ThsN° d'Ordre : D.U. 1723
EDSPIC : 237

Université Blaise Pascal - Clermont Ferrand II

ECOLE DOCTORALE
SCIENCES POUR L'INGENIEUR DE CLERMONT FERRAND

THESE

Présentée par

Anthony CAUMOND

Diplômé d'Etudes Approfondies d'Informatique
Ingénieur en informatique
pour obtenir le grade de

Docteur d'Université

Spécialité : INFORMATIQUE

Le problème de jobshop avec contraintes: modélisation et optimisation

Soutenue publiquement le 18 décembre 2006 devant le jury :

Monsieur	Alain QUILLIOT	Président du jury
Monsieur	Jean Charles BILLAUT	Rapporteur
Monsieur	Marino WIDMER	Rapporteur
Monsieur	Philippe LACOMME	Directeur de thèse
Monsieur	Nikolay TCHERNEV	Co-encadrant

Table des matières

Chapitre 2 Le problème de jobshop

Ce chapitre présente le problème de jobshop et les principales techniques utilisées pour sa résolution. Entre autres, le graphe conjonctif - disjonctif, son chemin critique et les voisinages afférents sont présentés. Tous ces outils sont réutilisés et adaptés dans les chapitres suivants.

Sommaire

Table des figures

Table des tableaux

Table des algorithmes

1 Introduction

Dans ce chapitre, nous présentons les techniques employées pour la résolution du problème de jobshop. Nous nous intéressons à ce problème car il est classique dans la littérature de l'ordonnancement et que c'est le plus général des problèmes classiques d'ordonnancement d'atelier (Jain et Meeran, 1999). En effet, le problème de jobshop :

- généralise d'autres problèmes classiques d'ordonnancement comme le problème à une machine, le flowshop de permutation et le flowshop général. Ainsi, un algorithme traitant des problèmes de jobshop peut résoudre des instances de ces problèmes particuliers.
- est un sous-problème du problème d'ordonnancement des ateliers flexibles (type jobshop flexible, flowshop hybride, ...). Ces ateliers intègrent en plus du problème d'ordonnancement un problème d'affectation des gammes aux jobs ou des opérations aux machines. La résolution de ce problème se compose alors de deux sous-problèmes résolus simultanément ou consécutivement. Les problèmes d'ateliers flexibles comportent donc un sous-problème d'ordonnancement pur et un problème d'affectation. La partie ordonnancement pur est donc un problème disjonctif dont la forme la plus générale est le problème de jobshop. Ainsi, même si le problème de jobshop ne généralise pas les problèmes d'ateliers flexibles, c'est une étape nécessaire à leur résolution.
- ne traite pas le même problème que le problème de RCPSP. En effet, le "Resource Constrained Project Scheduling Problem" est comme son nom l'indique dévolu aux problèmes d'ordonnancement de projet. Ces problèmes généralisent le problème de jobshop et les algorithmes du RCPSP sont donc capables de résoudre les problèmes de jobshop. Pourtant, étant plus généraux, ils sont moins adaptés et sont donc en général moins performants.

Ce chapitre est composé de quatre parties. La première présente le problème de jobshop en général (paragraphe 2). La deuxième (paragraphe 3) présente les principales classes d'ordonnancement qui caractérisent les ordonnancements recherchés. La troisième partie (paragraphe 4) présente un état de l'art du problème de jobshop. Cet état de l'art s'intéresse particulièrement aux outils communément utilisés par les meilleurs articles de la littérature : graphe conjonctif-disjonctif, plus long chemin et voisinage guidé. La dernière et cinquième partie présente plus en détail une des meilleures méthodes publiées : l'algorithme tabou de Nowicki et Smutnicki.

2 Définition du problème

Soit un problème de jobshop. Les notations suivantes sont introduites pour décrire le problème. Soit M l'ensemble des m machines $M_1, M_2, ..., M_m$ et J l'ensemble des n jobs $J_1, J_2, ..., J_n$. On note O l'ensemble des o opérations $O = \{O_1, O_2, ...O_o\}$ décrivant la réalisation des jobs sur les machines.

Les jobs sont décrits par des gammes. Une gamme définit l'ordre, la durée et la séquence des machines où sont traitées les opérations d'un job. La gamme définit pour chaque job J_i une suite K_i de k_i opérations dont la première est notée d_i : $K_i = (O_{d_i}, O_{d_i+1}, \ldots, O_{d_i+k_i-1})$. Les suites K_i sont disjointes, c'est-à-dire qu'une opération n'apparaît que dans une seule suite K_i. La suite des opérations de K_i est appelée ordre d'opérations de la gamme. Un job ne peut commencer le traitement d'une opération que quand l'opération précédente dans la gamme est terminée. On appelle contrainte de précédence cette contrainte et on note $A = (O_i, O_{i+1})$ l'ensemble des opérations successives soumises à une contrainte de précédence directe.

Une opération O_i doit être réalisée pendant un temps p_i sur une machine m_i. Cette opération doit être réalisée sans préemption.

On note E_k où $k \in M$ l'ensemble des opérations devant être traitées sur la machine M_k. Le nombre d'opérations à traiter sur la machine M_k est donc $e_k = |E_k|$. Une machine ne peut traiter qu'une

seule opération à la fois. Les opérations de E_k doivent donc être ordonnées, on dit qu'il y a disjonction entre elles car les intervalles de temps pendant lesquels la machine est occupée sont disjoints.

La taille d'une instance est notée $n \times m$.

Une instance rectangulaire est une instance dont tous les jobs passent sur toutes les machines ($k_i = Cte, \forall i$). Parce que les notations et les formules se simplifient pour ce type d'instance, il est assez courant dans la littérature que les algorithmes ne traitent que les instances rectangulaires. Pour ces instances, on a nm opérations à ordonnancer et toutes les machines doivent traiter toutes les opérations une et une seule fois ($e_k = n, \ \forall k \in M$). Les données de ce type d'instance peuvent donc être représentées par des matrices rectangulaires (d'où le qualificatif attribué à ces instances). De manière générale, ceci n'est pas restrictif et il est souvent possible d'adapter les algorithmes pour que les jobs ne passent pas sur toutes les machines. Par contre, certains algorithmes ne permettent pas de prendre en compte les jobshops avec recirculation. En effet, dans un jobshop avec recirculation, deux opérations du même job sont traitées par la même machine. C'est le cas par exemple des algorithmes où les opérations sont identifiées par le couple (machine, job).

Les hypothèses suivantes sont communément retenues pour le problème de jobshop. Certaines variantes du problème de jobshop consistent à supprimer certaines de ces hypothèses :

- Tous les jobs sont disponibles dès le début de l'ordonnancement.
- Les machines sont disponibles sur tout l'horizon d'ordonnancement (pas de panne, pas d'état initial non vide).
- Les temps de traitement p_i sont déterministes et connus à l'avance.
- Les temps de montage et de démontage sont inclus dans les temps de traitement
- Les temps de transport sont négligeables.
- Chaque machine ne peut réaliser à un instant donné qu'une seule opération.
- Un job peut être traité au plus par une machine à un instant donné.
- Les produits peuvent attendre dans les stocks de capacité illimitée.

2.1 Formalisation mathématique

Afin de décrire le problème de façon univoque, nous le décrivons dans un premier temps à l'aide du formalisme mathématique. La formalisation proposée par (Manne, 1960) peut aisément être adaptée aux notations ci-dessus pour le problème de jobshop :

t_i est la date de début de l'opération i,

(1) $t_i + p_i \le t_j \quad \forall (i,j) \in A$

(2) $t_j - t_i \ge p_i \quad oux \ \ t_i - t_j \ge p_j \qquad \forall (i,j) \in E_k, \forall k \in M,$

(3) $t_i \ge 0 \quad \forall i \in O,$

Dans cette formalisation, le *oux* désigne le ou exclusif, ainsi une seule des deux inéquations de (2) doit être valide (les deux ne peuvent être valides en même temps).

L'objectif est de minimiser makespan (i.e. la date de fin de la dernière opération).

2.2 Programme linéaire

La formalisation présentée dans la section ci-dessus peut aisément être linéarisée en introduisant les variables binaires a_{ij}. La variable a_{ij} est définie par $a_{ij} = 1$ si l'opération j est réalisée avant l'opération i, $a_{ij} = 0$ sinon. La contrainte (2) est alors remplacée par les contraintes suivantes :

H est une constante suffisamment grande ($H \ge \sum_{i \in O} p_i$) :

(2') $a_{ij} \in \{0;1\}$

(2") $(H + p_j) a_{ij} + (t_j - t_i) \ge p_i$

(2''') $(H + p_j)(1 - a_{ij}) + (t_i - t_j) \geq p_j$

où $\forall (i, j) \in E_k, \forall k \in M,$

L'ensemble des contraintes (1), (2'), (2''), (2''') et (3) définissent les contraintes du problème de jobshop. La contrainte (4) suivante permet de définir la fonction objectif :

(4) $Z \geq t_i + p_i, \forall i \in O$

Les contraintes (1), (2'), (2''), (2'''), (3) et (4) munies de la fonction objectif $\min(Z)$ forme un programme linéaire en nombres entiers modélisant le problème de jobshop.

Remarque : La contrainte d'intégrité sur les dates de début ($t_i \in N$) n'est pas imposée, même si la formulation ainsi obtenue est valide. Les principes de résolution des programmes linéaires mixtes montrent qu'il est préférable de ne pas imposer l'intégrité sur les variables qui ne le nécessitent pas.

La formalisation linéaire ci-dessus permet de résoudre tous les problèmes de jobshop. Mais, les temps de calcul obtenus peuvent rapidement devenir prohibitifs. La littérature du jobshop s'attache donc à développer des méthodes d'optimisation dédiées au problème de jobshop de manière à proposer les solutions pour des problèmes de taille moyenne ou importante.

3 Classe d'ordonnancement

Lors de la recherche d'un ordonnancement, on découvre généralement plus d'un ordonnancement répondant aux critères sélectionnés. En particulier, lorsque l'on recherche un ordonnancement de makespan minimum, il y a généralement de nombreux ordonnancements avec le même makespan. Par exemple, les dates de début de certaines opérations peuvent être retardées sans qu'il n'y ait d'effet sur la valeur du critère. L'intérêt des classes d'ordonnancement est de limiter la recherche à un sous ensemble des ordonnancements dont on sait qu'ils sont meilleurs que les ordonnancements non explorés.

Les classes d'ordonnancement sont présentées en détails et précisément dans (Baker, 1974). Dans cette section, nous ne répétons pas cette définition mais nous proposons une présentation un peu plus générale de ce que sont les classes d'ordonnancements. L'objectif est de fournir des définitions plus générales permettant de s'adapter à des problèmes plus complexes comme les problèmes que nous traitons dans les chapitres suivants.

L'exemple suivant est un exemple de problème d'ordonnancement. Il est utilisé dans la suite pour illustrer les notions présentées. Ce problème est composé de deux jobs à réaliser sur trois machines. Chaque job est composé de trois opérations. Une opération est réalisée sur une machine et durant un temps prédéterminé dont les temps sont donnés dans le tableau 2-1. L'objectif est de trouver un ordonnancement dont la date de fin de la dernière opération est minimale (i.e. makespan).

Job 1	O1 = Machine 1, Durée = 4	O2=Machine 2, Durée = 2	O3 = Machine 3, Durée = 1
Job 2	O4 = Machine 2, Durée = 7	O5 = Machine 3, Durée = 3	O6 = Machine 1, Durée = 2

Tableau 2-1. Instance exemple J1

Dans les exemples suivants, nous utilisons la borne inférieure qui consiste à calculer la somme des durées des opérations d'un job. Puisque deux opérations d'un même job ne peuvent être réalisées simultanément, un ordonnancement a donc pour durée minimum la durée du plus long de ses jobs. Autrement dit, la somme des temps de traitement d'un job est une borne inférieure du problème. Le job 1 a pour durée 7, le job 2 a pour durée 12. Le job 2 est donc le plus long : s'il commence dès le début de l'ordonnancement puisqu'il s'exécute sans interruption pour terminer en dernier, c'est que l'ordonnancement proposé est clairement optimal. On dira dans ce cas que le job 2 est critique.

Parmi les ordonnancements possibles, on cherche généralement un ordonnancement minimisant ou maximisant un ou plusieurs critères. Dans les paragraphes suivants, on présente des

classes d'ordonnancement tels qu'un ordonnancement qui n'est pas dans la classe peut toujours être modifier afin d'obtenir un ordonnancement de même critère ou inférieur et qui fait partie de la classe d'ordonnancement. Les classes d'ordonnancements présentées sont valables pour les critères réguliers, (i.e. critères dont la valeur n'augmente pas lorsque l'on diminue la date de début d'une opération). Cette classe de critères est largement étudiée dans la littérature et rassemble la plupart des critères étudiés. L'ensemble des classes d'ordonnancements et leurs inclusions respectives sont représentés par la figure 2-6 (page 62).

3.1 Point 1 : Ordonnancements semi-actifs

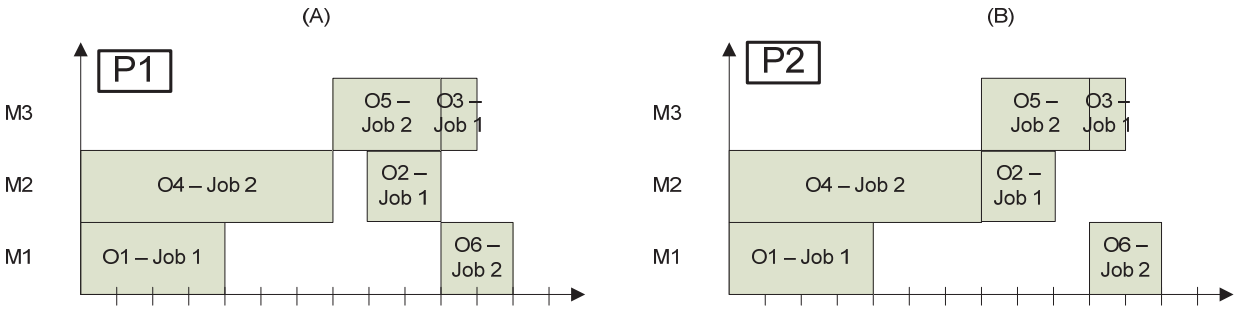

Figure 2-1. Ordonnancement optimal, non semi-actif (A)

et l'ordonnancement semi-actif correspondant(B)

L'ordonnancement de la figure 2-1(A) est un ordonnancement optimal car le job 2 est critique. Dans cet ordonnancement, on peut déplacer d'une unité vers la droite l'opération 3, on peut alors de nouveau déplacer l'opération 2 soit sur la gauche soit sur la droite. On vient de construire 4 ordonnancements qui ont tous le même makespan et qui sont tous optimaux. Ces ordonnancements ont tous le même makespan. Mais on ne construit en général que l'ordonnancement dont toutes les dates de début sont les plus faibles possibles (ordonnancement de la figure 2-1 (B)). Cet ordonnancement est dit semi-actif et est obtenu par décalages à gauche successifs.

Définition : décalage à gauche

Soit un problème d'ordonnancement *P*, et un ordonnancement *O*. Un décalage à gauche sur l'ordonnancement *O* consiste à diminuer la date de début d'exécution d'une opération de *O*. Dans la suite, on notera *G(O)* l'ensemble des ordonnancements obtenus par décalage à gauche.

On s'attend en général à ce qu'un décalage à gauche améliore un ordonnancement. Les critères d'optimisation qui vérifient cette propriété sont appelés "critères réguliers".

Définition : critère régulier

Soit un problème d'ordonnancement *P*, un critère *C* est dit régulier si tout ordonnancement obtenu par décalage à gauche mène à un ordonnancement de critère inférieur ou égal. $\forall O' \in G(O), C(O') \leq C(O)$

Les ordonnancements semi-actifs sont définis à partir de décalages à gauche pour les critères réguliers.

Définition : semi-actif

Un ordonnancement *O* est semi-actif s'il n'est pas possible de commencer une opération de *O* plus tôt sans violer une contrainte ou sans changer l'ordre des opérations sur les ressources ou sur les machines.

L'ordonnancement de la figure 2-1 (A) est optimal mais n'est pas semi-actif : la date de début de l'opération 2 peut être décalée à gauche sans qu'aucune contrainte ne soit violée et aucun ordre changé sur aucune machine. L'ordonnancement obtenu par décalage à gauche est représenté sur la figure 2-1 (B), cet ordonnancement est donc semi-actif et optimal. L'ordonnancement de la figure 2-1 (A) correspond au point 1 alors que l'ordonnancement de la figure 2-1 (B) correspond au point 2 de la figure 2-6.

3.2 Point 2 : Ordonnancement optimal

Lorsque l'on veut calculer un ordonnancement semi-actif à partir d'un ordonnancement quelconque, on procède par décalages à gauche successifs en préservant l'ordre des opérations et en respectant les contraintes. Si on change le critère d'arrêt de cette procédure de manière à s'arrêter quand tous les décalages à gauche mènent soit à une violation de contrainte soit à un ordonnancement de plus faible coût, on obtient un ordonnancement actif.

Définition : actif

Un ordonnancement *O* est actif si tout décalage à gauche oblige à retarder l'exécution d'une autre opération ou à violer une contrainte.

La définition ci-dessus permet de définir une procédure pour construire les ordonnancements actifs. Elle consiste à réaliser des décalages à gauche jusqu'à ce qu'une opération retarde une autre opération ou viole une contrainte. De par sa définition, il est clair que cette procédure ne peut pas produire d'ordonnancements de coût supérieur à l'ordonnancement de départ.

Suivant l'ordre dans lequel les décalages à gauche sont envisagés et effectués, on peut obtenir plusieurs ordonnancements actifs à partir d'un même ordonnancement de départ.

Cette procédure est rarement implantée pour construire un ordonnancement actif, on préfère utiliser l'algorithme de Giffler et Thompson (1960) (cet algorithme est présenté en détails, pour le problème de jobshop dans le paragraphe 6.3.1 du chapitre 3, page 124).

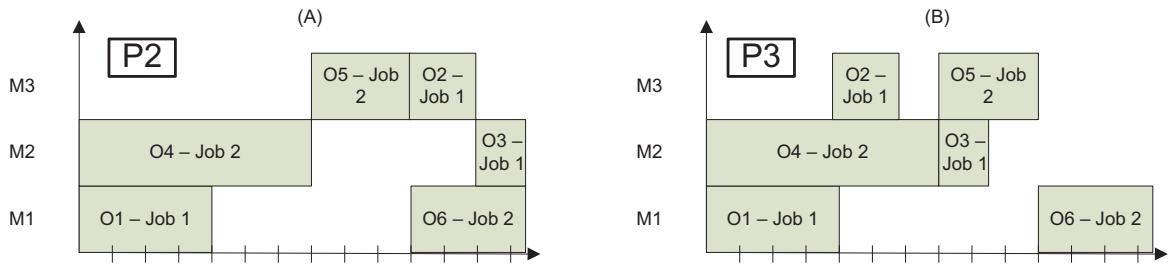

Figure 2-2. (A) est un ordonnancement optimal et non actif

et (B) est l'ordonnancement actif correspondant

Afin d'illustrer la notion d'actif, l'instance J1 est modifiée en permutant les machines sur lesquelles sont réalisées les deux dernières opérations. La figure 2-2 utilise cette instance et montre deux ordonnancements semi-actifs dont l'un des deux n'est pas actif. Cet ordonnancement de la figure 2-2 (A) est optimal car son makespan est la durée totale du job 2. Cet ordonnancement est semi-actif car tout décalage à gauche viole une contrainte ou change l'ordre des opérations sur une machine.

Pourtant il n'est pas actif car un décalage à gauche poussant l'opération 2 avant l'opération 5 mène à un ordonnancement de même critère et ne retarde l'exécution d'aucune autre opération. Cet ordonnancement représente le point 2 de la figure 2-6.

La figure 2-2 (B) présente l'ordonnancement actif obtenu par décalage à gauche de l'ordonnancement de la figure 2-2 (A). Cet ordonnancement est donc actif et optimal (Point 3 de la figure 2-6).

On peut prouver que pour tout problème d'ordonnancement dont le critère est régulier, il existe au moins un ordonnancement optimal actif.

3.3 Point 3 : Ordonnancements sans délai

Dans la figure 2-3 (A), on présente un ordonnancement optimal (car le job 2 est critique) qui contient des "trous" : à la fin de l'opération 1, on ne commence pas l'opération 2 alors que le job est disponible et que la machine l'est aussi. Cet exemple est basé sur l'instance J1 dans laquelle la durée de l'opération 1 est passée à 6, et la durée de l'opération 6 est passée à 3. Les ordonnancements sans délais interdisent qu'une opération ne soit retardée alors que ses ressources sont disponibles.

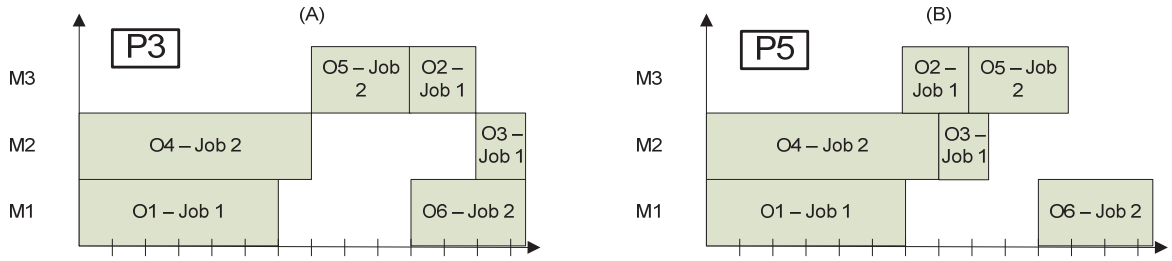

Figure 2-3. (A) est un ordonnancement optimal et non sans délai et

(B) est son ordonnancement sans délai

Définition : sans délai

Un ordonnancement *O* est sans délai si à chaque instant toute opération qui dispose des ressources nécessaires est commencée.

Pour obtenir un ordonnancement sans délai à partir de l'ordonnancement de la figure 2-3 (A), il faut décaler l'opération 2 à gauche. L'ordonnancement ainsi obtenu est présenté dans la figure 2-3 (B). Le décalage à gauche de l'opération 2 a retardé l'exécution de l'opération 5, ce qui entraîne le déplacement de l'opération 6 et augmente le makespan. L'ordonnancement de la figure 2-3 (B) est donc sans délai et non optimal (point 5 de la figure 2-6). L'ordonnancement de la figure 2-3 (A) est optimal et non sans délai (point 3)

3.4 Point 4 : Ordonnancement optimal et sans délai

La figure 2-4 présente un ordonnancement optimal et sans délai basé sur l'instance J1. L'ordonnancement est optimal car son makespan est la durée du job 2 et elle est sans délai car à aucun moment une opération n'est retardée alors que ses ressources sont disponibles. Cet exemple montre donc qu'il existe des problèmes pour lesquels il y a des ordonnancements sans délai et optimaux. Cet ordonnancement représente donc le point 4 de la figure 2-6.

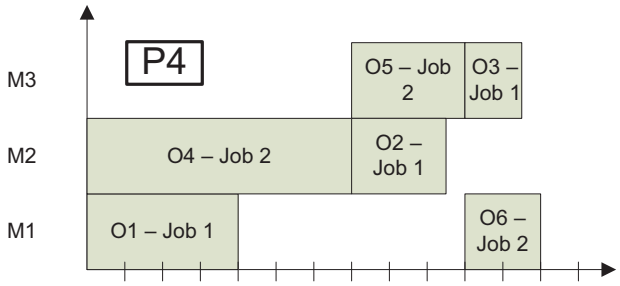

Figure 2-4. Ordonnancement optimal et sans délai

3.5 Existence du point 4

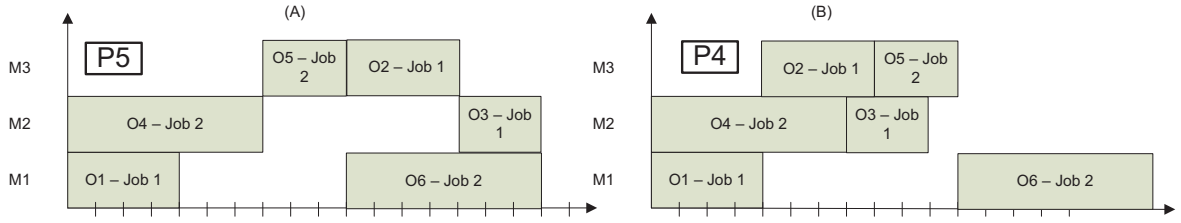

Figure 2-5. Ordonnancement optimal (A), sans délai et non optimal (B)

Suivant les problèmes, il est possible qu'il n'existe pas d'ordonnancements optimaux sans délai (i.e. pas de point 4). La figure 2-5(A) présente un ordonnancement d'un tel problème. Cet ordonnancement est basé sur l'instance J1 dans laquelle l'opération 2 est passé à 4, l'opération 6 est passée à 7 et l'opération 3 est passée à 3.

Le makespan de cet ordonnancement est égal à la durée du job 2, il est donc optimal et toute solution optimale ne doit pas retarder l'exécution du job 2. Cet ordonnancement n'est pas sans délai car à la fin de l'opération *O1*, le job *J1* est disponible ainsi que la machine *M3* où l'opération suivante doit être réalisée. Cet ordonnancement correspond donc au point 5 de la figure 2-6.

Obtenir un ordonnancement sans délai à partir de cet ordonnancement revient à faire un décalage à gauche de l'opération O2. Ce décalage retarde l'exécution de l'opération O5 et retarde donc l'exécution de l'opération O6. L'ordonnancement ainsi obtenu est donc non optimal (cf. figure 2-5(B)).

Ainsi, pour construire une solution optimale on doit réaliser le job 2 sans interruption depuis $t = 0$. Le job 1 commence au plus tôt, dès $t = 0$. L'opération suivante (opération O5) peut être réalisée avant ou après l'opération O2. Si elle est réalisée après l'opération O2, l'ordonnancement obtenu est au mieux celui de la figure 2-5(B) qui n'est pas optimal. Si elle est réalisée avant l'opération O2, on obtient un ordonnancement optimal (cf. figure 2-5(A)) mais non sans délai. Pour le problème présenté, il n'existe donc pas d'ordonnancement sans délai et optimal.

La figure 2-6 présente les relations entre les classes d'ordonnancements. Les ordonnancements sans délai sont inclus dans les ordonnancements actifs qui sont inclus dans les ordonnancements semi-actif et qui sont tous réalisables. La position de l'ensemble des ordonnancements optimaux dépend des problèmes. La figure 2-6 est décomposée en deux parties correspondant aux deux cas de figure possibles (A et B). En A, l'ensemble des solutions optimales coupe l'ensemble des solutions sans délai alors qu'il ne le coupe pas en B. Toute instance d'un problème d'ordonnancement dont le critère est régulier est dans un des deux cas ci-dessus. En effet, on montre qu'il existe au moins un ordonnancement actif optimal qui peut être obtenu par décalages à gauche successifs d'une solution optimale.

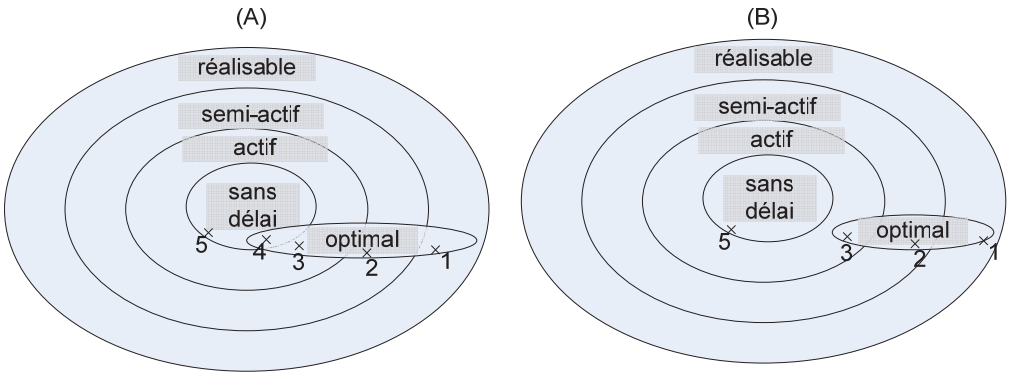

Figure 2-6. Classes d'ordonnancements et optimalité

3.6 Conclusion sur les classes d'ordonnancements

Lorsque l'on conçoit un algorithme d'optimisation, on doit choisir la classe des ordonnancements que l'on génère. Parmi toutes les classes d'ordonnancement, on tend à choisir celle qui contient le plus faible nombre d'ordonnancements. Pourtant, si l'on veut que l'algorithme puisse trouver la solution optimale, il faut que cette classe la contienne systématiquement.

Suivant l'objectif de l'algorithme d'optimisation (i.e. trouver forcément la solution optimale ou non), on choisit la classe d'ordonnancement la plus appropriée. Les ordonnancements sans délais sont généralement utilisés dans les heuristiques car elles ne permettent pas de trouver la solution optimale de manière certaine. Les ordonnancements actifs sont largement utilisés, mais ils souffrent d'un problème de représentation (cf. 4.3.8) : en quelques mots, ils ne sont pas adaptés à être informatiquement stockés. Ceci entraîne que plusieurs solutions stockées de manières différentes conduisent à un même ordonnancement actif. Les ordonnancements semi-actifs sont quant à eux très souvent utilisés. Parfois, leur utilisation est même tacite. C'est le cas en particulier quand on assimile un ordre d'opérations et un ordonnancement.

Même si l'intérêt des ordonnancements sans délai est limité aux heuristiques, ils sont tout de même largement employés dans le contexte de la simulation. En effet, en simulation classique, on décide si une nouvelle opération va être réalisée à chaque événement de fin d'occupation d'une ressource. Ainsi, on n'envisage en général pas de laisser une ressource libre alors qu'elle pourrait traiter une opération. La simulation classique ne peut donc générer que des solutions correspondant aux ordonnancements sans délai. Il est ainsi important de constater qu'une approche par simulation peut ne pas fournir la solution optimale. Cette remarque n'est plus vraie dans le cas de la simulation de type réflexive, car ce type de simulation permet d'anticiper les décisions à prendre à l'aide d'un modèle de simulation auxiliaire utilisé en interne. La simulation réflexive permet de générer tout type d'ordonnancements, mais est coûteuse en termes de ressources informatiques lors de sa mise en œuvre.

4 Etat de l'art du problème de jobshop

Le problème de jobshop est largement étudié dans la littérature et le nombre de références traitant de ce problème est trop élevé pour permettre une énumération exhaustive. Dans ce paragraphe, nous présentons donc quelques états de l'art de référence (paragraphe 4.1) et un aperçu historique de ce problème (paragraphe 4.2). Dans cet aperçu, nous mettons en évidence que de nombreux algorithmes performants utilisent le triplet (graphe conjonctif-disjonctif, chemin critique et voisinages basés sur le chemin critique). Ces trois outils sont présentés dans les paragraphes suivants.

Globalement, cette partie met en évidence l'importance du graphe conjonctif - disjonctif et des outils et méthodes associés dans la littérature du problème de jobshop.

4.1 Généralités

Principalement deux états de l'art font référence pour le problème de jobshop (Jain et Meeran, 1999) et (Blazewicz *et al.*, 1996).

Jain et Meeran (1999) décrivent précisément le problème et listent les principaux articles en les classant par méthode d'optimisation utilisée. Chaque méthode est accompagnée d'une courte explication décrivant son origine et sa spécificité. L'état de l'art de Jain et Meeran propose une liste complète des instances de jobshop connues de la littérature. Cette liste peut servir de référent pour la comparaison entre méthodes d'optimisation. Un travail important de synthèse a été réalisé pour récapituler dans des tableaux les résultats de la littérature par instances. Pour chaque instance ou paquet d'instances, le nom de la méthode ayant fourni le meilleur résultat sur cette instance et éventuellement les bornes supérieures et bornes inférieures sont donnés. Les auteurs proposent alors des critères quantitatifs basés sur la taille des problèmes pour mesurer leur difficulté. Ces observations permettent d'identifier les instances difficiles parmi les problèmes de la littérature ; les auteurs considèrent difficiles les instances ouvertes (i.e. celles pour lesquelles la solution optimale n'est pas connue). Pour finir une comparaison entre les différentes méthodes est proposée. Cette comparaison est étayée par des tableaux comparant la qualité des résultats obtenus en terme de valeur de critère et en terme de temps de résolution.

Le second état de l'art est celui proposé dans l'article de (Blazewicz *et al.*, 1996). Dans cet article, les auteurs décrivent formellement le problème de jobshop à l'aide d'équations linéaires et à l'aide du modèle du graphe conjonctif - disjonctif. Les auteurs présentent l'histoire de la résolution du problème en se focalisant sur la célèbre instance 10x10 de Fisher et Thompson (1963). Ensuite, différentes méthodes sont décrites en séparant les méthodes exactes des méthodes approchées. Les différentes briques nécessaires à l'exécution d'un algorithme exact sont décrites : bornes inférieures, méthodes de branchement, inégalités valides. Pour les algorithmes approchés, les méthodes suivantes sont décrites : quelques règles de priorités, la procédure à machine goulot, des procédures d'ordonnancements opportunistes (Opportunistic scheduling), les algorithmes de recherche locale. Ces derniers sont largement détaillés et les principaux voisinages utilisés pour le problème de jobshop y sont décrits.

4.2 Aperçu historique

Parmi les méthodes efficaces de la littérature, nous avons remarqué que beaucoup utilisent les composants suivants : graphe conjonctif - disjonctif, chemin critique et voisinage guidé. Ces composants utilisent la connaissance que l'on a des ordonnancements pour guider efficacement la recherche. Ils permettent de construire des algorithmes efficaces dédiés aux problèmes de jobshop.

Nous présentons ci-dessous des méthodes utilisant ces composants. Pour chaque grande classe de méthodes, un article représentatif est présenté.

Van Laarhoven *et al.* (1992) proposent un recuit simulé pour le problème de jobshop. La métaheuristique en elle-même est une application assez directe du recuit simulé. Par contre, le voisinage est intéressant car il utilise le chemin critique dans le graphe conjonctif - disjonctif. Le voisinage ainsi construit est un voisinage guidé ciblant les ordonnancements qui peuvent être améliorés en une seule itération (tout voisin obtenu par permutation est améliorant en une itération s'il est dans le voisinage). Pour obtenir un tel voisinage, van Laarhoven *et al.* (1992) utilisent les propriétés des chemins critiques. Ainsi, tout voisin obtenu par une modification qui ne concerne pas le chemin critique est forcément un voisin de moins bonne qualité.

Nowicki et Smutnicki (1996) ont proposé un algorithme tabou basé sur un voisinage très guidé, sous-ensemble des voisins de Laarhoven. Ce voisinage est défini tel que tout voisin améliorant dans le voisinage de Laarhoven est forcément dans le voisinage proposé par Nowicki et Smutnicki. Cette propriété est assurée par la notion de blocs. Un bloc est une sous suite maximale du chemin critique dont toutes les opérations sont traitées par la même machine. Nowicki et Smutnicki ont remarqué que toute permutation qui ne concerne pas les bords de blocs ne peut être améliorante. Une description plus précise de cet algorithme est donnée dans le paragraphe 5.

Dans Mattfeld (1995) et Jensen (2001), les auteurs proposent des algorithmes génétiques hybrides. Les deux algorithmes génétiques ont des structures très différentes. Mais, ils utilisent tous deux une descente dont le voisinage exploite le chemin critique. Ainsi, les deux algorithmes génétiques travaillent sur l'ensemble des optimaux locaux. Cet ensemble est de taille restreinte et comporte des ordonnancements de bonne qualité en moyenne.

4.3 Représentation

Ce paragraphe décrit les principales représentations utilisées dans la littérature pour le problème de jobshop. La notion de représentation en général et les définitions nécessaires sont présentées dans le paragraphe 5 du chapitre 1.

Toutes les représentations sont présentées comme suit : la description commence par une introduction sur la représentation suivie d'un exemple concret de solution stockée. Ensuite les ensembles de solutions codées et de solutions stockées sont décrits, les propriétés des représentations sont données ; le principe des algorithmes de codage et de décodage est donné, puis quelques références d'articles utilisant la représentation sont citées. Tous les exemples représentent la solution définie en tableau 2-3, solution non optimale de l'instance de jobshop dont les caractéristiques sont données en tableau 2-2. Pour aider à la compréhension des solutions stockées, le diagramme de Gantt de la solution S_1 est donné en figure 2-7.

Job 1	O1= Machine 1, Durée 4	O2= Machine 2, Durée 2	O3= Machine 3, Durée 2
Job 2	O4= Machine 2, Durée 7	O5= Machine 3, Durée 3	O6= Machine 1, Durée 2
Job 3	O7= Machine 1, Durée 5	O8= Machine 3, Durée 5	O9= Machine 2, Durée 4

Tableau 2-2. Instance exemple J1

Job 1	0	7	15
Job 2	0	7	10
Job 3	4	10	15

Tableau 2-3. Solution S1 de l'instance J1

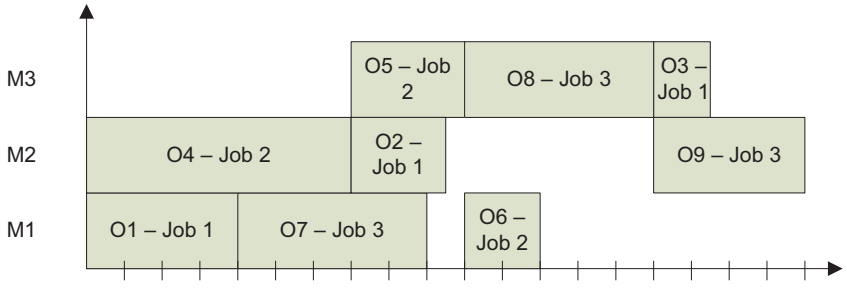

Figure 2-7. Diagramme de Gantt de la solution S1

4.3.1 Représentation directe (R1)

Dans la représentation directe, une solution est stockée "directement" par la solution du problème de départ. Ce type de représentation est très utilisé, et même préconisé, pour les algorithmes

génétiques. Dans le cadre du problème de jobshop, la représentation directe consiste à coder les dates de début des opérations.

Exemple de solutions

Opération	1	2	3	4	5	6	7	8	9
Date de début	0	7	15	0	7	10	4	10	15

Tableau 2-4. Représentation directe de S1

Ensemble des solutions stockées

Le nombre de solutions stockées est très important car toutes les dates de début de toutes les opérations peuvent à priori être envisagées. On peut tout de même calculer une date maximum de fin de l'ordonnancement (UB, borne supérieure du makespan) en sommant les dates de début de toutes les opérations par exemple. En utilisant cette date maximum le nombre d'ordonnancements à parcourir devient fini, mais ce nombre est trop important pour envisager parcourir toutes les solutions stockées. Dans l'exemple, la date maximum est la somme de toutes les durées de traitement ($=34$),et le nombre de solutions stockées est donc $UB^o = 34^9 = 60{,}176.10^{12}$.

Ensemble des solutions codées

Les solutions codées sont tous les ordonnancements du problème. On a donc $\wp = C$.

Codage/Décodage

L'algorithme de codage n'a pas de traitement particulier à réaliser.

L'algorithme de décodage ne nécessite lui aussi aucun calcul dans le cas d'une solution réalisable. Par contre, une solution obtenue par modification d'une solution réalisable risque de ne plus l'être. L'algorithme de décodage doit donc être capable de vérifier la réalisabilité d'une solution et éventuellement la réparer.

Propriétés

- Sur représentation : Oui, car cette représentation permet de stocker tous les ordonnancements y compris les irréalisables.
- Multi représentation : Non, par définition de la représentation directe il n'est pas possible de stocker deux fois la même solution de manière différente.
- Heuristique : Non, car tout ordonnancement peut être représenté, l'ordonnancement optimal peut donc aussi être représenté.

Références bibliographiques

Dans (Nakano et Yamada, 1992), les auteurs utilisent la représentation directe pour les besoins de leur algorithme génétique. Ils utilisent un opérateur de croisement de deux solutions basé sur l'algorithme de Giffler et Thompson. Durant l'exécution de cet algorithme de croisement, les opérations sont choisies en fonction de leur date de début, l'opération la plus prioritaire étant celle de date la plus faible. Grâce à cet opérateur, ils sont capables de ne générer que des solutions réalisables. Le décodage d'une solution est alors instantané.

4.3.2 Représentation semi active (R2)

Dans cette représentation, l'ensemble des solutions codées C est l'ensemble des solutions semi actives. Un ordonnancement semi-actif se caractérise par un ordre sur chaque machine des opérations à traiter sur cette machine. Pour stocker un ordonnancement semi-actif, il suffit donc de stocker l'ordre des opérations sur chaque machine.

Exemple de solutions

Machine 1	1 (job 1)	7 (job 3)	6 (job 2)
Machine 2	4 (job 2)	2 (job 1)	9 (job 3)
Machine 3	5 (job 2)	8 (job 3)	3 (job 1)

Tableau 2-5. Représentation semi active de S1

Ensemble des solutions stockées

Les solutions stockées sont tous les ordres possibles sur chaque machine. Ces ordres contiennent des ordres incompatibles avec l'ordre des opérations de la gamme des jobs. Des solutions irréalisables peuvent donc aussi être stockées. Dans un jobshop, le nombre de solutions stockées est $\prod_{i \in M} e_i!$, qui se simplifie en $(n!)^m$ pour les instances rectangulaires.

Ensemble des solutions codées

Les solutions codées sont les ordonnancements semi-actifs.

Codage/Décodage

L'algorithme de codage consiste à trier les opérations sur chaque machine en fonction de leur date de début. Les ordres ainsi obtenus forment la représentation semi active.

L'algorithme de décodage quant à lui est un peu plus complexe. Le graphe conjonctif - disjonctif détaillé dans le paragraphe suivant (page 76) permet de construire l'ordonnancement semi-actif associé à un ordre.

Propriétés

- Sur représentation : Oui, car tous les ordres représentables ne mènent pas à des ordonnancements semi-actifs. En effet, certains ordres peuvent être incompatibles avec les contraintes du problème telles que les précédences imposées par les gammes des jobs.
- Multi représentation : Non, car deux ordonnancements semi-actifs différents ont deux ordres d'opérations différents et sont donc stockés de manière différente.
- Heuristique : Non, les ordonnancements semi-actifs sont dominants pour tout critère régulier. Pour un critère non régulier, cette représentation est heuristique.

Références bibliographiques

Nous ne présentons pas de liste exhaustive des références utilisant cette représentation car elles sont trop nombreuses. Citons tout de même, Roy et Sussman (1964) qui ont introduit le graphe conjonctif - disjonctif, outil qui a permis l'essor des ordonnancements semi-actifs.

Parmi les articles de la littérature utilisant cette représentation, citons (Nowicki et Smutnicki, 1996) dont l'algorithme tabou est très performant.

4.3.3 Représentation par ordre total d'opérations (R3)

Dans la représentation par ordre total d'opérations, les opérations sont totalement ordonnées dans un unique vecteur. L'ordre relatif des opérations sur les machines est donc (entre autres) complètement déterminé. Cette représentation est sous jacente à d'autres représentations.

Exemple de solutions

1	7	4	5	2	8	6	9	3

1	4	2	7	5	8	3	9	6

Tableau 2-6. Représentation par deux ordres totaux de S1

Ensemble de solutions stockées

L'ensemble de solutions stockées est l'ensemble de toutes les permutations possibles. Le nombre de solutions stockées est donc $o!$ où o est le nombre d'opérations. Ce nombre est largement plus élevé que le nombre de solutions semi actives, ce qui explique le manque d'intérêt porté à cette représentation.

Ensemble de solutions codées

Les solutions codées sont tous les ordonnancements semi-actifs.

Codage/Décodage

Le codage peut être réalisé en triant toutes les opérations suivant leur date de début. De nombreux autres algorithmes sont possibles car l'ordonnancement peut être stocké de plusieurs manières différentes.

Le décodage consiste à prendre la solution stockée comme un ordre strict des opérations. En parcourant cet ordre opération après opération, on peut affecter la date de début de chaque opération de manière définitive. On calcule la date de début comme étant le maximum entre les dates de début des opérations précédentes sur la machine et précédente dans la gamme. L'ordonnancement généré est donc semi-actif car il respecte l'ordre total et les contraintes du problème. Pendant le décodage, il est possible que certains ordres totaux soient incompatibles avec les contraintes du problème.

Propriétés

- Surreprésentation : Oui, car cette représentation permet de générer des ordres incompatibles avec les contraintes de précédence.
- Multi représentation : Oui, car les solutions codées sont toutes des solutions semi actives. Or pour décrire une solution semi active, il suffit de définir un ordre partiel sur les machines. Tous les ordres totaux vérifiant cet ordre partiel et les contraintes de précédence du problème codent la même solution codée.
- Heuristique : Non si la fonction objectif est régulière car la représentation permet de coder toutes les solutions semi actives. Oui si la fonction objectif n'est pas régulière.

Références bibliographiques

Pas de référence bibliographique connue utilisant cette représentation. Par contre, le paragraphe suivant sur le graphe conjonctif - disjonctif (page 76) montre que cette représentation est un intermédiaire de calcul pour l'implantation efficace de nombreuses autres représentations.

4.3.4 Représentation par répétition (R4)

La représentation par répétition peut être construite à partir d'une représentation par ordre total dans laquelle on a remplacé les numéros des opérations par les numéros des jobs. Cette transformation permet de supprimer les informations concernant l'ordre des opérations de la gamme d'un job. On évite ainsi de représenter des solutions ne respectant pas les contraintes de précédence du problème. L'algorithme de décodage reconstruit donc forcément une solution respectant l'ordre des opérations dans la gamme des jobs.

Exemple de solutions

1	3	2	2	1	3	2	3	1

1	2	1	3	2	3	1	3	2

Tableau 2-7. Représentation par répétition de S1

Ensemble des solutions stockées

Les solutions stockées sont toutes les permutations du vecteur de répétition. Comme ce vecteur contient plusieurs fois le même numéro de jobs, le nombre de permutation à considérer est le nombre de permutations avec répétition :

$(\sum_{i\in M} e_i)!/\prod_{i\in J} k_i!$ qui se simplifie en $(nm)!/m!^n$ pour les instances rectangulaires.

Ensemble des solutions codées

Les solutions codées sont tous les ordonnancements semi-actifs.

Codage/Décodage

Le décodage de cette solution s'effectue en deux temps, premièrement les numéros des jobs doivent être remplacés par les numéros d'opérations. Cette opération est très simple et est linéaire en le nombre d'opérations. On obtient alors la représentation par ordre totale correspondante.

Le codage d'une solution est le même que celui de la représentation par ordre totale, excepté que les numéros d'opérations du résultat sont remplacés par leurs numéros de jobs.

Propriétés

- Sur représentation : Non, toute solution stockée représente un ordre réalisable des opérations sur les machines.
- Multi représentation : Oui, car deux ordres totaux correspondant aux même ordre sur les machines forment deux solutions stockées différentes (cf. tableau 2-7).
- Heuristique : Non si la fonction objectif est régulière car la représentation permet de coder les solutions semi actives. Oui si la fonction objectif n'est pas régulière.

Références bibliographiques

Dans (Bierwirth, 1995), l'auteur propose la représentation par répétition et met en œuvre cette représentation pour un algorithme génétique. Grâce à cette représentation et aux opérateurs utilisés, l'auteur ne produit que des solutions réalisables. L'auteur conclut en indiquant que l'algorithme génétique ainsi représenté a des performances équivalentes aux algorithmes génétiques de la littérature. Pourtant l'algorithme génétique de (Bierwirth, 1995) ne contient aucune hybridation ou information particulière au problème.

4.3.5 Représentation binaire (R5)

Cette représentation consiste à stocker l'ordre relatif des paires d'opérations en disjonction. Étant données deux opérations en disjonction i et j, on pose $prior(i,j)=1$ si l'opération i doit être réalisée avant l'opération j, $prior(i,j)=0$ sinon. Pour chaque paire d'opérations en disjonction, on dit qu'il faut arbitrer la disjonction, cet arbitrage définit la valeur de $prior(i,j)$. Les valeurs de $prior(i,j)$ peuvent définir un ordre réalisable des opérations sur les machines, mais elles peuvent aussi définir un ordre irréalisable car contradictoire. Par exemple, la représentation binaire peut définir que l'opération i est avant l'opération j, que l'opération j est avant l'opération k et que l'opération k est avant l'opération i : $prior(i,j) = prior(j,k) = prior(k,i) = 1$.

Pour un ordre réalisable, le modèle de graphe conjonctif - disjonctif permet de construire l'ordonnancement semi-actif correspondant.

Exemple de solutions

Machine 1 :	Prior(1,7)=1	Prior(6,7)=0	Prior(1,6)=1
Machine 2 :	Prior(2,4)=0	Prior(2,9)=1	Prior(4,9)=1
Machine 3 :	Prior(3,5)=0	Prior(3,8)=0	Prior(5,8)=1

Tableau 2-8. Représentation binaire de la solution S1

Ensemble des solutions stockées

Pour dénombrer les solutions stockées, il suffit de compter le nombre de paires d'opérations en disjonction, la représentation comporte une valeur binaire par paire. On compte e_i opérations à traiter sur la même machine, il y a donc $e_i(e_i-1)/2$ paires d'opérations en disjonction sur la machine M_i. Sur l'ensemble des machines, on a donc $\Sigma^{i\in M_i} e_i(e_i-1)/2$ paires d'opérations à arbitrer. Le nombre de solutions possibles est donc $2^{\sum_{i\in M_i} e_i(e_i-1)/2}$ solutions stockées possibles. Dans le cas d'une instance rectangulaire, on a $2^{mn(n-1)/2}$ solutions stockées.

Ensemble des solutions codées

Les solutions codées sont toutes des ordonnancements semi-actifs.

Codage/Décodage

Le codage d'une solution consiste à examiner toutes les paires d'opérations traitées sur la même machine pour positionner la valeur de $prior(i,j)$. Ce codage peut être réalisé facilement en triant les opérations par rapport à leur date de début dans autant de sous ensembles qu'il y a de machines.

Le décodage d'une solution peut être réalisé à l'aide du modèle de graphe conjonctif - disjonctif dans lequel on transforme toutes les arêtes en arcs. Un plus long chemin dans le graphe ainsi obtenu permet soit de détecter un cycle soit de fournir la solution semi active correspondante (cf. §4.4 page 76).

Propriétés

- Sur représentation : Oui, car les ordres des opérations en disjonction non valides ne codent pas des solutions du problème de départ.
- Multi représentation : Non, car si elles sont valides, deux solutions stockées différentes contiennent deux ordres d'opérations différents et donc deux opérations codées différentes.
- Heuristique : Non, si la fonction objectif est régulière car la représentation permet de coder toutes les solutions semi actives. Oui si la fonction objectif n'est pas régulière.

Références bibliographiques

Dans (Nakano et Yamada, 1991), les auteurs utilisent cette représentation pour développer un algorithme génétique. L'avantage de cette représentation est qu'elle est proche des représentations utilisées dans d'autres domaines pour les algorithmes génétiques. Les opérateurs classiques (1-point, 2-point et uniforme) peuvent donc être utilisés. Les ordres ainsi obtenus peuvent bien entendu ne pas être valides, dans ce cas les auteurs proposent un algorithme de réparation dédié qui permet de calculer un ordre valide à l'aide de leur algorithme de réparation appelé "harmonisation globale". L'utilisation de cet algorithme de réparation permet d'obtenir des solutions réalisables.

En utilisant l'algorithme de réparation, on peut remplacer un ordre irréalisable par un ordre irréalisable. Nakano et Yamada (1991) n'utilisent pas systématiquement cette possibilité dans leur algorithme génétique, ce qui aurait permis de supprimer la sur représentativité.

4.3.6 Représentation par liste circulaire (R6)

La représentation par liste circulaire consiste à stocker toutes les opérations dans un vecteur V d'opérations. Chaque élément du vecteur $V[i]$ consiste en un numéro de job. $V[1]=a$ signifie que le $a^{ème}$ job non terminé est le premier job à être ordonnancé le plus tôt possible en respectant les contraintes. Si $V[2]=b$, alors le $b^{ème}$ job non terminé est le deuxième job à être ordonnancé le plus tôt possible en respectant les contraintes. Suivant l'algorithme de décodage choisi, cette représentation permet de coder les ordonnancements actifs ou sans délai.

Cette représentation ressemble à la représentation par répétition, mais l'algorithme de décodage et les solutions stockées différent.

Exemple de solutions

1	2	3	1	2	2	2	1	3
1	2	3	1	2	2	2	1	2

Tableau 2-9. Deux représentations circulaires de la solution S1

Ensemble des solutions stockées

Les solutions stockées sont toutes les permutations du vecteur contenant des numéros de jobs. Ce nombre est différent du nombre de solutions stockées de la représentation par répétition car il n'y a aucune contrainte sur le nombre de fois où un numéro de job peut apparaître dans la liste. En effet, les numéros apparaissant dans la liste se réfèrent aux jobs restant à ordonnancer.

Le nombre de solutions stockées est donc n^o.

Ensemble des solutions codées

Les solutions codées sont les ordonnancements actifs.

Codage/Décodage

L'algorithme de codage consiste à trier les opérations d'après leur date de début et de construire l'ensemble des jobs restant à ordonnancer (triés par leur numéro de jobs). On parcourt ensuite les opérations dans l'ordre, chaque opération est remplacée par la position de son job dans la liste des jobs restant à ordonnancer. À chaque opération, on supprime éventuellement un job si l'opération traitée est la dernière de son job.

L'algorithme de décodage consiste à lire le vecteur de la première case à la dernière. À l'itération i, on détermine le job j comme étant le $V[i]^{ème}$ job à ordonnancer modulo le nombre de jobs qu'il reste à ordonnancer. La prochaine opération du job j est alors traitée au plus tôt en respectant les contraintes du problème.

Propriétés

- Surreprésentation : Non, car cette représentation permet de ne représenter que des ordonnancements actifs.
- Multi représentation : Oui, cf. tableau 2-9
- Heuristique : Non, si la fonction objectif est régulière car la représentation permet de coder toutes les solutions semi actives. Oui si la fonction objectif n'est pas régulière.

Références bibliographiques

Dans (Fang *et al.*, 1993) les auteurs proposent un algorithme génétique basé sur la représentation circulaire pour l'ordonnancement et le réordonnancement des problèmes de jobshop et d'openshop.

4.3.7 Représentation par matrice latine (R7)

Un ordonnancement semi-actif respecte deux ordres : l'ordre des opérations sur les machines et l'ordre des opérations imposé par les gammes des jobs. Fort de cette observation, on peut construire les matrices MO et JO pour représenter ces deux ordres. Ces matrices peuvent être définies pour les instances rectangulaires, la représentation par matrice latine ne s'applique donc qu'aux jobshops rectangulaires. La matrice MO représente l'ordre des machines visitées par un job (i.e. l'ordre imposé par les gammes) et JO représente l'ordre des jobs qui visitent une machine (i.e. l'ordre caractérisant la solution semi active). Formellement, on note $MO=[mo_{ij}]_{(i\in J, j\in M)}$ où mo_{ij} est la position de la machine j dans la gamme du job i. De même, on note $JO=[jo_{ij}]_{(i\in J, j\in M)}$ où jo_{ij} est la position à laquelle le job i est traité sur la machine j. Dans l'exemple proposé paragraphe suivant, on lit donc $mo_{21}=3$ car le job $J2$ visite la machine $M1$ en $3^{ème}$ position. De même, $jo_{21}=3$ car le job $J2$ est traité en $3^{ème}$ position par la machine $M1$.

Une matrice latine notée $LR[n,m,r]=[a_{ij}]$ est une matrice $n\times m$ dont les éléments font partie de l'ensemble $[1;r]$ de telle manière que chaque élément n'apparaisse qu'une seule fois dans chaque ligne et qu'une seule fois dans chaque colonne. On considère dans la suite seulement les matrices latines qui satisfont de plus la propriété suivante : si $a_{ij}>1$ alors $a_{ij}-1$ apparaît au moins une fois dans la ligne i ou dans la colonne j. On appellera "matrice latine contrainte" les matrices latines respectant cette contrainte supplémentaire.

Les matrices MO et JO permettent de construire une unique solution semi active, or ces deux matrices peuvent être synthétisées dans une unique matrice latine contrainte (cf. (Bräsel, 1990) pour la preuve en allemand de l'unicité). La matrice latine code dans une même matrice les informations relatives à l'ordre de traitement des opérations sur les machines et l'ordre des opérations imposé par la gamme du job. Dans cette matrice, on lit en ligne ou en colonne. Dans la colonne i, l'ordre dans lequel les nombres apparaissent est l'ordre dans lequel les jobs sont traités par la machine i. Le tableau 2-12 contient la matrice latine contrainte de la solution S_1. La colonne 1 contient (1, 3, 2) ce qui correspond à l'ordre des jobs $J1$, $J3$ et $J2$ traités par la machine $M1$. La ligne 3 contient (2, 4, 3), ce qui indique que le job 3 visite les machines $M1$, $M3$ et $M2$ dans cet ordre.

Exemple de solutions

MO	M1	M2	M3
J1	1	2	3
J2	3	1	2
J3	1	3	2

Tableau 2-10. Matrice MO de la solution S1

JO	M1	M2	M3
J1	1	2	3
J2	3	1	1
J3	2	3	2

Tableau 2-11. Matrice JO de la solution S1

LR	M1	M2	M3
J1	1	2	4
J2	3	1	2
J3	2	4	3

Tableau 2-12. Matrice latine LR[3, 3, 4] de la solution S1

Sur cet exemple on peut voir pourquoi il est nécessaire de contraindre les matrices latines pour avoir une relation un à un avec les ordonnancements semi-actifs. En effet, il suffit d'ajouter un à toutes les cases de la matrice pour obtenir une matrice dont les ordres sont équivalents.

Ensemble des solutions stockées

Les solutions stockées sont toutes les matrices latines contraintes $LR[n,m,.]$. Ce nombre n'est pas facilement calculable de manière formelle, mais on peut tout de même le borner. Le nombre de matrices latines dépend de la valeur de r, le troisième paramètre permettant de définir $LR[n,m,r]$. Pour borner le nombre de matrices latines, il faut donc borner r. Au minimum, r est égal à la plus grande quantité entre le nombre de jobs n et le nombre de machines m. Au maximum, r est égal au nombre d'opérations $o = nm$. Dans chaque case de la matrice, on peut mettre une valeur entre 1 et r. La matrice contient $o = nm$ cases, on obtient donc un total de r^{nm} borné par $\max(n,m)^{nm} \leq r^{nm} \leq nm^{nm}$.

Ensemble des solutions codées

Les solutions codées sont les ordonnancements semi-actifs.

Codage/Décodage

L'algorithme de codage consiste à construire les matrices MO et JO puis à en déduire la matrice latine. La construction des matrices MO et JO est triviale et a été expliquée dans les représentations précédentes. Pour la construction de la matrice latine, cela consiste en les opérations suivantes. On recopie la matrice MO dans la matrice $LR[n,m,m]$. On parcourt ensuite les colonnes de JO et LR en parallèle. Si l'ordre dans lequel les éléments de JO apparaissent est le même que l'ordre des éléments de LR alors on passe à la colonne suivante. Sinon, on renumérote les éléments de LR de manière à ce qu'ils apparaissent dans le même ordre que les éléments de JO. Les éléments sont renumérotés en ajoutant la plus faible quantité à chaque élément pour obtenir l'ordre souhaité. Ainsi, dans l'exemple S_1 la colonne 1 est (1, 3, 1) dans l'ordre fourni par MO. Or l'ordre souhaité dans JO est (1, 2, 3). La colonne 1 est donc renumérotée (1, 3, 4). Lorsqu'un élément a été renuméroté, la ligne doit être actualisée pour vérifier que l'ordre de MO est toujours valide. Cet algorithme peut être réalisé à la main sur de petits exemples, mais nous ne connaissons pas d'implantation efficace. Les auteurs (Werner et Winkler, 1995) restent discrets à ce sujet.

L'algorithme de décodage consiste à déterminer un ordonnancement semi-actif à partir de la matrice latine. Cet algorithme ressemble fortement aux algorithmes de plus longs chemins connus car la matrice latine contrainte est fortement liée à la notion de graphe conjonctif - disjonctif. Pour une opération donnée (i,j), on peut calculer sa date au plus tôt r_i et sa date au plus tard q_i. Les dates au plus tôt s'obtiennent en initialisant toutes les dates de début des opérations à la somme des durées des opérations précédentes dans la gamme. Ensuite, on peut parcourir les lignes et colonnes de la matrice pour connaître les opérations précédent l'opération (i,j) car ce sont celles qui ont un numéro inférieur. De même que pour l'algorithme de codage, les auteurs sont discrets sur la façon d'implanter efficacement cet algorithme.

Pour les besoins de cette thèse, les algorithmes de codage et de décodage ont été implantés en transformant la représentation par matrice latine contrainte en représentation semi active.

Propriétés

- Sur représentation : Non, d'après (Bräsel, 1990) il y a une relation un à un entre les ordonnancements semi-actifs codés et les matrices latines contraintes stockées.
- Multi représentation : Oui, car les matrices latines générées ne sont pas toutes des matrices latines contraintes. Il est donc possible de créer plusieurs matrices latines différentes. Par contre, pour une matrice latine donnée, on peut calculer la matrice latine contrainte correspondante. Il existe alors une correspondance un à un entre les matrices latines contraintes et les solutions codées (i.e. ordonnancements semi-actifs). La représentation par matrice latine est donc multi représentée restreinte.
- Heuristique : Non, si la fonction "objectif" est régulière car la représentation permet de coder toutes les solutions semi actives. Oui si la fonction "objectif" n'est pas régulière.

Références bibliographiques

La représentation par matrice latine a été proposée par Bräsel (1990). Werner et Winkler (1995) ont réutilisé cette représentation pour le développement d'heuristiques dédiées au problème de jobshop. Deux heuristiques sont proposées par les auteurs, la première est une heuristique de construction qui insère itérativement de nouveaux éléments dans la matrice latine. La seconde heuristique est un algorithme de "beam search" basé sur l'heuristique d'insertion.

4.3.8 Représentation active(R8)

Dans (Giffler et Thompson, 1960) les auteurs proposent un algorithme pour construire des ordonnancements actifs. Cet algorithme consiste à construire l'ordonnancement itérativement opération après opération. A chaque étape, on recherche l'opération dont l'exécution est prévue au plus tôt, et on détermine l'ensemble des opérations entrant en conflit avec l'opération trouvée. Giffler et Thompson (1960) propose de choisir cette opération en fonction d'une règle prédéterminée. Dans le cas d'une représentation active, l'opération est choisie en fonction d'une priorité prédéterminée : chaque opération se voit attribuer une priorité et quand plusieurs opérations sont présentes dans l'ensemble des opérations en conflit, on choisit celle de priorité la plus forte (ou la plus élevée selon les cas).

Les solutions stockées pour cette représentation peuvent être de format très différents suivant les auteurs. Il peut s'agir d'un vecteur associant une valeur numérique à chaque opération. Un vecteur contenant une permutation de la liste des opérations peut être utilisé, l'opération en position i dans le vecteur est alors une opération de priorité i. Le vecteur d'opération peut alors être vu comme l'ordre préféré des opérations menant à un ordonnancement actif.

Exemple de solutions

	O1	**O2**	**O3**	**O4**	**O5**	**O6**	**O7**	**O8**	**O9**
Priorités	12	-5	2	4	56	7	5	1	0
Ordre 1	1	7	6	4	2	9	5	8	3
Ordre 2	1	4	2	7	5	6	8	9	3

Tableau 2-13. Représentations actives de la solution S1

Les représentations ci-dessus utilisent des priorités avec valeurs numériques (première ligne) ou avec ordres (cf. deux exemples sur les deux dernières lignes).

Ensemble des solutions stockées

Les solutions stockées dépendent de la façon dont les priorités sont définies. En général, les priorités basées sur des valeurs numériques sont très nombreuses. Les priorités basées sur l'ordre des opérations définissent $o!$ solutions stockées différentes.

Ensemble des solutions codées

Les solutions codées sont les ordonnancements actifs.

Propriétés

- Sur représentation : Non, de par sa nature, l'algorithme de Giffler et Thompson ne peut que produire des ordonnancements actifs et donc réalisables.
- Multi représentation : Oui, car de nombreuses priorités différentes mènent au même ordonnancement.
- Heuristique : Non, si la fonction objectif est régulière car la représentation permet de coder toutes les solutions actives. Oui si la fonction objectif n'est pas régulière.

Références bibliographiques

Les articles utilisant cette représentation sont nombreux, les auteurs veulent tirer parti du relativement faible nombre d'ordonnancements actifs existant.

Dans (Dorndorf et Pesh, 1993), les auteurs proposent d'utiliser cette représentation dans laquelle les opérations ordonnancées en premières sont celles dont la date de début dans la solution stockée est la plus faible. On peut dire dans ce cas, que les auteurs ont choisi une représentation directe dans laquelle l'algorithme de décodage est l'algorithme de Giffler et Thompson basé sur les priorités.

4.3.9 Représentation sans délai (R9)

L'algorithme proposé dans (Giffler et Thompson, 1960) permet aussi de déterminer des ordonnancements sans délais. Pour cela, il suffit de modifier légèrement la façon de choisir les opérations à chaque étape. Dans la version sans délai, on choisit d'abord l'opération à ordonnancer dont la date de début est la plus faible. Les opérations qui peuvent commencer à la même date et sur la même machine forment l'ensemble des opérations en conflit. Finalement une règle doit permettre de choisir une opération dans l'ensemble des opérations en conflit, cette règle est généralement basée sur la notion de priorité.

Tout comme pour la représentation active, il est possible de définir les priorités de plusieurs manières différentes : attribuer une valeur numérique à chaque opération ou trier les opérations dans un ordre définissant leurs priorités.

Exemple de solutions

	O1	**O2**	**O3**	**O4**	**O5**	**O6**	**O7**	**O8**	**O9**
Priorités	12	-5	2	4	56	7	5	1	0
Ordre 1	1	7	6	4	2	9	5	8	3
Ordre 2	1	4	2	7	5	6	8	9	3

Tableau 2-14. Représentations sans délai de la solution S1

Les représentations ci-dessus utilisent des priorités avec valeurs numériques (première ligne) ou avec ordres (cf. deux exemples sur les deux dernières lignes).

Ensemble des solutions stockées

Les solutions stockées dépendent de la façon dont les priorités sont définies. En général, les priorités basées sur des valeurs numériques sont très nombreuses. Les priorités basées sur l'ordre des opérations définissent $o!$ solutions stockées différentes.

Ensemble des solutions codées

Les solutions codées sont les ordonnancements sans délai.

Propriétés

- Sur représentation : Non, de par sa nature, l'algorithme de Giffler et Thompson ne peut que produire des ordonnancements sans délai et donc réalisables.
- Multi représentation : Oui, car de nombreuses priorités différentes mènent au même ordonnancement.
- Heuristique : Oui, il n'est pas nécessaire qu'il existe un ordonnancement sans délai dont le critère est optimal.

Références bibliographiques

Cette représentation n'est pas largement utilisée dans la littérature car elle est heuristique : en effet, tout algorithme basé sur cette représentation risque de ne pas pouvoir trouver la solution optimale. Ce désavantage est contrebalancé par le fait qu'il existe peu d'ordonnancements sans délais et qu'en moyenne, ces ordonnancements sont de bonne qualité.

4.3.10 Synthèse des représentations

Le tableau 2-15 présente la synthèse des propriétés des représentations présentées ci-dessus. Pour chaque représentation, on précise son nom (colonne Nom), l'ensemble des solutions codées C qui peut être Ω (tous les ordonnancements possibles), SA (les ordonnancements semi-actifs), A (les ordonnancements actifs), SD (les ordonnancements sans délai). La colonne (Sur rep.) contient O si la représentation est sur représentative, N sinon. La colonne (Multi rep.) contient O si la représentation est multi représentative, N sinon. De plus, elle contient O_r si la représentation est multi représentée restreinte. Dans la colonne (heur.), un N signifie que la représentation n'est pas heuristique et un N_{reg} précise que la représentation n'est pas heuristique si le critère d'optimisation est régulier. La colonne (Sol. part.) précise si la représentation permet d'évaluer une solution partiellement définie (O) ou non(N). La colonne contient O_G si la solution peut être obtenue par un algorithme glouton, i.e. si une solution partielle peut être complétée en une solution optimale sans modifier la solution partielle. La possibilité pour une représentation de coder des solutions partielles est particulièrement intéressante pour les algorithmes constructifs et certaines méthodes de séparation / évaluation (branch and bound). La colonne (Nb. sol. stockées) donne formellement le nombre de solutions stockées pour chaque représentation, et la colonne (Nb. sol. stockées dans J_1) indique le nombre de solutions stockées pour l'instance de jobshop J_1.

Le tableau 2-15 donne une vue générale des différentes représentations rencontrées dans la littérature pour le problème de jobshop. Il apparaît clairement qu'aucune représentation n'est meilleure que les autres. Il est donc important de choisir la représentation de manière attentive en fonction des caractéristiques du problème.

	Nom	C	Sur Rep.	Multi. Rep.	Heur.	Sol. part.	Recirc	Nb. sol. stockées	Nb. sol. stockées pour J1
R1	directe	Ω	O	N	N	O	O	UB^o	$48^9 = 1{,}352.10^{15}$
R2	semi active	SA	O	N	Nreg	O	O	$\prod_{i\in M} e_i!$	216
R3	ordre total	SA	O	O	Nreg	O	O	$o!$	362880
R4	répétition	SA	N	O	Nreg	N	O	$\frac{(\sum_{i\in M} e_i)!}{\prod_{i\in J} k_i!}$	1 680
R5	binaire	SA	O	N	Nreg	O	O	$2^{\sum_{i\in M_i} e_i(e_i-1)/2}$	512
R6	liste circulaire	A ou SD	N	O	Nreg	OG	O	n^o	19 683
R7	matrice latine	SA	N	Or	Nreg	O	N	r^{nm} où $\max(n,m)^{nm} \le r^{nm}$ $r^{nm} \le nm^{nm}$	19 683 ≤ ≤ 387 420 489
R8	actif	A	N	O	Nreg	OG	O	$o!$	362880
R9	sans délai	SD	N	O	O	OG	O	$o!$	362880

Tableau 2-15. Synthèse des propriétés des représentations

Les différentes représentations et leurs utilisations dans les articles de la littérature font apparaître que le choix de l'ensemble des solutions codées n'est pas trivial. Les solutions codées et les solutions du problème sont assez souvent différentes, nous avons observé les raisons suivantes :

- Soit on connaît un ensemble de solutions dominant les solutions de $\wp$. Dans ce cas, il est souhaitable de ne parcourir que ces solutions, et toute représentation permettant de coder uniquement les solutions de S est la bienvenue. Ce cas s'observe par exemple pour les représentations semi actives et actives, car les ordonnancements générés dominent les autres ordonnancements pour les critères réguliers (cf 4.3.2).
- Soit les solutions de S sont meilleures, en moyenne, que les solutions de $\wp$. Ceci peut d'ailleurs avoir été démontré théoriquement, expérimentalement ou non démontré. La représentation par des ordonnancements sans délai est un exemple illustrant cette situation : les ordonnancements sans délai ne dominent pas d'autres ordonnancements, mais sont meilleurs que les autres ordonnancements (même si cela n'a été prouvé qu'expérimentalement).
- Soit les codages des solutions de $\wp$ ne sont pas adaptés à l'algorithme d'optimisation envisagé. Les solutions de C peuvent alors être introduites pour faciliter l'exploration de l'espace de recherche.

On observe aussi que même s'il existe des représentations dominantes (dont le nombre de solutions codées est faible) ou des représentations compactes (dont le nombre de solutions stockées est faible) ; il n'en reste pas moins que les autres représentations sont toujours utilisées. Les raisons d'une telle diversification tiennent dans les remarques suivantes :

Les représentations qui ne sont pas sur représentatives (répétition, liste circulaire, matrice latine, actif et sans délai) sont toutes multi représentatives. Or cette caractéristique est gênante pour les algorithmes évolutionnistes. En effet, dans le principe, un algorithme génétique ne doit pas pouvoir créer de nouveaux individus à partir de deux solutions identiques stockées de manière différente.

Inversement, les représentations qui ne sont pas multi représentatives ont toutes comme point commun d'être sur représentatives (représentations directe, semi active et binaire).

Cette synthèse illustre bien les caractéristiques du problème de jobshop : on ne sait pas représenter exactement les solutions du problème : des deux maux, il faut choisir le moindre multi ou sur représentativité. Ce choix se fait par rapport à l'algorithme de résolution utilisé. Par exemple, un algorithme d'énumération tend à souhaiter le moins de solutions stockées possibles. Alors qu'un algorithme évolutionniste tend à vouloir ne représenter qu'une seule fois une même solution pour éviter que deux codées de manière différente et représentant la même solution n'entrent en compétition.

4.4 Graphe conjonctif-disjonctif

Le modèle du graphe conjonctif-disjonctif a été introduit par Roy (1964). Ce modèle utilise un algorithme de plus longs chemins pour déterminer les dates de début des opérations. Le modèle de graphe conjonctif-disjonctif est utilisé pour représenter les problèmes disjonctifs, c'est-à-dire les problèmes dans lesquels certaines opérations ne peuvent être réalisées en même temps car elles utilisent une ou plusieurs ressources en commun. Dans ce cas, on dit que ces opérations sont "disjointes". On appelle "disjonction" le fait que deux opérations se partagent une même ressource et ne peuvent être traitées que dans des intervalles de temps disjoints. Deux opérations en disjonction doivent être arbitrées, c'est-à-dire que l'ordre de passage sur la ressource commune doit être déterminé. On parle donc de disjonction arbitrée et de disjonction non arbitrée.

Une sélection d'un problème disjonctif consiste à arbitrer toutes ses disjonctions. Si toutes les disjonctions sont arbitrées de manière cohérente, on dit que la sélection est valide (cf. définition ci-dessous) et on obtient un ordre de passage des opérations sur chacune des ressources. Une sélection peut être non valide car elle impose des ordres qui ne sont pas compatibles. La figure 2-8 montre un exemple de trois opérations en disjonction. Il y a $3!=6$ manières d'ordonner ces 3 opérations (nombre de permutations de 3 éléments). Or il y a $2^3=8$ sélections possibles (nombre de combinaisons possibles). Il est donc évident que certaines sélections ne sont pas valides, d'ailleurs la sélection de la

figure propose un ordre non valide. La sélection de la figure met l'opération 1 avant l'opération 2, et l'opération 2 avant l'opération 3. Il serait judicieux d'arbitrer la disjonction entre 1 et 3 en mettant l'opération 1 avant l'opération 3. Or la sélection propose le contraire. La détermination de disjonction cohérente (i.e. sélection valide) n'est pas toujours facile.

Sélection (ou ordre) Valide

Une sélection (ou ordre) valide est une orientation de toutes les disjonctions du problème telle que le graphe des orientations soit acyclique.

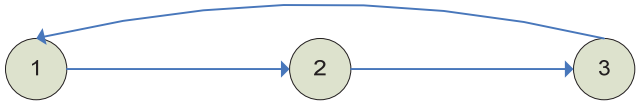

Figure 2-8. Sélection non valide de trois opérations

Une sélection valide permet d'orienter le graphe de manière cohérente. Mais rien n'indique que cette orientation sera associée à une solution. En effet, certaines orientations créent des cycles dans le graphe. Lorsque ces cycles sont de longueur positive, on dit qu'ils sont absorbants (cf. définition ci-dessous). Dans ce cas, il n'y a plus de plus longs chemins dans le graphe car tout chemin passant par un nœud du cycle absorbant peut être rallongé par autant de tours de cycle que voulu.

Cycle absorbant

C est un cycle absorbant pour les plus longs chemins si et seulement si :

C est un cycle

la longueur du cycle C est strictement positive

Il ne faut pas donc pas confondre les graphes contenant des cycles et ceux qui contiennent des cycles absorbants. A priori ces deux notions sont différentes : on peut calculer les plus longs chemins pour un graphe qui contient un cycle, on ne le peut pas pour un graphe contenant un cycle absorbant. Du point de vue des algorithmes, certains sont spécialisés dans les graphes acycliques, d'autres dans les graphes sans cycle absorbant et d'autres capables de détecter les cycles absorbants.

Pour le problème de jobshop classique, tous les arcs sont de longueur strictement positive. Tous les cycles sont donc de longueur strictement positive. Ainsi, pour le jobshop classique, tous les cycles sont absorbants et une solution est associée à chaque graphe conjonctif acyclique. C'est pourquoi les représentations n'envisageant que des sélections valides conduisent systématiquement à des solutions. C'est le cas par exemple de la représentation semi active.

Dans la suite, on détaille la construction des deux graphes : le graphe disjonctif (§4.4.1) qui modélise le problème et dans lequel à priori aucune disjonction n'est arbitrée et le graphe conjonctif (§4.4.2) qui modélise une solution, c'est-à-dire un ordonnancement semi-actif.

4.4.1 Graphe disjonctif

Job 1	O1=Machine 1, Durée : 4	O2=Machine 2, Durée : 5	O3=Machine 3, Durée : 3
Job 2	O4=Machine 2, Durée : 7	O5=Machine 3, Durée : 3	O6=Machine 1, Durée : 3
Job 3	O7=Machine 1, Durée : 5	O8=Machine 3, Durée : 10	O9=Machine 2 Durée : 8

Tableau 2-16. Instance exemple J1

On note $G=(V,A,E)$ un graphe disjonctif, où V est un ensemble de nœuds, A est un ensemble d'arcs et E est un ensemble d'arêtes. Les ensembles V, A et E sont construits de la manière suivante :

- Pour chaque opération du problème de jobshop, on crée un nœud dans V. Pour faciliter les explications, on pourra donc confondre les nœuds et les opérations. On crée de plus deux nœuds 0 et $*$ qui modélisent deux opérations fictives. Le nœud 0 représente une opération fictive réalisée avant toutes les autres opérations, on dit que ce nœud est la source du graphe. Le nœud $*$ représente une opération fictive réalisée après toutes les autres opérations, on dit que ce nœud est le puits du graphe.
- On crée un arc dans A entre deux opérations consécutives dans la gamme du job. Ainsi, pour chaque opération de nœud i, on crée un arc du nœud i vers le nœud $i+1$ où $i+1$ est l'opération suivante dans la gamme du job. Cet arc est de longueur p_i (i.e. la durée de traitement de l'opération i).
- On crée une arête dans V entre deux opérations traitées par la même machine. Cette arête représente la disjonction entre ces deux opérations et devra être orientée pour construire un graphe conjonctif. Il y a donc un ensemble d'arêtes reliant toutes les opérations à réaliser sur la même machine.

Pour illustrer la construction du graphe disjonctif, le tableau 2-16 propose une instance de jobshop J_1. La figure 2-9 présente le graphe disjonctif associé.

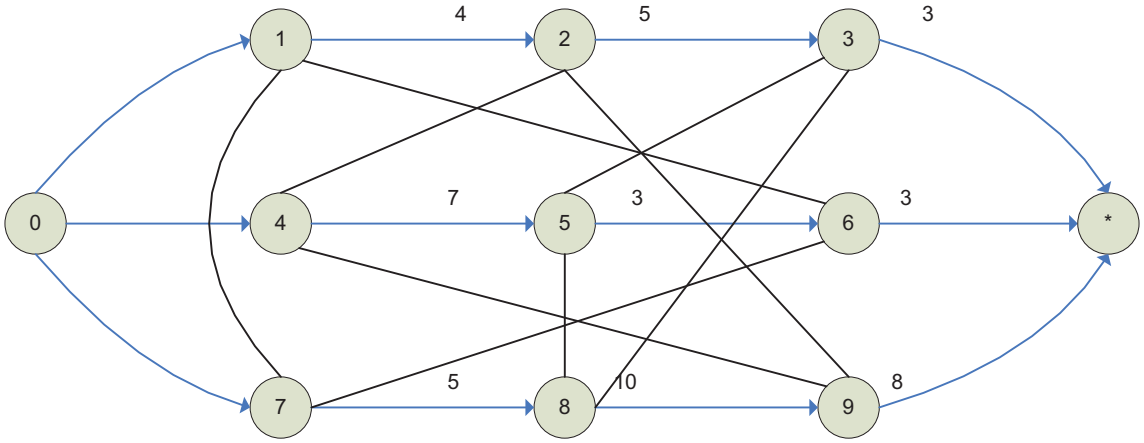

Figure 2-9. Graphe disjonctif du problème J1

4.4.2 Graphe conjonctif

À partir d'une représentation semi active, d'une représentation par ordre total d'opérations, d'une représentation par répétition ou d'une représentation binaire, on peut orienter les arêtes du graphe disjonctif. Orienter une arête entre deux opérations consiste à remplacer cette arête par un arc dans le sens de l'orientation. La longueur de cet arc est la durée de traitement de l'opération de départ de l'arc. Le tableau 2-17 est une sélection valide, correspondant à un ordonnancement S_1 de l'instance J_1. Le graphe conjonctif est présenté en figure 2-10.

Machine 1	1<7	7<6	1<6
Machine 2	4<2	2<9	4<9
Machine 3	5<8	8<3	5<3

Tableau 2-17. Solution S1 de l'instance J1

Le graphe conjonctif ainsi obtenu contient de nombreux arcs, car il contient un arc pour chaque paire d'opérations en disjonction. Il n'est pas nécessaire d'avoir autant d'arcs, comme le schématise la figure 2-11. Dans cette illustration, on a quatre opérations deux à deux en disjonction, comme il y a quatre opérations, il y a $4*(4-1)/2=6$ arcs. Parmi ces arcs, seuls les arcs de 1 vers 2, de 2 vers 3 et de 3 vers 4 sont nécessaires. Les autres arcs ne peuvent participer à un plus long chemin, par exemple, tout

plus long chemin qui emprunterait l'arc de 1 vers 3, pourrait être rallongé en passant par 2. On peut donc simplifier le graphe par réduction transitive des arcs conjonctifs, le graphe ainsi obtenu est appelé graphe conjonctif. La figure 2-12 présente le graphe ainsi réduit.

On peut directement obtenir le graphe conjonctif en transformant la sélection en un ordre des opérations. Puis pour chaque paire d'opérations successives dans cet ordre, on crée un arc allant d'une opération vers la suivante. Pour illustrer cette construction, le tableau 2-18 donne l'ordre des opérations correspondant à la sélection du tableau 2-17. Sur la machine 2, l'ordre des opérations est $4\,2\,9$. On crée donc dans le graphe conjonctif les arcs de $4 \rightarrow 2$ et de $2 \rightarrow 9$. Le graphe conjonctif correspondant à cet exemple est présenté dans la figure 2-13.

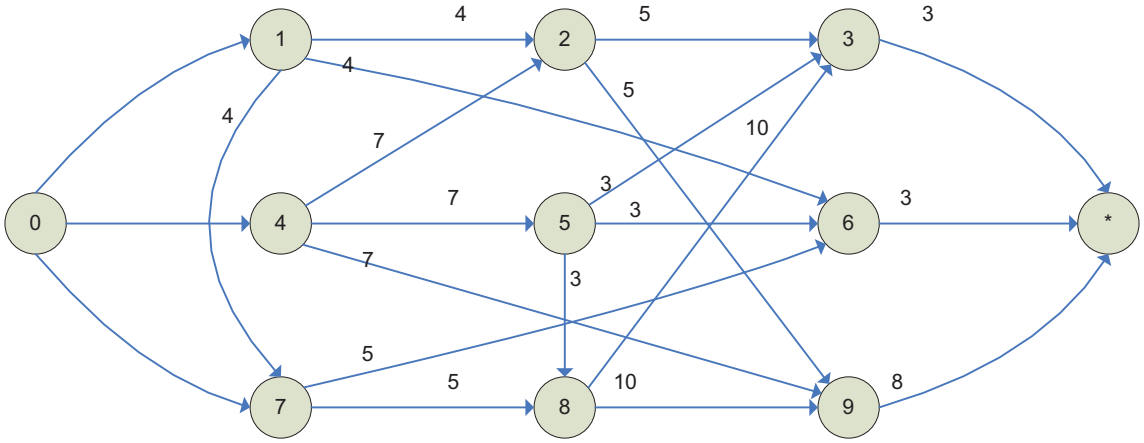

Figure 2-10. Graphe conjonctif orienté de la solution S1

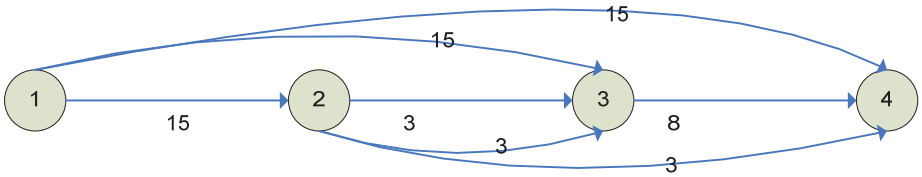

Figure 2-11. Quatre opérations en disjonction dont 4 arcs redondants

Figure 2-12. Réduction transitive du graphe précédent

Machine 1	1 7 6
Machine 2	4 2 9
Machine 3	5 8 3

Tableau 2-18. Ordre S1 de l'instance J1

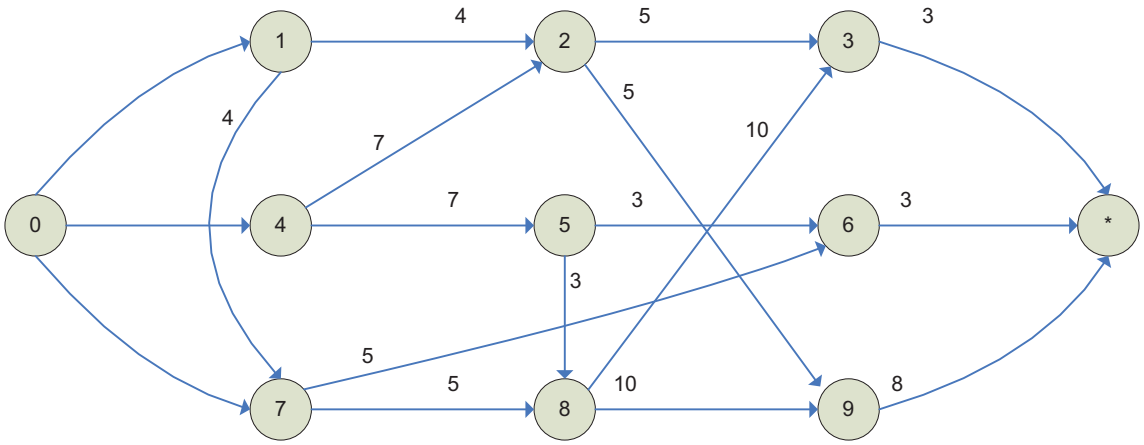

Figure 2-13. Graphe conjonctif de la solution S1

4.4.3 Évaluation du graphe

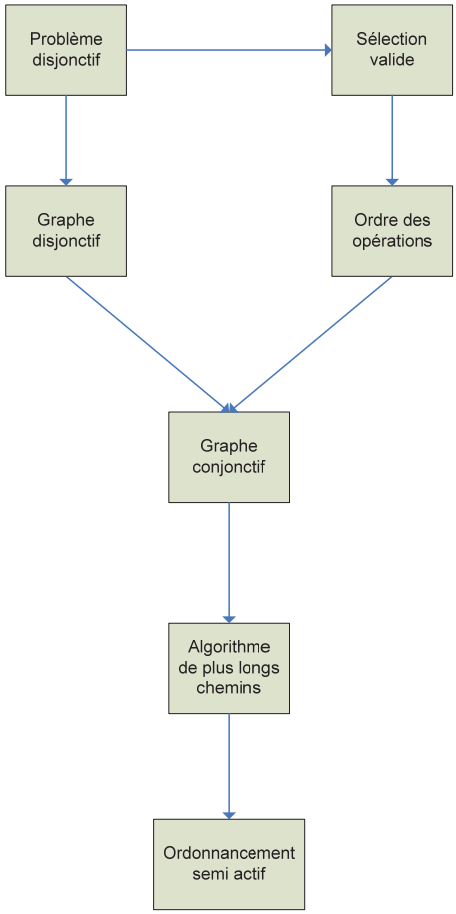

Figure 2-14. Vue d'ensemble du modèle de graphe conjonctif - disjonctif

Le modèle de graphe conjonctif - disjonctif permet de calculer l'ordonnancement semi-actif associé à une sélection. L'obtention de l'ordonnancement semi-actif passe donc par les étapes suivantes (cf. figure 2-14). Le problème de départ est analysé pour construire le graphe conjonctif - disjonctif. Puis, une sélection valide est considérée. Cette sélection peut être fournie par un expert, peut être calculée ou obtenue par un algorithme d'optimisation. La sélection valide permet ensuite de construire l'ordre des opérations. Le graphe conjonctif - disjonctif et l'ordre des opérations permettent de

construire le graphe conjonctif. Ensuite, un algorithme de plus longs chemins permet de calculer l'ordonnancement semi-actif associé à la sélection valide.

Job 1	0	7	20
Job 2	0	7	10
Job 3	4	10	20

Tableau 2-19. Solution S1 de l'instance J1

L'exemple de la solution *S1* est évalué dans le graphe de la figure 2-15. La date de début de l'opération * est la date de fin de la dernière opération, c'est-à-dire le makespan. Le makespan de cette solution est donc *28*.

De nombreux algorithmes de plus longs chemins sont disponibles dans la littérature, mais tous n'ont pas la même complexité. Parmi les algorithmes possibles, on cherche donc à appliquer celui de plus faible complexité et qui soit compatible avec les caractéristiques du graphe. Les deux paragraphes suivants sont des implantations de deux algorithmes de plus longs chemins adaptés aux graphes conjonctifs. Tous deux sont capables de détecter les cycles absorbants mais s'arrêtent dès que le cycle absorbant est détecté. D'autres algorithmes de plus longs chemins existent qui sont capables de déterminer les plus longs chemins élémentaires dans un graphe avec des cycles absorbants, mais ce problème étant NP-difficile, les algorithmes connus ne sont pas performants. Nous n'utilisons donc pas ce type d'algorithme.

Le calcul des plus longs chemins dans le graphe conjonctif-disjonctif est un problème polynomial (dans la mesure où l'on s'arrête dès qu'un cycle est détecter). La complexité des meilleurs algorithmes connus est linéaire en fonction du nombre d'arcs. Dans le cas général, la complexité dans le pire des cas de ce problème est polynomiale (en O(*nm*) où *n* est le nombre d'arcs et *m* est le nombre de noeuds). Dans les deux paragraphes suivants, nous présentons deux implantations d'algorithmes de plus longs chemins, la première est la plus générale et permet de traiter tout type de graphe, sa complexité est donc O(*nm*). La seconde est un peu particulière et suppose que l'on connaît un ordre topologique du graphe. Dans ce cas, on peut calculer les plus longs chemins en O(*n*) où *n* est le nombre d'arcs.

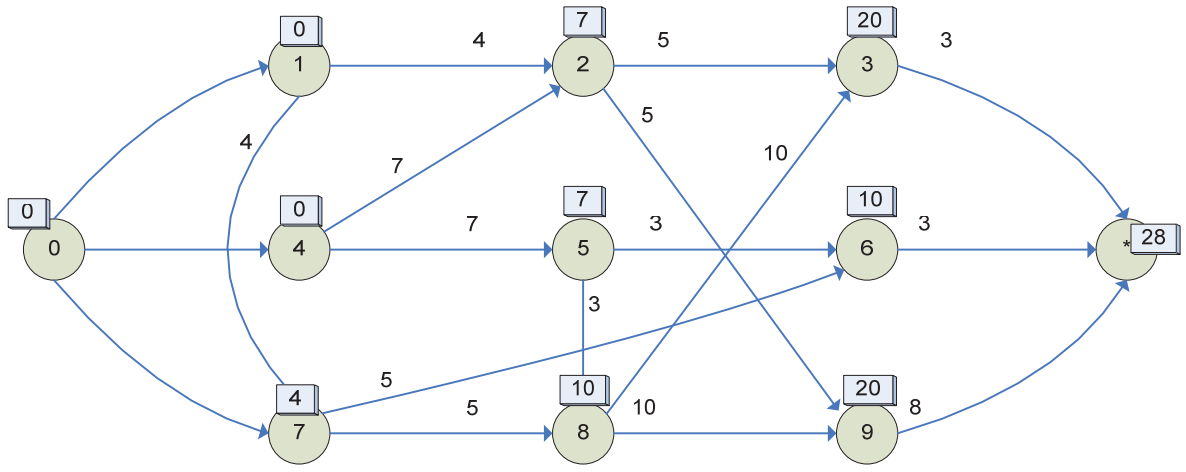

Figure 2-15. Graphe conjonctif évalué

4.4.4 Implantation de Bellman-Ford

L'algorithme de plus long chemin le plus général est celui de Bellmann-Ford. Celui-ci peut être utilisé pour tous les graphes conjonctifs et permet aussi la détection des cycles absorbants. La

complexité dans le pire des cas de cet algorithme est O(*nm*) où *n* est le nombre d'arcs et *m* est le nombre de nœuds.

Nous présentons dans l'algorithme 2-1, une implantation générale de cet algorithme. Dans cette implantation, on parcourt tous les arcs du graphe (à l'aide des deux boucles "pour"). Ce parcours permet de faire la mise à jour de toutes les dates de début. Si au cours de ce parcours, au moins une date de début a été modifiée (le booléen fin est alors positionné à faux), alors on recommence le parcours grâce à la boucle "tant que". L'algorithme s'arrête dès qu'un parcours des arcs a été réalisé sans qu'aucun d'entre eux n'ait été mis à jour. La deuxième condition d'arrêt permet de détecter les cycles absorbants. En effet, si un cycle absorbant existait dans le graphe, chaque itération permettrait de faire une mise à jour et l'algorithme serait alors infini. Grâce à la deuxième condition du "tant que", l'algorithme s'arrête au bout de n itérations (où n est le nombre de nœuds dans le graphe). D'après Bellman-Ford, on sait que la n+1 ème itération n'est pas nécessaire sauf si le graphe contient un cycle.

```
fin=faux;
cnt=0;
Tant que non fin et cnt < n faire
   fin = vrai;
   Pour tout nœud de i faire
         Pour tout arc issu de i et allant vers j faire
               si  t_i + p_i > t_j  alors
                     t_j = t_i + p_i ;
                     fin = faux;
               finsi
         finpour
   finpour
   cnt = cnt + 1;
fintantque
```

Algorithme 2-1. Implantation naïve de l'algorithme de Bellman-Ford

Cette implantation est très générale et ne s'appuie sur aucune propriété particulière du graphe. Dans la section suivante, nous présentons un algorithme beaucoup plus efficace pour le problème de jobshop qui exploite l'absence de cycle dans le graphe.

4.4.5 Implantation de Bellmann basée sur le tri topologique

Pour le problème de jobshop, l'algorithme de plus long chemin avec la plus faible complexité est l'algorithme de Bellmann basé sur le tri topologique des nœuds. En effet, cet algorithme permet de calculer les plus longs chemins en parcourant une et une seule fois chaque arc du graphe. Sa complexité est donc linéaire en fonction du nombre d'arcs (O(*n*) où *n* est le nombre d'arcs).

Pour pouvoir utiliser cette implantation, il est nécessaire de connaître un ordre topologique du graphe. Or, il n'est pas toujours nécessaire d'avoir recours au calcul de l'ordre topologique. Dans les approches classiques pour le problème de jobshop, il est usuel de disposer d'informations permettant de connaître l'ordre topologique (soit de manière immédiate, en O(1), soit de manière linéaire). Dans ces cas, cette implantation de l'algorithme des plus longs chemins révèle toute son efficacité.

Un tri topologique ordonne la liste des nœuds de manière à ce que tout nœud *i* apparaisse avant le nœud *j* dans l'ordre s'il existe un arc de *i* vers *j*. Par transitivité, on observe que s'il existe un chemin de *i* vers *j*, le nœud *i* est aussi avant le nœud *j* dans l'ordre topologique. Dans un graphe cyclique, il existe deux nœuds *i* et *j* tels qu'il y a un chemin de *i* vers *j* et un autre de *j* vers *i*. Dans un tel graphe, on

ne peut donc pas trouver d'ordre topologique. Ainsi, tout graphe dont on trouve un ordre topologique est un graphe acyclique. Inversement, il est toujours possible de trouver un ordre topologique pour un graphe acyclique.

Puisque le graphe conjonctif du jobshop est acyclique, on peut trier ses nœuds topologiquement. L'algorithme de plus longs chemins consiste alors à parcourir les nœuds dans l'ordre topologique et, pour chaque nœud, à parcourir tous les arcs issus de ce nœud pour faire les mises à jour des marques. Les marques ainsi calculées sont correctes car chaque mise à jour d'un arc est réalisée alors que la marque du nœud de départ est définitive. En effet, de par la définition d'un ordre topologique, on sait que la mise à jour d'aucun arc ne modifiera la marque d'un nœud positionné avant dans l'ordre topologique. Une implantation efficace de cet algorithme est présentée ci-dessous.

Ordre total	1 4 7 2 5 8 3 6 9

Tableau 2-20. Ordre de S1 de l'instance J1

En entrée, l'algorithme de plus long chemin nécessite un ordre total des opérations. Un exemple d'ordre total pour l'instance S1 est fourni dans le tableau 2-20. Cet ordre peut être obtenu par un tri topologique des nœuds du graphe conjonctif de la figure 2-16. L'algorithme propose en sortie les dates de début des opérations ($t(i)$) et le tableau *pere* permettant de coder un sous-arbre critique. Un élément de ce tableau *pere*(i) donne, si elle existe, l'opération précédent l'opération i dans le sous-arbre critique. L'utilisation du tableau *pere* induit donc que chaque nœud n'a qu'un unique prédécesseur dans le sous graphe critique. Autrement dit, si deux arcs critiques permettent d'arriver à un nœud, un seul de ces deux arcs sera codé par le tableau *pere*. Il est donc possible que certains arcs critiques ne soient pas identifiés comme tel.

Le chemin critique peut être obtenu en utilisant le tableau *pere* à partir de l'opération *. En effet, *pere*(*) est l'opération précédent l'opération * dans le chemin critique. Grâce au tableau *pere*, on peut remonter d'opération en opération le long du chemin critique. La dernière opération sur ce chemin est l'opération *0*. Celle-ci n'est pas codée explicitement dans notre algorithme, pour détecter la fin du chemin critique, on teste à chaque itération si on le tableau *pere* contient *–1*. En résumé, le tableau *pere* permet d'obtenir le chemin critique à rebours : *pere*(*), *pere*(*pere*(*)), ..., *0*.

L'initialisation de l'algorithme consiste en trois boucles successives. La première indique qu'aucune opération n'a encore été traitée sur aucune des machines. La seconde initialise les marques du graphe (tableau $t(i)$), et le sous graphe critique (tableau *pere*(i)). Cette boucle initialise aussi le tableau *suiv*(i) qui désigne l'opération traitée après l'opération i sur la même machine. La troisième boucle indique que les dernières opérations traitées sur chaque machine n'ont pas d'opération suivante. La boucle principale de l'algorithme est le calcul effectif des plus longs chemins. Les nœuds sont parcourus dans l'ordre des opérations T. Cet ordre étant un ordre topologique, les arcs sont parcourus de telle manière que chaque mise à jour est définitive. En effet, par définition de l'ordre topologique, on est sûr que ni l'itération i ni les itérations suivantes ne vont modifier les marques des opérations précédentes.

```
Entrée :
   T : ordre total de t opérations
Sortie :
   t(i) : Date de début de l'opération i
   pere(i) : Opération précédente dans le graphe critique
Temporaires :
   suiv(i) : Opération précédent i sur la même machine
   mach(i) : Dernière opération traitée sur la machine i
Début :              // Initialisation
Pour i de 1 à m faire
   mach(i)=-1;
```

```
FinPour
Pour i de 1 à t faire
   t(i)=0;
   pere(i)=-1;
   Si mach(m[i])≠-1 alors
         suiv(mach(m[i]))=i;
   Finsi
   mach(m[i])=i;
FinPour
Pour i de 1 à m faire      // Les dernières opérations n'ont
   suiv(mach(i))=-1;       // pas de suivant
FinPour
Pour i de 1 à t faire
   k = T(i);   // ième opération dans l'ordre T
   l = k+1;    // opération suivante du job
   Si k non derniere du job et t(k)+p(k) > t(l) alors
         t(l) = t(k)+p(k);
         pere(l) = k;
   Finsi
   l = suiv(i);      // Opération suivante sur machine
   Si l≠-1 et t(i)+p(i) > t(j) alors
         t(j) = t(i)+p(i);
         pere(j) = i;
   Finsi
FinPour
```

Algorithme 2-2. Algorithme de plus long chemin efficace pour le problème de jobshop

Cet algorithme est très efficace car il ne parcourt qu'une et une seule fois chacun des arcs du graphe. De plus, l'implantation à base de tableau est très efficace.

Pour utiliser cet algorithme, il faut donc disposer d'un ordre topologique. Cet ordre n'est pas très utilisé en général dans la littérature, mais beaucoup de représentations peuvent être transformées en un ordre total des opérations. En effet, pour la plupart des représentations, on a au moins l'ordre relatif des opérations sur les machines. À partir de cet ordre des opérations sur les machines et des contraintes de précédence dues aux gammes, il est aisé de calculer un ordre total généralisant les deux ordres partiels. C'est ce que réalise l'algorithme 2-3.

```
Entrée :
   T[j] : ordre des opérations sur la machine j
Sortie :
   TOPO : ordre topologique des opérations
Temporaires :
   nbPrec[i] : Nombre de prédecesseurs dans le graphe
   iInsere : dernier élément inséré dans TOPO
```

```
   iTopo : dernier élément traité dans l'ordre.
Début :
pour i de 1 à o faire
   nbPrec [i]  := 2;
finpour
pour i de 1 à n faire
   nbPrec [première opération du job i]  := 1;
finpour
iTopo  := 1;
iInsere  := 1;
pour i de 1 à m faire
   nbPrec [T[i][1]] := nbPrec [T[i][1]]-1;
   si nbprec [T[i][1]]== 0 alors
         TOPO[iInsere]  := T[i][1];
         iInsere  := iInsere + 1;
   finsi
   pour j de 1 à nombre d'opérations sur i faire
         pos[T[i][j]] :=j;
   finpour
finpour
Tant que iTopo<iInsere faire
   cour  := TOPO[iTopo];
   si pos[cour] <> nombre d'opérations sur la machine alors
         suiv  := T[machine de cour][pos[cour]+1];
         nbPrec[suiv] :=nbPrec[suiv]-1;
         si nbPrec[prec]=0 alors
               TOPO[iInsere] :=suiv;
               iInsere :=iInsere+1;
         finsi
   finsi
   si cour n'est pas la dernière opération de son job alors
         suiv  := cour+1;
         nbPrec[suiv] :=nbPrec[suiv]-1;
         si nbPrec[prec]=0 alors
               TOPO[iInsere] :=suiv;
               iInsere :=iInsere+1;
         finsi
   finsi

   iTopo  := iTopo + 1;
fintantque
```

Algorithme 2-3. Algorithme de plus long chemin efficace pour le problème de jobshop

L'algorithme consiste à parcourir simultanément les ordres d'opérations sur les machines et les contraintes de précédence dues aux gammes. Pour cela, il met à jour un tableau *nbPrec* qui indique le nombre de prédécesseurs de chaque nœud. À l'initialisation, tous les nœuds ont deux prédécesseurs. Puis, les premières opérations de chaque job et les premières opérations de chaque machine ont un prédécesseur en moins. Si en faisant ces décrémentations, un nœud n'a plus de prédécesseur, il est ajouté dans le tableau *TOPO* à la position *iInsere*. Ensuite, le tableau *pos* est calculé. Ce tableau permet de connaître la position d'une opération dans l'ordre des opérations sur les machines. Ce tableau sert dans la boucle principale pour pouvoir déterminer la prochaine opération sur la machine.

Ensuite, la boucle principale de l'algorithme commence. Le nœud courant (dénommé cour) est récupéré. Ce nœud est alors considéré comme trié et tous ses successeurs sont informés. Ensuite, les deux arcs pouvant sortir de ce nœud sont examinés. L'arc allant vers la prochaine opération sur la même machine est examiné en premier. Cet arc existe si l'opération n'est pas la dernière sur la machine. Si ce n'est pas le cas, l'opération suivante *suiv* est calculée. Cette opération est récupérée dans le tableau *T*, sur la même machine que *cour* mais à la position suivante. Le nombre de prédécesseur de cette opération est décrémenté. Puis, si ce nombre atteint zéro, l'opération est ajoutée au tableau *TOPO* à la position *iInsere*. De même, l'arc allant vers la prochaine opération du job est examiné.

Le tableau *TOPO* a donc trois usages simultanés : à la fin de l'algorithme il contient un ordre topologique complet, entre les positions *1* et *iTopo-1* se trouvent toutes les opérations déjà triées, entre les positions *iTopo* et *iInsere-1* (incluses) se trouvent toutes les opérations non triées mais qui n'ont plus de prédécesseurs non triés.

La boucle principale continue tant qu'il reste des nœuds non triés dans *TOPO*. Ainsi, la condition d'arrêt est *iTopo<iInsere*. Lorsque l'arrêt a lieu, deux raisons peuvent en être la cause. Soit tous les nœuds ont été triés dans *TOPO*, auquel cas on a réussi à trier le graphe topologique ou l'algorithme s'est arrêté prématurément et le graphe est cyclique.

4.4.6 Chemins critiques

Pendant l'évaluation du graphe conjonctif, certains arcs participent aux plus longs chemins et d'autres n'y participent pas. Connaître ces arcs permet de déterminer quelles opérations sont importantes pour l'ordonnancement. Les chemins critiques sont justement formés de ces arcs.

Un arc critique est un arc tel que tout retard des opérations au début et à la fin de l'arc retarde la fin du traitement. Ainsi, un arc critique vérifie la propriété suivante : la marque au début de l'arc augmenté de la longueur de l'arc est égale à la marque à la fin de l'arc. Dans l'ordonnancement, un arc critique de l'opération i vers l'opération j se transpose de la manière suivante : l'arc de i vers j est critique si l'opération j commence dès la fin de l'opération i. Ainsi, tout prolongement de l'opération au début de l'arc retarde l'opération suivante et tout retard de l'opération au début de l'arc retarde aussi l'opération suivante.

Le sous graphe critique est le sous graphe du graphe conjonctif formé par les arcs critiques. Un retard d'une opération dans le sous graphe critique retarde donc au moins une autre opération, le sous graphe critique contient donc l'ensemble des opérations dont l'exécution est critique.

La solution *S1* et son sous graphe critique sont affichés sur la figure 2-16. L'arc entre les opérations *8* et *3* est critique car la date de début de l'opération *8* augmenté de la longueur de l'arc entre *8* et *3* est exactement égal à la date de l'opération *3*. Tous les arcs en gras de la figure sont des arcs critiques.

Parmi les arcs du sous graphe critique, on s'intéresse particulièrement au chemin critique. Le chemin critique est le chemin du sous graphe critique allant de l'opération *0* à l'opération ***. Ce chemin est remarquable car tout retard d'une opération le long de ce chemin, ou tout allongement d'une opération le long de ce chemin retarde la date de fin de l'ordonnancement. Sur l'exemple de la figure 2-16, tous les arcs critiques sont en gras, mais les arcs du chemin critique sont en pointillés.

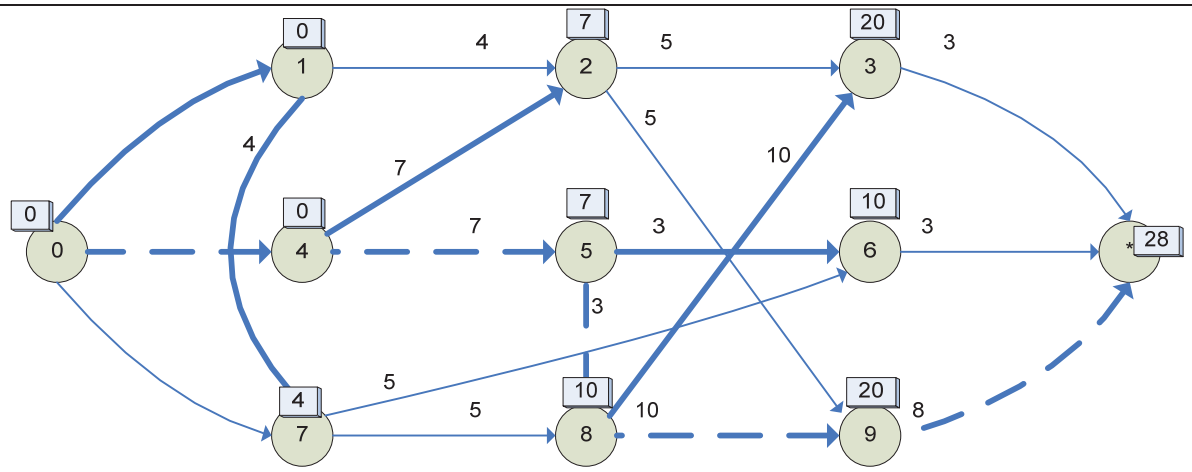

Figure 2-16. Graphe conjonctif évalué avec le sous graphe critique (en pointillés)

Le chemin critique est le chemin le plus long du graphe conjonctif, c'est lui qui définit le makespan de l'ordonnancement correspondant à ce graphe conjonctif. Pour diminuer le makespan d'une solution, il est donc nécessaire de modifier ce chemin. Toute modification de l'ordonnancement qui ne modifie pas le chemin critique ne peut pas améliorer la solution : elle ne peut que faire apparaître un nouveau chemin de longueur supérieure car le plus long chemin du graphe initial reste intact. Inversement, toute modification du chemin critique casse ce chemin, mais rien ne garantit que le nouvel ordonnancement soit égal meilleur ou pire. Les voisinages présentés ci-après exploitent avantageusement cette propriété.

4.5 Voisinages guidés

Job 1	O1= Machine 1, Durée : 1	O2= Machine 2, Durée : 3	O3= Machine 3, Durée : 6	O4= Machine 4, Durée : 7	O5= Machine 5, Durée : 3	O6= Machine 6, Durée : 6
Job 2	O7= Machine 2, Durée : 8	O8= Machine 1, Durée : 5	O9= Machine 3, Durée : 10	O10= Machine 4, Durée : 10	O11= Machine 5, Durée : 10	O12= Machine 6, Durée : 4
Job 3	O13= Machine 3, Durée : 5	O14= Machine 2, Durée : 4	O15= Machine 1, Durée : 8	O16= Machine 5, Durée : 9	O17= Machine 4, Durée : 1	O18= Machine 6, Durée : 7

Tableau 2-21. Instance J2

Machine 1	O1	O8	O15
Machine 2	O2	O7	O14
Machine 3	O13	O9	O3
Machine 4	O4	O10	O17
Machine 5	O16	O5	O11
Machine 6	O18	O6	O12

Tableau 2-22. Exemple de solution S2

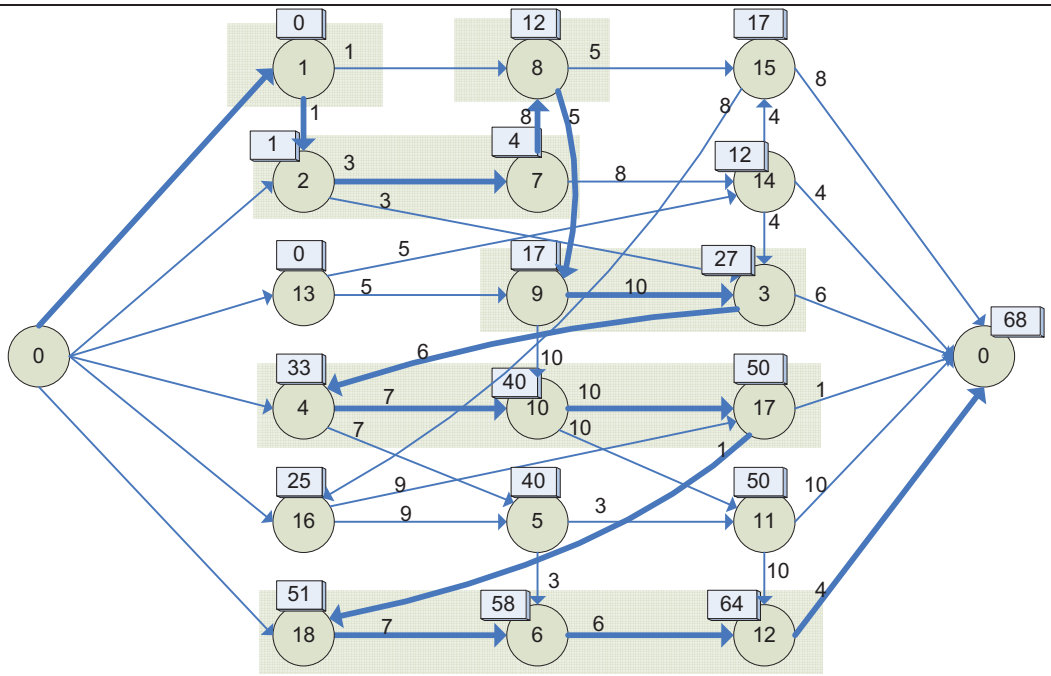

Figure 2-17. Chemin critique et voisinage de Laarhoven de S2

Le modèle du graphe conjonctif - disjonctif et la notion de chemin critique ont largement été utilisés dans la littérature pour concevoir des algorithmes performants. Ce paragraphe présente les principaux voisinages de la littérature et leurs propriétés respectives. Les voisinages sont classés dans un ordre de guidage croissant, c'est-à-dire que le premier voisinage est moins guidé que le dernier. Tous concernent le problème de jobshop.

Pour illustrer les voisinages, nous utilisons l'instance de jobshop appelée J2, dont les données sont fournies dans le tableau 2-21. Ce problème est constitué de 3 jobs et de 6 machines. On envisage la solution S1 de ce problème. Le graphe conjonctif de cette solution est présenté sur la figure 2-17. Dans ce graphe, les nœuds traités par la même machine sont tous sur la même ligne. Les arcs en gras définissent le chemin critique du puits jusqu'à la source. Les blocs gris horizontaux qui englobent les nœuds sont des blocs au sens du paragraphe 4.5.2.

4.5.1 Voisinage de Laarhoven

Le voisinage de Laarhoven *et al.* (1992) consiste à permuter deux opérations consécutives, traitées par la même machine et le long du chemin critique.

Sur l'exemple de la figure 2-17, le chemin critique est (1,2,7,8,9,3,4,10,17,18,6,12). Ce chemin peut être découpé en suites d'opérations réalisées sur la même machine, on obtient (1), (2,7), (8), (9,3), (4,10,17), (18,6,12). Pour construire le voisinage, il faut envisager toutes les permutations des opérations dans les sous suites, on obtient donc (2,7), (9,3), (4,10), (10,17), (4,17), (18,6), (18,12), (6,12).

Dans Laarhoven *et al.* (1992), les auteurs ont montré que ce voisinage est faiblement connecté. Ainsi, il est possible de trouver la solution optimale en appliquant itérativement ce voisinage.

Laarhoven *et al.* (1992) ont utilisé ce voisinage pour mettre en œuvre un recuit simulé. De même Lourenço (1995) propose une optimisation locale utilisant une descente stochastique et des grands sauts (large step optimization).

Il est possible que ce voisinage soit vide, c'est-à-dire qu'il n'existe pas de paires d'opérations réalisées sur la même machine. Dans ce cas, les arcs du chemin critique sont tous des arcs conjonctifs (arcs entre opérations consécutives du même job). Toutes les opérations du chemin critique sont donc du même job. Cela signifie que le job a commencé dès le début de l'ordonnancement et qu'il se termine à la fin de l'ordonnancement. Ce cas de figure n'est pas très courant, mais il correspond à une solution

optimale. Le fait que le voisinage soit vide n'est pas pénalisant dans ce cas, car quand la solution optimale est atteinte le processus d'optimisation peut s'arrêter.

4.5.2 Les blocs de Grabowski

Grabowski (1986) propose la notion de blocs pour structurer le chemin critique. On considère la suite d'opérations qui forme le chemin critique. Les blocs sont des sous suites maximales du chemin critique dont toutes les opérations sont traitées par la même machine.

Plus précisément, un chemin critique se note $U=\{u_1,u_2,...,u_w\}$ où $u_1=0$ et $u_w=*$, plusieurs chemins critiques existent à priori dans le même graphe mais nous n'en considérons qu'un à la fois. Le chemin critique U est découpé en r blocs : $U=\{B_1,B_2,...,B_r\}$ où chaque bloc B_i est composé d'opérations réalisées par la même machine. De plus, les blocs sont maximaux c'est-à-dire que deux blocs consécutifs sont traités par deux machines différentes.

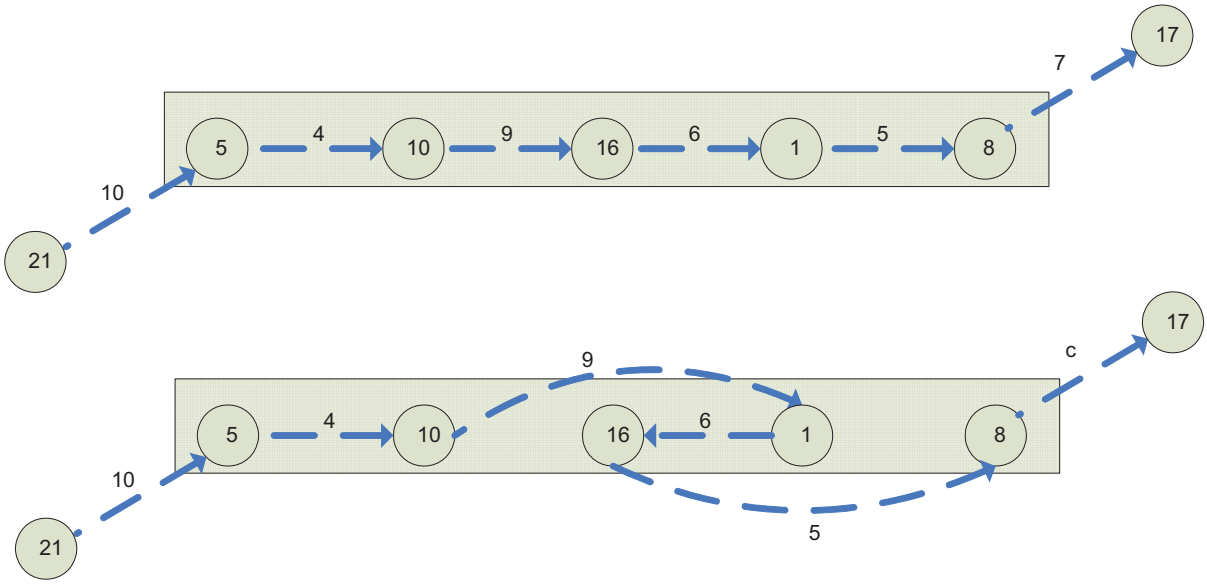

Figure 2-18. Modification à l'intérieur d'un bloc

L'intérêt de la notion de bloc réside dans le fait que toute modification à l'intérieur d'un bloc ne change pas sa longueur. La figure 2-18 illustre ceci en montrant un bloc puis le bloc modifié par la permutation de deux opérations consécutives. Les deux blocs ont la même longueur (4+9+6+5=24). En effet, la permutation des opérations 1 et 16 a modifié les arcs présents dans le bloc, mais les arcs insérés restent tous à l'intérieur du bloc et ont tous la même longueur que les arcs du bloc de départ. Ainsi, l'arc $10\rightarrow 16$ est remplacé par l'arc $10\rightarrow 1$ de même longueur, l'arc $16\rightarrow 1$ est remplacé par l'arc $1\rightarrow 6$ et l'arc $1\rightarrow 8$ est remplacé par l'arc $16\rightarrow 8$.

Les seules modifications qui peuvent améliorer le chemin critique ne sont pas à l'intérieur du bloc. Dans le problème de jobshop, il y a deux types d'arcs les arcs conjonctifs et disjonctifs (i.e. arcs issus d'arêtes disjonctives). Un arc entre deux blocs relie deux opérations traitées par des machines différentes. Un arc entre deux blocs est donc un arc conjonctif. Ces arcs sont présents quel que soit l'ordre des opérations, ils ne peuvent donc pas être supprimés pour améliorer l'ordonnancement. Finalement, les seules modifications qui peuvent améliorer l'ordonnancement doivent donc concerner les opérations sur les bords des blocs.

4.5.3 Voisinage de Dell'Amico

Dell'amico et Trubian (1993) ont proposé un voisinage qui consiste à déplacer une opération à l'intérieur du bloc vers le bord d'un bloc.

Les voisins ainsi obtenus sont illustrés sur la figure 2-19. Sur cette figure, le chemin critique, appelé *C*, est dessiné en arcs pointillés. Cette figure présente un bloc dont l'opération 16 est déplacée en début de bloc, ce qui modifie les arcs. L'ordre des opérations sur la machine passe de 5 10 16 1 8 à 16 5 10 1 8, les arcs sont donc modifiés en fonction. L'arc de 21 vers 5 est un arc conjonctif et il n'a pas à être modifié. Cette modification ne change pas la durée des opérations réalisées sur la machine, mais le chemin *C* a été modifié. En effet, le chemin *C* passait par l'arc $21 \rightarrow 5$ puis par le bloc, il se trouve raccourci de la longueur de l'opération 16 et entre directement à l'opération 5. Rien n'assure que le chemin *C* soit de nouveau un chemin critique après la modification de l'ordonnancement. De plus, rien n'assure non plus que cette modification va diminuer la valeur du makespan. Tout ce qui est sûr est que le chemin critique considéré n'existe plus dans la solution modifiée.

Ce voisinage a été utilisé par Dell'Amico et Trubian (1993) dans un algorithme tabou. Sun *et al.* (1995) ont proposé un algorithme tabou utilisant conjointement le voisinage de Dell Amico et celui de Laarhoven. Dell'Amico et Trubian (1993) ont montré que ce voisinage était lui aussi faiblement connecté (i.e. qu'il permet toujours d'obtenir l'optimum en un nombre fini d'étapes).

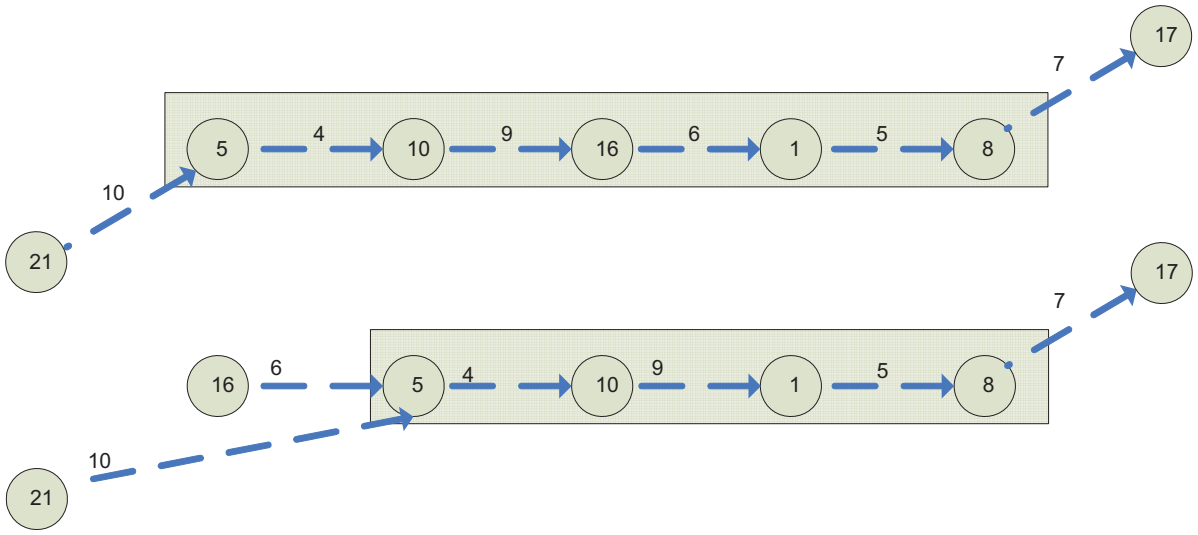

Figure 2-19. Voisinage de Dell Amico

4.5.4 Voisinage de Nowicki et Smutnicki

Nowicki et Smutnicki (1996) ont proposé un voisinage encore plus guidé qui consiste à permuter les deux premières opérations d'un bloc ou les deux dernières opérations d'un bloc, excepté pour le premier bloc dont on ne permute pas les deux premières opérations et pour le dernier bloc dont on ne permute pas les deux dernières opérations.

Outre la proposition d'un voisinage performant, Nowicki et Smutnicki ont prouvé que leur voisinage dominait le voisinage de Laarhoven. Cela signifie que tout voisin améliorant dans le voisinage de Laarhoven est dans le voisinage de Nowicki et Smutnicki. De plus, ils prouvent que toute solution dont le voisinage est vide (car cela peut arriver) est une solution optimale.

Nowicki et Smutnicki (1996) ont utilisé ce voisinage pour réaliser un algorithme tabou. Jusqu'à aujourd'hui leur algorithme est considéré comme étant un des algorithmes les plus performants pour le problème de jobshop.

De plus, Nowicki et Smutnicki (1996) ont montré que ce voisinage était lui aussi faiblement connecté (i.e. qu'il permet toujours d'obtenir l'optimum en un nombre fini d'étapes).

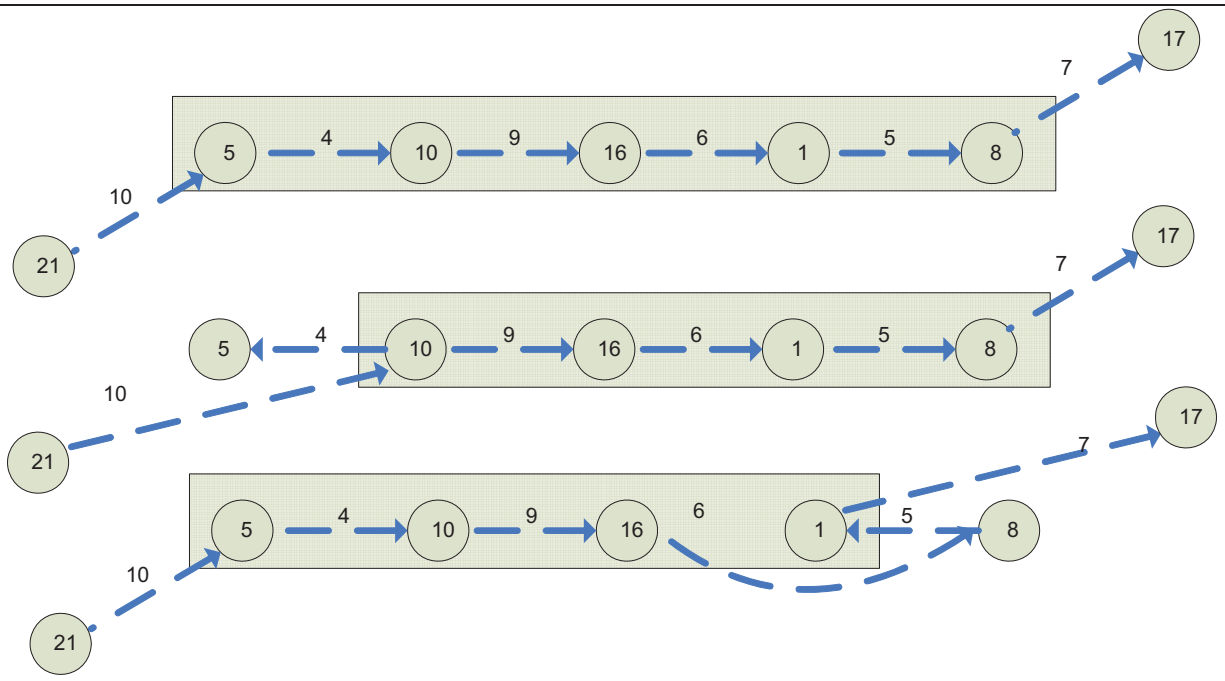

Figure 2-20. Voisinage de Nowicki et Smutnicki

4.5.5 Conclusion sur les voisinages

Les voisinages décrits ci-dessus sont à l'origine de méthodes de recherche locale et de méthodes arborescentes efficaces. Ils utilisent des propriétés des chemins critiques, typiques des problèmes disjonctifs. Dans les chapitres suivants, nous réutilisons ces voisinages et leur propriété pour des problèmes disjonctifs plus complexes : le problème de jobshop avec time lag (chapitre 3) et le problème de jobshop avec transport et contraintes additionnelles (chapitre 4).

5 Une des meilleures méthodes publiées : la méthode tabou Nowicki et Smutnicki

En 1999, Nowicki et Smutnicki (Nowicki et Smutnicki, 1996) ont proposé un algorithme tabou déterministe. Depuis, cet algorithme est considéré comme un des meilleurs algorithmes pour le problème de jobshop (Watson *et al.*, 2006). C'est pourquoi nous nous y intéressons en détail dans cette partie.

5.1 Présentation générale

L'algorithme tabou de Nowicki et Smutnicki (1996) est un algorithme tabou qui converge très rapidement vers des solutions de bonne qualité grâce à un voisinage très guidé. Pour éviter de rester bloqué dans une partie restreinte de l'espace, un mécanisme de retour arrière permet de repartir de la dernière solution améliorante pour envisager d'autres voisins. Grâce à ces deux mécanismes conjugués et grâce aux solutions de départ de bonne qualité, cet algorithme fournit de très bonnes solutions en un temps restreint.

L'algorithme tabou utilise la représentation semi-active (R2) (paragraphe 4.3.2). Ainsi, une solution est codée sous la forme d'un ordre d'opérations par machine. L'algorithme consiste donc à chercher l'ordre des opérations de plus faible makespan.

Comme tout algorithme tabou, cet algorithme se base sur la notion de voisinage pour générer de nouvelles solutions. Les auteurs utilisent un voisinage dédié, proposé par eux-mêmes, et basé sur la permutation de deux éléments. Ce voisinage est décrit dans le paragraphe 0. Ce voisinage est très guidé, car il comporte un très faible nombre de voisins de bonne qualité.

À partir d'une solution initiale, l'algorithme tabou fait évoluer une solution, appelée solution courante, afin de trouver la solution optimale. À chaque itération, la solution courante évolue. C'est le voisinage qui permet de définir un ensemble de solutions dont une, en particulier, est choisie (en général la meilleure solution est retenue). La solution courante se déplace vers ce voisin, et la prochaine itération calcule le voisinage de cette nouvelle solution.

D'itération en itération, il est possible que l'algorithme cycle, c'est-à-dire qu'il peut atteindre plus d'une fois la même solution et se mettre à répéter indéfiniment la même séquence. Pour éviter cela, et pour tenter de le guider vers les "bonnes régions", une mémoire des déplacements réalisés est conservée. L'algorithme de Nowicki et Smutnicki implante cette mémoire sous la forme d'une liste des $\max t$ dernières permutations. Ainsi, lorsque l'on explore la liste des voisins, toutes les permutations présentes dans la liste sont considérées comme taboues et ne sont pas envisagées. La seule exception à cette règle est due à la fonction d'aspiration qui supprime le statut tabou d'un voisin si celui-ci est meilleur que la meilleure solution connue.

Plus précisément, la stratégie appliquée pour chercher le meilleur voisin sépare les voisins en trois catégories : U l'ensemble des voisins autorisés, FP l'ensemble des voisins interdits mais profitables (voisins autorisés par la fonction d'aspiration) et FN les autres voisins non profitables. Cette stratégie est implantée dans la procédure nommée NSP.

De plus, pour permettre une meilleure exploration de l'espace de recherche, un mécanisme de retour arrière est implanté. Ce mécanisme consiste à explorer des régions voisines en repartant des points qui ont amélioré la solution courante. Cette stratégie repose sur l'intuition que c'est lors des améliorations que les décisions importantes sont prises et que d'autres portions de l'espace de recherche sont atteignables à ce moment. Plus précisément, lorsqu'une solution améliorante est trouvée, le reste du voisinage et les variables de l'algorithme sont sauvegardés. Lorsqu'un trop grand nombre d'itérations sans améliorations a lieu, un retour arrière est déclenché et l'algorithme revient à cette solution sauvegardée pour envisager un autre voisin que le voisin améliorant. Ainsi, de retour arrière en retour arrière, toutes les solutions dans le même voisinage que la solution améliorante sont visitées.

5.2 Algorithme détaillé

L'algorithme 2-4 présenté ci-dessous est l'algorithme de principe de l'algorithme tabou tel que nous l'avons implanté. Cet algorithme détaille l'algorithme de principe présenté dans (Nowicki et Smutnicki, 1996).

Dans l'algorithme, les ordres d'opérations π et π^* sont des ordres d'opérations par machine comme le décrit la représentation semi active. Un ordre π est donc formé d'autant d'ordres qu'il y a de machines. Chaque ordre indique sur chaque machine l'ordre dans lequel les opérations doivent être réalisée. L'ordre π^* initial est en fait construit par l'algorithme INSA.

L'ordre des opérations π est l'ordre courant des opérations. L'ordre π^* est l'ordre initial lors de l'initialisation de l'algorithme, mais c'est aussi l'ordre *record* lors des itérations suivantes. Les fonctions $C_{\max}(\pi)$ et $C_{\max}(\pi^*)$ permettent de calculer les critères de performance (en l'occurrence le makespan de la solution). Le détail d'implantation de cette fonction est fourni dans le paragraphe 4.4, lors de la description du graphe conjonctif-disjonctif.

Cet algorithme, proposé par Werner et Winkler, est une heuristique d'insertion qui construit itérativement un ordre d'opérations. À chaque itération, une opération est insérée dans la séquence à la meilleure position possible. Lorsqu'une opération est insérée, les arcs correspondant dans le graphe sont insérés et mis à jour par l'algorithme de plus long chemin.

La variable T désigne la liste taboue. D'un point de vue informatique, cette liste s'implante efficacement comme un buffer circulaire dans un tableau. Ce tableau est de taille $\max t$, i.e. le nombre d'éléments tabous, et est muni de deux indices t_{start} et t_{end}. L'indice t_{start} est initialisé à 0 et permet de déterminer la position du prochain élément tabou qui sera inséré dans la liste. À chaque nouvel élément, cet indice est incrémenté jusqu'à ce qu'il atteigne la valeur $\max t$ où il est alors immédiatement réinitialisé à 0 ($t_{start} \in [0; \max t - 1]$). De même pour l'indice t_{end}. Cet indice est incrémenté à chaque fois qu'un ancien élément tabou est supprimé de la liste.

La liste L, basée sur le même principe que la liste taboue, utilise un buffer circulaire pour contenir la liste des dernières améliorations. La liste L est donc aussi munie de deux indices l_{start} et l_{end}. Une amélioration a lieu lorsque la solution record est battue. La liste L contient donc la solution améliorée, la liste de ses voisins et la liste taboue. Ainsi, lors du prochain retour arrière, l'algorithme peut explorer les autres voisins que le premier retenu. Le fait d'avoir sauvegardé la liste taboue permet en outre de repartir dans les mêmes conditions que si l'algorithme n'avait pas choisi le voisin retenu mais l'avait rendu tabou.

Le booléen *save* est un indicateur positionné à vrai lorsqu'une amélioration a lieu pour être capable de sauvegarder l'état de l'algorithme (liste taboue, liste des voisins et solution courante) lors de la prochaine

La variable *iter* est un compteur du nombre total d'itérations réalisées. La variable *palier* compte le nombre d'itérations dans le palier courant, c'est-à-dire le nombre d'itérations sans améliorations.

C est la liste des valeurs de critères. Cette liste est un buffer circulaire contenant $(\_\max c + 1)\ \_\max \delta$ où $\_\max c$ et $\_\max \delta$ sont les deux paramètres de l'algorithme tabou.

cVoisins est un tableau surdimensionné des voisins de la solution courante (i.e. π).

nepascalculer est un indicateur permettant de ne pas recalculer la liste des voisins lors d'un retour arrière. En effet, lors d'une itération normale, on part d'une solution dont on calcule les voisins et dont on choisit le meilleur voisin. Alors qu'après un retour arrière, il n'est pas souhaitable de recalculer la liste des voisins car certains de ceux-ci ont déjà été explorés. Il faut donc repartir de la liste des voisins sauvegardée.

```
Soit π* une séquence de traitement (construite par l'algorithme INSA)
π = π*            // Solution courante=solution record=solution initiale
t_start=0;        // liste taboue
l_start=0;        // liste de backtrack
c_start=0;        // liste des valeurs de critère
iter=0;  // compteur global d'itération
palier=0;// compteur de palier
save=true;     // sauve la première position
nepascalculer= false;
tant que non fin faire
   iter=iter+1;
   si nepascalculer alors
         calcule les voisins de π dans cVoisins; // cf. voisinage Nowicki…
   finsi
   nepascalculer=false;
   pour y voisin de x faire
         Evaluer Cmax(y);        // utiliser le graphe conjonctif
```

```
finpour

si save et le nombre de voisins dans cVoisins> 1 alors
      Copie π et T dans la liste L à la position l_start ;
finsi
Applique l'algorithme NSP pour choisir π;
si save et le nombre de voisins dans cVoisins> 1 alors
      Supprime π de la liste des voisins cVoisins;
      ++l_start ;
      l_start = l_start modulo _max t ;
      si l_start==l_end alors
            l_end=l_end+1;
      finsi;
      l_end = l_end modulo _max t ;
finsi
Ajoute la permutation courante dans T à la postion t_start
t_start=(t_start+1) modulo _max t ;
si Cmax(π)<Cmax(π*) alors
      π*=π;
      palier = 0;
      save = true;
finsi
Ajoute le critère Cmax(π) dans la liste C à la position c_start ;
iter=iter+1;
palier=palier+1;

Détecte un cycle dans C
si un cycle est détecté alors
      palier = palier_max;
finsi
si palier ≥ palier_max alors
      si l_start<>l_end alors
            l_start=l_start-1;
            l_start = (l_start+_max l) modulo _max l ;
            récupère π, T et cVoisins dans la liste L à la position l_start
      sinon
            fin = true;
      finsi
      palier=0;
      palier_max=palier_max après retour arrière;
      save=true;
```

```
            nepascalculer=true;
      finsi
fintantque
```

Algorithme 2-4. TSAB - Algorithme Tabou

6 Conclusion

De nombreux outils et de nombreuses méthodes performantes permettent de résoudre de manière efficace le problème de jobshop. Dans ce chapitre, nous avons en particulier mis en évidence que le graphe conjonctif - disjonctif et les voisinages basés sur le chemin critique dans ce graphe sont des outils très efficaces. Relativement au théorème de no free lunch (cf. chapitre 1-7.4.1), on peut dire que ces outils permettent d'insérer de la connaissance du problème de jobshop dans les algorithmes proposés.

Comme nous l'avons proposé dans notre démarche de modélisation, nous allons réutiliser ces outils pour des problèmes plus complexes. Dans le chapitre 3, nous faisons des propositions pour le problème de jobshop avec time lags, ces propositions consistent à compléter le graphe conjonctif - disjonctif pour prendre en compte les contraintes de time lags mais aussi à adapter les représentations pour qu'elles prennent en compte la difficulté de résolution de ces problèmes.

Références bibliographiques

Aanen, E., Gaalman, G.J. et Nawijn, W.M. (1993). A scheduling approach for a flexible manufacturing system. International Journal of Production Research. 31(10), 2369-2385.

Baker, K.R. (1974). Introduction to sequencing and scheduling. Wiley & Sons, 1974.

Balas, E., Lenstra, J.K. et Vazacopoulos, A. (1995). The one-machine Problem with Delayed Precedence Constraints and its use in jobshop Scheduling. Management Science, 41(1), 94-109.

Bertel, S. (2001). Problèmes d'ordonnancement de type flowshop hybride avec recirculation et fenêtres de temps. Thèse de Doctorat. Université de Tours, décembre 2001.

Bierwirth, C. (1995). A generalized permutation approach to jobshop scheduling with genetic algorithms. OR spektrum, 17, 87-92.

Bilge, U. et Ulusoy G. (1995). A time window approach to simultaneous scheduling of machines and material handling system in an FMS". Operations Research, 43(6), 1058-1070.

Blanc X., (2005). MDA en action, Ingénierie logicielle guidée par les modèles. Edition Eyrolles, 2-212-11539-3.

Blazewicz, J., Ecker, K., Schmidt, G. et Weglarz J., (1993). Scheduling in Computer and Manufacturing Systems. Springer Verlag, Berlin, Heidelberg.

Blazewicz, J., Domschke, W. et Pesch, E., (1996). The job shop scheduling problem : Conventional and new solution techniques. European Journal of Operational Research, 93(1), 1-33.

Botta-Genoulaz, V. (2000). Hybrid flow shop scheduling with precedence constraints and time lags to minimize maximum lateness. International of Production Economics. 64, 101-111.

Bräsel, H. (1990). Lateinische Rechtecke und Maschinenbelegung, Dissertation B, TU Magdeburg.

Brauner, N. (1999). Ordonnancement dans des cellules robotisées. Thèse de doctorat. Université Joseph Fournier de Grenoble I, septembre 1999.

Brucker, P. et Heitmann, S. (2003). Flow-Shop Problems with Intermediate Buffers. 25, 549-574.

Brucker, P., Hilbig, T. et Hurink, J. (1999a). A branch and bound algorithm for a single-machine scheduling with positive and negative time-lags. Discrete Applied Mathematics, 94, 77-99.

Brucker, P. et Knust, S. (1999b). Complexity results for single-machine problems with positive finish-start time-lags. Computing, 63, 299-316.

Carlier, J. et Chrétienne, P. (1988). Problèmes d'ordonnancement : modélisation / complexité / algorithmes. Masson Edition.

Caumond, A., Gourgand, M., Lacomme, P. et Tchernev, N. (2004). Métaheuristiques pour le problème du jobshop avec time-lags. Actes de MOSIM'04. 1 au 3 septembre, Nantes.

Caumond, A., Gourgand, M., Lacomme, P. et Tchernev, N. (2005a). Proposition d'un algorithme génétique pour le job-shop avec time-lags. ROADEF' 05. Tours, France, 183-200.

Caumond A., Lacomme, P. et Tchernev, N. (2005b). Bi-objective optimization of the jobshop with time lags. 6ème conférence sur les métaheuristiques (MIC05). Vienne (Autriche) Aout 2005.

Caumond A., Lacomme P. et Tchernev N. (2005c). Feasible schedules generation with an extension of the Giffler and Thomson algorithm for the jobshop with timelags. International Conference on Industrial Engineering and System Management, Marrakech-Maroc.

Caumond A., Lacomme, P., Gourgand, M. et Tchernev, N. (2005d). Modélisation pour l'optimisation d'un atelier de forge: un problème de jobshop avec time lags. Journées JDMACS, Lyon (France).

Caumond A., Lacomme P., Moukrim A., Tchernev N. (2006a). An MILP for scheduling problems in an FMS with one vehicle. European Journal of Operational Research. A paraître.

Caumond A., Lacomme, P. et Tchernev, N. (2006b). A Memetic Algorithm for the jobshop with time lags. Computers and Operations Research. Soumis.

Coello Coello, C.A., Van Veldhuizen, D.A. et Lamont, G.B. (2002). Evolutionary algorithms for solving multi-objective problems. Kluwer, New York.

Cossard, N. (2004). Un environnement logiciel de modélisation et d'optimisation pour la planification de la production dans la chaîne logistique. Thèse de doctorat, Université de Clermont Ferrand II (France).

Deb, K. (2001). Multi-objective optimization using evolutionary algorithms. Wiley, Chichester, UK.

Dell'amico, M. (1996). Shop problems with two machines and time lags. Operations Research, 44(5), 777-787.

Dell'amico, M. et Trubian, M. (1993). Applying tabu search to the jobshop scheduling problem. Annals of Operations Research. 41, 231-252.

Deppner, F. (2003). Ordonnancement d'atelier avec contraintes d'écart minimal et maximal entre opérations. Actes de la conférence MOSIM'03, Toulouse (France). 430-436.

Deppner, F. (2004). Ordonnancement d'atelier avec contraintes temporelles entre opérations. Thèse de doctorat. LORIA Nancy.

Dorn, J. (1999). The DÉJÀ VU Scheduling Class Library. Dans Fayad, Schmidt, and Johnson (eds.) Implementing Application Frameworks, Wiley, 521-540

Dorndorf, U. et Pesch, E. (1993). Evolution based learning in a job shop scheduling environment. Computers and Operations Research. 22(1), 25-40.

Ehrgott, M. et Gandibleux, M. Multiobjective. (2002). "Combinatorial Optimization", Ehrgott M. et Gandibleux X. (éds), 52, International Series in Operations Research and Management Science, p. 369-444. Kluwer.

Fang, H.L., Ross, P. et Corne, D. (1993). A promising genetic algorithm approach to Jobshop scheduling, rescheduling and openshop scheduling problems. DAI Research paper no 623. Paru dans les actes de "Fifth International Conference on Genetic Algorithms", S. Forrest (ed.), San Mateo : Morgan Kaufmann, 1993, pages 375-382.

Fink, A. et Voß; S. (2002). HotFrame : A Heuristic Optimization Framework. Dans S. Voß, D.L. Woodruff (Eds.), Optimization Software Class Libraries, Kluwer, Boston, 81-154.

Finta, L. et Liu, Z. (1996). Single Machine Scheduling Subject to Precedence Delays. Discrete Applied Mathematics, 70(3), 247-266

Fisher, H. et Thompson, G.L. (1963). Probabilistic learning combinations of local job-shop scheduling rules. Dans J.F. Muth and G.L. Thompson (eds.), Industrial Scheduling, Prentice-Hall, Englewood Cliffs, NJ.

Fondrevelle, J., Oulamara, A. et Portmann M.C. (2006) Permutation flowshop scheduling problems with maximal and minimal time. Computers & Operations Research, 33(6), 1540-1556.

Gandibleux, X., Sevaux, M., Sörensen, K. et T'Kindt, V. (2004). Meta-heuristics for multi-objective optimisation. LNEMS, volume 535. Springer 2004. ISBN 3-540-20637-X.

Giffler, B. et Thompson, G.L. (1960). Algorithms for solving production scheduling problems. Operations Research, 8(4), 487-503.

Gourgand, M. (1984). Outils logiciels pour l'evaluation des performances des systèmes informatiques. Thèse d'état, Université Clermont Ferrand II (France).

Gourgand, M. et Tchernev, N. (1998). Un environnement de modélisation du processus logistique industriel. Paru dans les actes des deuxième rencontres internationales de "La recherche en logistique". Parutions par Nathalie Fabbe-Costes, Christine Roussat, 1998, pages 539-557.

Grangeon, N. (2001). Métaheuristiques et modèles d'évaluation pour le problème du flow-shop hybride hiérarchisée. Mémoire de thèse, Université de Clermont Ferrand II (France).

Grabowski, J., Nowicki, E. et Zdrzalka, S. (1986). A block approach for single machine scheduling with release dates and due dates. European Journal of Operations Research, 26, 278–285.

Hall, N.G. et Sriskandarajah, C. (1996). A survey of machine scheduling problems with blocking and no-wait in process. Operations Research, 44(3), 510-525.

Haro, C. (2002). Traitement des interblocages dans la conduite informatique des systèmes. Thèse de Doctorat de l'Université de Tours, décembre 2002.

Haupt, R. (1989). A survey of Priority Rule-Based Scheduling, OR-spektrum, 11, 3-16.

Hurink, J. et Keuchel, J. (2001). Local search algorithms for a single machine scheduling problem with positive and negative time-lags. Discrete Applied Mathematics (112), 179-197.

Hurink J., Knust S. (2002). A tabu search algorithm for scheduling a single robot in a job-shop environment. European Journal of Operational Research, 119(1-2), 181-203.

Hurink J., Knust S. (2005). Tabu search algorithms for job-shop problems with a single transport robot, European Journal of Operational Research, 162(1), p. 99-111.

Jain, A.S., Meeran, S. (1999). Deterministic job-shop scheduling : Past, present and future. European Journal of Operations Research, 113(2), 390-434.

Jensen, M.T. (2001). Robust and flexible scheduling with evolutionary computation. Thèse de doctorat. Université de Aarhus (Danemark). Octobre, 2001.

Keijzer, M., Merelo, J.J., Romero G. et Schoenauer, M. (2001). Evolving Objects: a general purpose evolutionary computation library. Dans les actes de EA'01, 29-31 octobre 2001, Bourgogne, France.

Kim, C.W., Tanchoco, J.M.A. and Koo, P.H. (1997). Deadlock Prevention in Manufacturing Systems with AGV System: Banker's algorithm Approach. Journal of Manufacturing Science and Engineering, 119, 849-854.

van Laarhoven P.J.M., Aarts E.H.L et Lenstra J.K. (1992). Job-shop scheduling by simulated annealing. Operations Research. 40(1), 113-125.

Laburthe, F. Caseau, Y. (1997). SaLSA : A Specification Language for Search Algorithms LIENS rapport 97-11 de l'Ecole Normale Supérieure.

Lacomme, P., Moukrim, A. et Tchernev, N. (2005). Simultaneously Job Input Sequencing and Vehicle Dispatching in a Single Vehicle AVGS : a Heuristic Branch and Bound Approach Coupled with a Discrete Events Simulation Model. International Journal of Production Research, 43(9), 1911-1942.

Lacomme, P., Prins, C. et Sevaux, M. (2006). A genetic algorithm for a bi-objective capacitated arc routing problem. Computers & Operations Research. 33(12), 3473-3493.

Ladhari, T. et Haouari, M. (2005). A computational study of the permutation flow shop problem based on tight lower bound. Computers and Operations Research. 32(7), 1831-1847.

van Laarhoven, P.J.M., Aarts, E.H.L. et Lenstra, J.K. (1992). Jobshop scheduling by simulated annealing. Operations Research, 40, 113-125.

Lourenço, H.R. (1995), Jobshop scheduling : Computational study of local search and large-step optimization methods. 83, 347-364.

MacCarthy, B.L. et Liu, J. (1993). A New Classification Scheme For Flexible Manufacturing Systems. International Journal of Production Research, 31(2), 299-309.

Manne, A.S. (1960). On the job-shop scheduling problem, Operations Research, 8, 219-223

Mangione, F. (2003). Ordonnancement des ateliers de traitement de surface pour une production cyclique et mono-produit. Thèse de doctorat. Université de Grenoble I.

Mascis, A., Pacciarelli, D. (2002). Jobshop scheduling with blocking and no-wait constraints. European Journal of Operational Research, 143, 498-517.

Mattfeld, D.C. (1995). Evolutionary search and the job shop. Thèse de doctorat. Université de Bremen (Allemagne).

Mélèse, J. (1990). Approche systémique des organisations - Vers l'entreprise à complexité humaine. (Suresnes Edition Hommes et Techniques).

Michel, L. et van Hentenrick, P. (1997). Localizer : a modeling language for local search. 3ème conférence sur la principes et la pratique de la programmation par contraintes. Schloss Hagenberg, Autriche.

Minsky, M.L. (1968). Matter, mind and models. MIT press.

Mitten, L.G., (1958). Sequencing n jobs on two machines with arbitrary time-lags. Management Science (5) 3.

Munier, A., Queryanne, M. et Schulz, A.S. (1998). Approximation Bounds for a General Class of Precedence Constrained Parallel Machine Scheduling Problems. In Proceedings of Integer Programming and Combinatorial Optimization (IPCO), 1412, 367-382.

Nakano, R. et Yamada, Y. (1991). Conventional genetic algorithm for job shop problems. 4ème congrès ICGA, 474-479.

Nakano, R. et Yamada, T. (1992). A genetic algorithm applicable for largescale jobshop problems. Elsevier Science Publishers. Dans R. Manner and B. Manderick. Actes de la deuxième conférence "Parallel Problem Solving in Nature". Elsevier Science Publishers, Amsterdam, 281-290.

Neveu, B. et Trombettoni, G. (2004). Hybridation de GWW avec de la recherche locale. Journal électronique d'intelligence artificielle 3-30. http://jedai.afia-france.org/ detail.php? PaperID = 30.

Nowicki, E. (1999). The permutation flow shop with buffers : A tabu search approach. European Journal of Operationnal Research. 116, 205-216.

Nowicki, E. et Smutnicki, C. (1996). A fast taboo search algorithm for the jobshop problem. Management Science 42(6), 797-813.

Pinson, E. (1997). The job shop scheduling problem : A concise survey and some recent developments, dans Chrétienne, P., Coffman, E.G., Lenstra, J.K., Liu, Z. (Eds), Scheduling Theory and Its Applications, Wiley, New York, 277-295.

Pirard, F. (2005). Une démarche hybride d'aide à la décision pour la reconfiguration et la planification stratégique des réseaux logistiques des enterprises multi-sites. Thèse de doctorat. Université de Mons (Belgique).

Popper, J. (1973). La dynamique des systèmes, principes et applications. Editions d'organisation, Paris.

Rayward-Smith, V.J. et Rebaine, D. (1992). Open-shop scheduling with delays. Theoretical Informatics Applications. 439-448.

Reeves, C.R. (1993). Modern heuristic techniques for combinatorial problems. J. Wiley and Sons, New York, 320.

Rebaine, D. et Strusevich V.A. (1999). Two-machine open shop scheduling with special transportation times. Journal of the Operational Research Society. 50, 756-764.

Riezebos, J. et Gaalman, G.J.C. (1998). Time lag size in multiple operations flowshop scheduling heuristics. European Journal of Operations Research (105), 72-90.

Roy, B. et Sussman, B. (1964). Les problèmes d'ordonnancement avec contraintes disjonctives. Note DS No. 9 bis, SEMA Paris.

Sarramia, D. (2002). ASCI-mi: une méthodologie de modélisation multiple et incrémentielle. Application aux systèmes de trafic urbain. Thèse de doctorat. Université de Clermont-Ferrand II (France).

Schuster, C.J. et Framinan, J.M. (2003). Approximative procedures for no-wait job shop scheduling. Operations Research Letters, 31, 308-318.

Smutnicki, C. (1998). A two-machine permutation flow shop scheduling problem with buffers. OR Spektrum. 20(4), 229-235.

Soukhal, A. (2001). Ordonnancement simultané des moyens de transformation et de transport. Thèse de Doctorat de l'Université de Tours, décembre 2001.

Strusevich, V.A. (1999). A heuristic for the two machine open shop scheduling problem with transportation times. Discrete Applied Mathematics. (93), 287-304.

Su, L.H. (2003). A hybrid two-stage flowshop with limited waiting time constraints. Computers and Industrial Engineering. 44, 409-424.

Sun, D., Batta, R., et Lin, L. (1995). Effective job shop scheduling through active chain manipulation. Computers and Operations Research. 22, 159-172.

Szwarc, W. (1983). Flow shop problems with time lags. Management Science. 29(4), 477-481.

Tagushi, G. (1987). System of Experimental Design. Traduction anglaise de Unipub Kraus International Publication.

Tercinet, F. (2004). Méthodes arborescentes pour la résolution des problèmes d'ordonnancement, conception d'un outil d'aide au développement. Thèse de doctorat. Université de Tours, juin 2004.

Tchernev, N. (1997). Modélisation du processus logistique dans les systèmes flexibles de production. Thèse de doctorat. Université de Clermont Ferrand II (France).

Wagner, S. et Affenzeller, M. (2004) HeuristicLab Grid - A Flexible and Extensible Environment for Parallel Heuristic Optimization. Actes de la conference ICSS'04, 7-10 Septembre 2004, Pologne.

Watson, J.F., Howe, A.E. et Whitley, L.D. (2006). Deconstructing Nowicki and Smutnicki's i-TSAB tabu algorithm for the jobshop scheduling problem. Computers & Operations Research. 33, 2623-2644.

Werner, F., Winkler, A. (1995). Insertion techniques for the heuristic solution of the job shop problem. Discrete Applied Mathematics. 58, 191-211.

Wikum, E.D., Llewellyn, C. et Nemhauser, G.L. (1994). One machine generalized precedence constrained scheduling problems. Operations Research Letters, 16, 87-99.

Wolpert, D.H. et Macready W.G. (1995). No Free Lunch Theorems for Search, Rapport Technique SFI-TR-95-02-010. Sante Fe, NM, USA : Santa Fe Institute.

Yang, D.L. et Chern, M.S. (2003). A two machines flowshop sequencing problem with limited waiting time constraints. Computers and Industrial Engineering Conference. 28(1), 63-70.