Modèles mathématiques pour la gestion off-line et on-line des changements d'outils sur une machine flexible

Caroline Privault

Thèse

présentée par

Caroline PRIVAULT

Pour obtenir le titre de

Docteur de l'université Joseph Fourier - Grenoble 1

(Arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : Mathématiques appliquées

Option : **Recherche Opérationnelle**

Modèles mathématiques pour la gestion off-line et on-line des changements d'outils sur une machine flexible

soutenue le 20 janvier 1994 devant le jury suivant :

MM	Catherine Roucairol	Présidente
	Marino Widmer	Rapporteur
	Christian Proust	Rapporteur
	Michel Burlet	Examinateur
	Gerd Finke	Directeur
	Thomas McCormick	Invité

Thèse préparée au sein du laboratoire ARTEMIS-IMAG

"Di monti maestosi, Sopra à i fiumilani, ...
È di scogli è di mare, È di luce turchina,
Eccu cum' elli sò, I lochi duve stò ..."

M.V. Acquaviva

J'exprime de vifs remerciements,

à Madame le professeur **Catherine Roucairol** qui me fait l'honneur et le plaisir de présider ce jury de thèse,

à Messieurs les professeurs **Marino Widmer** et **Christian Proust**, rapporteurs, pour le soin qu'ils ont porté à la lecture de ce travail, pour leurs conseils, et pour les améliorations qu'ils m'ont permis d'y apporter,

à Monsieur le professeur **Gerd Finke**, qui a dirigé de cette thèse au laboratoire ARTEMIS-IMAG, et qui m'a donné la possibilité de découvrir un domaine de recherche passionnant,

à Messieurs **Michel Burlet** et **Thomas McCormick** qui ont bien voulu juger ce travail et qui ont accepté de faire partie de mon jury.

Toute ma gratitude à monsieur **Brahim Chaourar**, pour sa patience, son attention et les précieux conseils qu'il m'a prodigués tout au long de ce travail.

Résumé

L'objet de ce travail est l'étude d'un problème d'ordonnancement, dont le critère d'optimisation est la minimisation du nombre total de changements d'outils sur une machine flexible. Différents problèmes liés à l'outillage et pouvant constituer un obstacle au fonctionnement d'un atelier flexible sont brièvement examinés, (de la gestion de l'inventaire aux changements d'outils). Nous nous concentrons ensuite sur le problème d'ordonnancement avec gestion d'outils sur une seule machine : ce problème est NP-complet.

Un premier aspect qui est la gestion off-line(*) des outils est étudié : différents modèles sont proposés pour la minimisation des changements en fonction d'une séquence des tâches donnée. Dans ce cas, le problème est polynomial. Nous revenons ensuite au problème d'ordonnancement proprement dit, pour lequel plusieurs types de méthodes heuristiques sont décrites et comparées.

La seconde partie du travail est consacrée à la gestion on-line(**) des changements d'outils ; elle se compose de deux chapitres : dans le premier, le modèle que nous allons utiliser est décrit en détail. Il s'agit de la modélisation des problèmes de k-serveurs. Le principe peut être résumé comme suit : sur un réseau de n clients potentiels, on dispose de k serveurs mobiles, avec lesquels on doit répondre on-line aux demandes unitaires et successives des clients, tout en optimisant les déplacements des serveurs. Après un tour d'horizon sur ces problèmes et leurs applications dans la gestion des mémoires en informatique, nous revenons dans le dernier chapitre aux changements d'outils on-line(**). Le modèle des k-serveurs "classique" est généralisé aux problèmes avec service par blocs de demandes, ce qui permet d'adapter l'algorithme de partitionnement, (fortement compétitif pour le service unitaire), au cas plus général des demandes groupées. Cet algorithme dont nous étudions les propriétés et la compétitivité, sert de base à la construction d'une heuristique d'ordonnancement avec gestion on-line des changements d'outils. Cette dernière méthode est comparée aux précédentes sur des exemples numériques.

(*) Gestion **off-line** : les outils sont gérés en fonction d'une séquence de pièces déjà déterminée.

(**) Gestion **on-line** : les décisions sont prises au fur et à mesure de l'arrivée des informations, sans connaissance préalable de la séquence, (séquence de pièces, ou séquence d'appels).

Table des matières

Chapitre 1

La gestion des outils dans un système flexible de production

Afin d'introduire le problème d'ordonnancement avec gestion d'outils dont l'étude fait l'objet de cette thèse, nous commençons par définir le contexte dans lequel il s'inscrit en abordant les principaux problèmes liés à l'outillage d'un atelier flexible. Parmi eux figurent les changements d'outils : ils dépendent des choix qui ont été faits pour l'équipement des machines et le renouvellement du contenu de leur magasin.

Le critère d'ordonnancement que nous étudions ici est précisément la minimisation du nombre de changements d'outils sur une seule machine flexible. Après une description des données du problème ainsi qu'une analyse de sa complexité, nous terminerons ce premier chapitre en donnant un bref aperçu du contenu des chapitres suivants.

1. 1. Les problèmes liés à la gestion de l'outillage

1. 1. 1. Les enjeux de la gestion d'outils

La gestion de l'outillage est devenue au cours des dix dernières années un aspect essentiel et un constituant à part entière des systèmes de contrôle et de gestion des ateliers flexibles : les problèmes d'outillage se posent le plus souvent dans les industries métallurgique (découpes, travail des métaux, ...), où la multiplicité des outils de découpe rend nécessaire une gestion informatisée de ces ressources. Ce phénomène s'est accentué avec les tendances actuelles de la production, qui correspondent aux exigences du marché : la production en petites et moyennes séries d'une grande variété de produits exige une flexibilité de plus en plus grande de l'appareil de production.

D'autres problèmes liés au fonctionnement des ateliers flexibles ont été optimisés comme le transport des pièces entre les machines, ou tout simplement les cadences des machines qui ont

longtemps été la préoccupation première des utilisateurs, (donc des constructeurs de machines), tandis que la gestion des outils pose encore souvent des problèmes et peut entrer pour une part importante dans le temps d'usinage global d'une pièce. Pour définir la gestion d'outils, on a souvent recours à la formule suivante :

Avoir le bon outil, au bon moment, au bon endroit, afin d'être en mesure de produire le type de pièce demandé dans la quantité demandée.

Une mauvaise gestion de l'outillage peut avoir les conséquences suivantes :

- Une sous-utilisation des machines due à un nombre élevé de changements ou à l'absence des outils nécessaires, d'où des temps d'usinage inacceptables.
- Une duplication ou un accroissement du nombre d'outils en circulation, (et une augmentation excessive du nombre de fixations), pour tenter de réduire les changements, d'où des coûts d'inventaire accrus.

Selon Mason (1986), l'accumulation de ces facteurs peut réduire un atelier flexible à une succession de machines indépendantes dans leur fonctionnement ou sous-utilisées, au détriment de la flexibilité. Les chiffres suivants donnent une idée des enjeux de la gestion d'outils :

. 16 % du taux de production prévu ne peut être atteint parce que les outils requis à un instant donné ne sont pas disponibles sur une machine.

. plus de 40 % du temps de travail du chef de fabrication est consacré à la recherche des fournitures et de l'outillage nécessaires.

. Dans les industries métallurgiques, le budget annuel d'une firme consacré à l'outillage (outils, fixations, pièces de rechange, fournitures consommables) est 7 à 12 fois plus grand que son budget en capitaux d'investissement, Mason (1986).

On peut définir plus précisément ce que représente la gestion d'outils dans un atelier, en distinguant cinq points essentiels :

1. 1. 2. Gérer l'information sur les outils

Ce premier point est lié au stockage des outils : pour une localisation rapide des outils, un système de stockage adapté doit être utilisé, ce qui n'est pas toujours le cas : on considère que plus de 30% de l'outillage d'un atelier est disséminé quelque part sur le sol de l'atelier sans utilisation ou justification, Mason (1986). Notons cependant que ces installations représentent souvent un investissement coûteux.

La gestion des informations sur l'outillage passe par la constitution d'une base de donnée contenant une description de chaque outil : identification, caractéristiques, état d'usure, ... ElMaraghy (1985). L'absence de moyens d'inventaire précis rend le contrôle de l'outillage impossible dès le départ, aussi de nombreux systèmes informatiques ont-ils été créés pour

l'inventaire,Veeramani et al (1992). Dans certains cas, ils sont accompagnés d'un système de code barres pour l'identification et le codage de chaque outil, Mason (1986).

1. 1. 3. Affecter les outils aux machines et gérer les remplacements

Pour chaque machine, on doit décider quels seront les outils qui seront affectés à son fonctionnement (outillage différent pour chaque machine, ou au contraire identique, ...). Cette affectation détermine les opérations que la machine pourra exécuter, donc elle conditionne également l'ordonnancement des pièces à produire dans l'atelier. De même, pour chaque type d'affectation les changements d'outils seront faits suivant une stratégie particulière visant à optimiser les manipulations d'outils.

1. 1. 4. Contrôler et remplacer les outils usés ou cassés

Lorsque tous les outils requis pour l'usinage d'une pièce sont dans le magasin de la machine, certains remplacements restent tout de même nécessaires pour cause d'usure, ou tout simplement parce qu'un ou plusieurs outils se sont cassés pendant leur utilisation. Les remplacements doivent se faire suivant une stratégie visant à minimiser le nombre d'arrêts de la machine pour cause d'outil cassé : certains outils peuvent être remplacés avant qu'ils ne cassent (une durée de vie maximum doit être déterminée pour chaque outil) ; d'autre part lorsqu'un outil est cassé et remplacé, on peut choisir de remplacer dans le même temps d'autres outils usés ou qui risquent d'être prochainement hors d'usage à leur tour... A cette occasion, l'information concernant l'état d'usure des outils doit être transmise à la base de donnée gérant les informations sur l'outillage. Le point 4 est donc souvent géré en même temps que le point 1 par l'intermédiaire d'un même système.

1. 1. 5. Contrôler les flux d'outils dans l'atelier

Parce que la capacité des magasins des machines est généralement insuffisante, ou pour cause d'usure comme on l'a vu au point 4, les outils sont amenés à être déplacés d'une machine à l'autre, ou bien entre l'aire de stockage des outils et les machines. Bien que ces déplacements soient encore souvent effectués manuellement, il importe qu'un système de transport adapté à l'atelier soit étudié et mis en place pour faciliter et contrôler les flux d'outils : déplacements par ponts roulants au dessus de l'atelier, par chariots porte-outils à hauteur des machines, utilisation de chariots filoguidés, ...

Ces quatre aspects sont discutés et décrits en détail dans Veeramani et al (1992), Kiran et Krason (1988) et Mason (1986). Notons qu'ils ne sont pas indépendants : ils peuvent difficilement être optimisés séparément. C'est d'ailleurs souvent un même système qui gère l'inventaire, et les flux d'outils dans l'atelier par exemple. Suivant leurs préoccupations, les professionnels auront souvent tendance à résumer la gestion d'outils à l'un des points

précédents. Par exemple c'est la raison pour laquelle il est fréquent qu'on définisse le problème comme un problème de gestion de l'information. Pour la gestion des flux on aura d'abord recours à une simulation sur l'atelier, pour déterminer les besoins en équipements. Crite et al (1985) décrivent un outil de simulation PATHSIM pour l'évaluation des systèmes de gestion des flux d'outils. De même, c'est au moment de la conception d'un atelier flexible que les besoins en outillage (types d'outils, nombre d'exemplaires, ...) devront être étudiés avec attention : sur ce thème une étude a été faite par Kouvelis (1991).

Mais selon Mason (1986), le problème central reste l'optimisation du nombre de remplacements ou d'échanges d'outils pour cause d'usure ou parce que la capacité du magasin de la machine est insuffisante. Naturellement dans certains cas, lorsque l'atelier comporte peu de machines et que le nombre de types de pièces à produire est inférieur ou égal à 10, l'ensemble des outils peut être petit, et la gestion automatisée des outils ne se justifie pas forcément. On peut alors se contenter de contrôler l'usure des outils, et de les remplacer à temps. Mais lorsqu'on voudra produire une grande variété de petits lots de pièces, le magasin de chaque machine ne pourra contenir à lui seul tous les outils requis par l'ensemble des pièces.

1. 2. Affectation des outils aux machines

1. 2. 1. Stratégies d'affectation

La gestion des remplacements d'outils est liée au système de répartition des outils sur les machines : plusieurs organisations sont possibles qui induisent différentes stratégies de remplacement. Elles dépendent du type de production qui est demandé à l'atelier comme de l'équipement disponible en machines et en outils. Carrie et Perera (1986) ont étudié un problème de chargement des machines et de remplacement des outils sur un cas particulier. Plus généralement, Kusiak (1990) a répertorié 4 façons d'équiper les machines :

- Chaque machine est équipée du même jeu d'outils : cette organisation nécessite la présence d'un magasin de grande capacité sur chaque machine, et son principal désavantage est d'entraîner un sur-équipement en outils et des coûts d'inventaire supplémentaires.

- Chaque machine est équipée d'un jeu d'outils qui lui est propre, chacun de ces outils étant dupliqué à l'intérieur du magasin d'une même machine, et un même outil ne pouvant être monté sur deux machines : les désavantages de cette répartition sont quasiment les mêmes que précédemment.

- Chaque machine possède un jeu d'outils qui lui est propre et a accès à un magasin d'outils de rechange commun à toutes les machines, grâce à un système de contrôle et de manutention. Chaque outil monté sur une machine figure également dans le magasin central : les désavantages des deux organisations précédentes sont atténués, mais en revanche le contrôle et la gestion de l'outillage deviennent plus complexes.

- Les trois organisations précédentes supposent au minimum la duplication de tous les outils. Elles nécessitent parfois de posséder un même type d'outil en autant d'exemplaires qu'il y a de machines dans l'atelier, or cet équipement représente un investissement très coûteux pour l'entreprise. Dans cette répartition, chaque machine dispose d'un jeu d'outils qui lui est propre, et a accès à un stock d'outils qui contient certains outils en magasin sur les machines, et d'autres outils permettant des variantes dans le processus de fabrication des pièces.

1. 2. 2. Les changements d'outils

Il existe plusieurs types de magasins pour le stockage des outils sur les machines : le stockage par chaînes ou chenilles à outils, les cylindres porte-outils, ou encore les barillets ou les tambours à outils. La capacité de ces magasins est très variable ; pour donner un ordre de grandeur, elle peut aller de 6 emplacements à plus d'une centaine. Certaines machines peuvent stocker environ 150 outils, mais dans le même temps le nombre d'outils nécessaires à la fabrication de tous les types de pièces est souvent également très grand :

Berrada (1983) a par exemple étudié un atelier comportant 6 machines dont la capacité va de 45 à 60 outils, devant effectuer 37 opérations différentes sur un ensemble de pièces dont la fabrication met en jeu l'utilisation de 491 outils différents. Ce rapport entre la capacité des magasins et le nombre total d'outils requis amène à opérer des changements d'outils. Lorsqu'on modifie la composition du magasin d'une machine plusieurs stratégies de remplacements sont envisageables :

a) Echange par blocs : lorsqu'un type de pièce vient d'être usiné et qu'on s'apprête à entamer la fabrication d'un autre, on transfère l'ensemble des outils présents dans le magasin de la machine (et qui est associé au dernier type de pièce usinée), vers le magasin de l'atelier, pour monter ensuite l'ensemble des outils requis par le nouveau type de pièce à traiter. Ce système simplifie la gestion des remplacements d'outils car le magasin est simplement vidé avant chaque nouveau type de pièce, mais il nécessite de posséder plus d'un exemplaire de chaque outil. Cette stratégie est plutôt adaptée à une production en grandes séries.

b) Partage de certains outils pendant une phase de production : cela consiste à repérer des pièces qui utilisent un même ensemble d'outils, à charger ces outils communs sur chaque machine, pour entamer une phase de production sans changements d'outils. A la fin

d'une phase, les outils sont remplacés par ceux qui sont requis dans la phase suivante. Cette méthode nécessite la présence d'un magasin de grande capacité sur chaque machine. De plus elle peut limiter les possibilités dans l'ordonnancement des pièces. Notons qu'une approche de ce genre, utilisant des méthodes de technologie de groupe a été développée par ElGomayel et Nader (1983). Tang et Denardo (1988) et Oerlemans (1992) ont étudié le problème du partitionnement des pièces en fonction des outils utilisés, afin de minimiser le nombre d'arrêts de la machine, c'est-à-dire le nombre de phases. Berrada (1983) a étudié un modèle de programmation non-linéaire pour le problème du chargement des machines avec affectation des outils.

c) Démontage en fin d'usinage : lorsqu'une pièce a été usinée par la machine, les outils qui ne sont requis que par cette pièce sont démontés et replacés dans le magasin de l'atelier, pour laisser la place à d'autres utilisés par les autres pièces. Cette méthode vise toujours à partager autant que possible les outils entre les différentes pièces, mais elle est plus difficile à gérer.

d) Ensemble d'outils résidents : cette stratégie est destinée à augmenter la flexibilité de l'atelier. Les outils fréquemment utilisés sont repérés et placés à demeure dans le magasin de chaque machine. Les autres seront montés et démontés au fur et à mesure des besoins. Naturellement cela a pour effet de multiplier le nombre d'outils dans l'atelier, mais si chaque machine est équipée de cette façon l'atelier peut répondre rapidement à une demande de pièces imprévue. En effet, l'atelier ne fonctionne pas toujours suivant un planning de fabrication préétabli qui permette de savoir longtemps à l'avance quelles pièces seront usinées, et dans quel ordre. Certains aléas peuvent bouleverser un ordonnancement d'où l'intérêt de conserver aux machines un maximum de flexibilité.

e) Outils entièrement résidents : cette méthode est la plus simple et aussi la plus coûteuse. Si on dispose de machines dont la capacité des magasins est très importante (plus d'une centaine d'outils), tous les outils requis par l'ensemble des pièces qu'il est d'usage de produire sont montés de façon permanente sur chaque machine. La flexibilité des machines est totale, puisqu'à chaque fois tous les outils nécessaires sont présents et prêts à l'emploi, mais en revanche le nombre d'outils employés dans l'atelier devient très important. Dans ce contexte, un logiciel gérant les remplacements d'outils n'est plus nécessaire car seuls les outils usés ou cassés sont remplacés.

Cette solution est la plus tentante de part sa simplicité et parce qu'elle aplanit tous les problèmes de changements, et réduit les déplacements. Aussi beaucoup d'industriels s'équipent avec des machines à grande capacité, ou organisent et planifient la production de façon à déterminer des familles de pièces et des phases de production durant lesquelles tous les outils

peuvent être contenus dans le magasin de chaque machine. Cependant, l'équipement nécessaire pour pouvoir adopter ce mode de fonctionnement n'est pas toujours disponible, et la production ne s'y prête pas forcément.

Remarque : Les machines sont souvent équipées d'un dispositif automatique pour la mise en place des outils pendant l'usinage de la pièce : il s'agit d'un bras robot qui effectue l'échange d'outil entre le magasin de la machine et la broche. Cet échange est très rapide. S'il s'agit d'un centre d'usinage et que le magasin de la machine est éloigné du bloc d'usinage proprement dit, les outils peuvent être transférés au moyen d'un pont roulant, (voir par exemple Carter (1985)).

Ce ne sont donc pas les temps des changements qui interviennent pendant l'usinage d'une pièce qui sont pénalisants, mais les échanges d'outils entre le magasin de la machine et le magasin à outils de l'atelier, avant la phase d'usinage. Pourquoi ces changements peuvent-ils devenir pénalisants pour la production ? Ils résultent d'un ou plusieurs des facteurs que nous avons évoqués précédemment, et dont l'accumulation aboutit à des temps inacceptables :

* Eloignement du stock d'outils
* Attente de la disponibilité des moyens de manutention : pont roulant, chariots ...
* Perte de temps due aux systèmes de fixation : nombreuses, complexes, ...
* Réglages : absence de gabarit de réglage, tâtonnements, essais puis réglages complémentaires, précision des réglages.

Suivant les cas, et les types d'outils manipulés, les temps de changements sont très variables. Des temps très importants seront observés lorsque les outils sont difficiles à manipuler à cause de leur poids par exemple, ce qui est le cas des formes montées sur les presses pour l'emboutissage des tôles. Kusiak et Finke (1987), et Blazewicz et al (1988) ont étudié des cas similaires. Pour donner un ordre de grandeur, nous citons quelques uns des temps de changement d'outils donnés par Béranger (1989) :

Tour multibroches : 10 mn
Tour Horizontal à commande numérique : 1 mn
Presse d'emboutissage : 22 mn
Presse d'injection d'aluminium en fusion : 84 mn

Dans plusieurs sociétés ayant porté une attention particulière à la réduction des temps de changements, (et particulièrement dans les sociétés japonaises), l'objectif est maintenant le changement d'outil en quelques minutes connu sous le nom de SMED : Single Minute Exchange Die. (voir Shingo (1987))

Notons enfin que le plus souvent, la disposition des outils à l'intérieur du magasin de la machine n'est pas fixée, c'est-à-dire que les outils peuvent être placés à n'importe quel endroit

dans la chaîne ou sur le barillet. Cependant, suivant le type de magasin qui équipe la machine, la localisation des outils pourra avoir une certaine influence sur le temps d'usinage de la pièce. Si le temps d'utilisation effective de l'outil est inférieur au temps nécessaire à sa localisation et son transfert, il peut être intéressant de minimiser le nombre de rotations de la chaîne ou du barillet pour cause de changements d'outils en cours d'usinage. Ce problème est lié à la gamme d'usinage de la pièce : il peut d'ailleurs faire partie des critères qui interviennent dans la génération des gammes d'usinage. L'arrangement optimal des outils dans le magasin a été étudié entre autres par Foulds et Wilson (1993), et Stecke (1989).

Les contraintes liés à l'outillage offrent donc une variété de problèmes d'optimisation dont la solution peut apporter des gains de temps et de coûts non négligeables : Veeramani et al (1992) ont fait un tour d'horizon très complet sur ces problèmes. Parmi eux, figurent des problèmes d'ordonnancement sur plusieurs machines avec gestion d'outils : Proust et al (1991) étudient un problème de flow shop avec prise en compte des temps de montage/démontage des outils, mais ces temps sont supposés connus à l'avance et indépendants de l'ordonnancement. Widmer (1991) a développé une approche tabou pour un problème de job shop avec contraintes d'outillage : la fonction objective utilisée prend en compte le temps d'achèvement total, le nombre de changements d'outils, mais aussi le dépassement des délais de production.

Hertz et Widmer (1993) ont également étudié le job shop avec contraintes d'outillage. Ils ont proposé un autre algorithme fondé sur la méthode tabou. Ces problèmes d'ordonnancement sont très complexes : le critère d'optimisation est le temps d'achèvement total, mais il prend en compte le nombre de changements d'outils, les temps d'usinage de chaque pièce ainsi que leurs gammes de fabrication..

Le problème d'ordonnancement dont l'étude fait précisément l'objet de cette thèse est très particulier car il s'agit d'un ordonnancement sur une seule machine dont le critère d'optimisation est uniquement la minimisation du nombre de changements d'outils ; mais comme nous le verrons, ces restrictions ne diminuent pas pour autant la complexité du problème. Dans la section suivante, nous décrivons précisément les données du problème, et nous étudions sa complexité.

1. 3. Minimisation des changements d'outils sur une seule machine

1. 3. 1. Les données du problème

On considère un ensemble de N pièces, qui doivent être usinées sur une machine flexible, et un jeu de M outils qui représente l'ensemble de tous les outils requis pour usiner ces pièces. Ces outils sont disponibles sur une aire de stockage à proximité. A chaque pièce est associé un ensemble d'outils. La machine est munie d'un dispositif automatique de changement d'outils et

d'un magasin de capacité C. On suppose naturellement que le nombre d'outils nécessaires à la fabrication de chaque pièce n'excède pas la capacité du magasin.

Lorsqu'on entreprend la fabrication d'une pièce, les outils correspondants doivent être placés dans le magasin de la machine avant que la phase d'usinage ne commence. Par la suite, quand on entame la fabrication d'une nouvelle pièce, des changements d'outils peuvent être nécessaires : si le magasin est plein, des outils non requis doivent être retirés pour faire place à d'autres. Il sera plus judicieux de conserver certains outils en priorité, en fonction des pièces qui restent à usiner. On peut ainsi être amené à perdre un temps important en changements et en manipulations d'outils dûs à une mauvaise gestion de l'ordre de passage des pièces sur la machine.

Les changements d'outils étant automatisés, les temps de remplacement sont à peu près équivalents pour chaque outil, et c'est le nombre de changements qui est pénalisant. On cherche donc à déterminer une séquence des pièces, et les plans de chargement du magasin d'outils qui minimisent le nombre total de changements.

Remarque : Notons que dans notre cas, c'est la première stratégie d'allocation des outils parmi celles que nous avons décrites, qui se rapproche le plus de celle qui a été adoptée : bien que le cas soit un peu différent car il n'y a qu'une seule machine, on considère que chaque outil n'est disponible qu'en un seul exemplaire que ce soit sur la machine ou dans le magasin de l'atelier. De même, la stratégie de remplacement qui est utilisée ici se rapproche plutôt de la troisième que nous avons décrite : on fonctionne toujours avec un magasin plein, et aucun outil n'est considéré comme outil résident a priori.

Tang et Denardo (1988), puis Bard (1988) sont les premiers à avoir étudié ce type de problème. Leur première approche a consisté à établir un modèle de programmation linéaire en variables (0-1), mais cette approche s'est révélée décevante à cause de la taille du problème. Crama et al (1991) ont ensuite établi plusieurs résultats de base dont la complexité du problème ; ils ont proposé ainsi que Follonier (1992) plusieurs méthodes de type heuristique.

1. 3. 2. Complexité

Le problème présente deux aspects : d'une part l'ordonnancement des tâches, d'autre part la gestion des entrées/sorties d'outils dans le magasin, (à un instant donné, si un nouvel outil est requis et s'il n'y a plus de place, quel outil doit-on sortir ?). Ces deux aspects sont naturellement liés, ce qui complique le problème. En effet, pour ordonner les pièces, on cherche à déterminer le nombre de changements occasionné par l'enchaînement de chaque couple de pièces. Or comme chaque pièce n'utilise a priori qu'une partie du magasin, le nombre de changements dépend des outils qui composent le reste du magasin ; ces outils dépendent de la stratégie de gestion des entrées/sorties qui est suivie, et de l'ordre des pièces.

Inversement, pour gérer les montages/démontages d'outils entre chaque couple de pièces, on doit observer l'ordre dans lequel les prochaines pièces seront usinées. Nous verrons dans le chapitre 2 que si on considère la séquence des pièces comme une donnée du problème, la gestion optimale des magasins s'effectue en temps polynomial. En revanche, lorsqu'on doit également déterminer l'ordre de passage des pièces, le problème d'ordonnancement avec gestion d'outils que nous avons défini est NP-difficile. Pour le vérifier, on peut effectuer une réduction du problème de l'existence d'une chaîne hamiltonienne dans un graphe cubique, (qui est NP-complet, voir Garey et Johnson (1979)), vers le problème d'ordonnancement avec gestion d'outils pour $C \geq 3$:

Soit G = (V, E) un graphe cubique simple. On pondère par 1 chaque arête du graphe, et on transforme le graphe G en un graphe complet G' en rajoutant toutes les arêtes absentes auxquelles on donne le poids 2. On construit ensuite la matrice A d'incidence arêtes/sommets du graphe G. A possède donc un nombre de lignes M égal à | E |, et un nombre de colonnes N égal à | V |. De plus elle possède exactement 3 "1" par colonne car G est cubique. Cette matrice détermine la matrice d'incidence outils/tâches d'un problème d'ordonnancement avec gestion d'outils de N tâches et M outils, avec une capacité C = 3. Notons que ce problème est particulier, car chaque pièce utilise le même nombre d'outils et sature la capacité du magasin. De plus le nombre de changements entre deux pièces de la matrice est soit 2, soit 3. Il est facile de vérifier que le graphe complété G' correspond au graphe complet dont les sommets sont les pièces, et où la pondération de chaque arête a été augmentée d'une unité.

Supposons qu'on sache résoudre cette instance du problème d'ordonnancement avec gestion d'outils, et soit S la séquence qui minimise le nombre de changements d'outils :

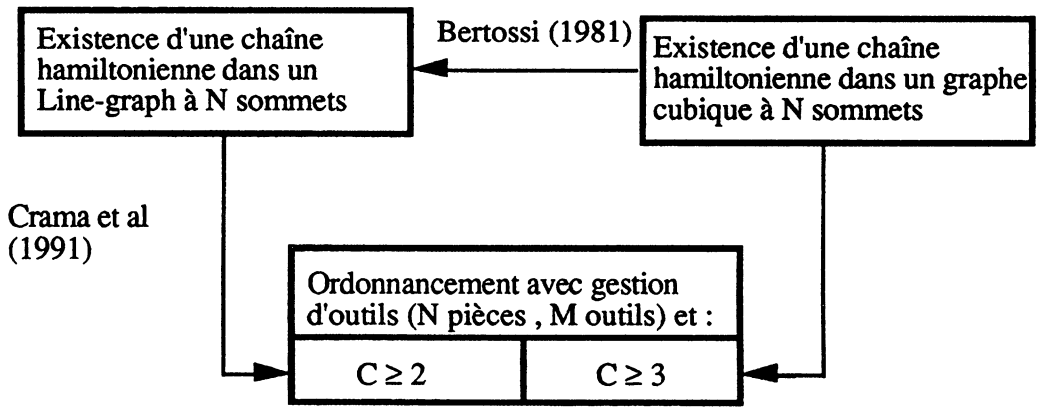
- Si S a un coût égal à 2(N-1), alors S détermine une chaîne hamiltonienne dans G, car chaque sommet est visité une fois, et seules les arêtes de G sont empruntées.

- Sinon, S a un coût strictement supérieur à 2(N-1), donc la chaîne hamiltonienne déterminée par S emprunte au moins une arête parmi celles qui ont été rajoutées pour constituer G'. Il n'existe donc pas de chaîne hamiltonienne dans G.

Cette réduction permet de conclure que le problème d'ordonnancement avec gestion d'outils est NP-difficile pour $C \geq 3$. En effet, on peut généraliser la réduction précédente pour une capacité supérieure à 3, en complétant la matrice A avec un certain nombre de lignes entièrement constituées de "1". Le nombre d'outils requis par chaque pièce reste égal à la capacité qui devient supérieure à 3, et de même le nombre de changements entre deux tâches reste égal à 2 ou 3.

La faiblesse de cette réduction réside dans le fait qu'elle ne permet pas de conclure en ce qui concerne le cas où C = 2, ce qui représente la plus petite capacité possible. On pourrait penser que le problème se trouve largement simplifié du fait qu'on ne dispose que de deux emplacements à gérer dans le magasin, mais en fait il n'en est rien, puisque Crama et al (1991) ont définitivement établi la complexité du problème, en montrant qu'il est NP-difficile même pour C = 2. Leur démonstration consiste à réduire le problème de l'existence d'une chaîne hamiltonienne dans un line-graph au problème d'ordonnancement avec gestion d'outils pour une capacité C $\geq$ 2.

Notons que Bertossi (1981) a établi la complexité du problème d'existence d'une chaîne hamiltonienne dans un Line-graph, en faisant une réduction du même problème d'existence dans un graphe cubique, qui lui est NP-complet. On peut donc finalement situer le problème dans le schéma suivant où chaque flèche indique une réduction polynomiale :



1. 4. Conclusion

Il est intéressant de noter qu'il existe deux autres applications à ce problème d'ordonnancement : la première se situe toujours dans le domaine de la gestion de production, puisqu'il s'agit du montage des composants électroniques sur les circuits imprimés. Bard (1988) a étudié un cas semblable. La seconde, dans le domaine de l'informatique, sera décrite au chapitre 4.

La diversité des questions d'organisation et de décision liées à l'outillage met en évidence la particularité du problème d'ordonnancement que nous avons présenté dans ce chapitre d'introduction. Au travers de l'étude que nous allons en faire, il est donc clair que nous ne prétendons pas résoudre le problème des changements d'outils en général : pour certains ateliers, l'organisation et les moyens disponibles rendent inutiles l'utilisation d'un logiciel de décision et d'optimisation, tandis que pour d'autres la situation et les problèmes de changements d'outils se

révéleront au contraire plus complexes, notamment si l'on considère plus d'une machine. L'étude qui va suivre se décompose en quatre parties :

Comme il est difficile de résoudre le problème globalement, nous nous intéresserons dans un premier temps uniquement à la gestion optimale des magasins d'outils en fonction d'une séquence des pièces donnée : nous présenterons dans le chapitre 2 plusieurs modèles et des algorithmes pour la résolution de ce problème.

Dans le chapitre 3 nous reviendrons au problème d'ordonnancement. Etant donnée sa complexité, le chapitre sera consacré à la présentation de plusieurs types de méthodes heuristiques. A la suite de ce chapitre, nous nous intéresserons à la gestion des magasins lorsque l'ordre des pièces à produire n'est pas connu à l'avance : par exemple, lorsque les pièces sont en cours d'ordonnancement, ou au fur et à mesure de l'arrivée des ordres de fabrication.

Dans cette optique, nous étudierons dans le chapitre 4 le modèle des k-serveurs, ainsi que différentes applications de ce problème, pour pouvoir appliquer ce modèle à la gestion des outils ; c'est l'objet du chapitre 5, qui décrit un modèle et un algorithme permettant de gérer les magasins "on-line" afin de minimiser globalement le nombre de changements d'outils.

1. 5. Bibliographie

J.F. Bard (1988)

"A heuristic for minimizing the number of tool switches on a flexible machine", *IIE Transactions*. vol 20 n°4. pp 382-391.

P. Béranger (1989)

Les nouvelles règles de la production : vers l'excellence industrielle. Collection Dunod Entreprise, Editions Dunod.

M. Berrada (1983)

Prise en compte des outils dans la gestion de production des ateliers flexibles. Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, Thèse n°84. Toulouse.

A.A. Bertossi (1981)

"The edge hamiltonian path is NP-complete", *Information Processing Letters*. vol 13 n°4, 5. PP 157-159.

J. Blazewicz, G. Finke, R. Haupt et G. Schmidt (1988)

"New trends in machine scheduling", *European Journal of operational research*. n°37. pp 303-317.

A.S. Carrie et T.S. Perera (1986)

"Work scheduling in FMS under tool availability constraints", *International Journal of Production Research*. vol 34 n°37. pp 303-317.

N. Carter (1985)

"The application of a flexible tooling system in a flexible manufacturing system", *Robotica*. vol 3 n°37. pp 221-228.

Y. Crama, A.W.J. Kolen, A.G. Oerlemans, et T.C.R. Spieksma (1991)

"Minimizing the number of tool switches on a flexible machine", *Research Memorandum* 91.010, Limburg University. February 1991.
(A paraitre dans *International Journal of Flexible Manufacturing Systems*, 1994 vol 6 n°1)

G.D. Crite, R.I. Mills, et J.J. Talavage (1985)

"PATHSIM, a Modular Simulator for an Automatic Tool Handling System Evaluation in FMS", *Journal of Manufacturing Systems*. vol 4 n°1. pp 15-27.

J. ElGomayel et V. Nader (1983)

"Optimization of machine setup and tooling using the principles of group technology", *Computer and Industrial Engineering*. vol 7 n°3. pp 187-198.

H.A. ElMaraghy (1985)

"Automated Tool Management in Flexible Manufacturing", *Journal of Manufacturing Systems*. vol 4 n°1. pp 1-13.

J.P. Follonier (1992)

"Minimization of the number of tool switches on a flexible manufacturing machine", *report ORWP* 92-32. DMA-EPFL, Lausanne, Switzerland.

L.R. Foulds et J.M. Wilson (1993)

"Formulation and solution of problems of tool positioning on a single machining centre", *International Journal of Production Research*. vol 31 n°10. pp 2479-2485.

M.R. Garey et D.S. Johnson (1979)

Computers and Intractability : A guide to the theory of NP-Completeness. Freeman. New York.

A. Hertz et M. Widmer (1993)

"A new approach for solving the job shop scheduling problem with tooling constraints", *report ORWP* 93-01. DMA-EPFL, Lausanne, Switzerland.

A.S. Kiran et R.J. Krason (1988)

"Automated tooling in a flexible manufacturing system", *Industrial Engineering*. April. pp 52-57.

P. Kouvelis (1991)

"An optimal tool selection procedure for the initial design phase of a flexible manufacturing system", *European Journal of Operational Research*. n°55. pp 201-210.

A. Kusiak (1990)

Intelligent manufacturing systems. Prentice Hall International series in industrial and systems engineering, New Jersey.

A. Kusiak et G. Finke (1987)

"Modeling and solving the flexible forging module scheduling problem", *Engineering Optimization*. vol 12. pp 1-12.

F. Mason (1986)

"Computerized Cutting-Tool Management", *American Machinist & Automated Manufacturing*. May. pp 105-120.

A.G. Oerlemans (1992)

Production Planning for flexible manufacturing systems. Ph.D. Thesis, University of Limburg. Maastricht.

C. Proust, J.N.D. Gupta, et V. Deschamps (1991)

"Flow shop scheduling with set-up, processing and removal times separated", *International Journal of Production Research*. vol 29 n°3. pp 479-493.

S. Shingo (1987)

Le système SMED. Editions d'Organisation.

K.E. Stecke (1989)

"Algorithm for Efficient Planning and Operation of a Particular FMS", *International Journal of Flexible Manufacturing Systems*. vol 1. pp 287-324.

C.S. Tang et E.V. Denardo (1988)

"Models arising from a flexible manufacturing machine, Part I : minimization of the number of tool switches ; Part II : minimization of the number of switching instants", *Operations Research*. vol 36 n°5. pp 767-784.

D. Veeramani, D.M. Upton, et M.M. Barash (1992)

"Cutting-Tool Management in Computer-Integrated Manufacturing", *The International Journal of Flexible Manufacturing Systems*. vol 3/4. pp 237-265.

M. Widmer (1991)

"Job shop scheduling with tooling constraints : a tabu search approach", *Journal of the Operational Research Society*. vol 42 n°1. pp 75-82.

Chapitre 2

Gestion off-line des changements d'outils

Dans ce chapitre, nous simplifions momentanément le problème pour ne traiter que d'un seul aspect : celui de la gestion des magasins d'outils. Nous supposons donc que la séquence des tâches est imposée ou bien que le problème d'ordonnancement a été résolu indépendamment.

Dans la première section, nous décrivons la stratégie de remplacement d'outils optimale pour le cas uniforme, ainsi qu'une des preuves de son optimalité. Nous construisons ensuite dans la deuxième section un modèle plus général pour réduire le problème au calcul d'un flot de valeur maximum et de coût minimum : deux formulations sont proposées qui prennent en compte une pondération non uniforme. Nous utilisons la seconde pour obtenir un autre algorithme optimal pouvant s'appliquer au cas uniforme.

Enfin, en formulant l'algorithme optimal pour le cas uniforme comme un flot de valeur maximum sur le réseau que nous avons construit, nous étudions de nouveau l'optimalité de cet algorithme dans une dernière section.

2. 1. Une stratégie optimale

2. 1. 1. L'algorithme KTNS

La séquence des tâches étant fixée, le problème consiste à déterminer pour chaque tâche quels sont les outils qui doivent être introduits dans le magasin, et surtout quels doivent être les outils démontés pour leur faire place au cas où le magasin serait déjà plein, afin de minimiser le nombre total de changements d'outils sur toute la séquence des tâches. Comme a priori toutes les tâches n'utilisent pas la totalité du magasin, (c'est même rarement le cas), plusieurs choix sont possibles : Pour chaque tâche, les outils correspondants doivent être placés en magasin (s'ils n'y sont pas déjà), mais on peut penser qu'introduire également à cet instant des outils avant qu'ils

ne soient requis, peut diminuer le nombre total de changements. D'autre part, en ce qui concerne les outils à démonter, il semble assez naturel de conserver si c'est possible des outils requis par la tâche qui suit immédiatement dans la séquence celle qu'on s'apprête à usiner.

Ce problème est résolu de façon optimale par l'algorithme KTNS qui a été proposé par Tang et Denardo (1988). L'appellation KTNS provient des initiales des mots anglais "Keep Tools Needed Soonest". Le principe consiste à respecter les deux règles suivantes :
Pour chaque tâche dans la séquence,

1- N'insérer un outil que s'il est requis par la tâche à usiner.

2- Si le magasin est plein, et qu'un outil doit être éliminé, choisir l'outil qui sera requis de nouveau au plus tard dans la séquence.

Cet algorithme est polynomial : il nécessite dans le pire des cas C*M*N opérations où M est le nombre d'outils, N le nombre de pièces à usiner, et C la capacité du magasin de la machine. Si on considère de plus que C est une donnée du problème, faisant partie des caractéristiques techniques de la machine sur laquelle on travaille, alors on peut considérer que KTNS est en O(MN). Nous illustrons son fonctionnement sur l'exemple de référence donné par Tang et Denardo (1988).

Matrice d'incidence outils/tâches (T_{ij}) :

T_{ij} = 1 si l'outil i est requis pour usiner la pièce j
= 0 sinon.

	tâches									
outils	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	1	0	0	1
2	0	0	1	0	0	0	0	0	0	1
3	0	1	0	0	0	1	0	1	0	0
4	1	0	0	0	0	0	0	0	0	1
5	0	1	0	0	0	0	1	1	1	0
6	0	0	1	0	1	0	0	0	0	0
7	0	0	1	1	0	0	1	0	1	0
8	1	0	1	0	0	0	0	1	0	0
9	1	0	0	0	0	0	1	0	0	0

Prenons pour séquence l'ordre croissant obtenu sur la numérotation désignant les tâches. En supposant que la capacité du magasin est 4, l'application de KTNS donne les 10 magasins suivants et détermine 14 changements d'outils :

1	2	3	4	5	6	7	8	9	10
1	1 -	2	2	2	2 -	1 -	3	3 -	1
4 -	3 -	6	6	6 -	3 -	5	5	5 -	2
8	8	8	8	8	8 -	9 -	8	8 -	4
9 -	5 -	7	7	7	7	7	7	7	7

2. 1. 2. Formulation par la programmation linéaire

Tang et Denardo ont établi l'optimalité de la stratégie KTNS en faisant une étude de cas : tout algorithme qui "n'est pas KTNS", c'est-à-dire qui lors de l'insertion ou du retrait d'un outil, viole une des deux règles (ou les deux), peut être modifié localement de façon à suivre le principe KTNS pour un coût inférieur ou égal. L'énumération de tous les cas qui peuvent se présenter est un peu longue. En revanche, Crama et al (1991) ont donné une preuve plus compacte, dans un esprit totalement différent puisqu'elle utilise la programmation linéaire.

L'observation de la matrice d'incidence outils/tâches où les colonnes ont été permutées suivant l'ordre des tâches dans la séquence, permet de distinguer des "0-blocs", c'est-à-dire une succession de "0" sur une même ligne encadrée par un "1", à gauche (en colonne j par exemple), et à droite (colonne j + k) ; concrètement, il s'agit d'un outil qui est entré en magasin comme outil requis, puis qui n'a plus été utilisé par les tâches suivantes, avant d'être de nouveau nécessaire à la fabrication d'une pièce (figure 1).

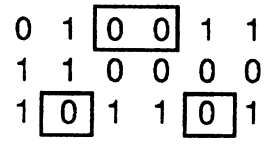


figure 1

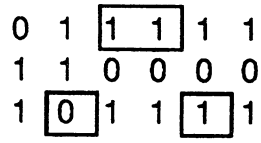


figure 2

Si les colonnes incidentes à ce bloc ne contiennent pas déjà chacune C "1", l'ensemble de ce bloc peut être "basculé à 1", ce qui signifie que chaque "0" du bloc passe à "1", donc que l'outil correspondant à la ligne où se situe le bloc est maintenu en magasin durant la fabrication des k tâches. Grâce à cette observation, gérer les outils off-line revient donc à faire en sorte que chaque colonne contienne C "1", en basculant certains "0-blocs" à 1 (figure 2). Pour décider du choix de ces blocs, on construit la matrice A d'incidence colonnes/blocs, en ordonnant les 0-blocs par extrémité terminale croissante. On calcule également le vecteur V, vecteur des déficits de chaque colonne par rapport à C : soit n_i le nombre de "1" dans la colonne i, on a $V_i = C - n_i$. Le problème se formule alors de la façon suivante :

$$\begin{aligned} &\text{Max} \quad \Sigma X_j \\ &\text{s.t} \quad A X_j \leq V \end{aligned}$$

où X_j = 1 si le bloc j est "basculé" à 1
= 0 sinon

La matrice A a la propriété d'être gloutonne : une définition des matrices gloutonnes consiste à dire que ce sont les matrices qui ne contiennent aucune des deux sous-matrices (3x2) suivantes :

1 1	1 1
0 1	1 0
1 0	0 1

Lorsque la matrice des contraintes est gloutonne, Hoffman, Kolen et Sakarovitch (1985) ont montré que ce problème linéaire en nombres entiers pouvait être résolu par un algorithme glouton polynomial, (justement en O(MN)). D'après Crama et al (1991), lorsqu'on interprète les choix de cet algorithme optimal en termes de gestion d'outils, on retrouve le principe KTNS ce qui prouve l'optimalité de ce principe.

Il est intéressant de noter que bien que le principe KTNS soit extrêmement simple à exprimer comme à mettre en oeuvre, établir son optimalité est par contre assez long et étonnamment complexe en comparaison, que ce soit par une étude de cas, ou par la programmation linéaire. Cette réflexion nous a conduits à rechercher une autre modélisation du problème de la minimisation off-line des changements d'outils, qui permettrait de vérifier d'une autre façon l'optimalité des deux règles de KTNS. A cette occasion, nous avons été amenés à généraliser le problème en prenant en compte une pondération non uniforme des temps de changements d'outils.

2. 2. Une modélisation plus générale avec pondérations

En établissant un modèle plus général pour la gestion des entrées/sorties d'outils, notre but est donc d'une part la recherche d'une autre stratégie optimale pouvant s'appliquer au cas des temps de changement uniformes, et d'autre part d'essayer de se donner la possibilité de prendre en compte des pondérations qui pourraient être liées au chargement de certains outils.

Certains outils sont d'un maniement aisé et rapide, en revanche, les temps de transfert, de montage, de réglages et de mise au point peuvent être beaucoup plus pénalisants pour d'autres. Une pondération peut donc être affectée à chaque outil : d_k sera le coefficient lié au temps de montage de l'outil k dans le magasin de la machine. Nous généralisons le modèle en supposant

qu'en plus des données du problème de gestion off-line des changements d'outils avec temps uniformes, que sont le nombre de tâches et la matrice d'incidence outils/tâches, nous disposons d'une matrice des pondérations (M, M), dont la diagonale est nulle, (dans le cas uniforme nous avions $d_{ij} = 1$ pour tout i différent de j).

L'objectif est de déterminer pour chaque tâche la composition du magasin de la machine, de façon à minimiser la somme des pondérations encourues sur toute la séquence. Naturellement, le fait d'avoir une idée relativement précise des temps de changements nécessités par le montage/démontage fréquent de certains outils modifie la stratégie de gestion des magasins, et l'algorithme KTNS n'est plus forcément adapté.

2. 2. 1. Premier modèle

Nous proposons une première modélisation qui consiste en la construction d'un graphe comprenant N niveaux de sommets, (chaque niveau correspondant à un instant dans la séquence, c'est à dire à une tâche), plus un niveau de départ qui définit la composition initiale du magasin de la machine s'il y a lieu.

Un niveau se compose d'autant de sommets que d'outils requis par l'ensemble des tâches de la séquence, et d'un certain nombre de sommets "supplémentaires" : en effet, sur les M sommets représentant les M outils, ceux qui correspondent aux outils requis à cet instant sont dédoublés et reliés par un arc "horizontal" de coût -K où K est un nombre arbitrairement grand.

Deux niveaux consécutifs sont reliés entre eux par tous les arcs possibles ; ces arcs conservent leur pondération d'origine (donnée par la matrice), et les arcs dits "horizontaux" ont un coût nul. Ainsi, deux niveaux consécutifs constituent un graphe complet biparti.

Toutes les arêtes sont de capacité 1. Sur ce graphe, la gestion off-line des changements d'outils se formule comme un problème de calcul de flot de valeur maximum et de coût minimum. Le premier niveau correspond à l'instant 0 dans la séquence ; chacun de ses sommets correspondant à un outil placé en magasin à l'origine, détient une unité de flot. On suppose donc que ou bien on démarre la fabrication de la séquence des tâches avec un magasin encore rempli par des outils requis pour les dernières opérations effectuées par la machine ; ou bien on commence avec un magasin vide dans lequel on insère d'emblée les C premiers outils requis dans la séquence.

On doit donc faire passer sur le graphe un flot de valeur C à moindre coût. Pour illustrer cette modélisation, on a choisi l'exemple qu'ont donné Crama et al (1991) pour illustrer leur démonstration de l'optimalité du principe KTNS. La matrice d'incidence outils/tâches est la suivante :

0 1 0 0 1 1
1 1 0 0 0 0 associée à la matrice des coefficients (d_{ij}) i = 1, ..., M et j = 1, ..., M
1 0 1 1 0 1

Les colonnes sont ordonnées suivant l'ordre des tâches dans la séquence, et le magasin ne comporte que deux emplacements. On obtient le graphe suivant :

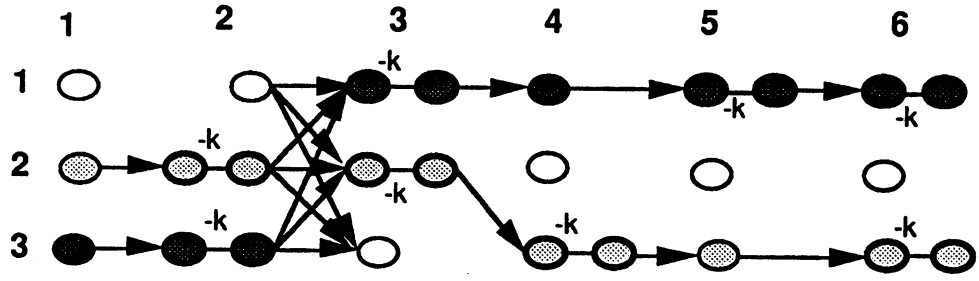


figure 3

Pour ne pas compliquer le schéma, nous n'avons pas fait figurer tous les arcs existant entre les deux niveaux, (sauf entre les tâches n°2 et n°3 dans la séquence), et la 4ième tâche a été supprimée car elle est exactement égale à la 3ième. La construction complète du graphe figure cependant en Annexe 1. Les deux chemins qui figurent sur le schéma définissent les contenus successifs de chaque emplacement du magasin : l'un contiendra l'outil 2, puis le 3 jusqu'à la fin de la séquence ; l'autre contiendra l'outil 3, puis le numéro 1 à partir de la troisième tâche jusqu'à la fin. Leur coût total est $d_{23} + d_{31}$. Les arcs existants entre les sommets dédoublés garantissent que les outils requis par chaque tâche seront bien insérés en magasin. En effet, si tel n'était pas le cas, le coût ne serait pas minimum, car ces arcs ont un poids fortement négatif, et doivent donc être saturés par un flot de coût minimum.

Cette première modélisation permet de calculer en temps polynomial une stratégie optimale de gestion des magasins. Toutefois, le graphe obtenu peut avoir une grande taille : étant donné que chaque tâche ne requiert pas forcément exactement C outils, chaque niveau doit conserver l'ensemble des M outils requis par l'ensemble des tâches, comme points de transit possibles pour chaque unité de flot. Ainsi, si on désigne par b_j le nombre d'outils requis par la $j^{ième}$ tâche dans la séquence, le graphe totalise $\Sigma\ (M + b_j,\ j = 1,\ ...,\ N\ ) + C$ sommets, c'est-à-dire $\Sigma\ (b_j,\ j = 1,\ ...,\ N\ ) + M*N + C$, et un nombre d'arêtes de l'ordre de $(N-1)M^2$. Cette réflexion nous conduit à rechercher une représentation plus compacte du problème. La modélisation suivante est une extension d'un modèle proposé par Chrobak, Karloff, Payne et Vishwanathan (1991).

2. 2. 2. Second modèle

a) Construction du graphe

Le graphe se compose d'une source et d'un puits, et de trois couches de sommets (voir l'illustration sur la figure 4). La première comprend C sommets : elle est destinée à définir la configuration initiale du magasin. Chacun de ses sommets détient une unité de flot et est connecté

à chaque sommet du deuxième niveau. Ce deuxième niveau correspond aux outils requis dans la séquence. Il se compose de sommets r_j : chaque sommet r_j correspond à un outil requis, et est numéroté dans l'ordre de son apparition dans la séquence. Les sommets qui correspondent à des outils requis par une même tâche sont ordonnés de façon arbitraire, suivant l'ordre croissant de leur numéro, (Privault et Finke (1993)). Par exemple, si on reprend l'exemple de Crama et al, (en confondant les tâches 3 et 4, qui sont identiques), on a la succession des tâches suivantes :

tâche n°1 : 2,3
tâche n°2 : 2,1
tâche n°3 : 3
tâche n°4 : 1
tâche n°5 : 3,1

On obtient une séquence d'outils :

$r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8$
(2, 3) (1, 2) (3) (1) (1, 3)

Tous les arcs du réseau ont pour capacité 1. Partant de chaque sommet v appartenant au premier niveau, il existe un arc qui joint chaque sommet r_j et dont le coût est égal au temps de changement entre l'outil r_j et l'outil v (si on commence avec un magasin plein, sinon le coût est nul). Chaque sommet r_j est dédoublé en un sommet r'_j. Ces sommets r'_j constituent le troisième niveau. Chaque r_j est relié à son double r'_j par un arc "horizontal" de coût -K où K est une valeur arbitrairement grande.

Le deuxième niveau de sommets est en fait divisé en blocs qui correspondent aux ensembles d'outils requis par chaque tâche de la séquence. Il existe un arc entre chaque r'_j et tout r_k ne faisant pas partie du même bloc que r_j avec $k > j$. Son poids est d_{jk} (pondération donnée par la matrice).

Ces arcs garantissent d'une part que les outils seront bien insérés dans l'ordre défini par la séquence, c'est-à-dire qu'il seront bien présents en magasin lorsque la tâche qui les requiert sera usinée par la machine, et d'autre part qu'aucun circuit n'existe dans le réseau. De plus, comme aucun arc n'existe entre deux sommets d'un même bloc, une unité de flot ne pourra passer que sur un seul sommet d'un même ensemble à la fois, en empruntant un arc horizontal de coût -K ; les outils faisant partie d'un même ensemble seront donc placés en magasin simultanément pour que la tâche correspondante soit usinée. Enfin, chaque unité de flot traversant un sommet r'_j peut éventuellement terminer sa course directement sur un sommet puits p en empruntant un arc de coût nul.

Sur ce graphe, la gestion off-line des magasins d'outils se réduit comme précédemment au calcul d'un flot de valeur maximum à coût minimum. Le chemin suivi par chaque unité de flot

décrit les chargements successifs d'un emplacement du magasin. Sur notre graphe, le flot maximum a pour valeur C, soit la somme des capacités des arcs issus de la source du réseau. Comme -K est une valeur infiniment petite, un flot de coût minimum doit saturer tous les arcs horizontaux (r_j, r'_j), ce qui implique que tous les outils seront bien montés au moment où ils sont requis. La construction du réseau est illustrée sur l'exemple précédent : comme toujours, afin de ne pas surcharger le graphe, nous n'avons pas dessiné tous les arcs, mais seulement ceux qui sont empruntés par le flot maximum de coût minimum. Cependant, on peut se référer à l'annexe 2 pour la construction complète du graphe.

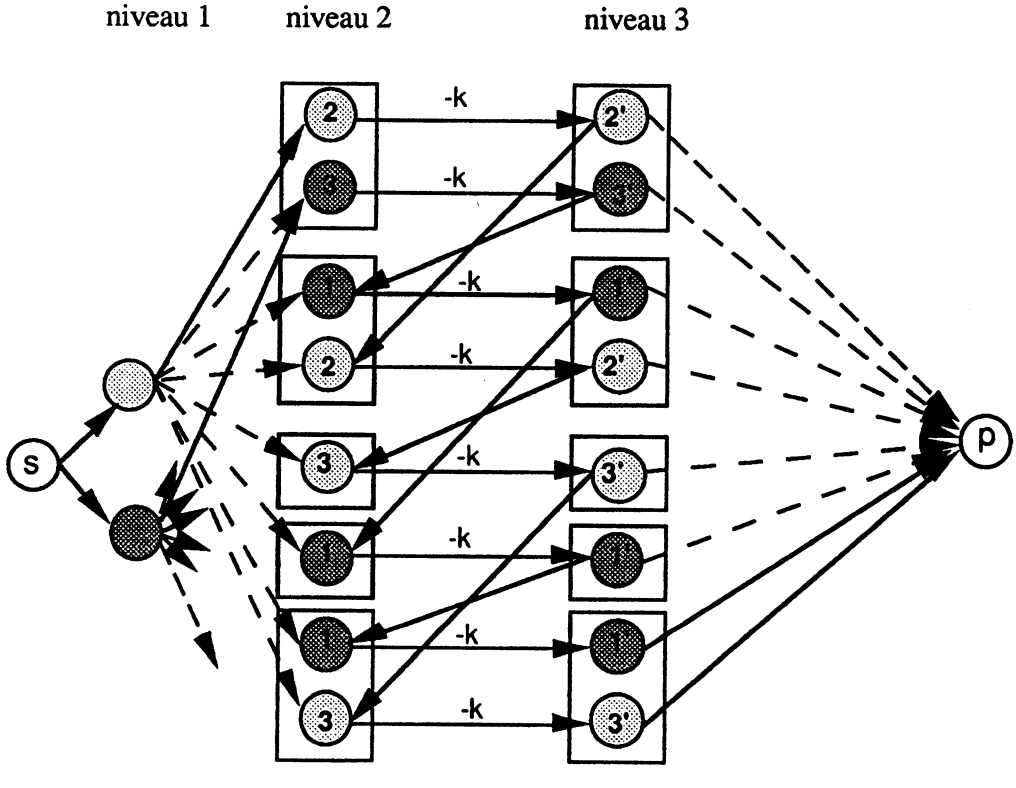


figure 4 : construction du réseau

Nous supposons ici qu'un flot de valeur 2 a été déterminé, et que ce flot atteint un coût minimum en fonction de la matrice des poids relatifs aux outils {1, 2, 3} : sur le réseau, les sommets coloriés en gris clair définissent le chemin emprunté par une unité du flot maximum, et les sommets qui figurent en gris foncé définissent le chemin suivi par l'autre unité du flot. Ce flot peut être visualisé sous la forme classique d'un plan de chargement des magasins successifs :

Tâches:	1	2	3	4	5
outils :	2	2 -	3	3	3
	3 -	1	1	1	1

avec un coût total égal à $d_{31} + d_{23}$

Première remarque : Lorsque les outils requis par la tâche qu'on s'apprête à usiner doivent être placés en magasin, les outils qui sont déjà en place sont naturellement conservés, tandis que les autres sont montés. Cette règle qui semble évidente si on veut réduire le nombre de changements d'outils doit être également suivie par un flot de valeur C et de coût minimum.

Propriété :

Soient f un flot maximum de coût minimum et i un sommet appartenant à deux blocs consécutifs. Supposons que la pondération des arcs vérifie l'inégalité triangulaire. Si la pondération est uniforme alors une unité du flot f traversant le sommet i dans le premier bloc traversera i dans le second bloc. Si la pondération est non uniforme alors f peut être modifié localement de façon à vérifier cette propriété, tout en conservant un coût inférieur ou égal.

Preuve :

Supposons que la propriété ne soit pas vérifiée par un flot f de valeur C et de coût minimum : une unité de flot traversant un sommet (correspondant à un outil numéroté 1 par exemple) dans le premier bloc, passe sur un sommet (soit l'outil numéroté 2), dans l'ensemble suivant, tandis qu'une unité quitte le sommet 3 dans le premier ensemble pour le sommet 1 dans le second bloc, (le sommet 3 peut faire partie du premier bloc ou d'un bloc antérieur dans la séquence). Le coût local sur cette portion du réseau est d12 + d31 (voir figure 5).

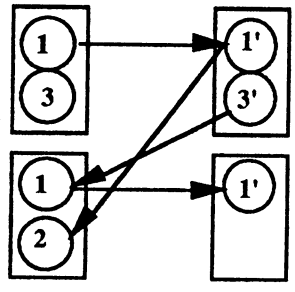


figure 5

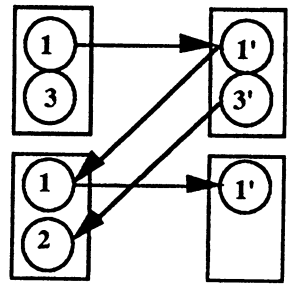


figure 6

Si l'unité de flot traversant le sommet 1 dans le premier ensemble est maintenue sur ce sommet entre les deux blocs (voir figure 6), tandis que l'autre unité passe du sommet 3 au sommet 2 dans le bloc suivant, alors le coût local devient : d32 + d11 = d32 ; ailleurs, le flot n'est pas modifié. Maintenant comme l'inégalité triangulaire est vérifiée, on a $d32 \leq d12 + d31$, et donc le flot modifié a un coût inférieur ou égal à celui de f, (si la pondération est uniforme alors f n'était pas de coût minimum).

Si l'inégalité triangulaire n'est pas satisfaite on peut donc observer dans une stratégie de gestion optimale des outils, des mouvements d'outils qui ne semblent pas "naturels" à première vue, et des insertions prématurées par exemple. Dans la suite, sur tous les exemples que nous étudierons, nous supposerons l'inégalité triangulaire vérifiée.

Deuxième remarque : On notera que cette fois, la construction du graphe nécessite un nombre de sommets égal à $2*\sum (b_j , j = 1, ..., N) + C + 2$, (et dans la majorité des cas un nombre d'arêtes largement inférieur au nombre d'arêtes engendrées par la première construction que nous avons proposée). Comme on a $\sum (b_j , j = 1, ..., N) < N*M$, la seconde modélisation est plus compacte que la première du seul fait qu'on ne fait pas figurer pour chaque tâche l'ensemble des M sommets correspondant à l'ensemble de tous les outils requis par la séquence.

b) Calcul du flot maximum de coût minimum

Comme nous l'avons signalé, le modèle que nous avons décrit est une généralisation d'un modèle proposé par Chrobak, Karloff, Payne et Vishwanathan (1991). Il a été construit pour résoudre un problème a priori sans rapport avec le problème de la gestion d'outils qui est la version off-line du problème des k-serveurs, (nous traiterons en détail de ces problèmes dans les chapitres 4 et 5 ; dans l'immédiat on peut décrire le problème des k-serveurs comme un problème de gestion d'outils dans une séquence où chaque tâche ne nécessiterait qu'un seul outil).

Pour obtenir une gestion optimale des chargements d'outils, nous appliquons la méthode d'augmentations successives à moindre coût ("minimum cost augmentation algorithm"), de Tarjan (1983). La valeur du flot est augmentée par unités successives en empruntant pour chaque unité un chemin de coût minimum dans le graphe résiduel. Rappelons que le graphe résiduel d'un graphe G s'obtient en fonction de la valeur du flot f : Soit (u,v) un arc de capacité cap(u,v) et f(u,v) la valeur du flot qui emprunte cet arc. La capacité résiduelle du flot sur l'arc (u,v) est res(u, v) = cap(u,v) - f(u, v). Le graphe résiduel est constitué des sommets du graphe G et de ses arcs (u, v) de capacité res(u, v) > 0.

Etant donné que le graphe ne contient aucun circuit, donc a fortiori aucun circuit de coût négatif, on commence avec un flot de valeur nulle sur tous les arcs et ce flot est bien de coût minimum. On l'augmente ensuite d'une unité en empruntant un chemin de coût minimum dans le graphe G : le graphe résiduel est toujours sans circuit de coût négatif, et on peut continuer à augmenter la valeur du flot de la même manière dans le graphe résiduel, sans y créer de circuit de coût négatif.

Dans notre cas, les capacités des arcs sont entières et on sait que la valeur maximum du flot est C. A chaque étape, le flot étant augmenté d'une unité et aucun circuit de coût négatif n'étant créé dans le graphe résiduel, l'algorithme nécessitera au plus C augmentations pour calculer le flot. Examinons maintenant la taille du réseau : Le nombre d'outils requis par la tâche j étant toujours désigné par b_j , le réseau a un total de $2*\sum(b_j, j = 1, \ldots, N) + C + 2$ sommets. De plus, pour chaque j on a $b_j \leq C$, donc le nombre de sommets est inférieur ou égal à $2C*N + C + 2$. On peut donc conclure que le calcul d'un flot de valeur maximum à coût

minimum nécessitera au plus C^3*N^2 opérations. Ce résultat peut certainement être amélioré par l'utilisation d'un algorithme performant pour le calcul des chemins de coût minimum successifs (voir Tarjan (1983)), mais il montre que le calcul de la stratégie optimale de gestion des magasins peut être calculée en temps polynomial.

Considérons maintenant l'intérêt que peut présenter ce modèle général et l'algorithme de résolution associé pour la gestion off-line des changements d'outils en temps uniformes. On suppose donc que toutes les pondérations non nulles du réseau sont égales à 1, (sauf pour les arcs horizontaux qui ont toujours un coût -K), et on dispose d'une autre méthode de gestion off-line que KTNS, qui est cette fois en $O(N^2)$, tandis que l'algorithme KTNS est en O(MN). Etant donné que le nombre d'outils requis par l'ensemble de toutes les tâches à usiner dans la séquence est la plupart du temps largement supérieur au nombre de tâches, on a $M > N$ et donc la méthode qui consiste à déterminer les magasins d'outils en vue de minimiser le nombre de changements par le calcul du flot max de coût min peut être considérée comme compétitive par rapport à KTNS.

Cependant, ce n'est pas son unique intérêt : dans la section suivante, nous allons utiliser ce modèle non plus pour déterminer une solution, mais pour montrer l'optimalité du principe KTNS.

2. 3. Formulation du principe KTNS comme un flot max de coût min

2. 3. 1. Correspondance entre un plan de chargement et un flot maximum

Un plan de chargement se définit de la façon suivante : étant donnée une séquence de N tâches, on donne dans un tableau, à C lignes et N colonnes, la composition du magasin au début de chaque nouvelle tâche, c'est-à-dire le numéro des C outils présents en magasin. Chaque ligne du tableau correspond donc à une case du magasin à outils, et indique les outils qui vont s'y succéder. Le numéro d'un outil ne peut figurer au plus qu'une fois dans chaque colonne du tableau car on ne dispose que d'un unique exemplaire de chaque outil. On associe à une ligne une unité de flot. Les outils qui se succèdent dans la case correspondante indiquent le chemin suivi par le flot sur le graphe :

- Lorsqu'un outil entre puis sort pour faire place à un autre à l'instant suivant, l'unité de flot emprunte un arc de coût +1 entre deux blocs successifs.

- Lorsqu'un outil reste durant plusieurs instants successifs dans le magasin, l'unité de flot correspondante emprunte un arc qui sort sur le puits p, ou qui dépasse successivement plusieurs blocs ; si l'outil reste jusqu'à réutilisation, l'arc de dépassement est de coût nul, et dans le cas contraire, l'arc emprunté est de coût +1.

Le plan de chargement définit ainsi C chemins disjoints portant chacun une unité de flot. La somme des coûts des C chemins est égale au nombre de changements d'outils occasionnés par le plan de chargement. Tout plan de chargement obtenu par l'algorithme KTNS peut donc être mis sous la forme d'un flot de valeur maximum sur le réseau dont les coûts sont 0 ou 1. Il reste donc à examiner le coût de ce flot issu de la stratégie KTNS.

2. 3. 2. Optimalité du flot et validité du modèle proposé

Dans notre cas où tous les arcs sont de capacité 1, le graphe résiduel pour un flot f contient les arcs non saturés par le flot (avec leur pondération d'origine c'est-à-dire 0 ou 1), les arcs empruntés par le flot figurant en sens inverse avec un coût opposé, (c'est-à-dire 0 ou -1). Une caractérisation bien connue du flot de coût minimum est la suivante :

Théorème :
Un flot f est de coût minimum si et seulement si son graphe résiduel ne contient pas de circuit de coût négatif.

Cette équivalence permet de vérifier la validité du modèle proposé ce qui revient à poser la question : tout flot détermine-t-il un plan de chargement ? En effet, sur le graphe, certaines configurations du flot, (qui ne correspondent pas à un plan de chargement), sont impossibles dès lors que le flot est de coût minimum. Examinons dans quels cas le flot obtenu ne correspond pas à un plan de chargement des magasins :

a) A un instant dans la séquence, tous les outils requis par une tâche ne sont pas présents en magasin :

Il y a deux interprétations possibles : ou bien le flot n'emprunte pas certains sommets du deuxième niveau, (et donc du troisième), et dans ce cas, il n'est pas maximum (car chaque plan de chargement est réalisable étant donné que aucune tâche ne requiert plus de C outils) ; ou bien certains arcs horizontaux de coût -K ne sont pas saturés, et dans ce cas, il n'est pas minimum, car il existe toujours dans le réseau C chemins disjoints tels que tous les arcs horizontaux soient empruntés.

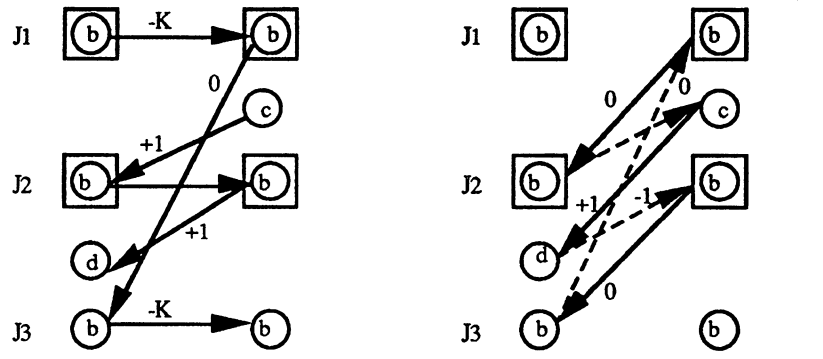
b) D'après le flot, un même emplacement du magasin serait utilisé par deux outils différents au même instant :

Ce cas est impossible, car tous les arcs ont comme capacité 1 : comme chaque chemin définit les contenus successifs d'un emplacement de magasin, et que les C chemins qui constituent le flot sont disjoints, chaque emplacement n'est occupé à chaque instant que par un unique outil.

c) Le flot détermine la présence de deux outils identiques occupant au même instant chacun un emplacement du magasin:

Ce scénario correspond aux configurations suivantes du flot, et il implique que le flot n'est pas de coût minimum :

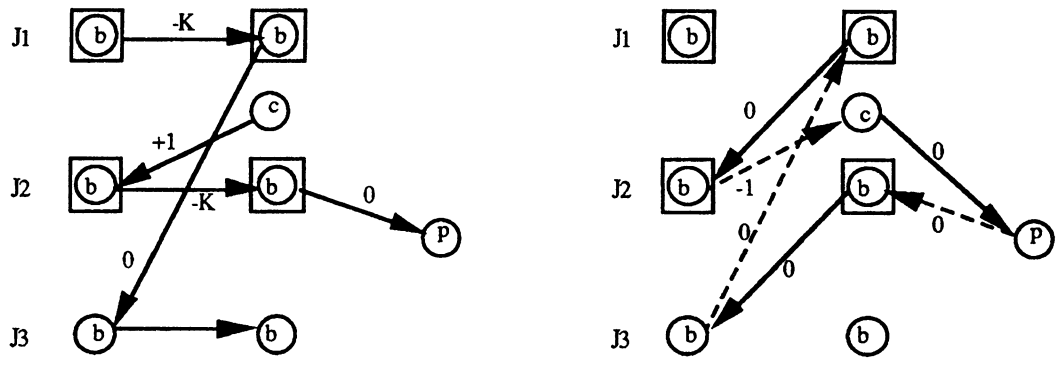
Première configuration :



Flot dans le graphe — Cycle dans le graphe résiduel

Ce flot ne détermine pas un plan de chargement (on a deux outils b pour l'exécution de la tâche J2), mais on peut identifier un circuit de coût négatif dans le graphe résiduel. De même, la configuration suivante ne peut se produire si le flot est de coût minimum.

Deuxième configuration :



Flot dans le graphe — Cycle dans le graphe résiduel

Dans ces deux configurations, le flot ne peut correspondre à un plan de chargement car il signifierait la présence de deux outils identiques en magasin à l'instant J2, alors que chaque outil

n'est disponible qu'en un unique exemplaire. En fait, il est aisé de vérifier que le flot n'est pas de coût minimum, en identifiant dans chaque exemple un circuit de coût -1 passant par le sommet d pour le premier cas, et par le puits p pour le deuxième.

De même, si le sommet b dans le bloc J3 est remplacé par un sommet différent, l'arc qui va du bloc J1 au bloc J3 est cette fois de coût +1, et on identifie le même circuit de coût négatif que l'on soit dans la première ou dans la deuxième configuration. Donc aucun flot de coût minimum ne peut déterminer la présence de deux outils identiques au même instant en magasin.

Conclusion:

Tous les flots de valeur maximum obtenus sur le réseau ne correspondent pas forcément à un plan de chargement. Seuls les flots max de coût minimum déterminent des plans de chargement, et donc constituent une solution pour le problème de la minimisation du nombre de changements d'outils.

Dans le paragraphe suivant, on étudie les conditions d'existence d'un circuit de coût négatif dans le graphe résiduel d'un flot pour le cas uniforme, lorsque le flot est issu d'un plan de chargement .

2. 3. 3. Conditions d'existence d'un circuit de coût négatif

On remarque d'emblée qu'un circuit négatif ne peut emprunter des arcs "retour" horizontaux, car ceux-ci ont un coût positif, arbitrairement grand. Le circuit doit donc emprunter des paires d'arcs constituées successivement d'un arc "retour" de coût 0 ou -1, puis d'un arc "descente", (c'est-à-dire qui n'a pas été emprunté par le flot), de coût 0 ou +1. Les seules paires d'arcs donnant un coût total négatif sont celles constituées d'un arc retour de coût -1, puis d'un arc de descente de coût 0. Pour qu'un circuit ait un coût total négatif, il doit donc emprunter un certain nombre de ces paires d'arcs de coût négatif, c'est-à-dire plus que de paires d'arcs de coût total positif.

Remarques :

Etant donné que le flot qu'on considère est issu d'un plan de chargement, le graphe résiduel correspondant ne peut pas contenir des paires d'arcs successifs ayant chacun un coût nul, (on serait alors dans le cas des configurations impossibles 1 et 2, décrites précédemment).

De plus, tout circuit contenant une telle paire d'arcs peut être modifié localement sans que son coût soit changé en introduisant une corde de coût +1 qui supprime la paire d'arcs nuls ainsi que l'arc de coût +1 qui la précède (voir figure 7).

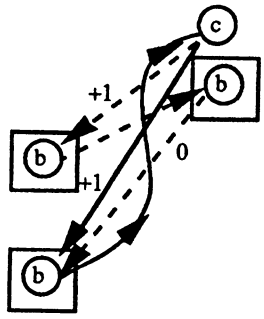


figure 7

Cette remarque nous amène à considérer dans la suite uniquement des circuits exempts de paires d'arcs successifs de coûts nuls. On définit de la façon suivante les autres paires possibles :

Définitions :

On appelle *paire négative* un arc "retour" de coût -1 suivi d'un arc "descente" de coût 0.
On appelle *paire positive* un arc "retour" de coût 0 suivi d'un arc "descente" de coût +1.
On appelle *paire nulle* un arc "retour" de coût -1 suivi d'un arc "descente" de coût +1.

Pour qu'un circuit dans le graphe résiduel ait un coût négatif, il faut qu'il contienne plus de paires négatives que de paires positives, (le nombre de paires nulles n'a naturellement pas d'importance).

2. 3. 4. Optimalité du flot issu d'un plan de chargement KTNS

On s'intéresse maintenant à un flot correspondant à un plan de chargement qui suit la règle KTNS, (tel que nous l'avons défini dans la première section de ce paragraphe), et on étudie les caractéristiques d'un circuit négatif issu d'un tel flot. Pour cela on définit la propriété suivante :

Propriété Pn : Soit un circuit Cn dans le graphe résiduel du flot obtenu à partir d'un plan de chargement KTNS, et qui contient n paires négatives : alors il contient au moins n paires positives.

On se propose de montrer cette proposition par récurrence sur n le nombre de paires d'arcs négatifs contenues dans un circuit. Montrons d'abord la propriété P1.

Propriété P1 : Soit un circuit C1 dans le graphe résiduel contenant une unique paire négative : alors C1 contient au moins une paire positive.

Preuve :

Soit C_1 un circuit ne contenant qu'une seule paire négative. Soit (**a**, **b**) l'arc "retour" de coût -1, du bloc j au bloc i, et (**b**, **b**) l'arc de coût 0 qui joint le bloc i au bloc k. Supposons que le chemin qui ferme le circuit, donc qui va du sommet **b** (dans le bloc k), à **a** (dans le bloc j) ne contienne aucune paire positive. Ce chemin est donc constitué uniquement de paires nulles (voir figure 8).

Soit (c_2, c_1) l'arc de coût -1 dans ce chemin, qui dépasse le bloc j, c'est-à-dire le bloc où se trouve le sommet **a**, (le sommet c_1 est distinct du sommet **b** sinon on aurait deux outils identiques en magasin). Comme KTNS est respecté à chaque bloc et en particulier au bloc j, si l'outil **b** sort pour faire place à l'outil **a**, tandis que c_1 reste en magasin, c'est que c_1 est redemandé dans un bloc qui se trouve avant le bloc où **b** est rappelé pour la première fois, donc avant le bloc k. De ce fait, c_2 se trouve entre **a** et **b** (sinon on aurait deux outils c_1 en magasin). De même, soit (c_4, c_3) l'arc de coût -1 qui précède (c_2, c_1) dans le chemin. Par KTNS, c_3 est redemandé dans un bloc qui se trouve avant le bloc où c_1 est rappelé, donc également avant le bloc où **b** est rappelé ; ainsi, c_4 se trouve avant le bloc k.

En appliquant le même raisonnement, on constate qu'aucun chemin de coût nul qui atteint le sommet **a** ne peut provenir d'un bloc d'un niveau inférieur ou égal à celui du sommet **b** dans le bloc k. Un tel chemin doit donc obligatoirement emprunter au moins un arc "retour" de coût 0, dont le sommet d'origine doit se situer entre le **premier** bloc où **b** est rappelé, et celui où se trouve **a**.

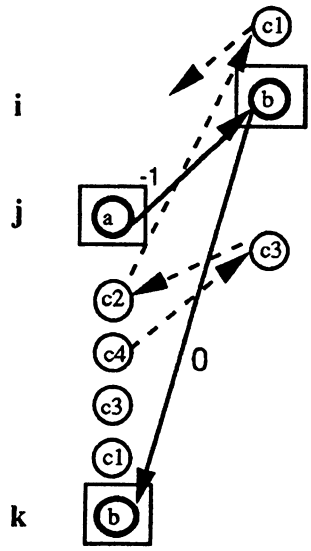


figure 8

Par conséquent, la proposition P_1 est vraie : le circuit C_1 contient au moins une paire positive. Supposons que tout circuit contenant n paires négatives contienne au moins n paires positives.

Propriété P_{n+1} : Soit C_{n+1} un circuit dans le graphe résiduel contenant n+1 paires négatives. Soit S un chemin dans C_{n+1} contenant exactement une paire négative, (c'est-à-dire un arc **(a,b)** de coût -1 et un arc **(b,b)** de coût 0), éventuellement des paires nulles, et une paire positive, c'est-à-dire un arc de coût nul dont l'origine est un sommet **c1** se trouvant dans un bloc situé entre **a** et **b**. Un tel chemin existe, sinon la propriété P_1 n'est pas vérifiée (figure 9).

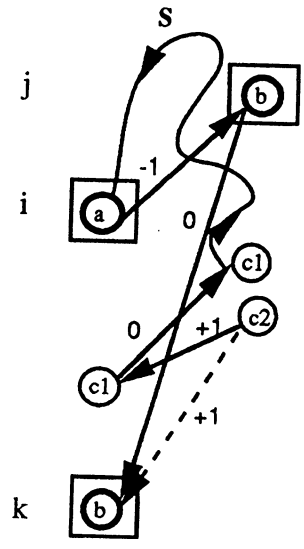


figure 9

Soit **(c2,c1)** l'arc de coût +1 qui précède l'arc **(c1,c1)** dans S : S joint **c2** à **a**. **c2** se trouve au dessus du sommet **c1**, donc avant le premier rappel du sommet **b** après le bloc j, (par la propriété P1). Par conséquent, le sommet **c2** est différent du sommet **b**, et il existe dans le graphe résiduel un arc de coût +1 qui joint **c2** au sommet **b** dans le bloc k.

Soit C_n le circuit obtenu à partir de C_{n+1} grâce à cette corde **(c2,b)**. C_n contient exactement n paires négatives, car on a soustrait de C_{n+1} une seule paire négative (voir figure 10). Le nouveau circuit contient donc au moins n paires positives. Par conséquent, C_{n+1} contient au moins n+1 paires positives car le chemin S contient au moins une paire positive.

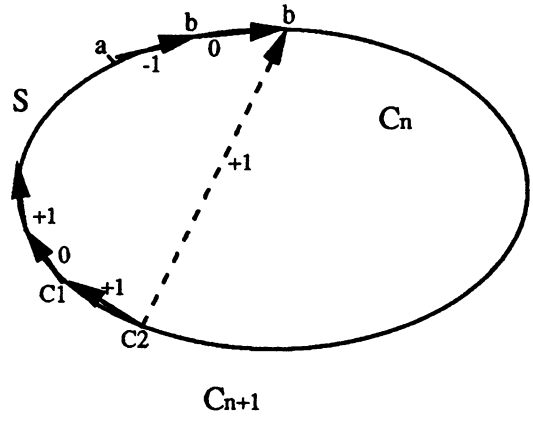


figure 10

La propriété Pn étant vraie pour tout n, il n'existe aucun circuit de coût négatif dans le graphe résiduel lorsque le flot respecte le principe KTNS. On peut donc en déduire la proposition suivante :

Proposition :

Tout flot de valeur maximum respectant le principe "Keep Tools Needed Soonest" est de coût minimum ; la stratégie de gestion off-line des outils KTNS est donc optimale.

2. 4. Conclusion

Le modèle que nous avons construit permet donc d'établir l'optimalité de la stratégie KTNS d'une façon différente de celles employées par Tang et Denardo (1988), et Crama et al (1991). Ceci dit, comparée à la simplicité d'expression des deux règles de KTNS, la démonstration même si elle n'est pas difficile, apparaît toujours longue. On peut aussi remarquer que la stratégie KTNS est en fait plus restrictive par rapport aux plans de chargement obtenus comme flots maximum de coût minimum. En effet, un plan de chargement peut être optimal tout en violant à certains instants l'une ou l'autre des règles de KTNS. De même, si dans la séquence un renouvellement complet du magasin est imposé à un instant donné pour le passage d'une pièce particulière, tous les critères de sortie peuvent être équivalents lors d'un instant précédant le renouvellement du magasin.

Une autre caractéristique appréciable du modèle est sa plus grande généralité : en effet, lorsque les temps de changements ne sont plus égaux pour tous les outils, ni l'algorithme KTNS, ni l'algorithme glouton de Hoffman et al (1985) employé pour résoudre le programme linéaire proposé par Crama et al (1991) ne sont adaptés. On ne disposait donc pas de modèle

approprié à l'exception de celui proposé par Oerlemans (1992) qui est le seul à notre connaissance : il consiste à reprendre le modèle linéaire construit pour le cas uniforme en introduisant cette fois des coûts dans la fonction objective, et en observant que la matrice des contraintes est totalement unimodulaire : on peut donc en relaxant la contrainte d'intégrité sur la variable (X_j) résoudre le problème en temps polynomial puisque le vecteur second membre V est lui-même entier.

Cependant si l'on revient après ces détours théoriques à des considérations d'ordre plus pratique, il faut souligner que la tendance chez les professionnels et les constructeurs de machines, est naturellement à la recherche d'une réduction des temps de changements, ce qui passe par leur uniformisation grâce à l'utilisation de fixations interchangeables par exemple, ou de dispositifs de déplacement automatisés : des progrès appréciables ont été réalisés dans ce domaine, ce qui tend à modifier la nature des problèmes d'outillages que l'on peut rencontrer. Ainsi, des disparités dans les temps de changements ne seront observables que dans le cas de fabrications très spécifiques pour lesquelles malgré les progrès technologiques, la manipulation de certains outils reste pénalisante (par exemple à cause de leurs poids, ce qui peut être le cas pour des presses).

Notons pour finir, que la gestion d'outils n'est pas la seule application de ce modèle : nous verrons dans le chapitre 5 qu'il présente un intérêt pour un problème a priori sans rapport qui est celui des k-serveurs avec service par blocs.

2. 5. Bibliographie

M. Chrobak, H. Karloff, T. Payne et S. Vishwanathan (1991)
"New results on server problems", *SIAM Journal on Discrete Mathematics* vol 4 n°2. pp 172-181.

Y. Crama, A.W.J. Kolen, A.G. Oerlemans, et T.C.R. Spieksma (1991)
"Minimizing the number of tool switches on a flexible machine", *Research Memorandum* 91.010, Limburg University. February 1991.
(A paraître dans *International Journal of Flexible Manufacturing Systems*, 1994 vol 6 n°1)

A.J. Hoffman, A.W.J. Kolen et M. Sakarovitch (1985)
"Totally balanced and greedy matrices", *SIAM Journal on Algebraic and Discrete Methods* vol 6 n°4. pp 721-730.

A.G.Oerlemans, (1992)
Production Planning for Flexible manufacturing systems, Ph.D. Thesis, University of Limburg, Maastricht. pp 81-106.

C. Privault et G. Finke (1993)
"Tool Management on NC-machines", *Proceedings of the International Conference on Industrial Engineering and Production Management IEPM'93*, june. Mons, Belgium. vol 2. pp 667-676.

C.S. Tang et E.V. Denardo (1988)
"Models arising from a flexible manufacturing machine, Part I : minimization of the number of tool switches", *Operations Research* vol 36 n°5. pp 767-777.

R.E. Tarjan (1983)
"Data structures and Network Algorithms", *BMS-NSF Regional Conference Series in Applied Mathematics* Vol 44. pp 109-111.

Chapitre 3

Heuristiques pour l'ordonnancement avec gestion d'outils

On considère maintenant que la séquence des tâches n'est plus fixée, et qu'on doit déterminer un ordre de passage des pièces sur la machine, qui minimise le nombre total de changements d'outils. Ce problème étant NP-difficile, la recherche d'heuristiques performantes est donc essentielle, et fait l'objet de ce troisième chapitre.

Nous présentons d'abord les principaux types de méthodes heuristiques qui ont été proposées pour résoudre le problème puis nous décrivons celles que nous avons élaborées. Nous discutons ensuite des performances comparées des types d'heuristiques décrits, puis nous expliquons et commentons les différents tests numériques que nous avons effectués pour évaluer nos heuristiques.

3. 1. Introduction

Nous avons vu dans le premier chapitre que le problème dans son ensemble présente deux aspects interdépendants qui sont :

- L'ordonnancement des pièces
- La gestion des entrées/sorties d'outils

Si le premier aspect est résolu, le second se résout facilement avec l'algorithme KTNS que nous avons étudié dans le chapitre précédent. On a donc tendance pour simplifier à séparer les deux problèmes, et c'est la raison pour laquelle la plupart des heuristiques sont composées de deux phases distinctes : d'abord création d'une séquence des tâches à partir de la matrice d'incidence outils/pièces, puis gestion optimale des changements d'outils induits par la séquence

grâce à la procédure KTNS. On peut ranger dans la même catégorie les méthodes d'amélioration qui suivent ce principe. Plus rares sont les méthodes qui gèrent les outils et construisent la séquence au fur et à mesure, sans doute parce qu'elles ont tendance à être plus complexes et plus coûteuses en temps de calcul. Les méthodes inspirées du modèle du voyageur de commerce appartiennent à la première catégorie : elles sont axées sur la recherche d'une séquence des pièces.

3. 2. Heuristiques du problème du voyageur de commerce

Pour déterminer un ordonnancement, il est assez courant d'avoir recours à une modélisation sous forme de problème de voyageur de commerce : on construit un graphe complet G = (V, E, Lb), où V est l'ensemble des sommets représentés par les pièces, et Lb(i,j) désigne la pondération des arêtes (i,j).

Une plus courte chaîne hamiltonienne sur le graphe G détermine un ordre de passage des pièces dont le coût est minimum si le poids des arêtes (i, j) représente le nombre de changements d'outils à effectuer pour l'enchaînement de la tâche i et de la tâche j. Or en dehors du cas où on a $|T_i| = |T_j| = C$ pour tout ensemble T_j (respectivement T_i) d'outils requis par la pièce j (respectivement par la pièce i), le nombre de changements entre deux tâches i et j dépend de la séquence. En conséquence, les arêtes de G ne peuvent être pondérées que par une approximation du nombre de changements entre chaque paire de tâches. La seule approximation dont on dispose est une borne inférieure qui a été proposée par Tang et Denardo (1988) :

$$Lb(i,j) = \text{Max}\,[0, |T_i| + |T_j| - |T_i \cap T_j| - C]$$

Tang et Denardo (1988) ont d'ailleurs expérimenté les premiers l'utilisation du modèle du voyageur de commerce en construisant une heuristique d'amélioration sur la base de la méthode *Shortest Edge*. Crama et al (1991) ont fait une étude des quatre heuristiques les plus connues pour le voyageur de commerce (voir Golden et Stewart (1985)) :

- *Shortest Edge* : Complexité : $O(N^2 \log N)$
- *Nearest Neighbour* : avec tous les sommets comme point de départ. Complexité : $O(N^3)$
- *Farthest Insertion* : avec tous les sommets comme départs possibles. Complexité : $O(N^4)$
- *Branch and Bound* : algorithme de Volgenant et Jonker (1982). Complexité : exponentielle

Pour chacune de ces méthodes, la séquence obtenue est évaluée par la procédure KTNS. D'après les tests effectués par Crama et al (1991), c'est l'heuristique *Farthest Insertion* qui donne les meilleurs résultats, particulièrement lorsque la taille des instances augmente. De plus la

méthode est très rapide (de l'ordre de 30 secondes sur les plus grandes instances testées, soit 60 outils et 40 pièces, sur un pc AT 80286, 12Mhz avec coprocesseur 80287).

3. 3. Heuristiques inspirées du problème du voyageur de commerce

Crama et al (1991) ont également proposé des méthodes inspirées du modèle du voyageur de commerce ou qui utilisent une des quatre heuristiques que nous avons citées :

3. 3. 1. Méthode 2-OPT

Il s'agit de la méthode de permutation classique qui consiste à trouver deux tâches dans la séquence dont l'échange améliore le coût, et à continuer sur la séquence obtenue. Cette méthode est utilisée telle qu'elle à partir d'une séquence générée aléatoirement ou comme procédure d'amélioration à partir d'une "bonne" séquence obtenue grâce à une autre méthode heuristique.

L'étape de recherche du couple de pièces à permuter nécessite au plus N^2 applications de la procédure KTNS, mais on ne sait pas si le nombre de permutations est polynomial ou non. Cette procédure peut donc s'avérer très coûteuse en temps. Nous avons utilisé un 2-OPT restreint qui consiste à n'intervertir que des pièces consécutives dans la séquence (voir Privault (1990)), et nous avons appliqué cette procédure d'amélioration à la fin d'une de nos heuristiques. Crama et al (1991) ont également testé une méthode de 2-OPT restreint.

3. 3. 2. Block Minimization

On construit le graphe D = (V, U, Ub), où V contient toutes les pièces à usiner plus un sommet fictif 0. L'ensemble U contient toutes les paires d'arêtes orientées entre les sommets ; ces arcs sont pondérés cette fois par une borne supérieure Ub du nombre de changements entre chaque couple de pièces i et j : $Ub(i,j) = | T_i \setminus T_j |$. Cette pondération est non symétrique. Chaque chemin hamiltonien sur le graphe D qui termine sur le sommet 0, définit une séquence de pièces. L'heuristique *Farthest Insertion* est utilisée pour calculer un chemin hamiltonien sur D, ce qui donne une heuristique en $O(N^4)$.

Remarque : Le nom de cette heuristique fait référence aux "0–blocks" définis par Crama et al (1991) pour établir l'optimalité de la procédure KTNS (voir chapitre 2). Il s'agit ici plutôt de "1–blocks" (succession de "1" sur une même ligne). Le nombre de "1–blocks" contenus dans la matrice d'incidence outils/tâches constitue une borne supérieure du nombre de changements : trouver un plus court chemin hamiltonien sur le graphe D revient à minimiser le nombre de "1–blocks".

Notons que dans le même ordre d'idée, Crama et al ont proposé une autre méthode qui est fondée sur la notion de "matrice intervalle" : une matrice intervalle est une matrice dont les colonnes peuvent être permutées de façon à ce que chaque ligne ne contienne qu'un unique

"1–blocks" (voir Nemhauser et Wolsey (1988)). Si une matrice d'incidence outils/tâche est une matrice intervalle, alors on peut déterminer une permutation des colonnes de la matrice d'incidence A, c'est-à-dire une séquence des pièces dont le nombre de changements est égal à M donc optimal. La méthode consiste donc à déterminer dans A la plus grande sous matrice intervalle possible, et la permutation correspondante qui peut être déterminée en O(MN).

3. 3. 3. Load and Optimize

Cette méthode est basée sur le fait que lorsque la matrice d'incidence du problème comporte exactement C "1" par colonne, l'approximation du nombre des changements Lb est exacte. On part donc d'une séquence s sur laquelle on applique KTNS. La matrice obtenue répond au critère que nous avons décrit ci-dessus. On utilise alors l'heuristique *Farthest Insertion* pour trouver une meilleure séquence à partir de s et de la matrice définie par KTNS. Crama et al (1991) ont mis en oeuvre cette méthode en admettant comme séquences "amélioratrices" les séquences s' dont le coût est identique à celui de s, et en arrêtant la procédure après 10 itérations sans diminution du nombre de changements.

3. 4. Méthodes gloutonnes

Les méthodes gloutonnes qui ont été proposées sont fondées sur le principe de l'heuristique *Nearest Neighbour*. La séquence est construite pièce par pièce : pour ajouter une nouvelle pièce à la fin de la séquence partielle s, on teste chaque pièce j qui n'est pas encore ordonnée en évaluant par la procédure KTNS le coût de chaque séquence (s, j) : la pièce dont l'ajout donne le plus petit nombre de changements d'outils est placée en fin de séquence s, et ainsi de suite jusqu'à ce que les N pièces soient ordonnées. Cette méthode fait appel N^2 fois à la procédure KTNS, elle est donc en $O(MN^3)$. Le choix de la première pièce dans la séquence n'est pas fixé. On peut donc par exemple commencer par une pièce qui utilise beaucoup d'outils ; on peut aussi essayer à leur tour toutes les tâches comme début de séquence (cette version "Multi-Start est alors en $O(MN^4)$. Crama et al (1991) ont proposé et testé ces deux versions. Dans la section 6, nous décrivons également une méthode gloutonne.

3. 5. Méthode TABOU

Une technique de type TABOU a été appliquée par Follonier (1992) à la recherche d'un ordonnancement qui minimise le nombre de changements d'outils. Rappelons brièvement que les méthodes TABOU ont été initiées par Glover ; leur principe essentiel consiste à partir d'une

solution admissible S, pour choisir une solution S* voisine de S, de telle sorte que d'étape en étape, on s'achemine vers une solution dont on espère qu'elle minimise une fonction objective donnée f(S), (voir Glover (1989) et Glover (1990)).

Le nom de la méthode est dû à la construction d'une liste de mouvements (de la solution S à une solution voisine S*) qui sont interdits (ou "tabous") lors d'une étape. Cette liste de longueur limitée est mise à jour à chaque étape. L'étape de base consiste à se diriger d'une solution S vers une solution S*, S* étant la meilleure solution dans un voisinage N(S), (elle peut cependant être plus mauvaise que S), et S→ S* ne constituant pas un mouvement Tabou ; la liste Tabou est alors mise à jour en fonction de ce déplacement.

Pour le problème d'ordonnancement avec gestion d'outils, une solution est une séquence S des N pièces, et un voisinage de S est l'ensemble des séquences qui peuvent être obtenues à partir de S, en déplaçant une pièce vers une autre position dans la séquence S.

La longueur de la liste a été fixée à 4 arbitrairement, et le temps d'exécution total limité au temps d'exécution de la méthode gloutonne de Crama et al (1991) (version Multi-Start) sur la même instance. La séquence initiale est soit générée aléatoirement, soit calculée grâce à une autre heuristique comme la méthode de meilleure insertion, (voir Follonier (1992)).

3. 6. Trois autres heuristiques

3. 6. 1. Stratégie de Regroupement

Cette méthode est la première que nous ayons proposée, (voir Privault 1990). Elle a depuis évolué dans sa forme mais son principe de base reste fondé sur l'observation de l'approximation du nombre de changements d'outils entre chaque paire de tâches (i, j) : pour des tâches i et j qui nécessitent peu d'outils, la borne Lb ne constitue pas une bonne approximation ; d'où l'idée de réunir les outils requis par plusieurs tâches, de façon à ce que ces outils remplissent presque totalement le magasin. On forme ainsi ce qu'on appelle une "super tâche". Les critères de réunion de deux tâches sont fondés sur un indice de ressemblance entre jobs, et l'approximation du nombre de changements Lb entre eux deux. Cette phase de regroupement est répétée sur les "super tâches" obtenues et les tâches restantes, jusqu'à ce que aucun regroupement ne soit possible sans dépasser la capacité du magasin.

Lorsque toutes les tâches obtenues remplissent presque totalement (voire totalement) le magasin, l'approximation Lb tend à devenir exacte. Toutes les super tâches sont alors ordonnées en appliquant l'heuristique *Shortest Edge* au graphe complet des tâches dont les arêtes sont pondérées par Lb. A l'intérieur d'un même groupe, les tâches sont placées arbitrairement dans l'ordre où elles ont été ajoutées au groupe.

Après application de KTNS sur les N tâches de la séquence obtenue, une procédure d'amélioration par permutations de tâches voisines est utilisée, afin d'améliorer éventuellement l'ordre arbitraire dans lequel les tâches ont été placées à l'intérieur d'un même groupe. La méthode peut se résumer de la façon suivante :

Calculer Lb(i,j)
Sim(i,j) = (| Ti ∩ Tj |)/Min(| Ti |, | Tj |) pour tout i ≠ j dans J = {1, ..., N}
Tant que ∃ (i,j) tel que Lb(i,j) = 0 **Faire**
    Choisir (j1, j2) tel que Lb(j1, j2) = 0 et Sim (j1, j2) = Max {Sim(i, j), i ≠ j, i,j ∈ J}
    Tj1 ← Tj1 ∪ Tj2
    J ← J \ {j2}
    Calculer Lb(j1, ∗) et Sim (j1, ∗)
Fin (Tant que)
Calculer une séquence S par **Shortest Edge** sur le graphe complet G = (J, Lb).
Tant que ∃ (jk, jk+1) consécutifs dans S dont l'échange améliore **KTNS(S) Faire**
    Permuter (jk, jk+1) dans S.

Remarque : Pour construire la séquence finale, c'est d'abord l'heuristique *Shortest Edge* qui a été choisie dans le même ordre d'idée que l'algorithme : grouper en priorité les tâches qui se ressemblent, autrement dit celles qui ont des outils communs. Mais ce choix ne semble pas déterminant : en effet, l'heuristique *Farthest Insertion* par exemple donne des résultats similaires pour la construction de la séquence finale. L'heuristique de regroupement est en $O(MN^2)$.

3. 6. 2. Meilleure Insertion[1]

Les pièces sont classées sur une liste de priorité en fonction de plusieurs critères (nombre d'outils utilisés, fréquence d'utilisation de ces outils, ...). Les deux premières tâches de la liste constituent dans un ordre arbitraire la séquence de départ, et la troisième est placée successivement devant les deux tâches, entre les deux, après les deux tâches. A chaque fois, on évalue le coût local de l'insertion de la pièce (par KTNS), et on la place là où le coût est moindre. On teste ensuite les quatre positions possibles pour la quatrième tâche sur la liste, et ainsi de suite jusqu'à ce que les N pièces soient ordonnées. La complexité de cette heuristique est $O(MN^3)$. Notons que parallèlement Follonier (1992) a également expérimenté une méthode semblable en testant d'autres listes de priorité.

On peut formuler le principe algorithmiquement en définissant Sk(i) comme la séquence S partielle de longueur l dans laquelle la pièce ji a été intercalée entre la $k^{ième}$ pièce de la séquence et la $(k+1)^{ième}$, et on a : S0(i) = {ji, S} et Sl(i) = {S, ji}.

[1] Cette méthode ainsi que la suivante ont été présentées au congrès AFCET/ASRO, (Paris 1991)

Ordonner l'ensemble J = {1, ..., N}
S ← {j1, j2}
Pour i := 3 à N **Faire**
    Choisir la meilleure séquence S* parmi les séquences S0(i), S1(i), ..., Si(i)
    S ← S*
Fin (pour)

L'heuristique suivante pourrait être classée dans une catégorie différente de celle à laquelle appartiennent toutes les méthodes que nous avons décrites jusqu'ici, dans la mesure où c'est la seule à notre connaissance qui tente de gérer efficacement le contenu de chaque magasin d'outils en même temps qu'elle crée la séquence. Les méthodes précédentes fonctionnaient en deux temps distincts : la création de la séquence, puis la gestion off-line des outils.

3. 6. 3. Next Best

La séquence est construite suivant le principe *Nearest Neighbour* de l'heuristique du voyageur de commerce : Lorsque la séquence est partiellement constituée, on choisit comme prochaine tâche celle qui réalise le plus fort "coefficient" de similarité avec le contenu du dernier magasin de la séquence. Soit j cette tâche. Le coefficient de similarité utilisé est le rapport entre le nombre d'outils communs et le nombre d'outils utilisés par la pièce candidate. La gestion des outils du magasin se fait ensuite suivant deux critères :

1- introduire les outils de la pièce j en cours uniquement.
2- si la place manque pour introduire un outil, sortir l'outil (non requis) dont le nombre d'utilisations sur l'ensemble des jobs restant à produire est le plus faible.

Chaque pièce est placée ainsi en fin de séquence jusqu'à ce que les N tâches soient ordonnées. Finalement, la séquence complète est évaluée par la procédure KTNS. Le principe peut être formulé comme suit : soit R l'ensemble des outils contenus dans le dernier magasin de la séquence partielle S.

S = { j0 } où j0 est la pièce déterminée suivant le critère de choix du début de séquence
J ← J \ {j0}
Tant que J ≠ ∅ **Faire**
    Choisir la pièce j dans J qui maximise $|R \cap T_j| / |T_j|$
    S ← {S, j}
    J ← J \ {j}
    Charger chaque outil de T_j qui n'est pas dans R en sortant un outil selon le critère k
Fin (Tant que)

La principale difficulté présentée par ce type d'heuristique réside dans le fait que les outils doivent être gérés à chaque instant sans que l'on sache quelles seront les pièces qui seront usinées juste après, (voir Privault et Finke (1993)). Il est donc difficile d'établir un bon critère de sortie des outils, c'est pourquoi nous en avons testé trois :

. **Critère 1** : sortir les outils les plus utilisés par les pièces non ordonnées
. **Critère 2** : sortir les outils les moins utilisés par les pièces non ordonnées
. **Critère 3** : sortir aléatoirement des outils non utilisés par la tâche courante

De même, on doit décider du choix de la première pièce de la séquence. Ce choix peut être fait suivant plusieurs critères comme le nombre d'outils utilisés ou la fréquence d'utilisation des outils requis, mais toutes les pièces peuvent aussi être placées à leur tour en début de séquence, ce qui donne deux versions de l'algorithme :

. Version "Multi-Start" ou "multiple" : chaque pièce est testée comme début de séquence. La meilleure séquence parmi les N obtenues est retenue. Complexité $O(MN^3)$.
. Version "Single-Start" ou "simple" : la tâche nécessitant le plus d'outils est placée en début de séquence. Complexité $O(MN^2)$.

3. 7. Comparaisons de performances et résultats numériques

3. 7. 1. Performances des méthodes

Les méthodes appartenant aux quatre grands types que nous avons décrits ont été évaluées par leurs auteurs. En ce qui concerne le dernier type, les tests qui ont été réalisés ne permettent que partiellement de se faire une idée des performances de la méthode TABOU. Notons que dans notre cas, la méthode telle qu'elle a été utilisée s'apparente dans sa phase de recherche d'un voisinage de la solution courante, à une autre méthode amélioratrice qui est la méthode *2-OPT*, méthode qui fonctionne par ailleurs plutôt bien. D'autre part, quand on "amorce" la méthode TABOU avec une solution initiale calculée par la méthode de *Meilleure Insertion*, la solution obtenue lorsque la recherche est stoppée est forcément meilleure que celle de départ, mais elle n'est pas significativement meilleure que celle donnée d'emblée par l'heuristique *Meilleure Insertion* : sur la plus grande instance testée (60 tâches et 90 outils), le plus grand écart enregistré entre les deux méthodes est observé lorsque *Meilleure Insertion* se situe en moyenne à 10,2% du meilleur score tandis qu'après amélioration par TABOU la solution est à 3,8% pour un temps d'exécution largement supérieur, voir Follonier (1992).

Il serait intéressant de comparer TABOU plutôt aux autres méthodes "amélioratrices" telles que le Recuit Simulé, ou des algorithmes génétiques appliqués au problème de la gestion d'outils ; mais l'emploi de telles méthodes sous-entend une démarche et des exigences particulières qui consistent à rechercher une solution de grande qualité sans restrictions sur les temps de calcul.

Pour ce qui est des trois autres groupes d'heuristiques, les tests effectués par Crama et al (1991) permettent de mettre en évidence deux heuristiques qui fonctionnent particulièrement bien ; ce sont celles qui nécessitent les plus longs temps de calcul : il s'agit de la *Méthode Gloutonne*, (avec tous les départs de séquence possibles), et de la méthode *2-OPT*, (employée à partir d'une séquence générée aléatoirement) : 1 heure pour la première et 30 minutes pour la seconde sur une matrice de 40 tâches, et de 60 outils, (sur PC AT 80386, 12 Mhz avec coprocesseur 80287).

Si on désire obtenir une solution de bonne qualité rapidement l'heuristique *Block Minimization*, (implantée à partir de la méthode *Farthest Insertion*), ou la version simple de la *Méthode Gloutonne* sont tout à fait appropriées (1 à 3 secondes sur une matrice (40, 60)).

Notons enfin que *Farthest Insertion* est aussi rapide que les deux précédentes, et qu'elle permet d'obtenir des résultats comparables, mais ces résultats dépendent fortement de la qualité de l'approximation Lb sur laquelle repose toute l'efficacité de la méthode : la méthode fonctionne bien, lorsque les matrices d'incidence sont denses.

Il est intéressant de comparer ces méthodes aux trois heuristiques que nous avons proposées. Pour tester ces méthodes dans des conditions comparables, l'idéal serait de disposer du code des auteurs, afin de comparer réellement les versions des différentes méthodes qui ont été programmées, telles qu'elles ont été implantées.

Cependant, on peut remarquer que nous n'avons proposé que des méthodes constructives et non amélioratrices, dans un esprit d'économie des temps de calcul. Une comparaison avec la méthode *2-OPT* ne serait donc pas significative. D'autre part, on peut également remarquer que parmi les 4 méthodes restantes, les méthodes gloutonnes s'apparentent à la méthode *Next Best* que nous avons décrite, tandis que les deux autres fonctionnent suivant le principe *Farthest Insertion*. Nous avons donc entrepris deux séries de tests : la première pour déterminer parmi nos trois méthodes laquelle se comporte le mieux ; la seconde pour comparer la meilleure et la plus rapide de ces méthodes à l'heuristique *Farthest Insertion*. Dans la section suivante, nous décrivons la façon dont les jeux d'essais ont été générés, et nous indiquons quels paramètres on a choisi de faire varier.

3. 7. 2. Génération aléatoire de Matrices d'incidence outils/tâches

Nous avons défini une instance du problème d'ordonnancement avec gestion d'outils par les paramètres N, M, C et un intervalle $[\alpha, \beta]$, où α désigne le plus petit pourcentage d'utilisation de la capacité du magasin par les pièces, et β le plus grand pourcentage d'occupation du

magasin : l'intervalle [30%, 60%] par exemple, signifie qu'on ne doit générer aléatoirement pour une pièce que des ensembles d'outils qui remplissent entre 30% et 60 % du magasin. Ce paramètre a été ajouté afin de faire varier la densité des matrices générées, et d'évaluer la robustesse des heuristiques du type voyageur de commerce qui utilisent la borne Lb, mais comme nous le verrons, d'autres méthodes sont également sensibles à ce facteur.

Les matrices (MxN) sont générées colonne par colonne, de la façon suivante : pour chaque tâche j, on initialise un vecteur Tj à 0, et on génère le nombre d'outils requis n_j. Cet entier est généré aléatoirement uniformément sur l'intervalle [α*C, β*C]. On génère ensuite n_j numéros de lignes différents (c'est-à-dire n_j outils), tirés aléatoirement uniformément sur l'intervalle [1, M]. On obtient ainsi un vecteur Tj en (0,1) qu'on compare aux autres vecteurs déjà générés. S'il existe un vecteur Ti, tel que Ti $\subset$ Tj ou Tj $\subset$ Ti, alors le vecteur Tj n'est pas stocké avec les autres. En effet, si on a Tj $\subset$ Ti par exemple, l'ensemble de ses outils est un sous-ensemble des outils utilisés par la tâche i. On peut donc exécuter la tâche j juste après la tâche i sans provoquer de changements d'outils. En conséquence, la pièce j n'a pas d'intérêt puisqu'elle peut être supprimée sans modification du nombre de changements d'outils. On ne génère donc que des pièces "différentes", c'est-à-dire que le nombre de colonnes de la matrice ne peut pas être réduit par inclusions. On génère ainsi N vecteurs qui respectent l'intervalle de densité [α, β].

Notons pour finir, qu'il peut arriver qu'une fois la matrice générée, certains outils ne soient utilisés par aucune pièce de la matrice, ce qui signifie que certaines lignes sont entièrement nulles. Si le nombre de ces lignes reste limité (moins de 1%), on conserve la matrice telle qu'elle est, sinon on génère une autre matrice. Cependant, nos expériences numériques montrent que dans la pratique, ce cas se produit rarement.

3. 7. 3. Première série de tests

Les heuristiques ont été implantées sur station Sun (Sparc2). Pour chaque matrice (M, N), on teste deux capacités C1 et C2, et pour chaque triplet (M, N, C1), et (M, N, C2) on teste deux intervalles de densité [α, β] = [30%, 60%] et [α, β] = [50%, 100%]. Après chaque exécution sur un type d'instance I, on a retenu l'heuristique donnant le meilleur coût (soit Min(I) le coût minimum obtenu sur l'instance I), et calculé l'écart en pourcentage Δh(I) entre ce coût et le score de chacune des autres heuristiques : soit Ch(I) le score de l'heuristique h sur l'instance I

$$\Delta h(I) = \left(\frac{Ch(I)-Min(I)}{Min(I)}\right)*100$$

Les chiffres indiqués correspondent à une moyenne de ces pourcentages Δh(I) pour chaque heuristique sur 10 instances différentes d'un même type I de problème. De même, les temps d'exécutions indiqués sont des moyennes pour chaque heuristique sur 10 exécutions. Au total, 16 types de problèmes ont été testés, (ce qui correspond donc à 160 instances), et les

résultats numériques sont rassemblés dans les quatre tableaux suivants. Dans ces tableaux figurent les 5 heuristiques qui ont été testées :

La méthode de Regroupement : l'heuristique est suivie de la phase d'amélioration par application d'un 2-OPT restreint.

La méthode Meilleure Insertion : pour constituer la liste de priorité, on calcule la borne Lb entre chaque paire de tâches ; les tâches j qui dans le meilleur des cas enregistrent le plus grand nombre de changements Lb(j, .) sont placées en premier sur la liste. Ce classement coïncide le plus souvent avec le classement obtenu par nombre d'outils requis décroissant. En effet, il est plus facile d'insérer dans une séquence déjà partiellement constituée une pièce dont la fabrication requiert peu d'outils, plutôt qu'une tâche mobilisant la totalité du magasin par exemple.

La méthode Next Best : pour cette méthode nous testerons trois variantes suivant le critère de sortie d'outil qui est utilisé pour chaque magasin, afin de discerner si l'une des stratégies de sortie est mieux adaptée que les autres. D'autre part, nous utiliserons à chaque fois la version simple de l'heuristique, car les séquences obtenues par les deux méthodes précédentes ne sont pas le résultat de méthodes à départs multiples.

Séquence aléatoire : comme dernier moyen de comparaison, nous générons sur chaque instance une permutation aléatoire des N tâches que nous évaluons par la procédure KTNS.

N = 10 M = 20

Capacité	C1 = 7		C2 = 13	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
Stratégie de Regroupement	8,13	5,15	1,42	6,41
Next Best (critère 1)	12,05	12,1	0	15,43
Next Best (critère 2)	8,21	11,98	0	7,53
Next Best (critère 3)	9,29	14,69	0	12,23
Meilleure Insertion	2,95	4,34	8,57	1,95
Séquence Aléatoire	40,72	34,78	32,85	52,26
Temps d'exécution (secondes)				
Stratégie de Regroupement	0,1	0,2	0,2	0,2
Next Best (critère 1)	2,4	2,5	2,4	3,6
Next Best (critère 2)	2,4	2,4	2,3	3,6
Next Best (critère 3)	2,9	2,5	2,1	3,6
Meilleure Insertion	0,6	0,7	0,7	0,7
Séquence Aléatoire	-	-	-	-

tableau 1

N = 20 M = 50

Capacité	C1 = 15		C2 = 26	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
Stratégie de Regroupement	1,91	7,72	1,71	5,89
Next Best (critère 1)	6,39	10 ,72	6,06	10,03
Next Best (critère 2)	6,51	10,69	9,04	12,67
Next Best (critère 3)	7,78	10,25	13,92	11,66
Meilleure Insertion	4,14	0,12	6,47	0
Séquence Aléatoire	27,15	30,97	31,79	37,13
Temps d'exécution (secondes)				
Stratégie de Regroupement	0,6	0,8	0,6	1
Next Best (critère 1)	12,7	14,1	18	27,8
Next Best (critère 2)	12,6	17,4	17 ,6	27,2
Next Best (critère 3)	12,6	17,6	17 ,8	27,5
Meilleure Insertion	8,2	9	8,5	10,2
Séquence Aléatoire	- -	- -		

tableau 2

N = 40 M = 60

Capacité	C1 = 20		C2 = 35	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
Stratégie de Regroupement	2,64	6,53	5,25	6,36
Next Best (critère 1)	11,53	7,95	14,36	12,13
Next Best (critère 2)	13,26	10,85	16,16	14,95
Next Best (critère 3)	12,39	10,25	18,49	14,87
Meilleure Insertion	1,83	0	0,23	0
Séquence Aléatoire	42,28	36,54	40,46	40,93
Temps d'exécution (secondes)				
Stratégie de Regroupement	1,9	2,8	2,1	3,4
Next Best (critère 1)	55,2	84,6	87	140,6
Next Best (critère 2)	54,4	83,5	86	140,5
Next Best (critère 3)	54,6	84,3	86,6	140,7
Meilleure Insertion	65,2	82,5	65,6	87,3
Séquence Aléatoire	- -	- -		

tableau 3

N = 50 M = 100

Capacité	C1 = 30		C2 = 60	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
Stratégie de Regroupement	2,67	6,3	5,66	6,92
Next Best (critère 1)	14,31	9,84	16	12,2
Next Best (critère 2)	13,56	11,15	14,77	13,3
Next Best (critère 3)	14,37	10,10	16,6	12,27
Meilleure Insertion	1	0	0	0
Séquence Aléatoire	31,9	29,35	36,74	35,23
Temps d'exécution (minutes)				
Stratégie de Regroupement	0,07	0,17	0,09	0,15
Next Best (critère 1)	2,15	3,27	3,88	6,33
Next Best (critère 2)	2,12	3,3	3,85	6,32
Next Best (critère 3)	1,7	3,31	3,86	6,34
Meilleure Insertion	4,04	5,8	4,1	6,35
Séquence Aléatoire	-	-	-	-

tableau 4

Les résultats numériques font apparaître le bon comportement de la méthode *Meilleure Insertion* par rapport aux deux autres. Viennent ensuite la *Stratégie de Regroupement*, puis les méthodes *Next Best*. Pour ces dernières, on constate qu'aucune tendance significative ne se dessine entre les trois critères de sortie testés. On retiendra donc le critère qui paraît le plus "logique", (c'est-à-dire la sortie des outils les moins utilisés), mais le fait qu'un choix aléatoire des outils à démonter donne des résultats comparables laisse penser que ce critère est peu fiable, et qu'il serait intéressant de rechercher d'autres façons de gérer les magasins au cours de la construction de la séquence , afin d'améliorer les résultats obtenus par cette méthode.

Si on compare les scores obtenus sur les intervalles [30%, 60%] d'une part, et [50%, 100%] d'autre part, on observe pour toutes les heuristiques des variations dans la qualité des résultats obtenus. Sur le premier intervalle, on a tendance à obtenir de moins bons résultats que sur le second, excepté pour la *Stratégie de Regroupement* pour laquelle c'est le phénomène inverse qui se produit. Ces variations s'expliquent par le fait que les intervalles déterminent la densité des matrices d'incidence : si les matrices sont denses, l'approximation Lb tend à se rapprocher de la valeur exacte du nombre de changements entre les pièces, et dans le même temps, la gestion des outils devient plus facile en ce sens que la composition de chaque magasin est pratiquement déterminée par chaque pièce.

Inversement, si chaque tâche requiert un grand nombre d'outils, voire la totalité des emplacements du magasin, la *Stratégie de Regroupement* s'avère moins efficace car dans le meilleur des cas, on ne peut regrouper les tâches que deux à deux, et on passe donc rapidement à la phase d'ordonnancement sur le modèle du problème du voyageur de commerce.

Notons, toujours pour la méthode de regroupement, que la phase d'amélioration par 2–OPT restreint n'apporte pas beaucoup de modifications aux séquences qui ont été construites : en effet, on observe au plus 2 ou 3 permutations de pièces voisines, mais le plus souvent aucune permutation n'est effectuée, ce qui permet de dire que l'ordre dans lequel les tâches sont placées à l'intérieur d'un groupe n'a pas d'influence sur la qualité de l'ordonnancement final.

Considérons maintenant les temps d'exécution : il faut préciser que ces temps ne sont donnés qu'à titre indicatif, car d'une part ce sont des temps moyens, et d'autre part les heuristiques n'ont pas toujours été programmées dans un esprit d'optimisation des temps de calculs. Cependant, malgré ces imprécisions, les tableaux des temps font ressortir des écarts très nets. La *Stratégie de Regroupement* est sans conteste la plus rapide, (de l'ordre de quelques secondes sur les plus grandes instances), tandis que les temps des autres heuristiques sont largement supérieurs, (plus de 6 minutes en moyenne sur les matrices (50, 100) les plus denses).

Conclusion : si on désire obtenir une solution de qualité sans restrictions sur les temps de calcul, on peut avoir recours à la méthode *Meilleure Insertion*. Si on désire obtenir une bonne solution très rapidement, on aura plutôt recours à la *Stratégie de Regroupement* : en effet, tout en offrant l'avantage d'être largement plus rapide, elle se situe dans le pire des cas à 6 % en moyenne des résultats obtenus par la méthode de *Meilleure Insertion*.

3. 7. 4. Deuxième série de tests

Nous comparons maintenant l'heuristique que nous avons sélectionnée à l'heuristique du voyageur de commerce qui semble le mieux fonctionner, c'est-à-dire *Farthest Insertion*. La version de cette heuristique que nous avons programmée consiste à utiliser l'approximation Lb, et à prendre comme chaîne initiale la chaîne constituée des deux sommets du graphe les plus éloignés. Lors de cette deuxième série de tests, il nous a semblé intéressant d'observer également si la version avec départs multiples de l'heuristique *Next Best* permet d'améliorer sensiblement les résultats de cette méthode, (jusqu'ici plutôt décevante), en conservant des temps de calcul raisonnables. Les trois plus grands types d'instances ont été testés : (20, 50), (40, 60), et (50, 100).

N = 20 M = 50

Capacité Taille des jobs	C1 = 15 [30%,60%]	[50%,100%]	C2 = 26 [30%,60%]	[50%,100%]
Next Best (multiple)	1,9	1,9	4,3	3,2
Stratégie de Regroupement	6,4	7,9	6,3	8,3
Farthest Insertion	23,4	1,1	26,9	1
Séquence Aléatoire	32,5	34,2	36,6	34,8
Temps d'exécution (secondes)				
Next Best (multiple)	165	230	229,1	346,8
Stratégie de Regroupement	0,4	0,5	0,5	0,6
Farthest Insertion	0,6	0,7	0,6	0,7
Séquence Aléatoire	-	-	-	-

tableau 5

N = 40 M = 60

Capacité Taille des jobs	C1= 20 [30%,60%]	[50%,100%]	C2 = 35 [30%,60%]	[50%,100%]
Next Best (multiple)	6	2,5	6,8	4
Stratégie de Regroupement	5,2	6,3	7,7	5,8
Farthest Insertion	37	1,8	42,6	1,3
Séquence Aléatoire	40,8	40,2	49,7	43
Temps d'exécution (secondes)				
Next Best (multiple)	1365	2100	2130	3404
Stratégie de Regroupement	1	1,5	1,3	2
Farthest Insertion	4	4,5	3,9	5
Séquence Aléatoire	-	-	-	-

tableau 6

N = 50 M = 100

Capacité	C1= 30		C2 = 60	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
Next Best (multiple)	6,1	2,6	6,1	3,13
Stratégie de Regroupement	6,2	5,1	3,7	3,8
Farthest Insertion	27,3	0,6	34	1
Séquence Aléatoire	32,9	27,7	37,6	29,3
Temps d'exécution (secondes)				
Next Best (multiple)	3952,6	5992,2	7069	11571,6
Stratégie de Regroupement	2,7	3,5	3,1	5,8
Farthest Insertion	11,1	12,2	11,2	14,3
Séquence Aléatoire	0,5	0,5	0,5	0,6

tableau 7

Nos résultats numériques confirment les tests de Crama et al (1991) : l'heuristique *Farthest Insertion* fonctionne très bien sur les matrices d'incidence très denses, et mieux que la méthode de Regroupement, tandis que sur les intervalles [30%, 60%] les scores sont nettement moins bons : sur les 10 matrices (40, 60, 20) par exemple, *Farthest Insertion* est à 37% en moyenne du meilleur score, ce qui est pratiquement le pourcentage obtenu par les séquences générées aléatoirement (40,8%).

Ce phénomène s'explique une fois de plus par le fait que lorsque le nombre d'outils par pièce est faible comparé à la capacité du magasin de la machine, l'approximation Lb est trop optimiste. Lb est pratiquement nulle pour tous les couples de pièces, donc la recherche d'une plus courte chaîne hamiltonienne par *Farthest Insertion* équivaut à la génération aléatoire d'une séquence. En comparaison, la qualité des résultats obtenus par la *Stratégie de Regroupement* est constante quel que soit le type de l'instance.

La version à départs multiples de l'heuristique *Next Best* permet d'améliorer sensiblement les résultats de cette méthode qui devient presque aussi efficace que la *Stratégie de Regroupement*, mais si on prend en compte les temps d'exécution, le temps atteint par *Next Best* est de l'ordre de 35 minutes sur des matrices (40, 60), et il devient prohibitif pour les matrices (50, 100) les plus denses (plus de 3 heures !).

3. 8. Conclusion

Pour conclure ce chapitre sur les méthodes heuristiques, on peut d'abord remarquer que "l'éventail" des méthodes dont on peut disposer pour l'ordonnancement des pièces est très large, et parmi elles, ce ne sont pas toujours les plus sophistiquées qui produisent les meilleurs résultats.

On pourrait encore multiplier les tests et faire varier différemment les paramètres, cependant les expériences numériques qui ont déjà été réalisées permettent de préconiser pour l'obtention rapide d'une bonne solution l'emploi de la méthode *Farthest Insertion* lorsque les pièces utilisent beaucoup d'outils, et dans le cas contraire, l'emploi de la *Stratégie de Regroupement.* Si le temps de calcul de la solution n'entre pas en ligne de compte, c'est la méthode *Meilleure Insertion* qui garantit les meilleurs résultats.

Comme nous l'avons signalé au début de ce chapitre, ces deux méthodes et quasiment toutes celles qui ont été testées sont constituées de deux phases indépendantes : la création de la séquence, puis la gestion optimale des outils. La seule méthode qui fasse intervenir ces deux phases simultanément et que nous ayons testée est la méthode *Next Best.* Comparés aux résultats de autres méthodes, ceux de cette heuristique sont plutôt décevants. Pourtant, la caractéristique principale de la méthode qui est une gestion des magasins d'outils à chaque instant est intéressante car elle revient à essayer d'imiter autant que possible la gestion off-line optimale des changements d'outils, sans connaissance de l'ordre dans lequel les outils seront requis par les prochaines pièces usinées sur la machine. Cette situation peut d'ailleurs se produire indépendamment de tout problème d'ordonnancement, or les critères que nous avons testés ne semblent pas efficaces.

C'est l'étude de ce problème qui fait l'objet des deux chapitre suivants : pour une instance d'un problème d'optimisation donné, à chaque instant et sans connaissance du futur, comment prendre une ou plusieurs décisions dont la qualité est conditionnée par le futur déroulement de l'instance ? Cet aspect caractérise les *problèmes de k–serveurs* que nous présentons dans le chapitre 4. Après un tour d'horizon sur ces problèmes et leurs différentes applications, nous verrons dans un dernier chapitre comment les outils de chaque magasin peuvent être gérés sur ce modèle.

3. 9. Bibliographie

Y. Crama, A.W.J. Kolen, A.G. Oerlemans, et T.C.R. Spieksma (1991)
"Minimizing the number of tool switches on a flexible machine", *Research Memorandum* 91.010, Limburg University. February 1991.
(A paraître dans *International Journal of Flexible Manufacturing Systems* 1994, vol 6 n°1)

J.P. Follonier (1992)
"Minimization of the number of tool switches on a flexible manufacturing machine", *ORWP* 92-32. DMA-EPFL, Lausanne, Suisse.

F. Glover (1989)
"Tabu search, Part I", *ORSA Journal on Computing* vol 1. pp 190-206.

F. Glover (1990)
"Tabu search, Part II", *ORSA Journal on Computing* vol 2. pp 4-32.

B.L. Golden et W.R.Stewart (1985)
Empirical analysis of heuristics. In *The Traveling Salesman Problem*, E.L. Lawler et al (eds). John Wiley & Sons, Chicester. pp 207-249.

G.L. Nemhauser et L.A. Wolsey (1988)
Integer and Combinatorial Optimization. John Wiley & Sons, New York.

C. Privault (1990)
"Gestion des changements d'outils sur machines à commande numérique", *Mémoire de DEA de Recherche Opérationnelle*. Laboratoire ARTEMIS-IMAG, Grenoble I.

C. Privault et G. Finke (1993)
"Une extension des problèmes de serveurs appliquée à la gestion des outils dans un système flexible de production", *Actes du Congrès biennal AFCET'93*, Versailles. vol 1. pp 175-184.

C.S. Tang et E.V. Denardo (1988)
"Models arising from a flexible manufacturing machine, Part I : minimization of the number of tool switches", *Operations Research* vol 36 n°5. pp 767-777.

T. Volgenant et R. Jonker (1982)
"A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation", *European Journal of Operationnal Research*, vol 9. pp 83-89.

Chapitre 4

Les problèmes de k-serveurs

Ce chapitre est destiné à présenter une synthèse des principaux résultats qui ont été établis pour les problèmes de serveurs. Il commence par une description générale de ces problèmes et d'une de leurs applications, puis il se décompose suivant deux axes : l'étude des problèmes uniformes d'une part, suivie de celle des problèmes symétriques et asymétriques d'autre part. Chaque partie débute par l'étude de l'aspect off-line du problème, a priori plus facile, et elle se poursuit par la description d'algorithmes on-line particuliers.

4. 1. Introduction aux problèmes de k-serveurs

4. 1. 1. Introduction

Le problème des k-serveurs consiste à planifier et à décider du déplacement de k serveurs identiques sur les n sommets d'un graphe (complet ou non), suivant une séquence d'appels de ces sommets ; les sommets correspondent à des "clients" sur un réseau. La séquence des appels initialement n'est pas connue, ce qui constitue la principale difficulté du problème. Elle se constitue au fur et à mesure des appels successifs des sommets. Un appel est représenté par le numéro d'un sommet et signifie que l'on doit déplacer un serveur sur ce sommet. On dit alors que le sommet est "couvert" ou "servi". (Dans le même temps, un autre sommet aura été découvert pour satisfaire la demande).

Les arêtes du graphe ont une pondération d : le coût total de réponse à une séquence d'appels est la somme des poids des arêtes parcourues par l'ensemble des serveurs. Si la distance est symétrique, on dira que le problème est symétrique ; si toutes les distances sont égales, on dira que le problème est uniforme.

Le but est de construire un algorithme qui sans connaître à l'avance l'ordre dans lequel se manifesteront les sommets, déterminera à chaque appel quel serveur doit être déplacé parmi les k disponibles, de façon à minimiser la distance totale parcourue.

Ces problèmes ont été introduits par Manasse, McGeoch et Sleator en 1988. Ils ont donné une définition générale de la compétitivité des algorithmes de gestion de serveurs ainsi que deux algorithmes compétitifs pour deux cas uniformes particuliers (Manasse et al (1990)).

Pour ces mêmes cas particuliers mais non uniformes, Fiat et al (1991) ont proposé deux algorithmes aléatoires performants. Et Chrobak, Karloff, Payne et Vishwanathan (1991) sont les premiers à avoir déterminé un algorithme optimal polynomial pour le cas où l'ordre dans lequel les sommets se manifestent est connu à l'avance.

On peut d'ores et déjà remarquer que l'ensemble des articles sur les problèmes de k–serveurs ne concerne que les cas uniformes, ou bien non-uniformes mais symétriques. (Chrobak et al (1991) sont les seuls à apporter un résultat sur le cas non-symétrique pour le problème des 2-serveurs).

4. 1. 2. Une application : la pagination

Le problème des k-serveurs est d'un type assez général qui peut servir à modéliser différents problèmes que l'on rencontre le plus souvent en informatique : on peut citer Raghavan et Snir (1990) pour la gestion des chargements de polices de caractères sur une imprimante (problème de k-serveurs asymétrique), Calderbank, Coffman et Flatto (1985) et Hofri (1983) pour le pilotage d'un disque muni de deux têtes de lecture/écriture (problème de 2-serveurs non uniforme), et Fiat et al (1991) pour le problème de la pagination.

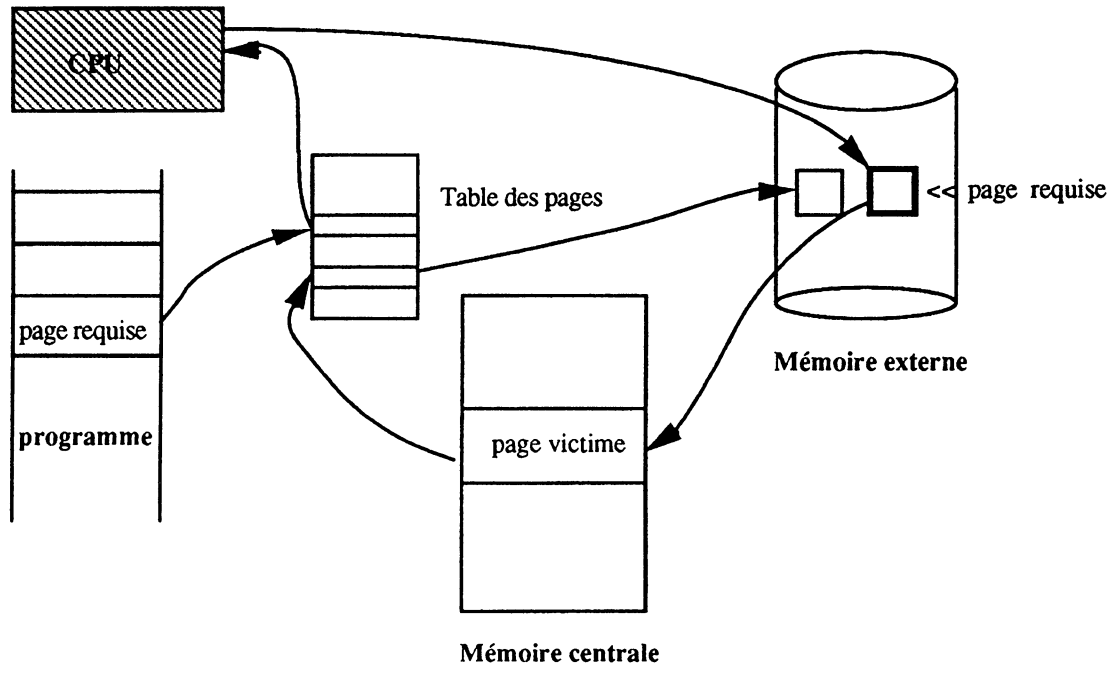
La pagination est un exemple qui se prête particulièrement bien à une telle modélisation. Ce problème se pose plus précisément en architecture des systèmes lorsque le système utilise la pagination pour gérer la mémoire. Notre propos n'est pas de donner ici une description précise et exhaustive de ce problème, (dans un domaine d'étude qui n'est pas précisément le notre), mais plutôt de donner les éléments indispensables à la compréhension de l'intérêt que peut avoir le modèle du problème des k-serveurs pour la pagination. Dans la pratique, la gestion de la mémoire est certainement plus complexe, (pour plus de précisions, on peut par exemple se référer à Tanenbaum (1992)).

On peut décrire schématiquement la mémoire d'un ordinateur comme étant composée de deux types :

- La mémoire **centrale** qui contient entre autres les programmes du système.
- Les mémoires **auxiliaires** ou externes comme les disques ou les disquettes contenant des données et des programmes.

Ces mémoires sont divisées en "cases" ou "pages mémoire" de dimension identique. La taille de la mémoire centrale est le nombre maximum de pages réelles qu'elle peut contenir, soit k. Dans les dernières années, elle a nettement évolué grâce aux progrès techniques qui ont été réalisés, mais dans le même temps la taille des programmes, les techniques de multiprogrammation, l'emploi d'environnements graphiques, etc ... ont également progressé.

De ce fait, il n'est généralement pas possible d'implanter la totalité d'un programme en mémoire centrale : une partie du programme reste stockée sur une mémoire auxiliaire. Lors de son exécution, le programme fait appel à un certain nombre de pages de mémoire virtuelle (données, instructions), dans un ordre qui ne peut être connu à l'avance, (car il dépend des données qui lui ont été fournies, et du déroulement des instructions en fonction de ces données). Une page virtuelle référencée par le programme doit être présente dans une page réelle de la mémoire centrale pour que l'exécution du programme puisse se poursuivre normalement. Lorsqu'une page appelée est absente, on dit qu'il y a "défaut de page" : si la mémoire centrale n'est pas saturée, le système situe la page virtuelle en mémoire auxiliaire puis la charge dans une page réelle de la mémoire centrale. Sinon, il doit déterminer quelle page réelle doit être "vidée" de la mémoire centrale pour faire place à celle qui est demandée.



Ces échanges sont réalisés par l'intermédiaire d'une table des pages, et portent le nom de "swapping". S'ils sont trop fréquents, ils ralentissent de façon non négligeable la vitesse globale de travail de l'unité centrale de l'ordinateur, d'où l'importance du choix d'une stratégie de remplacement des pages qui minimise le nombre de défauts de pages.

La modélisation par des serveurs est simple : Le réseau des clients constitue un graphe complet dont chaque sommet représente une page virtuelle d'un processus. Toutes les pages du système sont ainsi représentées. Chaque arête du réseau est pondérée par 1. On dispose sur ce réseau de k serveurs (autant que la capacité en pages réelles de la mémoire vive). Lorsqu'un serveur se trouve placé sur un sommet cela signifie que la page réelle correspondant au serveur contient la page viruelle représentée par le sommet ; déplacer un serveur d'un sommet à un autre revient à faire sortir une page de la mémoire pour en introduire une nouvelle.

La pagination est donc un problème de k-serveurs uniforme. Elle a été très étudiée dès la fin des années 60 et durant les années 70, en particulier par Belady (1966), Mattson, Gecsei, Slutz et Traiger (1970), et Denning (1970). Les algorithmes LRU et FIFO sont parmi les plus connus :

- Algorithme LRU (**L**east **R**ecently **U**sed) : en cas de remplacement, sortir la page qui n'a pas été référencée depuis le plus longtemps.
- Algorithme FIFO (**F**irst **I**n **F**irst **O**ut) : remplacer la page qui est en mémoire depuis le plus longtemps.

En 1985, Sleator et Tarjan ont apporté une nouvelle approche du problème en comparant les performances des algorithmes de gestion de pages à celles d'un algorithme optimal qui connaîtrait d'avance la séquence des pages requises. Ils ont montré entre autres deux résultats importants :

1- Dans le pire des cas certains algorithmes (LRU et FIFO) pour la pagination peuvent engendrer un coût k fois plus grand que celui d'un algorithme optimal, même si en général ces algorithmes donnent des résultats satisfaisants.

2- Aucun algorithme de gestion de pages (sans connaissance préalable de leur ordre d'appels) ne peut garantir un facteur de compétitivité qui dans le pire des cas reste inférieur strictement à k.

Ces résultats montrent la difficulté du problème. Ils ont inspiré la notion de compétitivité pour le problème des k-serveurs. Cette notion est définie dans le paragraphe suivant.

4. 1. 3. Définitions et propriétés

a) on-line/off-line

La principale difficulté rencontrée dans la construction des algorithmes de gestion de serveurs provient du fait que la qualité des choix qu'ils opèrent est conditionnée par l'ordre dans lequel les sommets se manifesteront : or ces décisions doivent être prises sans connaissance du futur. On emploie le terme **on-line** pour désigner ce type de problèmes.

Un algorithme est dit **on-line** si la ou les décisions qu'il prend à chaque instant sont déterminées sans connaissance du futur déroulement du problème (la séquence en l'occurence). Le problème de la pagination est typiquement on-line.

Par opposition, on définit un algorithme **off-line** comme étant un algorithme qui fait ses choix en fonction d'une complète connaissance du futur (ici la séquence dans sa totalité).

Pour ce dernier type de problèmes, il semble naturellement plus facile de déterminer un algorithme optimal. On peut ensuite s'inspirer de ces stratégies optimales off-line pour essayer de traiter la gestion on-line des serveurs de façon performante. Cette recherche de performance nécessite la définition de critères de comparaison, (de compétitivité), pour les algorithmes on-line.

b) Compétitivité

Soit A un algorithme on-line quelconque. On note $C_A(\sigma)$ le coût de A sur la séquence d'appels σ. On dit que A est **c-compétitif** si quelque soit l'emplacement initial des serveurs il existe une constante a telle que :
Quelle que soit la séquence σ, et pour tout algorithme B on a:

$$C_A(\sigma) \leq c.C_B(\sigma) + a$$

Cette relation s'applique en particulier si B est un algorithme off-line optimal. Si elle est vérifiée, elle garantit que dans le pire des cas, le coût de l'algorithme A ne peut pas excéder c fois celui de l'algorithme optimal ; c est le coefficient de compétitivité. Naturellement, plus il est petit, meilleur est l'algorithme. Un algorithme dont le coefficient est le plus petit possible pour un problème donné est dit *fortement compétitif*. Karlin, Manasse, Rudolph et Sleator (1988) utilisent ce critère de compétitivité pour les problèmes de "cache" dont nous décrivons un aspect à la fin de ce chapitre.

c) Propriétés

Avant de se lancer dans la recherche de tels algorithmes, il est intéressant de s'interroger sur la manière de servir un sommet, en se posant les questions préliminaires suivantes :

Premièrement, lorsqu'un sommet requis dans la séquence porte déjà un serveur, est-il judicieux d'effectuer tout de même un déplacement d'un autre serveur sur ce sommet, (en vue par exemple d'un rapprochement pour le service prochain d'un sommet découvert) ?

Deuxièmement, à chaque nouvel appel, est-il intéressant de déplacer un serveur sur un sommet autre que celui qui se manifeste : autrement dit, des déplacements "prématurés" peuvent-ils réduire le coût total du service d'une séquence ?

Les lemmes suivants apportent la réponse à ces questions : à chaque instant, on ne déplace au plus qu'un seul serveur.

Lemme 1 : (Manasse et al (1990))
Supposons que la pondération d du réseau respecte l'inégalité triangulaire.
Pour tout algorithme B, il existe un algorithme B', (algorithme B modifié), qui ignore les demandes provenant de sommets déjà couverts, et dont le coût n'est pas supérieur à celui de B. (Si B est on-line, alors B' est également on-line).

Preuve :

Supposons qu'un sommet v déjà couvert par B soit requis dans la séquence, et que B effectue tout de même un déplacement d'un serveur s placé sur un sommet u, vers le sommet v. Soit w le premier sommet vers lequel B déplace s par la suite. Si B conserve s sur le sommet u, puis le déplace directement sur w le moment venu :

La différence de coût est d(u,v) + d(v,w) - d(u,w) qui est négative puisque d respecte l'inégalité triangulaire. On considérera donc par la suite que tous les algorithmes étudiés ignorent les demandes provenant de sommets déjà couverts.

Lemme 2 :
Si d respecte l'inégalité triangulaire, tout algorithme B peut être transformé en un algorithme B' qui est dit "économe", (c'est-à-dire qui n'effectue pas de déplacements "prématurés"), et dont le coût n'est pas supérieur à celui de B. (Si B est on-line, alors B' est également on-line).

Preuve :

La preuve se fait par induction sur la séquence des appels. Supposons que l'instant i est le premier dans la séquence où B opère une insertion prématurée : un sommet u est découvert pour servir un sommet v qui n'est pas requis pour l'instant. On modifie B en B' qui imite B jusqu'à l'instant i où il garde u couvert. Il imite ensuite B, jusqu'à ce que un des trois cas suivants se présente :

- v est découvert par B pour servir un sommet w : B' sert alors w en découvrant u, et la différence de coût entre B' et B est de $d(u,w) - [d(u,v) + d(v,w)] \leq 0$

- B couvre u (prématurément ou non) en découvrant un sommet x (on peut avoir x = v) : B' sert v avec x, et la différence de coût entre B' et B est de $d(x,v) - [d(u,v) + d(x,u)] \leq 0$

- v est requis dans la séquence. B ne fait rien, tandis que B' déplace le serveur couvrant u sur v. La différence de coût entre B' et B est nulle.

Dans tous les cas, B' n'opère aucune insertion prématurée jusqu'à l'instant i compris, et possède un coût au plus égal à celui de B. B devient B', puis on répète cette transformation de B sur chaque insertion prématurée observée dans la séquence. Par conséquent, lorsque l'inégalité triangulaire sera respectée, tous les algorithmes étudiés dans la suite seront supposés "économes".

lemme 3 : (Manasse et al (1990))
On appelle séquence "dure" d'un algorithme une séquence sur laquelle l'algorithme est obligé de déplacer un serveur à chaque appel : Si un algorithme est c-compétitif sur chacune de ses séquences "dures", alors il est c-compétitif sur toute séquence.

On peut donc vérifier la compétitivité d'un algorithme en construisant des séquences dites "dures" pour cet algorithme, pour le comparer au coût obtenu avec un algorithme optimal.

4. 2. L'algorithme off-line optimal pour les problèmes uniformes

4. 2. 1. Introduction

L'application du cas uniforme que nous avons décrite, (la pagination), permet de faire le lien entre l'algorithme off-line optimal utilisé pour le problème des k-serveurs uniforme, et l'algorithme décrit habituellement pour la gestion off-line des pages mémoires. Etant donné la similitude existant entre les deux problèmes, cet algorithme est exactement le même : il est appelé MIN par Belady (1966) et OPT par Mattson et al (1970), et son principe est le suivant :

En cas de défaut de page et lorsque la mémoire physique est saturée, on remplace la page qui sera référencée de nouveau au plus tard dans la suite de la séquence.

Cet algorithme fait de la pagination à la demande : il est donc bien économe au sens où nous l'avons défini (lemme 2). Si on effectue un parallèle avec le problème de la gestion off-line des outils, en assimilant la mémoire physique du système au magasin à outils d'une machine, on reconnaît ici exactement la stratégie KTNS utilisée dans le cas où un seul outil serait requis pour chaque tâche. En terme de serveurs, on a donc un algorithme "économe", (que nous désignerons également par OPT), qui lorsque le sommet appelant est découvert, découvre le sommet requis de nouveau au plus tard dans la séquence, (ou qui n'est plus du tout requis par la suite).

Comme pour l'algorithme KTNS, l'optimalité de la stratégie OPT reste plutôt longue et fastidieuse à établir, même si elle ne présente en fait pas de réelle difficulté en soi. Mattson et al (1970) ont montré d'abord que le service à la demande donne un meilleur coût que le service "prématuré", (ou "préventif"), puis ils ont fait une étude de cas détaillée pour montrer que le critère de choix du sommet découvert, (i.e la page sortie), est optimal. McGeoch et Sleator (1991) proposent une démonstration plus compacte, en donnant un algorithme de transformation qui modifie tout algorithme off-line jusqu'à ce qu'il observe la même règle de déplacement des serveurs que OPT, tout en lui conservant un coût inférieur ou égal à celui qu'il avait au départ. Cet algorithme de transformation est décrit dans le paragraphe suivant. Nous en détaillons les différentes étapes, car il semble qu'il y ait une imprécision concernant les cas de déplacements prématurés.

4. 2. 2 Algorithme de transformation de McGeoch et Sleator

Soit A un algorithme off-line qui gère les serveurs (ou les pages). A est modifié en une succession d'algorithmes qui tendent vers l'algorithme OPT. A est transformé en A' : A' a un coût inférieur ou égal à celui de A. Puis A devient A', et ainsi de suite.

Supposons que A et OPT démarrent sur le même état initial, et que jusqu'au ième appel (compris), ils opèrent les mêmes choix. A l'instant i+1, le sommet v est appelé, et A et OPT diffèrent : A déplace x, tandis que OPT déplace w, or w est redemandé après x dans la suite de la séquence.

Soit A' l'algorithme qui imite A jusqu'à la ième demande puis qui déplace w pour couvrir v à l'instant i+1. Ensuite, A' imite A tant que c'est possible, et jusqu'à ce que un des deux cas se présente :

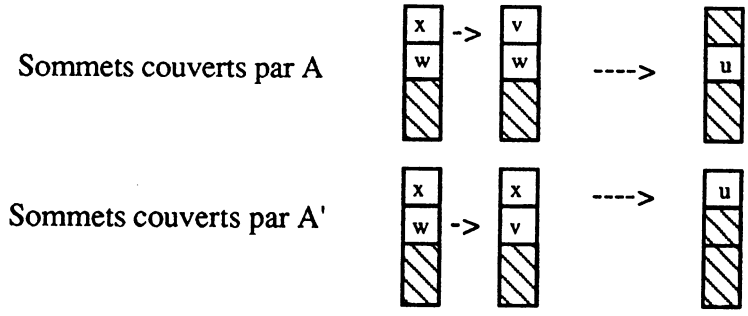
cas 1 : A utilise w pour servir un sommet u, or A' n'a pas w en magasin.
cas 2 : x est appelé (ou inséré prématurément par A), et le sommet u est découvert.

Remarque : Ces deux cas se produiront avant que w ne soit demandé, car w est redemandé après x. (le cas où il serait préventivement inséré par A est impossible car A devrait d'abord sortir w ce qui correspond au cas 1).

Tant que le cas 1 ou le cas 2 ne s'est pas produit, les magasins de A et A' ne diffèrent que d'une page (A contient w tandis que A' contient x). On notera que les cas 1 et 2 peuvent s'intersecter : A sert alors x en utilisant w, et A' ne fait rien.

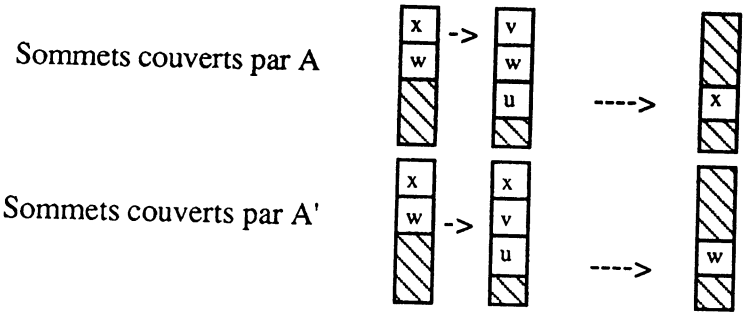
Modification de A dans le cas 1 :

A sert u avec w : A' sert u avec x, et les magasins de A et A' redeviennent identiques, jusqu'à la fin de la séquence. Le coût de A' est le même que celui de A.



Modification de A dans le cas 2 :

A insère x, et pour cela il découvre un sommet u. Dans ce cas, A' déplace le serveur de u sur w pour obtenir le même magasin que celui obtenu par l'algorithme A. Puis A' imite A sur la fin de la séquence. A et A' ont le même coût. C'est seulement lorsque les cas 1 et 2 s'intersectent que A' coûte un changement de moins que A.



La réponse au cas 1 ne pose pas de problème, tandis que celle du cas 2 introduit une insertion préventive (celle de w) qui n'existait pas dans A (sauf si u = w). Toutefois, le coût de A' même avec cette insertion est égal à celui de A. Mais, lors de la transformation de A' en A", puis de A" en A"' et ainsi de suite, rien ne permet de modifier cette insertion préventive qui peut être incompatible avec la solution optimale, (toutes les insertions préventives ne sont pas incompatibles avec l'optimum, mais certaines peuvent occasionner un changement supplémentaire). L'exemple ci-dessous décrit une transformation de A en A' incompatible avec la solution optimale.

S : 1, 2, 3, 4, 5, 2, 3, 1, 5, 2, 4, 6 et C = 4.

A :	4	5	2	3	1	5	2	4	6	
	1 --> 5	5	5	5	5 --> 2	2	2			
	2	2	2	2 --> 1	1	1	1	1		
	3	3	3	3	3	3	3	3	3	
	4	4	4	4	4	4	4	4 --> 6		Coût = 4

OPT :	4	5	2	3	1	5	2	4	6	
	1	1	1	1	1	1	1 --> 4	--> 6		
	2	2	2	2	2	2	2	2	2	
	3	3	3	3	3	3	3	3	3	
	4 --> 5	5	5	5	5	5	5	5		Coût = 3

L'instant 5 est le premier où A (= A1) diffère de OPT : A sort 1 tandis que OPT sort 4 pour servir 5, et 4 est redemandé après 1 : x = 1, w = 4, v = 5
A est modifié en A2, et c'est le cas 2 qu'on rencontre en premier.

A2 :	4	5	2	3	1	5	2	4	6	
	1	1	1	1	1	1	1	1	1	
	2	2	2	2 -->	4	4	4	4 -->	6	
	3	3	3	3	3	3	3	3	3	
	4 -->	5	5	5	5	5 -->	2	2	2	Coût = 4
					*					
					*					
					A = A2					

A2 et A ont le même coût, mais A2 contient une insertion prématurée à l'instant 8 qui n'était pas dans A.

Transformation de A2 en A3 :

Le premier instant où A2 et OPT diffèrent est l'instant 8 lors de l'insertion de 4, tandis que OPT ne fait rien, car c'est 1 qui est demandé et il est déjà servi. Rien n'est prévu pour ce cas : on ne peut pas identifier x à 2 et v à 4. Si on poursuit l'examen des magasins, le prochain instant où A2 et OPT diffèrent est l'instant 10 : A2 déplace x = 5 sur v = 2 qui est appelé, tandis que OPT ne fait rien, (ou sort w = 2 sur v = 2). La suite des transformations s'arrête donc là, et A2 n'est pas optimal à cause de cette insertion "prématurée" qui a été créée : en effet, le coût de A2 est 4, tandis que le coût minimum est égal à 3.

Si on ne modifie A que sur les instants où le sommet découvert n'est pas le sommet requis au plus tard dans la séquence, la suite des algorithmes issus de A peut ne pas converger vers l'algorithme optimal. Pour que la preuve fonctionne complètement, il faut prendre en compte le cas où A opère des insertions "prématurées".

Mattson et al (1970), après avoir examiné le cas des insertions prématurées ont choisi de ne considérer que des algorithmes économes. Pour un algorithme économe A, ils étudient les cas 1 et 2 décrits par McGeoch et Sleator. Dans le cas 1, ils apportent les mêmes transformations sur A, mais dans le cas 2, ils ne peuvent pas modifier A en introduisant une insertion prématurée, car A' doit rester un algorithme économe. Ils sont alors contraints de considérer trois cas supplémentaires en fonction du sommet u découvert au cas 2, et de l'instant où il est de nouveau requis dans la séquence.

L'intérêt de la preuve de McGeoch et Sleator est d'introduire justement une insertion prématurée qui conserve un coût identique au coût de A pour l'algorithme A', mais simplifie la transformation entre les deux algorithmes. Pour garder une preuve compacte, il est donc intéressant de conserver ce principe tout en le complétant.

4. 2. 3 Prise en compte des "insertions prématurées"

La preuve peut être complétée de la façon suivante : Si A et OPT diffèrent à l'instant i+1 par le choix du sommet qu'ils découvrent, on modifie A comme l'indiquent McGeoch et Sleator. S'ils diffèrent par une insertion "prématurée", alors on a :

- A insère v qui n'est pas demandé et sort x.
- OPT ne fait rien.

On applique sur A la même modification que celle que nous avons décrite au paragraphe 1 : A' imite A jusqu'à l'instant i compris. A l'instant i+1, A' ne fait rien, puis il imite A jusqu'à ce que un des cas suivants se présente à un instant j :

Cas 1 : v est utilisé par A pour faire place à un nouveau sommet. A' utilise x pour servir ce sommet. Les magasins de A et A' se retrouvent donc dans la même configuration et font les mêmes choix sur la fin de la séquence : le coût de A' est inférieur strictement à celui de A.

Cas 2 : x est appelé (ou inséré prématurément par A) : A déplace un autre sommet u sur x, alors A' sort u pour insérer v, et occasionne un changement de moins que A. A' a un coût strictement inférieur à celui de A, et les magasins suivants sont identiques entre les deux algorithmes.

Remarque : si A déplace v sur x (car x est demandé ou inséré préventivement), alors on est à l'intersection du cas 1 et du cas 2, et A' ne fait rien donc occasionne deux changements de moins que A.

Cas 3 : v est demandé : A ne fait rien pour servir v, et A' déplace x sur v. (Cette modification est possible, car le cas 1 ne s'est pas encore produit, donc x est toujours servi par A'). Dans ce cas 3, A et A' ont exactement le même coût, et se retrouvent dans une situation identique.

Si le cas 1 se présente en premier, cela a pour effet de supprimer purement et simplement dans A' l'insertion préventive de l'instant i, et on gagne un déplacement.

Si c'est le cas 2 qui arrive d'abord, ou bien x est inséré à la demande à l'instant j, et donc A' comporte une insertion prématurée de moins et autant que A à l'instant j, ou bien A' crée à l'instant j une nouvelle insertion prématurée après avoir supprimé celle de l'instant i. Donc

de toute façon A' opère un nombre d'insertions prématurées inférieur ou égal à celui de A, avec un coût strictement meilleur que celui de A.

Si c'est le cas 3 qui se présente le premier, A' occasionne une insertion prématurée en moins pour un coût identique à celui de A.

Dans tous les cas, l'algorithme A' issu de A possède un coût inférieur ou égal à celui de A et n'opère plus de déplacements non conformes à ceux de OPT jusqu'à l'instant i compris.

Avant de poursuivre par l'étude des problèmes on-line uniformes, il est intéressant de revenir momentanément au problème de la gestion d'outils, car le principe de remplacement de l'algorithme OPT rappelle fortement celui de l'algorithme KTNS.

4. 2. 4 Application à l'algorithme KTNS

L'algorithme de transformation s'applique tel quel à tout algorithme off-line B de gestion d'outils qui ne respecterait pas les deux principes de KTNS (cf page 18) : un sommet découvert correspond ici à un outil sorti du magasin.

Dans le cas où B et KTNS ne sortent pas le même outil, on suppose cette fois que x appartient à un bloc qui est placé avant le bloc auquel appartient le sommet w dans la séquence, ou bien que x et w appartiennent au même bloc, mais que lexicographiquement, x vient avant w. Pour passer de B à B' les mêmes modifications sont apportées dans les cas 1 et 2. On notera toutefois que l'intersection des deux cas ne peut se produire que lorsque x est inséré prématurément par B, (si c'est le cas, l'algorithme B' ne fait rien).

Lorsque B et KTNS diffèrent par une insertion prématurée, on observe les trois mêmes cas, (l'intersection des cas 1 et 2 n'étant toujours possible que si x est inséré préventivement par B), en modifiant B' suivant celui qui se présente le premier :

- **Dans le cas 1,** B' découvre x pour couvrir un sommet u, ce qui est possible, car x n'est pas demandé dans le même bloc que u puisque le cas 2 ne s'est pas encore présenté.
- **Dans le cas 3,** B' découvre x pour couvrir le sommet v, ce qui n'est pas incompatible, car on suppose toujours que le cas 2 ne s'est pas encore présenté.

Cet algorithme de transformation offre une façon supplémentaire de formuler l'optimalité de la stratégie KTNS pour la gestion off-line des outils, qui tout en restant dans le même esprit que celle de Tang et Denardo (1988) devient un peu plus synthétique.

4. 3. Algorithmes "aléatoires" on-line pour les problèmes uniformes

Le terme d'algorithme "aléatoire" est employé ici par opposition à celui d'"algorithme déterministe" que nous emploierons plus loin). Il signifie que des choix aléatoires peuvent entrer dans les décisions prises pour gérer les serveurs.

4. 3. 1. Compétitivité

Avant de décrire les algorithmes aléatoires qui ont été élaborés pour les problèmes uniformes, il est utile d'étendre la définition de la compétitivité aux algorithmes aléatoires : Un algorithme aléatoire A est c-compétitif si il existe une constante a telle que pour toute séquence σ, on a : $E[C_A(\sigma)] \leq c.C_B(\sigma) + a$, où $E[C_A(\sigma)]$ représente le coût moyen sur tous les choix aléatoires que peut faire A sur σ. Dans la suite, on notera simplement $C_A(\sigma)$ pour désigner le coût moyen.

En fait, la notion de c-compétitivité est quasiment la même que celle qui a déjà été énoncée, et cette fois A est comparé à l'ensemble des algorithmes (on-line ou off-line) déterministes. Les algorithmes suivants sont dus à Fiat et al (1991). Les auteurs ont d'abord établi une borne pour le coefficient de compétitivité.

Théorème 1 :
Soit un réseau de n clients, uniforme, portant k serveurs $(1 \leq k \leq n-1)$.
Il n'existe pas d'algorithme aléatoire dont le coefficient de compétitivité soit strictement inférieur à Hk (*avec* $Hk = 1+1/2+1/3+...+1/k$).

Remarque : Nous reviendrons sur la façon d'établir cette borne dans le dernier chapitre, mais on peut d'ores et déjà constater que le coefficient de forte compétitivité décroît dans le cas des algorithmes aléatoires, par rapport à celui des algorithmes déterministes établi par Sleator et Tarjan (1985). En effet, on a :

$$Ln(k+1) \leq Hk \leq Ln(k)+1 \leq k$$

4. 3. 2. L'algorithme de marquage

Un algorithme aléatoire de marquage est proposé pour résoudre le cas uniforme avec un nombre de serveurs k quelconque. Cet algorithme est le suivant : il maintient une liste de sommets marqués de longueur maximale k+1. Initialement, on suppose que les k serveurs se trouvent placés sur les sommets 1, 2, ..., k, et que ces k sommets sont marqués. La procédure se divise en deux temps : marquer et servir.

- Lorsque un sommet est appelé, il est automatiquement marqué. Si le nombre de sommets marqués devient plus grand que k+1, alors toutes les marques sont effacées, sauf celle du sommet le plus récemment référencé dans la séquence.

- On ne déplace un serveur que si le sommet appelé n'est pas déjà couvert. Si ce n'est pas le cas, on choisit aléatoirement et de façon équiprobable un serveur parmi les serveurs qui se trouvent sur des sommets non marqués.

Théorème 2 :
L'algorithme de marquage M est 2.Hk compétitif pour le problème des k-serveurs uniforme (1 ≤ k < n-1).

Théorème 3 :
Pour k = n-1, l'algorithme de marquage est Hn-1 compétitif donc ***fortement compétitif.***

Remarque : Les problèmes à n-1 serveurs peuvent être considérés comme plus faciles dans la mesure où un seul sommet du réseau est découvert à chaque instant. On peut ainsi pressentir plus facilement le comportement des séquences dures et prévoir une réponse en conséquence. Inversement, le problème des 2-serveurs offre à chaque instant un choix de réponse réduit.

4. 3. 3. Le problème des 2-serveurs

L'autre cas particulier étudié est celui des 2-serveurs. Il existe pour ce problème un algorithme fortement compétitif aléatoire : l'algorithme EATR (**E**nd **A**fter **T**wice **R**equested). Cet algorithme est défini par des phases : une phase nouvelle commence quand un sommet est appelé pour la deuxième fois au cours de la phase courante. Elle se compose d'abord d'un certain nombre d'appels de sommets qui ne se sont pas encore manifestés au cours de son déroulement (et qui n'étaient pas couverts). En réponse, EATR choisit aléatoirement et déplace un de ses deux serveurs. La phase se termine par un appel d'un sommet qui a déjà été servi. Dans ce dernier cas, EATR répond en maintenant ses serveurs sur les deux sommets les plus récemment référencés.

Théorème 4 :
L'algorithme EATR est 3/2 -compétitif pour le problème des 2- serveurs uniforme.
Il est donc fortement compétitif (3/2 = H2).

En dehors de ces cas à 2 et n-1 serveurs, c'est à dire pour un nombre de serveurs k quelconque, la recherche d'un algorithme fortement compétitif restait un problème ouvert : McGeoch et Sleator (1991) ont résolu cette question en proposant un algorithme de partitionnement. Il consiste à maintenir une partition des sommets du réseau, destinée à cerner autant que possible le comportement de l'algorithme off-line optimal. Sa structure est décrite de façon détaillée au paragraphe suivant.

4. 3. 4. L'algorithme de partitionnement

Cet algorithme est le seul qui soit fortement compétitif sur les problèmes uniformes, quel que soit k. A ce titre, il nous intéresse tout particulièrement : il est à la base du modèle que nous avons construit pour le service par blocs, et des propriétés qui s'y rapportent qui avec l'élaboration d'une nouvelle méthode heuristique pour la gestion d'outils font l'objet du chapitre 5. Le principe de l'algorithme de partitionnement est de "calquer" dans la mesure du possible,

(car les informations arrivent on-line), les choix de l'algorithme off-line optimal OPT. Il consiste à réaliser une partition des sommets en une suite d'ensembles ordonnés $S\alpha$, $S\alpha+1$, ..., $S\beta-1$, $S\beta$. Cette répartition des sommets ainsi que le nombre d'ensembles peuvent varier à chaque instant. La probabilité qu'un sommet soit réellement couvert dans le stratégie optimale, (schéma off-line), varie suivant l'ensemble auquel il appartient. On distingue deux ensembles particuliers : $S\alpha$ contient des sommets dont on est certain qu'ils ne sont pas couverts par l'algorithme optimal.

A l'inverse, l'ensemble $S\beta$ contient les sommets qui font partie des sommets couverts dans la stratégie optimale. Si l'ensemble $S\beta$ déterminé par l'algorithme de partitionnement contient après le service d'un appel k sommets, alors on se trouve exactement dans la configuration optimale déterminée par l'algorithme off-line à cet instant.

L'algorithme de partitionnement tente de faire les mêmes choix que l'algorithme off-line optimal : il place donc des serveurs sur les sommets de $S\beta$, et laisse ceux de $S\alpha$ découverts. Ainsi plus les ensembles $S\alpha$ et $S\beta$ sont grands plus les décisions prises par l'algorithme de partitionnement sont proches de celles prises par l'algorithme optimal. Pour décrire l'algorithme, plusieurs définitions sont nécessaires :

a) Etat initial :

Les ensembles se créent au fur et à mesure de l'apparition des demandes. L'état initial est constitué de deux ensembles qui sont déterminés par la configuration des serveurs au départ : l'ensemble $S\alpha$ contient les n-k sommets non couverts, et l'ensemble $S\beta$ les k sommets couverts. Après un certain nombre de demandes, on obtient une succession d'ensembles $S\alpha$, $S\alpha+1$, $S\alpha+2$,..., $S\beta-1$, $S\beta$, en appliquant l'une des trois règles que nous décrivons plus loin.

b) Marquage des sommets :

On dispose d'un système de marquage des sommets qui obéit à des règles précises : la marque i, ($\alpha+1 \leq i \leq \beta-1$), ne peut être affectée qu'aux sommets de l'ensemble Si^* :

$$Si^* = S\alpha \cup S\alpha+1 \cup S\alpha+2 \cup \ldots \cup Si.$$

Elle marque soit les sommets possédant déjà une marque i -1, soit les sommets de l'ensemble Si. La marque $\beta-1$ désigne les sommets servis par l'algorithme parmi les sommets de l'ensemble $S^*\beta-1$, en plus de ceux de l'ensemble $S\beta$. La marque $\beta-2$ par exemple, est affectée à des sommets qui à l'instant précédent ont porté (ou qui portent encore) la marque $\beta-1$, etc...

c) Etiquetage des ensembles :

A chaque ensemble Si^* est affectée une étiquette k_i. Cette étiquette a deux significations : d'une part elle désigne le nombre exact de marques de type i qui ont été distribuées. D'autre part,

elle désigne le nombre maximum de serveurs que l'algorithme optimal peut avoir placés sur l'ensemble Si^*. A chaque appel, les étiquettes ainsi que les marques sont mises à jour suivant les règles 1, 2, ou 3.

Cet étiquetage permet d'appliquer la règle suivante : si un ensemble Si porte l'étiquette 0 après mise à jour, cela signifie qu'aucun des sommets de Si^* ne porte de serveur dans la stratégie optimale ; on peut donc avec certitude intégrer tous les sommets de Si^* dans l'ensemble $S\alpha$. Ceci a pour effet de faire disparaître un certain nombre de marques, (les k_i sont remis à jour en conséquence suivant une des 3 règles), et surtout de réduire le nombre d'ensembles. Après cette modification de la partition, on se rapproche de la stratégie optimale car on augmente le nombre de sommets découverts avec certitude. Les étiquettes vérifient les propriétés et relations de récurrence suivantes :

$$
\begin{aligned}
k_\alpha &= 0 && \text{(et } k_\beta \text{ n'existe pas)} \\
k_i &> 0 && \alpha < i < \beta \\
k_i &= k_{i-1} + |Si| - 1 && \alpha < i < \beta \\
k_{\beta-1} &= k - |S\beta|
\end{aligned}
$$

d) Service d'un appel :

Pour chaque appel d'un sommet v, une règle particulière est appliquée en fonction du type d'ensemble auquel il appartient.

Règle 1 :

Si v appartient à l'ensemble $S\beta$, il est déjà couvert : on ne déplace donc aucun serveur et la partition ainsi que les étiquettes ne sont pas modifiées.

Règle 2:

Si v appartient à un ensemble Si, $\alpha < i < \beta$, il y a deux cas possibles :

1 - v porte la marque $\beta-1$. Il n'est pas nécessaire de déplacer un serveur. On transfère le sommet dans l'ensemble $S\beta$, et on diminue de un tous les k_j pour $j \geq i$.

2 - v ne porte pas de serveur. On doit choisir un sommet "victime", c'est-à-dire un sommet marqué par $\beta-1$, dont on déplace la marque sur v, avant le transfert de v dans $S\beta$ et de mettre à jour les k_j comme au point 1 : $k_j = k_j - 1$ pour $j \geq i$. Le déplacement de la marque s'effectue de la manière suivante :

- Transférer toutes les marques j, avec $j \geq i$, sur le sommet v en choisissant à chaque fois la victime de façon aléatoire, uniformément, parmi tous les sommets portant la marque j. Lors du transfert d'une marque, on déplace également toutes les marques d'indice supérieur se trouvant sur le sommet victime : à cette occasion, la marque $\beta-1$ peut être déplacée.

Règle 3 :

Si v appartient à $S\alpha$ on doit déplacer un serveur : On crée un nouvel ensemble $S\beta$ où on transfère v. Les sommets qui constituaient anciennement $S\beta$ sont "refoulés" dans un nouvel ensemble $S\beta-1$ et ainsi de suite. $S\beta-1$ prend l'étiquette k-1, les autres ensembles gardent leur étiquette. Les marques $\beta-1$ sont redéployées de façon aléatoire, uniformément parmi les sommets du nouvel ensemble $S\beta-1$ et les sommets qui portaient la marque $\beta-1$ avant l'appel du sommet v.

Théorème 5 :

L'algorithme de partitionnement est un algorithme aléatoire Hk-compétitif donc fortement compétitif pour le problème des k-serveurs uniforme.

Exemple : La superposition des marques $\beta-1$, $\beta-2$, ... complique très vite la lecture du déroulement de l'algorithme. Sur l'exemple ci-dessous, seule la marque $\beta-1$ est représentée pour des raisons de clarté, mais il va de soi que les autres marques sont présentes.

Séquence d'appels on-line : 9, 6, 8, 1, 9, 6, 3, 5

Etat Initial	$(7, 8, 9)_0$	(1, 2, 3, 4, 5, 6)		
Règle 3	$(7, 8)_0$	$(1, 2, 3, 4, 5, 6)_5$ $\beta-1\ \beta-1\ \beta-1\ \beta-1\ \beta-1$	(9)	
Règle 2	$(7, 8)_0$	$(1, 2, 3, 4, 5)_4$ $\beta-1\ \beta-1\ \beta-1\ \beta-1$	(9, 6)	
Règle 3	$(7)_0$	$(1, 2, 3, 4, 5)_4$ $\beta-1\ \beta-1\ \beta-1\ \beta-1$	$(9, 6)_5$ $\beta-1$	(8)
Règle 2	$(7)_0$	$(2, 3, 4, 5)_3$ $\beta-1\ \beta-1\ \beta-1$	$(9, 6)_4$ $\beta-1$	(8, 1)
Règle 2	$(7)_0$	$(2, 3, 4, 5)_3$ $\beta-1\ \beta-1\ \beta-1$	$(6)_3$	(8, 1, 9)

Règle 2	$(7)_0$	$(2,\ 3,\ 4,\ 5)_3$	$()_2$	(8, 1, 9, 6)
		β–1 β–1		
Règle 2	$(7)_0$	$(2,\ 4,\ 5)_2$	$()_1$	(8, 1, 9, 6, 3)
		β–1		
	(7, 2, 4)	(8, 1, 9, 6, 3, 5)		

Remarques : On constate que les relations de récurrence entre les étiquettes et la cardinalité des ensembles sont conservées quelle que soit la règle appliquée pour le service d'un sommet, ce qu'on peut d'ailleurs facilement vérifier en raisonnant par induction sur la séquence des appels.

L'algorithme de partitionnement suit quasiment "à la trace" la stratégie off-line optimale : en effet, l'une des principales caractéristiques de la partition est de permettre à chaque instant de reconstituer l'ensemble des sommets couverts par OPT, à condition que l'on connaisse à partir de cet instant la suite de la séquence. La reconstitution de cet ensemble se fait suivant une procédure que nous étudierons en détail dans le chapitre suivant.

4. 4. Les problèmes non-uniformes

4. 4. 1. Introduction

On considère maintenant un réseau de clients dont chaque arête (u,v) a une pondération d(u,v) : le coût de service d'un client dépend donc de la position du serveur qui sera déplacé pour le servir. En fait le terme de "problème non-uniforme" n'est pas vraiment approprié, car ce sont le plus souvent des problèmes dans lesquels la pondération du réseau est une distance qui sont étudiés. La stratégie off-line optimale pour les cas uniformes n'est plus appropriée, mais le principe de recherche d'un algorithme économe reste toujours justifié. Il en est de même pour les problèmes asymétriques à condition que la pondération d vérifie toujours l'inégalité triangulaire. Nous décrivons une application de ce dernier cas - le "weighted cache problem"- traitée par Chrobak et al (1991).

4. 4. 2. Un Algorithme off-line optimal

Etant donné une séquence d'appels, il existe un nombre fini de façons de répondre à ces appels. On peut donc imaginer un algorithme off-line (très coûteux !) qui examinerait toutes ces possibilités et retiendrait la meilleure, ce qui garantit l'existence d'un algorithme off-line optimal. Manasse, McGeoch et Sleator (1990) ont utilisé la programmation dynamique pour déterminer une stratégie optimale. Cet algorithme nécessite moins de calculs qu'une simple énumération, mais il est tout de même en $(C_n^k)^2$ où k est le nombre de serveurs et n le nombre de sommets. Son principe est le suivant :

Un état S est défini par une des positions possibles des serveurs après un appel dans la séquence. A chaque instant du déroulement de la séquence, il y a au maximum C_n^k états possibles, mais seuls ceux qui couvrent le dernier sommet appelé sont cohérents. Pour passer d'un appel dans la séquence au suivant, on examine toutes les transitions de tous les états S cohérents aux nouveaux états T possibles. Pour terminer, c'est l'état dont le coût est le plus petit après le dernier appel de la séquence, qui donne le coût optimal. Ensuite, la stratégie optimale de l'algorithme se reconstitue par "backtracking", comme dans toute méthode de programmation dynamique.

Manasse et al (1990) ont utilisé ce calcul par la programmation dynamique pour calculer des "c-résidus" $Rc(\sigma,T)$, c'est-à-dire la différence entre c fois le coût d'un algorithme optimal finissant dans l'état T sur la séquence σ et le coût d'un algorithme on-line A dont on veut évaluer la compétitivité. En effet, si A est c-compétitif, alors il existe une constante a telle que pour toute séquence σ et pour état final T on ait : $-a \leq Rc(\sigma,T)$.

Cette dernière propriété est utilisée pour prouver les coefficients de compétitivité des algorithmes qu'ils proposent.

Cette première approche était plutôt décevante, dans la mesure où pour traiter les problèmes uniformes, on connaissait déjà un algorithme off-line qui lui était polynomial : l'algorithme OPT utilisé pour la gestion des pages mémoire. On pouvait donc espérer déterminer par une approche différente un algorithme off-line optimal et polynomial pour le problème général des k-serveurs. En effet, le problème peut par exemple être exprimé comme un problème de transport.

Chrobak et al ont effectivement proposé en 1991 un algorithme optimal en $O(kn^2)$. Pour cela, on ramène le problème au calcul d'un flot maximum de coût minimum dans un réseau sans cycle, autre manière de formuler un problème de transport.

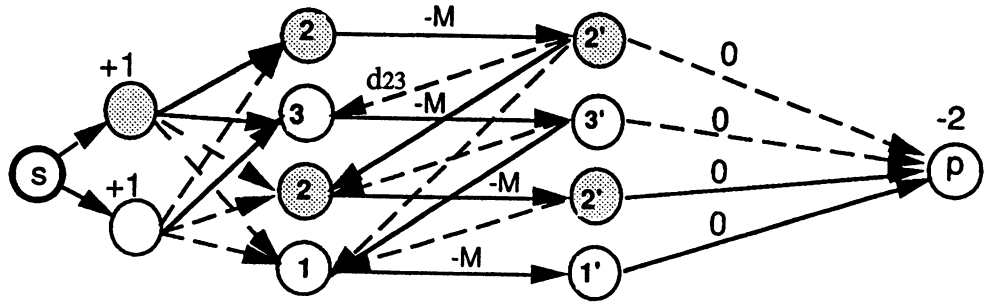
Modélisation :

On suppose qu'initialement, tous les serveurs se trouvent groupés sur un même sommet appelé "sommet d'origine". Le graphe comporte un sommet source et un sommet puits fictifs auxquels sont reliés autant de sommets que de serveurs (on appellera ces derniers sommets des "sommets-serveurs"). Le coût de ces arcs est nul. La séquence est représentée par une série de sommets $r_1, \ldots, r_n$ pour une séquence de longueur n. Ces sommets sont reliés aux "sommets-serveurs" par des arcs dont le coût représente la distance entre le sommet correspondant à un appel r_j, et le sommet d'origine.

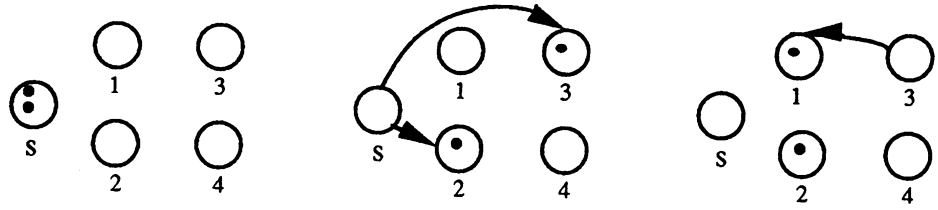
Tous les sommets r_j sont dédoublés en une série de sommets $r'_1, \ldots, r'_n$. Pour $i < j$, il existe un arc entre r'_i et r'_j ; son coût est -M où M représente une valeur arbitrairement grande. La création de ces arcs permet de faire circuler un serveur dans le graphe sur une succession de sommets d'appel, sans pour autant créer de circuits. De plus comme ces arcs ont un coût infiniment négatif, ils seront empruntés par le flot max de coût minimum. Le réseau ainsi

constitué comprend (2 + k + 2n) sommets. L'algorithme utilisé pour calculer le flot max de coût minimum sur ce réseau est celui de Tarjan (1983), que nous avons utilisé au chapitre 2.

L'exemple ci-dessous illustre la construction du réseau. Il s'agit d'un réseau de taille très réduite (4 sommets et 2 serveurs), car la multiplicité des arêtes (r'i, rj) complique très vite la lecture de l'exemple. On considère la séquence partielle σ : 2, 3, 2, 1. Chaque couleur représente le trajet d'un serveur.



Ce flot correspond aux mouvements de serveurs suivants :



Remarque : On pourrait également appliquer la modélisation par niveaux que nous avons décrite au chapitre 2 pour la gestion d'outils, mais dans ce cas la taille du réseau est plus grande.

4. 4. 3. Algorithmes déterministes pour une distance d symétrique

Les résultats qui suivent proviennent des travaux de Manasse, McGeoch et Sleator (1990). Ils ne s'appliquent que si d est réellement une distance, c'est-à-dire si l'inégalité triangulaire est vérifiée et si d est symétrique. Ils peuvent donc éventuellement être utilisés pour des problèmes uniformes.

Théorème 6 :
Soit un réseau de n clients, pondéré par une distance d symétrique, portant k serveurs (1 ≤ k ≤ n-1). Il n'existe pas d'algorithme déterministe dont le coefficient de compétitivité soit strictement inférieur à k.

Manasse et al (1990) ont étudié deux cas extrêmes : le problème des 2-serveurs, et celui des (n–1)-serveurs.

Un algorithme pour le problème des n-1 serveurs :

Cet algorithme est appelé Balance (BAL) car il tend à bouger tous ses serveurs de façon à peu près équivalente à tour de rôle. Pour chaque serveur, Bal tient à jour la distance totale parcourue depuis le début de la séquence d'appels. Soit Di cette distance pour le sommet i. BAL déplace le serveur qui aura le coût cumulé le plus petit après son déplacement, c'est-à-dire que lorsque un sommet j est référencé par la séquence, BAL déplace le serveur du sommet i tel que :

Di + dij est minimum.

Théorème 7 :

BAL est (n-1)-compétitif pour le problème des (n-1)-serveurs (symétrique sur un réseau de n sommets).

Remarques :

1 - Bal est donc fortement compétitif pour k = n-1.

2 - Pour le cas général, lorsque k < n-1, Balance n'est pas un algorithme k-compétitif.

Un algorithme pour le problème des 2-serveurs :

Cet algorithme est fondé sur les résidus qui proviennent du calcul de la stratégie optimale par programmation dynamique, c'est pourquoi les auteurs l'ont appelé RES. Pour décrire précisément l'algorithme, il est nécessaire de définir les résidus : on note Ri le résidu obtenu en comparant le coût de RES à deux fois celui d'un algorithme optimal qui, après un appel, couvrirait le dernier sommet référencé, et un sommet i. A chaque instant on numérote le dernier sommet référencé par 1, et le sommet portant l'autre serveur par 2. Lors d'un appel d'un sommet j découvert, le choix du serveur déplacé s'effectue de la façon suivante :

- **Si** MIN i { $R_i + 2d_{ij}$ } ≥ $2d_{1j} + d_{12}$ **alors** RES déplace le serveur du sommet **1**.
- **Sinon** Res déplace le serveur du sommet **2**.

Théorème 8 :

Res est 2-compétitif pour le problème des 2-serveurs (symétrique), donc Res est fortement compétitif pour ce problème.

En dehors de ces deux cas extrêmes, on ne connait pas pour l'instant d'algorithme déterministe fortement compétitif pour les problèmes de k-serveurs symétriques. Toutefois, si l'on modifie légèrement les données du problème, on peut considérer qu'il existe une exception : Chrobak et al (1991), ont étudié un problème dans lequel le réseau des clients présente une forme bien particulière :

Les clients (et les serveurs) sont placés en ligne. Une demande peut donc être satisfaite au plus par deux serveurs : soit le serveur se trouvant à la droite du client, soit celui se trouvant à sa gauche, à moins que tous les serveurs soient situés du même coté par rapport au client, auquel cas il n'y a pas le choix. Supposons que le serveur situé à droite du client à servir se déplace : le coût enregistré par l'algorithme est la distance réelle qui sépare la position du client de celle de son serveur de droite.

Ce modèle de k-serveurs particulier peut également s'appliquer à la gestion du déplacement des deux bras de lecture/écriture d'un disque, application étudiée par Calderbank, Coffman et Flatto (1985). Chrobak et al ont donné un algorithme k-compétitif : l'algorithme Double-Coverage. On peut considérer qu'il représente pour l'instant le seul algorithme fortement compétitif existant pour le problème des k-serveurs symétrique.

4. 4. 4. Le cas asymétrique

L'application du problème des k-serveurs asymétrique qui est la plus fréquemment étudiée, est en fait un cas particulier : il s'agit du "weighted cache problem" (Manasse, McGeoch et Sleator (1990)). Raghavan et Snir (1990) ont donné un algorithme aléatoire k-compétitif pour ce problème, et Chrobak et al (1991) en proposent un autre. Mais en ce qui concerne le cas général asymétrique, on trouve peu de résultats.

Un algorithme k-compétitif pour le "weighted cache problem" :

Comme pour le problème de la pagination, cette application du cas asymétrique est un problème qui se pose dans la conception des systèmes d'exploitation ou en architecture des ordinateurs. La notion de "cache" ou "cache system" provient du verbe français "cacher" : il s'agit simplement de stocker des ressources fréquemment utilisées, (parties de programmes, données, polices de caractères pour une imprimante, ...), dans une zone d'accès rapide.

Un système de cache est souvent utilisé pour la gestion de la mémoire : lors de l'accès en lecture/écriture à un bloc mémoire, le bloc est d'abord recherché dans le cache. S'il est présent, il peut être traité immédiatement. Sinon, il est recherché en mémoire centrale, puis recopié dans le cache suivant un algorithme de remplacement car la taille du cache est naturellement limitée (de l'ordre de 1% de la mémoire centrale). Des informations plus complètes sur l'utilisation des caches dans les ordinateurs peuvent être obtenues dans Tanenbaum (1991), Clements (1991), et Krakowiak (1987).

Dans le cas du "weighted cache problem", un poids est associé à chaque bloc mémoire. La modélisation sous forme de problème de k-serveurs est la même que pour la pagination, mais ici un poids est associé à chaque sommet. Si le sommet x a le poids W(x), cela signifie que le chargement du bloc mémoire x dans le cache occasionne un coût W(x). On est donc en présence d'un cas asymétrique de k-serveurs. Pour le résoudre, on utilise principalement des algorithmes dont le fonctionnement est fondé sur l'enregistrement (partiel ou total), et l'étude du

comportement des serveurs sur une partie de la séquence qui s'est déjà manifestée (LRU par exemple). De telles méthodes utilisent le passé, par opposition à d'autres dites "sans mémoire".

Lemme 4 :
Pour le "weighted cache problem", il n'existe pas d'algorithme déterministe compétitif sans mémoire.

De fait, les stratégies de gestion des mémoires caches sont fondées sur le stockage d'informations relatives aux appels de pages qui se sont déjà manifestés précédemment. Chrobak et al (1991) donnent un algorithme de ce type sur le même principe que l'algorithme Balance, (que nous avons décrit pour le problème des (n-1)-serveurs symétrique), et qui est k-compétitif. Toutefois, ces algorithmes sont adaptés à la structure particulière du "weighted cache problem". Si au contraire le réseau des clients a une structure et une pondération asymétrique quelconques, on ne peut plus parler d'algorithmes compétitifs, et a fortiori fortement compétitifs : comme le montre le théorème de "non-existence" suivant, on ne peut pas déterminer de coefficient de compétitivité pour le cas général asymétrique.

Théorème 9 :
Il n'existe pas d'algorithme compétitif pour le problème des k-serveurs lorsque les poids des arêtes sont asymétriques, (même pour k = 2).

En effet, si on suppose qu'il existe un algorithme c-compétitif pour ce problème, il est toujours possible de construire un réseau de clients et une pondération asymétrique dont la structure particulière est fonction du coefficient c, et une séquence S tels que le coût de l'algorithme soit très largement supérieur à c fois celui de l'algorithme optimal sur S (voir modèle de construction du graphe dans Chrobak et al (1991)).

4. 5. Conclusion

Les cas particuliers ainsi que les diverses applications des problèmes de k-serveurs que nous avons décrits montrent l'intérêt qui motive l'étude de ces problèmes : les principaux résultats que nous avons recensés concernant les algorithmes de gestion de serveurs permettent aussi d'en apprécier la difficulté. En effet, on constate qu'il y a peu de cas auxquels on sache d'emblée apporter une réponse satisfaisante :

Premièrement, pour le problème des k-serveurs sur un réseau dont les arêtes portent des poids asymétriques, il n'est pas possible d'envisager un algorithme fortement compétitif au sens où il a été défini, et d'un point de vue général, on en sait très peu sur ces problèmes. On peut toutefois penser que leurs applications sont plus rares que celles des problèmes uniformes dans la pratique.

Deuxièmement, pour le cas général symétrique, on ne connaît pas à proprement parler pour l'instant d'algorithme fortement compétitif. Manasse et al (1990) conjecturent qu'un tel algorithme existe, mais ils n'ont pas réussi pour le moment à adapter leur algorithme fortement compétitif pour le problème des 2-serveurs aux problèmes de 3-serveurs et plus. Il faut noter que même si on disposait d'un tel algorithme, son coût dans le pire des cas pourrait atteindre k fois celui d'un algorithme optimal, ce qui pour certaines applications peut être tout à fait inacceptable, (ce n'est que pour le cas uniforme qu'on dispose d'algorithmes déterministes fortement compétitifs, LRU et FIFO par exemple).

En général, les cas particuliers étudiés sont souvent ceux des 2-serveurs, ou des (n-1)-serveurs. Ils sont plus faciles dans la mesure où la variété des situations qu'ils engendrent et les choix de décisions qu'ils offrent sont plus restreints : pour les 2-serveurs, il n'y a à chaque instant que deux décisions possibles pour un déplacement. Pour les (n-1)-serveurs, il est plus facile de parer chaque demande car il n'existe qu'un seul sommet découvert dans tout le réseau. Ce n'est en général pas le cas pour les applications qui nous intéressent : la pagination, et la gestion on-line des changements d'outils.

En revanche, l'algorithme de partitionnement élaboré par McGeoch et Sleator (1991) apporte une solution intéressante pour les problèmes uniformes. En effet, les deux meilleurs algorithmes, FIFO et LRU, jusqu'alors les plus répandus, sont k-compétitifs tandis que l'algorithme de partitionnement est Hk-compétitif. Etant donné que la capacité de la mémoire centrale d'un ordinateur peut être de l'ordre de plusieurs centaines de pages, on peut s'attendre à des gains de temps non négligeables, bien qu'aucune étude comparative ne semble encore avoir été réalisée dans la pratique.

Pour le domaine d'application qui est le nôtre, cet algorithme est également très intéressant : nous l'avons adapté afin de construire un modèle pour la gestion on-line des changements d'outils, que nous étudions de façon détaillée dans le chapitre suivant.

4. 6. Bibliographie

L.A. Belady (1966)
"A study of replacement algorithms for virtual storage computers", *IBM System Journal* vol 5 n°2. pp 78-101.

A.R. Calderbank, E.G.Coffman et L. Flatto (1985)
"Sequencing problems in two-server systems", *Mathematics of operations research* vol 10 n°4. pp 586-598.

M. Chrobak, H.Karloff, T.Payne, et S.Vishwanathan (1991)
"New results on serveur problems", *SIAM Journal on Discrete Mathematics* vol 4, n°2. pp 172-181.

A. Clements (1991)
The principles of computer hardware, Oxford University Press. 2° Ed. pp 284-291.

P.J. Denning (1970)
"Virtual Memory", *ACM Computing Surveys* vol 2 n°3. pp 153-189.

A. Fiat, R.Karp, M.Luby, L.McGeoch, D.Sleator et N.E.Young (1991)
"Competitive Paging Algorithms", *Journal of algorithms* vol 12. pp 685-699.

M. Hofri (1983)
"Should the two-headed disk be greedy ? ...", *Information Processing Letter* vol 16. pp 83-85.

A. Karlin, M.Manasse, L.Rudolph et D.Sleator (1988)
"Competitive Snoopy Caching", *Algorithmica* vol 3. pp 79-119

S. Krakowiak (1987)
Principes des systèmes d'exploitation des ordinateurs, Dunod informatique, Bordas Paris. pp 378-381.

L. McGeoch et D.Sleator (1991)
"A Strongly Competitive Randomized Paging Algorithm", *Algorithmica* vol 6. pp 816–825.

M. Manasse, L.McGeoch, et D.Sleator (1988)
"Competitive algorithms for on-line problems", *Proceedings, 20th ACM symposium on Theory of Computing*. pp 208-230.

M. Manasse, L.McGeoch, et D.Sleator (1990)
"Competitive Algorithms for server problems", *Journal of algorithms* vol 11. pp 208-230.

R. Mattson, J.Gecsei, D.Slutz, et I.Traiger (1970)
"Evaluation techniques for storage hierarchies", *IBM System Journal* n°2. pp 78-117.

P. Raghavan et M.Snir (1990)
"Memory versus randomization in on-line algorithms", Technical Report, IBM Research Report RC 15622 (march 1990).

D. Sleator et R.E.Tarjan (1985)
"Amortized efficiency of list update and paging rules", *Comm. ACM* vol 28 n°2. pp 202-208.

A.S. Tanenbaum (1991)
Architecture de l'ordinateur, InterEditions, Paris. 3° Ed. pp 284-291.

A.S. Tanenbaum (1992)
Modern Operating Systems, Prentice Hall International Editions, Englewood Cliffs, N.J. pp 89-128.

C.S. Tang et E.V. Denardo (1988)
"Models arising from a flexible manufacturing machine, Part I : minimization of the number of tool switches", *Operations Research* vol 36 n°5. pp 767-777.

R.E. Tarjan (1983)
"Data structures and Network Algorithms", *BMS-NSF Regional Conference Series in Applied Mathematics* Vol 44. pp 109-111.

Chapitre 5

Le service par blocs

Ce chapitre est composé de deux parties a priori relativement différentes. Mais la relation existant entre elles deux apparaît clairement dans la seconde partie. Dans les cinq premières sections, nous développons une généralisation du modèle des k-serveurs. Nous proposons un algorithme pour résoudre ce problème (directement inspiré de celui de McGeoch et Sleator), dont nous étudions les propriétés et la compétitivité. Nous recherchons ensuite une borne inférieure du coefficient de compétitivité pour deux cas particuliers.

Dans une seconde partie, nous revenons au problème d'ordonnancement avec gestion d'outils auquel nous appliquons la modélisation (sous forme de problème de k-serveurs généralisé), que nous avons définie.

5. 1. Extension du problème des k-serveurs uniforme

5. 1. 1. Le service par blocs

Les données du problème sont les mêmes que celles du problème des k-serveurs classique ; c'est uniquement la structure de la séquence d'appels qui change, ainsi que la manière de servir : à chaque instant, ce n'est plus un unique sommet qui se manifeste, mais un ensemble de clients qui demandent à être servis *simultanément* ; c'est-à-dire qu'un même serveur ne peut pas couvrir un sommet d'un ensemble, puis se déplacer pour aller servir un autre sommet de ce même ensemble. Cet ensemble correspond à une phase à la fin de laquelle tous les sommets qui se sont manifestés portent un serveur. Le nombre de clients qui constituent un ensemble (ou bloc d'appels), ne peut donc pas excéder k le nombre de serveurs. On peut d'ores et déjà examiner deux cas extrêmes :

Si à chaque instant, ce sont des blocs de k clients qui se manifestent, la gestion des serveurs est simplifiée ; on laisse d'abord en place les serveurs qui se trouvent déjà sur des sommets du bloc. On doit ensuite décider du déplacement du reste des serveurs. Ce problème peut se représenter sous la forme d'un graphe biparti entre les sommets portant un serveur (et pouvant être découverts), et les sommets demandant à être servis. Chaque arête garde sa pondération d'origine. Le calcul d'un couplage de poids min sur ce graphe détermine le meilleur déplacement des serveurs. S'il s'agit d'un problème uniforme, le cas est trivial : toutes les affections des k serveurs aux k sommets demandeurs entraînent le même coût à condition qu'elles laissent en place les serveurs déjà positionnés sur des clients du bloc d'appels, (l'inégalité triangulaire étant naturellement vérifiée par la pondération uniforme).

L'autre cas extrême rejoint la gestion unitaire des serveurs, c'est-à-dire lorsque chaque bloc ne contient qu'un seul sommet ; on retrouve le problème classique des k-serveurs.

Pour le cas général, il faut compter maintenant avec une difficulté supplémentaire par rapport au service unitaire. En effet, l'incertitude qui portait auparavant uniquement sur le numéro du prochain sommet requis, porte ici non seulement sur la composition du bloc, (les numéros des sommets qui le composent), mais également sur la longueur du bloc, c'est-à-dire le nombre de clients qui appellent simultanément.

Avec cette extension au service par bloc, la stratégie off-line optimale n'est plus donnée par l'algorithme OPT, mais par l'algorithme KTNS : en effet, nous avons déjà fait plusieurs fois l'analogie entre ces deux stratégies off-line, et il est clair que KTNS est adapté à la gestion off–line des blocs d'appels.

En revanche, pour la gestion on-line des serveurs, l'algorithme de McGeoch et Sleator n'est naturellement pas adapté au service par blocs, mais son principe essentiel : "traquer la stratégie optimale", reste applicable et nécessite des modifications en particulier des règles de traitement des sommets.

5. 1. 2. Adaptation de l'algorithme de partitionnement

Le principe de la partition des sommets, le système d'étiquetage des ensembles et de marquage des clients sont conservés. Bien que les sommets demandent à être servis simultanément, on maintient de façon artificielle un service unitaire et séquentiel de chaque appel faisant partie d'un bloc, tout en assurant une couverture simultanée de ces appels une fois que tout le bloc a été traité.

Lorsqu'un bloc de sommets se manifeste, il est traité de la façon suivante : Il faut d'abord remarquer que les sommets à l'intérieur d'un bloc ne sont pas ordonnés. En effet, rien a priori n'indique l'ordre dans lequel les sommets doivent être servis. On décide donc d'un ordre artificiel : Les sommets portant déjà la marque β–1 sont placés en premier, (dans l'ordre croissant de leur numéro), ce qui est naturel car ils sont déjà servis ; examinés en premier, ils resteront

couverts pendant le traitement des autres appels. Les autres sommets sont placés à la suite (par ordre croissant de leurs numéros). Une fois le bloc d'appels ordonné, ses sommets sont traités comme une séquence de demandes unitaires connues à l'avance :

Pour chaque appel, une des quatre règles que nous décrivons est appliquée en fonction du type d'ensemble auquel le sommet appartient dans la partition : les règles 1 et 2 n'ont pas été modifiées, mais la règle 3 a été adaptée, et la règle 4 ajoutée afin de s'adapter au service par bloc et de maintenir les propriétés et les relations entre les étiquettes. Soit v le sommet appelé :

Règle 1 :

Si v appartient à l'ensemble $S\beta$, il est déjà couvert, donc on ne déplace aucun serveur.

Règle 2:

Si v appartient à un ensemble Si, $\alpha < i < \beta$, il y a deux cas possibles :

1 - v porte la marque $\beta-1$. Il n'est pas nécessaire de déplacer un serveur. On transfère le sommet dans l'ensemble $S\beta$, et on diminue de un tous les k_j pour $j \geq i$.

2 - v ne porte pas de serveur. On doit choisir un sommet "victime", c'est-à-dire qu'on déplace la marque $\beta-1$ de ce sommet vers v, avant de transférer v dans $S\beta$ et de mettre à jour les k_j comme au point 1. Le déplacement de la marque s'effectue de la manière suivante :

- Transférer toutes les marques j, avec $j \geq i$, sur le sommet v en choisissant à chaque fois la victime de façon aléatoire, uniformément, parmi tous les sommets portant la marque j. Lors du transfert d'une marque, on déplace également toutes les marques d'indice supérieur se trouvant sur le sommet victime : à cette occasion, la marque $\beta-1$ peut être déplacée. Notons que pendant ce transfert de marques, aucun sommet déjà couvert et faisant partie du bloc en cours de traitement, ne peut être fortuitement découvert, car grâce à l'ordre qui a été défini sur le bloc et à la règle 4, tous les sommets déjà couverts appartiennent à l'ensemble $S\beta$.

Règle 3 :

Si v appartient à $S\alpha$ on doit déplacer un serveur : On crée un nouvel ensemble $S\beta$ où on transfère v. Les sommets qui constituaient anciennement $S\beta$ "sont refoulés" dans un nouvel ensemble $S\beta-1$ et ainsi de suite. $S\beta-1$ prend l'étiquette k-1, les autres ensembles gardent leur étiquette. Les marques $\beta-1$ sont redéployées parmi les sommets "elligibles" : c'est à dire les sommets du nouvel ensemble $S^*\beta-1$ et les sommets portant la marque $\beta-1$, (mais en priorité sur les sommets qui font partie du bloc en cours et qui ont déjà été traités). Si à l'occasion de la translation des sommets de $S\beta$ vers le nouvel $S\beta-1$, des sommets du bloc en cours de traitement sont "refoulés" dans $S\beta-1$, alors on applique la règle 4 :

Règle 4 :

Transférer chaque sommet de $S\beta-1$ faisant partie du bloc, dans $S\beta$ en diminuant à chaque fois de 1 la valeur de $k_{\beta-1}$. Cette opération peut vider totalement l'ensemble $S\beta-1$, ce qui ne le fait pas disparaître pour autant, à moins que son étiquette ne devienne nulle.

Les règles ont été modifiées de façon à satisfaire trois impératifs : premièrement, l'ensemble $S\beta$ doit contenir tous les sommets d'un bloc une fois que tous ses appels ont été traités, ce qui est une condition nécessaire si on veut suivre la stratégie optimale. Deuxièmement, un même serveur ne doit pas passer d'un appel à un autre faisant partie du même bloc. Et troisièmement, les relations de récurrence de base entre les ensembles et leurs étiquettes doivent toujours être vérifiées :

$$k_i > 0 \qquad \alpha < i < \beta \qquad \text{et } k_\alpha = 0$$

$$k_i = k_{i-1} + |Si| - 1 \qquad \alpha < i < \beta \qquad \text{et } k_{\beta-1} = k - |S\beta|$$

Proposition :

Après chaque appel, quel que soit le bloc dont cet appel fait partie et quel que soit l'ensemble qui le contient dans la partition, les relations de récurrence entre les ensembles et leurs étiquettes sont toujours vérifiées.

Preuve :

Avant le premier bloc d'appels dans la séquence, les relations sont clairement vérifiées (on a $k\alpha = 0$ et $\alpha+1 = \beta$). Supposons qu'elles le sont toujours jusqu'à un appel v d'un bloc B, et montrons que quel que soit l'ensemble auquel v appartient, (donc quelle que soit la règle appliquée), il en est de même une fois que v a été traité. Le cas où v fait déjà partie de $S\beta$ étant sans intérêt, examinons celui où v appartient à un ensemble S_j ($\alpha < j < \beta$) : Après le déplacement des marques, les étiquettes et les ensembles sont inchangées pour $i < j$. Par contre, on a $k'_i = k_i - 1$ pour $i \geq j$ et $S'_j = S_j \setminus \{v\}$. On a donc :

- Pour $i < j$, les égalités sont toujours vérifiées.
- Pour $i = j$, on a $k'_j = k_j - 1 = (k_{j-1} + |S_j| - 1) - 1 = k'_{j-1} + |S'_j| - 1$
- Pour $i > j$ on a $k'_i = k_i - 1 = (k_{i-1} + |S_i| - 1) - 1 = k'_{i-1} + |S'_i| - 1$
- Pour $i = \beta$, $k'_{\beta-1} = k_{\beta-1} - 1 = k - |S\beta| - 1 = k - |S'\beta|$

Supposons maintenant que le sommet v appartienne à l'ensemble $S\alpha$: si aucun sommet du bloc B n'appartient à $S\beta$ avant que le sommet v ne soit traité, alors seuls les ensembles $S\alpha$ et $S\beta$ sont modifiés et la partition augmente d'un ensemble. Les étiquettes et les ensembles ne sont pas modifiés, sauf pour l'avant dernier ensemble :

$$k'_{\beta-1} = k - 1 = k - |S'\beta|$$

$$k'_{\beta-2} + |S'\beta-1| - 1 = k_{\beta-1} + |S\beta| - 1$$

$$= k - |S\beta| + |S\beta| - 1$$

$$= k - 1 = k'_{\beta-1}$$

Si la règle 4 entre en action, et qu'un certain nombre de sommets (soit x sommets), sont rapatriés dans l'ensemble de tête S'β, alors on a :

$$k'_{\beta-1} = k - 1 - x = k - |S'\beta|$$

$$k'_{\beta-2} + |S'\beta-1| - 1 = k_{\beta-1} + |S\beta| - x - 1$$

$$= k - |S\beta| + |S\beta| - x - 1$$

$$= k'_{\beta-1}$$

Notons que l'ensemble S'β–1 peut être vide sans que cela perturbe la validité des inégalités. Les relations de récurrence sont donc respectées non seulement à la fin de chaque bloc mais aussi après chaque appel d'un sommet.

5. 2. Propriété de l'algorithme de service par blocs

Au cours du déroulement de la séquence, et après avoir servi un certain nombre de blocs B1, B2, ..., Bt, l'algorithme KTNS couvre un ensemble de k sommets. Cet ensemble peut être "récupéré" à partir de l'algorithme de partitionnement adapté au service par bloc, **et en supposant qu'on connaisse à partir du bloc t+1 la suite de la séquence**, en utilisant la procédure suivante donnée par McGeoch et Sleator pour le service unitaire :
On note $S^*i = S\alpha \cup S\alpha+1 \cup \ldots \cup Si-1 \cup Si$

- $S = S\beta$
- Ordonner les sommets de $S^*\beta-1$ par ordre d'apparition dans la suite de la séquence : le sommet de $S^*\beta-1$ qui apparaît le premier dans la suite de la séquence est placé en premier, et ainsi de suite. Si plusieurs sommets sont appelés dans le même bloc ou bien n'apparaissent plus dans la suite de la séquence, ces sommets sont placés par ordre croissant de leur indice.
- **Pour** chaque sommet v pris dans l'ordre déterminé comme ci-dessus, et jusqu'à ce que S contienne effectivement k sommets, **faire** :

  Ajouter v à S à condition que S ne contienne pas ensuite plus de ki sommets de chaque ensemble S^*i.

Exemple :

Séquence des blocs d'appels : σ = (9, 6, 1), (1, 8), (5, 2, 4), (6, 7), (6), ...

Nombre de serveurs : 5

Etat initial : {6, 7, 8, 9}{1, 2, 3, 4, 5}

{7, 8, 9}	$\{1, 2, 3, 4, 5\}_4$		{6}	
{7, 8, 9}	$\{2, 3, 4, 5\}_3$	{1, 6}		
{7, 8}	$\{2, 3, 4, 5\}_3$	$\{1, 6\}_4$	{9}	
{7, 8}	$\{2, 3, 4, 5\}_3$	$\{\}_2$	{9, 1, 6}.....	Bloc B1
{7}	$\{2, 3, 4, 5\}_3$	$\{\}_2$	$\{9, 1, 6\}_4$	{8}
{7}	$\{2, 3, 4, 5\}_3$	$\{\}_2$	$\{9, 6\}_3$ {1, 8}.....	Bloc B2
{7}	$\{3, 4, 5\}_2$	$\{\}_1$	$\{9, 6\}_2$	{1, 8, 2}
{7}	$\{3, 5\}_1$	$\{\}_0$	$\{9, 6\}_1$	{1, 8, 2, 4}
{7, 3, 5}	$\{9, 6\}_1$	{1, 8, 2, 4}		
{7, 3}	$\{9, 6\}_1$	$\{1, 8, 2, 4\}_4$	{5}	
{7, 3}	$\{9, 6\}_1$	$\{1, 8\}_2$	{2, 4, 5}.....	Bloc B3
...	...			

Recherche des sommets couverts par KTNS après service du deuxième bloc :

Après traitement du deuxième bloc, la partition donne $S^*\beta-1$ ={7, 2, 3, 4, 5, 9, 6}. Parmi ces sommets, c'est le sommet numéro 2 qui est rappelé le plus tôt (au bloc suivant, avec 4 et 5 lexicographiquement), puis le sommet 6 et ainsi de suite, ce qui donne l'ensemble ordonné : {2, 4, 5, 6, 7, 3, 9}.

S qui initialement contient les sommets 8 et 1, est complété d'abord par les sommets 2 et 4. Le sommet 5 ne peut pas être intégré à S, car il y aurait 3 sommets de l'ensemble $S^*\beta-2$ dans S, c'est-à-dire plus de $k\beta-2$ sommets. C'est donc le sommet 6 qui complète S. On obtient ainsi l'ensemble des sommets couverts par KTNS.

Remarque : L'ensemble S obtenu par cette procédure ne contiendra jamais de sommets de $S\alpha$. Cela provient du fait que lorsqu'un sommet v est ajouté à S on s'assure à chaque fois que S ne contient pas plus de ki sommets de chaque ensemble Si ; l'ensemble $S\alpha$ ayant toujours une étiquette nulle, aucun de ses sommets ne peut faire partie de S ce qui est cohérent avec le fait que S contient les sommets couverts par KTNS.

Propriété :
Après chaque bloc d'une séquence d'appels σ, l'ensemble S de k sommets déterminé par la procédure décrite ci-dessus correspond exactement à l'ensemble des sommets couverts par la stratégie optimale KTNS.

Démonstration :

La preuve se fait par induction sur la séquence de blocs. Dans l'état initial, les sommets sont répartis en deux ensembles Sα et Sβ, et par conséquent S égale exactement Sβ qui représente l'ensemble des sommets couverts. La propriété est donc vérifiée avant le premier bloc d'appels dans la séquence. Supposons qu'elle le soit jusqu'au bloc t et montrons qu'elle est encore vraie après le service du bloc t+1. Etant donné la structure de l'algorithme de partitionnement adapté, il faut traiter chaque sommet à l'intérieur du bloc t+1 de manière séquentielle, et on est amené naturellement à distinguer trois cas. Soit v le sommet du bloc t+1 qui doit être servi.

Premier cas :

v fait partie de l'ensemble Sβ. Dans ce cas, l'algorithme ne modifie pas la partition, et donc l'ordre obtenu dans la procédure sur l'ensemble $S^*_{\beta-1}$ est le même : l'ensemble S obtenu est le même que celui obtenu après la dernière requête.

Deuxième cas :

Le sommet v fait partie des sommets non couverts de l'ensemble Sα. Il n'appartient donc pas à S, et par hypothèse de récurrence, il n'est pas couvert par KTNS : KTNS découvre le sommet ne faisant pas partie du bloc t+1 qui est requis au plus tard dans la suite de la séquence. La partition est mise à jour suivant la règle 3 et si il y a lieu la règle 4. Soit S'β le nouvel ensemble Sβ qui est créé ; il ne contient que v. Les sommets du bloc t+1 éventuellement refoulés dans l'ensemble $S'_{\beta-1}$ sont transférés dans l'ensemble S'β. Soit y le nombre de ces sommets. La partition évolue de la façon suivante :

Avant : S_α, $S_{\alpha+1}$, ..., $S_{\beta-1}$, S_β

Après : S'_α, $S'_{\alpha+1}$, ..., $S'_{\beta-2}$, $S'_{\beta-1}$, $S'_\beta = \{ v \}$

Avant application de la règle 4, on a : $S'^*_{\beta-1} = S_\beta \cup S^*_{\beta-1} \setminus \{ v \}$
Soit x la cardinalité de l'ensemble Sβ . On a les égalités suivantes :

$| S'\beta-1 | = x - y$ $\qquad$ $k'\beta-1 = k - y - 1$

$| S'\beta-2 | = | S\beta-1 |$ $\qquad$ $k'\beta-2 = k'\beta-1 = k - x$

S' contient initialement v ainsi que les y sommets transférés dans S'β, soit au total y + 1 sommets. En appliquant la procédure, S' doit être complété avec k – y – 1 sommets issus de S'β–1 dans les limites imposées par les étiquettes, ce qui donne :

- Au plus (k – x) sommets de S'*β–2
- Au moins (x – y – 1) sommets de S'β–1, (sur les k – y – 1 possibles)

L'ordre des sommets de l'ensemble S'*β–2 n'a pas changé, mais il faut maintenant tenir compte dans l'ordonnancement de S'*β–1, des sommets anciennement contenus dans Sβ. La procédure sélectionne parmi tous ces sommets les k – y – 1 réutilisés au plus tôt, c'est-à-dire un de moins sur l'ensemble de ceux sélectionnés pour composer S. Ce sommet découvert sera donc celui qui est requis le plus tard parmi les sommets de S'*β–1, ce qui donne :

- (k – x – 1) sommets de S'*β–2 }
et } si le sommet requis au plus tard fait partie de S'*β–2
- (x – y) sommets de S'β–1 }

Ou bien

- (k – x) sommets de S'*β–2 }
et } si le sommet requis au plus tard fait partie de S'β–1
- (x – y – 1) sommets de S'β–1 }

Conclusion :

L'ensemble S' est le même que celui obtenu après le dernier appel précédant l'appel du sommet v, à un sommet près : le sommet de S, (ne faisant pas partie du bloc t+1), requis au plus tard dans la suite de la séquence est remplacé par v, ce qui est exactement le choix opéré par KTNS.

Troisième cas :

Le sommet v appartient à un ensemble Si de la partition, avec $\alpha < i < \beta$. Il est transféré dans Sβ ce qui suppose naturellement que Sβ est tel que $| S\beta | < k$, donc que S contient au moins

un sommet de $S^*\beta-1$. Parmi les sommets de $S^*\beta-1$, v est le premier dans l'ordre déterminé par la procédure, donc v appartient à l'ensemble S. Par hypothèse de récurrence, v fait partie des sommets déjà couverts par KTNS. Pour répondre à cette demande du bloc t + 1, KTNS ne déplace donc aucun serveur. Montrons que de même S n'est pas modifié par la procédure :

Suivant la procédure, S contient v appartenant à un ensemble S_i et pas plus de k_j sommets des ensembles S^*j, $\forall$ j tel que $i \leq j \leq \beta-1$. L'ordre déterminé précédemment sur chaque S^*j est inchangé. Or après application de la règle 2 on a : $k'j = kj - 1$ pour $i \leq j \leq \beta-1$. Donc, la procédure n'inclue dans S' pas plus de sommets de chaque ensemble S^*j que pour le traitement de la requête précédente, et l'ordre dans chacun de ces ensembles n'étant pas modifié, elle sélectionne exactement les mêmes sommets que ceux qui ont été sélectionnés pour composer S. On peut donc conclure que S et S' contiennent les mêmes sommets.

Conclusion : L'ensemble S' de sommets obtenus après service de la demande v, (quel que soit l'ensemble auquel ce sommet appartient), correspond bien à l'ensemble des sommets couverts par KTNS. Le même raisonnement est appliqué pour chaque requête du bloc t+1 en traitement séquentiel, en supposant que KTNS traite également chaque demande du bloc t + 1 une par une, dans le même ordre que l'algorithme de partitionnement.

Remarque : Cette propriété nous montre que KTNS n'enregistre un déplacement supplémentaire que pour les requêtes issues de l'ensemble $S\alpha$. A partir de la partition, on peut donc conclure que le coût de l'algorithme KTNS est exactement égal au nombre d'appels $D(\sigma)$ provenant des sommets de $S\alpha$: $C_{ktns}(\sigma) = D(\sigma)$.

On en déduit également que tout algorithme on-line A de service par bloc, a un coût au moins égal à $D(\sigma)$, quelle que soit la séquence σ : $C_A(\sigma) \geq D(\sigma)$

5. 3. Compétitivité et forte compétitivité du service par blocs

5. 3. 1. Compétitivité de l'algorithme adapté au service par blocs

L'algorithme PART est un algorithme aléatoire pour le problème de k-serveurs uniforme avec service par bloc. Ainsi que nous l'avons décrit, cet algorithme directement inspiré de celui proposé par McGeoch et Sleator, maintient une partition des sommets à l'aide d'un système d'étiquettes et de marquage des sommets qui est mis à jour suivant les règles 1, 2 , 3, et 4. Dans tous les cas, les marques sont redéployées de façon aléatoire, uniformément parmi les sommets susceptibles de recevoir ces marques. Les sommets susceptibles de recevoir la marque i sont les sommets de Si, ainsi que les sommets portant la marque i-1, soit au total $k_{i-1} + |S_i|$ sommets.

Les formules de récurrence entre ki et ki-1 étant toujours respectées lors du service des blocs par l'algorithme de partitionnement, on a : ki-1 + | Si |= ki + 1. Il y a donc ki + 1 façons d'arranger la marque i, que ce soit après l'application de la règle 2, ou des règles 3 et 4.

Comme l'ont montré McGeoch et Sleator, ces ki + 1 arrangements ont tous la même probabilité de se produire quelle que soit la marque i, ce qui permet de calculer une borne pour la probabilité qu'un serveur soit déplacé lors de l'application de la règle 2. Cette borne est donnée dans le lemme suivant :

Lemme :

Lorsqu'un sommet v appartenant à un ensemble Si appelle, (avec $\alpha < i < \beta$), la probabilité que l'algorithme de partitionnement déplace un serveur sur v est bornée par :

$$\Sigma \ (1/ k_j + 1, \ i \leq j < \beta)$$

Démonstration :

La règle 2 n'ayant pas été modifiée dans l'adaptation de l'algorithme de partitionnement, le déplacement des marques i, i+1, ... jusqu'à β–1 se fait de la même façon. La preuve est donc identique à celle donnée pour le service unitaire : La marque β–1 peut être déplacée au cours du déplacement de la marque i, ou i+1, et ainsi de suite jusqu'à la marque β–1 elle-même, si quel que soit j, ($i \leq j < \beta$), la marque j n'est pas présente après déplacement de la marque j–1 sur le sommet v. La probabilité que le sommet v reçoive la marque β–1 lors du déplacement de la marque j est égale à la probabilité que v n'ait pas la marque j, c'est-à-dire $1/(k_j + 1)$, et ainsi de suite, ce qui donne :

P[β–1 est déplacée] = P[(β–1 est déplacée en même temps que i) ou
(β–1 est déplacée en même temps que i+1) ou
...
(β–1 est déplacée en même temps que β–2) ou
(β–1 est déplacée sur v déjà marqué par β–2)]

D'où :

$$P[\beta\text{–}1 \text{ est déplacée}] \leq \Sigma \ (P[v \text{ ne possède pas la marque } j], \ i \leq j < \beta)$$
$$= \Sigma \ (1/(k_j + 1), \ i \leq j < \beta)$$

En fait, cette probabilité peut facilement être calculée exactement :

P[β–1 est déplacée] = 1– P[v est déjà marqué par β–1]

Si le sommet v appartenant à l'ensemble Si, avec $i \leq j < \beta$, possède déjà la marque β–1, c'est qu'il possède la marque β–2, donc nécessairement qu'il possède la marque β–3, et ainsi de suite jusqu'à la marque i. La probabilité que v possède la marque j est : $k_j/(k_j + 1)$. Ces probabilités étant indépendantes, la probabilité que v possède toutes les marques de i à β–1 est

$$1 - \Pi (k_j/(k_j + 1),\ i \leq j < \beta)$$

A l'aide de cette probabilité de déplacement d'un serveur, notre but est d'établir pour l'algorithme de partitionnement adapté au service par bloc, un coefficient de compétitivité. Dans la suite, cet algorithme sera désigné par PART. Dans un premier temps, nous nous attachons à montrer que l'algorithme avec les modifications que nous lui avons apportées, conserve le coefficient de compétitivité Hk qui est celui de l'algorithme de McGeoch et Sleator. Ce coefficient n'est pas forcément le plus petit possible pour le service par bloc, mais il nous permet en tout cas d'en déterminer une borne supérieure. Dans le paragraphe suivant, nous cherchons donc à établir l'inégalité suivante, pour toute séquence de blocs σ, et pour tout algorithme on-line B de service par bloc :

$$C_{PART}(\sigma) \leq Hk\ .\ C_B(\sigma) + a\ ,\ \text{où a est une constante.}$$

Pour montrer cette inégalité, nous suivrons point par point la démonstration de McGeoch et Sleator pour l'algorithme de partitionnement, en montrant en quoi le service par bloc apporte dans certains cas des compléments d'informations et donc des différences, ou au contraire se comporte comme l'algorithme de service à l'unité.

Théorème :
L'algorithme de partitionnement adapté au service par bloc PART est un algorithme Hk–compétitif pour le problème uniforme des k-serveurs avec service par bloc.

Démonstration :

On définit la fonction $\phi = \Sigma (H_{k_i + 1} - 1,\ \alpha \leq i < \beta)$. On montre par récurrence sur chaque sommet de la séquence σ pris de façon séquentielle, l'inégalité suivante :

$$C_{PART}(\sigma) + \phi \leq Hk\ .\ D(\sigma)$$

où $D(\sigma)$ représente comme nous l'avons défini plus haut le nombre d'appels provenant de l'ensemble $S\alpha$, en fonction de la partition qui a été obtenue à ce stade de développement de la séquence σ.

Au départ, c'est-à-dire avant le premier appel, les coûts $C_{PART}(\sigma)$ et $D(\sigma)$ sont nuls, et ϕ est également nul car dans l'état initial de la partition, toutes les étiquettes ki sont nulles. L'inégalité est donc vérifiée. Supposons maintenant que l'inégalité est vérifiée jusqu'à ce que un sommet v appelle, et montrons qu'une fois le sommet v servi par PART, elle l'est encore. Comme pour chaque propriété de l'algorithme de partitionnement, il faut naturellement distinguer trois cas, suivant l'ensemble auquel appartient le sommet v. Dans le premier cas, qui est toujours le plus simple, v appartient à $S\beta$. $D(\sigma)$ n'augmente pas, et la partition n'est pas modifiée, donc aucune quantité n'est modifiée dans l'inégalité.

Si le sommet v appartient à un ensemble Si, avec $\alpha < j < \beta$, $D(\sigma)$ n'augmente pas, mais la partition évolue, (v est transféré dans $S\beta$), et en particulier les étiquettes kj sont décrémentées de 1 pour $i \leq j < \beta$. De ce fait, si une des étiquettes kj devient nulle, le nombre d'ensembles de la partition diminue, et α devient α'. La valeur de la fonction ϕ va donc changer : soit $\Delta\phi$ la différence entre la nouvelle valeur de la fonction et l'ancienne.

$$\begin{aligned}
\Delta\phi &= \Sigma\,(H_{k'j+1} - 1,\ \alpha' \leq j < \beta) - \Sigma\,(H_{kj+1} - 1,\ \alpha \leq j < \beta) \\
&\leq \Sigma\,(H_{k'j+1} - 1,\ \alpha \leq j < \beta) - \Sigma\,(H_{kj+1} - 1,\ \alpha \leq j < \beta) \\
&= \Sigma\,(H_{k'j+1} - H_{kj+1},\ \alpha \leq j < \beta)
\end{aligned}$$

Pour $j < i$, on a $k'j = kj$, d'où $H_{k'j+1} = H_{kj+1}$

Pour $i \leq j < \beta$, on a $k'j = kj - 1$, d'où $H_{k'j+1} = H_{kj} = H_{kj+1} - (1 / kj + 1)$, ce qui donne

$$\Delta\phi \leq -\Sigma\,((1 / kj + 1),\ i \leq j < \beta)$$

La fonction ϕ décroît donc, tandis que le coût de PART augmente de une unité avec une probabilité bornée par $\Sigma\,(1/kj + 1,\ i \leq j < \beta)$, comme l'indique le lemme précédent. Donc l'inégalité est toujours vérifiée, et en fait PART se comporte comme l'algorithme de partitionnement pour le service par bloc.

Dernier cas : v fait partie de l'ensemble $S\alpha$ et donc n'est pas couvert. $D(\alpha)$ augmente de une unité, et d'autre part, la partition étant également modifiée, la valeur de la fonction ϕ change. Soit $C'_{PART}(\sigma)$ le nouveau coût de l'algorithme de partitionnement à la suite du service de v.

On a maintenant :

$$C'_{PART}(\sigma) = C_{PART}(\sigma) + 1$$

Le sommet v est transféré dans l'ensemble de tête, et il constitue à lui seul le nouvel ensemble Sβ. On a alors $k\beta-1 = k - 1 - y$, (où y est le nombre de sommets contenus dans l'ancien ensemble Sβ et qui font partie du même bloc que v), tandis que les autres étiquettes ne changent pas. La fonction ϕ prend la valeur

$$\phi' = \Sigma \, (H_{k'i+1} - 1, \ \alpha \leq i < \beta')$$

$$= \phi + H_{k\beta-1+1} - 1$$

$$= \phi + H_{k-y} - 1$$

Et l'inégalité devient

$$C'_{PART}(\sigma) + \phi' = C_{PART}(\sigma) + 1 + \phi + H_{k-y} - 1$$

$$\leq Hk \, . \, D(\sigma) + H_{k-y}$$

Or soit b(v) la cardinalité du bloc auquel appartient v. On a $0 \leq y \leq b(v)$ donc

$$0 \leq k - b(v) \leq k - y \leq k.$$

Par conséquent, $H_{k-y} \leq Hk$,
d'où

$$C'_{PART}(\sigma) + \phi' \leq Hk \, . \, [D(\sigma) + 1] = Hk \, . \, D'(\sigma)$$

Ce dernier cas, montre que lorsque le service se fait par blocs, l'accroissement de la fonction ϕ est plus faible, ce qui laisse penser que le coefficient de compétitivité peut être inférieur à Hk pour le service par bloc. L'inégalité permet cependant de montrer la Hk compétitivité de PART :

On a $C_{PART}(\sigma) + \phi \leq Hk \, . \, D(\sigma)$ donc pour tout algorithme on-line B de service par blocs, et sur toute séquence de blocs σ :

$$C_{PART}(\sigma) \leq Hk \, . \, D(\sigma) - \phi \leq Hk \, . \, C_B(\sigma) - \phi \leq Hk \, . \, C_B(\sigma)$$

car ϕ est toujours positive ou nulle, ce qui montre par définition que PART est Hk compétitif.

5. 3. 2. Forte compétitivité

Fiat, Karp, Luby, McGeoch, Sleator et Young (1991) ont montré qu'un algorithme on–line pour le problème de k-serveurs uniforme ne peut pas avoir un coefficient de compétitivité inférieur strictement à Hk : pour tout algorithme A, ils indiquent comment construire une séquence σ sur laquelle $C_A(\sigma) \geq Hk \, . \, D(\sigma)$. Pour le problème de k-serveurs uniforme avec service par blocs, on admet que plusieurs sommets demandent à être servis en même temps : à chaque instant, on ignore la composition du prochain bloc ainsi que le nombre de sommets qui le composent. Ce nombre peut varier a priori entre 1 et k ; de plus ce nombre varie d'un bloc à l'autre dans une même séquence. On peut donc avoir une séquence constituée entièrement de blocs de k sommets : dans ce cas, tous les algorithmes de gestion on-line des serveurs par blocs sont équivalents car tous les serveurs sont mobilisés à chaque instant, et leur affectation sur les sommets n'a donc plus d'importance. De même, l'extrême inverse peut aussi bien se présenter : une séquence composée uniquement de blocs de un sommet.

Soit B un algorithme on-line pour le service par bloc, dont le coefficient de compétitivité est c. Si on applique B à la séquence construite par Fiat et al (1991), c'est-à-dire à une séquence dont les blocs se réduisent à un appel unique, on obtient l'inégalité $c \geq Hk$.

Par conséquent, étant donné que l'algorithme de partitionnement PART adapté au service par bloc est Hk-compétitif, on peut conclure qu'il est également fortement compétitif, dans la mesure où aucune restriction n'est imposée sur la structure de la séquence et les blocs qui la composent, ce qui implique que le cas des séquences unitaires doit être pris en compte. On peut toutefois être plus précis et définir une séquence d'appels par blocs comme étant composée de groupes d'appels d'au moins deux sommets. Dans le paragraphe suivant, nous étudions un tel cas en considérant les séquences composées entièrement de blocs de deux sommets.

5. 4. Cas des séquences composées de blocs de deux appels

5. 4. 1. Forte compétitivité

En suivant l'algorithme proposé par Fiat et al (1991), (légèrement modifié pour s'adapter au service par bloc), nous allons construire une séquence σ entièrement composée de blocs de deux appels, sur laquelle quel que soit l'algorithme A de service par bloc pour le problème des (n-1)-serveurs, on a : $C_A(\sigma) \geq Hn\text{-}2 \, . \, D(\sigma)$.

En supposant ensuite que pour un nombre de serveurs quelconque k, seuls k+1 sommets du graphe sont utilisés pour construire la séquence, (sur les n formant le réseau), on obtient le résultat suivant :

Proposition :
Pour le service d'une séquence de blocs de cardinalité 2, il n'existe pas d'algorithme on-line dont le coefficient de compétitivité soit inférieur strictement à Hk-1.

Démonstration :

Comme pour le service unitaire, on doit construire une séquence sans connaître à chaque instant la position des serveurs sur le réseau : ceci pour tenir compte du fait que l'algorithme A est un algorithme de service par bloc "aléatoire", (voir Fiat et al (1991)). On peut toutefois s'aider d'un vecteur de probabilités pi qui indique pour chaque sommet i la probabilité qu'il ne soit pas couvert. Ce vecteur est mis à jour à chaque requête dans la séquence, en simulant le comportement de A sur tous les arrangements de serveurs possibles à un instant donné. A l'intérieur de la séquence, les blocs seront toujours constitués d'un appel du dernier sommet servi, puis d'un appel d'un autre sommet, c'est-à-dire un appel d'un sommet couvert et un appel d'un sommet i ayant une probabilité pi >0.

La séquence est constituée de phases sur lesquelles le coût de l'algorithme A est d'au moins Hn-2, tandis que celui de KTNS est exactement 1. Pour définir ces phases, au fur et à mesure de leurs appels, les sommets sont marqués, (s'ils ne le sont pas déjà). Une nouvelle phase commence quand n-1 sommets sont déjà marqués, et qu'un sommet i non marqué doit être servi, toutes les marques sont effacées, sauf sur le (n-1)-ième sommet marqué, tandis que le sommet i reçoit une marque. Cette mise à jour des marques permet de diviser la séquence en sous-phases : chaque nouveau sommet marqué indique le début d'une nouvelle sous-phase, qui peut être constituée de plusieurs blocs de deux appels. Le nombre de sommets non marqués diminue donc de n-2 à 1, puis repasse à n-2 pour constituer une nouvelle phase de la séquence.

Soit S l'ensemble des sommets marqués, et u le nombre de sommets non marqués. On construit les blocs de façon à induire sur A un coût supérieur ou égal à 1/u, et à marquer à chaque fois un nouveau sommet. De cette façon, u prenant successivement toutes les valeurs entre n-2 et 1, l'algorithme A aura un coût au moins égal à (1/n-2) + (1/n-3) + ... + (1/2) + 1 = Hn-2.

Etant donné qu'une nouvelle phase de la séquence commence lorsqu'un $n^{ième}$ sommet va être marqué, si u = n-2 les deux sommets marqués constituent le dernier bloc d'appels. Ces deux sommets sont donc couverts quel que soit l'algorithme de service A.

Soit $P = \Sigma$ (pi, i $\in$ S). On a donc P = 0, et il existe nécessairement un sommet non marqué i tel que pi $\geq$ (1/u). Le bloc suivant est donc constitué d'une part du sommet marqué lors de la dernière sous-phase, (et on a toujours P = 0 car le vecteur des probabilités n'est pas modifié), et d'autre part du sommet i. Sur cette sous-phase le coût de A est d'au moins 1/(n-2)

car le premier appel du bloc ne nécessite pas de déplacement de serveur. Le nombre de sommets non marqués passe à n-3, et P peut être maintenant strictement positif car A peut avoir découvert un sommet marqué pour servir i. Comme toujours le bloc suivant contient i qui est déjà couvert ce qui n'entraîne pas de coût supplémentaire, et un autre sommet dont le choix dépend de P : Si P = 0, on choisit le deuxième sommet comme pour le bloc précédent. Sinon, si P > 0, alors il existe au moins un sommet v tel que $p_v > 0$. Le sommet v constitue le deuxième appel du bloc, et on utilise l'algorithme de Fiat et al (1991) pour créer les blocs suivant :

| Soit $p_v = \varepsilon$.
| **Tant que** $P > \varepsilon$ **et** que le coût de la sous-phase n'a pas atteint au moins 1/u **faire**
| **Créer** un nouveau bloc contenant le dernier sommet appelé
| **Choisir** comme $2^{ième}$ sommet le sommet i tel que : $p_i = \text{Max}\{p_j, j \in S\}$

A chaque passage dans la boucle, on a $P > \varepsilon$ c'est-à-dire $\Sigma (p_j, j \in S) > \varepsilon$, d'où $|S| \cdot \text{Max}\{p_j, j \in S\} > \varepsilon$. Le coût de la sous phase augmente donc à chaque fois d'un coût $p_i > \varepsilon / |S|$, ce qui prouve que l'algorithme termine dans le pire des cas après un appel de chaque sommet marqué. Quand on sort de la boucle, deux situations peuvent se présenter :

- Si l'algorithme termine avec un coût total au moins égal à 1/u, alors on termine la sous-phase par un appel du dernier sommet appelant, et un appel d'un sommet non marqué quelconque. Dans ce cas, le coût de A sur le sous-phase est clairement supérieur ou égal à 1/u.

- Si le coût de la sous-phase n'a pas atteint au moins 1/u, on a $P \leq \varepsilon$. On procède comme pour les autres blocs à un appel du dernier sommet couvert, puis comme l'indiquent Fiat et al (1991), on choisit le sommet j non marqué qui a la plus forte probabilité d'être découvert. La probabilité qu'un sommet non marqué soit non couvert sachant que l'unique sommet découvert est un sommet non marqué est d'au moins 1/u, ce qui donne $p_j \geq (1-P).(1/u)$. En effet, on a :

p_j = P(j est non couvert)
= P(j non couvert et $\forall$ i$\in$ S, i est couvert)
= P(j non couvert / $\forall$ i$\in$ S, i est couvert) . P($\forall$ i$\in$ S, i est couvert)
= P(j non couvert / $\forall$ i $\in$ S, i est couvert) . (1-P)
$\geq$ (1/u) . (1-P)

Le coût total enregistré par l'algorithme A sur cette sous-phase est au moins égal à $\varepsilon + p_j$, ce qui induit un coût supérieur ou égal à 1/u sur cet ensemble de blocs constituant la sous-phase en effet, on a :

$$\varepsilon + p_j \geq \varepsilon + (1/u).(1-P) \geq \varepsilon + (1/u).(1-\varepsilon) \geq 1/u$$

Sur la totalité de la phase, l'algorithme A atteint un coût au moins égal à Hn-2 tandis que, l'algorithme off-line optimal déplace exactement un serveur : en effet, la phase comprend n-2 appels sur n-2 sommets non marqués donc différents. Sur ces n-2 sommets, il n'y a que n-3 serveurs, car deux serveurs sont déjà placés sur les deux sommets marqués en début de phase. Un appel parmi les n-2 proviendra d'un sommet non servi, donc l'algorithme optimal aura forcément un coût au moins égal à 1. Comme l'algorithme optimal connaît à l'avance les n-2 prochains appels, pour servir le sommet découvert, il déplace le serveur se trouvant sur le sommet qui est rappelé au plus tard parmi les n-1 sommets couverts. Son coût est donc exactement égal à 1 comme sur chaque phase de la séquence, et on a $C_A(\sigma) \geq H_{n-2} \, . \, D(\sigma)$.

5. 4. 2. Compétitivité de l'algorithme PART

Hn-2 constitue donc une borne inférieure du coefficient de compétitivité sur les séquences constituées de blocs de deux appels. Cette borne inférieure du coefficient de compétitivité pour le problème de k-serveurs avec service par blocs de deux sommets amène naturellement à examiner la compétitivité de l'algorithme PART sur une séquence constituée uniquement de blocs de cardinalité 2. Dans le paragraphe suivant, nous montrons que pour toute séquence σ, on a :

$$C_{PART}(\sigma) + \psi \leq H_{k-1} \, . \, D(\sigma)$$

avec $\psi = \Sigma \, (H_{k_i} - 1, \; \alpha < i < \beta)$

Cette inégalité est montrée comme précédemment par induction sur chaque requête de la séquence. Dans l'état initial, elle n'a pas de sens puisque ψ n'est définie que pour $i > \alpha$, donc lorsqu'au moins un bloc de deux appels (s1 et s2), a été servi par l'algorithme, et a entraîné la création d'au moins un nouvel ensemble dans la partition.

Vérification sur le premier bloc : Ce bloc peut être constitué de deux façons : soit il contient deux sommets provenant de l'ensemble $S\alpha$, soit il contient un sommet appartenant à $S\alpha$ et un sommet appartenant à $S\beta$. Le cas où les deux sommets appartiendraient tous deux à $S\beta$ n'est pas considéré, car les deux sommets étant alors déjà couverts dans l'état initial, la partition ne serait pas modifiée.

Dans le premier cas $\{s1, s2\} \subset S$, l'algorithme suivant la règle 3 crée d'abord un nouvel ensemble $S\beta$ réduit à s1, et le coût de l'algorithme off-line optimal ainsi que de l'algorithme PART (initialement nuls), passent à 1 :

$D'(\sigma) = D(\sigma) + 1 = 1$

$C'_{PART}(\sigma) = C_{PART}(\sigma) + 1 = 1$

On a d'autre part

$$\psi = H_{k\alpha+1} - 1 = H_{k\beta-1} - 1 = H_{k-1} - 1$$

D'où

$$\begin{aligned} C'_{PART}(\sigma) + \psi &= C_{PART}(\sigma) + 1 + H_{k-1} - 1 \\ &= H_{k-1} \\ &= H_{k-1} \cdot D'(\sigma) \end{aligned}$$

Pour le sommet s2, un quatrième ensemble est construit, et après application des règles 3 et 4, l'ensemble $S\beta$ est constitué de s1 et s2 : on a $k\beta-1 = k-2$ et $k\beta-2 = k\alpha+1 = k-1$, tandis que les coûts $C_{PART}(\sigma)$ et $D(\sigma)$ ont encore augmenté de 1. On a donc :

$$\begin{aligned} C''_{PART}(\sigma) + \psi &= 2 + (H_{k-1} - 1) + (H_{k-2} - 1) \\ &\leq 2 \cdot H_{k-1} \\ &= H_{k-1} \cdot D''(\sigma) \end{aligned}$$

Après traitement des deux sommets s1 et s2, l'inégalité est donc vérifiée. Dans le deuxième cas, ($s1 \in S\beta$ et $s2 \in S\alpha$), la démonstration est analogue : c'est le sommet appartenant à $S\beta$ dans l'état initial, qui est traité en premier, car il est déjà couvert. Son traitement n'entraîne ni modification de la partition, ni coût supplémentaire pour les algorithmes. Le sommet s2 étant non couvert, $C_{PART}(\sigma)$ et $D(\sigma)$ augmentent d'une unité, et après application des règles 3 et 4, l'ensemble $S\beta$ contient s1 et s2, tandis que $k\alpha+1 = k\beta-1 = k-2$. On a donc :

$$\begin{aligned} C'_{PART}(\sigma) + \psi &= 1 + H_{k-2} - 1 \\ &= H_{k-2} \cdot 1 \\ &\leq H_{k-1} \cdot D'(\sigma) \end{aligned}$$

L'étude de ces deux cas montre que quel que soit le premier bloc de la séquence, tel que l'on sorte de l'état initial, l'inégalité est vérifiée. Supposons qu'elle le soit maintenant après service de plusieurs appels, et jusqu'à ce que un sommet v se manifeste, et étudions les deux cas où la partition et les coûts sont modifiés :

Premier cas : $v \in Si$, avec $\alpha < i < \beta$

La partition ainsi que les étiquettes évoluent suivant la règle 2, et on a :

$$\Delta \psi = \Sigma (H_{k'_j} - 1,\ \alpha' < j < \beta) - \Sigma (H_{kj} - 1,\ \alpha < j < \beta)$$

$$\leq \Sigma (H_{k'_j} - 1,\ \alpha < j < \beta) - \Sigma (H_{kj} - 1,\ \alpha < j < \beta)$$

$$= \Sigma (H_{k'_j} - H_{kj},\ \alpha < j < \beta)$$

Pour $j < i$, on a $k'_j = k_j$, d'où $H_{k'_j} = H_{kj}$

Pour $i \leq j < \beta$, on a $k'j = kj - 1$, d'où $H_{k'_j} = H_{kj-1} = H_{kj} - (1 / k_j)$

ce qui donne,

$$\Delta \psi \leq - \Sigma ((1 / k_j),\ i \leq j < \beta)$$

D'autre part, le coût de l'algorithme off-line optimal n'augmente pas, tandis que l'algorithme PART enregistre un coût supplémentaire : ce coût est borné comme nous l'avons vu dans le lemme par : $\Sigma (1/(k_j + 1),\ i \leq j < \beta)$. On a donc

$$C'_{PART}(\sigma) + \psi' \leq C_{PART}(\sigma) + \Sigma (1/ k_j + 1,\ i \leq j < \beta) + \psi - \Sigma ((1 / k_j),\ i \leq j < \beta)$$

$$\leq C_{PART}(\sigma) + \psi$$

$$\leq H_{k-1} . D(\sigma)$$

$$= H_{k-1} . D'(\sigma)$$

Ce qui montre que l'inégalité est vérifiée après tout appel d'un sommet appartenant à un ensemble Si, avec $\alpha < i < \beta$.

Deuxième cas : $v \in S\alpha$

Le sommet v étant découvert, les coûts $C_{PART}(\sigma)$ et $D(\sigma)$ augmentent de une unité. Après application de la règle 3, le sommet v est transféré dans l'ensemble de tête, et si la règle 4 n'est pas appliquée, il constitue à lui seul le nouvel ensemble $S\beta$. Dans le cas contraire, $S\beta$ contient les deux sommets du bloc auquel appartient v, et $k\beta-1 = k - 2$. Les autres étiquettes ne changent pas.

La fonction ψ devient :

$$\begin{aligned}\psi' &= \Sigma\,(\,H_{k'_i} - 1,\ \alpha < i < \beta'\,)\\ &= \psi + H_{k\beta-1} - 1\\ &\leq \psi + H_{k-1} - 1 \quad \text{car } H_{k-2} \leq H_{k-1}\end{aligned}$$

Et l'inégalité devient

$$\begin{aligned}C'_{PART}(\sigma) + \psi' &\leq C_{PART}(\sigma) + 1 + \psi + H_{k-1} - 1\\ &\leq H_{k-1} \,.\, D(\sigma) + H_{k-1}\\ &= H_{k-1} \,.\, D'(\sigma)\end{aligned}$$

Le cas où v fait partie de l'ensemble $S\beta$ est trivial, car aucun coût supplémentaire n'est induit sur les algorithmes PART et off-line optimal, et la partition n'est pas modifiée.

Cette inégalité montre la H_{k-1} compétitivité de l'algorithme de partitionnement pour le service par blocs de deux sommets. Cet algorithme est donc fortement compétitif sur ce type de séquence.

5. 5. Généralisation aux blocs d'au moins deux appels

5. 5. 1. Compétitivité de l'algorithme PART

Le coefficient de compétitivité que nous avons établi pour le cas b = 2, laisse penser que pour des blocs de taille b plus grande, ce coefficient peut être inférieur à H_k. Dans le paragraphe suivant, nous cherchons à établir la H_{k-b+1} compétitivité de l'algorithme de partitionnement, lorsque l'on sait que la séquence σ est entièrement constituée de blocs de taille b, avec $b \geq 2$.

Soit ξ la fonction : $\xi = \Sigma\ (\ H_{ki-b+2} - 1,\ \ \alpha < i < \beta\ )$. Cette fonction est toujours positive ou nulle, et elle n'est définie que lorsqu'un premier sommet du premier bloc d'appels entraîne la création d'un ensemble supplémentaire dans la partition. Montrons que cette fois, la partition établie par l'algorithme vérifie l'inégalité suivante :

$$C_{PART}(\sigma) + \xi \leq H_{k-b+1} \,.\, D(\sigma)$$

Supposons que l'inégalité est vraie après le premier bloc de b appels, (ce que nous vérifierons plus loin), et après plusieurs appels dans la séquence, et montrons qu'un nouvel appel d'un sommet v ne la met pas en défaut :

Si v appartient à un ensemble Si, avec $\alpha < i < \beta$, la fonction ξ varie de $\Delta\xi$ et on a

$$\Delta\xi \leq \Sigma\ (H_{k'j-b+2} - H_{kj-b+2},\ \alpha < j < \beta)$$
$$= -\Sigma\ (1/(kj-b+1),\ i \leq j < \beta)$$

On a $D'(\sigma) = D(\sigma)$, tandis que le coût supplémentaire enregistré par l'algorithme PART est toujours borné par :

$$\Sigma\ (1/(k_j + 1),\ i \leq j < \beta).$$

On a donc

$$C'_{PART}(\sigma) + \xi' \leq C_{PART}(\sigma) + \Sigma\ (1/(k_j + 1), i \leq j < \beta) + \xi - \Sigma\ (1/(kj-b+1), i \leq j < \beta)$$
$$\leq C_{PART}(\sigma) + \xi$$
$$\leq H_{k-b+1} \cdot D'(\sigma)$$

Si on a maintenant $v \in S\alpha$, les coûts $C_{PART}(\sigma)$ et $D(\sigma)$ augmentent de une unité. Après application de la règle 3, le sommet v est transféré dans l'ensemble de tête, et si la règle 4 n'est pas appliquée, il constitue à lui seul le nouvel ensemble $S\beta$: on a $k_{\beta-1} = k - 1$. Dans le cas contraire, $S\beta$ contient un ou plusieurs sommets du bloc auquel appartient v, ces sommets ayant déjà été servis : l'étiquette $k_{\beta-1}$ est donc comprise entre k - 2 et k - b + 1, tandis que les autres étiquettes ne changent pas. On a donc

$$\xi' = \xi + H_{(k\beta-1) - b + 2} - 1$$
$$\leq \xi + H_{k - b + 1} - 1 \quad \text{car } k_{\beta-1} \leq k\text{-}1$$

L'inégalité devient

$$C'_{PART}(\sigma) + \xi' \leq C_{PART}(\sigma) + 1 + \xi + H_{k - b + 1} - 1$$
$$\leq H_{k - b + 1} \cdot [D(\sigma) + 1] = H_{k - b + 1} \cdot D'(\sigma)$$

ce qui montre que l'inégalité est toujours vérifiée. Il reste à examiner ce que devient cette inégalité lors du premier appel de la séquence permettant de sortir de l'état initial.

Cet appel doit provenir d'un sommet de l'ensemble $S\alpha$. Il peut avoir été précédé de un ou plusieurs appels de sommets provenant de l'ensemble $S\beta$: Il y a exactement b types de blocs pouvant faire évoluer la partition hors de l'état initial. Tous les sommets sont contenus dans $S\alpha$, ou bien x sommets appartiennent à $S\alpha$ et b-x sont dans $S\beta$, $(1 \leq x < b)$. Si tous les sommets du bloc sont contenus dans l'ensemble $S\alpha$ donné par l'état initial, $C_{PART}(\sigma)$ et $D(\sigma)$ passent de 0 à b, car tous les sommets sont découverts. La partition passe de deux ensembles à b + 2 ensembles après applications successives des règles 3 et 4 pour chacun des b sommets du bloc :

$S\alpha, S\alpha+1, \ldots, S\alpha+b, S\beta$ avec $S\beta = \{s1, s2, \ldots, sb\}$

On a $k\alpha+1 = k-1$

$k\alpha+2 = k-2$

$\ldots$

$k\beta-2 = k-b-1$

$k\beta-1 = k-b$

d'où $\xi = \Sigma\ ((H_{ki-b+2} - 1),\ \alpha < i < \beta) = H_{k-b+1} + H_{k-b} + H_{k-b-1} + \ldots + H_{k-2b+2} - b$

$$\begin{aligned} C'_{PART}(\sigma) + \xi &= b + (H_{k-b+1} + H_{k-b} + H_{k-b-1} + \ldots + H_{k-2b+2}) - b \\ &\leq b \cdot H_k - b + 1 \\ &= H_{k-b+1} \cdot D'(\sigma) \end{aligned}$$

De même, si le premier bloc est constitué de x sommets dans $S\alpha$ et b-x dans $S\beta$, $(1 \leq x < b)$, les b-x sommets de $S\beta$ étant déjà couverts, ils sont traités en premier, puis le service des x autres entraîne la création de x ensembles supplémentaires. Le coût de l'algorithme PART et de l'algorithme off-line optimal passent de 0 à x et on a :

$$\xi = H_{k-2b+x+1} + H_{k-2b+x} + H_{k-2b+x-1} + \ldots + H_{k-2b+2} - x$$

$$\begin{aligned} C'_{PART}(\sigma) + \xi &= x + (H_{k-2b+x+1} + H_{k-2b+x} + H_{k-2b+x-1} + \ldots + H_{k-2b+2}) - x \\ &\leq x \cdot H_{k-b+1} \\ &= H_{k-b+1} \cdot D'(\sigma) \end{aligned}$$

Conclusion : Quelle que soit la forme du premier bloc dans la séquence permettant de sortir de l'état initial, l'inégalité $C_{PART}(\sigma) + \xi \leq H_{k-b+1} \cdot D(\sigma)$ est vérifiée après service de ce bloc par l'algorithme de partitionnement. L'étude de ce dernier cas complète la démonstration de la H_{k-b+1} compétitivité de PART.

5. 5. 2. Une borne inférieure du coefficient de compétitivité

Afin d'en déduire que PART est également fortement compétitif pour le service on-line de blocs de cardinalité $b \geq 1$, on doit reprendre comme pour l'étude des blocs de deux sommets, la méthode de construction d'une séquence particulière proposée par Fiat et al (1991), lorsque le nombre de serveurs est égal à n-1. Sur cette séquence, PART doit avoir un coût au moins égal à H_{n-b}, tandis que l'algorithme off-line optimal a un coût exactement égal à 1.

Pour l'étude des blocs à deux sommets, on a rajouté, (avant chaque nouvel appel déterminé par l'algorithme de construction), un appel du dernier sommet marqué et couvert. On rajoute cette fois b-1 appels des b-1 derniers sommets couverts et déjà marqués, pour définir des blocs de b sommets. Pour servir chaque nouveau sommet non couvert, l'algorithme A étant un algorithme de service par blocs, les b-1 sommets faisant partie du bloc et qui ont déjà été servis ne sont pas découverts.

Une phase commence avec b sommets marqués, et se termine quand un (n-b)-ième sommet non marqué appel. Le nombre de sommets non marqués prend toutes les valeurs de n-b à 1, et chaque nouveau sommet non marqué est choisi suivant la méthode donnée par Fiat et al. Le coût de tout algorithme A de service par bloc est donc au moins égal à H_{n-b} et le coût de l'algorithme off-line optimal est égal à 1 : sur les n-b appels, il existe forcément un sommet découvert car on dispose de n-1 serveurs, et en début de phase b serveurs sont placés sur le dernier bloc d'appels. Il reste donc n-1-b serveurs répartis sur les n-b sommets de la phase. Connaissant l'ordre dans lequel ces sommets vont appeler, l'algorithme optimal découvre le sommet qui sera requis au plus tard et n'enregistre sur la phase qu'un seul déplacement. On en déduit la proposition suivante :

Proposition :
Pour un nombre de serveurs k, ($b \leq k \leq n-1$), aucun algorithme de service par blocs de b sommets ne peut avoir un coefficient de compétitivité inférieur strictement à H_{k-b+1}.

5. 6. Application à la gestion d'outils

L'extension du problème des k-serveurs que nous avons étudiée peut trouver diverses applications ; dans le domaine de l'informatique par exemple, ce modèle présente toujours un intérêt pour la pagination, lorsque la notion d'"ensemble de travail" est employée pour gérer la mémoire (Tanenbaum (1992)). En effet, lorsqu'on a une certaine connaissance du comportement d'un programme, on peut déterminer à la suite du chargement d'une page en mémoire, quelles pages seront ensuite demandées pour poursuivre l'exécution du programme : ce peut être le cas

pour des instructions placées dans une boucle par exemple, mais le plus souvent cet ensemble de pages peut être identifié lorsqu'un processus au bout d'un certain temps d'exécution, dispose en mémoire de la plupart des pages qui lui sont nécessaires : il entame alors une phase de son exécution durant laquelle le système enregistre peu de défauts de pages. Lorsqu'une nouvelle phase commence, l'ensemble des pages associées est chargé en bloc dans la mémoire.

Dans ce contexte, on peut considérer que les processus font des demandes de pages groupées : pendant une phase, ils ne référencent qu'un nombre restreint de pages, qu'on définit comme le "working set" ou ensemble de travail, Denning (1970). Il ne s'agit donc plus vraiment de pagination à la demande, mais plutôt de "prepaging" ou "préchargement". Naturellement, ce procédé oblige le système à mémoriser le working set associé à chaque phase d'un processus ou à différents programmes qui s'exécutent en même temps ; les pages n'étant plus chargées à l'unité mais par blocs, le modèle des k-serveurs uniforme avec service par blocs pourrait s'appliquer à la gestion de ce système de préchargement, mais l'application qui nous intéresse en premier lieu est la gestion des changements d'outils, (Privault et Finke (1993)).

Nous avons eu plusieurs fois l'occasion au cours des précédents chapitres d'évoquer l'analogie existant entre les problèmes de k-serveurs et la gestion d'outils. Nous redonnons ici de façon plus détaillée une description du modèle afin d'utiliser l'algorithme de partitionnement adapté au service par blocs :

Le réseau des clients est constitué par un graphe complet dont chaque sommet représente un outil. Tous les outils répertoriés sur l'ensemble des jobs à fabriquer sont ainsi représentés. Chaque arête du réseau est pondérée par 1. On dispose sur ce réseau de C serveurs (autant que la capacité du magasin à outils de la machine). Lorsqu'un serveur se trouve placé sur un sommet cela signifie que l'outil correspondant est dans le magasin ; déplacer un serveur d'un sommet à un autre revient à faire sortir un outil pour en monter un nouveau sur la machine. Les blocs d'appels correspondent aux outils des jobs qui vont être usinés.

Si des pièces sont usinées au fur et à mesure de l'arrivée de commandes par exemple, ou si au contraire elles arrivent de façon imprévue et urgente au cours d'un processus de fabrication déjà entamé, elles doivent être usinées on-line, donc les changements d'outils correspondants doivent être décidés on-line. Pour le problème d'ordonnancement avec gestion d'outils que nous étudions, la situation est relativement différente : le nombre de tâches ainsi que leurs outils sont connus à l'avance, ce qui n'est pas le cas dans un problème de k-serveurs.

En revanche, si l'heuristique utilisée pour l'ordonnancement des tâches nécessite une gestion on-line des changements d'outils occasionnés par chaque bloc d'outils requis, on se trouve en présence d'un problème de C-serveurs uniforme avec service par blocs. Nous avons évoqué ce problème à la fin du chapitre 3 : il se posait particulièrement pour les méthodes heuristiques de type glouton comme l'heuristique *Next Best* : la séquence étant construite "pièce après pièce", on se ne sait pas à l'avance quel bloc succédera à quel bloc d'outils. Pour résoudre

ce problème, on dispose maintenant de l'algorithme de partitionnement adapté au service par blocs. Cet algorithme va servir de base à l'élaboration d'une heuristique qui crée la séquence et gère les outils au fur et à mesure.

L'idée de départ est simple : pour simuler des demandes de blocs et de cette façon construire la séquence job après job, on a retenu la méthode heuristique "nearest neighbour", (c'est-à-dire la méthode Next Best que nous avons testée). Pour commencer la séquence, on ne dispose pas de critère particulier pour le choix du premier job : on essaiera donc successivement chaque job comme début de séquence S ; chaque séquence sera ensuite évaluée en appliquant l'algorithme KTNS, et l'ordonnancement donnant le plus petit nombre de changements parmi les N séquences sera retenu. Le principe peut s'écrire de la façon suivante :

```
min = -1
POUR chaque job j de 1 à N FAIRE
    k = j
    S = { k }
    TANT QUE la longueur de S est inférieure ou égale à N FAIRE
        Gérer les outils du job k par Partitionnement
        Déterminer le job i qui maximise le coefficient de similarité avec le magasin
        Placer i à la suite de k dans la séquence S
        k = i
    FIN TANT QUE
    Calculer le coût de la séquence S par KNTS
    SI KTNS(S) < min ALORS
        min = KTNS(S)
        Meilleure-Séquence = S
FINPOUR
```

5. 7. Tests numériques

Cette heuristique appelée *PART* a été implantée sur Sun (Sparc2). Les paramètres qu'on a choisi de faire varier pour évaluer ses performances sont les mêmes que ceux que nous avons utilisés pour tester les méthodes heuristiques présentées au chapitre 3 : N le nombre de jobs, M le nombre d'outils, C la capacité, et le pourcentage d'occupation des magasins par les jobs, (c'est-à-dire les intervalles [30%, 60%] et [50%, 100%]).

On a choisi de comparer *PART* aux deux méthodes donnant de bons résultats en un temps raisonnable parmi celles que nous avons décrites et testées au chapitre 3, ainsi qu'à l'heuristique *Next Best* afin de comparer les critères de gestion d'outils. Nous avons également utilisé la

génération aléatoire de séquences, car comme nous l'avons observé au cours de nos tests précédents, il est intéressant de comparer ses scores à ceux d'autres méthodes dans le pire des cas. Après chaque exécution sur un type d'instance, comme la solution optimale n'est pas connue, on a retenu l'heuristique donnant le meilleur coût et on calcule l'écart en pourcentage entre ce coût et le score de chacune des 4 autres. Les chiffres indiqués correspondent à une moyenne de ces pourcentages pour chaque heuristique sur 10 instances différentes d'un même type de problème. De même, les temps d'exécutions indiqués sont des moyennes pour chaque heuristique sur 10 exécutions.

Au total, 12 types de problèmes ont été testés, (ce qui correspond donc à 120 matrices d'incidence) ; les couples de paramètres (N, M) que nous avons d'abord testés sont : (10, 20), (20, 50), (40, 60), et (50, 100). Toutefois, nos premiers tests ont fait apparaître que le premier type d'instances (10, 20) ne présente pas beaucoup d'intérêt, car le nombre de jobs est trop réduit pour qu'il y ait des écarts significatifs entre les différentes méthodes : les scores obtenus sont pratiquement identiques pour les quatre méthodes et aucune tendance ne se dégage vraiment de ces tests. Nous n'avons donc poursuivi que les tests résultant d'instances de tailles plus significatives, et ces résultats sont rassemblés dans les trois tableaux suivants.

N = 20 M = 50

Capacité	$C_1 = 15$		$C_2 = 26$	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
PART	1,4	1,1	0,5	1,5
Next Best	1,9	1,9	4,3	3,2
Stratégie de Regroupement	6,4	7,9	6,3	8,3
Farthest Insertion	23,4	1,1	26,9	1
Séquence Aléatoire	32,5	34,2	36,6	34,8

Temps CPU (secondes)

PART	5,1	7,1	5	7,9
Next Best	165	230	229,1	346,8
Stratégie de Regroupement	0,4	0,5	0,5	0,6
Farthest Insertion	0,6	0,7	0,6	0,7
Séquence Aléatoire	-	-	-	-

tableau 1

N = 40 M = 60

Capacité	$C_1 = 20$		$C_2 = 35$	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
PART	0,1	0,3	0,1	0,3
Next Best	6	2,5	6,8	4
Stratégie de Regroupement	5,2	6,3	7,7	5,8
Farthest Insertion	37	1,8	42,6	1,3
Séquence Aléatoire	40,8	40,2	49,7	43
	Temps CPU (secondes)			
PART	26	41	26	39
Next Best	1365	2100	2130	3404
Stratégie de Regroupement	1	1,5	1,3	2
Farthest Insertion	4	4,5	3,9	5
Séquence Aléatoire	-	-	-	-

tableau 2

N = 50 M = 100

Capacité	$C_1 = 30$		$C_2 = 60$	
Taille des jobs	[30%,60%]	[50%,100%]	[30%,60%]	[50%,100%]
PART	0	0,2	0,2	0,2
Next Best	6,1	2,6	6,1	3,13
Stratégie de Regroupement	6,2	5,1	3,7	3,8
Farthest Insertion	27,3	0,6	34	1
Séquence Aléatoire	32,9	27,7	37,6	29,3

Temps CPU (secondes)

PART	122,3	171,2	102,6	166,6
Next Best	3952,6	5992,2	7069	11571,6
Stratégie de Regroupement	2,7	3,5	3,1	5,8
Farthest Insertion	11,1	12,2	11,2	14,3
Séquence Aléatoire	0,5	0,5	0,5	0,6

tableau 3

Ces trois tableaux font apparaître le bon comportement de l'heuristique de *Partitionnement*. Ce comportement est le même quels que soient les paramètres, que les matrices d'incidence soient creuses ou au contraire très denses. Ce n'était d'ailleurs pas le cas pour l'heuristique *Farthest Insertion*, et on peut noter que cette tendance est confirmée par ce nouveau jeu de tests. Toutefois, lorsque les jobs occupent presque la totalité du magasin à outils, la méthode *Farthest Insertion* s'avère toujours aussi performante, et ses scores sont presque aussi bons que ceux obtenus par la méthode de *Partitionnement*. La *Stratégie de Regroupement* en revanche donne toujours des résultats satisfaisants (bien que moins bons), mais dont la qualité est constante quel que soit le type de l'instance.

En ce qui concerne la motivation essentielle de ces tests : la comparaison entre les critères de remplacements d'outils de *Next Best* et de l'heuristique de *Partitionnement*, la première remarque que nous pouvons faire porte sur les temps de calcul.

Il faut noter que l'heuristique de *Partitionnement* que nous avons programmée fonctionne dans sa version "Multiple", c'est-à-dire que chaque job est placé à son tour comme début de séquence, et que la meilleure séquence est choisie parmi les N obtenues. Afin de pouvoir tester réellement l'efficacité de la modélisation des changements d'outils sous forme de problème de k–serveurs, nous avons également programmé la version "Multiple" de la méthode *Next Best* pour que la possibilité de choisir entre N séquences n'intervienne pas dans les écarts qui pourraient éventuellement apparaître entre les deux méthodes. Or si on examine les tableaux indiquant les temps moyens d'exécution, l'écart entre les deux méthodes apparaît très nettement, même si ces temps sont des moyennes et ne sont donnés qu'à titre de comparaison :

Les temps d'exécution de la méthode *Next Best* sont beaucoup plus importants que ceux de la méthode de *Partitionnement*. Cela provient du fait que plus le nombre de lignes des matrices d'incidence est important plus les temps de parcours de ces matrices pour la recherche de l'outil le moins utilisé sont longs, d'autant plus qu'ils doivent être répétés pour chaque

séquence. On atteint des temps de l'ordre de 35 minutes pour des matrices "denses" (40,60) par exemple, tandis que l'exécution de la méthode de *Partitionnement* ne nécessite qu'une quarantaine de secondes, mais ces temps deviennent prohibitifs pour des matrices (50,100), même creuses (presque deux heures en moyenne comparées à 2 minutes environ pour la méthode de *Partitionnement*).

Si on examine maintenant les scores réalisés par les deux méthodes, l'écart est en faveur de la méthode de *Partitionnement* : de faible amplitude pour des matrices de taille moyenne (20, 50), il se creuse nettement lorsque la taille des instances augmente : pour 40 tâches et 60 outils, *PART* produit le meilleur score sur pratiquement toutes les instances, (et quel que soit leur type), tandis que *Next Best* se situe à 6% en moyenne de ces scores pour les matrices les plus creuses (voir tableau 2). Cet écart est confirmé sur les matrices de 50 jobs et 100 outils.

Comme toujours, les résultats obtenus sur des matrices denses ont tendance à être meilleurs, et la raison en est toujours la même : lorsque les magasins sont presque saturés par les outils requis pour la fabrication de chaque job, les critères de changements d'outils sont beaucoup moins déterminants dans la mesure où le choix des outils à démonter est restreint. Notons que dans ce cas, la *Stratégie de Regroupement* est tout aussi performante que *Next Best*, et qu'elle présente l'avantage d'être beaucoup plus rapide.

5. 8. Conclusion

Le modèle que nous avons défini tout en étant plus général, est naturellement moins complexe que celui du problème des k-serveurs dans sa forme classique. En effet, le fait de connaître plusieurs demandes, en quelque sorte à l'avance, réduit les choix possibles pour les déplacements de serveurs en induisant des contraintes supplémentaires. On peut presque considérer que le service des appels se fait toujours de façon séquentielle et unitaire, mais que la séquence des appels entre dans des phases au cours des quelles les serveurs peuvent être gérés off-line. De ce fait, la valeur des coefficients de compétitivité que nous avons établis pour les problèmes uniformes avec service par blocs sont inférieurs à ceux des algorithmes employés pour les problèmes classiques.

Nous avons proposé un algorithme qu'on peut considérer comme fortement compétitif si aucune restriction n'est faite sur la forme de la séquence, mais ce n'est certainement pas le seul, et il devrait en exister d'autres qui prendront en compte la composition spécifique que peuvent présenter certaines séquences d'appels. De plus, la recherche d'un algorithme compétitif pour les cas non uniformes reste un problème ouvert, pour le service par bloc comme pour le service

unitaire. Les problèmes non-uniformes ne sont pas sans intérêt pour la gestion on-line des changements d'outils, car il semble assez naturel d'être amené à prendre en compte dans certains cas des pondérations liées aux temps de changements relatifs à des types d'outils spécifiques.

Pour les temps de changements uniformes, les tests numériques que nous avons effectués permettent de mettre en évidence l'influence de la gestion des magasins d'outils dans la qualité de l'ordonnancement. La gestion on-line des changements effectuée par l'algorithme de *Partitionnement* apparaît plus efficace comparée aux autres critères de sortie d'outils ou même à l'utilisation d'approximations du nombre des changements calculées off-line.

5. 9. Bibliographie

P.J. Denning (1970)
"Virtual Memory", *ACM Computing Surveys* vol 2 n°3. pp 153-189.

A. Fiat, R.Karp, M.Luby, L.McGeoch, D.Sleator et N.E.Young (1991)
"Competitive Paging Algorithms", *Journal of algorithms* vol 12. pp 685-699.

L. McGeoch et D.Sleator (1991)
"A Strongly Competitive Randomized Paging Algorithm", *Algorithmica* vol 6. pp 816–825.

C. Privault et G. Finke (1993)
"Une extension des problèmes de serveurs appliquée à la gestion des outils dans un système flexible de production", *Actes du Congrès biennal AFCET'93*, Versailles, France. vol 1. pp 175-184.

A.S. Tanenbaum (1992)
Modern Operating Systems, Prentice Hall International Editions, Englewood Cliffs, N.J. pp 107-128.

Conclusion

Nous avons présenté différents modèles pour la gestion off-line et on-line des changements d'outils qui peuvent contribuer, du moins l'espérons nous, à une gestion plus efficace de l'outillage. N'oublions pas qu'auparavant des efforts auront été portés sur la réduction des temps de changements proprement dits, et que dans le même ordre d'idée, une réflexion aura également été menée en amont, au niveau de la planification et de l'organisation de la production : un regroupement par familles de produits, et une réaffectation des opérations sur les machines peuvent réduire considérablement l'effet des changements sur le temps total de production. De même, la prise en compte de ce critère lors de la conception des produits et de la génération de leurs gammes d'usinage peut contribuer à améliorer ces résultats.

La solution la plus simple mais la plus coûteuse reste l'achat de matériels de plus grande capacité. Pour répondre à cette demande, des machines équipées de magasins de grande capacité sont proposées aux entreprises, certaines étant pourvues d'éléments adaptables permettant d'étendre encore cette capacité, afin de produire sans aucun changements d'outils.

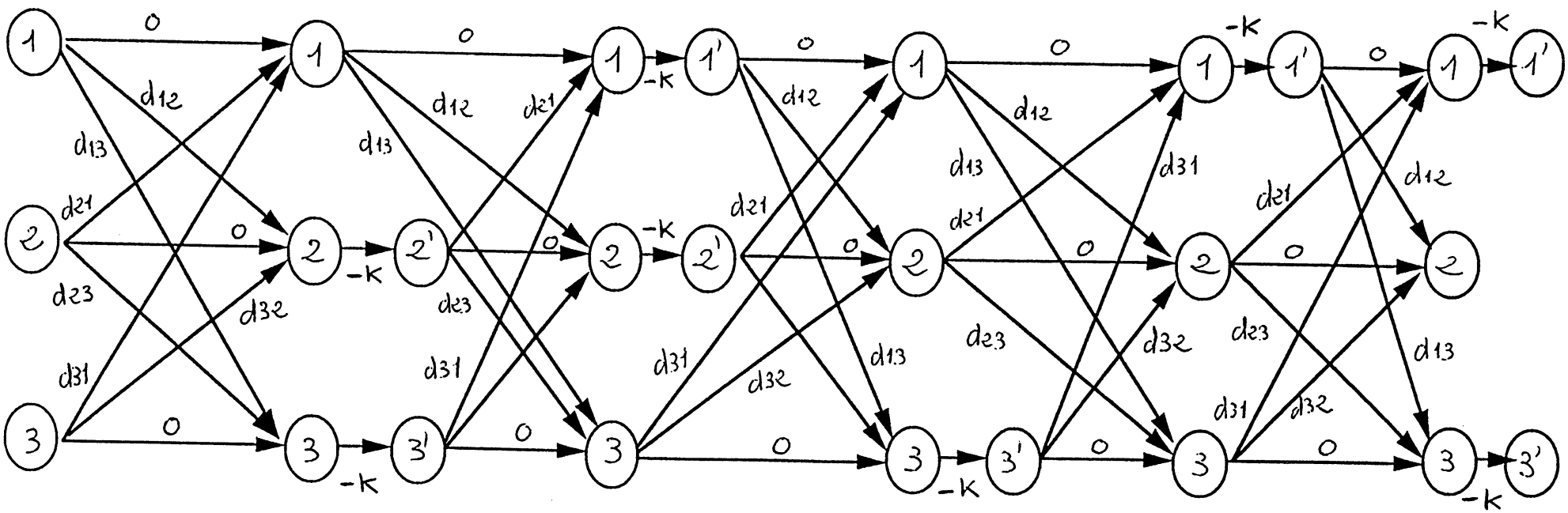
Cependant, cette situation idéale est loin d'être la règle pour toutes les entreprises. Pour les cas où le nombre des changements reste pénalisant pour la production, le modèle des k–serveurs que nous avons étudié constitue une nouvelle approche du problème d'ordonnancement avec gestion d'outils, qui se révèle pour le moins aussi efficace. De plus, cette approche peut s'avérer particulièrement intéressante dans la mesure où on exige une grande flexibilité de la part de l'appareil de production : dans ce contexte, il est de plus en plus fréquent que des décisions doivent être prises on-line au niveau de l'atelier.

Nous n'avons considéré le problème des changements d'outils que sur une seule machine, soit en fait un centre d'usinage flexible ; évoquons pour finir le cas de plusieurs machines flexibles. Dans le cas de machines différentes, ou effectuant des opérations différentes, nous avons mentionné dans le chapitre 1 les problèmes de job shop avec contraintes d'outillage. S'il s'agit d'une ligne flexible de production, chaque machine requiert un jeu d'outils particuliers associé aux opérations qui lui sont affectées. Les modèles de gestion off-line des outils peuvent donc être appliqués à chaque machine indépendamment en fonction de l'ordonnancement qui a été lancé sur la ligne de production.

S'il s'agit de cellules d'usinage, composées de plusieurs machines identiques destinées à effectuer les mêmes opérations, le problème de l'affectation des types de pièces sur les machines, en fonction des outils correspondants vient s'ajouter au problème d'ordonnancement. On rejoint alors la théorie de l'ordonnancement avec ressources sur machines parallèles.

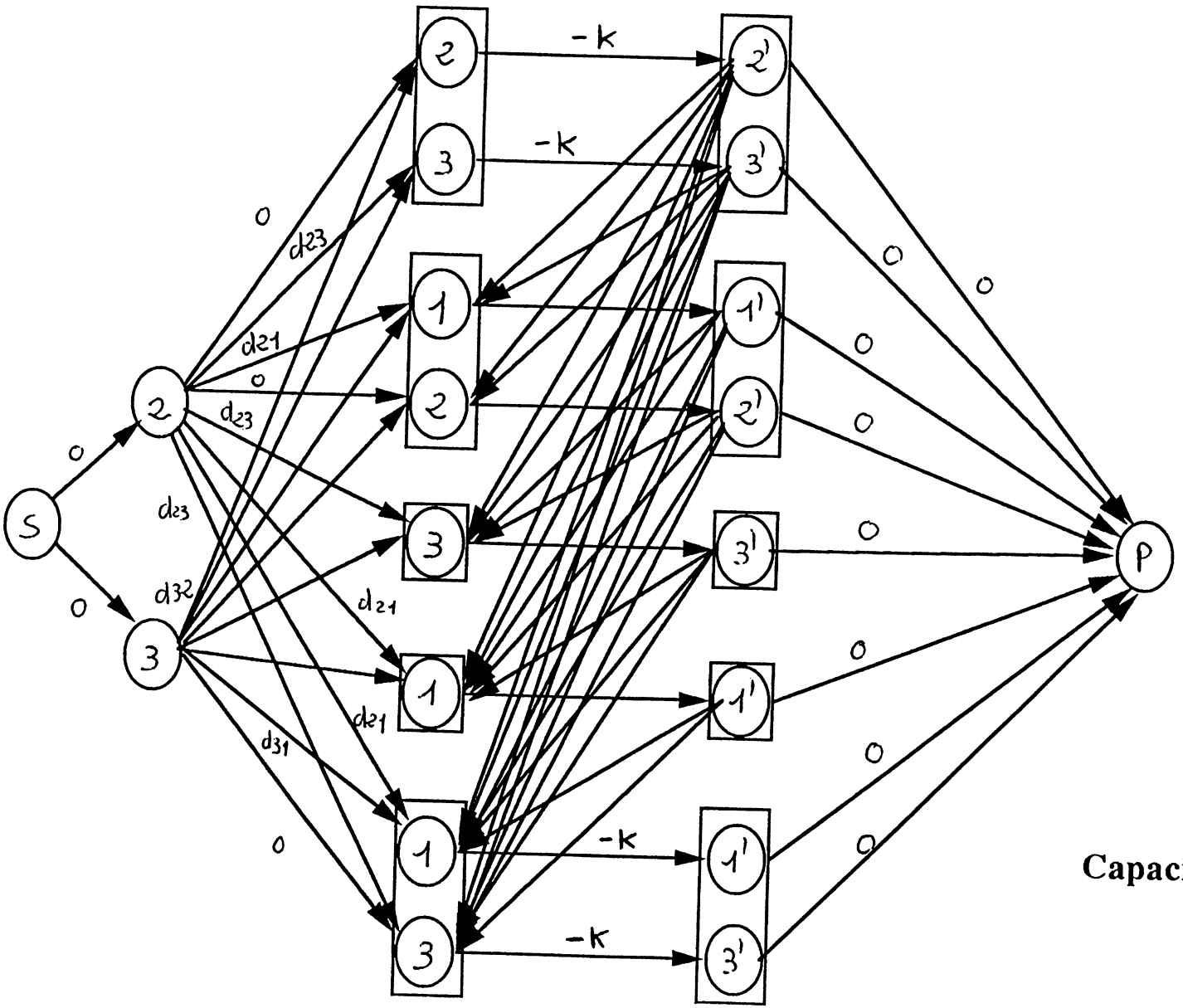
Index des auteurs

ANNEXE 1 : PREMIER MODELE

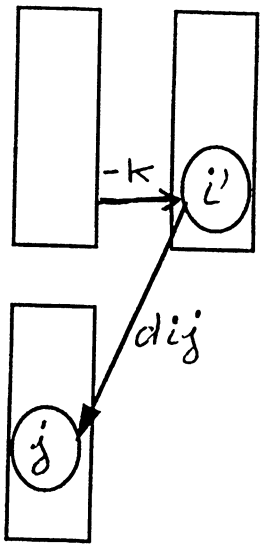


Capacités : 1 sur tous les arcs

ANNEXE 2 : SECOND MODELE



Pondérations : 0/1 ou dij



Capacités : 1 sur tous les arcs

Résumé

L'objet de ce travail est l'étude d'un problème d'ordonnancement, dont le critère d'optimisation est la minimisation du nombre total de changements d'outils sur une machine flexible. Différents problèmes liés à l'outillage et pouvant constituer un obstacle au fonctionnement d'un atelier flexible sont brièvement examinés, (de la gestion de l'inventaire aux changements d'outils). Nous nous concentrons ensuite sur le problème d'ordonnancement avec gestion d'outils sur une seule machine : ce problème est NP-complet.

Un premier aspect qui est la gestion off-line des outils est étudié : différents modèles sont proposés pour la minimisation des changements en fonction d'une séquence des tâches donnée. Dans ce cas, le problème est polynomial. Nous revenons ensuite au problème d'ordonnancement proprement dit, pour lequel plusieurs types de méthodes heuristiques sont décrites et comparées.

La seconde partie du travail est consacrée à la gestion on-line des changements d'outils ; elle se compose de deux chapitres : dans le premier, le modèle que nous allons utiliser est décrit en détail. Il s'agit de la modélisation des problèmes de k-serveurs. Le principe peut être résumé comme suit : sur un réseau de n clients potentiels, on dispose de k serveurs mobiles, avec lesquels on doit répondre on-line aux demandes unitaires et successives des clients, tout en optimisant les déplacements des serveurs. Après un tour d'horizon sur ces problèmes et leurs applications dans la gestion des mémoires en informatique, nous revenons dans le dernier chapitre aux changements d'outils on-line. Le modèle des k-serveurs "classique" est généralisé aux problèmes avec service par blocs de demandes, ce qui permet d'adapter l'algorithme de partitionnement, (fortement compétitif pour le service unitaire), au cas plus général des demandes groupées. Cet algorithme dont nous étudions les propriétés et la compétitivité, sert de base à la construction d'une heuristique d'ordonnancement avec gestion on-line des changements d'outils. Cette dernière méthode est comparée aux précédentes sur des exemples numériques.

Mots clefs : Atelier flexibles, ordonnancement, gestion d'outils, heuristiques, flots, problèmes de k-serveurs

Abstract

The aim of this thesis is to study scheduling problems that arise in connection with the tool management on flexible machines (NC-machines). Several tooling problems are examined, covering tooling inventory and tool switching problems. Our main focus is on scheduling problems which minimize the number of tool switches, i.e. the number of tool loading and unloading opeartions at the tool magazine of the NC-machine. This problem is in general NP–complete and is, in fact, more general than the traveling salesman problem.

The first aspect concerns the off-line tool management. Several models are proposed for the minimization of the number of tool switches for a given sequence of jobs. In particular, network flow formulations are presented. These also solve more general cases at a slightly improved polynomial complexity.

The second part is devoted to the on-line tool management. It is composed of two sections. First we describe existing and new heuristical methods that are mainly related to the traveling salesman problem. Extensive empirical comparisons are given. Then we relate the model to the so-called k-server problem which has been studied in depth connection with the memory management of computer systems. Its principle may be summarized as follows : on a network of n potential costumers, k mobile servers have to be moved to customers requiring service at minimum cost. The service requests arrive one at a time. This classical k-server problem is generalized to cases with bulk requests. With this modification, the model is applicable to the on-line tool management problem in manufacturing. We show that it is possible to adapt the so-called Partitionning Algorithm, which is strongly competitive for unitary services, to the more general case of bulk requests. Properties and competitiveness of this modified algorithm are studied. Then it is used to construct a heuristical method for the on-line tool switching problem. This algorithm proves to be very efficient when compared to the previous procedures.

Key words : Flexible manufacturing systems, scheduling, tool management, heuristics, network flows, k-server problems.