do not know how to schedule the task parts, i.e. how to assign these parts to processors in time. A schedule may be constructed in the way given in [Bla94].

The procedure is summarized in algorithm10.

Algorithm 10. *Construction of an optimal schedule corresponding to LP solution for R / pmtn / C_{max} .*

begin
C_{max} : = C^*_{max} ;
while C > 0 do
  begin
  Construct set U
    -- thus a subset of tasks to be processed in a partial schedule has bee chosen
  ν_{min} : = $\min_{\nu_{ij} \in U} \{\nu_{ij}\}$;
  ν_{max} : = $\min_{j \in \{j' / \nu_{ij} \notin U \text{ for } i=1,\ldots,m\}} \left\{\sum_i \nu_{ij}\right\}$
  if C - $\nu_{min} \geq \nu_{max}$
  then δ : = ν_{min}
  else δ : = C - ν_{max} ;
    -- the length of the partial schedule is equal to δ
  C : = C - δ ;
  for each $\nu_{ij} \in U$ do ν_{ij} : = ν_{ij} - δ;
    -- matrix V is changed ;
    -- note that due to the way δ is defined, the elements of V can never become negative
  end ;
end ;

We need an algorithm that finds set U for a given matrix V. One of the possible algorithms is based on the network flow approach. The network has m nodes corresponding to machines(row of V) and n + M nodes corresponding to tasks (columns of V) A node i from the first group is connected by an arc to a node j of the second group if and only if $\nu_{ij} > 0$. Arcs flows are constrained by b from below and by c = 1 from above, where the value of b is

1 for arcs joining the source with processor-nodes and critical task nodes with the sink, and b = 0 for the other arcs. Finding a feasible flow in this network is equivalent to finding set U.

The complexity of the above approach is bounded from above by a polynomial in the input length. The transformation to the LP problem is polynomial and the LP problem may be solved in polynomial time using Khachiyan's algorithm. The loop in the algorithm 8 is repeated at most O(mn) times and solving the network flow problem requires $O(z^3)$ time, where z is the number of network nodes[Kar74].

Problem R / pmtn, prec / C_{max}

If dependent tasks are considered, linear programming problems similar to those discussed above and on the activity network presentation, can be formulated. In the later formulation one defines x_{ijk} as a part of task T_j processed on processor P_i in the main set S_k . Solving the LP problem for x_{ijk} , one then applies algorithm 8 for each main set. If the activity network is uni-connected, an optimal schedule is constructed in this way, otherwise only an approximate schedule is obtained.

24. Complexity Results

We summarize scheduling problems complexity with jobs to be processed on parallel machines.

Problems polynomially Solvable	**Complexity**	**Author(s) and date**
P / pmtn / C_{max}	O(n)	McNaughton (1959)
P / $p_i = 1$; r_i ; d_i rational / C_{max}	$O(n^3 \log \log n)$	Simons (1983)
Q / pmtn / ; r_i / C_{max}	O(n logn + m n)	Labetoulle et al (1984)
P / tree ; $p_i = 1$ / C_{max}	O(n)	Hu (1961)
P 2 / tree ; pmtn / C_{max}	O(nlogm)	Gonzalez & Johnson(1983)

Problems NP-hard	**Author(s)**
P 2 // C_{max}	Karp (1972)
P/ prec ; $p_i = 1$ / C_{max}	Ullmann(1975)

P/ prec ; pmtn ; $p_i = 1 / C_{max}$	Ullmann(1976)

3. Minimizing Mean Flowtime And Weignted Mean Flowtime.

Consider the problem of minimizing $\overline{F}$. Adopt the following notation :

$p_{i[j]}$: processing time of the j th job in sequence on the i th machine

$F_{i[j]}$: flowtime of the j th job in sequence on the i th machine

n_i : number of jobs processed by the i th machine, then the objective function is :

$$\overline{F} = \frac{1}{n} \sum_{i=1}^{m} \sum_{j=1}^{n_i} F_{i[j]} = \frac{1}{n} \sum_{i=1}^{m} \sum_{j=1}^{n_i} (n_i - j + 1)\, p_{i[j]} .$$

An algorithm 5.3 is given for Minimizing $\overline{F}$ with parallel machines[Bak74].

Except for ties, this algorithm will produce a unique schedule.

*) This algorithm is a dispatching procedure, so that scheduling decisions can be implemented in the order that they are made. It can be extended to the case of arrival pattern of jobs.

*) No direct algorithm has been developed for constructing an optimal schedule when $\overline{F}_w$ is the criterion.

Dynamic programming formulations are possible, but the « curse of dimensionality » renders the procedure impractical for problems of even moderate size. Any optimal solution must have WSPT job orderings at each machine. A lower bound on the optimum of $\overline{F}_w$ can be computed, as shown by Eastman, Even and Isaacs[Eas64]. Let

B(1) : the minimal value of $\overline{F}_w$ for the given job set if there were only one machine (obtained via WSPT)

B(n) : the minimal value of $\overline{F}_w$ for the given job set if there were n machines (obtained by assigning each job to a different machine) then a lower bound for m machines ($1 \leq m \leq n$) is

$$B(m) = \frac{1}{2m} [\, (m - 1)\, B(1) + 2\, B(n) \,] ;$$

B = Max {B(m), B(1)} is also a valid lower bound and may be better.

A procedure denoted H_m is given.

In the case where the machines are identical, the second subscript can be omitted from p_{ij} and

$p_{ij} = p_i$. One could renumber the n jobs in order of increasing processing-time :

$p_1 \quad p_2 \quad \ldots \quad p_n$ .Then one could number the machines and simply assign jobs in rotation :

Job	1 2 ... m	M+1 m + 2 ... 2m	2m + 1 .....
Machine	1 2 ... m	1 2 ... m	1

The rule would simply be that each time a machine finished a job, it would be assigned, from among those jobs waiting, the job with the shortest processing time. This rule minimizes the mean flow time, mean waiting time, and mean lateness.

Interchange of jobs in equivalent positions in sequence on different machines has no effect on the mean flow time. However, this shortest processing time rule does not necessarily minimize the maximum flow time and the mean number of jobs in the shop[Con67].

3.1 . Identical Processors[Bla94]

Problem P // $\sum C_j$

In the case of identical processor and equal ready times, preemptions are not profitable from the viewpoint of the value of the mean flow time [McN59]. Thus we consider only non-preemptive schedules. When analyzing the nature of criterion $\sum C_j$, one might expect that, as in the case of one processor, by assigning tasks in non-decreasing order of their processing times the mean flow time will be minimizing. A proper generalization of this simple rule leads to an optimization algorithm for P // $\sum C_j$ [Con67]. It is as follows.

Algorithm 11. *SPT rule for problem P // $\sum C_j$ [Con67].*

begin
Order tasks on list L in non-decreasing order of their processing times ;
while L $\neq \oslash$ do
  begin
  Take the m first tasks from the list (if any) and assign theses tasks arbitrarily to the
    m different processors ;
  Remove the assigned tasks from list L ;
  end ;
Process tasks assigned to each processor in SPT order ;
end ;

The complexity of the algorithm is O(nlogn).

Introducing different ready times makes the problem strongly NP-hard even for the case of one processor[Len77]. Also if we introduce different weights, the 2-processor problem without release times, P2 // $\sum w_j C_j$ **,** is already NP-hard[Bru74].

Problem P / $p_i = 1$; r_i ; d_i / $\sum C_j$ [Bru95]

The same results and algorithm of the Problem P / $p_i = 1$; r_i; d_i / C_{max} hold when the objective function is $\sum C_j$ [see part 2, makespan criterion].

Problems P / $p_i = 1$ / $\sum w_i C_j$ and P / $p_i = 1$; pmtn / $\sum w_i C_j$ [Bru95]

We get an optimal schedule for problem P / $p_i = 1$ / $\sum w_i C_i$ by scheduling the jobs in order of non-increasing weights w_i. An optimal schedule for P / $p_i = 1$ / $\sum w_i C_i$ is optimal even if preemption is allowed.

For P / pmtn / $\sum w_i C_i$, there exists an optimal schedule without preemption.

The following problems are NP-hard :

P2 / / $\sum w_i C_i$; P2 / pmtn, r_i / $\sum w_i C_i$ and P2 / pmtn / $\sum w_i C_i$

Problem P / prec / $\sum C_j$

Let us pass to the case of dependent tasks. Here, P / out-tree, $p_j = 1$ / $\sum C_j$ is solved by an adaptation of Hu's algorithm to the out-tree case, and P / prec, $p_j = 1$ / $\sum C_j$ is strongly NP-hard

[Len78]. In the case of arbitrary processing times recent results by Du et al [Du91] indicate that even simplest precedence constraints result in computational hardness of the problem.

That is problem P2 / chains / $\sum C_j$ is already NP-hard in the strong sense. It was also proved in [Du91] that preemption cannot reduce the mean weighted flow time for a set of chains. This implies that the problem P2 / chains, pmtn / $\sum C_j$ is also NP-hard in the strong sense. No approximation algorithms for these problems are evaluated from their worst-case behavior.

Problem P / $p_j = 1$; out-tree / $\sum C_j$ [Pin95]

The special case of precedence constraints of out-tree and all processing times equal to one can be solved in polynomial time. The *CP* (critical path) *rule* is the *highest level first* rule .

It is optimal for P / $p_i = 1$; out-tree / $\sum C_j$. The CP rule is, not necessarily optimal for in-trees.

Problem P / $p_j = 1$; M_j / $\sum C_j$ [Pin95]

Another problem that is of practical interest arises when job j is only allowed to be processed on subset M_j of the m machines in parallel is $P / p_i = 1 ; M_j / \sum C_j$ problem. Assume that the set M_j is *nested* (see 211.).

Under these conditions, the *least flexible job first* (LFJ) rule is of importance. The LFJ rule selects, every time a machine is freed, the job that can be processed on the smallest number of machines, that is the least flexible job. Ties may be broken arbitrary.

If the set M_j sets are nested, the LFJ rule is optimal.

The LFJ rule is optimal for $P / p_i = 1 ; M_j / \sum C_j$ when the M_j sets are nested [Pin95].

3.2. Uniform and Unrelated Processors

Scheduling dependent tasks on uniform or unrelated processors is NP-hard problem in general. No approximations algorithms have been investigated either. On the other hand, in the case of independent tasks, preemption may be worth-while.

Problem Q // $\sum C_j$

Let us start with uniform processors and non-preemptive schedules. In this case the flow time has to take into account processor speed ; so the flow time of task $T_{i[k]}$ processed in the k th position on processor P_i is defined as $F_{i[k]} = \frac{1}{b_i} \sum_{j=1}^{k_i} p_i[j]$.

Let us denote by n_i the number of tasks processed on processor P_i . Thus, $n = \sum_{i=1}^{m_i} n_i$.

The mean flow time is then given by $\overline{F} = \frac{1}{n} \sum_{i=1}^{m} \frac{1}{b_i} \sum_{k=1}^{n_i} (n_i - k + 1)\, p_i[k]$.

The numerator of this formula is the sums of n terms each of which is the product of a processing time and one of the following coefficients :

$$\frac{1}{b_1} n_1, \frac{1}{b_1}(n_1 - 1), \ldots, \frac{1}{b_1}, \frac{1}{b_2} n_2, \ldots, \frac{1}{b_2}(n_2 - 1), \ldots, \frac{1}{b_2}, \ldots, \frac{1}{b_m} n_m, \frac{1}{b_m}(n_m - 1), \ldots, \frac{1}{b_m}.$$

It is shown in [Con67] that such a sum is minimized by matching n smallest coefficients in non-decreasing order with processing times in non-increasing order. This rule is an O(nlogn) implementation given by Horowitz and Sahni[Hor76]

Assume that $i_1, i_2, \ldots, i_r$ is the sequence of jobs to be processed on machine M_j. Then the contribution of these jobs on machine M_j in the objective function is given by :

$p_{i_1} \frac{r}{s_j} + p_{i_2} \frac{r-1}{s_j} + \ldots + p_{i_r} \frac{1}{s_j}$

This implies that in an optimal schedule the jobs on machine M_j are sequenced according to non-decreasing processing requirements p_i.

To find an optimal distribution of jobs to machines we may proceed as follows. Let

t_1 , t_2 , ..., t_n be a non-decreasing sequence of the n smallest numbers in the set

$$\{ \frac{1}{s_1}, \frac{1}{s_2}, \ldots, \frac{1}{s_m}, \frac{2}{s_1}, \frac{2}{s_2}, \ldots, \frac{2}{s_m}, \frac{3}{s_1}, \frac{3}{s_2}, \ldots, \frac{3}{s_m}, \ldots \}.$$

If $t_i = \frac{k}{s_j}$, then schedule job i on machine M_j as the k-th last job because we assume

$p_1 \geq p_2 \geq \ldots \geq p_n$. These ideas lead to the following algorithm in which, for j = 1,..., m, we denote by Π_j the sequence of jobs currently scheduled at machine M_j.

Algorithm 12 . *Problem Q // $\sum C_j$*

For j = 1 to m do

Begin $\Pi_j := \oslash$; $w_j = \frac{1}{s_j}$; **END**;

For i : = 1 to n do

Begin

Find the largest index j with $w_j = \min_{\upsilon=1,..,m} \frac{1}{s_\upsilon}$

$\Pi_j := i \circ \Pi_j$;

$w_j := w_j + \frac{1}{s_j}$

END

Each w-value can be calculated in O(logm) time if we use a priority queue. Thus, the overall complexity is O(nlogm) if the p_i -values are sorted. Otherwise, we have a total complexity of O(nlogn) if n > m.

Next we will consider the preemptive version of this problem.

Problem Q / pmtn / $\sum C_j$

In the case of preemptive scheduling, it is possible to show that there exists an optimal schedule for Q / pmtn / $\sum C_j$ in which $C_j \leq C_k$ if $p_j < p_k$. The following algorithm has been proposed by Gonzales[Gon77].

Algorithm 13. *Algorithm by Gonzalez for Q / pmtn / $\sum C_j$ [Gon77].*

Begin

Order processors in non-increasing order of their processing speed factors ;

Order tasks in non-decreasing order of their standard processing times ;
for j = 1 to n do
    begin
    Schedule task T_j to be completed as early as possible, preemptive when necessary ;
        -- tasks will create a staircase pattern « jumping » to a faster processor
        -- whenever a shorter task has been finished
    end ;
end ;

The complexity of this algorithm is O(n logn + m n).

In [Pin95], the SRPT-FM rule is considered. It is the S*hortest remaining processing time on the fastest machine first* rule. This rule assigns at any time the job with the shortest remaining processing time among the remaining jobs to the fastest machine ; the job with the second shortest remaining processing time to the second fastest machine, and so on. It usually requires a number of preemptions equal to the number of remaining jobs. There exists an optimal schedule for

Q / pmtn / $\sum C_j$ in which $C_j \leq C_k$ if $p_j < p_k$.

SRPT-FM rule yields an optimal schedule for Q / pmtn / $\sum C_j$ [Pin95].

Problem R / / $\sum C_j$

An approach to the solution of this problem is based on the observation that task

$T_j \in \{ T_1, .., T_n \}$ processed on Processor $P_i \in \{ P_1, .., P_m \}$ as the last task contributes its processing time p_{ij} to $\overline{F}$. The same task processed in the last but one position contributes 2 p_{ij} and so on[Bru74] This reasoning allows one to construct an (m n) x n matrix Q presenting contributions of particular tasks processed in different positions on different processors to the value of $\overline{F}$. The problem is to choose n elements from matrix Q such that

-- Exactly one element is taken from each column,

-- at most one element is taken from each row,

-- the sum of selects elements is minimum.

It is a variant of the assignment problem which may be solved using Branch and Bound method. It can be solved in $O(n^3)$ time [Bru74].

The complexity of problem R / pmtn / $\sum C_j$ is still an open problem.

3.3. Complexity Results

We summarize scheduling problems complexity with jobs to be processed on parallel machines and flow time and mean flow time criteria. .

Problems polynomially Solvable	**Complexity**	**Author(s) and date**
P / $p_i = 1$; r_i; d_i rational / $\sum C_j$	$O(n^3 \log \log n)$	Simons (1983)
P / $p_i = 1$/ $\sum w_i C_i$	O(nlogn)	McNaughton(1959)
P / $p_i = 1$/ pmtn / $\sum w_i C_i$	polynomial	//
Q / pmtn / $\sum C_j$	O(n logn + m n)	Gonzalez (1977) .
R / / $\sum C_j$	$O(n^3)$	Bruno et al (1974)

Problems NP-hard	**Author(s)**
P2 / / $\sum w_i C_i$	Bruno et al (1974)
P2 / pmtn ; r_i / $\sum C_i$	Due et al (1988)
P2 / pmtn / $\sum w_i C_i$	Bruno et al (1974)
P2 / prec ; p_i / $\sum C_i$	Lenstra & Rinnooy Kan (1978)

4. Minimizing Due Date Involving Criteria

41. Identical processors (Bla94)

Single processor problems with due dates that allow for simple algorithms usually have as objective the maximum lateness ; for example, 1 // L_{max} ; 1 / prmp / L_{max} ; 1 / r_j , prmp / L_{max}. However, single machine problems with the sum of the tardinesses or the sum of the weighted tardiness as objectives are NP-hard in most cases. P m // L_{max} is not as easy as 1 // L_{max}.

Consider the special case where all jobs have due date 0. Finding a schedule with a minimum L_{max} is equivalent to the P m // C_{max} problem and is a problem NP-hard [Pin95].

We concentrate on minimization of L_{max} criterion. In this case, the general rule should be to schedule tasks according to their earliest due dates. This simple rule of Jackson [Jac55]

produces optimal schedules under very restricted assumptions only. In other cases more sophisticated algorithms are necessary, or the problems are NP-hard.

Problem P // L_{max}

We consider nonpreemptive scheduling of independent tasks. Taking into account simple transformations between scheduling problems and the relation-ship between the C_{max} and L_{max} criteria, all the problems that are NP-hard under the C_{max} criterion remain NP-hard under L_{max} criterion. Hence, for example, P 2 // L_{max} is NP-hard. On the other hand, unit processing times of tasks made the problem easy, and P / p_j = 1, r_j / L_{max} can be solved in polynomial time by an application of EDD rule [Bla77] It can be solved by an extension of the single processor algorithm[Gar81]. Unfortunately very little is known about the worst-case behavior of approximation algorithms for the NP-hard problems in question.

Problem P / pmtn / L_{max}

The preemptive mode of processing makes the solution of the scheduling problem much easier.

The fundamental approach in that area is testing feasibility of problem P / pmtn, r_j , $\tilde{d}_j$ / - via the network flow approach [Hor74]. Using this approach repetitively, one can then solve the original problem P / pmtn / L_{max} by changing due dates (deadlines) according to a binary search procedure. Finding a feasible flow pattern corresponds to constructing a feasible schedule and this test can be made in $O(n^3)$ time.

A schedule is constructed on the basis of flow values on arcs between interval and task nodes. In the next step a binary search can be conducted on the optimal value of L_{max} , with each trial value of L_{max} including deadlines which are checked for feasibility by means of the above network flow computation. This procedure can be implemented to solve problem

P / pmtn, r_j /L_{max} in $O(n^3 \min \{ n^2, \log n + \log \max \{p_j \}\}$ time [Lab84]

Problem P / pmtn ; r_i / L_{max}

This problem may be reduced to a maximum flow problem in a network[Bru95]. There exists a schedule respecting all time windows if and only if the maximum flow has the value $\sum_{i=1}^{n} p_i$.

If this is the case, the flow x_{iK} on the arc (J_i , J_K)corresponds with the time period in which job J_i is processed in the time interval I_K and we have $\sum_{K=1}^{r-1} x_{iK} = p_i$ for i = 1,..,n and $\sum_{i=1}^{n} x_{iK} \leq m\ T_K$ for K = 1,..., r - 1.

To solve P / pmtn; r_i / L_{max} , we apply binary search on different-values.

We assume $d_i \leq n \max_{j=1,...n} p_j$, which implies $-n \max_{j=1,...n} p_j \leq L_{max} \leq n \max_{j=1,...n} p_j$.

This yields an algorithm which approximates the solution value with absolute error ε in at most $O(n^3 (\log n + \log(1/\varepsilon) + \log (\max_{j=1,...n} p_j))$ steps.

Problem P / p_i = 1 ; r_i / L_{max}

It is convenient to assume that the jobs are indexed in such a way that $r_1 \leq r_2 \leq ... \leq r_n$.

The problem has an easy solution if all release times r_i are integer. In this case, we get an optimal schedule by scheduling available jobs in order of non-decreasing due date. More specifically, if at the current time t not all machines are occupied and there is an unscheduled job Ji with $r_i \leq t$, we schedule such a job with smallest due date.

Technical details are given by the following algorithm in which K counts the number of machines occupied immediately after Step 7 at current time t, m is the number of machines, M is the set of unscheduled jobs available at time t, and j is a counter for the number of scheduled jobs.

Algorithm 14. *Algorithm for P / p_i = 1; r_i integer / L_{max}[Bru95]*

```
1. j : = 1 ;  K : = 1 ;
2. WHILE  j  ≤  n   Do
       BEGIN
3.     t : = r_j ; M : = { J_i / r_i ≤ t } ;
4.     WHILE  M ≠ ∅  Do
                BEGIN
5.              Find job J_i in M with the smallest due date ;
6.             M : = M \ { J_i } ;
7.             Schedule J_i at time t ;
8.             j : = j + 1;
9.            IF  K + 1 ≤ m  THEN   K : = K + 1 ;
              ELSE
                    BEGIN
10.                   t : = t + 1 ;
11.                  K : = 1 ;
12.                  M : = M ∪ { J_i / r_i ≤ t } ;
```

```
            END;
        END ;
    END ;
```

The algorithm runs in O (n log n) time if a priority queue data structure is used for M.

Problem P / prec, p_j = 1 / L_{max}

Let us now pass to dependent tasks. A general approach in this case consists in assigning modified due dates to tasks, depending on the number and due dates of their successors. The way in which modified due dates are calculated depends on the parameters of the problem. When scheduling nonpreemptable tasks on a multiple processor system only unit processing times can results in polynomial time scheduling algorithms. Let us star with in-tree precedence constraints and assume that if $T_i < T_j$ then i > j. The following algorithm minimizes L_{max} where isucc(j) denotes the immediate successor of T_j [Bru76b].

Algorithm 15. *Algorithm by Brucker for P / in-tree, $p_j = 1 / L_{max}$ [Bru76b]*

begin
$d^*_1 = 1 - d_1$;
for k = 2 step 1 until n do
    begin
    Calculate modified due dates of T_k according to the formula
        $d^*_k := \max \{ 1 + d^*_{isucc(k)}, 1 - d_k \}$;
    end ;
Schedule tasks in nonincreasing order of their modified due dates subject to precedence
    constraints ;
end ;

This algorithm can be implemented to run in O(nlogn) time. Surprisingly out-tree precedence constraints result in the NP-hardness of the problem [Bru77]

However, when we limit ourselves to two processors, a different way of computing modified due dates can be proposed which allows one to solve the problem in O(n^2) time [Gar76].

In the algorithm below g(k, d^*_i) is the number of successors of T_k having modified due dates not greater than d^*_i.

Algorithm 16. *Algorithm by Garey and Johnson for problem P2/ prec, $p_j = 1/ L_{max}$[Gar76]*

begin
Z : = T
while Z $\neq \oslash$ do
  begin

Choose $T_k \in_k Z$ which is not yet assigned a modified due date and all of whose successors have been assigned modified due dates ;
Calculate a modified due date of T_k as :
$d^*_k := \min \{ d_k , \min \{ (d^*_i - |\frac{1}{2} g(k,d^*_i)|) / T_i \in succ(T) \}\}$;
$Z := Z - \{ T_k \}$;
end ;
Schedule tasks in nondecreasing order of their modified due dates subject to precedence constraints ; r
end ;

For m > 2 this algorithm may not lead to optimal schedules. However, the algorithm can be generalized to cover the case of different ready times too, but the running time is then $O(n^3)$ [Gar77] and this is as much as we can get in non-preemptive scheduling.

Problem P / pmtn, prec / L_{max}

Preemptions allow one to solve problems with arbitrary processing times. In Law82b] algorithms have been presented that are preemptive counter-parts of Algorithms 15 (by Brucker[Bru76b]) and 16 (by Garey and Johnson[Gar76]) and the one presented by Garey and Johnson[Gar77] for non-preemptive scheduling and unit-length tasks.

Hence problems P / pmtn, in-tree / L_{max} , P 2 / pmtn, prec / L_{max} and P2/ pmtn, prec, r_j / L_{max} are solvable in polynomial time. Algorithms for these problems employ essentially the same techniques for dealing with precedence constraints as the corresponding algorithms for unit-length tasks. The algorithms are more complex.

4.2. Uniform and Unrelated Processors

Problem Q // L_{max}

We see that non-preemptive scheduling to minimize L_{max} is in general a hard problem. Only for the problem Q / p_j = 1/ L_{max} a polynomial time optimization algorithm is known. This problem can be solved via a transportation problem formulation , where now C_{ijk} = k / b_i - d_j .

Problem Q / pmtn / L_{max}

This problem is one of the few parallel machine problems with a due-date related objective that can be solved in polynomial time.

Finding a feasible schedule with all jobs completing their processing before these deadlines is equivalent to solving the problem Qm / r_j, prmp / C_{max}. Reverse the direction of time in the

due-date problem. The deadlines in the original problem play the role of release dates in the reverse problem, which is then equivalent to Qm / r_j, prmp / C_{max}. f applying the LRPT-FM rule backwards results in a feasible schedule with all the jobs in the original problem starting at a time larger than or equal to zero, then there exists a schedule for Qm / prmp / L_{max} with $L_{max} < z$. To find the minimum L_{max} a simple search has to be done to determine the appropriate minimum of value of z. This problem can be formulated as a network flow problem that can be solved in polynomial time.

We will consider uniform processors first. One of the most interesting algorithms in that area has been presented for problem Q / pmtn, r_j / L_{max} by Federgruen and Groenevelt[Fed86].

It is a generalization of the network flow approach to the feasibility testing of problem P / pmtn, r_j , $\tilde{d}_j$ / -described above. The feasibility testing procedure for problem Q/ pmtn, r_j , $\tilde{d}_j$ / - uses tripartite network formulation of the scheduling problem, where the first set of nodes corresponds to tasks, the second corresponds to processors-interval (period) combination and the third corresponds to interval nodes.

Finding a feasible flow with value $\sum_{j=1}^{n} p_j$ in such a network corresponds to a construction of a feasible schedule for Q/ pmtn, r_j , $\tilde{d}_j$ / - . This can be done in $O(mn^3)$ time.

Problem Q/ pmtn, r_i / L_{max}

Again we consider the problem of finding a schedule with the property that each job i is processed in the time window [r_i , d_i] defined by a release time r_i and a deadline d_i of job i. We call such a schedule feasible with respect to the time windows [r_i , d_i]. In the second step, we apply binary search to solve the general problem. As above, we solve the feasibility problem by reducing it to a network flow problem. The same condition for the existence of the feasible schedule is established.

Because a maximal flow in the expanded network can be calculated in $O(m n^3)$ steps, a feasible check can be done with the same complexity. This yields an ε-approximation algorithm with complexity $O(m n^3 (\log n + \log(1/\varepsilon) + \log (\max_{j=1,\ldots n} p_j))$.

Problem Q/ pmtn, prec / L_{max}

In case of precedence constraints, Q2 / pmtn, prec / L_{max} and Q2 / pmtn, r_j / L_{max} can be solved in $O(n^2)$ and $O(n^6)$ time, respectively, by the algorithm of [Law82b].

Problem R/ pmtn / L_{max}

As far as unrelated processors are concerned, problem R/ pmtn / L_{max} can be solved by a linear programming formulation, where x^k_{ij} denotes the amount of T_j processed on P_i in time interval $[d_{k-1} + L_{max}, d_k + L_{max}]$ and where due dates are assumed to be ordered $d_1 < d_2 < ... < d_n$. Thus, we have the following formulation :

$$\text{minimize} \quad L_{max}$$

$$\text{subject to} \quad \sum_{i=1}^{m} p_{ij} x_{ij}^{(1)} \leq d_1 + L_{max}, \qquad j = 1, 2, ..., n$$

$$\sum_{i=1}^{m} p_{ij} x_{ij}^{(k)} \leq d_k - d_{k-1}, \quad j = k, k+1,...,n, \quad k = 2,3,..,n$$

$$\sum_{j=1}^{n} p_{ij} x_{ij}^{(1)} \leq d_1 + L_{max}, \qquad i = 1, 2, ..., m$$

$$\sum_{j=k}^{n} p_{ij} x_{ij}^{(k)} \leq d_k - d_{k-1}, \quad i = 1, 2, ...,m, \quad k = 2,3,..,n$$

$$\sum_{i=1}^{m} \sum_{k=1}^{j} x_{ij}^{(k)} = 1 \quad , \qquad j = 1, 2, ..., n.$$

Solving the LP problem we obtain n matrices $T^{(k)} = [t^{(k)*}_{ij}]$, k = 1, 2, .., n ; then an optimal solution is constructed by an application of Algorithm 10 for R / pmtn/ C_{max} to each matrix separately.

When precedence constraints forma uni-connected activity network, can also be solved via the same modification of the LP problem as described for the C_{max} criterion[Slo78].

43. Complexity Results

We summarize scheduling problems complexity with jobs to be processed on parallel machines.

Problems polynomially solvable	**Complexity**	**Author(s) and date**
Q / pmtn / L_{max}	O(n logn + mn)	Labetoulle et al (1984)
Q / pmtn; r_i / L_{max}	$O(mn^3 (\log n + \log(1/\varepsilon) + \log(\max p_i)$	Federgruen & Gronevelt (1986)
R / pmtn; r_i / L_{max}	Linear programming	Lawler & Labetoulle (1978)
P / intree ; $p_i = 1$ / L_{max}	O(n)	Brucker et al (1977)
P 2 / prec, $p_i = 1$ / L_{max}		Momma(1982)

P 2 / prec, $p_i = 1$; r_i / L_{max}	$O(n^{\log 7})$	Garey & Johnson(1976)
P / intree ; pmtn / L_{max}	$O(n^3 \log m)$	Garey & Johnson(1977)
Q2 / prec; pmtn / L_{max}	$O(n^2)$	Lawler (1982)
Q2 / prec; pmtn ; r_i / L_{max}	$O(n^2)$	Lawler (1982)
	$O(n^6)$	Lawler (1982)

Problems NP-hard	**Author(s)**
P / outtree ; $p_i = 1$ / L_{max}	Brucker & al (1977)

5. Open Problems : [Sch99]

In a recent paper, Schuurman and Woeginger [Sch99] discuss what they consider to be the most vexing open questions in the area of polynomial time approximation algorithms for NP-hard deterministic machine scheduling problems. They summarize what is known on these problems, discuss related results and they provide pointers for literature.

We consider only scheduling problems that are NP-hard. A distinction is made between *positive results*, which establish the existence of some approximation algorithm, and *negative results*, which disprove the existence of certain approximation results under the assumption $P \neq NP$. They define Positive results and negative results.

A standard way of dealing with NP-hard problems is not to search for an optimal solution, but to go for *near-optimal* solutions. An algorithm that return near-optimal solutions is called an *approximation algorithm*. If it does this in polynomial time, then it is called a *polynomial time* approximation algorithm.

An approximation algorithm that always returns a near-optimal solution with cost at most a factor ρ above the optimal cost (where $\rho > 1$ is some fixed real number) is called a ρ-approximation algorithm, and the value ρ is called the *worst-case performance guarantee*. A family of $(1 + \varepsilon)$ -approximation algorithms over all $\varepsilon > 0$ with polynomial running times is called a *polynomial time approximation scheme* or PTAS, for short. If the time complexity of a PTAS is also polynomially bounded in $1 / \varepsilon$, then it is called a *fully* polynomial time approximation scheme or FPTAS.

FPTAS is essentially the strongest possible polynomial time approximation result that we can derive for an NP-hard problem.

Positive results in the area of approximation concern the design and analysis of such polynomial time approximation algorithms and schemes. For every hard problem, we would like to know whether it possesses a polynomial time approximation algorithm with constant

worst-case performance guarantee, or a PTAS, or even an FPTAS. Negative(inapproximatively) results disprove the existence of good approximation algorithms under the assumption $P \neq NP$.

Two kind of algorithms exist. The " compound algorithms ", which selects the best schedule produced by two or more methods and the " improvement algorithms " , which iteratively improve schedules produced by a single method.

Four open problems deal with the makespan criterion, in parallel machine environments.

Open problem 1. Design a polynomial time approximation algorithm for P/ prec/ C_{max} or for P / prec, pj = 1 / C_{max} with worst-case performance 2 - δ (in fact, even an algorithm whose time complexity is exponential in m might be interesting). Provide a 4/3 + δ inapproximability result for P / prec / C_{max}

Design a PTAS for problem P 2 / prec / C_{max} . Design a PTAS (or even an FPTAS) for problem P3/ prec, $p_j = 1$ / C_{max} .

Open problem 2. Design a polynomial time approximation algorithm for Q/ prec/ C_{max} with constant performance guarantee (i.e. independent of the number of machines), or prove a non-constant lower bound under the assumption $P \neq NP$ (i.e. a bound that tends to infinity as m goes to infinity) .

Open problem 3. For P / prec, $p_j = 1$, c = 1/ C_{max} , improve the performance guarantee to 7/3 - δ or improve the improve the inapproximability bound to 5/4 +δ.

For P∞ / prec, $p_j = 1$, c = 1/ C_{max}, improve the performance guarantee to 4/3 - δ or improve the inapproximability bound to 7/6 + δ.

Decide whether there exists a polynomial time approximation algorithm with constant performance guarantee for P∞ / prec, $p_j = 1$, c = 1/ C_{max}.

Decide whether there exists a polynomial time approximation algorithm with constant performance guarantee for P∞ / prec, c_{jk} / C_{max}.

Open problem 4. Design a polynomial time approximation algorithm for R // C_{max} with worst-case performance of 2 - δ or provide a 3/2 + δ inapproximability result for R // C_{max} .

Chapitre 8 : Résolution approchée des problèmes difficiles

Le but de ce chapitre est de présenter une classe de méthodes pour résoudre les problèmes d'optimisation combinatoire NP-difficiles : les heuristiques et les méta-heuristiques.

1. Les heuristiques : Présentation générale

Une méthode approchée ou heuristique pour un POC est un algorithme qui a pour but de trouver une solution réalisable, tenant compte de la fonction objectif, mais sans garantie d'optimalité. On a vu précédemment qu'on ne connaît pas d'algorithmes polynomiaux pour les problèmes NP-difficiles et la conjecture P ≠ NP fait qu'il est peu probable qu'il en existe. Les méthodes exactes ont une complexité exponentielle sur ces problèmes, et seules les heuristiques peuvent résoudre des cas de grande taille.

En pratique, on trouve beaucoup de problèmes mal formulés, ou ayant de nombreuses contraintes, ou encore de complexité inconnue. Même si le problème est facile, l'ajout de nouvelles contraintes peut le rendre NP-difficile. Dans ces conditions, il vaut mieux dès le départ une approche heuristique, beaucoup plus facile à modifier.

Il existe un très grand nombre d'heuristiques selon les problèmes à traiter, et il est aisé d'en inventer. On distingue cependant les grands types suivants qui vont être détaillés.

a) méthodes construisant une seule solution, par une suite de choix partiels et définitifs. On les appelle " méthodes gloutonnes " (greedy) quand elles cherchent à chaque itération à faire le choix le plus avantageux.
b) Recherches locales, appelées recherches de voisinages (neighborhood or local search). On part d'une solution initiale et par transformations successives, on construit une suite de solutions de coûts décroissants. Le processus s'arrête quand on ne peut plus améliorer la solution courante.
c) Recherches globales (global search). Une recherche locale peut être piégée dans un minimum local. Les méthodes dites de recherche globale, appelées aussi " méta-heuristiques" sortent de ces pièges en construisant aussi une suite de solutions, mais dans laquelle la fonction économique peut temporairement remonter.

2. Evaluation des heuristiques

Le problème de l'évaluation des heuristiques est crucial. Elles n'offrent aucune garantie d'optimalité : elles peuvent trouvé l'optimum pour certaines données ou en être très éloignées.

Supposons qu'on étudie un POC pour lequel on dispose déjà d'une méthode exacte, optimale, de référence. Pour une heuristique H et une donnée d, on note H(d) le coût de la solution heuristique et OPT(d) le coût optimal.

On appelle " performance relative " de H sur d le quotient $R_H(d) = \frac{H(d)}{opt(d)}$. Pour un problème de minimisation, $R_H(d) \geq 1$.

On préfère parfois " la distance à l'optimum " en % : 100 ($R_H(d)$ -1). La performance relative peut être bornée à priori ou être imprévisible et constatée à posteriori, après exécution de l'algorithme.

2.1. Evaluation à priori : performance relative au pire (worst case performance ratio)

On appelle performance relative au pire P_H d'une heuristique H sa plus mauvaise performance relative sur l'ensemble des données possibles, $P_H = \max_{d} R_H(d)$.

C'est une garantie de performance, qu'on obtient mathématiquement par analyse théorique de H et non par des statistiques ou par énumération des données possibles.

Les démonstrations se font en deux temps : on borne $R_H(d)$ pour toute donnée d, puis on construit une donnée montrant que ce pire cas peut être effectivement atteint. Ce genre de résultat est en général très difficile à obtenir et on n'en connaît que pour quelques problèmes.

- une heuristique H avec un P_H = 1.5 pour un POC donné signifie que cette heuristique n'est jamais à plus de 50% de l'optimum.
- un écart de 50% semble décevant mais certains problèmes difficiles n'ont même pas d'heuristique avec une performance relative au pire égale à une constante c.
- Les pires écarts à l'optimum sont souvent obtenus sur des problèmes théoriques construits exprès.
- Les heuristiques peuvent être très bonnes en moyenne sur des problèmes pratiques.

On n'a pas pu trouver pour les performances relatives au pire, l'équivalent de la transformation polynomial des problèmes NP-complets.

Par conséquent, une bonne heuristique pour un problème NP-difficile ne permet pas d'en déduire une aussi bonne pour un autre problème NP-complet.

2.2. Evaluation à posteriori

a) bornes inférieures de l'optimum

Le plus souvent, on ne sait pas établir mathématiquement le comportement au pire. On ne peut alors évaluer les résultats qu'après exécution de l'heuristique H. Pour évaluer le résultat pour une donnée d, on peut calculer la performance relative $R_H(d)$ à condition de connaître l'optimum. Mais il n'existe pas de méthode exacte de complexité polynomiale pour les problèmes NP-difficile. La valeur exacte de l'optimum n'est donc pas calculable en une durée acceptable pour les POC de grande taille. On peut cependant obtenir une évaluation moins serrée si on dispose d'une "évaluation par défaut", un minorant, B(d) pour l'optimum opt(d).

$$\text{Alors } R_H(d) = \frac{H(d)}{opt(d)} \leq \frac{H(d)}{B(d)}.$$

En particulier si H(d) = B(d), alors on s'est " à posteriori" qu'on a atteint l'optimum. On peut toujours trouver des bornes inférieures, mêmes grossières, de l'optimum opt(d).

b) Evaluation statistique

Même si on connaît la performance relative au pire, elle peut être très mauvaise ou atteinte seulement sur des cas particuliers. L'heuristique peut être bonne en moyenne. Une évaluation statistique est recommandée. Pour évaluer k heuristiques H_1, H_2,…, H_k, on génère aléatoirement une série de problèmes, qui comprend typiquement 100 problèmes. Pour chaque matrice de données, on exécute les k heuristiques à évaluer. On stocke dans des tableaux ou sur un fichier le numéro du problème, la borne inférieure calculée, et l'optimum opt(d) si on dispose d'une méthode exacte. Pour chaque problème et chaque heuristique, on note les résultats de chaque heuristiques à tester et le temps d'exécution. L'exploitation des résultats pour une série de problèmes peut se faire par le programme qui a exécuté les heuristiques, par un programme séparé ou par un logiciel de statistiques.

3. Recherches locale et globales

3.1. Méthodes de recherche locale

Une des idées les plus fécondes et les plus utilisées dans les méthodes d'optimisation est de procéder, itérativement en examinant le voisinage de la solution courante, dans l'espoir d'y détecter des directions de recherche prometteuses. Dans les problèmes d'optimisation sur un espace continu, cette idée se concrétise, dans des méthodes de gradient (steepest descent) dont on peut prouver, sous certaines hypothèses (régularité, convexité ou concavité de la fonction à optimiser, la convexité de l'ensemble des solutions admissibles), qu'elles conduisent à un optimum global. Lorsqu'on travaille sur un ensemble discret de solutions, on peut appliquer, un algorithme de descente (ou de plus forte descente) si l'on dispose :

- d'une notion de voisinage qui structure l'espace Ω de solutions;
- d'un moyen efficace pour trouver la meilleure (ou une bonne) solution dans le voisinage d'une solution quelconque.

Partant d'une solution initiale quelconque, on progresse alors " de proche en proche ", tant que l'on trouve une solution meilleure que la solution courante dans le voisinage de celle-ci.

L'algorithme s'arrête lorsque la solution courante est meilleure que la solution courante dans son voisinage. Ce qui est la définition d'un optimum local.

Un schéma général d'algorithme de descente est du type:

Algorithme 1 (de descente)

On suppose que pour toute solution $x \in \Omega$, il est défini un ensemble $V(\Omega) \subseteq \Omega$ de solution voisines de x .

Initialisation : soit x_0 , une solution courante;

Etape n : soit $x_n \in \Omega$, la solution courante;
Sélectionner une solution $x^* \in V(x_n)$;
Si $F(x^*) \leq F(x_n)$;
Faire $x_{n+1} = x^*$ et passer à l'étape n + 1
Sinon x_n est la meilleure solution trouvée;
Stop.

Si $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ est une permutation de tâches :

- le voisinage " 2-échange" consiste en la permutation de 2 tâches quelconques
- le voisinage " k-opt" : toutes les permutations qui peuvent s'obtenir en rejetant k tâches de la permutation courante.

C'est ce type de voisinage qui est utilisé dans une heuristique de descente.

3.2. Méthodes de recherche globale

Ces méthodes sont des recherches locales modifiées dans l'espoir d'éviter le piégeage dans un minimum local. Leur conception commence par l'étude d'une recherche locale, définition d'un voisinage avec transformations simples, que l'on promet en recherche globale si elle s'avère insuffisante. Il s'agit des méthodes appelées aussi " Méta-heuristiques ".

LE RECUIT SIMULE (simulated annealing)

Le " recuit simulé" a été inventé par Kirkpatrick, Gelatt et Vecchi (1983). Ils ont pu résoudre quasi-optimalement des problèmes de voyageur de commerce à 5000 sommets. Ils s'inspirent de la méthode de simulation de Métropolis (1950) en mécanique statistique. L'analogie s'inspire du " recuit des métaux " en métallurgie. Un métal refroidi trop vite présente de nombreux défauts microscopiques, c'est l'équivalent d'un minimum local pour un POC. Si on le refroidit lentement, les atomes se réarrangent, les défauts disparaissent et le métal a une structure ordonnée, équivalent du minimum global pour un POC.

L'énergie d'un système est représentée par un réel T, la température. Une méthode de recuit simulé s'obtient :

- On tire au sort une transformation, une solution s' de V(s) au lieu de chercher la meilleure ou la première solution voisine améliorante.
- On construit la solution résultante s' et sa variation de coût $\Delta f = f(s') - f(s)$.
- Si $\Delta(f) \leq 0$, le coût diminue et on affecte la transformation améliorante comme dans une recherche locale (s : = s').
- Si $\Delta(f) > 0$, le coût remonte, c'est un rebond, qu'on va pénaliser d'autant plus que la température est basse et que $\Delta(f)$ est grand. Une fonction exponentielle a les propriétés désirées. On calcule une probabilité d'acceptation (fonction de Bolzman) $a = e^{\Delta(f) / T}$, puis on tire au sort une valeur p de [0, 1]. Si $p \leq a$, la transformation est acceptée bien qu'elle dégrade le coût, et on fait s : = s'. Sinon, la transformation est rejetée : on conserve s pour l'itération suivante.

Pour assurer la convergence, T est diminuée lentement à chaque itération, par exemple T : = n T, k < 1 mais proche de un. On peut aussi décroître T par paliers.

Pour être efficace, un recuit doit diminuer T assez lentement, en plusieurs milliers ou dizaines de milliers d'itérations. Il dure beaucoup plus longtemps qu'une recherche locale.

On s'arrête quand T atteint un seuil fixé ε, proche de zéro. Le réglage des paramètres est assez délicat. Il est prudent de prévoir deux tests d'arrêt supplémentaires : un limitant le nombre d'itérations à une valeur MaxIter, et un limitant le nombre d'itérations sans changement de coût à une valeur MaxGel. MaxGel doit être assez grand.

Contrairement à une recherche locale, un recuit simulé où un refroidissement trop rapide donne une solution finale qui n'est pas la meilleure trouvée.

Il vaut mieux stocker en cours de route toute solution améliorante. Les valeurs typiques sont MaxIter = 10.000, $\varepsilon = 10^{-2}$, MaxGel = 200, k = 0.9995.

Algorithme 2 (Général du recuit simulé)

Le schéma général du recuit est :
Construire une solution initiale quelconque s
Z* : = f(s) { meilleur coût obtenu }
S* : = s { meilleure solution connue }
Initialiser T et ε
Initialiser MaxIter et MaxGel
NIter : = 0 { nombre d'itérations}
NGel : = 0 { nombre d'itérations sans améliorations }
Répéter
NIter : = NIter + 1
Tirer au sort une solution s' dans V(s)
Calculer la variation de coût Δ(f).
Si Δ(f) < 0
Alors Accept : = vrai
Sinon
Tirer au sort p dans [0, 1]
Accept : = p ≤ exp(-Δ(f)/T)
FS
Si Accept alors
Si Δ(f) = 0 alors NGel : = Ngel + 1
Sinon NGel : = 0
Si f(s') < Z' alors
Z* : = f(s')
S* : = s'
FS
S : = S'
FS
T : = kT
Jusqu'à (T ≤ ε) ou (NbIter = MaxIter) ou (NGel = MaxGel)

LES METHODES TABOUES

a) principe

Les recherches taboues ont été inventé par Glover (1985). Elles sont de conception plus récente que le recuit, n'ont aucun caractère stochastique et paraissent meilleures à temps d'exécution égal. Elles sont caractérisées par trois points:

- A chaque itération, on examine complètement le voisinage V(S) de la solution actuelle s, et on va sur la meilleure solution s', même si le coût remonte.
- On s'interdit de revenir sur une solution visitée dans un passé proche grâce à une liste taboue T stockant de manière compacte la trajectoire parcourue. On cherche donc s' dans V(s) - T.
- On conserve la meilleure solution trouvée, contrairement au recuit, c'est rarement la dernière. On stoppe après un nombre maximal d'itérations sans améliorer la meilleure solution ou quand V(s) - T = ⊘. Il ne se produit que sur de très petits problèmes pour lequel le voisinage tout entier peut se retrouver enfermé dans T.
- Une méthode taboue échappe aux minima locaux, même si s est un minimum local, l'heuristique va s'échapper de la région V(s) en empruntant un " col ".

En début de calcul, la méthode trouve une suite de solutions améliorées, comme une recherche locale. On voit ensuite le coût osciller puis redescendre vers un meilleur minimum local. Les améliorations deviennent de plus en plus rares au cours des itérations.

b) la liste tabou

Le point délicat est la capacité NT de la liste taboue T. Glover montre que NT de 7 à 20 suffit pour empêcher les cyclages, quelle que soit la taille du problème.

T fonctionne comme une " mémoire à court terme ".

A chaque itération, la NT-ième transformation de T, la plus ancienne, est écrasée par la dernière effectuée. En pratique, T se gère simplement avec une structure de données de type " file ". En théorie, il faudrait stocker les NT dernières solutions sur lesquelles l'algorithme est passé. En ordonnancement, il faudrait conserver les NT dernières permutations des n tâches. Si la solution actuelle es " s ", la prochaine solution sur laquelle on va se déplacer doit être dans V(s) - T. Il faut vérifier que la permutation générée n'est pas déjà dans T.

Les voisinages considérés étant souvent grands, il est évident que le test répétitif de présence dans T est très coûteux, sans parler de la mémoire nécessaire pour coder le détail de NT solutions. Le stockage explicite des solutions n'est donc jamais pratiqué.

On recommande les critères moins coûteux pour éviter le cyclage. Ces critères sont aussi plus restrictifs : ils peuvent interdire certains mouvements qui ne conduiraient pas à un cyclage. Les plus utilisés de ces critères consiste à stocker dans T les transformations ayant permis de changer de solution, et d'interdire de faire les transformations inverses pendant NT itérations, on stockerait les deux tâches permutées à chaque itération, et on s'interdirait de les remettre dans une permutation pendant NT itérations. Un critère plus simple est d'interdire d'utiliser les NT dernières valeurs de la fonction objectif. Il suffit de stocker un nombre par itération.

Algorithme 3 : (général Tabou)

```
Construire une solution initiale " s " et calculer son coût Z = f(s)
Z* : = Z { meilleur coût obtenu }
S* : = s { meilleure solution connue }
Initialiser MaxIter { nombre maximum d'itérations }
T : = ⊘ { la liste tabou T est vide }
NIter : = 0
Répéter
        NIter : = NIter + 1 {exploration du voisinage V(s) de s }
        Z" : = + ∞
        Pour toute solution s' de V(s)
              Si s' ∉ T alors       { si s' n'est pas taboue }
                    si f(s') < Z"   alors     { mise à jour du voisin " le moins pire " }
                       s" : = s'
                       Z" : = f(s')
                       Stocker l'inverse de la transformation dans u
                   FS
              FS
         FP
Si  Z" <  + ∞    alors    { si un voisin non tabou a été trouvé }
   S : = s"
   Z : == Z"
Enlever la transformation en tête de T  ( la plus ancienne )
  Ajouter u en fin de T
      Si Z < Z"   alors    { mise à jour de la meilleure solution }
         S* : = s
          Z* : = Z
      FS
FS
Jusqu'à  ( NIter = MaxIter ) ou ( Z" = + ∞ )
```

Les algorithmes génétiques

Cette classe de méthodes a été inventée par Holland dans les années soixante, pour imiter les phénomènes d'adaptation des êtres vivants. L'application aux problèmes d'optimisation a été développée ensuite par Goldberg.

Par analogie avec la reproduction des êtres vivants, on part d'une population initiale de N solutions aléatoires. Le point délicat est d'arriver à coder chaque solution comme une chaîne de caractères (analogue à un chromosome d'une cellule vivante). Le chromosome est formé de sous chaînes appelées " gènes ", chacun codant une caractéristique de la solution.

Une itération est appelée " génération ". A chaque génération, on choisit au hasard NC < N paires de chromosomes à reproduire, avec une probabilité croissante avec leur adaptation. Chaque paire (x, y) choisie subit une opération de "croisement" (cross-over). Un gène est choisi aléatoirement dans x, puis permuté avec le gène de même position dans y. On pratique également un faible " taux de mutation " : NM chromosomes sont choisis et subissent une modification aléatoire d'un gène. La nouvelle population est formée de la population précédente et des chromosomes nouveaux générés par mutation ou croisement.

On définit pour chaque solution une " mesure d'adaptation au milieu " appelée " fitness ". Pour un POC, cette fonction est la fonction objectif. On élimine alors les solutions les moins adaptées pour maintenir un effectif constant de N solutions. On recommence le même processus pour l'itération suivante. L'intérêt est que les bonnes solutions sont encouragées à échanger par croisement leurs caractéristiques et à engendrer des solutions encore meilleures. De plus, les solutions finales vont être concentrées autour des minima locaux, et la méthode fournira donc un choix de plusieurs solutions possibles au décideur.

Notons G la fonction d'évaluation choisie. Pour engendre la génération $X^{(n+1)}$ de solutions à partir de la population courante $X^{(n)}$, on applique le schéma décrit ci dessous.

Sélectionner dans $X^{(n)}$ un ensemble de paires de solutions de hautes qualités.

Appliquer à chacune des paires de solutions sélectionnées un opérateur de croisement qui produit une ou plusieurs solutions " enfants ".

Remplacer une partie de $X^{(n)}$ formée de solutions de basse qualité par des solutions " enfants " de bonne qualité;

La population $X^{(n+1)}$ est obtenue après avoir appliqué un opérateur de mutation aux solutions ainsi obtenues.

Algorithme 4 (génétique)

Initialisation : soit $X^{(0)} \subseteq X$, une population initiale;

Etape n : soit $X^{(n)} \subseteq X$, la population courante;

- sélectionner dans $X^{(n)}$ un ensemble de paires de solutions da haute qualité;
- appliquer à chacune des paires de solutions sélectionnées un opérateur de croisement qui produit une ou plusieurs solutions " enfants ".
- remplacer une partie de $X^{(n)}$ formée de solutions de basse qualité par des solutions enfants de haute qualité;
- appliquer un opérateur de mutation aux solutions ainsi obtenues; les solutions éventuellement mutées constituent la population $X^{(n+1)}$;

si la règle d'arrêt est satisfaite, stop;

sinon, passer à l'étape n + 1.

a) La sélection

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

L'individu " k " est choisi tel que : $\sum_{i=1}^{k-1} p_i < r_i \leq \sum_{i=1}^{k} p_i$.

La probabilité que x_k soit choisi est aussi bien égale à p_k.

Cette procédure est appelée " la roulette russe ". La procédure est itérée jusqu'à ce que l'on chois un nombre fixé d'individus.

b) Le croisement

Soit deux solutions x et y sélectionnées parmi les solutions de haute qualité. Un opérateur de croisement (croosover) fabrique une ou deux nouvelles solutions x' et y' en combinant x et y.

Si x et y sont deux vecteurs de 0 et 1, un opérateur de croisement classique (two-point crossover) consiste à sélectionner aléatoirement deux positions dans les vecteurs et à permuter les séquences de 0 et 1 figurant entre ces deux positions dans les deux vecteurs.

Pour des vecteurs x = 0 1 1 0 1 1 0 0 et y = 1 1 0 0 1 0 1 0, si les positions " après 2 " et " après 5 " sont choisies, on obtient après croisement :

x' = 0 1 ı 001ı100 et y' = 1 1ı 1 0 1 ı 0 1 0.

De nombreuses variantes d'un tel opérateur peuvent être imaginées. Ces variantes doivent être adaptées au codage des solutions et favoriser la transmission des " bonnes sous-structures " des solutions parents aux enfants.

Pour le problème d'ordonnancement avec le codage liste des tâches ne convient pas.

x = A B C D E F G H et y = B E F H A D G C, en croisant entre les positions " après 2 " et " après 5 ", on obtiendrait : x' = A B ı F H A ı F G H et y' = B E ı C D E ı D G C qui ne sont pas des permutations.

Un opérateur de croisement spécialement conçu pour les listes données (OX-crossover) implémente l'idée suivante. Les deux solutions parentes sont " préparées " avant l'échange des séquences situées entre les deux positions choisies au hasard.

Dans cette exemple de tâches, la zone d'échange de x est préparée à accueillir la séquence des villes F, H, A de y.

Pour ce faire, on remplace chacune des villes F, H et A dans le vecteur x par une place vide symbolisée par une *, soit * B ı C D E ı* G * et en commençant à la droite de la zone d'échange, on tasse les tâches restantes de la permutation dans l'ordre de la permutation x en oubliant les *, ce qui donne : D E ı * * *ı G B C. Les * se retrouvent dès lors dans la zone d'échange, alors que l'ordre de parcours des autres tâches n'a pas été changé. On procède de même pour y : B * ı F H A ı* G * devient H A ı * * *ı G B F.

On procéde alors à l'échange des séquences, ce qui donne deux permutations enfants x' et y' :

x' = D E ı F H A ı G B C et y' = H A ı C D E ıG B F.

Un certain nombre de paires d'enfants sont ainsi générés et remplacent une partie des parents choisis parmi les moins performants.

c) La Mutation

Une mutation est une perturbation introduite pour modifier une solution individuelle, par exemple la transformation d'un 0 en un 1 ou inversement dans un vecteur binaire. Dans l'ordonnancement de type permutation, une mutation peut être une permutation arbitraire de deux tâches. En général, on décide de muter une solution avec une probabilité assez faible. Le but de la mutation est d'introduire un élément de diversification et d'innovation.

Bibliographie et Références

[Aus03] Ausiello, G, Crescenzi, P, Gambosi, G, Kann, V, Marchetti-Spaccamela, A et Protasi, M. Complexity and Approximation. Combinatorial optimization problems and their approximability properties. Springer 2003.

[Bak74] Baker, K.R. Introduction to sequencing and scheduling. John Wiley & sons, Inc. 1974.

[Bla77] Blazewicz, J. Simple algorithms for multiprocessor scheduling to meet deadlines, *Infor. Process. Lett.*6, pp. 162-64, 1977.

[Bla76a] Blazewicz, J ; Cellary, W ; Slowinski, R and Weglarz, J. Deterministyczne problemy szeregowania zadan na rownoleglych procesorach, Cz. II. *Zbiory zadan niezaleznych Podstawy sterowania, No. 6, pp. 155-178.1*976.

[Bla77] Blazewicz, J ; Cellary, W and Weglarz, J. A strategy for scheduling splittable tasks to reduce schedule length, *Acta Cybernitica* 3, pp. 99-106, 1977.

[Bla94] Blazewicz, J ; Ecker, K.H ; Schmidt, G and Weglarz, J. *Scheduling in computer and manufacturing systems*. Second revised Edition. Springer-verlag, 1994.

[Bru95] Brucker, P. Scheduling algorithms. Springer, 1995.

[Bru76b] Brucker, P.J. Sequencing unit-time jobs with treelike precedence on m machines to minimize maximum lateness. *Proceedings IX International Symposium o Mathematical Programming, Budapest*, 1976.

[Bru77] Brucker, P ; Garey, M.R and Johnson, D.S. Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness, *Math. Oper. Res.*2, pp. 275-284, 1977.

[Bru74] Bruno, J ; Coffman, E.J Jr. And Sethi, R. Scheduling independent tasks to reduce mean finishing time, *Comm. ACM* 17, pp. 382- 387, 1974.

[Chen75] Chen, N.F and Liu, C.L. On a class of scheduling algorithms for multiprocessors computing systems, in T.Y Feng edition, Parallel processing, Lecture notes in computer science 24, Springer Verlag, pp. 1-16, 1975.

[Con67]Conway, R.W, Maxwell, W.L and Miller, L.W. *Theory of scheduling*, Addison-Wesley, Reading, Mass. 1967.

[Cof72] Coffman, E.G Jr. and Graham, R.L. Optimal scheduling for two-processor systems, Acta Informatica, NO1. , pp. 200-213, 1972.

[Cof 76a] Coffman, E.G Jr. . Scheduling in computer and job shop systems. J. Wiley, 1976.

[Cof76b] Coffman, E. Jr and Sethi, R. A generalized bound on LPT sequencing, *RAIRO-Informatique*, No10, pp. 17-25, 1976.

[Cof84] Coffman, E.G Jr, Frederickson, N and Luecker, G.S. A note on expected makespan for largest-first sequences of independent task on two processors, *Mathematics of Operations Research*. No9, pp. 260-266, 1984.

[Cof91] Coffman, E.G Jr. And Gary, M.R. Proof of the 4/3 conjecture for preemptive versus non-preemptive two-processor scheduling, Report Bell Laboratories, Murray Hill, 1991.

[Du88] Du, J ; Leung, Y.T and Young, G.H. Minimizing mean flow time with release time constraints, T*echnical report*, Computer Science Program , University of Texas, Dallas, 1988.

[Du91] Du, J ; Leung, Y.T and Young, G.H. Scheduling chain structured tasks to minimize makespan and mean flow time, *Inform. And Compt.* 92, pp. 219-236, 1991.

[Eas64] Eastman, W.L, Evon, S and Isaacs, I.M. Bounds for the optimal scheduling of n jobs on m processors. *Management science*, vol. 11, No.2, pp., Nov-1964.

[Fed86] Federgruen, A and Groenevelt, H. Preemptive scheduling of uniform machines by ordinary network flow techniques, *Management Sciences* 32, pp. 341-349, 1986.

[Gar76] Garey, M.R and Johnson, D.S. Scheduling tasks with non-uniform deadlines on two processors, *J. Assoc. Comput. Mach.* 23, pp. 461-426, 1976.

[Gar77] Garey, M.R and Johnson, D.S. Two-processor scheduling tasks with start-times and deaadlines, *SIAM J. Compt.* 6, pp. 416-426. 1977.

[Gar81] Garey, M.R ; Johnson, D.S ; Simons, B.B and Tarjan, R.E. Scheduling unit time tasks with arbitrary release times and deadlines, *SIAM J. Comput.* 10, pp. 256- 269, 1981.

[Gar83] Garey, M.R ; Johnson, D.S ; Tarjan, R.E and Yannakakis, M. Scheduling opposing forests, SIAM Journal Algebraic and discrete mathematics, No4. , pp. 72-93, 1983.

[Gon77] Gonzalez, T. Optimal mean finish time preemptive schedules, Technical report 220 ; Computer science department, Pensylvania state Univ. 1977.

[Gon80] Gonzalez, T and Johnson, D.S. A new algorithm for preemptive scheduling of trees, *J. Assoc. Comput. Mach.* 27, pp. 287-312, 1980.

[Gon78] Gonzalez, T and Sahni, R. Preemptive scheduling of uniform processor systems. J. Assoc. Comput. Mach., No. 25, pp.92-101, 1978.

[Got04] Groupe Gotha. Sous la coordination de P. Baptiste, E. Néron et F. Sourd. Modèles et algorithmes en ordonnancements. Exercices & problèmes corrigées. Ellipses, 2004.

[Gra66] Graham, R.L. Bounds for certain multiprocessing anomalies, *Bell System Technology Journal*, No 45, pp. 1563-1581. 1966.

[Gra79] Graham, R.L ; Lawler, E.L ; Lenstra, J.K and Rinnooy Kan, A.H.G. Optimization and approximation in deterministic sequencing and scheduling theory : a survey, *Annals of discrete mathematics*, No. 5, pp. 287-326, 1979.

[Hor74] Horn, W.A. Some simple scheduling algorithms, *Naval Research Logistics Quarterly.*21, pp. 177-185, 1974.

[Hor76] Horowitz, E and Sahni, R. Exact and approximation algorithms for scheduling non-identical processors, *J. Assoc. Comput. Mach.* 23, pp. 317-327, 1976.

[Hor77] Horvath, E.G ; Lam, A and Sethi,R. A level algorithm for preemptive scheduling. Journal of association computing machines, No. 24, pp. 32-43, 1977.

[Hu61] Hu, T.C. Parallel sequencing and assembly line problems. *Operations Research*, vol.9, No.6, pp.841-848,Nov-1961.

[Iba77] Ibarra, O.H and Kim, C.E. Heuristic algorithms for scheduling independent tasks on non-identical processors. *J. Assoc. Comput. Mach.*, No. 24, PP. 280-289, 1977.

[Jac55] Jackson, J.R Scheduling a production line to minimize tardiness, Res. Report 43, Management Research Project, University of California, Los Angeles, 1955.

[Kar72] Karp, R.M. *Reducibility among combinatorial problems*, in R.E.Miller, J.W.Thatcher edition, complexity of computer computations, Plenum Press, New York, pp. 85-104 ; 1972.

[Kar74] Karzanov, A.W. Determining the maximal flow in a network by the method of preflow, *Soviet Math. Dokl.* 15, pp.434-37. 1974.

[Kun76] Kunde, M. Beste Schranke beim LP scheduling, Bericht 7603, Institut fur informatik und praktische Mathematik, University Kiel, 1976.

[Lab84] Labetoulle, J ; Lawler, E.L ; Lenstra, J.K and Rinnooy Kan, A.H.G. Preemptive scheduling of uniform machines subject to release dates, in H.R. Pulleyblank(ed.) : *Progress in Combinatorial Optimization* , Academic Press, 1984.

[Lam77] Lam, S and Sethi, R. Worst case analysis of two scheduling algorithms, SIAM Journal of computing, No. 6, pp. 518-536, 1977.

[Law82b] Lawler, E.L. Preemptive scheduling in precedence-constrained jobs on parallel machines, in : M.A.H Dempster ; J.K.Lenstra and A.H.G.Rinnooy Kan (eds.), *Deterministic and stochastic scheduling,* Reidel, Dordrecht, pp. 101-123, 1982.

[Law78] Lawler, E.L and Labetoulle, J. Preemptive scheduling of unrelated parallel processors by linear programming. *J. Assoc. Comput. Mach.* 25, pp. 612-619, 1978.

[Len77]Lenstra, J.K ; Rinnooy Kan, A.H.G and Brucker, P. Complexity of machine scheduling problems, *Ann. Discrete. Math.*1, pp. 343-362, 1977.

[Len78]Lenstra, J.K and Rinnooy Kan, A.H.G. Complexity of scheduling under precedence constraints, *Oper. Res.*26, pp. 22- 35, 1978.

[Liu74] Liu, J.W.S and Liu, C.L. Performance analysis of heterogeneous multiprocessor computing systems, in E.Gelenbe, R.Mah edition, *Computer architecture and networks,* pp. 331-343, North Holland, Amsterdam, 1974.

[McN59] McNaughton, R. Scheduling with deadlines and loss functions. *Management science*, vol.6, No.1, pp.1-12, Oct-1959.

[Mom82] Momma, C.L. Linear-time algorithms for scheduling on parallel processors, *Operations Research*, 30, pp. 116-124, 1982.

[Mun69] Muntz, R.R and Coffman, E.. Optimal preemptive scheduling on two-processor systems. *IEEE Transactions on computers,* vol. 18, No.11, pp.1014-1029, Nov-1969.

[Mun70] Muntz, R.R and Coffman, E.. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the ACM,* vol. 17, No.2, pp.324-338, April 1970.

[Pin95] Pinedo, M. Scheduling. Theory, Algorithms and systems. Prentice Hall, 1995

[Rot66] Rothkopt, M.H. Scheduling independent tasks on parallel processors. *Management science,* vol. 12. No.5, pp., Jan-1966.

[Sch99] Schuurman, P and Woeginger, J. Polynomial time approximation algorithms for machine scheduling : Ten open problems. *Journal of scheduling*, 2 , pp. 203-213, 1999.

[Sim83] Simons, B. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines, *SIAM J. Comput.* 12, pp. 294-299. (1983)

[Slo78] Slowinski, R. Scheduling preemptive tasks on unrelated processors with additional resources to minimize schedule length, in G.Bracci, R.C.Lockermann(eds.), Lecture Notes in computer Science, vol. 65, Springer Verlag, Berlin, pp. 536-547, 1978.

[Ull75] Ullman, J.D. NP-complete scheduling problems, *J. Comput. System Sci.*10, pp. 384-393, 1975.

[Ull76] Ullman, J.D. Complexity of sequencing problems, Chapter 4. In [Cof76a].