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

Remerciements

Je remercie Philippe Lacomme et Nikolay Tchernev, pour leur encadrement que ce soit avant la thèse pour la recherche de financement et la définition du sujet de thèse, mais aussi pendant et après ma thèse. Je les remercie aussi pour leur gande disponibilité ainsi que l'esprit d'équipe dont ils ont fait preuve, cet esprit restera un exemple pour la suite de ma carrière.

Je remercie Alain Quilliot, directeur de l'ISIMA, de m'avoir accueilli durant mes études d'ingénieur, et surtout durant mon travail de thèse en tant que directeur du Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes (LIMOS) de l'Université Blaise Pascal de Clermont-Ferrand.

Je remercie les membres du jury pour leur relecture attentive. En particulier, j'adresse mes remerciements à Jean Charles Billaut, Professeur au Laboratoire d'Informatique de l'Université de Tours, pour sa lecture minutieuse du manuscrit. De même, je remercie Marino Widmer, Professeur à l'Université de Fribourg (Suisse), pour sa relecture en particulier sur les systèmes de transport. Tous deux ont permis d'améliorer grandement la qualité de ce manuscrit.

Je remercie mes amis et collègues de travail : michaël, nicolas, philippe, sylvain, sylverin pour les nombreuses discussions que nous avons eues ensemble. Tous m'ont été d'une grande aide, tant scientifique que professionnelle et personnelle.

Je remercie mes collègues enseignants avec qui j'ai eu grand plaisir à travailler. En particulier, je remerice l'équipe EPOC avec qui j'ai partagé mes années de monitorat et d'ATER, et dont j'ai apprécié le professionalisme.

Je remercie ma femme pour son soutien infini, la patience dont elle a fait preuve durant toute la durée de cette thèse. Sa présence et son aide ont été un support moral inestimable. Sans elle, tout ce travail perdrait tout son sens.

Introduction générale

Les systèmes de production de biens ou de services sont largement étudiés car on les rencontre :

- dans l'industrie : les usines, les ateliers et les îlots sont des systèmes de production.
- dans les systèmes informatiques : un processeur, une architecture multiprocesseur, une grille de calcul sont aussi des systèmes de production.
- dans le secteur des services : un guichet, une agence ou un hôpital sont des systèmes de production.

La gestion de ces systèmes pose de nombreux problèmes dans des domaines variés : gestion de production, marketing, gestion des ressources humaines, environnement, brevets, ... Pour résoudre ces problèmes, des techniques issues de disciplines très différentes sont employées. Dans cette thèse, nous nous intéressons particulièrement aux problèmes d'optimisation des systèmes de production. On appelle "problème d'optimisation" tout problème réel que l'on peut traiter à l'aide des techniques d'optimisation. Ces problèmes apparaissent dans les différents domaines cités ci-dessus, mais ils ont historiquement le plus d'applications en planification. Plus particulièrement, on rencontre des problèmes d'optimisation dans les domaines suivants : l'ordonnancement (planification opérationnelle), l'optimisation de la charge du système (planification tactique), l'optimisation de la capacité (planification stratégique) ou l'optimisation d'une politique de gestion...

L'étude des systèmes de production est complexe à cause du grand nombre d'entités qu'ils contiennent et de leurs interactions. En effet, comme l'indique le principe d'émergence, l'étude des parties d'un système ne permet pas, en général, de comprendre le système entier : "le tout est plus que la somme des parties". Pour gérer cette complexité, la modélisation aide au choix des entités pertinentes et donc à la négligence des entités non pertinentes. Le résultat de ces choix est appelé "modèle" et contient donc la description d'une partie "pertinente" du système. Parce que la construction d'un modèle est une activité difficile, on tente de diminuer cette complexité en séparant explicitement les phases de spécification et d'exploitation. On obtient donc le processus appelé "processus de modélisation" présenté dans la figure 0-1 (Gourgand, 1984). Ce processus détaillé dans (Tchernev, 1997) se compose de quatre phases. La première est la phase de spécification et les trois suivantes forment la phase d'exploitation.

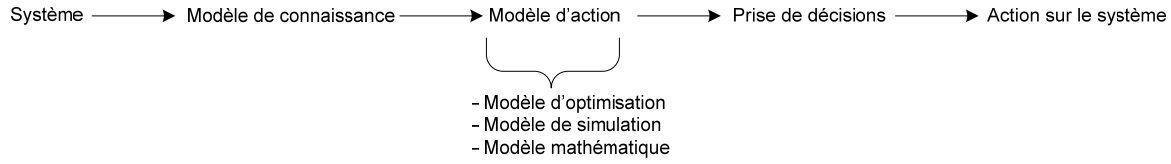

Figure 0-1. Processus de modélisation

Dans ce processus, la connaissance que l'on a du système est rassemblée dans un modèle appelé "modèle de connaissance". Ce modèle contient la description des entités et de leurs interactions, mais ne contient aucun élément de la solution mise en œuvre. Tant que le système n'est pas modifié, le modèle de connaissance doit donc rester valide. L'obtention de ce premier modèle marque la fin de la phase de spécification. La phase d'exploitation du modèle peut alors commencer avec l'élaboration d'un ou de plusieurs modèles d'action. Le modèle d'action est le modèle mis en œuvre pour avoir une action sur le système. Suivant le type de modèle d'action, on parle de "modèle d'optimisation" si le modèle fait appel aux techniques d'optimisation (heuristiques, métaheuristiques, méthodes arborescentes, ...), de "modèle de simulation" s'il fait appel aux techniques de simulation (simulation à événements discrets, simulation continue, simulation par éléments finis, ...) et de "modèle mathématique" s'il met en œuvre des techniques mathématiques (programmation linéaire, en nombres entiers, quadratique, ...). Le modèle d'action produit des résultats dont l'exploitation aide les experts à prendre une décision. Par cet aspect, les problèmes d'optimisation des systèmes de production sont fortement liés aux problèmes d'aide à la décision. Une décision étant prise, une action peut être initiée sur le système.

Les difficultés des problèmes d'optimisation des systèmes de production sont multiples. On peut les séparer en deux catégories distinctes : les difficultés de la phase de modélisation et les difficultés de la phase d'optimisation. Les deux dernières phases "prise de décision" et "action sur le système" dépendent largement du système et ne sont pas discutées dans cette thèse.

Lors de la phase de modélisation, on doit construire une abstraction d'un système en ne retenant que certaines entités et certains phénomènes de ce système. Les "hypothèses simplificatrices" sont formées par les entités et les phénomènes qui ne sont pas modélisés. Elles sont choisies en fonction des objectifs de la modélisation. Une autre modélisation, avec des objectifs différents pourra choisir des hypothèses simplificatrices complètement différentes. Le bon choix des hypothèses simplificatrices conditionne la suite du travail, or il n'existe aucune méthode ni aucun outil qui permettent de déterminer si une hypothèse simplificatrice est nécessaire ou bien choisie. En résumé, la détermination des hypothèses simplificatrices constitue la principale difficulté rencontrée lors de la phase de modélisation.

Les difficultés rencontrées lors de la phase d'optimisation résident dans la nature des problèmes étudiés. La plupart du temps, il est difficile de mettre en œuvre des algorithmes efficaces. On peut séparer les problèmes en deux catégories : les problèmes qui peuvent être résolus par des algorithmes efficaces et ceux qu'on ne sait pas résoudre de manière efficace. En général, pour les problèmes qui peuvent être résolus par des algorithmes efficaces, les algorithmes construits nécessitent une expertise importante et sont très dépendants des problèmes envisagés. Pour les problèmes qui ne peuvent être résolus de manière efficace, il est courant que l'on ne sache pas garantir les performances des approximations proposées. Ceci fait que les algorithmes d'approximation et particulièrement les métaheuristiques sont largement utilisés pour la résolution des problèmes réels. Ces approximations fournissent des résultats de bonne qualité en un temps relativement court. Par contre, si l'on utilise telles quelles les métaheuristiques de la littérature sur les problèmes classiques (flowshop, jobshop, RCPSP, TSP, …), la qualité des résultats obtenus est largement inférieure aux algorithmes spécialement mis en œuvre pour des problèmes particuliers. Ainsi, les approximations performantes utilisent de la connaissance spécifique au problème dans le but d'améliorer le processus d'optimisation. La dernière difficulté réside dans la grande diversité des méthodes d'optimisation proposées par la littérature. On ne trouve aucun classement entre ces différentes méthodes car il n'est pas possible d'en établir un comme le montre certains développements théoriques. En résumé, les difficultés de la phase d'optimisation sont que les algorithmes approchés n'ont pas de performances garanties en général, que ces algorithmes doivent inclure des connaissances spécifiques pour être performants et qu'il n'est pas possible de savoir à priori quel algorithme fonctionne le mieux.

Une difficulté supplémentaire réside dans l'utilisation conjointe de la modélisation et de l'optimisation. Lors de la phase de modélisation, on préfère utiliser un niveau de finesse suffisamment élevé alors qu'en optimisation, on exploite un modèle relativement simple. En effet, pendant la phase de modélisation, les experts doivent déterminer les hypothèses simplificatrices à réaliser. Or, si on veut évaluer les performances d'un système, on est d'autant plus précis que le modèle est fin. Bien sûr, il n'est pas souhaitable que le modèle soit le plus fin possible mais d'un degré de finesse suffisamment élevé. D'autre part, il n'est pas souhaitable de considérer des modèles trop complexes lors de l'optimisation. Un modèle très complexe se prête généralement mal à l'optimisation car l'espace de recherche n'est pas adapté et parce que la connaissance spécifique au problème est plus difficile à exploiter. Or, l'optimisation des systèmes de production doit à la fois optimiser et évaluer les configurations envisagées. Il est donc nécessaire de trouver le degré de finesse le plus juste entre ces deux extrêmes.

Pour résoudre ces difficultés et tirer profit des propositions de la littérature, nous proposons la démarche suivante. Pour un problème à étudier, nous cherchons quel modèle théorique sous jacent est le plus pertinent. Un modèle théorique est un modèle connu dans la littérature en optimisation. En général, ces modèles sont plus restrictifs que les problèmes réels. Ils sont généralement décrits par un relativement faible nombre de contraintes et nécessitent donc pour leur utilisation un grand nombre d'hypothèses simplificatrices. Il n'est donc pas rare que la phase de modélisation mette en évidence que les modèles théoriques ne sont pas adaptés aux objectifs du problème. Dans ce cas, nous proposons de

partir de ce modèle théorique et de l'enrichir progressivement afin de prendre en compte des contraintes supplémentaires. En procédant ainsi, nous pouvons récupérer les bonnes pratiques et les outils proposés dans la littérature pour des problèmes théoriques.

Dans cette thèse, nous avons choisi de mettre en œuvre cette démarche sur deux problèmes : le problème d'ordonnancement de l'atelier de forge de l'entreprise Aubert & Duval et le problème d'ordonnancement des systèmes flexibles de production. Nous avons retenu le problème de jobshop comme modèle théorique sous-jacent à ces deux problèmes. Ainsi, nos deux problèmes sont modélisés par deux extensions du problème de jobshop : le jobshop avec time lags et le jobshop avec transport et contraintes additionnelles. Pour résoudre ces deux problèmes, nous utilisons notre démarche afin d'adapter les outils proposés dans la littérature du jobshop aux problèmes plus complexes. En particulier, nous nous focalisons sur le graphe conjonctif-disjonctif. En effet, ce graphe est utilisé pour calculer le meilleur ordonnancement (dans le cas des critères réguliers), mais il est aussi utilisé pour mettre en évidence des propriétés intéressantes des ordonnancements qui permettent de perfectionner les algorithmes d'optimisation. L'utilisation de ce graphe est synthétisée dans la figure 0-2. Le graphe disjonctif modélise le problème en son ensemble, puis un ordre des opérations (qui modélise une solution) est utilisé afin de transformer le graphe disjonctif en un graphe conjonctif. Lorsque l'on dispose de ce graphe, l'exécution d'un algorithme de plus long chemin permet d'obtenir la meilleure solution relativement à l'ordre choisi. L'intérêt de la modélisation sous la forme d'un graphe est multiple : la meilleure solution relativement à l'ordre choisi est trouvée, des algorithmes efficaces et éprouvés existent pour la détermination des plus longs chemins et la solution ainsi obtenue peut être analysée (à l'aide du sous graphe critique). Dans la suite, nous utilisons ce schéma pour proposer des heuristiques et des métaheuristiques pour les problèmes de jobshop avec time lags et jobshop avec transport et contraintes additionnelles.

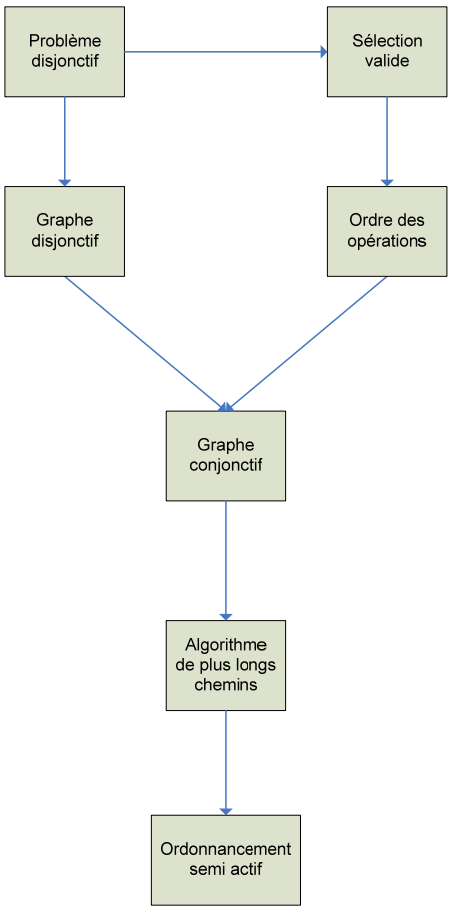

Figure 0-2. Graphe conjonctif -disjonctif

Cette thèse est structurée en cinq chapitres.

Le premier chapitre est une présentation de la démarche que nous avons utilisée. Il présente à la fois la classe de problèmes qui nous intéresse et les modèles théoriques associés. Nous présentons alors en détail notre démarche et ses conséquences à la fois dans le processus de modélisation et dans la phase d'optimisation. Les principales méthodes d'optimisation sont aussi présentées.

Le second chapitre traite du problème de jobshop en tant que problème théorique de base. Pour ce problème, nous présentons donc un état de l'art ainsi que les principaux algorithmes et outils utilisés dans la littérature. En particulier, nous présentons une modélisation linéaire en nombres entiers et une modélisation sous la forme d'un graphe conjonctif-disjonctif. Cette présentation servira de référence pour les développements des autres problèmes.

Dans le troisième chapitre, nous appliquons notre démarche au problème de jobshop avec time lags. Nous proposons donc : une modélisation mathématique, un programme linéaire en nombres entiers, et une adaptation de la modélisation sous forme de graphe conjonctif-disjonctif. A l'aide de ces outils, nous proposons des algorithmes d'optimisation variés allant des méthodes de construction jusqu'aux méthodes itératives en passant par la résolution exacte par solveur linéaire.

Le chapitre quatre concerne le problème de jobshop avec transport. Nos propositions pour ce problème rejoignent celles faites pour le jobshop avec time lags : nous proposons une modélisation mathématique, un programme linéaire en nombres entiers, une adaptation de la modélisation sous forme de graphe conjonctif-disjonctif et une méthode itérative basée sur ce graphe.

Dans le cinquième et dernier chapitre, nous proposons un cadriciel (la BCOO) permettant d'aider à la mise en œuvre des algorithmes proposés. En effet, les différents algorithmes proposés ont de nombreuses parties en commun et la différence d'un algorithme à l'autre est relativement limitée. Grâce à la BCOO (Bibliothèque et Cadriciel orienté Objet pour l'Optimisation), on ne développe que le juste nécessaire et toutes les "briques logicielles" s'assemblent automatiquement pour former l'application d'optimisation.

Chapitre 1 Présentation de la problématique

Ce chapitre présente la problématique de cette thèse : l'optimisation pour la planification des systèmes de production.

Sommaire

Table des figures

Table des tableaux

Table des algorithmes

1 Introduction

Ce chapitre est consacré à la présentation de la problématique : l'optimisation pour la planification des systèmes de production. La complexité de cette problématique réside dans l'utilisation conjointe de domaines complémentaires : la connaissance des systèmes de production, la modélisation, l'optimisation et les interfaces homme/machine (IHM).

Une bonne connaissance des systèmes de production est nécessaire car elle permet de faciliter les discussions avec les experts du système et de nous guider dans la définition des problèmes et enjeux. Premièrement, les experts du système sont des interlocuteurs privilégiés dans les phases d'analyse, de spécifications, de validation et de construction de l'interface homme/machine. Ce sont eux qui connaissent le besoin et qui ont l'expérience du système existant et de son environnement. C'est pourquoi la connaissance des systèmes de production raccourcit les discussions et évite les problèmes de communication. Deuxièmement, une bonne connaissance des systèmes de production permet de cibler directement les bonnes questions et de filtrer les informations pertinentes. Ceci est particulièrement utile pendant les phases d'analyse et de spécification, car celles-ci sont l'occasion de soulever de nouveaux problèmes et d'exprimer les intérêts contradictoires entre les différents interlocuteurs.

La modélisation est nécessaire pour appréhender la complexité des systèmes de production. Elle permet de les caractériser selon différents critères pour satisfaire un nombre maximal d'experts industriels. Le but est de recueillir l'information jugée pertinente, de la structurer, de l'organiser et d'identifier les règles de fonctionnement. La difficulté de cette phase réside principalement dans la taille, l'environnement, et les interactions complexes entre les entités des systèmes de production. Pourtant, la réussite d'un projet est étroitement liée à la qualité des modèles construits. Un modèle trop grossier ne contient pas toutes les contraintes nécessaires, et le résultat final peut proposer des solutions irréalisables. Inversement, un modèle trop détaillé peut être impossible à alimenter en données, ou trop complexe pour être résolu de manière efficace.

La plupart des problèmes de planification sont, ou se ramènent à, des problèmes d'optimisation. C'est pourquoi on s'intéresse aux techniques d'optimisation pour les problèmes de planification. L'optimisation est utilisée pour obtenir des performances accrues ou pour rechercher les performances optimales. L'optimisation est également utilisée pour trouver une solution de bonne qualité (pas forcément optimale) dont on sait qu'elle peut être modifiée. Ceci apparaît par exemple quand la charge du système évolue avant que la solution ne soit complètement mise en œuvre. La solution peut aussi être modifiée à cause de contraintes non prises en compte dans le modèle. Suivant les cas, on s'intéresse donc à différents types de méthodes d'optimisation.

Les interfaces homme/machine sont aussi nécessaires pour mener une étude de planification d'un système de production. Dans la plupart des cas, les solutions proposées par les algorithmes d'optimisation ne sont que des propositions faites à un expert en planification. De nombreuses raisons font qu'une solution n'est pas mise en œuvre directement : toutes les contraintes du système ne sont pas prises en compte, le système évolue pendant la mise en œuvre de la solution, l'expert en planification dispose d'informations complémentaires qui ne sont pas dans le système d'informations... C'est pourquoi il est important de permettre à l'utilisateur final de modifier le planning proposé et de mettre en œuvre des interfaces permettant de visualiser une solution sous diverses formes et organiser un dialogue entre l'utilisateur et le moteur de planification. Mais faire une modification dans un planning a souvent de nombreuses répercussions dans la suite du planning (voir dans ce qui précède). En plus de la construction du planning initial, le système de planification doit donc définir comment un utilisateur peut agir sur le planning et comment réagir à cette action. La réaction peut être un simple déplacement d'une tâche suivie du déplacement de toutes les tâches nécessaires, mais il peut aussi être considéré comme une tâche dont l'exécution (date et ressource utilisée) est imposée. De nombreuses modalités sont possibles, et les différentes techniques possibles relèvent du domaine de l'IHM. Dans cette thèse, nous n'approfondissons pas plus ces techniques même si elles sont nécessaires et doivent être mise en œuvre sur tous les projets.

Dans la première partie (paragraphe 2), nous présentons les systèmes de production. Après une description générale, nous présentons les trois classes de problèmes de planification des systèmes de production ainsi que des exemples de systèmes étudiés.

La seconde partie (paragraphe 3) contient une présentation des principaux modèles théoriques pour la planification à court terme (ordonnancement) des systèmes de production. Outre les principaux problèmes, les extensions majeures sont aussi présentées.

La troisième partie (paragraphe 4) met en évidence notre démarche pour la construction d'applications pour l'optimisation des systèmes de production. Cette démarche consiste à séparer les solutions partielles et complètes, les modules d'évaluation des performances et d'optimisation.

Dans la quatrième partie (paragraphe 5), nous décrivons notre démarche de modélisation. Cette démarche aide à déterminer le degré de finesse avec lequel le système est pris en compte dans le modèle d'optimisation.

La cinquième partie (paragraphe 7) décrit les méthodes de résolution qui peuvent être mises en œuvre sur ce type de problèmes. Elle détaille particulièrement les méthodes de la famille des métaheuristiques.

2 Les systèmes de production

2.1 Description

Un système de production est un système artificiel composé d'unités organisées qui interagissent et interfèrent dans le but de produire des biens ou des services. Un système de production est soumis à une charge qui définit l'ensemble des biens ou des services que le système doit réaliser. Pour écouler cette charge, le système utilise une ou plusieurs ressources.

Plus précisément, la charge d'un système de production est constituée de jobs. Un job suit la réalisation d'un produit dans toutes les étapes de production (de la matière première jusqu'au produit fini). Chaque job a une gamme qui décrit la suite des opérations qu'il doit réaliser. La gamme décrit les opérations à réaliser en donnant :

- leur durée,
- la ou les machines qui doivent réaliser cette opération,
- les ressources consommées (type de ressource et quantité utilisée),
- l'ordre entre les opérations à réaliser (facultatif).

Une gamme correspond donc à un type particulier de produits. Un job a donc une seule gamme mais par contre, plusieurs jobs peuvent avoir la même gamme.

Comme tout système de la vie réelle, les systèmes de production sont finis. Ils ne peuvent donc pas écouler une charge illimitée de travail. Ce qui les limite, ce sont des ressources présentes en quantité limitée : des machines, des opérateurs, des matières premières, des places en stocks, ... On qualifie donc ces systèmes de "système de production à partage de ressources".

C'est en réalisant des opérations que le système produit. Les opérations peuvent être de différents types : opérations de transformation, d'assemblage, de stockage ou de transport. Les ressources sont ce que le système utilise pour produire, elles peuvent être des machines, des opérateurs, des moyens de transport, des emplacements de stockage, des palettes, des outils... Il est difficile d'énumérer tous les types d'opérations et toutes les ressources possibles ; il est tout aussi difficile de donner une définition générale utilisant une terminologie générale car le nombre de domaines comportant des systèmes de production est élevé et leurs terminologies sont diverses et variées. Le tableau 1-1 donne quelques exemples choisis pour être de natures très différentes. Pour chaque domaine, la terminologie employée est illustrée par quelques termes. La terminologie la plus utilisée en systèmes de production est celle des systèmes industriels. Dans la suite de cette thèse, on aura recours à cette terminologie même si nos propositions peuvent être adaptées à d'autres domaines. Sur les

exemples, le lecteur pourra juger de la diversité des systèmes de production sur le plan applicatif comme celui des règles de fonctionnement.

La plupart de ces systèmes sont des systèmes conçus par l'homme. On pourrait donc s'attendre à ce que les différentes questions qu'ils posent trouvent une réponse relativement simple. Cette remarque est d'autant plus vraie si on s'intéresse au cas déterministe, c'est-à-dire au cas où l'on considère connue et déterminée la liste des opérations, ainsi que leurs durées et les ressources nécessaires. Pourtant il n'en est rien, même si on maîtrise ou si on peut maîtriser chacun des aspects et tous les éléments constituants de ces systèmes, leur gestion reste difficile car elle doit être réalisée conjointement sur de nombreux aspects. La difficulté réside en fait dans la complexité du système : nombre élevé d'entités aux relations complexes, grand nombre d'événements possibles, difficulté d'en formaliser certaines parties...

Domaine d'applications	**Type de ressources**	**Type d'opération**
Systèmes informatiques	Microprocesseur, bus, interface homme/machine, console, …	Traitement, sauvegarde sur disque, lecture d'un fichier
Systèmes manufacturiers	Machine, opérateur, outils, moyen de transport, …	Transformation, assemblage, transport, maintenance, …
Systèmes hospitaliers	Docteurs, infirmières, lits, bloc opératoire, équipements, …	Opération chirurgicale, accueil du patient, diagnostic, …
Service	Guichetier, Hôtesse d'accueil, …	Traitement de la demande, redirection vers un autre service, …

Tableau 1-1. Exemples de domaines d'applications, de leurs ressources et de leurs opérations

2.2 Classes de problèmes

Parmi les nombreux problèmes que posent les systèmes de production, on s'intéresse particulièrement aux problèmes liés à leur gestion. Ces problèmes sont généralement regroupés en trois catégories formant trois classes de problèmes, chacune ayant ses propres objectifs en fonction de son horizon : conception, planification, et contrôle (Pirard, 2005).

La conception est l'ensemble des activités effectuées avant l'existence du système ou avant l'existence d'une de ses évolutions. Elle consiste à déterminer les caractéristiques d'un système inexistant (nouveau système) ou d'un système à modifier (futur système). Les problèmes généralement considérés en conception sont le dimensionnement du système, le choix des politiques de gestion des différentes entités, le débit des machines utilisées, le trajet des flux dans le système, ...

La planification consiste à prévoir la réalisation d'un ensemble de tâches, appelé "charge du système", en déterminant les ressources qu'elles utiliseront et comment elles seront traitées dans le temps. On distingue en général trois niveaux de planification en fonction de l'horizon temporel retenu : long terme (stratégique), moyen terme (tactique) et court terme (opérationnel). Ces trois niveaux de planification coexistent et se complètent dans l'entreprise : ils ne sont en général pas réalisés conjointement, mais doivent être cohérents.

Le niveau stratégique de planification correspond aux problèmes de configuration et de reconfiguration des systèmes de production. Les décisions qui peuvent être prises à ce niveau sont l'ajout de ressources supplémentaires (nouvel îlot de production, nouvelle usine, ...), la modification de la charge du système en acceptant la production de nouvelles références, ... Les décisions stratégiques

sont réalisées sur le plus long horizon de planification (de l'ordre de 2 à 5 ans), cette durée dépend largement des systèmes envisagés.

Le niveau tactique de planification correspond à l'utilisation du système, et à son adéquation ressources / besoins. Il s'intéresse à la résolution des problèmes de distribution, de besoins de matières et d'élaboration du plan directeur de production. Les décisions qui peuvent être prises à ce niveau sont par exemple (dans une moindre mesure que le niveau stratégique) : ajout de machine, augmentation de la plage d'ouverture des machines, ... L'horizon de ce niveau de planification est, suivant les systèmes, de l'ordre de 6 à 18 mois.

Le niveau opérationnel de planification correspond au pilotage et à l'ordonnancement du système de production. Il doit résoudre les problèmes de réapprovisionnement des entrepôts, d'élaboration du plan de transport, de détermination des tailles de lots, d'ordonnancement et de suivi d'atelier et enfin d'élaboration des plans d'effectifs et de réapprovisionnements. L'horizon à ce niveau de planification est de l'ordre de la demi-journée à la semaine.

Quel que soit le niveau auquel on s'intéresse, la charge du système est toujours une charge prévisionnelle : aux niveaux stratégique et tactique, la charge dépend des commandes clients ou des consommations des stocks qui sont fluctuantes par essence. Au niveau opérationnel, il est possible que la charge découle directement des activités internes de l'entreprise et donc d'activités planifiées par l'entreprise. Mais, même dans ce cas, des imprévus peuvent survenir : des retards peuvent surgir, des défauts peuvent retarder ou compromettre des livraisons, des opérations jugées plus prioritaires peuvent apparaître. Cette notion de charge prévue montre qu'il est vain de tenter d'optimiser un système à outrance dans la mesure où les décisions prises sur une charge donnée risquent d'être sérieusement remises en cause lorsque l'environnement du système fluctue. Dans les trois niveaux de planification, on doit pouvoir répondre aux questions suivantes : dans quel ordre dois-je écouler la charge prévue, comment respecter la planification du niveau supérieur, comment puis-je intégrer telle ou telle charge non prévue dans le planning actuel ? ...

Le contrôle du système consiste à résoudre les problèmes éventuels lors de la réalisation des opérations. Ainsi, le contrôle doit déterminer comment les conflits d'accès aux ressources doivent être résolus et comment les événements aléatoires interférent sur le déroulement du planning. Ainsi, si le planning prévoit la réalisation d'une opération et que cette dernière ne peut être réalisée par manque de matière ou de ressources, il faut déterminer quelle opération doit être réalisée à sa place : la prochaine opération prévue, une opération du même type ou d'un type approchant...

Schématiquement, on peut dire que, durant la phase de conception, on résout les problèmes avant que le système ou son évolution n'existe. La planification détermine pour un système donné, la charge qui peut être écoulée. Enfin, pour un système et une charge donnés, le contrôle consiste à tenter d'écouler la charge de manière à suivre au mieux le planning. Tous ces problèmes font ou peuvent faire appel aux techniques d'optimisation. Dans la suite de cette thèse, nous étudions particulièrement ces techniques.

2.3 Exemples de systèmes étudiés

Pour illustrer nos propos et justifier le choix des modèles théoriques auxquels on s'intéresse, nous présentons des exemples de systèmes de production. Ces systèmes ont fait l'objet de maquettes informatiques pour un logiciel d'aide à la décision pour la planification. Dans les courtes descriptions suivantes, nous mettons en évidence les modèles théoriques sous-jacents. Ces modèles sont présentés en détail dans la partie suivante.

2.3.1 L'atelier de forge de l'entreprise Aubert & Duval

Aubert & Duval est une entreprise sidérurgique du groupe ERAMET. Le principal site de production (les Ancizes) produit environ 500 nuances de superalliages et d'acier de propriétés différentes. Ce site est composé de secteurs de production comme : la forge, l'aciérie et le laminoir.

Nous nous sommes particulièrement intéressés à l'atelier de forgeage qui confère certaines propriétés mécaniques aux pièces grâce à la déformation à chaud du métal.

Un atelier de forgeage contient des fours dont le rôle est de chauffer les pièces à haute température (1200 à 3500°C). La durée de ces chauffages varie de quelques heures à plusieurs jours. Lorsque la pièce est à la température voulue, elle est conservée dans le four jusqu'à ce que la forge soit disponible et équipée du bon outil. Dès que possible, la pièce est forgée sur la presse pendant un temps qui peut varier de 10 minutes à plusieurs heures. Or, cette pièce refroidit car elle est à température ambiante. C'est pourquoi des allers et retours entre la forge et les fours sont prévus pour maintenir les pièces dans un intervalle correct de températures.

On s'intéresse au problème de planification à court terme de l'atelier de forge dans le cadre d'un outil d'aide à la décision (Caumond *et al.*, 2005d). Actuellement, ce problème est résolu manuellement par une équipe d'experts. Chaque planning est réalisé sur un horizon d'une semaine et nécessite de deux à trois jours de travail. Or ce planning est remis en cause trois à quatre fois par semaine pour prendre en compte les aléas. Chaque modification du planning nécessite de deux heures à plusieurs jours de travail. Ainsi, les experts actuels sont occupés à plein temps pour réaliser les plannings. Pourtant, la direction envisage d'augmenter l'horizon de planification à 6 semaines pour améliorer la capacité d'engagement de l'usine envers ses clients. Ainsi, chaque planning est plus long à réaliser car plus volumineux mais en plus, il est remis en cause plus souvent. En effet, le nombre d'aléas sur une semaine est largement plus faible que le nombre d'aléas sur 6. Il n'est donc pas envisageable d'augmenter l'horizon de planification sans fournir aux experts des outils facilitant leur construction. C'est pourquoi l'outil de planification proposé est orienté "aide à la décision".

Pour réaliser un tel outil, une description précise du système doit être faite. Celle-ci est donnée ci-dessous en trois parties : le sous-système physique qui décrit les entités physiques, le sous-système logique qui décrit les entités traversant le système et le sous-système décisionnel qui contient les règles de gestion.

Sous-système physique : le sous-système physique se compose d'un stock d'entrée et de deux types de machines : les fours et les presses. Les machines sont regroupées en un étage de presses et un étage de fours. La préemption n'est pas autorisée. Les machines sont soumises à des indisponibilités durant lesquelles aucune opération ne peut avoir lieu.

Sous-système logique : l'atelier contient des pièces regroupées en deux types de lots : les fournées et les jobs. Une fournée est composée d'un lot de pièces traitées simultanément par un four. Un job est un lot de pièces traitées en séquence par une presse. Les jobs sont imposés et connus à l'avance, alors que les fournées sont à déterminer. Les jobs passent sur les presses pour réaliser des opérations. Une opération décrit la machine et la durée du traitement. Chaque job possède plusieurs gammes, une gamme définit une liste ordonnée d'opérations. Le choix d'une gamme est réalisé au moment de l'entrée du job dans le système de manière à respecter des contraintes d'incompatibilité (interdiction à priori d'affecter certaines fournées à certains fours). De plus, toutes les gammes respectent la contrainte suivante : toutes les pièces d'un job passent alternativement d'un four à une presse. Lorsque les pièces passent sur le four, elles sont regroupées en fournées. Lorsque les pièces passent sur les presses, elles sont regroupées en job. Les jobs et les fournées sont à priori des lots différents. De plus, entre deux opérations sur la presse, un temps de reconfiguration est nécessaire et une date de début au plus tôt et au plus tard du traitement des jobs est imposée.

Sous-système décisionnel : le sous-système décisionnel est constitué de quatre centres de décisions. Le centre de décision *CD0* coordonne les autres centres. *CD0* contient un centre de décision *CD1* pour les fours et *CD2* pour les presses. Le centre *CD2* doit prendre les décisions suivantes : choisir les gammes des jobs et ordonnancer les jobs sur les presses. Les décisions prises par *CD1* sont globales à l'étage, elles concernent : la constitution des fournées, leur ordonnancement et leur affectation aux fours. Enfin, lorsqu'une fournée est affectée à un four, le centre de décision *CD3* doit résoudre un problème de Bin Packing *3D* pour déterminer le rangement dans le four des pièces constituant la fournée.

En résumé, la résolution du problème industriel inclut la résolution des sous-problèmes suivants (figure 1-1) : un problème d'ordonnancement des jobs en entrée du système, un problème de constitution des fournées, un problème d'affectation des fournées aux fours, un problème de bin packing *3D* de rangement des pièces dans les fours, un problème de choix de gammes (affectation des pièces aux presses) et un problème de planification des jobs sur le four et sur la presse.

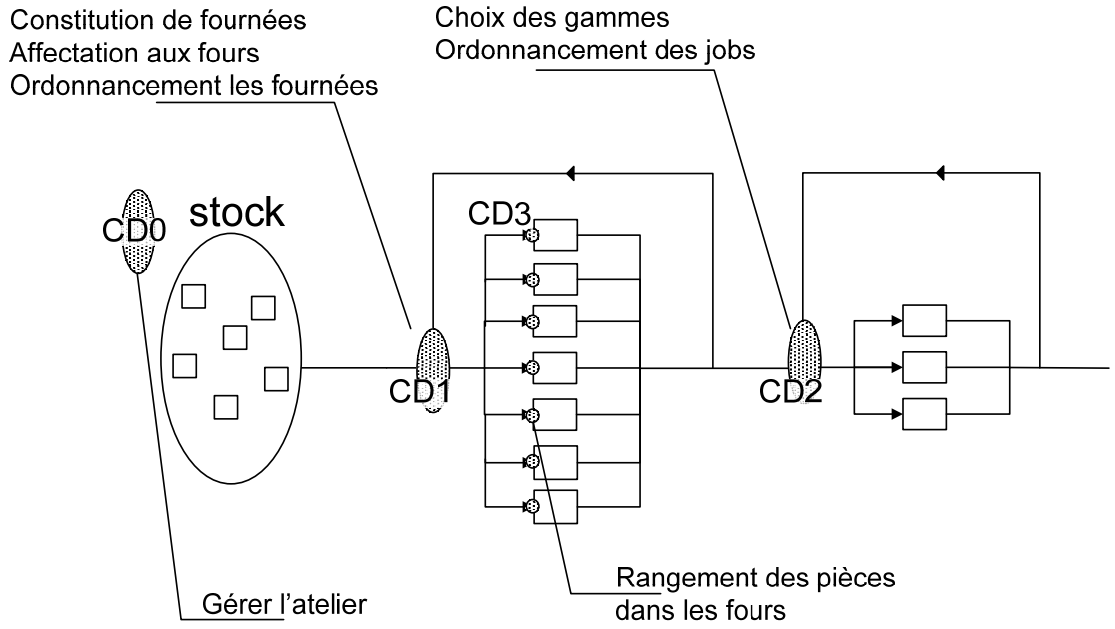

Figure 1-1. Aubert & Duval : Problèmes à résoudre

Nous centrons notre étude sur l'ordonnancement des jobs sur la presse en négligeant la disjonction sur les fours. Ainsi, deux opérations peuvent avoir lieu simultanément sur le même four. Cette hypothèse simplificatrice a à la fois des justifications scientifiques et industrielles :

- le taux d'utilisation des machines montre clairement que les ressources critiques de l'atelier sont les presses,
- le problème de constitution des fournées est résolu par un planificateur dédié à cette tâche. Celui-ci utilise des connaissances non formalisées à ce jour qui lui permettent dans quasiment tous les cas de déterminer des solutions efficaces respectant le processus industriel.

De plus, nous considérons les hypothèses simplificatrices suivantes :

- les indisponibilités peuvent être négligées : sur un horizon d'une semaine, les ressources peuvent être considérées comme disponibles ou indisponibles sur toute la durée de l'horizon de planification,
- les dates au plus tôt et au plus tard de traitement sont imposées par un système de planification à un niveau supérieur. L'outil de gestion de production proposé réalise une planification à la maille semaine. Puisque notre outil planifie semaine par semaine, la contrainte au plus tôt et au plus tard devient donc inutile,
- à la sortie de l'atelier, une pièce peut visiter un autre atelier et éventuellement revenir à l'atelier de forge. Il y a donc des contraintes de précédence entre deux jobs qui concernent les mêmes pièces. Pour un niveau de planification à la semaine, il est improbable que deux jobs violent une éventuelle contrainte de précédence. Nous négligeons donc cette contrainte, toute solution irréalisable peut éventuellement être réparée au niveau de l'outil d'aide à la décision,
- entre deux jobs, les temps de reconfiguration sont sensiblement identiques et monopolisent la ressource pendant le temps de la reconfiguration. On peut donc les incorporer dans les temps de traitement.

2.3.2 Systèmes Flexibles de Production (SFP)

Les systèmes flexibles de production sont définis par MacCarthy et Liu (1993) :

> *Un système flexible de production est un système de production capable de produire différents types de pièces, composé de machines à commande numérique ou à contrôle numérique et d'un système automatisé de stockage connectés par un système automatisé de manutention. Le fonctionnement du système entier est sous le contrôle et le pilotage d'un système informatique.*

Plus précisément, un système flexible de production se compose d'un ensemble de stations reliées entre elles par un véhicule. Chaque station se compose de trois parties principales (cf. figure 1-3) : un stock d'entrée, une unité de traitement et un stock de sortie. Chaque pièce qui entre dans une station passe tout d'abord par le stock d'entrée. Lorsque l'unité de traitement est disponible, une des pièces du stock d'entrée est choisie puis transite vers l'unité de traitement. L'opération sur l'unité de traitement commence alors. À la fin de cette opération, et sous la condition qu'une place soit disponible dans le stock de sortie, la pièce quitte l'unité de traitement pour aller dans le stock de sortie. Dès son entrée dans ce stock, la pièce est détectée et un appel au véhicule est réalisé. Le véhicule se déplace d'une machine à l'autre à l'aide d'un réseau appelé "réseau de guidage" (cf. figure 1-2). Lorsque le véhicule est disponible, le système informatique de pilotage choisit quelle pièce va être traitée. Un déplacement commence alors entre la position courante du véhicule et le stock de sortie qui contient la pièce à déplacer. Ce déplacement est appelé "déplacement à vide". À l'aide d'un système automatisé, le véhicule est capable de prendre la pièce sur le stock de sortie. Le transport de la pièce commence alors vers le stock d'entrée de la prochaine machine à visiter. Ce déplacement est appelé "déplacement en charge". La pièce est alors automatiquement transférée du transporteur vers le stock d'entrée de la machine. Tous les déplacements sont réalisés suivant un trajet de durée prédéterminée. Pour des contraintes techniques, l'ordre d'entrée et l'ordre de sortie des pièces dans les stocks sont souvent PAPS (Premier Arrivé Premier Servi).

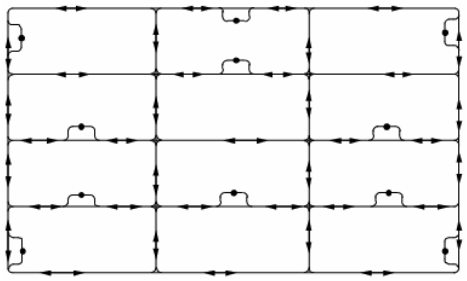

Figure 1-2. Réseau de transport

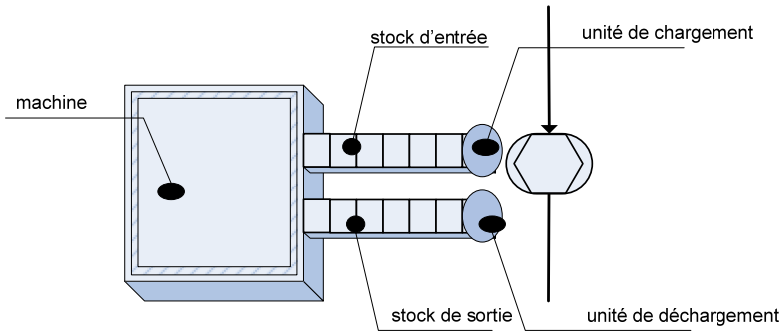

Figure 1-3. Une station

Les systèmes flexibles de production (SFP) sont soumis aux interblocages. Une situation d'interblocage a lieu lorsque le système est dans une configuration telle qu'une intervention manuelle est nécessaire. La figure 1-4 présente un exemple de situation d'interblocage : les stocks d'entrée et de sortie des stations sont pleins car leur capacité est atteinte : de plus, les unités de traitement sont occupées par une pièce. Chaque machine est donc bloquée car une intervention extérieure est nécessaire pour pouvoir libérer une place en stock. Or la pièce sur l'unité de déchargement de la station de gauche doit visiter la station de droite, et inversement. Le transporteur ne peut donc commencer aucun des deux transports. Il n'est pas possible non plus d'attendre le déblocage de cette situation.

Pour éviter les interblocages, beaucoup de systèmes flexibles de production limitent le nombre de jobs simultanément autorisés dans le système. Soit N la somme des capacités des stocks d'entrée et de sortie des deux machines ayant la plus faible capacité. Si on limite le nombre de jobs simultanément autorisé à N, il est certain que la situation d'interblocage n'aura pas lieu. Pour un système qui ne limite

pas le nombre de jobs, on peut considérer qu'il est soumis à la contrainte avec une valeur suffisamment grande de N.

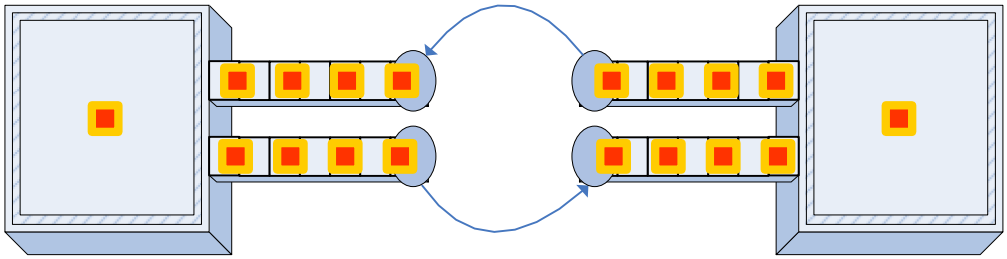

Figure 1-4. Situation d'interblocage d'un système flexible de production

3 Modèles théoriques

Les systèmes de production sont très variés et on pourrait penser que leurs modèles le sont aussi. Pourtant, la littérature a mis en évidence un relativement faible nombre de modèles théoriques pour l'ordonnancement des systèmes de production. Ces modèles sont très généraux et peuvent être appliqués à des problèmes variés, ils modélisent des parties de systèmes de production et / ou des systèmes de production simplifiés. Même s'ils peuvent être appliqués à des domaines plus larges, la terminologie qu'ils emploient est essentiellement tournée vers les systèmes industriels de production (machine, job, opération et ressources, …).

Plusieurs définitions des problèmes d'ordonnancement sont présentées dans la littérature suivant les auteurs.

D'après Blazewicz (1996) (traduit de l'anglais) :

> *En général, les problèmes d'ordonnancement sont caractérisés par trois ensembles : l'ensemble T constitué de n tâches T = {T1, T2, ..., Tn}, l'ensemble P = {P1, P2, ..., Pm} de m processeurs (machines) et l'ensemble R = {R1, R2, ..., Rs} des ressources additionnelles. L'ordonnancement, de manière générale, est l'affectation des processeurs de P et (éventuellement) des ressources de R aux tâches de T dans le but de réaliser toutes les tâches en respectant les contraintes.*

Ou encore, d'après Carlier et Chrétienne (1988) :

> *Ordonnancer, c'est programmer l'exécution d'une réalisation en attribuant des ressources aux tâches et en fixant leurs dates d'exécution.*

3.1 Problèmes classiques

Tous les problèmes d'ordonnancement sont constitués d'un ensemble de jobs à ordonnancer sur un ensemble de machines. Un job doit réaliser des traitements sur une ou plusieurs machines. La description des traitements à réaliser et leur éventuel ordre respectif sont appelés "gamme". Suivant le type de problème d'ordonnancement étudié, les gammes sont très différentes. Sauf mention contraire, la durée des opérations à réaliser est donnée et connue à l'avance. Les problèmes suivants ne diffèrent que par la définition de leur gamme :

- Pour le problème à une machine, les gammes consistent à visiter l'unique machine durant un temps prédéterminé,

- Pour le problème de flowshop, tous les jobs ont la même gamme. Une gamme consiste à visiter toutes les machines dans un ordre donné,
- Pour le problème de jobshop, chaque job a sa propre gamme. Une gamme consiste à visiter un ensemble de machines dans un ordre donné,
- Pour le problème d'openshop, chaque job a également sa propre gamme. Une gamme consiste alors à visiter un ensemble de machines dans un ordre à déterminer,
- Pour le problème à machines parallèles, tous les jobs ont la même gamme. Une gamme consiste à visiter une des machines parallèles, le choix de la machine est à déterminer,
- Pour le problème du RCPSP, chaque job doit réaliser une opération en consommant des ressources. Les ressources sont disponibles en nombre limité et sont typées, i.e. on ne peut pas, à priori, utiliser une ressource à la place d'une autre. On distingue les ressources renouvelables, qui sont disponibles pour une nouvelle opération après la fin d'une opération, et les ressources non renouvelables qui ne peuvent être utilisées qu'une seule fois.

Nous ne saurions être exhaustifs et nous présentons donc simplement une typologie adaptée à nos besoins et permettant d'organiser les modèles théoriques connus. Cette typologie, adaptée de (Bertel, 2001), est présentée en figure 1-5.

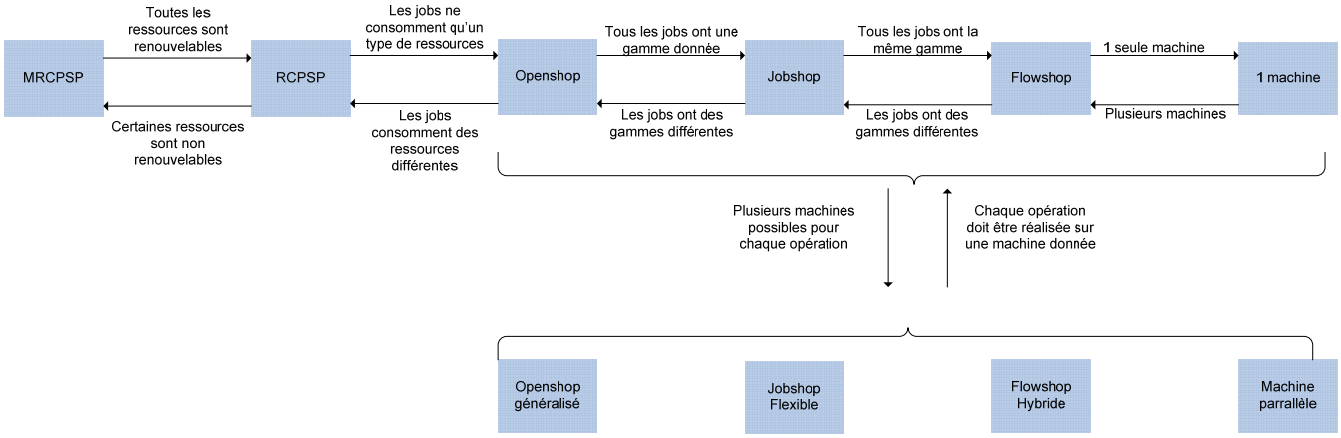

Figure 1-5. Typologie des problèmes d'ordonnancement

3.2 Extensions

Tous ces problèmes ont des extensions possibles, les principales extensions sont :

- la flexibilité : essentiellement présente pour les problèmes de flowshop, de jobshop et d'openshop ; on parle alors de "flowshop hybride", de "jobshop flexible" et d'"openshop généralisé". On trouve dans la littérature plusieurs types de flexibilité. Cette flexibilité consiste à proposer plus d'une machine pour réaliser l'opération. L'ordonnancement consiste alors à choisir les dates de début des opérations mais aussi à choisir sur quelle machine on réalise chaque opération. On doit donc résoudre un problème supplémentaire d'affectation des opérations aux machines,
- des données stochastiques sont introduites pour prendre en compte l'incertitude des données récoltées pour le modèle. Par exemple, on peut introduire une incertitude sur les temps de traitement et la modéliser par une variable aléatoire.
- la disponibilité des ressources : sur l'horizon de temps considéré, il est possible qu'une ou plusieurs ressources soient non disponibles,
- les versions réactives ou prédictives : dans une version prédictive, toutes les informations sont disponibles dès le début du processus d'optimisation. Dans une version réactive, les décisions sont prises au fur et à mesure que le temps avance et que

de nouvelles informations arrivent. Les versions réactives sont particulièrement adaptées aux problèmes temps réels dans lesquels certaines décisions sont prises avant que toutes les informations ne soient disponibles,

- les contraintes de temps : elles restreignent les dates de début des opérations. Parmi les contraintes de temps, on distingue les time lags, les dates au plus tôt (due dates) et les dates au plus tard (release dates),
- les contraintes de précédence : elles imposent des ordres relatifs entre opérations. Ainsi si une opération *a* précède une opération *b* c'est que son exécution doit avoir lieu avant celle de l'opération *b*. Certains problèmes contiennent dans leur définition des contraintes de précédences comme le problème de flowshop ou de jobshop,
- les ressources supplémentaires : cette extension permet de modéliser des ressources qui ne sont pas des machines mais qui doivent tout de même être considérées lors de l'ordonnancement. On trouve par exemple les types de ressources suivants : ressources de transport, ressources humaines comme les opérateurs de maintenance, ou les opérateurs classiques.

3.3 Conclusion

Parmi les problèmes d'ordonnancement présentés, nous nous sommes particulièrement intéressés au problème de jobshop car il généralise la plupart des modèles d'atelier. Depuis la deuxième moitié des années 60, ce problème suscite de très nombreux travaux. Entre autres, des algorithmes d'optimisation particulièrement efficaces ont été développés (cf. chapitre 2).

Pourtant, le problème de jobshop pose les hypothèses simplificatrices suivantes : capacité de stockage suffisante (considérée infinie), temps et ressources de transport négligés, disponibilités complètes des pièces et des machines, temps de traitement des opérations connus et déterminés, ... Mais ces hypothèses sont parfois trop restrictives, et des modèles plus complexes doivent souvent être envisagés. En particulier, les problèmes présentés dans le paragraphe 2.3 peuvent être vus comme des problèmes de jobshop avec extensions : le problème "Aubert & Duval" peut être vu comme un problème de jobshop avec time lags, les "Systèmes Flexibles de Production" se modélisent comme un problème de jobshop avec transport.

A priori, il n'est pas possible de réutiliser directement les algorithmes d'optimisation efficaces pour le jobshop. Ces algorithmes doivent être réadaptés pour prendre en compte des contraintes supplémentaires. Dans la suite, nous proposons une démarche d'optimisation et une démarche de modélisation qui permettent de construire un algorithme d'optimisation tout en tirant partie des algorithmes efficaces.

4 Démarche d'optimisation

La plupart des problèmes de planification des systèmes de production sont, ou se ramènent à des problèmes d'optimisation. Dans ce paragraphe, nous décrivons la démarche d'optimisation que nous avons adoptée. Cette démarche est basée sur l'utilisation de l'architecture présentée dans (Grangeon, 2001) (cf. figure 1-6). Elle consiste à séparer (au moins conceptuellement) l'optimisation de l'évaluation des performances. On obtient alors les deux modules distincts : le module d'évaluation des performances et celui d'optimisation.

Les deux modules communiquent afin de construire une solution optimisée. Afin d'illustrer notre démarche, considérons le problème de jobshop. Le module d'optimisation propose des solutions partiellement définies, c'est-à-dire des solutions dans lesquelles manquent des informations. Dans la littérature, on trouve différentes solutions partielles utilisées pour le problème de jobshop : un ordre total des opérations, un ordre des opérations sur chaque machine, des matrices de précédence, … Le module d'évaluation des performances répond par une solution complétée et les valeurs des critères de performances pour cette solution. Pour le problème de jobshop, la solution complétée est un ordonnancement qui décrit pour chaque opération la date de début de celle-ci. La valeur du critère de

performance est lui aussi variable, mais dans le problème de jobshop classique, c'est le makespan (i.e. la date de fin de la dernière opération). Ces valeurs sont exploitées par le module d'optimisation pour proposer d'autres solutions.

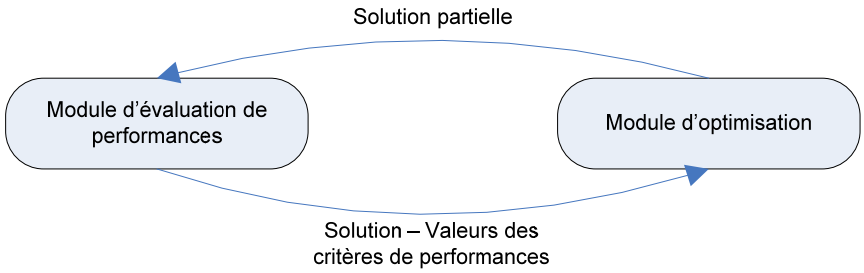

Figure 1-6. Architecture optimisation / évaluation des performances

(Grangeon, 2001)

Les trois paragraphes suivants décrivent plus précisément les éléments du schéma ci-dessus.

4.1 Solutions partielles

(Grangeon, 2001) qualifie de solutions "partielles" les solutions proposées par un algorithme d'optimisation, car elles ne contiennent pas toutes les informations nécessaires à la description complète de la solution. Par exemple, en planification, une solution est un planning définissant précisément et pour chaque ressource la liste des tâches réalisées ainsi que les dates auxquelles elles sont réalisées. Une "solution partielle" de ce problème est par exemple : une liste de jobs par ordre de priorité ou des ordres d'opérations sur les ressources ...

L'utilisation de solutions partielles permet de rendre plus compactes les informations manipulées et surtout permet de les rendre plus facile à modifier.

4.2 Module évaluation des performances

Le module d'évaluation des performances reçoit une solution partiellement définie pour renvoyer les valeurs des critères de performances. Il a donc pour tâche de retourner la valeur du (ou des) critère(s) de performance correspondant à une solution partielle. Puisque la solution fournie n'est que partielle, il peut y avoir plusieurs solutions correspondantes de coûts variés. Parmi toutes ces solutions, le module d'évaluation ne doit en retenir qu'une seule dont il renvoie les valeurs de critères de performances.

On peut classer les modules d'évaluation des performances en trois classes (Sarramia, 2002) dont la complexité est croissante (cf. figure 1-7).

La première classe est constituée des fonctions triviales pour le calcul des performances. Ces fonctions apparaissent en optimisation quand le calcul des critères de performances à partir d'une solution partielle est trivial (i.e. il peut être décrit par une fonction littérale). Les problèmes suivants ont un module "évaluation des performances" trivial : problème de voyageur de commerce, de flowshop ou du plus long chemin... Il n'est pas courant de faire apparaître explicitement un module d'évaluation des performances quand l'évaluation est aussi triviale mais nous en voyons l'intérêt dans la suite de cette thèse.

La deuxième classe est constituée des "procédures d'évaluation des performances". Ce type d'évaluation apparaît quand le module d'évaluation des performances est explicite dans le logiciel d'optimisation et réalise des calculs relativement complexes. Il s'agit par exemple des : flowshop avec

stock, Systèmes Flexibles de Production, ... Pour ces problèmes, les procédures d'évaluation sont par exemple des calculs de plus longs chemins, ...

La troisième classe de modules d'évaluation des performances est constituée des modèles qui sont explicitement des modèles de simulation. Des exemples de tels modules d'évaluation sont des modules d'évaluation des performances écrits dans un langage de simulation ou écrit à l'aide d'une bibliothèque de simulation. On parle dans ce contexte de "couplage simulation / optimisation" pour désigner l'utilisation conjointe de ces deux approches. Le modèle de simulation peut être réalisé à l'aide d'un langage de simulation, à l'aide d'une bibliothèque de simulation ou bien à l'aide d'un simulateur dédié écrit dans un langage de programmation. Il est difficile de faire la différence entre un simulateur dédié et une procédure d'évaluation des performances. En effet, tous deux sont écrits dans un langage de programmation quelconque. Et même si la procédure ne fait pas apparaître de manière explicite un modèle de simulation, il est assez souvent possible de la réécrire comme un modèle de simulation. Ceci n'est pas toujours possible et nous montrons dans le paragraphe 3.3 du chapitre 2 que tous les ordonnancements ne peuvent être générés à l'aide d'un modèle classique de simulation.

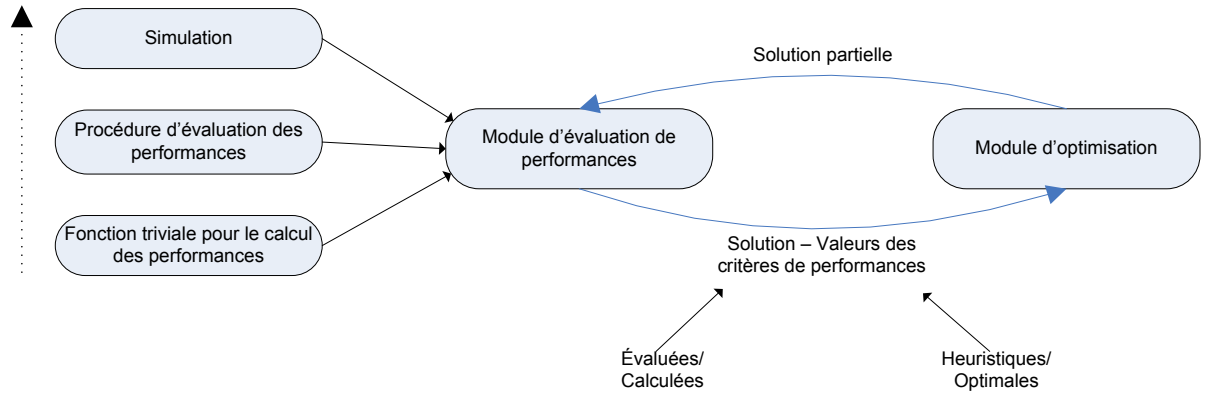

Figure 1-7. Principaux types de modules d'évaluation des performances (Sarramia, 2002).

4.3 Solutions - Critères

À l'issue du module d'évaluation des performances, on obtient une solution et ses critères de performances.

Critère calculé et critère évalué

Un module d'évaluation propose un critère calculé si le résultat du calcul est déterministe, c'est-à-dire qu'une solution partielle mène toujours à la même valeur de critère calculé.

Sinon, on dit que le critère est évalué. C'est le cas lorsque le module d'évaluation comporte des parties stochastiques. Ainsi, le module a recours à des estimateurs statistiques dont le résultat dépend des générateurs aléatoires utilisés.

4.4 Module d'optimisation

Le module d'optimisation est le module qui construit ou recherche une solution optimisée. Suivant les types d'optimisation envisagés, le module peut utiliser zéro, une unique ou plusieurs solutions afin d'en proposer une nouvelle. Quoi qu'il en soit, il nécessite toujours le module d'évaluation des performances pour évaluer la ou les solutions générées.

5 Représentation

Du point de vue du module d'optimisation, un problème est complètement défini à partir du moment où l'on peut associer un coût à chaque solution partielle. La fonction réalisant cette association est appelée "fonction coût".

Si la donnée d'une fonction coût permet de définir complètement un problème, la réciproque n'est pas vraie : à un problème donné correspond plusieurs fonctions coûts possibles. Les différentes manières de décrire le problème qui amènent à différentes "fonction coûts" sont appelées "représentation" dans la suite. Pour un problème donné, nous envisagerons donc plusieurs représentations pour un même problème.

5.1 Introduction

Suivant les propriétés de la solution calculée par le module d'évaluation, on qualifie le module d'évaluation des performances d'optimal ou d'heuristique, de calculer ou d'évaluer.

Suivant la manière dont les solutions complètement définies sont construites, on distingue deux catégories de modules : les modules d'évaluation des performances optimaux et heuristiques.

Module d'évaluation des performances optimal

Un module d'évaluation est optimal s'il prend les meilleures décisions relativement à ce qui est décrit dans la solution partielle.

Si un module d'évaluation n'est pas optimal, on dit qu'il est heuristique.

En règle générale, et si possible, on préfère utiliser des modules optimaux. Mais, lorsque le problème est trop complexe, on considère fréquemment des heuristiques pour résoudre de manière approchée une partie des décisions. Soit, par exemple, un problème de planification dont une solution partielle est un ordre de jobs. Si l'affectation des jobs aux ressources est réalisée de manière heuristique par le module d'évaluation, on dit que ce module est heuristique.

Les modules heuristiques sont principalement utilisés pour des problèmes complexes où ils permettent de résoudre de manière approchée une partie des décisions à prendre. Ce faisant, les décisions représentées par une solution partielle sont donc moins nombreuses et donc plus faciles à appréhender. Dans un problème de planification dont les opérations peuvent être réalisées sur plusieurs machines, la solution partielle contient à la fois un ordre des opérations et l'affectation des opérations aux machines. Choisir de manière heuristique l'affectation des opérations aux machines permet de restreindre la solution partielle à l'ordre des opérations et donc à en réduire la taille. Ainsi, l'espace de recherche est plus facile à appréhender et à manipuler.

5.2 Propriétés des représentations

Étant donné un modèle, il existe plusieurs manières de construire la fonction coût. Suivant les cas, on obtient des résultats différents et les méthodes d'optimisation sont plus ou moins efficaces. Autrement dit, la définition de la fonction coût est essentielle pour la réussite d'un projet en optimisation. C'est pourquoi nous nous intéressons particulièrement à cette transformation. Dans la suite, on appelle "représentation" une façon de transformer un problème en une fonction coût. Une représentation est donc dépendante d'un problème, et plusieurs représentations sont donc possibles pour un problème donné. Par exemple, nous présentons 9 représentations possibles et non équivalentes pour le problème de jobshop.

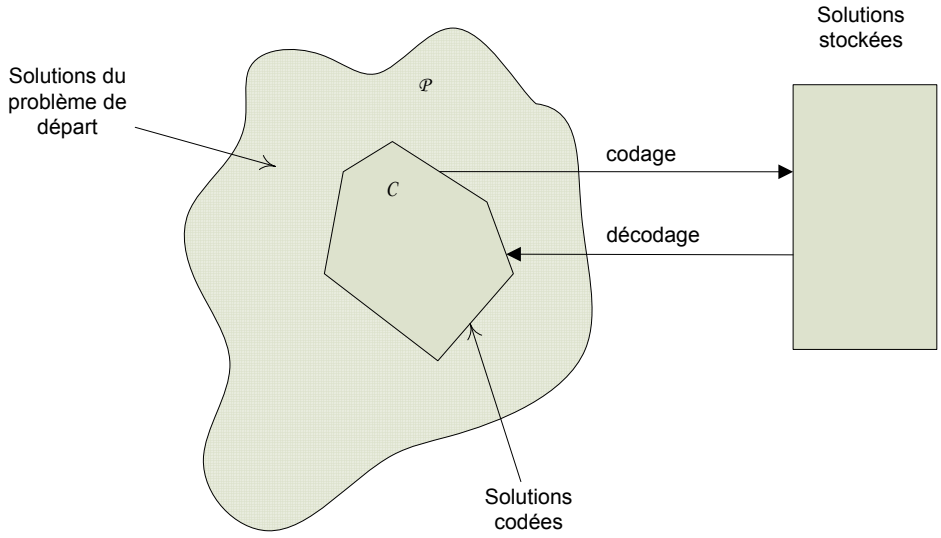

Figure 1-8. Schéma de principe de la représentation

Plus précisément, une représentation définit les solutions envisagées par l'algorithme d'optimisation, appelées "solutions codées", et les solutions telles qu'elles sont stockées informatiquement (appelées "solutions stockées"). Les algorithmes permettant de passer d'une solution stockée à une solution codée sont appelés "algorithmes de décodage". Les algorithmes permettant de passer d'une solution codée à une solution stockée sont appelés "algorithmes de codage" (cf. figure 1-8). La représentation s'attache particulièrement à vérifier les propriétés des solutions stockées par rapport aux solutions codées et des solutions codées par rapport aux solutions du problème. Ces deux liaisons sont caractérisées par les propriétés de "Sur représentativité", "Multi représentativité" et "Heuristique", propriétés que nous détaillons par la suite.

Introduisons les notations suivantes : $\wp$ est l'ensemble des solutions du problème de départ, C est le sous-ensemble des solutions codées et S est l'ensemble des solutions stockées. L'algorithme de codage transforme une solution de C en une solution de S, inversement, l'algorithme de décodage transforme certaines solutions de S en solutions de C.

Une solution stockée est la structure de données qui sert de support au stockage des solutions, cela peut être un tableau à une ou deux dimensions, contenant des dates, un ordre, des priorités, des opérations .... En optimisation, il est d'usage d'appeler "solution" cette donnée, mais il n'est pas nécessaire que toutes les solutions stockées représentent des solutions du problème de départ, il n'est même pas nécessaire que toutes les solutions du problème de départ correspondent à une solution stockée. La définition seule d'une solution stockée ne suffit donc pas à définir une représentation, il est aussi important de définir comment une solution stockée est interprétée, c'est-à-dire comment sont définis par les algorithmes de codage et de décodage. Ainsi, une représentation n'a de sens que pour un problème donné, alors qu'un codage (terme employé en algorithme génétique : codage binaire, de codage par matrice, par matrice latine, par liste, …) est constant d'un problème à l'autre : c'est pourquoi il ne faut pas confondre représentation et codage.

Pour définir les représentations, nous proposons d'utiliser les propriétés suivantes :

- Sur représentation : une représentation est dite "sur représentée" si toutes les solutions stockées ne correspondent pas à une solution codée. En d'autres termes, une représentation est "sur représentée" si l'algorithme de décodage peut échouer. On dit dan s ce cas que la solution générée est irréalisable. Ce type de représentation apparaît fréquemment dans les problèmes fortement contraints ou dans les représentations faiblement contraintes.
- Multi représentation : une représentation est dite "multi représentée" s'il existe deux solutions stockées correspondant à une même solution codée. Dans ce cas, l'algorithme de codage possède plusieurs solutions stockées à sa disposition et doit en choisir une ;

on dit que l'algorithme de décodage est équivoque (sinon, on dit qu'il est univoque). On fait de plus la distinction entre les représentations multi représentées et les représentations dont une restriction ne l'est pas. Dans ce deuxième cas, ce sont des représentations multi représentées pour lesquelles on peut transformer toutes les solutions stockées de manière à ce qu'il existe un algorithme de décodage univoque. Elles sont alors qualifiées de multi représentative restreinte.

- Heuristique : une représentation est dite heuristique si elle ne permet pas de coder de manière certaine la solution optimale. Une optimisation basée sur une représentation heuristique ne permet pas de trouver de façon certaine la solution optimale, elle permettra dans le meilleur des cas de trouver la meilleure solution pour la représentation.

6 Démarche de modélisation

Le rôle de la démarche de modélisation est de guider l'expert dans la création d'applications en optimisation. Dans la partie précédente, nous proposons de scinder ces applications en deux modules : le module d'optimisation et le module d'évaluation des performances. Dans cette partie, nous présentons la démarche de modélisation qui aide à la construction de ces deux modules. Une étape importante de la démarche est de construire un modèle de connaissances. C'est en utilisant ce modèle de connaissances que l'on peut construire le module d'évaluation des performances. On peut, en outre, déterminer les axes d'amélioration d'une solution, ce qui permet de construire le module d'optimisation.

Outre l'aide méthodologique, nous proposons une démarche pour la détermination de la finesse du modèle. Cette démarche consiste à introduire les hypothèses simplificatrices de manière progressive et incrémentielle. Plus précisément, elle consiste à repartir des modèles théoriques des problèmes de la littérature car ces algorithmes sont particulièrement efficaces et particulièrement travaillés. Ensuite, nous levons les hypothèses simplificatrices les unes après les autres. À chaque hypothèse supprimée, le module d'évaluation des performances est éventuellement modifié pour mettre à jour le calcul des critères de performances. De plus, les axes d'amélioration d'une solution peuvent aussi avoir besoin d'être adaptés tout en essayant de ménager les fonctionnalités introduites pour le modèle plus simple.

Dans la suite, nous présentons la notion de modèle (paragraphe 6.1). Puis, nous présentons le processus de modélisation dans le cadre général (paragraphe 6.2), puis notre adaptation du processus aux applications d'optimisation (paragraphe 6.3). Dans le paragraphe 6.4, nous présentons les trois types de modèles retenus pour la modélisation.

6.1 Notion de modèle

Le monde que nous avons créé est le résultat de notre niveau de réflexion, mais les problèmes qu'il engendre ne sauraient être résolus à ce même niveau. *Einstein, Albert*

Cette citation d'Albert Einstein s'applique à la modélisation des systèmes de production. En effet, les systèmes de production sont des systèmes conçus par l'homme et donc à notre niveau de réflexion. Chaque entité du système est conçue pour un but précis et l'organisation du système est imposée par les concepteurs. Pourtant, les problèmes qu'ils posent sont à un niveau de réflexion supérieur : pour résoudre ces problèmes il est généralement nécessaire d'avoir recours à la modélisation pour pouvoir travailler sur des simplifications et sur des décompositions du système à étudier.

Minsky donne une définition de la notion de modèle.

Modèle (Minsky, 1968)

Pour un observateur *A*, ß est un modèle du système *B* si *A* peut, à partir de ß apprendre quelque chose d'utile sur le fonctionnement de *B*.

Pour illustrer cette définition, nous introduisons la figure 1-9 qui schématise la notion de modèle. Dans le schéma, le système réel est la forme irrégulière, le modèle est le polygone à l'intérieur. Ce schéma illustre graphiquement les trois propriétés que le modèle doit avoir relativement au système modélisé (Popper, 1973) :

- Il doit ressembler au système qu'il modélise. Cette ressemblance n'est malheureusement ni quantifiable ni formalisable. On dit que le modèle ressemble au système si on peut apprendre quelque chose d'utile sur le système avec le modèle. C'est donc tardivement, au moment de l'exploitation du modèle, que l'on peut se rendre compte si un modèle est ressemblant ou non. La ressemblance est schématisée par le fait que les formes du modèle et du système sont similaires.
- Il doit être une simplification du système. Si le modèle renferme toute la complexité, c'est que le modèle sera probablement difficile à exploiter. Les simplifications usuelles sont l'ablation d'une partie du système ou la négligence de phénomènes observés (pannes, événements aléatoires, opération de durée négligeable ...). Sur le schéma, les bords du modèle sont à l'intérieur des bords du système, ce qui signifie que certaines parties n'ont pas été prises en compte.
- Il doit être une idéalisation du système. Pour certaines parties ou pour certains phénomènes, on peut vouloir ne pas les supprimer mais les prendre en compte de manière idéale (opération de durée constante, disponibilité complète des matières premières, etc.). Sur le schéma, les contours du modèle sont plus réguliers que ceux du système.

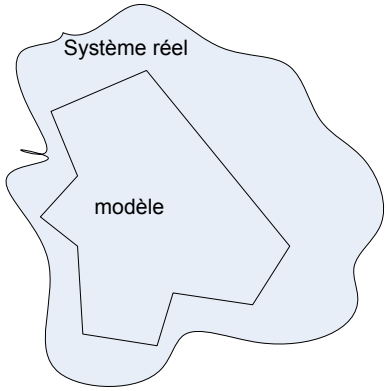

Figure 1-9. Un modèle et son système

Parmi ces trois propriétés, la propriété de ressemblance est la plus implicite. Pendant la phase de modélisation, on discute principalement des propriétés de simplification et d'idéalisation du modèle. Ces deux propriétés sont regroupées sous le vocable "hypothèses simplificatrices". Une hypothèse simplificatrice s'énonce par une phrase négative indiquant que telle ou telle caractéristique du système n'est pas prise en compte. Des exemples d'hypothèses simplificatrices sont "On néglige les frottements d'une roue sur le sol", "On néglige l'influence de la gravité sur une particule électrique en mouvement", "On néglige les pannes du système", ...

La définition faite par Minsky indique que le modèle β est utile à l'observateur A. En effet, les hypothèses simplificatrices réalisées sur ce modèle dépendent de l'observateur et des questions qu'il se pose. Par exemple, si le problème consiste à déterminer les performances du système de transport, il n'est pas souhaitable de trop simplifier celui-ci. On peut par contre l'idéaliser dans la mesure où l'on reste conscient des hypothèses faites : étudier les problèmes aux carrefours des systèmes de transport n'est pas pertinent si ceux-ci ne sont pris en compte qu'à l'aide de leur temps de transport. Ce point est très important en modélisation et les exemples traités peuvent laisser penser que ces considérations sont triviales; pourtant, il n'est malheureusement pas rare que ce soit un facteur d'échec pour de nombreux projets de modélisation.

Puisque l'observateur est central en modélisation, les modèles développés sont forcément relatifs à la problématique qui l'intéresse. A priori, chaque nouvelle problématique nécessite donc le

développement d'un nouveau modèle. Ainsi, on ne peut pas développer un modèle universel pour un système. La réutilisation des modèles est donc envisageable dans une certaine mesure. Certains aspects de réutilisation sont discutés dans la suite de cette thèse.

De nombreux types de modèles existent. Un modèle physique est traduit sous forme de phénomènes concrets. Il passe par une maquette qui représente le système soit de manière réduite (modèle réduit automobile), soit de manière dynamique (modèle aéronautique), soit extrait de son milieu naturel (skieur en soufflerie), ...

Un modèle symbolique, par opposition au modèle physique, est traduit dans un formalisme abstrait. Il peut s'agir d'un formalisme mathématique (on parle de modèle mathématique), d'un langage de simulation (on parle de modèle de simulation), ...

Quel que soit le support utilisé, construire un modèle n'est jamais une chose facile. Comme le souligne J. Mélèse (Mélèse, 1990) :

> *un manager qui présente et exprime dans son langage et à son niveau, ses préoccupations à un spécialiste, va recevoir des réponses totalement différentes suivant le corps de métier auquel il a cru bon s'adresser.*

Pour réussir l'étude, il est donc nécessaire de faire communiquer directement ou indirectement les différents experts du système. En effet, il est rare qu'un seul individu possède toute la connaissance nécessaire et qu'il soit capable de la formaliser. L'étude passe donc généralement par des phases de rédaction pour permettre les échanges. Contrairement aux langages naturels (tel que le Français), les formalismes de modélisation permettent de rédiger de manière univoque. Les formalismes de modélisation sont utiles pour :

- Recueillir les connaissances des différents experts concernés,
- Confronter ces connaissances entre les différents experts,
- Exploiter ces connaissances.

Les formalismes de modélisation sont nombreux, nous citerons : UML (Unified Modeling Language), diagrammes Entités / Associations, SADT, ... Malgré l'existence de ces formalismes, la phase de modélisation reste difficile. Il est largement reconnu que "la modélisation est un art", car le succès de la phase de modélisation ne peut être assuré et dépend largement des compétences de l'expert en modélisation : aucune méthode ne permet de déterminer les hypothèses simplificatrices à réaliser, rien ne permet d'affirmer que le modèle ressemble suffisamment au système étudié, le modèle est tributaire de la qualité des connaissances recueillies auprès des experts. Notons tout de même qu'un modèle peut être vérifié (la cohérence interne du modèle est étudiée) et validé (les résultats sont comparés avec des états connus du système).

Même si on ne peut guider un expert en modélisation sur le choix des hypothèses simplificatrices, il est tout de même possible de lui apporter une aide méthodologique. Les méthodologies de modélisation fournissent un cadre, conseillent l'expert en lui donnant les étapes par lesquelles il doit passer, en proposant des outils et des méthodes pour lui permettre de mener à bien le projet de modélisation.

6.2 Processus de modélisation

Même s'il n'est pas possible de guider l'expert sur le choix des hypothèses simplificatrices, nous présentons le processus de modélisation issu de (Gourgand, 1998) et (Tchernev, 1997). Ce processus (figure 1-10) guide l'expert durant la phase de modélisation, il préconise la construction des modèles de connaissances et d'action et est composé de quatre étapes : la première consiste à construire le modèle de connaissances, la deuxième à élaborer le modèle d'action, la troisième à exploiter le modèle ainsi réalisé et la dernière étape consiste à prendre des décisions et à agir sur le système.

La première étape (aussi appelée "phases d'analyse et de spécification du système") consiste à construire le modèle de connaissances. Ce modèle contient la description des entités du système et de

leurs interactions. Par contre, il ne doit contenir aucun élément de la solution qui sera éventuellement mise en œuvre pour résoudre le problème.

Dans la deuxième étape, on élabore le modèle d'action à partir du modèle de connaissances. Le modèle d'action est le modèle que l'on souhaite mettre en œuvre pour résoudre le problème. Il décrit donc, dans le formalisme choisi, la partie modélisée du système ainsi que la démarche de résolution. On élabore le modèle d'action relativement à la connaissance acquise dans le modèle de connaissances.

L'étape trois consiste en l'exploitation du modèle d'action. Elle contient donc les éventuels réglages des méthodes proposées ainsi que l'exploitation et la synthèse des résultats. C'est grâce au modèle de résultats que l'on peut décrire les différents résultats obtenus et la manière dont ils sont synthétisés.

La quatrième et dernière étape contient la prise de décisions et les éventuelles actions sur le système. Ci-dessous, nous détaillons la deuxième étape du processus de modélisation en l'adaptant aux problèmes d'optimisation.

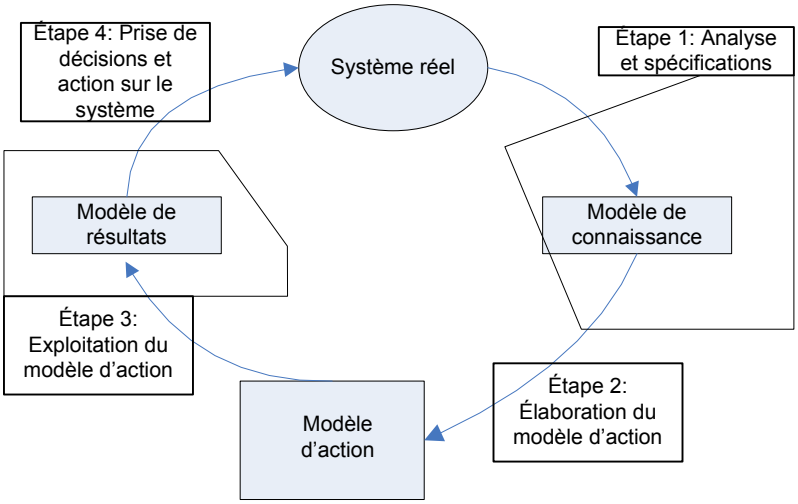

Figure 1-10. Processus de modélisation simplifié

6.3 Processus de modélisation pour l'optimisation

Le processus de modélisation présenté dans le paragraphe précédent est général, nous présentons ci-après (figure 1-11) son adaptation aux problèmes d'optimisation. Il précise plus particulièrement le rôle des hypothèses simplificatrices et des axes d'amélioration des solutions.

Comme nous l'avons vu dans le paragraphe 4, le modèle d'optimisation se compose de deux parties : le module d'évaluation des performances et le module d'optimisation. Le module d'évaluation des performances est utilisé pour évaluer la qualité d'une solution proposée par un algorithme d'optimisation. Ce module peut prendre, suivant la complexité du problème, l'une des formes suivantes : un modèle de simulation, une procédure d'évaluation ou une fonction d'évaluation. En retour, le module d'optimisation des performances utilise l'évaluation pour éventuellement proposer de nouvelles solutions. Cette communication est symbolisée par une double flèche entre les deux modules.

Pour élaborer le modèle d'action, on a deux possibilités : soit le modèle de connaissance est directement traduit en un modèle de simulation, soit il est dérivé en modèles d'évaluation des performances. La traduction consiste en la réécriture du modèle de connaissance en un modèle de simulation dans un langage approprié. La dérivation consiste à traduire le modèle de connaissance en lui adjoignant des hypothèses simplificatrices. Les hypothèses simplificatrices définissent le degré de finesse du module d'évaluation des performances, c'est-à-dire qu'elles définissent quelles sont les fonctionnalités du système qui seront prises en compte et quelles sont celles qui seront ignorées. Les hypothèses réalisées sont donc relatives au modèle de connaissance obtenu à l'étape précédente. Il est

donc essentiel de construire le modèle de connaissance le plus fin et le plus complet possible relativement aux objectifs.

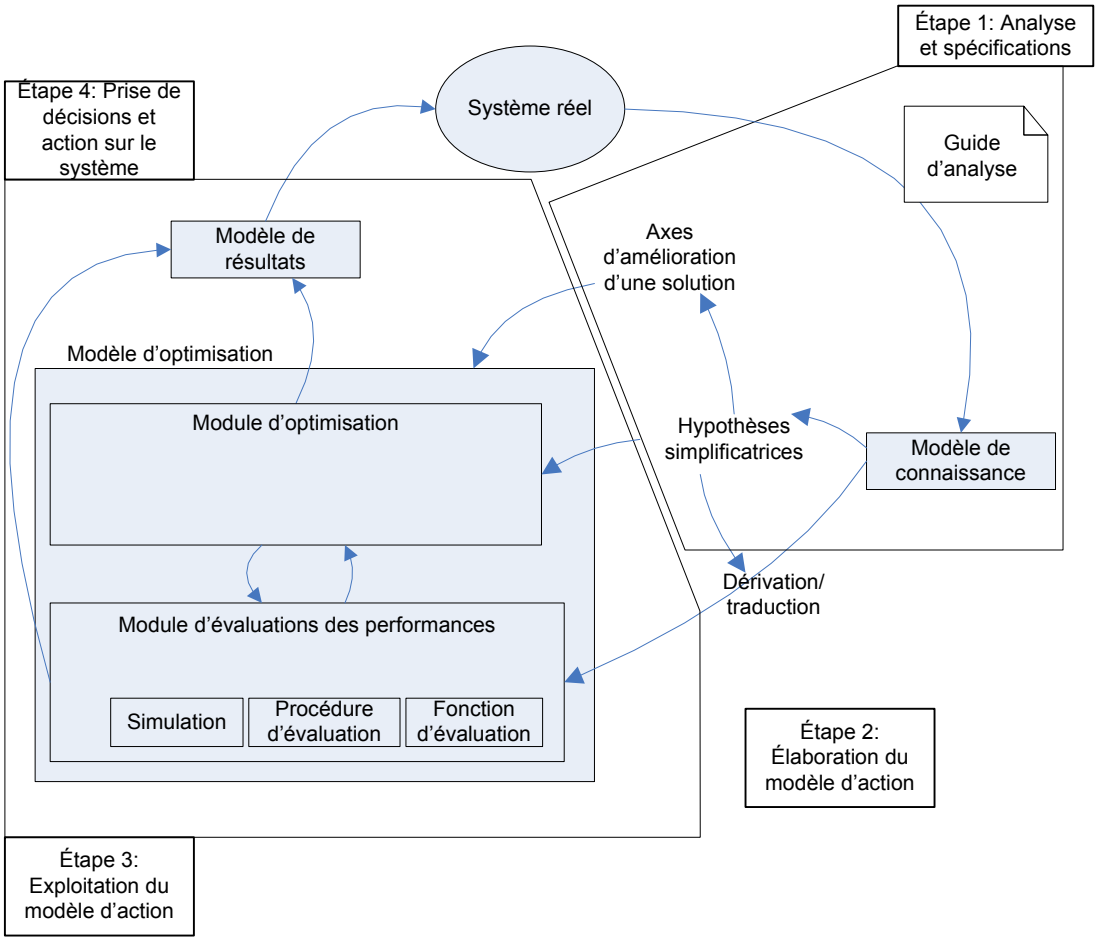

Figure 1-11. Processus de modélisation adapté aux problèmes d'optimisation

Dans le contexte de l'optimisation, il est aussi nécessaire de formuler des hypothèses simplificatrices relativement à la méthode d'optimisation choisie. En effet, certaines méthodes et certaines approches d'optimisation nécessitent des hypothèses particulières : la programmation linéaire ne permet pas de prendre en compte des événements aléatoires, il est difficile d'utiliser les métaheuristiques quand il y a de nombreuses décisions à prendre, etc. ...

De plus, pour permettre la construction du modèle d'optimisation, il est nécessaire de définir les modifications possibles que l'on peut apporter à une solution pour l'améliorer. On appelle "axes d'amélioration d'une solution" ces modifications. Les axes d'amélioration dépendent largement des hypothèses simplificatrices réalisées. Considérons un système de transport modélisé uniquement par le temps de transport. Dans cet exemple, le planning du transport découle directement du planning des opérations sur les machines et il n'est donc pas nécessaire d'envisager des améliorations sur les dates de début des opérations transport. Les axes d'amélioration peuvent aussi être limités par une contrainte : considérons par exemple un système dans lequel un ensemble de jobs passent sur une ressource. Suivant que les hypothèses simplificatrices modélisent ou non des contraintes de précédence entre jobs, les axes d'amélioration peuvent ou non intégrer ces contraintes.

La démarche de la figure 1-11 présente comment construire un modèle d'optimisation. Nous préconisons de réaliser plusieurs modèles d'optimisation en partant du modèle le plus simple vers le plus complexe. Notre démarche consiste donc à réaliser des modèles d'action de complexité croissante, on l'appelle donc "démarche itérative pour l'optimisation" (figure 1-12).

On utilise itérativement le processus de modélisation pour le développement des modules d'optimisation et d'évaluation des performances. A la première itération, nous préconisons de réaliser un grand nombre d'hypothèses simplificatrices, de manière à obtenir des modules d'optimisation et d'évaluation des performances les plus simples possibles. Aux itérations suivantes, les modèles sont raffinés en diminuant le nombre d'hypothèses simplificatrices. À chaque itération, les axes d'amélioration d'une solution doivent aussi être raffinés et les deux modules doivent être complétés de manière à prendre en compte les nouvelles hypothèses. Il y a donc une relation "étends" entre les modules de chaque itération et une relation "inclus" entre les hypothèses simplificatrices et axes d'amélioration de chaque itération. La relation "étends" signifie que de nombreuses fonctionnalités et concepts utilisés à l'itération précédente vont pouvoir être réutilisés et adaptés à l'itération suivante.

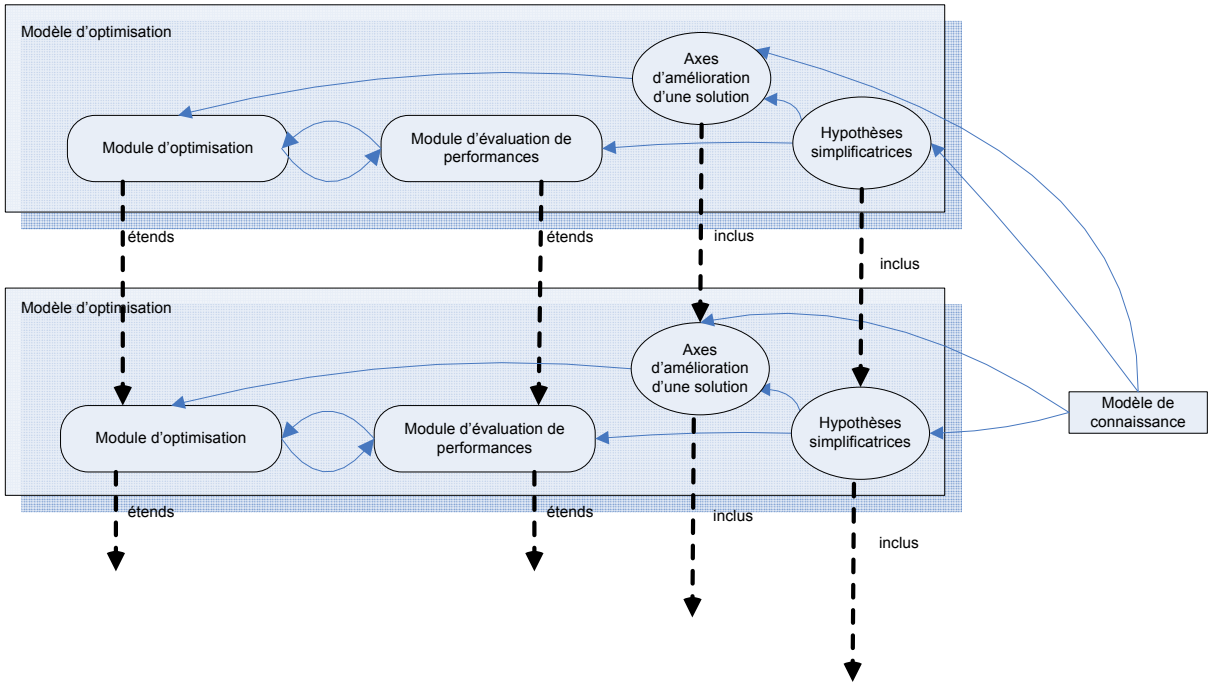

Figure 1-12. Démarche itérative pour l'optimisation

6.4 Modèles proposés

Quel que soit le formalisme retenu, le processus décrit ci-dessus permet de créer des modèles d'action. Dans ce paragraphe, nous présentons les trois modèles mis en œuvre pour les problèmes de jobshop avec time lags et jobshop avec transport et contraintes additionnelles. Pour la résolution de ces deux problèmes, nous avons envisagé un modèle mathématique, un modèle de graphe conjonctif disjonctif et un modèle de simulation à événements discrets.

Ces trois modèles sont quasiment équivalents, c'est-à-dire que presque toutes les contraintes exprimées dans un modèle peuvent être reformulées dans un autre. Pour autant, la difficulté de prise en compte de certaines contraintes dépend fortement du formalisme utilisé pour décrire le modèle. Autrement dit, il y a des contraintes qui s'écrivent très simplement en formalisation mathématique et qui sont plus difficiles à modéliser en simulation.

L'intérêt de disposer de trois modèles différents réside dans les méthodes d'optimisation que l'on peut mettre en œuvre. En effet, à chaque type de modèle est associé un ensemble de méthodes possibles. Par exemple, les méthodes de programmation linéaire en nombres entiers sont plus adaptées aux modèles mathématiques.

6.4.1 Modèle mathématique

Lorsqu'un modèle de connaissance est précisément connu, on peut le décrire dans un formalisme mathématique (appelé dans la suite "modèle mathématique") qui a l'avantage d'être relativement facile à écrire, d'être non ambigu et de pouvoir être traduit en un modèle linéaire facile à mettre en œuvre.

Il ne faut pas confondre un modèle mathématique dont les contraintes sont écrites de manière formelle à l'aide de notations mathématiques et un modèle linéaire qui permet d'exprimer un problème sous forme de contraintes linéaires. Bien sûr, les modèles mathématiques contiennent des contraintes linéaires, mais certaines contraintes du formalisme mathématique ne sont pas linéaires, comme : une contrainte faisant intervenir des termes quadratiques ou une contrainte dont l'expression des indices utilisent des variables. Un modèle dans un formalisme mathématique et sa traduction en un modèle linéaire sont présentés dans le chapitre 4 pour le problème de jobshop avec transport et contraintes additionnelles.

Il est souvent possible de passer d'un modèle mathématique à un modèle linéaire. Ce dernier est particulièrement utile pour :

- vérifier la solution optimale,
- valider la solution optimale,
- valider des méthodes approchées.

La vérification et la validation permettent de détecter si des contraintes ont été oubliées ou si d'autres doivent être supprimées. La vérification se fait par l'étude du modèle lui-même, elle montre que le modèle est conforme à ce qu'on a initialement prévu et que l'on n'a pas introduit d'erreurs pendant la réalisation du modèle. Considérons le cas où des contraintes ont été oubliées, celles-ci ont pu être omises soit au niveau des phases d'analyse et de spécification, soit au niveau de leur traduction dans un formalisme mathématique. Dans tous les cas, le modèle obtenu ne contient pas assez de contraintes et les solutions optimales trouvées sont donc à priori meilleures que les solutions recherchées. La vérification peut consister, par exemple, à générer des solutions optimales pour des problèmes simples. Les solutions trouvées permettent de mettre en évidence certains problèmes de modélisation : les solutions générées respectent les contraintes, mais sont manifestement fausses ou sous optimales. Une solution manifestement fausse respecte toutes les contraintes formalisées, mais n'est pas une solution du problème. C'est ce qui arrive quand des contraintes manquent dans la formalisation. Une solution manifestement sous optimale est une solution respectant toutes les contraintes, mais dont la qualité est inférieure à des solutions connues. C'est ce qui arrive quand certaines contraintes sont trop restrictives, quand par exemple certains cas limites ont été oubliés.

La validation d'une solution consiste à montrer que le modèle est conforme aux objectifs que l'on s'était fixés. Ainsi, on vérifie si le modèle est conforme aux exigences initiales. Deux types de validation d'une solution optimale peuvent être envisagées : la validation ad-hoc et la validation par comparaison au système réel. La validation ad-hoc consiste à modifier une solution pour vérifier si la solution optimale évolue dans le sens attendu. Par exemple, en augmentant la capacité du système la charge doit être écoulée plus rapidement. La validation par comparaison au système réel est généralement réalisée après une validation ad-hoc. Elle consiste à confronter la solution au système réel et à la faire valider par les experts. Ainsi, on peut rapidement mettre en évidence les lacunes du modèle. Le fait que les problèmes traités par résolution directe soient de taille réduite est plutôt un atout car un petit exemple est plus facile à appréhender et à discuter.

Pour finir, les solutions obtenues grâce à un modèle linéaire permettent de valider les méthodes approchées, construites par la suite. En effet, un modèle linéaire permet d'obtenir des solutions exactes sur des petites instances et éventuellement de bonnes solutions issues de résolutions optimales tronquées. Ces solutions sont utiles pour valider les méthodes approchées, car celles-ci sont supposées trouver les solutions optimales sur les petites instances ou tout du moins des solutions de très bonne qualité.

La formalisation mathématique a donc un double intérêt : formaliser de manière univoque la connaissance du système, et permettre la résolution optimale, par des solveurs génériques, de problèmes de petite taille.

6.4.2 Graphe conjonctif - disjonctif

Le deuxième modèle que nous préconisons est le graphe conjonctif - disjonctif (Roy, 1964). Ce modèle est spécialement adapté aux problèmes d'ordonnancement à ressources disjonctives. Un des principaux avantages de ce modèle est qu'il transforme un problème d'ordonnancement (i.e. problème de recherche de dates de début des opérations) en un problème de recherche de séquences. En effet, pour un problème modélisé sous la forme d'un graphe conjonctif-disjonctif et dont le critère est régulier, il est possible de chercher le meilleur ordonnancement respectant la séquence (i.e. l'ordonnancement le plus calé à gauche). Les problèmes de recherche de séquence sont des problèmes combinatoires. Et les algorithmes d'optimisation combinatoire sont plus adaptés à la recherche d'une séquence qu'à la recherche de dates de début des opérations.

Plus précisément, le graphe conjonctif - disjonctif consiste en un graphe dont les nœuds représentent des opérations, il consiste aussi en des arcs et des arêtes qui modélisent les contraintes temporelles entre opérations. Ces contraintes sont de la forme $t_i + \alpha_{ij} \leq t_j$ où α_{ij} est une constante de temps et où t_i (resp. t_j) est la date de début de l'opération i (resp. j). Le modèle prend en compte toutes les contraintes de la forme $t_i + \alpha_{ij} \leq t_j$, que les opérations i et j soient connues à l'avance ou qu'elles dépendent de l'ordre des opérations envisagé. Ainsi le modèle de graphe est divisé en deux modèles distincts : le graphe conjonctif-disjonctif qui modélise le problème en général et qui contient les contraintes indépendantes de l'ordre des opérations et le graphe conjonctif qui modélise toutes les contraintes pour un ordre donné des opérations.

Pour un ordre donné des opérations, le graphe conjonctif contient toutes les contraintes relativement à l'ordre fixé des opérations. Ainsi, toutes les contraintes du type $t_i + \alpha_{ij} \leq t_j$ sont exprimées par un arc de i vers j de longueur α_{ij}. Il ne reste alors plus qu'à rechercher les dates de début des opérations. Or, un calcul de plus long chemin dans le graphe fournit des marques pour chacun des nœuds. On peut montrer que la marque d'un nœud correspond à la date au plus tôt de l'opération telle que toutes les contraintes du problème soient respectées.

Le modèle de graphe conjonctif - disjonctif permet de modéliser tous les problèmes disjonctifs. Pourtant, il a été principalement étudié pour les problèmes de flowshop, jobshop et RCPSP. Dans les chapitres suivants, nous proposons de formaliser les problèmes étudiés à l'aide de ce graphe.

6.4.3 Simulation à événements discrets

Le troisième modèle que nous préconisons est un modèle de simulation. La simulation est un outil d'analyse très puissant utilisé pour la conception et pour la mise en œuvre de processus de systèmes complexes. Elle doit être vue comme une approche, une technique, permettant de construire une abstraction de la réalité et de la faire évoluer en fonction du temps. L'abstraction construite est un modèle du système étudié et on parle de "modèle de simulation". L'objectif est de reproduire les activités des différentes entités du système simulé et donc d'apprendre quelque chose d'utile sur ses comportements et ses performances. Contrairement aux méthodes empiriques, la simulation fournit une réponse scientifique et étayée à des problèmes difficiles. Longtemps perçue comme la méthode de la dernière chance, elle peut aujourd'hui être vue comme une méthodologie.

Pour effectuer des simulations, on peut utiliser des environnements de simulation dont le but est d'aider à la mise en œuvre de modèles grâce à :

- des langages qui fournissent les primitives nécessaires à l'écriture du modèle de simulation,
- des composants réutilisables pour réduire le temps de développement,
- des outils statistiques générant automatiquement ou permettant de générer les résultats sous forme de rapports de simulation.

Par exemple, les environnements de simulation SIMAN Arena ou Witness sont largement reconnus.

Un modèle de simulation "ne prend pas de décision", il ne fait que répondre à une question du type "que se passe-t-il si ?". Ainsi, toute optimisation à l'aide d'un modèle de simulation passe par le couplage avec une autre méthode. On peut utiliser par exemple : une approche intuitive qui consiste à tester manuellement des hypothèses, ou une approche plus systématique utilisant les plans d'expériences. Ces plans ont pour but de déterminer quels sont les paramètres influents et comment les explorer. La méthode Tagushi (1987) aide à la construction de plans d'expériences, mais ces plans ne sont efficaces que pour un nombre relativement restreint de configurations à explorer. Lorsque ce nombre devient trop important, il est nécessaire d'avoir recours à d'autres techniques. Dans le paragraphe suivant, nous présentons les principales méthodes d'optimisation. Ces méthodes peuvent être utilisées pour déterminer le meilleur jeu de paramètres.

7 Méthodes de résolution

Les problèmes d'optimisation et les méthodes permettant de les résoudre sont nombreux. Nous ne présentons donc, dans ce paragraphe, que les principaux problèmes d'optimisation et les principales méthodes. Pour cela, nous présentons les problèmes d'optimisation combinatoire en général (§7.1). Puis, nous présentons les grandes familles de méthodes (§7.2). Les métaheuristiques (§7.3) forment une famille de méthodes à laquelle on s'intéresse plus particulièrement dans la suite de cette thèse. Parmi toutes ces méthodes, le choix de la méthode à mettre en œuvre n'est pas trivial. Pour pouvoir réaliser ce choix, il est important de pouvoir comparer ces algorithmes. Le paragraphe 7.4 fournit des outils et des résultats théoriques intéressants pour l'évaluation de méthodes de résolution.

7.1 Introduction

Définition : problème d'optimisation combinatoire

Soit P un problème, soit Ω l'ensemble discret et fini des solutions du problème P, et c une fonction univoque appelée "fonction coût".

Trouver $x^* \in \Omega$ tel que $c(x^*) = \min_{x\in\Omega} c(x)$ et $x^* \in \Omega$ tel que $c(x^*) = \max_{x\in\Omega} c(x)$ sont deux problèmes d'optimisation combinatoire.

Un des éléments les plus importants dans cette définition est la nature discrète des problèmes d'optimisation combinatoire. Les problèmes auxquels on s'intéresse sont en général modélisés de manière discrète et sont donc naturellement des problèmes d'optimisation combinatoire.

Un problème d'optimisation combinatoire peut être un problème de minimisation ou un problème de maximisation. Dans la suite, et pour simplifier les explications, nous ne considérons que les problèmes de minimisation car on peut toujours transformer un problème de maximisation en un problème de minimisation. Toutes les remarques et définitions restent valables dans le cas contraire.

Cette définition doit être complétée par les remarques suivantes : dans le cas d'une maximisation, la fonction coût est généralement appelée "fonction profit". De plus, le terme "solution" utilisé dans cette définition est légèrement différent de son acceptation habituelle. Une solution en optimisation combinatoire est un point de l'espace de recherche. Suivant les problèmes considérés, il se peut que tous les points de l'espace de recherche ne correspondent pas à des solutions du problème de départ. Autrement dit, lors du développement du problème d'optimisation combinatoire, il est possible que l'on soit contraint ou que l'on préfère introduire dans l'espace de recherche des points qui ne représentent pas des solutions. Pour reconnaître ces points, on utilise les notions de "solution réalisable" / "solution irréalisable".

Définition : solution réalisable / irréalisable

Soit *D* le problème de départ et *P* un problème d'optimisation combinatoire modélisant le problème *D*. On dit qu'une solution de *P* est réalisable si elle est solution du problème de départ. Dans le cas contraire, on dit que la solution est irréalisable.

La présence des solutions irréalisables dans l'espace de recherche augmente fortement sa taille, mais il est quelquefois préférable d'avoir un espace plus grand mais plus adapté à la recherche. La figure 1-13 illustre cette remarque, elle comporte des cercles représentant des solutions réalisables. L'éloignement entre ces cercles figure la difficulté qu'un algorithme d'optimisation aura pour passer d'un paquet de solutions réalisables à un autre. Les tubes reliant les cercles figurent les solutions irréalisables introduites. Grâce à eux, les solutions réalisables sont plus proches et l'optimisation est simplifiée.

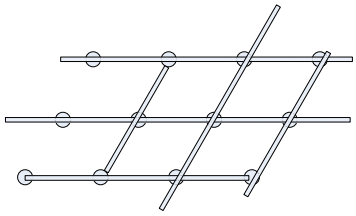

Figure 1-13. Solutions irréalisables introduites pour faciliter la recherche

La présence de solutions irréalisables ne doit pas perturber la recherche de la meilleure solution. Il est donc important de faire en sorte que la solution proposée par l'algorithme d'optimisation combinatoire ne soit pas une solution irréalisable. Pour cela, on dénombre trois cas de figure :

- Soit les solutions irréalisables sont complètement exclues de l'espace de recherche : à chaque fois qu'un algorithme générera une solution irréalisable, il stoppera l'exploration et cette solution ne sera pas envisagée. Suivant les cas, la solution peut être évitée à priori (solutions irréalisables non générées) ou non (solutions irréalisables générées puis refusées).
- Soit les solutions irréalisables sont exclues par leur coût. Un coût prohibitif est affecté aux solutions irréalisables, on dit dans ce cas que le coût d'une solution irréalisable est infini. En pratique, c'est un nombre largement supérieur aux coûts de toutes les solutions réalisables.
- Soit les solutions irréalisables sont partiellement exclues par leur coût. Dans ce cas, il faut choisir le coût des solutions irréalisables de manière à ce que la meilleure solution irréalisable soit forcément de coût supérieur au minimum recherché. Dans le cas des heuristiques, l'algorithme peut trouver une solution qui n'est pas l'optimum. Il faut donc veiller à ce que l'algorithme renvoie tout de même une solution réalisable.

Les problèmes d'optimisation combinatoire considérés ci-dessus doivent être résolus par des algorithmes. Un algorithme énonce une résolution sous la forme d'une série finie d'opérations à effectuer. La mise en œuvre de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation dans le but d'être exécuté par un ordinateur. Pour classifier les algorithmes, il est nécessaire de disposer d'une modélisation d'un ordinateur : la machine de Turing. On distingue deux machines de Turing différentes : la machine de Turing déterministe et la non déterministe. La machine de Turing déterministe est celle qui modélise les ordinateurs actuels. A chaque étape, une machine de Turing déterministe désigne la prochaine étape à réaliser. Par opposition, la machine de Turing non déterministe associe à chaque étape un certain nombre d'étapes suivantes possibles. La machine de Turing non déterministe ne permet pas de choisir quelle sera la prochaine étape.

Définition : Problème P

Un problème est dit dans *P* (Polynomial time) s'il peut être résolu en un temps polynomial par une machine de Turing déterministe.

Un problème *P* est donc un problème que l'on peut résoudre à l'aide d'un algorithme polynomial (par un ordinateur). Typiquement, la recherche d'un élément dans une liste et le tri d'une liste de valeurs sont des problèmes *P*.

Définition : Problème NP

Un problème est dit dans *NP* (Nondeterministic Polynomial time) s'il peut être résolu en un temps polynomial par une machine de Turing non déterministe.

Les définitions de *P* et *NP* ne sont pas exclusives. En particulier, les problèmes dans *P* sont des aussi dans *NP*, car un algorithme qui est en temps polynomial sur une machine de Turing déterministe et aussi en temps polynomial sur une machine de Turing non déterministe. On a donc $P \subseteq NP$.

Par contre, les problèmes dans *NP* ne sont pas forcément dans *P*. En tout cas, il n'a jamais pu être prouvé que les ensembles des problèmes *P* et *NP* sont distincts ou égaux. Aujourd'hui, il est communément admis que $P \neq NP$, mais cette inégalité n'est pour l'instant qu'une conjecture.

Une réduction polynomiale d'un problème P1 en un problème P2 consiste à transformer les données du problème P1 en données du problème P2. Cette transformation doit conserver toutes les caractéristiques du problème et ne doit pas créer une instance de taille exponentielle du problème. Ainsi, toute personne disposant d'un algorithme polynomial pour P2 est capable de résoudre polynomialement tous les problèmes P1. C'est à l'aide de la notion de réduction polynomiale que l'on définit la classe des problèmes *NP-complets*.

Définition : Problème NP-complet

Un problème est dit *NP*-complet s'il est dans *NP* et si tout problème de *NP* peut être polynomialement réduit en ce problème.

Les problèmes *NP-complets* sont donc des problèmes centraux pour la résolution des problèmes dans NP. Car toute personne qui trouverait un algorithme polynomial sur une machine de Turing déterministe pour un problème *NP-complet* pourrait adapter cet algorithme pour qu'il résolve en temps polynomial tous les problèmes dans *NP*. Si l'on admet que $P \neq NP$, cela signifie que l'on ne peut pas trouver d'algorithme polynomial sur une machine de Turing déterministe pour résoudre les problèmes *NP-complets*.

Pour un problème *NP-complet*, toute résolution exacte passe donc par l'utilisation d'un algorithme polynomial sur une machine de Turing non déterministe. Il est possible d'adapter cet algorithme pour qu'il fonctionne sur un ordinateur actuel, mais cette adaptation fait qu'il consomme des ressources (temps et espace mémoire) de manière non polynomiale. On qualifie ce type d'algorithmes d'"exponentiels".

Pour illustrer ce qu'est un algorithme exponentiel, nous considérons l'exemple suivant récapitulé dans le tableau 1-2. Soit un problème d'optimisation combinatoire dont la taille est caractérisé par le paramètre *n*. Un algorithme est proposé dont la complexité est *n !*, c'est-à-dire que le nombre d'itérations qu'il réalise est en *n !*. On pose qu'une itération a une durée constante de 1 Million d'Instructions. Un ordinateur cadencé à 1 Million d'Instructions Par Secondes (MIPS) mettra donc une seconde à exécuter une itération. Dans le tableau 1-2, figurent en colonne différentes valeurs de *n* représentant différentes tailles de problèmes. Sur chaque ligne du tableau figure une date et le nombre de MIPS atteint à cette date. Ce chiffre fournit une approximation de l'évolution de la puissance des ordinateurs depuis 1972 (Source Intel). Les cases faiblement grisées représentent un temps de résolution inférieur à la minute. Les cases fortement grisées représentent un temps de résolution

supérieur à la journée. Pour de nombreux problèmes, le temps de résolution acceptable se situe entre ces deux limites et correspond donc aux cases blanches du tableau. Les deux dernières lignes du tableau sont des projections dans l'avenir. La ligne 2007 correspond aux microprocesseurs en cours de développement chez Intel et dont la commercialisation est prévue pour 2007. La ligne 2100 est une extrapolation grossière qui prévoit que l'évolution des microprocesseurs dans les 100 prochaines années sera la même que celle des 30 dernières années. Pour atteindre ces prévisions, les experts s'accordent à dire que la technologie actuelle ne sera pas suffisante et qu'un saut technologique devra avoir lieu.

		5	6	7	8	9	10	11	12	13	14	15	50	100
	n!	1E+2	7E+2	5E+3	4E+4	4E+5	4E+6	4E+7	5E+8	6E+9	9E+10	1E+12	3E+64	9E+157
Date	**MIPS**													
1972	0,06	2E+3	1E+4	8E+4	7E+5	6E+6	6E+7	7E+8	8E+9	1E+11	1E+12	2E+13	5E+65	2E+159
1982	0,9	1E+2	8E+2	6E+3	4E+4	4E+5	4E+6	4E+7	5E+8	7E+9	1E+11	1E+12	3E+64	1E+158
1992	50	2E+0	1E+1	1E+2	8E+2	7E+3	7E+4	8E+5	1E+7	1E+8	2E+9	3E+10	6E+62	2E+156
2002	2500	5E-2	3E-1	2E+0	2E+1	1E+2	1E+3	2E+4	2E+5	2E+6	3E+7	5E+8	1E+61	4E+154
2007	125000	1E-3	6E-3	4E-2	3E-1	3E+0	3E+1	3E+2	4E+3	5E+4	7E+5	1E+7	2E+59	7E+152
2100	5000000	2E-5	1E-4	1E-3	8E-3	7E-2	7E-1	8E+0	1E+2	1E+3	2E+4	3E+5	6E+57	2E+151

Tableau 1-2. Temps de résolution à l'aide d'algorithmes exponentiels

Il apparaît clairement dans ce tableau que l'évolution des ordinateurs a permis la résolution de problèmes sensiblement plus grands que ceux qui pouvaient être résolus en 1972. Il apparaît aussi que cette évolution n'est pas prête de permettre la résolution de problèmes beaucoup plus grands (problèmes de taille 50 et 100).

Une caractéristique importante des algorithmes exponentiels est qu'ils sont très sensibles à la taille des problèmes résolus. Ainsi, en 2007, on peut résoudre un problème de taille 12 en 4000 secondes. Dans un contexte d'application réelle, il est tout à fait envisageable que la taille du problème change d'une exécution sur l'autre. Or passer à une taille de problème de 13 augmente le temps de résolution à 50000 secondes (12,5 fois plus de temps). Ce cas de figure n'est en général pas envisageable pour une application réelle.

En conséquence, les algorithmes exponentiels peuvent être utilisés pour résoudre des instances suffisamment petites. Pour des applications réelles, il est nécessaire de vérifier que la taille des problèmes est largement inférieure aux capacités de l'ordinateur. Ainsi, on peut garantir un temps de calcul constant aux utilisateurs.

Pour les problèmes de taille supérieure, il n'est pas possible de fournir un algorithme exact. On s'intéresse alors naturellement à des algorithmes non exacts appelés "heuristiques". D'après Reeves 93 :

> *A heuristic is a technique which seeks good(i.e. near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is.*

> *Une heuristique est une technique de recherche de bonnes solutions (i.e. quasi-optimale) obtenue par un effort de calcul raisonnable sans que l'on soit capable de garantir la faisabilité ni l'optimalité ; et même, dans beaucoup de cas, sans être capable de quantifier l'écart d'une solution réalisable à une solution optimale.*

Le paragraphe suivant donne une typologie des méthodes d'optimisation incluant les méthodes exactes et les méthodes heuristiques.

7.2 Familles de méthodes d'optimisation

Les principales méthodes d'optimisation sont décrites ci-dessous, la liste n'est pas exhaustive et n'a d'autre intérêt que de montrer le panel relativement large des méthodes d'optimisation combinatoire. Les méthodes proposées sont :

- Programmation linéaire
- Programmation dynamique
- Procédures de séparation / évaluation
- Méthodes d'énumération partielles
- Algorithmes constructifs
- Méthodes de recherche locale

Programmation Linéaire

Il y a trois grands types de programmes linéaires : les programmes linéaires simples, les programmes linéaires en nombres entiers, et les programmes linéaires mixtes. Tous ces programmes sont des programmes dit linéaires, ils sont donc composés de contraintes sous forme d'équation (signe =) ou d'inéquation (signes ≤ ou ≥) faisant intervenir des variables de manière linéaire.

Les programmes linéaires simples ont des variables dites "continues". Ces programmes linéaires peuvent être efficacement résolus par la méthode du simplexe ou la méthode du point intérieur. La méthode du point intérieur est polynomiale, la méthode du simplexe est exponentielle dans le pire des cas. En pratique, la méthode du simplexe est plus performante et plus répandue dans les outils de résolution.

Les programmes linéaires en nombres entiers font intervenir des variables entières. Ces programmes peuvent paraître plus simples que les précédents dans la mesure où le nombre de solutions possibles est fini. Pourtant, c'est ce type de problème qui est le plus difficile à résoudre. D'ailleurs certains programmes linéaires en nombres entiers sont reformulés (sous certaines réserves et sous certaines conditions) en programmes linéaires simples.

Les programmes linéaires mixtes comportent à la fois des variables continues et des variables en nombres entiers. D'un point de vue résolution, ces programmes linéaires ressemblent fortement au programme linéaire en nombre entiers et on ne retient en général que la taille du sous problème en nombre entiers pour mesurer la taille du problème mixte.

De nombreux solveurs génériques sont disponibles dans le commerce, et la résolution du programme linéaire peut être faite de manière relativement efficace. Pourtant, toutes les tentatives de résolution de problèmes disjonctifs (problèmes apparaissant en planification) à l'aide de ce type de solveurs ont été décevantes (Jain et Meeran, 1999).

Programmation Dynamique

Ces méthodes tirent partie des propriétés de décomposition du problème. Ce type de méthode peut être utilisé quand on peut décomposer le problème comme suit : la solution d'un problème *P* de taille *n* peut être construite à partir de la solution d'un sous-problème de taille *n-1*, *n-2*, *n-3* ... Une des méthodes de programmation dynamique les plus connues est l'algorithme de Bellmann pour le calcul des chemins de longueurs optimales dans un graphe.

Les procédures de séparation évaluation (P.S.E.)

Ces méthodes réalisent une énumération implicite de toutes les solutions : elles consistent à trouver une solution optimale sans parcourir toutes les solutions du problème (Tercinet, 2004). Les PSE peuvent être implantées pour minimiser ou maximiser un critère. Dans la suite du paragraphe, on ne considère que le cas de la minimisation. Pour y parvenir, deux fonctionnalités sont introduites : la séparation et l'évaluation. La séparation consiste à partitionner un problème en deux ou plusieurs sous-problèmes. Les sous-problèmes engendrés sont des problèmes plus faciles à résoudre et dont la donnée des solutions permet de résoudre le problème principal. Les séparations successives réalisées durant la

procédure créent un arbre appelé "arbre de recherche" (cf. figure 1-14). L'évaluation fournit pour chaque nœud de l'arbre une valeur appelée "borne inférieure". Tous les nœuds sous le nœud évalué (et donc tous les sous-problèmes du problème considéré) sont des solutions de coût supérieur à la borne inférieure. C'est grâce à cette propriété que l'on peut éviter de parcourir certaines solutions de l'arbre (i.e. faire une énumération implicite plutôt qu'explicite). En effet, si une solution a déjà été trouvée, tout nœud dont la borne inférieure est supérieure à la solution trouvée n'a pas besoin d'être exploré, on dit que ce nœud et tout son sous-arbre ont été coupés. La figure 1-14 présente un exemple de résolution par séparation / évaluation. Au moment où l'arbre est dessiné, l'algorithme a déjà trouvé une solution S1 de borne inférieure 101 et de coût 103. Le nœud courant dans l'algorithme est le nœud N1 dont la borne inférieure est 110. Ainsi, on sait que tous les nœuds du sous-arbre N1 ont un coût au moins égal à 110. Or une solution à 103 a déjà été trouvée, il n'est donc pas nécessaire de parcourir le sous-arbre de N1. Même si le nombre de solutions coupées peut être très grand, il est clair que les algorithmes ainsi générés sont tout de même exponentiels (Blazewicz, 1993). La figure 1-14 présente une illustration de méthode de séparation / évaluation. Chaque nœud est représenté par un heptagone. La solution *S1* dont le critère vaut *101* a été trouvée. Le nœud *N1* a une borne inférieure à *110*. Le sous-arbre de *N1* n'a donc pas besoin d'être exploré.

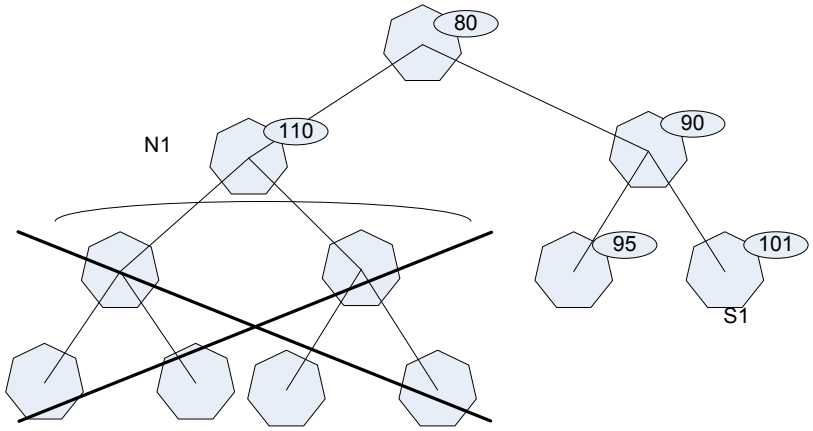

Figure 1-14. Méthode de séparation / évaluation

Les méthodes ci-après sont toutes des méthodes heuristiques (l'optimalité de la solution trouvée n'est pas garantie).

Les méthodes d'énumération partielle

Ces méthodes sont basées essentiellement sur une approche arborescente. Elles ressemblent beaucoup aux procédures de séparation évaluation mais l'optimalité de la solution trouvée n'est pas prouvée. Ces méthodes mélangent les connaissances utilisées en séparation/évaluation pour éviter d'énumérer toutes les solutions avec des idées heuristiques de résolution. On peut citer par exemple les méthodes par machine goulot.

Les algorithmes constructifs

Ils procèdent par construction progressive de la solution : à chaque étape, une nouvelle partie de la solution est construite. Ces algorithmes sont à la base de nombreuses heuristiques dédiées. Certains algorithmes constructifs sont optimaux, citons par exemple le problème de sous-arbre de poids minimum pour lequel l'algorithme de Kruskal offre une solution optimale.

De nombreuses familles d'algorithmes constructifs existent. Citons les algorithmes gloutons, les méthodes de listes, ...

Dans les chapitres suivants, nous présentons des algorithmes de liste.

Les méthodes de recherche locale

Ces méthodes consistent à définir autour de toute solution *S* un voisinage *V(S)* contenant des solutions que l'on considère comme proches de *S*. Ce type de méthode part d'une solution initiale et, de proche en proche, passant d'une solution voisine à une autre, tend à obtenir des solutions de meilleure qualité. Parmi les méthodes de recherche locale on distingue deux grandes catégories : les méthodes à individu qui considèrent une unique solution et la font évoluer, et les méthodes à population qui utilisent un ensemble de solutions.

Dans le paragraphe suivant, nous détaillons les principales métaheuristiques. Ces méthodes d'optimisation illustrent parfaitement les principales méthodes de recherche locale existantes.

7.3 Les métaheuristiques

Après quelques années passées à développer des heuristiques dédiées à des problèmes particuliers, il est apparu que certains schémas pouvaient être réutilisés pour de nombreux problèmes d'optimisation combinatoire. Ces méthodes, appelées "métaheuristiques", forment une famille de méthodes d'optimisation combinatoire qui ont la particularité d'être adaptables à différents types de problèmes. Le recuit simulé, l'algorithme tabou et les algorithmes génétiques sont sûrement les métaheuristiques les plus connues. La littérature fournit régulièrement de nouvelles métaheuristiques, mais nous n'en présentons dans ce paragraphe qu'un nombre restreint.

7.3.1 Méthodes de recherche locale

La plupart des métaheuristiques sont des algorithmes de recherche locale. Ce paragraphe introduit le vocabulaire nécessaire à la description des métaheuristiques. Une méthode de recherche locale est une méthode utilisant la notion de voisinage.

Définition : voisinage

Soit *P* un problème d'optimisation combinatoire et Ω son ensemble de solutions.

Un voisinage V est une fonction associant à chaque solution un ensemble de solutions considérées "proches". V est donc une fonction de Ω dans $P(\Omega)$.

On appelle "combinatoire du voisinage V" et on note $|V|$ le nombre de voisins moyenné sur Ω.

Remarque : La combinatoire d'un voisinage peut rarement être calculée de manière exacte.

Les méthodes de recherche locale appliquent les voisinages de manière itérative. La principale différence entre les différentes méthodes de recherche locale est la manière dont les solutions auxquelles on applique le voisinage sont choisies. Quelle que soit la méthode retenue, on part toujours d'une solution que l'on améliore (l'amélioration peut ne pas être immédiate) itérativement à l'aide du voisinage. Grâce au voisinage V on transforme donc une solution x en une solution $y \in V(x)$. Par abus de notation, on note quelquefois $V(x)$ un point obtenu par application du voisinage V sur x. On peut donc noter, $y = V^6(x) = V(V(V(V(V(x)))))$. La figure 1-15 illustre le passage de x à y. Sur cette figure, les voisinages successifs s'enchevêtrent : les voisinages $V(x), V^2(x) et V^3(x)$ ont une partie en commun, il aurait donc été possible de passer directement de x à $V^3(x)$ sans passer par $V^2(x)$ et $V^3(x)$. Ceci illustre le fait qu'une méthode de recherche locale est généralement heuristique et que les voisins qu'elle choisit peuvent ne pas être les meilleurs possibles.

A force d'itérations successives, et puisque l'on cherche à obtenir des solutions de qualité croissante, il est courant d'obtenir une solution dont tous les voisins sont de qualité inférieure : c'est ce qu'on appelle un "optimum local".

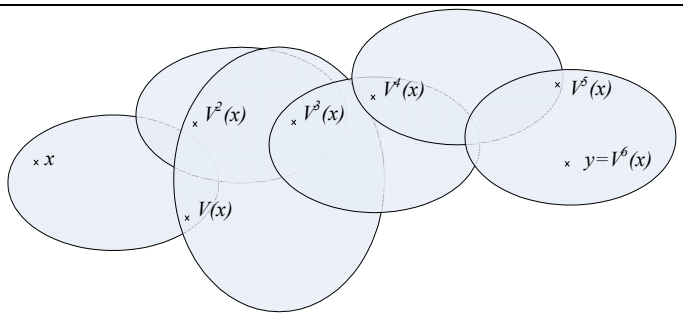

Figure 1-15. Points obtenus par itérations successives d'un voisinage

Définition : optimum local

Soit *P* un problème d'optimisation combinatoire, Ω son ensemble de solutions et *c* une fonction coût. Une solution *x* est dite optimum local pour le voisinage *V* si et seulement si $\forall y \in V(x), c(y) \geq c(x)$ (aucune solution voisine de *x* n'est meilleure que *x*). On note *OL* l'ensemble des optimums locaux.

Bien sûr, ce type de solution n'est en général pas satisfaisant, on souhaite plutôt obtenir la meilleure solution du problème, i.e. l'optimum du problème. Par opposition aux optimums locaux, on appelle ces derniers "optimums globaux".

Définition : optimum global

Soit *P* un problème d'optimisation combinatoire, Ω son ensemble de solutions et *c* une fonction coût. Une solution *x* est dite optimum global si et seulement si $\forall y \in \Omega, c(y) \geq c(x)$ (aucune solution de Ω n'est meilleure que x).

On note *OG* l'ensemble des optimums globaux.

Remarque : un optimum global est un optimum local pour tous les voisinages. A cause du nombre élevé de voisinages, cette remarque ne permet pas de construire une méthode de résolution.

Les notions d'optimums locaux et globaux sont très importantes en recherche locale. En effet, même les algorithmes de recherche locale les moins évolués trouvent facilement et rapidement des optimums locaux. Par opposition, les méthodes de recherche locale évoluées mettent en œuvre des mécanismes pour explorer d'autres solutions. Quelles que soient les méthodes envisagées de recherche locale, elles ont toutes du mal à trouver des optimums globaux. En effet, s'il est facile de trouver un optimum local (i.e. optimum pour le voisinage), on ne sait pas trouver d'optimum global pour les problèmes NP-complets. Il est donc raisonnable d'attendre d'une métaheuristique qu'elle trouve systématiquement un optimum local mais pas un optimum global.

Même s'il n'est pas possible de trouver l'optimum de manière systématique, les méthodes de recherche locale sont tout de même conçues pour trouver les meilleures solutions possibles et la plupart des méthodes de recherche locale se réservent la possibilité d'atteindre l'optimum global. Pour cela, le voisinage utilisé doit avoir certaines propriétés. Suivant la méthode de recherche locale utilisée, les propriétés demandées diffèrent. Les propriétés suivantes sont les plus répandues.

Définition : Voisinage accessible ou connecté

On dit qu'un voisinage *V* est accessible s'il est possible de passer de tout point *x* de l'espace de recherche Ω à tout autre point *y* par un nombre fini d'itérations successives du voisinage *V* :

$\forall x \in \Omega,\ \forall y \in \Omega,\ \exists n \in N \ \ tq \ \ y \in V^n(x)$

Définition : Voisinage faiblement accessible ou propriété d'accessibilité faible

On dit qu'un voisinage V est faiblement accessible s'il est possible de passer de tout point x de l'espace de recherche Ω à un point y optimum global par itérations successives du voisinage V :

$\forall x \in \Omega,\ \forall y \in OG,\ \exists n \in N \ \ tq \ \ y \in V^n(x)$

Définition : Voisinage réversible

On dit qu'un voisinage V est réversible si le voisinage du point x contient le point x :

$\forall x \in \Omega,\ x \in V(x)$

Puisque les problèmes auxquels on s'intéresse sont NP-complets, on sait que l'on ne trouvera pas facilement ni systématiquement d'optimums globaux. Par contre, on peut trouver assez facilement des optimums locaux. Le choix du voisinage est donc primordial car c'est lui qui définit les optimums locaux.

On peut classer les voisinages existants en deux catégories : les voisinages généraux et les voisinages guidés. Les voisinages généraux sont basés sur un codage particulier, il s'agit par exemple du voisinage de permutation qui consiste à permuter deux éléments dans un vecteur représentant la solution. De même, le voisinage par insertion, consiste à retirer un élément du vecteur et à le réinsérer à un autre endroit. De nombreux autres voisinages existent, on trouve par exemple 1-opt, 2-opt et 3-opt qui sont des voisinages généraux mais dont l'interprétation est dans le problème du voyageur de commerce (TSP).

Les voisinages guidés sont en général basés sur des voisinages généraux, mais utilisent en plus des connaissances spécifiques du problème pour supprimer à priori des solutions du voisinage :

Définition : voisinage guidé

Soit un voisinage V, on dit que V' est un voisinage guidé issu de V si

- $V' \subseteq V$
- les solutions de V' sont "meilleures" que les solutions de V

Le terme " meilleur " utilisé dans cette définition est à prendre au sens large. Il existe trois interprétations possibles :

- soit tous les voisins supprimés sont de qualité inférieure aux voisins non supprimés : $\forall x \in V',\ \forall y \in V, c(x) \le c(y)$
- soit le voisinage guidé conserve les bonnes propriétés du voisinage
- soit c'est en moyenne que les voisins de V' sont de meilleure qualité que ceux de V.

Les techniques de guidage permettent en général d'améliorer la qualité des méthodes de recherche locale. Il faut faire attention toutefois à ne pas trop restreindre la recherche par le choix de voisinages trop restreints. Toute la problématique du choix du voisinage est là : il faut un voisinage suffisamment grand pour permettre l'exploration de tout l'espace mais pas trop grand pour ne pas perdre l'efficacité de la recherche.

Les méthodes de recherche locale ont deux possibilités pour utiliser un voisinage : soit elles visitent l'ensemble des points du voisinage, soit elles choisissent de manière aléatoire un ou plusieurs points dans ce voisinage :

Définition : Recherche locale déterministe

Dans une méthode de recherche locale déterministe, on choisit un voisin après avoir parcouru tous les points du voisinage.

Les recherches locales déterministes doivent donc de plus définir l'ordre dans lequel le voisinage est parcouru. Suivant les méthodes, le voisin retenu peut être le meilleur voisin, le meilleur voisin non déjà visité, ... Comme leur nom l'indique les méthodes de recherche locale déterministe n'utilisent pas le hasard, ainsi, leurs résultats ne dépendent pas de la façon dont le générateur aléatoire est initialisé.

Définition : Recherche locale stochastique

Dans une méthode de recherche locale stochastique, on choisit un voisin de manière aléatoire.

Comme leur nom l'indique les recherches locales stochastiques fournissent un résultat différent à chaque exécution. On parle donc de "réplication" de la méthode. La qualité d'une recherche locale stochastique ne peut donc pas être mesurée par une seule réplication, car une réplication peut être particulièrement mauvaise ou particulièrement bonne.

7.3.2 Quelques métaheuristiques

```
Entree : x : solution de départ,
         cx : cout de x
Tant que nécessaire faire
   Pour tout voisin y de x
         cy=c(y);
         si cy<cx alors
               cx=cy;
               x=y;
         finsi
   FinPour
FinTantque
```

Algorithme 1-1. Algorithme de principe de la descente déterministe

Nous présentons dans cette partie les métaheuristiques les plus connues et peut-être les plus simples à mettre en œuvre. Tous les pseudos-algorithmes proposés sont des implantations efficaces dans la mesure où le nombre d'appels à la fonction coût (fonction c) a été minimisé. Les deux métaheuristiques déterministes les plus connues sont la descente déterministe et la méthode taboue.

```
Entree : x : solution de départ,
         cx : cout de x
T = ∅
Tant que nécessaire faire
   Pour tout voisin y de x
         si y ∉ T alors
               cy=c(y)
               si cy<cx alors
                     cx=cy
                     x=y
               finsi
         finsi
   finPour
   Mettre à jour la liste T en y insérant le nouveau x
FinTantque
```

Algorithme 1-2. Algorithme de principe du tabou

La descente déterministe est une recherche locale déterministe qui consiste à choisir le meilleur voisin à chaque itération. Cette méthode trouve très rapidement un optimum local et y reste piégée. Rarement utilisée telle quelle, la descente déterministe a deux utilisations principales : pour caractériser un voisinage et couplée à une autre méthode.

```
Entree : x : solution de départ,
               cx : cout de x
Tant que nécessaire faire
   y voisin de x (i.e. V(x))
   cy=c(y) // y tiré aléatoirement dans V(x)
   si cx ≤ cy alors
         cx=cy
         x=y
   finsi
FinTantque
```

Algorithme 1-3. Algorithme de principe de la descente stochastique

```
Entree : x : solution de départ,
         cx : cout de x
T=Température initiale
Tant que nécessaire faire
   y=V(x) // y tiré aléatoirement dans V(x)
   cy=c(y)
   si cx ≤ cy alors
         cx=cy
         x=y
   sinon si proba() ≤ e^(-(cy-cx)/T) alors
         x=y
         cx=cy
   finsi
   Modifier T
FinTantque
```

Algorithme 1-4. Algorithme de principe du recuit simulé

L'algorithme tabou a été largement utilisé dans la littérature. Le schéma présenté dans l'algorithme 1-2 en est une description relativement générique car la gestion de la liste T peut être réalisée de différentes manières. Dans le chapitre 2[1], nous présentons un algorithme tabou détaillé pour le problème de jobshop.

Les métaheuristiques stochastiques les plus connues sont la descente stochastique, le recuit simulé, les algorithmes génétiques et la méthode du kangourou.

La méthode de descente stochastique ne doit pas être confondue avec son homologue déterministe. Mise à part la façon dont on parcourt les voisinages, les deux méthodes diffèrent aussi par les voisins acceptés. On appelle "palier" un ensemble de solutions voisines de même coût dont au moins une solution est localement optimale. La méthode de descente déterministe s'arrête dès le premier optimum local rencontré. La méthode de descente stochastique va quant à elle être capable de parcourir les solutions du palier. Or, il est possible que l'une de ces solutions ait dans son voisinage une solution de coût inférieur. En parcourant les solutions du palier, la recherche stochastique peut donc être capable de s'extirper du premier optimum local qu'elle rencontre. Cette situation peut être observée sur des problèmes simples et de taille relativement modérée.

Historiquement, le recuit simulé est la première métaheuristique apparue, elle est basée sur des principes thermodynamiques. C'est pourquoi les noms des paramètres du recuit simulé sont empruntés à la thermodynamique, on parle de température, de palier de température, ... Le recuit simulé doit être réglé à l'aide de plusieurs paramètres : la température initiale (température au début du processus d'optimisation), la température finale (température à laquelle le processus est considéré comme gelé) et la façon dont on modifie la température. Plusieurs façons de modifier la température sont possibles, citons principalement les paliers de température (la température est constante pendant un certain

[1] chapitre 2, section 5, page 91

nombre d'itérations puis décroît) et la décroissance géométrique (la température décroît à chaque itération grâce à un coefficient proche de 1 inférieur à 1).

7.4 Évaluation d'un algorithme d'optimisation combinatoire

Le paragraphe précédent présente de nombreux algorithmes, mais tous ne sont pas équivalents. Tous les algorithmes n'ont pas les mêmes objectifs de qualité de solutions et de temps passé à la résolution. Suivant les besoins et les objectifs de l'optimisation, on peut donc réaliser un premier tri parmi les algorithmes d'optimisation. En général, après ce tri, il reste encore de nombreux algorithmes. Pour choisir lequel implanter, on doit pouvoir comparer les algorithmes entre eux.

Cette section contient deux parties. Dans la première partie, on énonce le théorème de "no free lunch" qui montre, qu'en général, aucun algorithme ne surclasse tous les autres. La seconde partie définit des mesures pour comparer les algorithmes sur des jeux de tests communs.

7.4.1 Le théorème " no free lunch "

Dans (Wolpert et Macready, 1995), les auteurs montrent qu'il est vain de vouloir classer les algorithmes d'optimisation combinatoire en général :

> *"(This means in particular that) if some algorithm a1's performance is superior to that of another algorithm a2 over some set of optimization problems, then the reverse must be true over the set of all other optimization problems."*

> *"(Cela signifie en particulier que) si la performance d'un algorithme a1 est supérieure à celle d'un algorithme a2 sur un ensemble de problèmes d'optimisation, alors le contraire est vrai sur l'ensemble des problèmes d'optimisations restants. "*

Nous interprétons ce résultat de la manière suivante : la supériorité d'un algorithme sur un problème vient du fait que cet algorithme est adapté à ce problème. En partant de cette idée, il est tout naturel de développer des algorithmes, des méthodes et des outils dédiés à une classe de problèmes. Ainsi, dans cette thèse nous nous intéressons particulièrement aux algorithmes d'optimisation combinatoire intégrant de la connaissance du problème dans le processus de résolution.

7.4.2 Evaluation de la qualité d'un algorithme

Définition : borne inférieure

Soit P un problème d'optimisation combinatoire, c sa fonction coût et Ω l'ensemble des solutions.

Le nombre lb est une borne inférieure du problème P si toute solution de Ω a un coût supérieur à lb ($lb \leq c(x),\ \forall x \in \Omega$).

Définition : borne supérieure

Soit P un problème d'optimisation combinatoire, c sa fonction coût et Ω l'ensemble des solutions.

Le nombre ub est une borne supérieure du problème P si une solution optimale Ω a un coût inférieur à ub ($c(x) \leq ub,\ \forall x \in OG$).

Pour évaluer la qualité d'un algorithme d'optimisation combinatoire, on utilise généralement les mesures suivantes.

Définition : Qualité d'une instance

Soit A un algorithme d'optimisation combinatoire fournissant une solution $S(I)$ à une instance I d'un problème P. $Z(S(I))$ la valeur du critère pour la solution $S(I)$, $LB_A(I)$ une borne inférieure de la solution pour l'instance I (et l'algorithme A), dans le meilleur des cas $LB_A(I)=O(I)$ la valeur optimale du critère pour l'instance I.

On a $Q_I^A = {(Z(S(I) - LB_A(I))}/{LB_A(I)}$ la qualité de l'algorithme A sur l'instance I.

Mesurer la qualité d'un algorithme sur une instance particulière n'a pas beaucoup de sens. En général, on préfère connaître la qualité de l'algorithme sur plusieurs instances. Comme il est souvent impossible de les tester toutes, on procède par échantillonnage des instances.

Deux grandes méthodes « s'opposent » : soit on travaille sur un ensemble d'instances reconnues par la communauté comme étant représentatives des instances possibles, soit on génère aléatoirement des instances au hasard suivant des plans d'expérience reconnus pour le problème (lois de distribution, écarts-types, etc…). Chacune de ces méthodes a ses limites. La première méthode est subjective car elle dépend des connaissances que la communauté scientifique a d'un problème donné : elle n'est pas utilisable sur un problème nouveau, et peut être erronée sur un problème trop récent sur lequel les instances connues sont particulières : trop faciles, trop difficiles, trop adaptées à certains algorithmes,... La seconde méthode, qui consiste à générer des instances au hasard, est relativement difficile à reproduire puisqu'elle nécessite en général de nombreux paramètres dont le choix du générateur aléatoire. De plus, l'ensemble des instances que l'on peut générer peut ne pas être représentatif des instances rencontrées en réalité.

8 Conclusion

La présentation de la problématique a mis en évidence que l'optimisation pour la planification des systèmes de production nécessite une expertise dans plusieurs domaines complémentaires.

Entre autres, ce chapitre a présenté en détail la modélisation et l'optimisation. Ces deux activités résultent de la double complexité (Grangeon, 2001) des problèmes envisagés : une complexité structurelle et fonctionnelle qui se traduit par la difficulté d'évaluer simplement les critères de performances et une complexité, dite "algorithmique", qui désigne la complexité des problèmes d'optimisation combinatoire soulevés (cf. figure 1-16). Pour affronter conjointement ces deux complexités, nous nous reposons sur les deux démarches suivantes : démarche d'optimisation et démarche de modélisation.

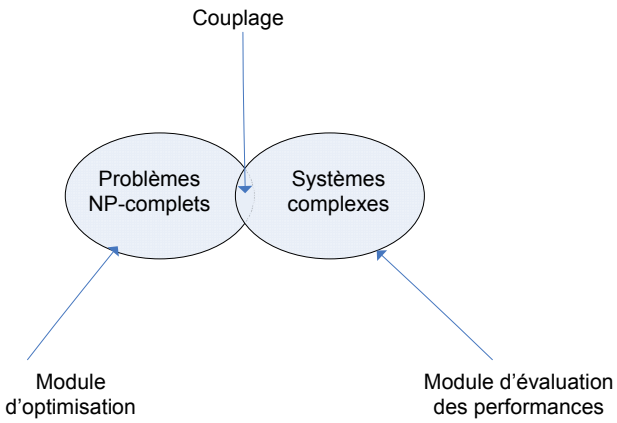

Figure 1-16. Double complexité

Dans la démarche d'optimisation, nous séparons le problème en deux parties distinctes, mais reliées, appelées module d'optimisation et module d'évaluation des performances. Le module d'évaluation des performances répond à la complexité structurelle et fonctionnelle alors que le module d'optimisation répond à la complexité algorithmique. La séparation en deux modules est surtout conceptuelle, car de nombreuses interactions existent.

Dans la démarche de modélisation, nous guidons l'expert dans la construction des modèles. En particulier, la démarche préconise de partir des modèles théoriques existants pour les enrichir progressivement. Ainsi, les méthodes efficaces de la littérature peuvent être réutilisées et étendues.

En suivant ces deux démarches, on aboutit à la conception d'un logiciel d'aide à la décision pour la planification de systèmes de production. Le niveau de détail du module d'évaluation des performances est alors dicté par ses affinements successifs. La réussite du projet de modélisation repose sur la détermination de ces affinements successifs car la prise en compte des deux complexités est plutôt contradictoire : un problème dont la modélisation est trop précise est souvent très difficile à optimiser de manière efficace et inversement, un problème trop simplement modélisé conduit à des solutions non exploitables.

C'est en appliquant cette démarche, que nous nous sommes intéressés aux problèmes de jobshop avec time lags et jobshop avec transport et contraintes additionnelles. Ces deux problèmes sont les modèles théoriques sous-jacents aux exemples de systèmes de production présentés dans le paragraphe 2.3.

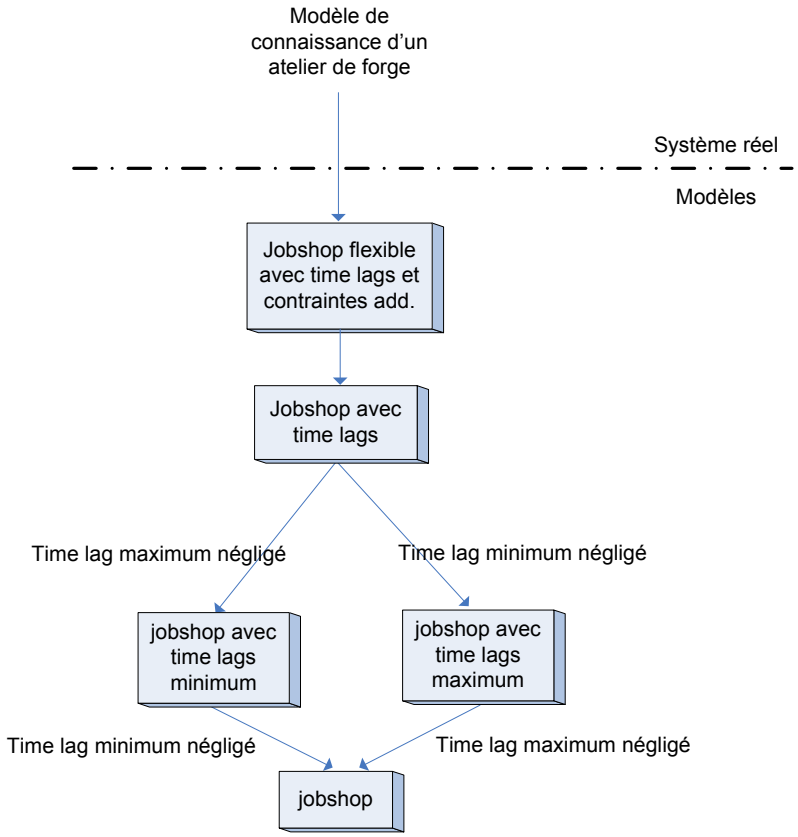

Figure 1-17. Démarche de modélisation appliquée au problème d'atelier de forge

Ainsi, le problème Aubert & Duval se ramène à un problème de jobshop avec contraintes de time lags où les contraintes de time lags imposent des intervalles de temps limités entre deux opérations consécutives. Ces contraintes modélisent les temps de séjour des pièces dans les fours. Les chauffages des pièces étant soumis à des délais minimum et maximum, les délais entre deux opérations sur les presses sont soumis à des délais minimum et maximum. La figure 1-17 est une synoptique de la

démarche appliquée au problème d'atelier de forge. Le schéma se lit du haut vers le bas. En haut est présenté le modèle de connaissances de l'atelier de forge. Ce modèle repose sur un modèle théorique connu qui est présenté en bas du schéma : le problème de jobshop. Entre les deux se situent différents problèmes théoriques avec un nombre d'autant plus grand de contraintes supplémentaires que l'on est haut sur le schéma. Chaque flèche va d'un problème à un problème simplifié. L'hypothèse simplificatrice qui permet de passer d'un problème au suivant est détaillée sur la flèche. Entre les deux figurent plusieurs problèmes de complexité croissante vers le modèle. Le problème le plus difficile étant le problème de jobshop flexible avec time lags et contraintes additionnelles. Le chapitre 3 présente nos propositions pour le problème de jobshop avec time lags dans le but de résoudre un problème d'atelier de forge.

D'autre part, le problème d'ordonnancement des Systèmes Flexibles de Production (SFP) se ramène quant à lui à un jobshop avec transport et contraintes supplémentaires. Le transport modélise le véhicule qui déplace les pièces d'une machine à l'autre alors que les contraintes supplémentaires modélisent le nombre de jobs simultanément autorisés, les capacités des stocks, ... La figure 1-18 est une synoptique de la démarche de modélisation appliquée aux Systèmes Flexibles de Production (SFP). Le schéma se lit du haut vers le bas. En haut est présenté le modèle de connaissances des Systèmes Flexibles de Production. Ce modèle repose sur un modèle théorique connu qui est présenté en bas du schéma : le problème de jobshop. Entre les deux se situent différents problèmes théoriques avec un nombre d'autant plus grand de contraintes supplémentaires que l'on est haut sur le schéma. Chaque flèche va d'un problème à un problème simplifié. L'hypothèse simplificatrice qui permet de passer d'un problème au suivant est détaillée sur la flèche. Dans le chapitre 4, nous présentons nos propositions pour le problème de jobshop avec transport et contraintes additionnelles.

Dans le chapitre 5, nous montrons que la démarche d'optimisation est généralisable et qu'elle peut servir de cadre à la conception et à la mise en œuvre de logiciels d'optimisation.

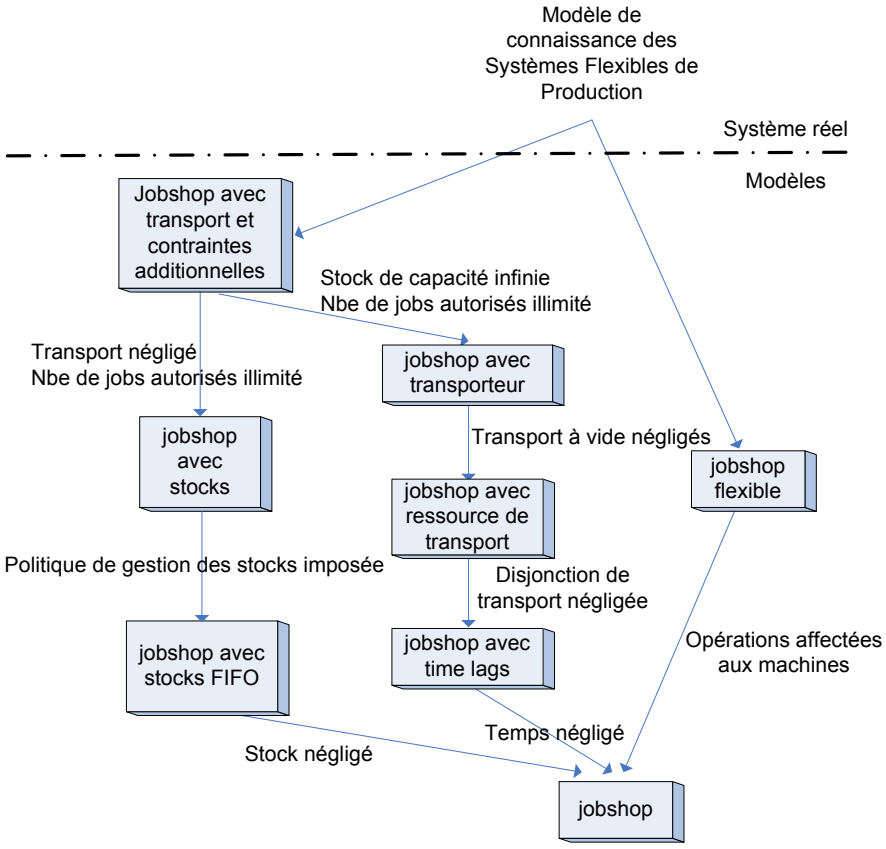

Figure 1-18. Démarche de modélisation appliquée au Systèmes Flexibles de Production

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

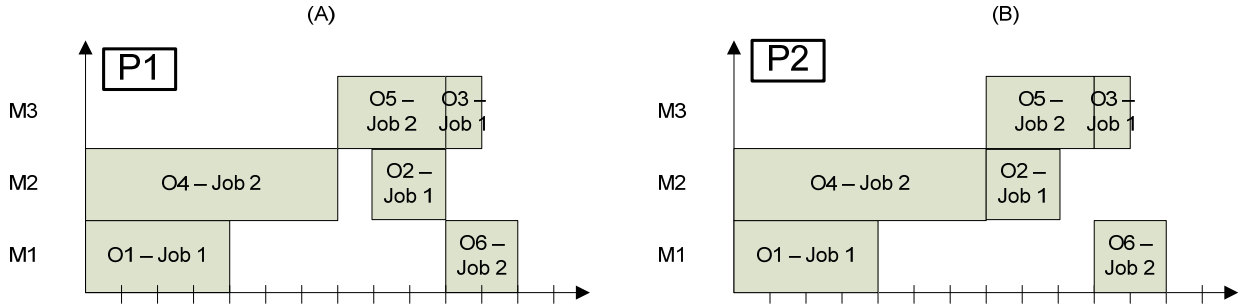

Figure 2-1. Ordonnancement optimal, non semi-actif (A)

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).

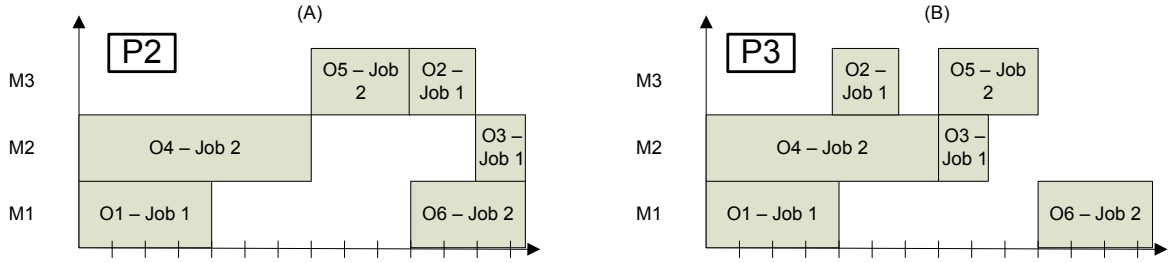

Figure 2-2. (A) est un ordonnancement optimal et non actif

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.

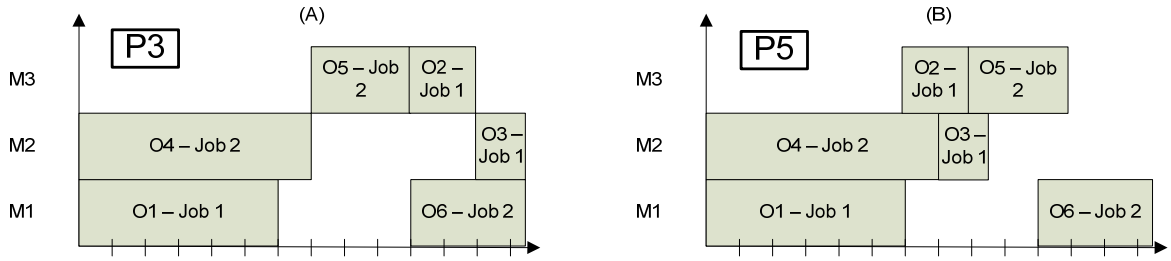

Figure 2-3. (A) est un ordonnancement optimal et non sans délai et

(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.

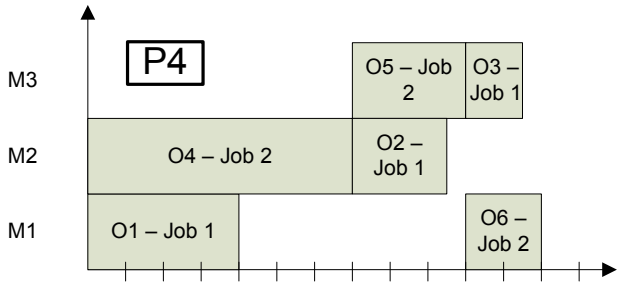

Figure 2-4. Ordonnancement optimal et sans délai

3.5 Existence du point 4

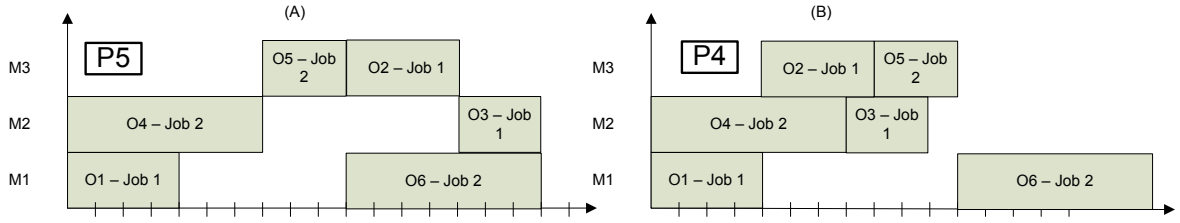

Figure 2-5. Ordonnancement optimal (A), sans délai et non optimal (B)

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.

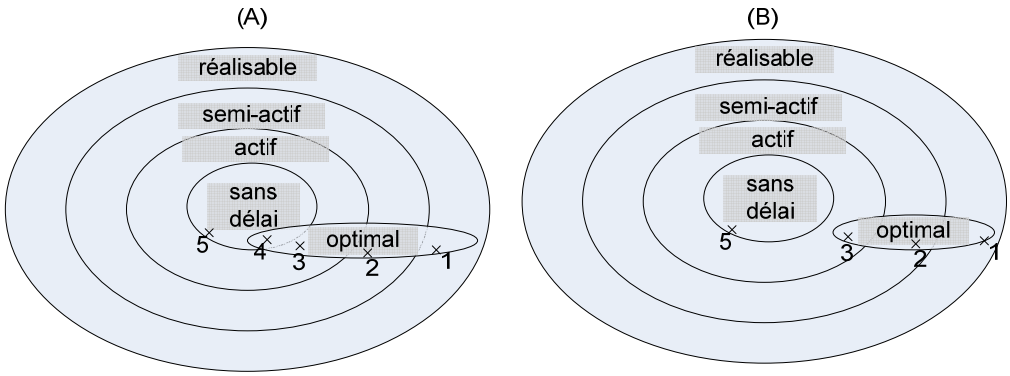

Figure 2-6. Classes d'ordonnancements et optimalité

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

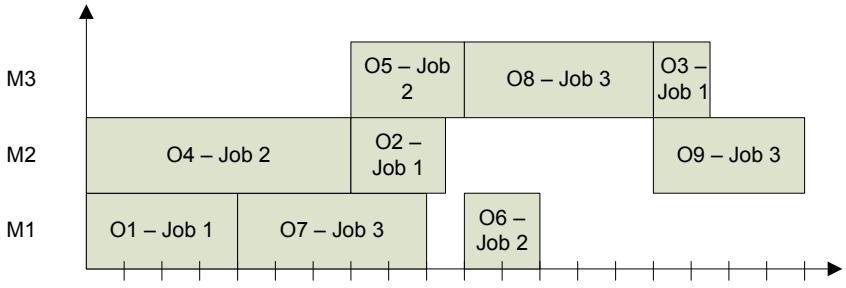

Figure 2-7. Diagramme de Gantt de la solution S1

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 $\max(n,m)^{nm} \le r^{nm} \le nm^{nm}$	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.

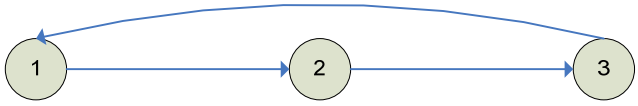

Figure 2-8. Sélection non valide de trois opérations

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.

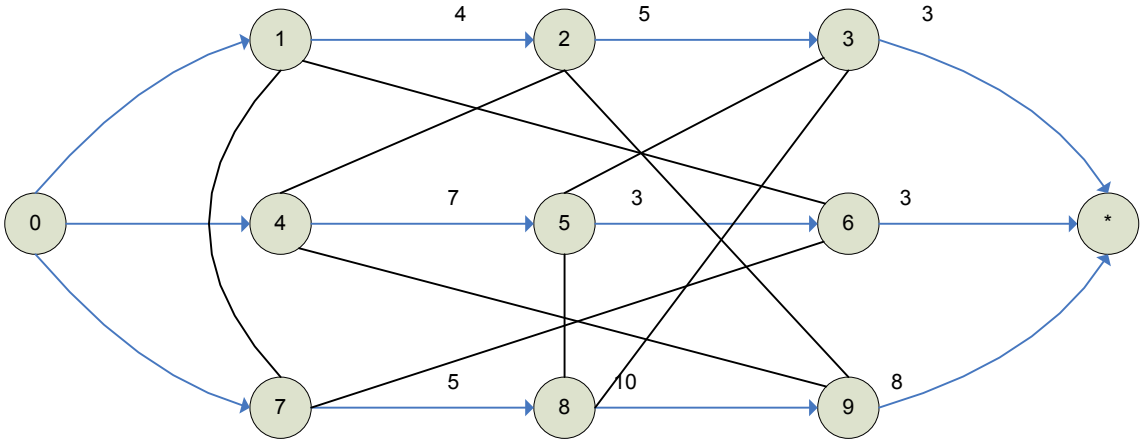

Figure 2-9. Graphe disjonctif du problème J1

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 $4 \to 2$ et de $2 \to 9$. Le graphe conjonctif correspondant à cet exemple est présenté dans la figure 2-13.

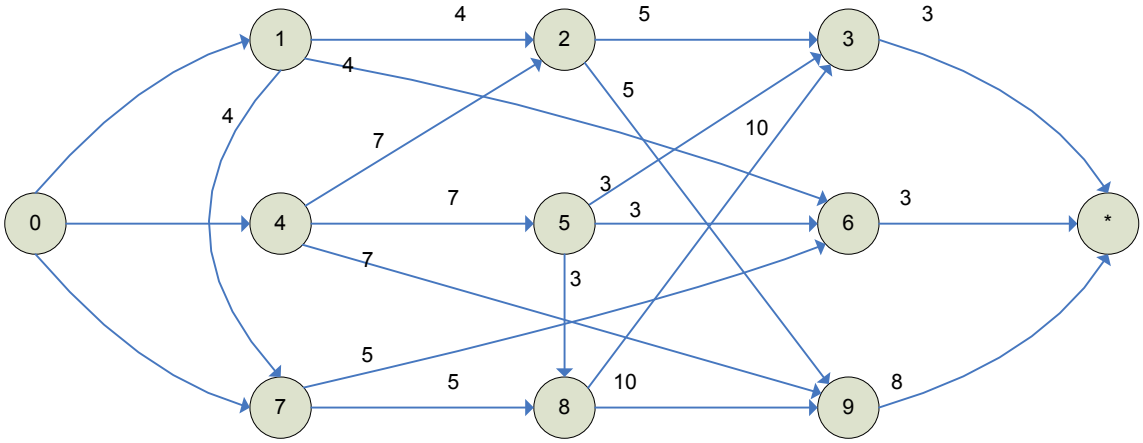

Figure 2-10. Graphe conjonctif orienté de la solution S1

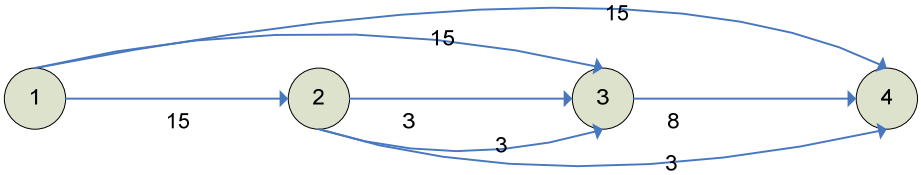

Figure 2-11. Quatre opérations en disjonction dont 4 arcs redondants

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

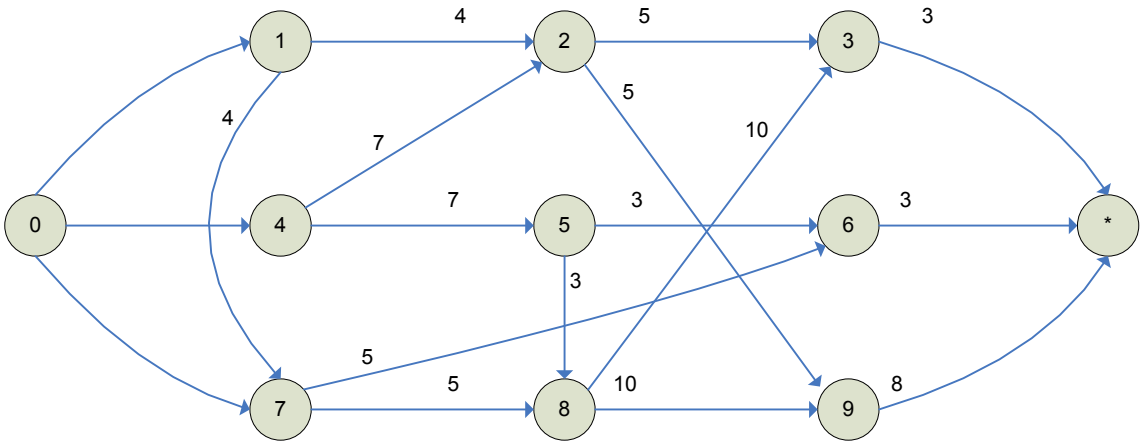

Figure 2-13. Graphe conjonctif de la solution S1

4.4.3 Évaluation du graphe

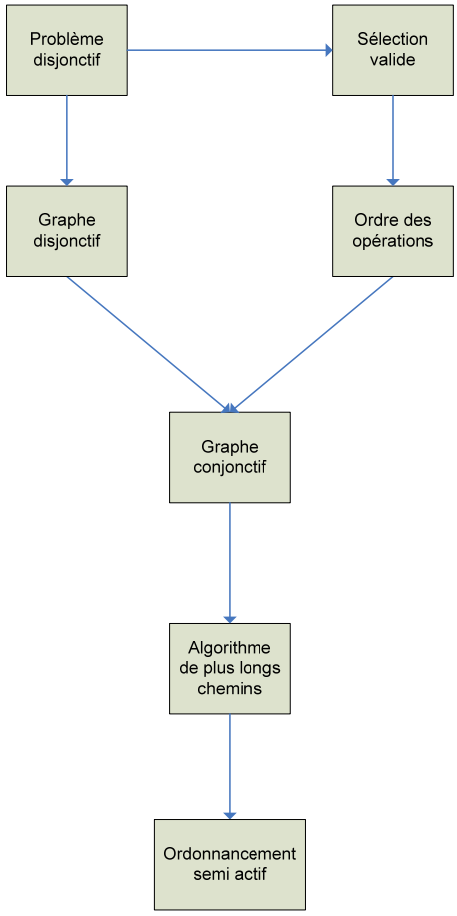

Figure 2-14. Vue d'ensemble du modèle de graphe conjonctif - disjonctif

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.

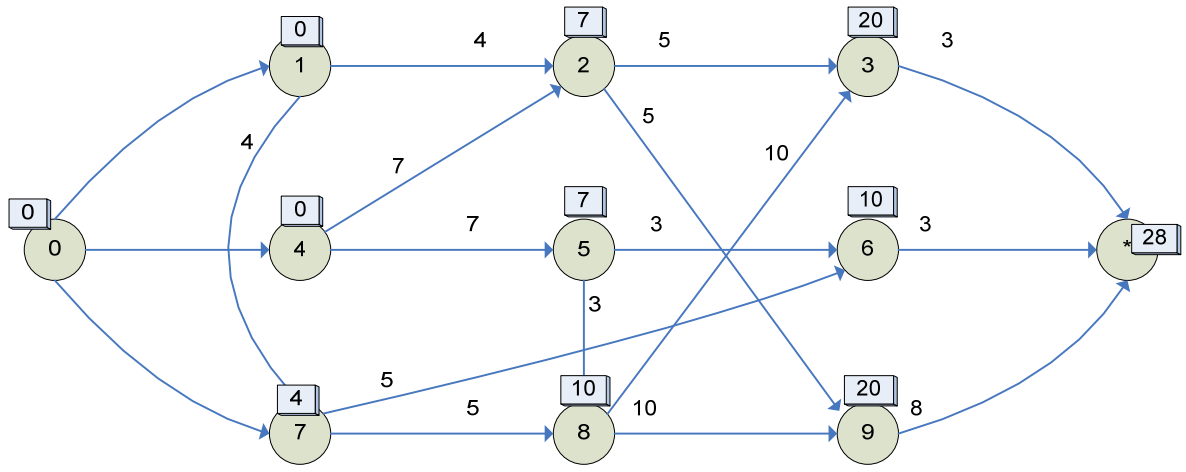

Figure 2-15. Graphe conjonctif évalué

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.

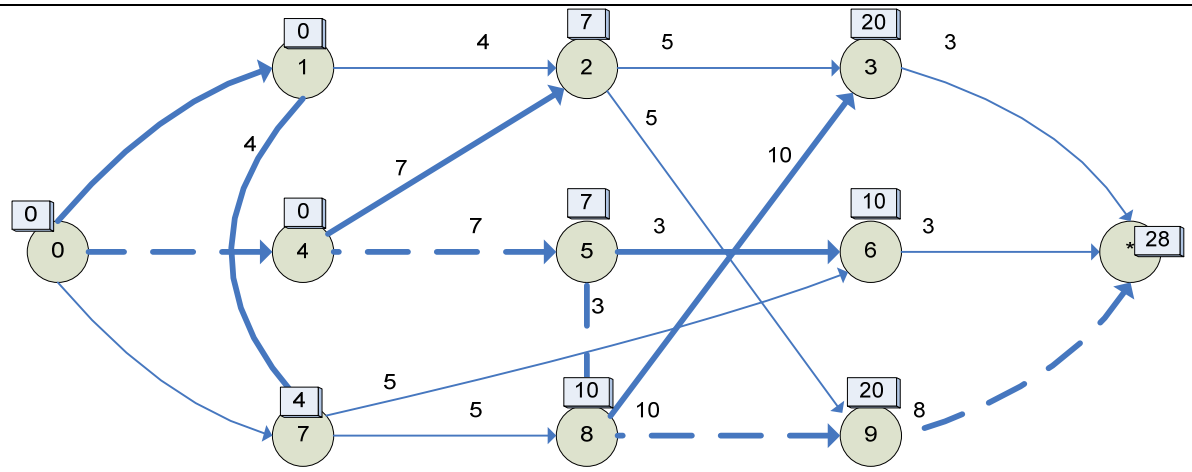

Figure 2-16. Graphe conjonctif évalué avec le sous graphe critique (en pointillés)

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

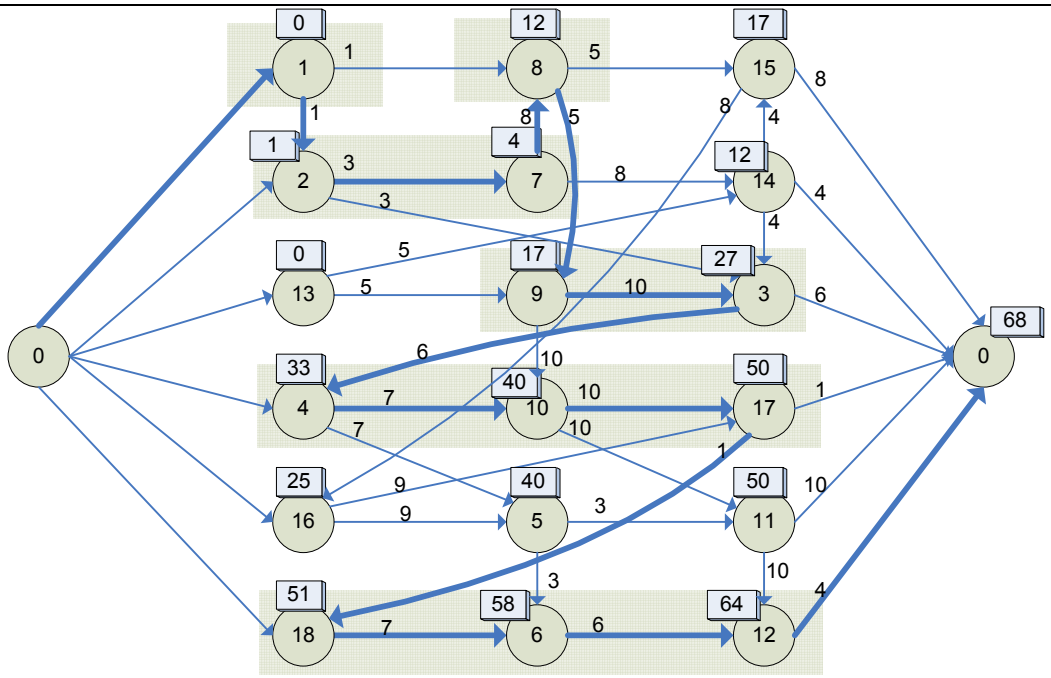

Figure 2-17. Chemin critique et voisinage de Laarhoven de S2

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.

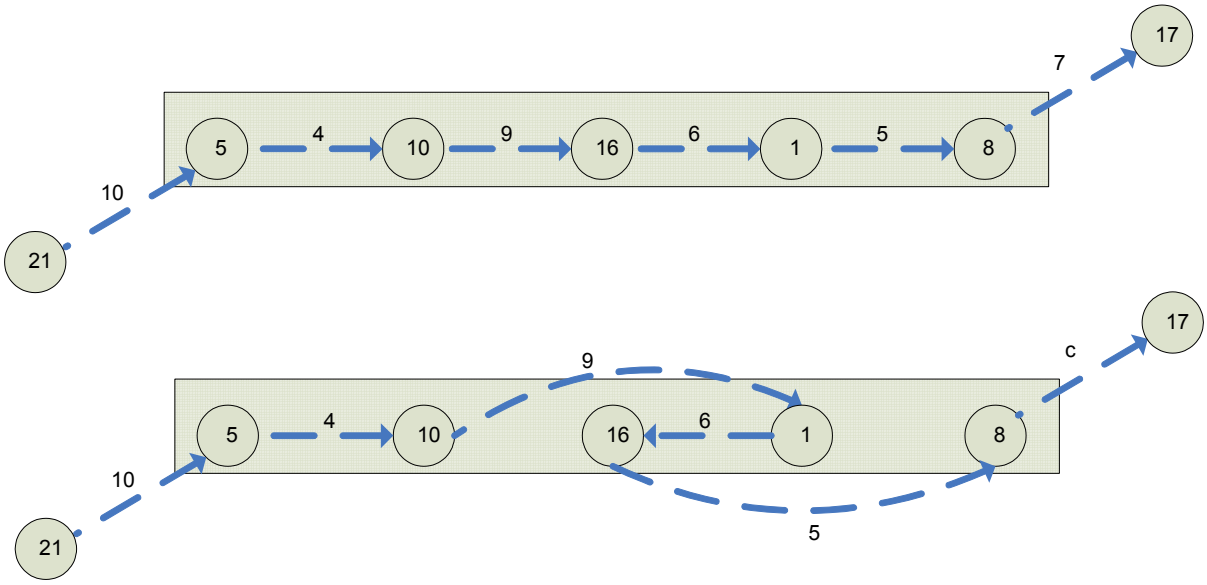

Figure 2-18. Modification à l'intérieur d'un bloc

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).

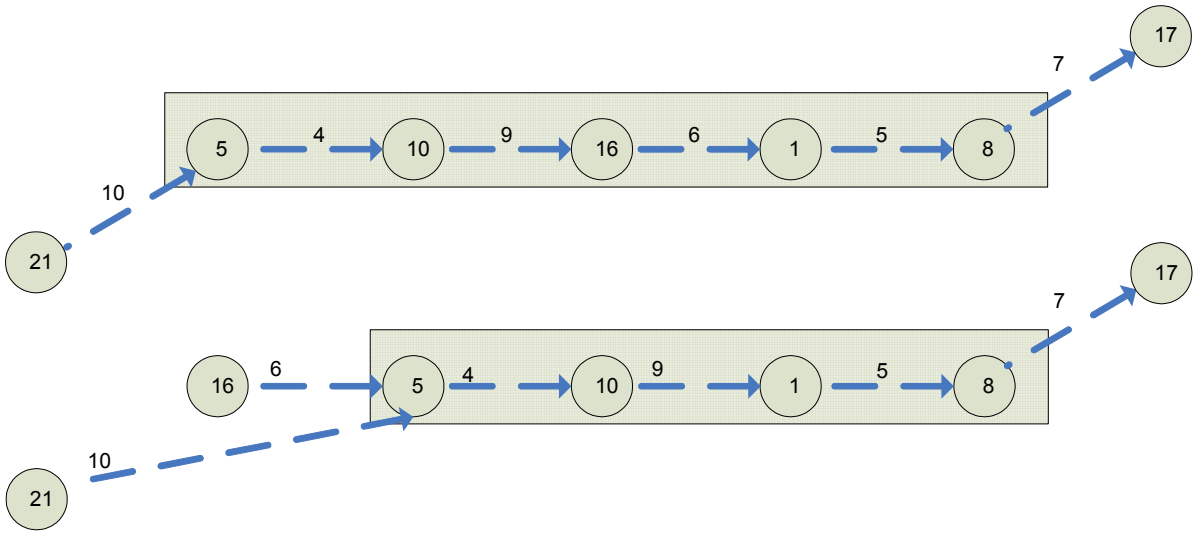

Figure 2-19. Voisinage de Dell Amico

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).

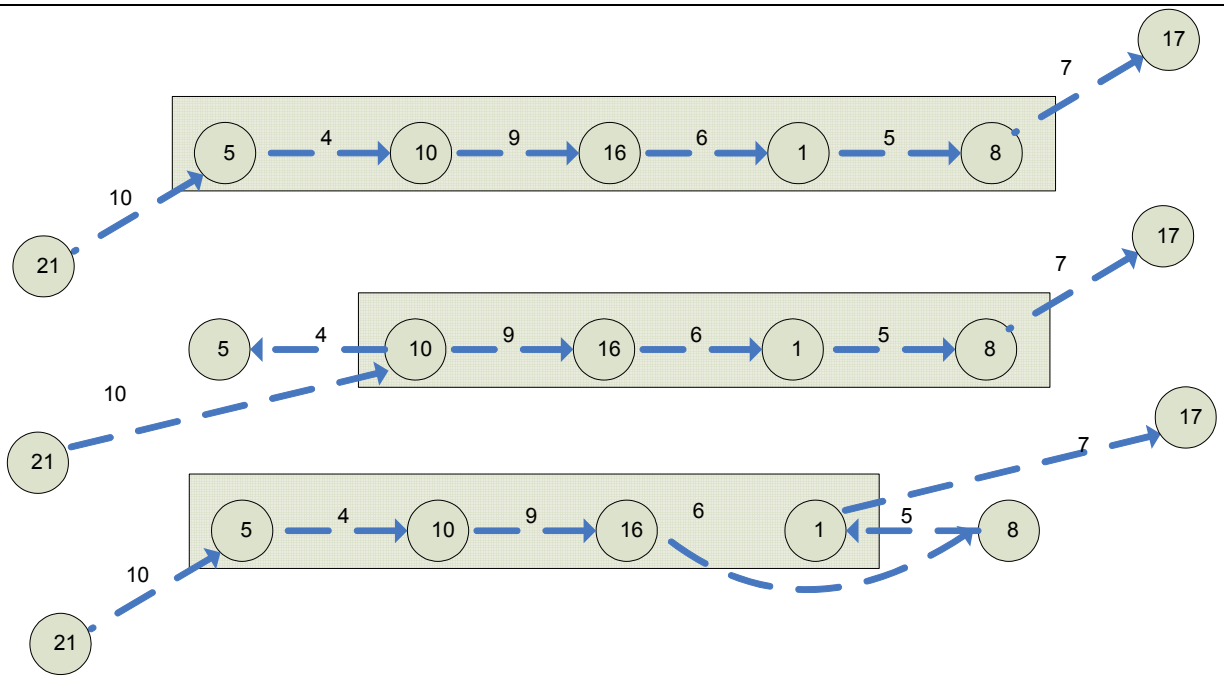

Figure 2-20. Voisinage de Nowicki et Smutnicki

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.

Chapitre 3 Problème de jobshop avec time lags

Ce chapitre est consacré à la résolution du problème de jobshop avec time lags.

Sommaire

Table des figures

Table des tableaux

Table des algorithmes

1 Introduction

La contrainte de time lag vient compléter l'énoncé d'un problème initial en imposant des inégalités entre les dates de début ou de fin des opérations (une définition plus précise est fournie dans la suite de ce chapitre). Dans ce chapitre, nous nous intéressons particulièrement à un cas particulier des contraintes de time lags appelé time lag minimum et maximum.

Dans le chapitre 1, nous avons proposé de partir du problème de jobshop afin d'étudier le problème d'atelier de forge de l'entreprise Aubert & Duval. De nombreuses méthodes efficaces ont été proposées pour la résolution du problème de jobshop (cf. chapitre 2). Nous proposons donc d'étendre les méthodes existantes pour le problème de jobshop afin de prendre en compte les contraintes de time lags.

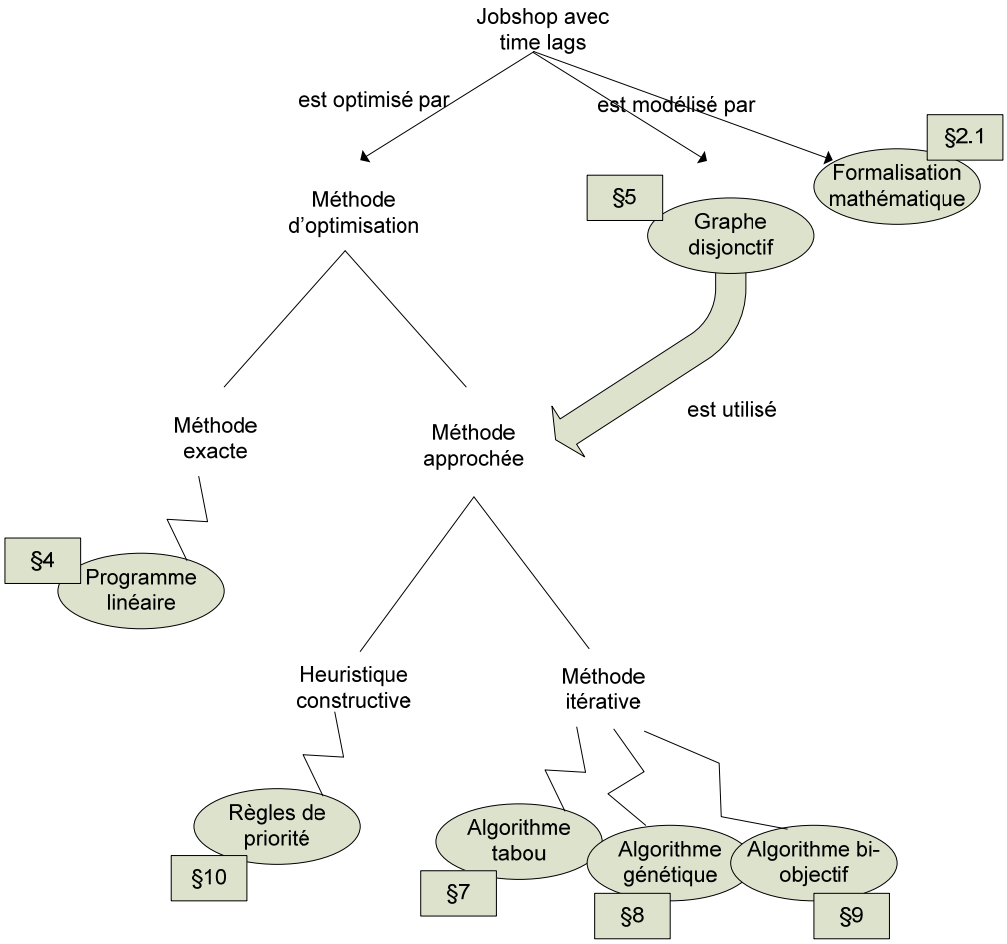

Figure 3-1. Synoptique de nos propositions pour le jobshop avec time lags

La figure 3-1 est un synoptique de nos propositions pour le jobshop avec time lags. Ces propositions concernent à la fois des modèles et des méthodes d'optimisation adaptés à la contrainte de time lags. Les deux modèles proposés sont la formalisation mathématique (présentée dans la section 2.1) et un modèle de graphe conjonctif - disjonctif (présenté dans la section 5). Ces deux modèles sont utilisés pour mettre en œuvre des méthodes d'optimisation : la formalisation mathématique est utilisée pour écrire un programme linéaire en nombres entiers (présenté dans la section 4) alors que le graphe conjonctif - disjonctif est utilisé pour mettre en œuvre des méthodes approchées (sections 7 à 10).

Grâce à ces deux modélisations, nous proposons des méthodes d'optimisation variées : le programme linéaire permet de déterminer la solution optimale par résolution directe des plus petites instances alors que les méthodes approchées permettent de construire des solutions approchées (et à performances non garanties) de toutes les instances. En guise de méthode approchée, nous proposons

une heuristique de construction permettant de construire rapidement des solutions faisables de bonne qualité (cf. section 10), mais nous proposons aussi des méthodes itératives permettant de trouver des solutions proches de l'optimal. Nous proposons ainsi trois méthodes itératives complémentaires de par les instances qu'elles permettent de résoudre.

2 Présentation et état de l'art des contraintes de time lag

La contrainte de time lag est introduite pour la première fois en 1958 par Mitten (Mitten, 1958). Celui-ci généralise l'algorithme polynomial de Johnson pour la résolution du flowshop à deux machines en proposant une méthode de génération d'ordonnancements de permutation. Cependant, et contrairement au problème sans time lag, Mitten remarque qu'il n'existe pas forcément d'ordonnancement optimal qui soit un ordonnancement de permutation. Depuis cet article fondateur, les articles traitant de time lags sont relativement peu nombreux.

Parmi les contraintes de time lags, on distingue celles qui concernent des opérations consécutives du même job. On sépare ces dernières en deux catégories : les time lags minimum, qui imposent une durée d'attente minimum entre deux opérations, et les time lags maximum, qui imposent une durée d'attente maximum entre deux opérations. Ce sont principalement ces deux types de contraintes de time lags que nous étudions dans cette thèse. Or, elles ne s'utilisent pas au même niveau dans notre démarche de modélisation. Par exemple, les contraintes de time lags minimums sont utilisées pour modéliser un processus chimique ou thermique dans lequel une durée minimum est imposée pour conférer les bonnes propriétés aux produits. En outre, les contraintes de time lags minimums sont aussi utilisées pour modéliser des opérations dont les ressources ne sont pas, en général, critiques : opérations transport réalisées par un transporteur dédié, opérations réalisées par un opérateur toujours disponible, … La contrainte de time lag est donc soit une contrainte issue de la simplification d'une ou plusieurs contraintes plus complexes, soit elle peut être l'expression directe et exacte d'une contrainte imposée par la gamme du produit (cas exposé dans le chapitre 1 concernant un atelier de forge). Pour la contrainte de time lag maximum, la plupart des applications utilisent le time lag pour modéliser une durée maximum imposée par la gamme. Nous ne connaissons pas d'application dans laquelle le time lag maximum est une simplification d'une contrainte plus complexe. Ces deux contraintes n'apparaissent donc pas toujours au même niveau dans notre démarche de modélisation.

2.1 Formalisation mathématique

Afin de définir précisément le problème étudié et pour faciliter l'écriture de la formalisation linéaire, nous commençons par décrire le problème de jobshop avec time lags à l'aide d'une formalisation mathématique.

Pour le sous-problème de jobshop, la formalisation mathématique est détaillée dans le chapitre 2[2]. Pour formaliser mathématiquement une contrainte de time lag, on utilise deux opérations i et j ainsi qu'une constante de temps l_{ij}. La contrainte s'exprime alors par l'inéquation suivante : $t_i + l_{ij} \leq t_j$. Suivant le signe de la constante et les opérations i et j concernées, les contraintes de time lag ont une difficulté et un champ d'application très différents. Dans cette présentation, nous présentons une typologie des contraintes de time lag. On distinguera les time lags généraux, les time lags minimum et maximum, qui constituent les sections suivantes.

La formalisation ci-dessous résume donc le problème de jobshop avec time lags général. Dans cette formalisation, des contraintes de time lags sont introduites entre toutes paires d'opérations. Si l'on

[2] chapitre 2, section 2.1, page 56

ne souhaite pas activer la contrainte entre deux opérations i_0 et j_0, il suffit de choisir la constante $l_{i_0,j_0} = -\infty$.

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

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

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

(4) $t_i + l_{ij} \le t_j \quad \forall (i,j) \in O^2$

Remarque : Dans la suite et pour simplifier les explications, on appelle simplement "time lag" la contrainte de time lag.

2.2 Time lags généraux

Afin de détailler précisément tous les types de time lags existants, ce paragraphe commence la présentation en s'intéressant aux time lags généraux, c'est-à-dire les contraintes de time lags dont l'expression est la plus générale possible (i, j et l_{ij} sont quelconques). Il est à noter que la contrainte exprimant les time lags généraux fait intervenir les dates de début des opérations (contrainte début/début). À travers cette présentation, les quelques articles traitant de time lags généraux sont présentés.

Les time lags généraux rendent la résolution très difficile. Par exemple, considérons le problème à une machine sans contrainte supplémentaire (dont l'objectif est de minimiser le makespan), ce problème est trivial car tout ordonnancement semi-actif est optimal. Pourtant, le même le problème à une machine et des time lags généraux est NP-complet (Brucker *et al.* 1999b). En outre, les auteurs montrent que de nombreux problèmes d'ordonnancements peuvent être réduits à un problème à une machine et des time lags généraux. Par exemple, les instances de problèmes de "general shop", de problèmes multiprocesseurs, et de problèmes à machines dupliquées sont transformés en problème à une machine et time lags généraux. Le nombre de jobs dans le problème réduit augmente de manière quadratique avec le nombre de jobs dans le problème original. Les auteurs montrent aussi que la découverte d'une solution réalisable est un problème NP-difficile et peut être résolu à l'aide d'une procédure de séparation / évaluation (Brucker *et al.* 1999a). Hurink et Keuchel (2001) proposent un algorithme tabou pour le problème à une machine. À l'aide de cet algorithme et de la réduction proposée par Brucker, les auteurs résolvent des instances à plusieurs machines à travers des instances à une machine et time lags généraux.

Les articles traitant des problèmes à une machine sont relativement peu nombreux. Dans (Balas, 1995), les auteurs proposent des bornes inférieures et supérieures de bonne qualité. Finta et Liu (1996) montrent que le problème est NP-difficile dans le cas où les temps de traitement sont unitaires et où les time lags entiers. Par contre, les auteurs proposent un algorithme polynomial dans le cas où les temps de traitement sont entiers et les time lags unitaires. Les auteurs traitent aussi bien le cas préemptif que le cas non préemptif. Pour le problème à une machine et time lags maximaux, Wikum *et al.* (1994) proposent des résultats de complexité et montrent que les problèmes très simples avec time lags sont déjà NP-difficiles. Les auteurs proposent des heuristiques à performances garanties pour une configuration particulière des contraintes.

Mis à part les problèmes à une machine, les time lags généraux ne sont pas étudiés dans la littérature. En effet, le problème à une machine est déjà particulièrement difficile et aucune application pratique n'est apparue pour les problèmes à plus d'une machine. Par contre, des contraintes de time lags plus particulières sont étudiées. On distingue deux utilisations principales :

- La contrainte est utilisée pour modéliser une contrainte du système réel. La plupart des applications prennent donc en compte des problèmes à machines multiples.
- Ou la contrainte de time lag est utilisée pour modéliser une simplification d'un problème plus complexe. En particulier, dans le contexte de la procédure par machine goulot, cette contrainte permet d'écrire des relaxations plus fines que les relaxations habituelles. Par exemple, des procédures par machine goulot améliorées ont été développées par

Ladhari et Haouari (2005) pour le flowshop et Balas (1995) pour le jobshop. Dans ce contexte, les problèmes étudiés comportent une ou deux machines.

Les deux parties suivantes présentent les contraintes de time lag minimum et maximum.

2.3 Time lag minimum

Un time lag minimum est un time lag entre une opération i et son opération suivante dans la gamme $SJ(i)$. Ces contraintes sont généralement exprimées en fonction de la date de fin d'une opération et de la date de début de l'opération suivante (contrainte fin/début). La constante d'un time lag minimum est notée $t_i^{\min}$, elle vient donc s'ajouter à la durée de l'opération.

La contrainte de time lag minimum s'écrit alors $t_i + p_i + t_i^{\min} \le t_{SJ(i)}$. Suivant la valeur de la constante, on distingue trois cas (figure 3-2).

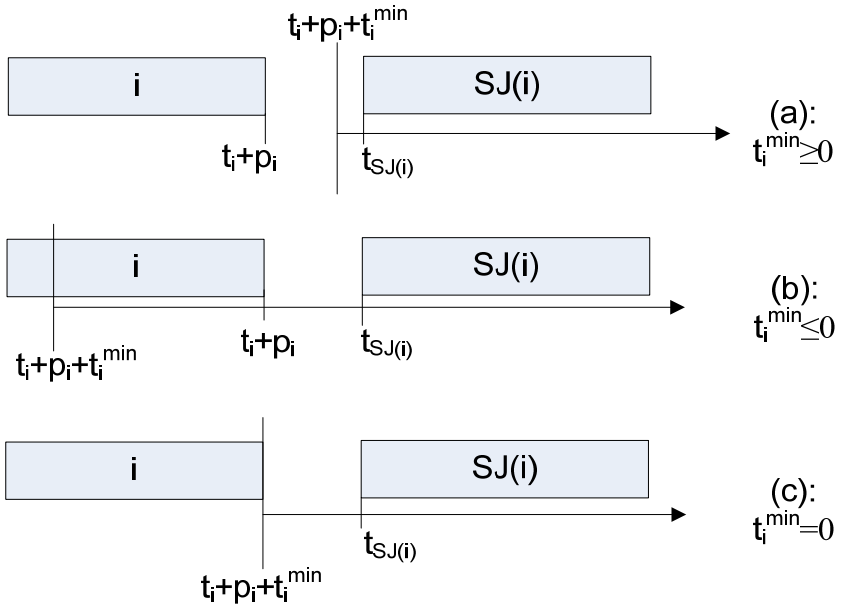

Figure 3-2. Time lag minimum - time lag positif

Dans le cas (a), la constante de time lag minimum est de durée positive. Cela signifie que la prochaine opération de la gamme du job doit attendre un délai supplémentaire de $t_i^{\min}$ unités de temps avant de commencer son traitement sur la prochaine machine. Grâce à cette contrainte, on peut modéliser qu'un job reste indisponible plus longtemps que la durée de l'opération sur la machine. La durée $t_i^{\min}$est le temps pendant lequel le job reste indisponible alors que la machine redevient disponible.

Ce type de contrainte est utilisé, par exemple, pour modéliser un temps de transport entre deux opérations consécutives. Ainsi, le délai $t_i^{\min}$ correspond au temps de transport du job. Par contre, le time lag minimum ne permet pas de prendre en compte la disponibilité du transporteur, c'est-à-dire que deux jobs peuvent simultanément être en cours de transport. La contrainte est donc adaptée au cas où la ressource de transport n'est pas une ressource critique (Aanen, *et al.* 1993), (Strusevich, 1999), (Rebaine et Strusevich, 1999). Le time lag minimum peut aussi modéliser une opération réalisée par d'autres types de ressources non critiques. On peut citer par exemple, des opérateurs en nombre suffisant, ... (Botta-Genoulaz, 2000), (Dell'Amico, 1996), (Mitten, 1958).

Dans le cas (b), la durée du time lag minimum est négative. Cela signifie que la prochaine opération dans la gamme anticipe la fin de l'opération. Ce type de contrainte est utilisé par exemple pour modéliser des ateliers de production par lots. Si un lot commence son traitement sur une machine, il arrive qu'on puisse anticiper le début de la production sur la machine suivante lorsqu'une certaine fraction du lot est terminée sur la première machine. Ainsi les deux opérations consécutives qui

modélisent le traitement des deux lots se chevauchent dans les ordonnancements. Dans ce cas et pour que le modèle soit valide, il ne faut pas considérer les contraintes technologiques classiques aux modèles d'atelier (i.e. les contraintes imposant qu'une opération ne commence que lorsque la précédente est terminée). (Kim, 1997) utilise la contrainte de time lag dans ce contexte.

Enfin le cas (c) concerne un time lag minimum de durée nulle. Cela exprime une contrainte de précédence classique entre deux opérations consécutives (appelée contrainte technologique). Ces contraintes sont très classiques et apparaissent dans les problèmes de flowshop, jobshop, openshop, RCPSP...

Les articles traitant des time lags minimums sont les plus nombreux dans la littérature. Cet état de l'art couvre les principaux résultats, mais n'est pas exhaustif. En effet, les contraintes de time lag minimum apparaissent sous de nombreux noms, dans de nombreuses formes et dans de nombreux problèmes différents. Par rapport aux travaux de cette thèse, la prise en compte des contraintes de time lag minimum ne constitue pas la difficulté majeure.

Les problèmes à deux machines restent polynomiaux dans le cas où on cherche un ordonnancement de permutation (Mitten, 1958). Pourtant, Dell'Amico (1996) montre que la plupart des problèmes deviennent NP-difficiles quand on y introduit les contraintes de time lag minimum. Dans cet article, les auteurs fournissent des bornes inférieures et supérieures et analysent leurs performances dans le pire des cas. De plus, les auteurs proposent un algorithme tabou. D'autres bornes inférieures sont proposées par Ladhari et Haouari (2005). Des algorithmes exacts ainsi que des approximations sont proposés pour ce problème. Dans (Rebaine et Strusevich, 1999) et (Strusevich, 1999), les auteurs proposent une 8/5 approximation pour ce problème complétée par une 3/2 approximation si tous les time lags sont égaux. Un algorithme de Branch and Bound est proposé par (Aanen, *et al.* 1993).

Pour les problèmes à machines multiples, des problèmes variés ont été traités. Pour le problème de flowshop, Riezebos (1998) propose une borne inférieure et une heuristique basée sur cette borne. Kim (1997) utilise les time lags pour modéliser le traitement de lots dont une opération peut commencer avant que l'opération précédente ne soit terminée. Les auteurs proposent plusieurs métaheuristiques dont un algorithme tabou, un recuit simulé, une recherche à voisinage variable, ... (Botta-Genoulaz, 2000) propose six nouvelles heuristiques pour ce problème dont les performances et la robustesse sont étudiées. Munier *et al.* (1998) proposent des algorithmes de liste pour le problème à machines parallèles.

2.4 Time lag maximum

Une contrainte de time lag maximum est une contrainte de time lag entre une opération et son opération précédente dans la gamme et dont la constante de temps est négative. Tout comme les contraintes de time lag minimum, les contraintes de time lags maximums sont généralement exprimées en fonction de la date de fin d'une opération et de la date de début de l'opération suivante (contrainte fin/début). La contrainte s'écrit donc $t_{SJ(i)} - \left|l_{iSJ(i)}\right| \le t_i$. Pour faciliter les explications et les notations, on réécrit cette inéquation en $t_{SJ(i)} \le t_i + p_i + t_i^{\max}$. Par identification, on a $t_i^{\max} + p_i = \left|l_{i,SJ(i)}\right|$, cette durée appelée "durée du time lag maximum", cette durée est positive.

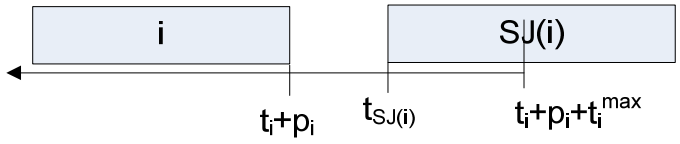

Figure 3-3. Contrainte de time lag maximum - interprétation

Les contraintes de time lag maximum permettent de modéliser un délai maximum entre deux opérations. Les exemples d'application sont multiples, ils concernent par exemple les processus

thermiques ou chimiques : temps limité entre deux opérations pour éviter le refroidissement d'un produit, pour des raisons d'hygiène dans l'agro-alimentaire, … (cf. (Deppner, 2003) pour une liste d'applications étendue).

D'un point de vue de résolution, l'introduction de ces contraintes pose deux difficultés majeures : l'évaluation d'une solution et la présence de nombreux ordonnancements irréalisables. Pourtant et contrairement au cas général, la recherche d'une solution réalisable reste un problème simple puisque polynomial. Dans la suite, nous présentons un ensemble de *n !* solutions réalisables (cf. paragraphe 6.4) et une heuristique construisant des solutions forcément réalisables (cf. paragraphe 10). Il n'en reste pas moins que trouver une solution qui soit à la fois réalisable et de bonne qualité reste un problème difficile. Pour illustrer ces deux difficultés, nous considérons le cas où l'on recherche le meilleur ordonnancement pour un ordre donné. Ce cas est largement répandu : nous avons vu dans le chapitre 2 que la plupart des représentations utilisées pour le jobshop sont basées sur des ordres d'opérations, si elles ne codent pas directement un ordre, celui-ci est sous jacent.

Lorsque l'on passe par un ordre pour créer une solution, on génère des ordres pour lesquels il faut chercher le meilleur ordonnancement respectant cet ordre. En général cet ordonnancement est l'ordonnancement semi-actif (i.e. le plus calé à gauche). Pour le problème de jobshop avec time lag maximum, les propriétés des ordonnancements semi-actifs changent car, il est possible, que pour un ordre donné :

- des attentes doivent être insérées pour rendre l'ordonnancement réalisable, c'est-à-dire que certaines opérations ne commencent pas immédiatement leur traitement alors que le job et la machine sont disponibles.
- pour certains ordres, aucun ordonnancement ne respecte à la fois l'ordre fixé et les contraintes de time lag maximum.

Nous illustrons ces deux cas par l'ordonnancement de la figure 3-4. À cette instance, nous ajoutons deux contraintes de time lag maximum de durée 2, une entre les opérations *O2* et *O3* puis une autre entre les opérations *O1* et *O2*. La solution présentée est en cours de détermination : les opérations ont une date de début respectant l'ordre et toutes les contraintes sauf la contrainte de time lag maximum entre les opérations *O2* et *O3*.

L'ordonnancement présenté respecte déjà la contrainte de time lag maximum entre les opérations *O1* et *O2*. Pour cela, il a été nécessaire de décaler l'opération *O1* d'une unité de temps vers la droite. On dit qu'une attente a été insérée avant l'opération *O1*. En effet, l'opération *O1* peut commencer dès la date *0*, et si on décale cette opération c'est pour une raison qui apparaît plus tard dans l'ordonnancement. C'est à cause de cette anticipation que l'évaluation d'une solution n'est pas triviale. De plus, ces décalages peuvent s'enchaîner car une opération peut être décalée à cause d'un time lag maximum et le fait de décaler cette opération peut entraîner le décalage d'une autre opération, ... Le paragraphe suivant illustre le cas où ces modifications sont cycliques.

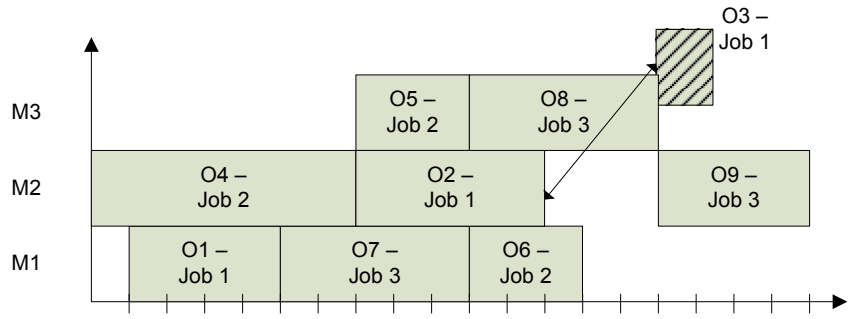

Figure 3-4. Ordonnancement partiel et contrainte de time lag maximum à respecter

On souhaite modifier l'ordonnancement de la figure pour qu'il respecte toutes les contraintes y compris la contrainte de time lag maximum entre les opérations *O2* et *O3*. Pour cela, il est nécessaire de réduire l'intervalle de temps entre ces deux opérations. Puisque l'ordonnancement est calé à gauche, c'est l'opération *O2* qui sera décalée vers la droite. En décalant cette opération d'une unité, on réduit

suffisamment l'intervalle de temps pour que la contrainte de time lag maximum soit respectée. Par contre, la contrainte entre les opérations *O1* et *O2* devient violée. Pour rendre cette contrainte réalisable, il faut donc aussi décaler l'opération *O1*. Or ce dernier décalage entraîne les opérations *O5*, *O6*, *O7*, *O8* et *O3*. On se retrouve donc dans l'état initial, la contrainte entre les opérations *O2* et *O3* est de nouveau violée. Aucun décalage des opérations ne permet de trouver un ordonnancement réalisable. Nous verrons par la suite que cette situation est caractérisable par un cycle absorbant dans le graphe conjonctif-disjonctif.

Pour s'assurer qu'il n'est pas possible de rendre la solution réalisable, il suffit de remarquer qu'entre les opérations *O1*et *O3* se trouvent les opérations *O7* et *O8* de durée 5 et 5. L'intervalle de temps minimum entre *O1*et *O3* est donc de 10 unités de temps. Or les deux contraintes de time lag font que la durée maximum entre la fin de l'opération *O1* et le début de l'opération *O3* est 2+5+2=9. Il y a donc 10 unités de temps minimum entre les deux opérations et seulement 9 unités de temps sont permises par les time lags maximums. L'ordre des opérations n'est donc pas réalisable.

Les articles traitant des time lag maximum sont relativement peu nombreux. À notre connaissance, l'état de l'art suivant est représentatif.

Dès les problèmes à deux machines, les problèmes avec time lag maximum sont NP-difficiles : par exemple, le problème de flowshop à deux machines avec contraintes de time lag maximum est NP-difficile (Yang et Chern, 2003). Les problèmes d'openshop à deux machines avec des time lag maximum uniformes sont NP-complets (Rayward-Smith et Rebaine, 1992). Dans le cas où les opérations ont un temps de traitement unitaire, un algorithme polynomial est proposé par Rayward-Smith et Rebaine (1992). Dans les autres cas, le problème peut être résolu par la procédure de séparation / évaluation proposée par Yang et Chern (2003).

Pour les problèmes à plusieurs machines, peu de résultats sont disponibles. Essentiellement des algorithmes de séparation / évaluation sont envisagés, ces algorithmes sont particulièrement bien adaptés aux contraintes de time lag maximum car ils permettent de couper naturellement de grandes parties de l'arbre. Ainsi, Szwarc (1983) propose des bornes inférieures pour le problème de flowshop. Fondrevelle (2006) propose aussi bien des bornes inférieures pour le problème de flowshop que des algorithmes de séparation / évaluation.

Enfin, la plupart des problèmes étant NP-difficiles, des heuristiques sont proposées pour les résoudre. Deppner (2003) a proposé des heuristiques basées sur les règles de priorité adaptées à la contrainte de time lag. Par contre, l'auteur traite un problème général d'ordonnancement, et pour ce problème, on ne sait pas déterminer s'il existe une solution réalisable. C'est pourquoi ses heuristiques relâchent les contraintes si de trop nombreuses réparations sont nécessaires. Aucun résultat n'est disponible, ni dans la littérature, ni auprès de l'auteur pour le problème de jobshop. De plus, Su (2003) montre qu'un flowshop hybride à deux étages et une seule machine au deuxième étage est NP-difficile. L'auteur propose donc une heuristique pour ce cas particulier.

2.5 Conclusion sur l'état de l'art

Les contraintes de time lag généralisent de nombreuses autres contraintes. Pour présenter les différents types de contraintes de time lags et les contraintes qu'elles généralisent, nous proposons de les regrouper sous la forme d'arbre (cf. figure 3-5). En haut de cet arbre se trouve la contrainte de time lag dans toute sa généralité. En descendant dans cet arbre, on traite des problèmes de plus en plus particuliers. Les boîtes grisées dans la figure représentent les contraintes rencontrées dans la littérature : contrainte technologique, de précédence, nowait, dates au plus tôt, dates au plus tard, lead time. Plus on descend dans l'arbre et plus les contraintes sont particulières, ainsi un problème prenant en compte la contrainte de time lag dans le cas général sait en particulier prendre en compte les contraintes en dessous dans l'arbre.

La figure 3-5 montre que les contraintes de time lag minimum et maximum permettent de prendre en compte les contraintes nowait, les contraintes classiques de précédence et les contraintes technologiques. Les lignes et les colonnes du tableau sont fonctions des times lag maximum et minimum. Ces time lags sont exprimés à l'aide de coefficients appelés coefficient de time lags (cf.

définition page 109). Le tableau 3-1 dresse un récapitulatif des instances que l'on peut traiter en fonction des coefficients choisis de time lag minimum et maximum. Dans tous les cas, les coefficients du time lag minimum doivent être supérieurs à ceux du time lag maximum. S'ils sont identiques, on obtient des contraintes de délai fixe. À notre connaissance, ces contraintes ne sont pas étudiées dans la littérature, à part dans le cas où les deux coefficients sont nuls. Ce dernier cas correspond aux problèmes sans attente. Lorsqu'il n'y a que des contraintes de time lag minimum, on modélise des temps de transport, sauf si le temps minimum est nul auquel cas on modélise le problème classique de jobshop.

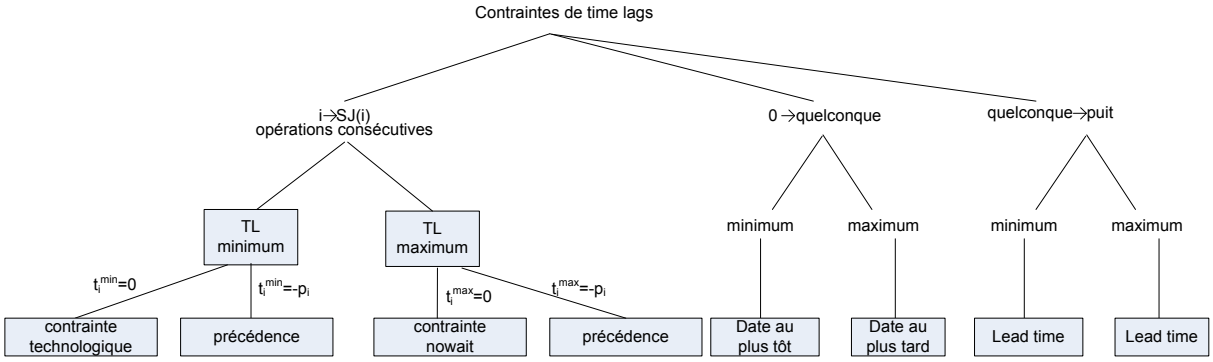

Figure 3-5. Typologie des contraintes de time lag

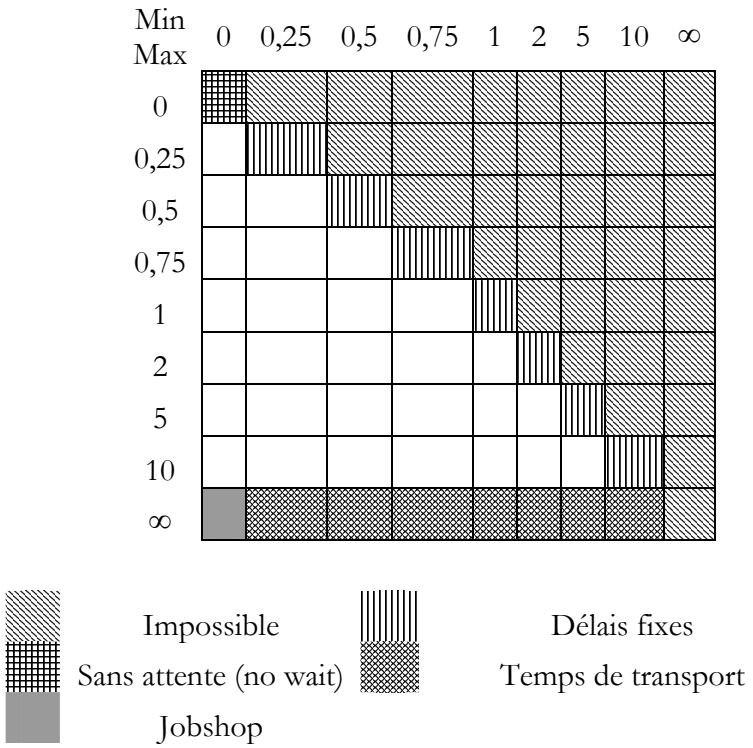

Tableau 3-1. Cas particuliers du jobshop avec time lags

Dans cette thèse, on s'intéresse particulièrement au problème de jobshop et à ses extensions. Ainsi, les deux problèmes avec lesquels on va pouvoir tester nos méthodes sont : le problème de jobshop classique (case pleine, problème présenté dans le chapitre 2) et le problème de jobshop sans attente (case quadrillée). Pour ce second problème, un état de l'art complet est présenté dans (Hall et Sriskandarajah, 1996).

Nous utilisons donc des résultats de la littérature du jobshop sans attente pour évaluer la qualité des solutions que nous proposons. On utilise en particulier les algorithmes proposés par Schuster et Framinan (2003). Ces auteurs proposent deux méthodes de recherche locale : VNS (Variable

Neighborhood Search) et GASA (Genetic Algorithm with Simulated Annealing). Ces deux méthodes de recherche locale sont basées sur une représentation efficace du problème de jobshop sans attente. Par contre, cette représentation est heuristique, i.e. elle peut ne pas être capable de coder la solution optimale. Mascis et Pacciarelli (2002) proposent des heuristiques basées sur les règles de priorité pour le problème de jobshop sans attente.

L'état de l'art montre que les contraintes de time lag rendent le problème difficile, que ce soit pour les time lags généraux, minimums ou maximums.

L'introduction des contraintes de time lag introduit les difficultés suivantes :

- le problème à une machine devient NP-difficile (Brucker *et al.*, 1999b)
- le problème à une machine, à temps de traitement unitaire et time lags entiers est NP-difficile (Finta et Liu, 1996).
- trouver une solution réalisable est NP-difficile (Brucker *et al.*, 1999a)

L'introduction des contraintes de time lag minimum ne change pas radicalement la difficulté des problèmes. Pourtant, les ordonnancements de permutation ne sont plus optimaux. (Mitten, 1958)

L'introduction des contraintes de time lag maximum introduit les difficultés suivantes :

- un ordonnancement partiel réalisable peut ne pas être complétable (Mascis et Pacciarelli, 2002),
- les problèmes de flowshop à deux machines sont NP-difficiles (Wikum *et al.*, 1994).

À notre connaissance, il n'y a pas dans la littérature d'algorithme trouvant systématiquement des solutions réalisables aux problèmes multi-machines avec contraintes de time lag maximum. Dans la suite de cette thèse, nous proposons plusieurs méthodes le permettant.

3 Difficulté de résolution des instances avec time lags

Dans cette partie, nous présentons des résultats préliminaires à l'élaboration d'une méthode d'optimisation. Ces résultats montrent que le problème de time lag est difficile et que sa prise en compte ne fait pas que modifier la valeur de la fonction à optimiser, elle nécessite aussi l'introduction de fonctionnalités spécifiquement dédiées à un espace de recherche comprenant de nombreuses solutions irréalisables.

Pour pouvoir réaliser des campagnes de tests, il est nécessaire de disposer d'instances sur lesquelles expérimenter. Dans la section 3.1, nous proposons une série d'instances pour le jobshop avec time lags. Les sections 3.2 et 3.3 sont deux expériences complémentaires : la section 3.2 montre la faible proportion d'ordres réalisables présents dans l'espace de recherche alors que la section 3.3 montre même pour les ordres réalisables, l'espace de recherche est fortement modifié par l'introduction des contraintes de time lags.

3.1 Instances[3]

Pour illustrer nos propos et évaluer nos algorithmes, nous avons construit des instances de jobshop avec time lag. Pour définir de telles instances, il faut préciser les mêmes données que pour le problème de jobshop et y adjoindre les constantes de temps des contraintes de time lags. Nous avons donc basé nos instances sur les instances de jobshop de la littérature.

[3] Instances téléchargeables à http://www.isima.fr/caumond/recherche/instances.html

On note *NOM_X_Y* une instance de jobshop avec time lag dont les données de jobshop sont issues de l'instance de jobshop appelée *NOM* et dont les coefficients *X* et *Y* permettent de calculer les constantes de temps des time lags. Le coefficient *X* concerne les time lags minimums et *Y* concerne les time lags maximums. La durée d'un time lag entre deux opérations consécutives d'un même job est calculée en multipliant le coefficient (*X* ou *Y*) par la moyenne des durées des opérations des jobs. Ainsi, tous les time lags minimums d'un job ont la même durée dont la valeur est *X*dm* où *dm* est la durée moyenne des opérations des jobs. Concrètement, les tableaux suivants détaillent l'exemple de l'instance ft06_1_2. Cette célèbre instance de Fischer et Thompson comporte 6 jobs passant sur 6 machines. Le tableau 3-2 décrit chaque job de l'instance, chaque ligne du tableau correspondant à un job. Une case du tableau correspond à une opération, une case contient deux valeurs, la première indique le numéro de la machine devant réaliser l'opération, alors que le second donne la durée de l'opération. Les tableaux suivants définissent les constantes de temps des time lags pour l'instance ft06_1_2. Le tableau 3-3 donne les time lags minimums de l'instance, le tableau 3-4 donne les time lags maximums.

Job 1	2 1	0 3	1 6	3 7	5 3	4 6
Job 2	1 8	2 5	4 10	5 10	0 10	3 4
Job 3	2 5	3 4	5 8	0 9	1 1	4 7
Job 4	1 5	0 5	2 5	3 3	4 8	5 9
Job 5	2 9	1 3	4 5	5 4	0 3	3 1
Job 6	1 3	3 3	5 9	0 10	4 4	2 1

Tableau 3-2. Données de l'instance ft06

4	4	4	4	4	4
7	7	7	7	7	7
5	5	5	5	5	5
5	5	5	5	5	5
4	4	4	4	4	4
5	5	5	5	5	5

Tableau 3-3. Time lags minimums de ft06

8	8	8	8	8	8
15	15	15	15	15	15
11	11	11	11	11	11
11	11	11	11	11	11
8	8	8	8	8	8
10	10	10	10	10	10

Tableau 3-4. Time lags maximum de ft06

Le tableau 3-5 présente la taille des principales instances de jobshop avec time lags. Le nombre de jobs d'une instance est noté *n*, le nombre de machines est noté *m*. Les tailles d'instances présentées ne dépendent pas des contraintes de time lag, en particulier elles ne dépendent pas des time lags maximaux. Pourtant, celles-ci interviennent dans la difficulté de résolution. Dans cette thèse, toutes les instances ont autant de time lags maximaux que de paires d'opérations successives dans la gamme.

Instances	la01-05	la06-10	la11-15	car5	car6	car7	car8	ft06
nxm	10x5	15x5	20x5	10x6	8x9	7x7	8x8	6x6

Tableau 3-5. Taille des instances

3.2 Proportions de solutions réalisables

Pour illustrer la difficulté des contraintes de time lag, nous proposons d'évaluer le nombre d'ordres réalisables (i.e. solutions réalisables).

Pour calculer la proportion d'ordres réalisables, nous considérons les trois types d'ordres suivants :

- Un ordre d'opérations en général définit l'ordre de passage des opérations sur les machines.
- Un ordre valide propose des ordres compatibles avec les contraintes technologiques. À chaque ordre valide correspond donc un ordonnancement semi-actif solution du problème de jobshop sans time lags.
- Un ordre "réalisable" correspond à une solution réalisable du problème avec time lags, un ordre réalisable est donc compatible avec les contraintes de time lags.

Pour évaluer le nombre d'ordres réalisables, on considère l'espace de recherche formé des ordres valides d'opérations dans lequel on compte la proportion de solutions réalisables. . Parmi ces ordres valides, on compte les ordres réalisables en vérifiant qu'ils respectent en plus les contraintes de time lags (minimum et maximum).

Le tableau 3-6 présente les résultats de cette procédure pour l'instance ft06 et différentes valeurs du coefficient de time lag maximum. Les problèmes avec time lags maximum à 100 ont des contraintes de time lags maximum tellement lâches qu'ils sont identiques au problème de jobshop. Pour des time lags maximums inférieurs à *10*, le nombre de points réalisables décroît rapidement. À partir de *5*, il est 200 fois plus faible que pour des time lags infinis.

Coefficient de time lag maximum	**Taux de solutions réalisables**
0	$<10^{-7}$
1	$<10^{-7}$
2	$<10^{-7}$
3	$<10^{-7}$
5	10^{-7}
7	10^{-6}
10	$4{,}3.\ 10^{-6}$
100	$1{,}95.\ 10^{-6}$
$+\infty$	$2.\ 10^{-6}$

Tableau 3-6. Nombre de solutions admissibles pour l'instance ft06

3.3 Valeur de time lag maximum et makespan optimal

Dans ce paragraphe, nous mettons en évidence que l'introduction des time lags maximum modifie aussi les ordres d'opérations réalisables. Ainsi, le makespan des solutions associées à ces ordres est fortement modifié comme nous le montrons dans le tableau 3-7. Ce tableau est organisé comme le tableau 3-1, suivant les valeurs des coefficients de time lag minimum et maximum. Il représente donc des problèmes de jobshop, jobshop sans attente, jobshop avec temps de transport ou jobshop avec délai fixe.

Puisque l'on travaille sur un problème de minimisation, toute contrainte supplémentaire ne peut qu'augmenter la valeur du makespan optimal. Ainsi, la valeur du makespan optimal pour le problème de jobshop est plus faible que celle pour le problème de jobshop avec time lag. De même, un ordonnancement vérifiant les contraintes de time lag maximum pour une valeur de coefficient est réalisable pour une valeur supérieure. C'est pourquoi la valeur du makespan optimal diminue quand les coefficients de time lag maximum augmentent.

De même, un ordonnancement vérifiant les contraintes de time lag minimum pour une valeur de coefficient est réalisable pour une valeur inférieure. C'est pourquoi la valeur du makespan optimal diminue quand les coefficients de time lag maximum diminuent.

Le tableau 3-7 confirme ces observations. Il contient les valeurs optimales du makespan pour l'instance ft06 avec des coefficients de time lag variés. Ces valeurs optimales sont calculées par résolution directe du programme linéaire présenté dans la section suivante.

Min Max	**0**	**0,25**	**0,5**	**0,75**	**1**	**2**	**5**	**10**	**∞**
0	94								
0,25	91	105							
0,5	79	94	109						
0,75	77	84	99	119					
1	61	79	88	117	136				
2	55	60	66	77	83	186			
5	55	58	63	73	83	122	294		
10	55	58	63	73	83	122	242	437	
∞	5	58	63	73	83	122	242	437	

Tableau 3-7. Influence des time lags sur les solutions optimales de ft06

Le plus faible makespan optimal est obtenu pour le problème classique de jobshop dont le time lag minimum est nul et dont le time lag maximum est infini. Ensuite, plus on diminue le time lag maximum, et plus on augmente le time lag minimum et plus la valeur du makespan optimal augmente. La valeur 55 obtenue pour le problème sans contrainte de time lag est donc une borne inférieure pour tous les autres problèmes. Par contre, la qualité de cette borne décroît rapidement. L'écart entre le cas nowait et le cas classique est de 70,9%. Cela signifie que prendre la solution optimale du problème de jobshop classique comme borne inférieure de la solution optimale est dramatiquement mauvais. Pourtant, mis à part le problème nowait et le problème classique, nous n'avons pas d'autres valeurs de référence pour le jobshop avec time lag.

3.4 Time lag maximum et voisinages

Dans les deux paragraphes précédents, nous avons montré que la prise en compte des time lags maximum diminuait fortement le nombre de solutions réalisable et que les solutions optimales cherchées étaient elles-aussi influencées par les constantes des contraintes de time lags.

Dans cette thèse, nous envisageons la résolution des problèmes avec contraintes de time lags avec des méthodes essentiellement basées sur les méthodes de voisinages. Nous proposons aussi des méthodes de construction. Ces deux types de méthode sont particulièrement difficiles à mettre en œuvre à cause de la présence de time lags.

Durant toutes nos expérimentations, nous avons observés que les solutions réalisables sont très différentes les unes des autres. En particulier, nous n'avons pas trouvé de voisinage adapté à la résolution des time lags. Les voisinages habituels (insertion, permutation) souffrent en effet tous du même défaut : dès que les constantes de time lags diminuent, le nombre de voisins réalisables décroît fortement jusqu'à être quasiment systématiquement nul.

3.5 Synthèse

Le tableau 3-6 donne la proportion de solutions réalisables, ce qui constitue un indicateur pour déterminer si le problème est très contraint ou non. Le tableau 3-7 fournit le makespan de la solution optimale, ce qui permet de mesurer l'influence des time lags sur la solution optimale.

En examinant conjointement ces deux tableaux de résultats, on peut se rendre compte de l'influence des time lags maximum sur la difficulté de résolution. Tout d'abord, la difficulté d'un problème ne se mesure pas uniquement sur la valeur du makespan de la solution optimale. En effet, la dernière ligne du tableau représente le problème de jobshop classique car le time lag maximum y est infini. Par contre, lorsque le time lag maximum diminue, le problème devient de plus en plus contraint comme le montre le tableau 3-6. Pourtant, les valeurs de makespan optimaux restent constantes relativement longtemps. Par exemple, pour les problèmes dont le time lag minimum est compris entre *0* et *2*, l'influence du time lag maximum ne se voit pas sur la valeur du makespan optimal. Or, entre le problème classique et le problème avec un coefficient de *5*, le nombre de solutions réalisables a été divisé par *20*. Cette observation montre que même si le nombre de points dans l'espace de recherche diminue, les solutions qui sont supprimées ne sont pas forcément les meilleures solutions. Ceci peut s'expliquer par le fait que les contraintes de time lag maximum sont plus facilement violées sur un ordonnancement de grand makespan que sur un ordonnancement du même problème avec un makespan plus faible.

4 Formalisation linéaire

Dans la section 2, nous avons présenté une formalisation mathématique du problème de jobshop avec contraintes de time lags générales. Ci-dessous, nous présentons la formalisation linéaire pour le cas particulier du problème de jobshop avec contraintes de time lag minimum et maximum.

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

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

(1) $(H + p_j)a_{ij} + (t_j - t_i) \geq p_j \quad \forall (i,j) \in E_k, \forall k \in M,$

(2) $(H + p_j)(1 - a_{ij}) + (t_j - t_i) \leq p_i \quad \forall (i,j) \in E_k, \forall k \in M,$

(3) $t_i + p_i + t_i^{\min} \leq t_{SJ(i)} \quad \forall (i, j = SJ(i)) \in A,$

(4) $t_{SJ(i)} \geq t_i + p_i + t_i^{\max} \quad \forall (i, j = SJ(i)) \in A,$

(5) $t_i \geq 0 \quad \forall i \in O,$

(6) $a_{ij} \in \{0;1\} \quad \forall (i,j) \in E_k, \forall k \in M,$

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

Nous détaillons chaque paquet de contraintes de cette formalisation linéaire.

- Les paquets de contraintes (1) et (2) modélisent la disjonction sur les machines : une seule opération ne peut être réalisée à la fois sur la machine. Pour une machine k donnée, et pour deux opérations i et j en disjonction sur cette machine, les contraintes expriment soit que l'opération i est avant l'opération j soit qu'elles sont dans l'ordre inverse.
- Le paquet de contraintes (3) représente la contrainte de time lag minimum. Cette contrainte généralise la contrainte technologique habituellement présente pour le problème de jobshop. En effet, si on prend $t_i^{\min} = 0$ la contrainte devient exactement identique à la contrainte technologique.

- Le paquet de contraintes (4) concerne la contrainte de time lag maximum.
- Les contraintes (5) et (6) donnent le domaine de définition des variables. Les variables a_{ij} sont binaires, elles modélisent la disjonction des opérations. Les variables t_i sont simplement définies comme étant positives.
- La contrainte (7) permet de calculer le makespan, qui est le critère d'optimisation retenu.

Dans ce programme linéaire, il n'est pas indiqué que les variables t_i sont entières. Ceci n'est pas nécessaire car toute solution calculée sans l'intégrité des variables t_i peut être transformée en une solution dont les dates de début sont entières. En effet, il suffit d'arrondir les dates de début des opérations à l'entier inférieur. On obtient alors une solution entière de même makespan.

On pourrait vouloir préciser l'intégrité des variables t_i aux solveurs de programmes linéaires. Pourtant, cette information supplémentaire n'aide pas les solveurs : ils interprètent l'intégrité des variables t_i comme une contrainte supplémentaire. Or les contraintes d'intégrité sont difficiles à prendre en compte, et les solveurs en sont généralement perturbés.

Le modèle linéaire proposé permet de résoudre de manière optimale des instances de jobshop avec time lags. Pourtant, la taille des problèmes résolus est relativement modérée et nous proposons donc des modèles permettant de résoudre efficacement et de façon approchée des instances de taille supérieure.

5 Modèle de graphe conjonctif - disjonctif

Comme nous l'avons indiqué dans le chapitre 2, le modèle de graphe conjonctif-disjonctif est utilisé dans de nombreuses méthodes efficaces pour le problème de jobshop. Puisque nous nous intéressons à une extension du problème de jobshop, il paraît naturel de vouloir étendre ce modèle pour qu'il prenne en compte les contraintes de time lags. Nous allons apporter deux modifications principales sur le graphe : la modification de la longueur d'un type d'arcs et l'ajout d'arcs de longueurs négatives. À cause des éventuels cycles de longueur négative, l'algorithme de plus longs chemins proposé pour le problème de jobshop doit être modifié.

5.1 Construction

L'ajout des contraintes de time lag modifie le modèle dès la construction du graphe conjonctif - disjonctif. Les modifications sont de deux types : tout d'abord la longueur des arcs entre opérations successives du même job est modifiée, puis des arcs sont ajoutés pour modéliser la contrainte de time lag maximum.

Dans le graphe conjonctif - disjonctif, on modifie la longueur des arcs entre opérations successives. En effet, pour le problème de jobshop, un arc entre l'opération i et l'opération $SJ(i)$ a une longueur p_i; à cause des contraintes de time lag minimum, cette longueur passe à $p_i + t^i_{\min}$. L'arc ainsi obtenu modélise la contrainte $t_i + p_i + t_i^{\min} \le t_{SJ(i)}$.

De plus, la contrainte de time lag maximum ($t_{SJ(i)} \ge t_i + t_i^{\max} + p_i$) est modélisée par un arc de l'opération $SJ(i)$ vers l'opération i. La longueur de cet arc est $-(p_i + t_i^{\max})$. Cet arc a deux particularités : sa longueur est négative et il forme des cycles.

Nous construisons un exemple complet pour illustrer ce nouveau graphe. Le tableau 3-8 rappelle les données de l'instance J1 de jobshop. Le tableau 3-9 donne les durées des time lags minimum et maximum pour cette instance. La figure 3-6 présente le graphe conjonctif - disjonctif ainsi

obtenu. Nous ne rappelons pas la démarche pour construire le graphe complet, celle-ci se trouve dans le chapitre 2 [4].

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 3-8. Instance exemple J1

Job 1	O1, O2 : (O,1)	O2, O3 : (1,12)
Job 2	O4, O5 : (0,2)	O5,O6 : (1,10)
Job 3	O7, O8 : (0,4)	O8,O9 : (0,10)

Tableau 3-9. Time lag pour J1 ($t^i_{\min}, t^i_{\max}$)

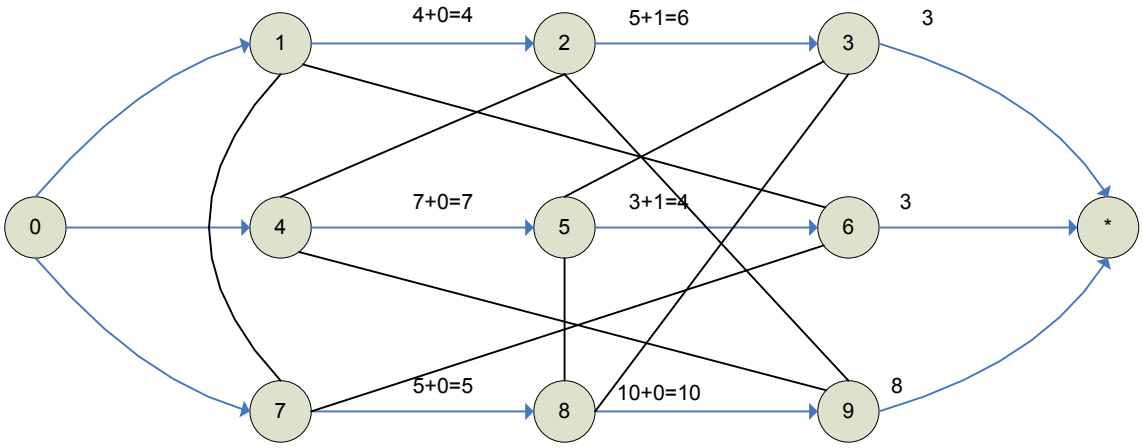

Figure 3-6. Graphe conjonctif - disjonctif du problème J1 avec time lags minimum

Le graphe conjonctif - disjonctif est la première étape du modèle, la seconde consiste à construire le graphe conjonctif à partir d'une sélection. Pour construire le graphe conjonctif, on oriente les arêtes du graphe conjonctif - disjonctif. Cette orientation pour le problème de jobshop avec time lag n'est pas détaillée ici car elle ne comporte aucune spécificité et est conforme aux explications du chapitre 2 [5]. Dans la figure 3-7, nous présentons le graphe conjonctif associé à l'ordre du tableau 3-10. Ce graphe comporte de nombreux cycles.

Chaque contrainte de time lag maximum crée un cycle car il y a deux arcs entre deux opérations consécutives de la même gamme : un pour la contrainte de time lag minimum et l'autre pour la contrainte de time lag maximum. Par contre, les cycles ainsi formés sont tous de longueur négative

[4] chapitre 2, section 4.4.1, page 77

[5] chapitre 2, section 0, page 78

pour peu que le time lag maximum soit supérieur au time lag minimum (sans quoi il n'existe aucune solution au problème).

D'autres cycles apparaissent dans ce graphe comme le cycle $(8 \xrightarrow{10} 3 \xrightarrow{-17} 2 \xrightarrow{-5} 1 \xrightarrow{4} 7 \xrightarrow{5} 8)$ de longueur -3. De tels cycles sont moins triviaux à détecter et dépendent de l'ordre des opérations sur les machines. Ce cycle est aussi de longueur négative.

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

Tableau 3-10. Ordre S1 de l'instance J1

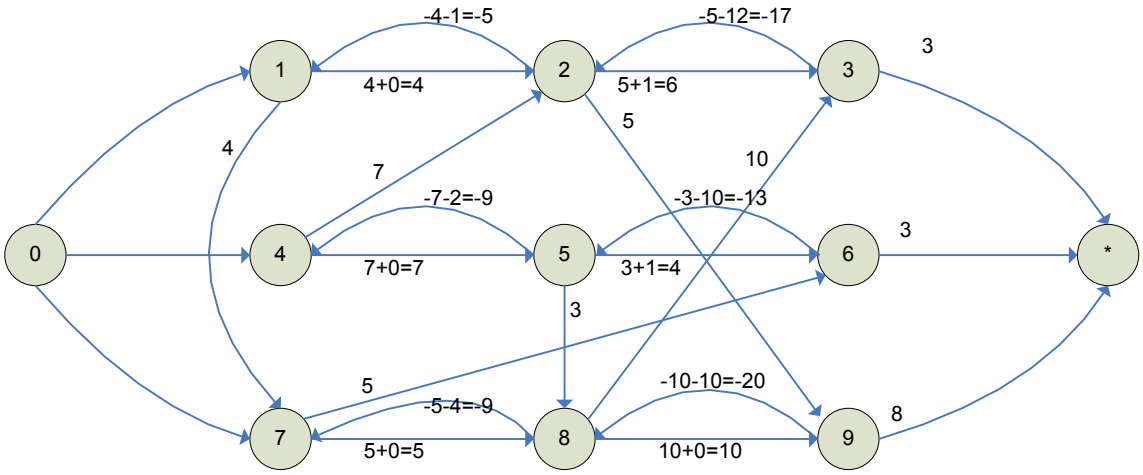

Figure 3-7. Graphe conjonctif du problème J1 avec time lags minimum et maximum

5.2 Plus longs chemins dans le graphe

Dans ce paragraphe, nous proposons un algorithme de plus longs chemins efficace pour le graphe conjonctif-disjonctif du problème de jobshop avec time lags. L'algorithme proposé est une adaptation de l'algorithme de Ford-Bellmann, les deux algorithmes ont une performance dans le pire des cas identiques, mais l'algorithme que nous proposons a un meilleur comportement en moyenne et permet une implantation informatique très proche de l'algorithme de plus longs chemins dédié au problème de jobshop.

Lorsque le graphe conjonctif est construit, on souhaite alors lui appliquer un algorithme de plus longs chemins pour calculer les dates de début des opérations. Pour le problème de jobshop avec time lags, le graphe conjonctif contient à priori des cycles absorbants. Il faut donc utiliser un algorithme détectant ces cycles. Dans le chapitre 2, nous avons proposé un algorithme de plus long chemin compatible avec les caractéristiques du graphe conjonctif du jobshop avec time lag. Cet algorithme a une complexité dans le pire des cas en $O(nm)$ (où n est le nombre d'arcs et m est le nombre de nœuds). Les algorithmes de plus long chemin de complexité inférieure ne sont pas applicables à cause de la structure du graphe (qui contient des cycles et des arcs de longueur négative). Pourtant, nous proposons un algorithme de plus long chemin efficace pour le problème de jobshop avec time lags. Cet algorithme a la même complexité dans le pire des cas, mais est plus performant.

L'algorithme proposé ci-dessous est basé sur l'algorithme de Ford-Bellmann. Deux améliorations principales sont apportées : la première concerne l'ordre dans lequel on visite les opérations, la seconde concerne la détection de cycles.

La première amélioration consiste à imposer un ordre de parcours des opérations. Dans l'algorithme général, aucun ordre de parcours des opérations n'est imposé, pourtant le choix de cet ordre est primordial. Par exemple, parcourir les nœuds du graphe dans un ordre topologique permet de ne faire qu'un seul parcours d'un graphe acyclique. Dans le même graphe, tout ordre non topologique nécessite plusieurs parcours du graphe. Dans le pire des cas, il peut être nécessaire de faire (*m-1*) parcours du graphe (où m est le nombre de nœud du graphe).

Puisque les graphes conjonctifs du problème de jobshop avec time lag sont cycliques, il n'est pas possible de trouver un ordre topologique. Par contre, nous proposons un ordre des opérations compatibles avec un sous-ensemble des arcs : on omet les contraintes de time lag maximum, le sous-graphe conjonctif obtenu est acyclique pour peu que la sélection soit valide. En effet, ce graphe représente le problème de jobshop avec time lag minimum et pour ce problème, toute sélection valide d'opérations est réalisable. On peut donc dire que le graphe contient des cycles "à cause" des time lags maximums. Par conséquent, en omettant ces contraintes, on peut calculer un ordre topologique. Nous proposons d'utiliser cet ordre pour parcourir les opérations. Ci-dessous, dans la description de l'algorithme proposé, nous verrons que cet ordre accélère la mise à jour.

La deuxième amélioration proposée consiste à détecter les cycles avant la dernière itération, et ceci sur un nombre limité d'arcs. La stratégie utilisée pour découvrir les cycles dans algorithme 2-1 n'est pas adaptée si la probabilité d'avoir un cyclique absorbant est non négligeable. Une seconde stratégie consiste à utiliser le tableau *pere* pour vérifier que l'on n'introduit pas de cycle absorbant. Cette technique est relativement lourde si elle doit être appliquée à toutes les mises à jour d'arcs. Mais pour le problème de jobshop avec time lag, on sait que tous les cycles passent par au moins un arc de time lag maximum. Grâce à cette remarque et à la structure de l'algorithme, nous allons pouvoir limiter la recherche de cycles aux mises à jour d'arcs de time lag maximum.

Fort de l'algorithme de Bellman-Ford et des deux améliorations proposées, nous construisons l'algorithme de plus longs chemins adapté aux problèmes de jobshop avec time lag (algorithme 3-1). Cet algorithme consiste à parcourir les nœuds dans l'ordre T et à faire les mises à jour des marques (cf. boucle interne). Le corps de ce parcours ressemble donc à l'algorithme général de Bellman-Ford, sauf que seule la mise à jour de l'arc de time lag maximum entraîne une nouvelle itération. Le booléen "fin" n'est donc positionné que lors de la mise à jour de cet arc. S'il n'y a pas besoin de le positionner pour les deux arcs précédents, c'est grâce à l'ordre de parcours des opérations. En effet, les nœuds aux extrémités des arcs disjonctifs (nœud $SM(i)$) et de time lag minimum (nœud $SJ(i)$) apparaissent plus tard dans l'ordre T utilisé. Toutes les mises à jour concernant ces arcs seront donc correctement réalisées.

Les cycles absorbants ne sont détectés que pendant la mise à jour de l'arc de time lag maximum. Pendant cette mise à jour, on sait que l'arc de l'opération courante i vers l'opération précédente dans la gamme $PJ(i)$ va devenir critique. La détection du cycle consiste à vérifier que le sous-graphe critique ne contient pas de chemin de $PJ(i)$ vers i. Si tel est le cas, c'est qu'il existe un cycle absorbant dans le graphe. La figure 3-8 résume la situation. Il y a un chemin de longueur L entre les opérations $PJ(i)$ et i. L'arc de time lag maximum est mis à jour, les dates courantes des opérations vérifient donc l'inéquation suivante : $t_{PJ(i)} + p_{PJ(i)} + t^{\max}_{PJ(i)} < t_i$. La longueur du cycle est $L = t_i - t_{PJ(i)}$, on peut donc réécrire l'inéquation en $L > p_{PJ(i)} + t^{\max}_{PJ(i)}$. Or, le chemin de $PJ(i)$ vers i complété de l'arc de i vers $PJ(i)$ est un cycle de longueur $L - p_{PJ(i)} - t^{\max}_{PJ(i)}$. Donc, d'après l'inéquation ci-dessus, la longueur du cycle vérifie donc $L - p_{PJ(i)} - t^{\max}_{PJ(i)} > 0$. Le cycle mis en évidence est donc forcément un cycle absorbant.

De plus, nous devons montrer que cet algorithme détecte tous les cycles absorbants. Pour cela, nous allons considérer un cycle quelconque et montrer qu'il est forcément détecté. Soit un cycle absorbant dans le graphe conjonctif, ce cycle possède au moins un arc de time lag maximum. Pendant la première itération, l'arc de time lag maximum a forcément conduit à une mise à jour (car tous les nœuds du cycle absorbant peuvent être indéfiniment augmenté). Ainsi, une deuxième itération commence forcément et va elle aussi mener à une mise à jour. Puisque le graphe contient un cycle absorbant, les mises à jour sont infinies. D'itération en itération, tous les nœuds du cycle sont poussés. Au bout d'un certain nombre d'itérations, toutes les opérations du cycle auront pour opération "pere" une autre opération du cycle. Quand tel est le cas, c'est que les arcs du tableau "pere" forment un cycle. L'algorithme proposé peut alors le détecter. D'un point de vue algorithmique, lorsqu'un arc de time lag maximum est mis à jour, il suffit de parcourir le tableau "pere", de précédent en précédent en partant

du nœud de départ de l'arc de time lag maximum. Deux cas sont alors possibles, soit on arrive jusqu'au nœud 0 et aucun cycle n'est détecté, soit on arrive vers le nœud d'arrivée de l'arc de time lag maximum. Dans ce deuxième cas, on est sur le point de créer le cycle dans le tableau "pere" et l'on peut stopper l'algorithme de plus longs chemins.

```
T                              // Ordre d'opérations
   Initialisation
t_i = 0, ∀i ∈ O ;               // Initialise les dates de début
pere_i = 0, ∀i ∈ O ;        // Initialise le tableau des pères
fin = faux;
cycle = faux;
Tant que non fin et non cycle faire
   fin = vrai;
   Pour i de 0 à * dans l'ordre T faire
         Si  t_i + p_i > t_SM(i)  alors          // Disjonction
               t_SM(i) = t_i + p_i ;
               pere_SM(i) = i ;
         finsi
         Si  t_i + p_i + t_i^min > t_SJ(i)  alors      // Time lag minimum
               t_SJ(i) = t_i + p_i + t_i^min;
               pere_SJ(i) = i
         finsi
         Si  t_PJ(i) + p_PJ(i) + t_PJ(i)^max < t_i alors      // Time lag maximum
               t_PJ(i) = t_i − p_PJ(i) − t_PJ(i)^max ;
               pere_SJ(i)=i;
               fin = faux;
               // Détection des cycles
               k=i;
               Tant que k≠0 et k≠PJ(i) faire
                     k =  pere_k ;
               fintantque
               si k=PJ(i) alors
                     cycle = vrai;
               sinon
                     cycle = vrai;
               sinon
         finsi
   fintantque
fintantque
```

Algorithme 3-1. Algorithme de plus longs chemins adapté au jobshop avec time lag

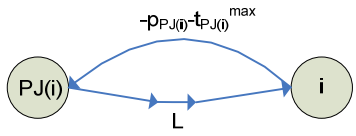

Figure 3-8. Détection de cycle

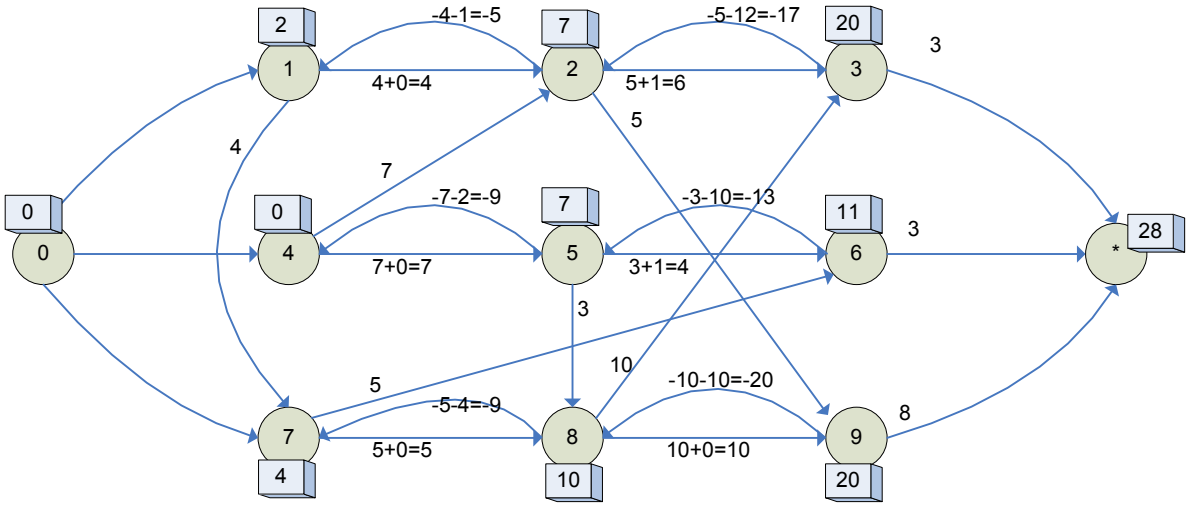

Figure 3-9. Première itération de l'algorithme

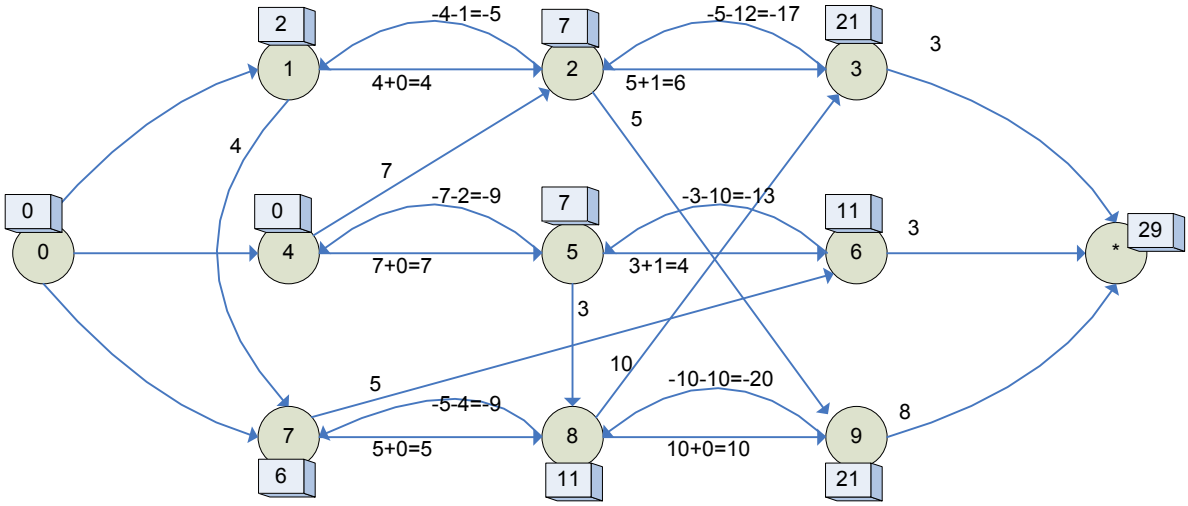

Figure 3-10. Deuxième itération de l'algorithme

Nous appliquons cet algorithme à l'exemple J1 de jobshop avec time lag. La première étape consiste à déterminer un ordre topologique des opérations. Parmi les plusieurs ordres possibles, nous retenons l'ordre *1, 4, 7, 2, 5, 6, 8, 9, 3*. La première itération de l'algorithme débute alors. Les marques obtenues à la fin de cette itération sont présentées sur le graphe de la figure 3-9. Les marques du graphe sont données dans des boîtes au-dessus des nœuds. Les arcs en gras représentent les arcs du sous-graphe critique calculé à partir du tableau *pere*. Tous les arcs en gras ont donc conduit à une amélioration, c'est le cas de l'arc de time lag maximum de *7* vers *2*. Puisqu'un arc de time lag maximum a conduit à une amélioration, c'est qu'une seconde itération de l'algorithme est nécessaire. On peut vérifier que les marques du graphe ne sont pas à jour : l'arc de 1 vers 7 peut en effet conduire à une amélioration. Une deuxième itération est donc nécessaire, le résultat de cette seconde itération est

présenté dans la figure 3-10. Pendant l'exécution de cette seconde itération, l'arc de 1 vers 7 est mis à jour. La nouvelle date de l'opération 1 modifie de nombreuses opérations ainsi que le makespan de l'ordonnancement.

6 Propositions de module d'évaluations

Le modèle de graphe présenté ci-dessus permet d'évaluer un ordre d'opérations pour le problème de jobshop avec time lag. En utilisant ce modèle, on peut construire plusieurs modules d'évaluation. Les sections 6.1, 6.2, 6.3 et 6.4 et présentent quatre modules d'évaluation basés sur le modèle de graphe conjonctif - disjonctif. Tous les modules génèrent des ordonnancements semi-actifs et tous utilisent le graphe conjonctif-disjonctif. La différence entre ces modules est dans la façon de prendre en compte les solutions irréalisables.

6.1 Module d'évaluation 1 : Représentation semi-active

Le premier module d'évaluation est basé sur la représentation semi-active. Cette représentation est la même que celle présentée pour le problème de jobshop classique dans le paragraphe 4.3.2 : les solutions stockées, les solutions codées, les propriétés de sur représentation, de multi représentation et heuristique sont les mêmes pour les deux représentations. La différence réside dans le décodage d'une solution : chaque décodage utilise le graphe conjonctif - disjonctif qui lui est propre.

Les deux représentations codées sont identiques, ce sont les ordonnancements semi-actifs. Pourtant, le nombre d'ordonnancements semi-actifs n'est pas du tout identique dans les deux cas. Il y a beaucoup moins d'ordonnancements semi-actifs pour le problème de jobshop avec time lag (cf. §3.2). Ceci est d'autant plus vrai pour les problèmes avec des time lags serrés (i.e. time lags dont les constantes de temps sont faibles relativement au temps de traitement). Ceci rend cette représentation difficilement utilisable pour les problèmes avec des time lags serrés.

Exemple de solutions

M1	1	7	6
M2	4	2	9
M3	5	8	3

Tableau 3-11. Exemple de représentation semi-active

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 ainsi que des ordres incompatibles avec les contraintes de time lags maximum. 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.

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 ou les contraintes de time lag maximum.
- 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.

6.2 Module d'évaluation 2 : Espace stratifié

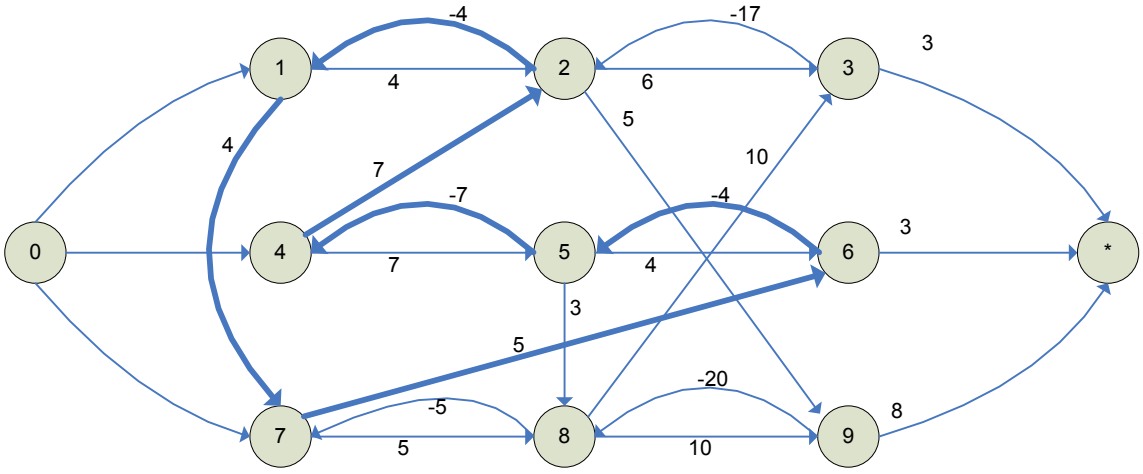

Figure 3-11. Graphe avec cycle

Ce module d'évaluation est basé sur une adaptation de la représentation semi-active. La représentation de ce module d'évaluation élargit l'ensemble des solutions codées en y incorporant celles qui violent les contraintes de time lag maximum. Pour cela, l'évaluation d'une solution consiste à déterminer le nombre de time lags qui doivent être enlevés pour rendre la solution réalisable.

Cette mesure n'est pas précise dans le sens où le nombre de time lags à enlever dépend des time lags qui sont effectivement supprimés. La figure 3-11 présente un exemple de graphe conjonctif comportant des cycles. Nous montrons que le nombre d'arcs à supprimer varie suivant ceux que l'on supprime. Dans ce graphe, il y a deux cycles. Le cycle $1 \to 7 \to 6 \to 5 \to 4 \to 2 \to 1$ contient deux time lags maximums $6 \to 5$, $5 \to 4$ et $2 \to 1$. Le cycle $6 \to 5 \to 8 \to 7 \to 6$ contient deux time lags maximums $6 \to 5$, $8 \to 7$. Il y a donc trois arcs différents de time lag maximum qui forment des cycles. En supprimant ces trois arcs, on obtient un graphe sans cycle absorbant. Pourtant, il suffit de supprimer l'arc $6 \to 5$ pour que les deux cycles absorbants soient cassés et que la solution soit réalisable. Suivant les arcs supprimés, il y a donc soit un ou trois time lags à supprimer. Le nombre de time lags à supprimer n'est donc pas un nombre absolu, mais dépend de la manière dont il est calculé.

Nous proposons un algorithme pour calculer un nombre de time lags à supprimer. Nous avons envisagé deux stratégies pour calculer ce nombre. Dans la première stratégie, on commence à envisager tous les time lags puis on les supprime un à un tant qu'ils créent des cycles. Dans la seconde stratégie, on commence à envisager le graphe conjonctif sans time lag maximum, puis on les ajoute un à un. Chaque ajout est suivi du retrait du time lag si celui-ci conduit à un cycle.

Entre ces deux stratégies, nous choisissons la deuxième car elle est plus performante et que le nombre de time lag à supprimer est mieux évalué. En effet, dans la seconde stratégie, chaque itération consiste en un algorithme de plus long chemin, et l'itération suivante consiste en un autre plus long chemin dans le graphe avec un arc supplémentaire. Cette seconde exécution du plus long chemin peut être accélérée en repartant des marques précédentes. Ainsi, on diminue le nombre d'itérations de l'algorithme. De plus, seul l'arc de time lag inséré peut conduire à une amélioration; c'est pourquoi on utilise une implantation basée sur les tas. Dans cette implantation, on initialise le tas avec l'opération au début de l'arc inséré. Si l'arc ne permet pas une amélioration, l'algorithme s'arrête. Si l'arc permet une amélioration, celle-ci est réalisée puis propagée dans le graphe.

```
Evaluer le plus long chemin sans time lag maximum
Conserver les marques
Pour chaque time lag i faire
   Insérer le time lag i
   Lancer l'algorithme des plus longs chemins en repartant des marques M
   Si un cycle est détecté alors
         supprimer le cycle i
   sinon
         Conserver les marques dans M
   finsi
finpour
```

Algorithme 3-2. Détermination du nombre de time lag à supprimer

Lorsque l'algorithme de détermination du nombre de time lag est terminé, on peut calculer trois critères : le makespan (*M*) de l'ordonnancement obtenu, le nombre de time lags supprimés (*S*), et une mesure de la violation des contraintes (*V*). Ce dernier critère consiste à mesurer de combien le time lag maximum devrait être augmenté pour que la contrainte soit vérifiée. Cette valeur se calcule de la manière suivante, pour toute contrainte de time lag supprimée entre deux opérations i et $PJ(i)$, on calcule $t_{SJ(i)} - (t_i + p_i + t_i^{\max})$. La mesure V est la somme de tous les $t_{SJ(i)} - (t_i + p_i + t_i^{\max})$ pour toutes les contraintes supprimées.

Nous proposons d'agréger ces trois critères par pondération : $\alpha M + \beta V + \gamma S$. Parmi ces trois critères, seul *M* est un critère du problème de départ. Les critères *V* et *S* servent à évaluer la réalisabilité d'une solution : ces deux critères sont nuls pour une solution réalisable et leur valeur est d'autant plus grande que la solution est irréalisable.

On propose donc de hiérarchiser les solutions en privilégiant la minimisation du critère *S* puis la minimisation du critère *V* et enfin la minimisation du critère *M*. On choisit donc les valeurs adéquates pour les coefficients α, β et γ. On prend $\alpha = 1$. β est choisi pour être supérieur à la plus grande valeur possible du makespan M. Le coefficient γ est choisi pour être supérieur à β fois la plus grande valeur possible de dépassement V. Ainsi, lors de la comparaison de deux solutions, la meilleure est celle qui compte le plus faible nombre de time lag supprimé. En cas d'égalité, on retient celle qui compte la plus faible violation de contrainte. Et en cas d'égalité, on retient la solution de plus faible makespan. Si ces trois critères sont identiques, les solutions sont évaluées comme étant identiques.

Exemple de solutions

M1	1	7	6
M2	4	2	9
M3	5	8	3

Tableau 3-12. Exemple de représentation stratifiée

Le graphe conjonctif correspondant à cet exemple est présenté dans la figure 3-11.

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 gammes. 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 tous les ordres semi-actifs plus les ordonnancements semi-actifs dans lesquels certains time lags sont supprimés.

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.

6.3 Module d'évaluation 3 : Évaluation par règles de priorité

Nous proposons une heuristique dédiée au problème de jobshop avec time lag. Cette heuristique est basée sur l'algorithme de Giffler et Thompson (1960). Pour la décrire, nous procédons en deux étapes. Tout d'abord, nous formalisons les algorithmes basés sur les règles de priorité (algorithme de la famille de Giffler et Thompson). Puis nous utilisons cette formalisation pour décrire l'heuristique dédiée au jobshop avec time lag.

```
   Initialisation
1) U=0                  // Opérations non ordonnancées
2) S=∅                  // Ordonnancement partiel
3) Tant que U≠∅ faire
4)       E := Eligible(U,S);     // Opérations éligibles
5)       t_i :=Evaluate(S,i); ∀i∈E // Evalue les dates de début des opérations
6)       F := Restriction(E,U,S) // F est un sous ensemble de E
7)       o := Rule(F)            // Choisit une opération
8)       S := Update(S,o)  // Ajoute l'opération o à S
9)       U := U-{o}        // Supprime l'opération o de U
10) fintantque
```

Algorithme 3-3. Formalisation des algorithmes basés sur les règles de priorité

L'algorithme ci-dessus est un pseudo-algorithme décrivant les algorithmes basés sur les règles de priorité. Dans ce pseudo-algorithme, on complète un ordonnancement partiel itération après itération. D'abord, les ensembles U (ensemble des opérations non ordonnancées) et S (ordonnancement partiel) sont initialisés de manière à représenter l'ordonnancement vide initial (lignes 1 et 2). L'ordonnancement est alors construit de manière itérative par la boucle principale (lignes 3 à 10). Chaque itération consiste à ordonnancer une nouvelle opération choisie par une règle de priorité. Les procédures *Eligible*, *Evaluate*, *Restriction*, *Rule* et *Update* sont appelées, mais leur contenu n'est pas explicitement défini par le pseudo-algorithme car leur contenu dépend du problème.

L'itération commence (ligne 4) par construire l'ensemble E des opérations éligibles. Une opération est éligible pour l'ordonnancement partiel S si c'est une opération non ordonnancée ($S \subset E$) et qu'il est possible de commencer cette opération sans violer aucune contrainte. L'ensemble des opérations éligibles S est donc calculé à partir des ensembles U et S. La procédure *Eligible* a donc deux paramètres U et S, elle retourne l'ensemble des opérations éligibles.

La seconde étape consiste à évaluer les dates de début au plus tôt des opérations éligibles (ligne 5). Bien entendu ce calcul dépend largement du problème étudié, et la procédure de calcul *Evaluate* ne peut être décrite de manière générale. Par contre, pour la plupart des problèmes, il est utile de connaître un ordonnancement valide sans la contrainte. C'est pourquoi la procédure *Evaluate* prend comme paramètre l'ordonnancement *S*. On peut en général repartir des dates de *S* pour calculer l'ordonnancement qui inclut la nouvelle opération. Suivant les cas, ce nouvel ordonnancement peut conserver les dates de l'ordonnancement *S* ou avoir besoin de les modifier. Typiquement, dans le problème de jobshop classique, on ne modifie pas les dates de début des opérations lorsque l'on ajoute une nouvelle opération. Par contre, pour le même problème soumis à la contrainte de time lag maximum, il est alors possible et probable que l'insertion d'une opération déplace des opérations déjà ordonnancées.

La procédure *Restriction* construit un sous-ensemble *F* des opérations de *E* (ligne 6). Pour cela, il utilise l'ensemble *E* des opérations éligibles, l'ensemble *U* des opérations non ordonnancées et l'ordonnancement *S*. Suivant l'ensemble *F* retenu, on peut lui conférer une des propriétés suivantes : semi-actif, actif ou sans délai.

Ensuite, la procédure *Rule* retourne une opération *o* choisie parmi les opérations de *F* (ligne 7). L'opération est choisie en fonction de la règle de priorité, cette opération sera ajoutée dans l'ordonnancement.

L'insertion de l'opération *o* dans l'ordonnancement partiel est réalisée par la procédure *Update* (ligne 8). Cette procédure ajoute l'opération dans l'ordonnancement et met éventuellement à jour les dates de début des autres opérations de l'ordonnancement. La procédure *Update* renvoie un ordonnancement partiel qui devient l'ordonnancement partiel courant *S*.

Finalement, on retire l'opération o de l'ensemble des opérations à ordonnancer U (ligne 9).

6.3.1 Utilisation de la formalisation pour le problème de jobshop

Dans cette section, nous présentons l'algorithme de Giffler et Thompson (1960) dédié aux problèmes de jobshop. Pour décrire cet algorithme, nous tirons partie de la formalisation présentée ci-dessus en ne décrivant que le fonctionnement de chaque procédure :

- La procédure *Eligible* détermine pour chaque job la première opération de la gamme qui n'a pas déjà été ordonnancée. Pour implanter cette procédure de manière efficace, on utilise en général un tableau indiquant pour chaque job son degré d'avancement dans la gamme.
- La procédure *Evaluate* calcule les dates de début d'une opération en respectant les contraintes (et donc les dates de début des opérations). Cette procédure utilise par exemple le modèle de graphe conjonctif - disjonctif présenté dans le paragraphe 5.
- Suivant la procédure *Restriction* utilisée, on peut conférer différentes propriétés aux ordonnancements générés. Les propriétés semi-actif, sans-délai et actif sont obtenues de la manière suivante (dans ces définitions, t_i désigne la date de début de l'opération i prévue par la procédure *Evaluate*) :
  - Les ordonnancements semi-actifs peuvent être généré en prenant $F = E$.
  - Pour générer les ordonnancements sans-délais, on calcule d'abord la date t comme étant la plus faible date de début parmi les opérations de F. L'ensemble E est alors constitué des opérations de F commençant à la date t.
  - Les ordonnancements actifs sont générés si F contient les opérations commencées avant c_k sur la machine μ_k où k est l'opération de F avec la plus faible date de fin.
- La procédure *Rule* choisit une opération dans l'ensemble F conformément à la règle de priorité choisie. Un grand nombre de règles de priorités peuvent être utilisées pour le problème de jobshop (cf. (Haupt, 1989) pour un état de l'art des règles de priorité).
- La procédure *Update* permet d'insérer l'opération o après la dernière opération traitée sur la même machine. Pour calculer la date de début l'opération o, il suffit de calculer le maximum entre les dates de fin des opérations $PM(o)$ et $PJ(o)$.

6.3.2 Utilisation de la formalisation pour le problème de jobshop avec time lag

Dans cette section, nous présentons une heuristique de construction dédiée au problème de jobshop avec time lag. Cette heuristique est basée sur celle de Giffler et Thompson pour le problème de jobshop présentée ci-dessus. Pour généraliser cette heuristique, nous proposons d'inclure une procédure de réparation afin d'obtenir des solutions réalisables. En effet, une des principales difficultés du problème de jobshop avec time lag est que certains ordonnancements partiels ne peuvent pas être complétés pour obtenir une solution réalisable. Ceci rend particulièrement difficile le développement d'algorithme de type glouton comme l'heuristique de Giffler et Thompson. Mais grâce à l'introduction d'une procédure de réparation, nous pouvons proposer la première heuristique permettant de construire des ordonnancements systématiquement réalisables. (Deppner (2003), propose une heuristique respectant les time lags jusqu'à une certaine date prédéfinie qui sert de seuil).

Les procédures *Eligible* et *Restriction* ont été présentées dans la section ci-dessus pour le problème de jobshop, elles sont identiques pour notre problème et nous ne les répétons pas dans cette section.

- La procédure *Evaluate* consiste d'abord à récupérer l'ordre de traitement des opérations dans l'ordonnancement *S*, puis à ajouter l'opération *i* à cet ordre en dernière position. L'ordre obtenu permet de construire un graphe conjonctif dont on calcule les plus longs chemins à l'aide de l'algorithme préconise dans la section 5.2. Il y a alors deux cas possibles suivant que le graphe contienne ou non un cycle absorbant. Si le graphe contient un cycle absorbant, c'est qu'il n'est pas possible d'ordonnancer cette opération en respectant l'ordre de traitement. La procédure *Evaluate* renvoie alors une date de début de traitement infinie pour signifier que cette opération n'est pas réalisable. Par contre, s'il existe des plus longs chemins dans le graphe, le calcul retourne alors un ordonnancement incluant l'opération *i* et respectant l'ordre des opérations dans *S*. La procédure *Evaluate* renvoie alors la date de début de l'opération *i* dans l'ordonnancement calculé. Rien ne garantit que l'ordonnancement partiel *S* et l'ordonnancement calculé ne soient identiques car les opérations déjà ordonnancées peuvent être déplacées pour satisfaire la contrainte de time lag maximum introduite avec l'opération *i*. Durant l'exécution de l'heuristique, un grand nombre d'appels à la procédure de plus longs chemins sont effectués. C'est pourquoi il est important d'utiliser une version efficace de cet algorithme. De plus, nous préconisons d'utiliser les calculs de plus longs chemins précédents pour accélérer les nouveaux calculs. En effet, lors d'une itération, on dispose de l'ordonnancement précédent *S*. Cet ordonnancement peut être vu comme l'ordonnancement le plus calé à gauche respectant toutes les contraintes sauf celles de l'opération *i*. On peut donc récupérer les dates de début de l'ordonnancement *S* pour initialiser l'algorithme de plus long chemin. Ce faisant, on diminue le nombre d'itérations de l'algorithme de plus longs chemins de manière significative.
- La procédure *Rule* consiste à choisir une opération *o* dans l'ensemble *F*. Toutes les règles utilisables pour le problème de jobshop peuvent être appliquées au problème de jobshop avec time lag. Par contre, ces règles ne prennent pas en compte les opérations irréalisables (i.e. opération pour laquelle la procédure *Evaluate* a renvoyé l'infini). Nous choisissons d'ordonnancer en premier les irréalisables pour la raison suivante : toute opération irréalisable ne peut le devenir en insérant de nouvelles opérations à la fin de l'ordonnancement. C'est pourquoi il est inutile d'attendre les itérations suivantes et c'est pourquoi nous choisissons de traiter immédiatement toute opération irréalisable. En d'autres termes, la procédure *Rule* pour le problème de jobshop avec time lag consiste à choisir d'abord les solutions irréalisables puis si toutes les opérations éligibles sont réalisables, on applique une des règles de priorité classiques.
- La procédure *Update* insère l'opération *o* (choisie par la procédure *Rule*) dans l'ordonnancement partiel. Si l'opération o est réalisable (sa date de début évaluée t_i est finie), la procédure *Update* insère donc l'opération comme l'a fait la procédure *Evaluate*. Il est d'ailleurs possible et souhaitable de sauvegarder le résultat de la procédure *Evaluate*

pour éviter de réaliser le même calcul dans la procédure *Update*. Par contre, si l'opération o n'est pas réalisable (sa date de début évaluée t_i est infinie), la procédure *Update* doit alors réparer l'ordre de traitement des opérations. Puis, l'ordonnancement obtenu devient l'ordonnancement *S*.

La procédure de réparation est présentée dans l'algorithme 3-4. Elle consiste à décaler sur la droite les opérations du job de l'opération *o*. Pour cela, l'algorithme procède de manière itérative. À chaque itération, on considère une opération *k* que l'on échange successivement avec l'opération à sa droite. Si l'ordonnancement devient réalisable, on arrête la procédure, sinon on décale de nouveau l'opération sur la droite. Si l'opération ne peut plus être décalée, c'est qu'elle est devenue la dernière opération dans le sens où il n'y a pas d'autre opération traitée après elle sur la même machine. Dans ce cas, l'opération précédant celle-ci dans la gamme devient l'opération *k*.

```
Entrée :
   o : opération à insérer
   S : ordonnancement partiel
Début :
1) k = PJ(o);  // PJ(o) est la première opération à décaler
2) Tant que S n'est pas réalisable faire
3)       Si k est la dernière opération sur sa machine alors
4)             k = PJ(k);  // change d'opération
5)       sinon
6)             décale l'opération k dans S;
7)       finsi
8) fintantque
```

Algorithme 3-4. Procédure de réparation

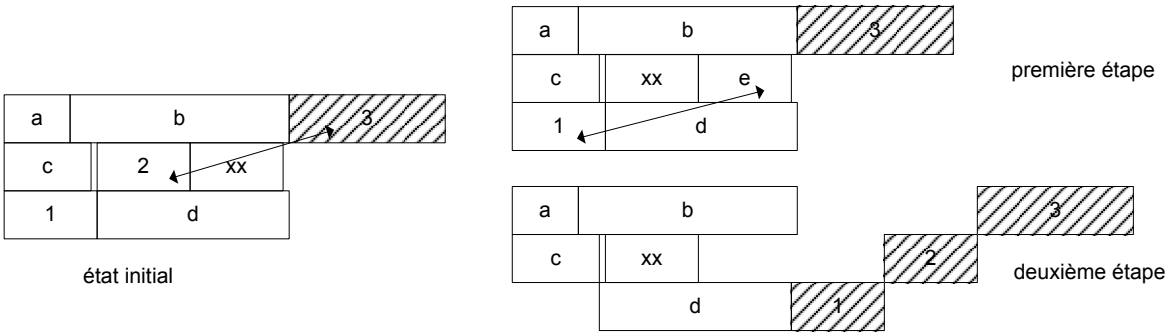

Figure 3-12. Procédure de réparation

Nous illustrons cet algorithme sur l'exemple de la figure 3-12. Le job 1 contient trois opérations (1, 2 et 3). Les opérations *a*, *b*, *c*, *d* et *e* concernent d'autres jobs dans l'ordonnancement *S*. L'opération *3* est en cours d'insertion dans l'ordonnancement. Le time lag maximum entre les opérations *2* et *3* est violé, et la procédure de réparation commence. Les contraintes de time lag violées sont représentées par une double flèche. La première étape de l'algorithme consiste à initialiser *k* avec l'opération *2=PJ(3)*. Cette opération peut être décalée sur la droite et le décalage est donc réalisé à la ligne *6* de l'algorithme. L'ordonnancement correspondant est appelé "première itération" sur la figure. Dans cet ordonnancement, le time lag maximum entre les opérations *1* et *2* est violé et une deuxième itération de l'algorithme commence alors. Cette itération met en évidence que l'opération *2* ne peut plus être décalée sur la droite puisque c'est la dernière opération traitée sur sa machine. La ligne *4* de l'algorithme fait donc passer *k* à l'opération précédente dans l'ordre de traitement i.e. opération *1*). Une troisième

itération de l'algorithme commence alors dans laquelle l'opération *1* est décalée sur la droite. À la fin de cette itération, les opérations *1*, *2* et *3* ont été complètement calées à droite et l'ordonnancement est devenu réalisable. Il est à remarquer que la procédure de réparation change l'ordre des opérations sur les machines mais ne change pas l'ordre relatif des opérations des autres jobs.

Proposition 1 :

L'algorithme 3-4 crée systématiquement des ordonnancements réalisables.

Preuve : Si l'algorithme se déroule normalement, la condition de la boucle "tant que" est fausse à la fin du programme. Dans ce cas, la proposition est triviale à démontrer. Mais il faut tout de même montrer que lorsqu'il n'y a plus d'opération précédente dans le job et que l'opération courante est calée à droite alors l'ordonnancement est forcément réalisable.

Pour prouver ceci, on se place donc dans le cas où l'opération *k* est la première opération du job *j* et où cette opération est complètement calée sur la droite. On veut donc montrer qu'un tel ordonnancement est réalisable. On décompose cet ordonnancement en deux parties : *S'* d'un côté et les opérations du job *j* de l'autre. Les opérations du job *j* sont toutes calées sur la droite comme dans l'exemple de la figure 3-12. Cette partie est forcément réalisable car, dans le pire des cas, le job *j* est ordonnancé comme s'il était seul. Pour le problème de jobshop avec time lag que l'on traite, il n'y a pas de contraintes de time lag entre jobs. Il n'y a donc pas de contrainte entre le job *j* et les jobs dans *S'*. Il ne reste donc qu'à vérifier que *S'* est réalisable.

Les opérations de l'ordonnancement *S'* sont présentes dans le même ordre dans l'ordonnancement *S*, il n'y a que les opérations du job *j* qui ont été retirées. Dans l'ordonnancement S, les dates de début de ces opérations de *S'* vérifient toutes les contraintes. Il est donc possible de trouver un ordonnancement réalisable pour *S'*, ce que l'algorithme de plus long chemin va donc réaliser. Ainsi, l'ordonnancement *S'* est forcément réalisable.

En résumé, S' est réalisable, les opérations du job *j* sont réalisables et il n'y a pas de contrainte entre les opérations de *S'* et du job *j*. L'ordonnancement proposé est donc forcément réalisable.

Remarque sur la procédure de réparation :

Si la procédure *Restriction* est configurée pour générer des ordonnancements sans délai ou actif, il est possible que les ordonnancements générés ne possèdent pas ces priorités à cause de la procédure de réparation. En effet, la réparation transforme les ordonnancements partiels et modifie leurs propriétés. Par contre, tous les ordonnancements générés sont semi-actifs car ils sont calculés à l'aide de l'algorithme de plus longs chemins.

Exemple de solutions

Opération	1	2	3	4	5	6	7	8	9
Priorité	1	2	3	1	1	3	2	2	3

Tableau 3-13. Exemple de représentation par règle de priorité

Le tableau 3-13 présente un exemple de solution stockée pour la représentation par règle de priorité. Elle fournit pour chaque opération sa priorité en tant que valeur numérique. Cette façon de stocker une solution est la plus adaptée aux algorithmes d'optimisation, mais une solution peut aussi être définie par une règle permettant de définir quelle opération est la plus prioritaire. Citons par exemple la règle SPT (Shortest Processing Time) qui consiste à choisir l'opération dont la durée de traitement est la plus faible.

Ensemble des solutions stockées

Les solutions stockées sont toutes les priorités possibles pour chacune des opérations. Une solution stockée est donc un tableau de valeurs numériques pour chaque opération. Il y a donc un très grand nombre de solutions stockées différentes.

Ensemble des solutions codées

Les solutions codées sont constituées de tous les ordres semi-actifs (ces ordonnancements sont réalisables par définition).

Propriétés

- Sur représentation : Non, car tout ensemble de valeurs de priorité mène à un ordonnancement réalisable.
- Multi représentation : Oui, car de nombreuses priorités mènent au même ordonnancement.
- Heuristique : Non, les ordonnancements semi-actifs sont dominants pour tout critère régulier. Pour un critère non régulier, cette représentation est heuristique.

6.4 Module d'évaluation 4 : Évaluation par ordonnancements canoniques

La représentation semi-active est largement utilisée pour les problèmes disjonctifs en général car elle permet de construire des ordonnancements réalisables et de nombreuses heuristiques l'utilisent à cette fin. Pour le problème de jobshop avec time lag, la représentation semi-active ne permet pas de construire des ordonnancements systématiquement réalisables. Grâce à la représentation canonique, nous proposons un ensemble d'ordonnancements faciles à construire et tous réalisables. Dans la suite, on appelle « ordonnancements canoniques », les ordonnancements que l'on peut obtenir à partir de ce module d'évaluation.

Exemple de solutions

1	2	3

Tableau 3-14. Exemple de représentation canonique

Une solution qui utilise la représentation canonique est un ordre de jobs.

Ensemble de solutions stockées

L'ensemble des solutions stockées est l'ensemble de toutes les permutations des numéros de jobs. Pour un problème comportant n jobs, il y a donc $n!$ solutions stockées possibles.

Ensemble de solutions codées

Les solutions codées sont tous les ordonnancements canoniques.

Codage/Décodage

Étant donné un ordre de jobs, l'algorithme de décodage peut être réalisé de plusieurs manières. Pour expliquer de manière intuitive à quoi ressemblent les ordonnancements canoniques, nous montrons une première manière de les construire puis nous présentons une manière plus directe et plus efficace.

On construit d'abord un premier ordonnancement dans lequel chaque job est planifié comme s'il était seul dans le système. On planifie donc toutes les opérations du premier job, puis toutes les opérations du deuxième, ... Un exemple d'ordonnancement ainsi construit est présenté dans la figure 3-13. Cet ordonnancement est forcément réalisable car chaque job est planifié seul dans le système. Un ordonnancement canonique est l'ordonnancement semi-actif obtenu par décalage à gauche des opérations de cet ordonnancement. L'ordonnancement semi-actif correspondant à l'ordonnancement de la figure 3-13 est présenté dans la figure 3-14.

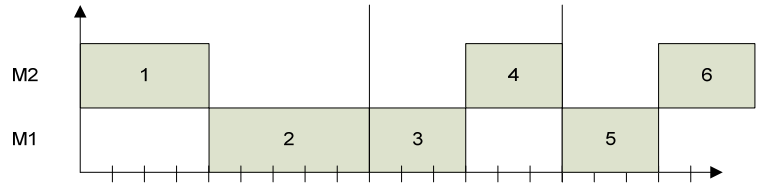

Figure 3-13. Ordonnancement réalisable

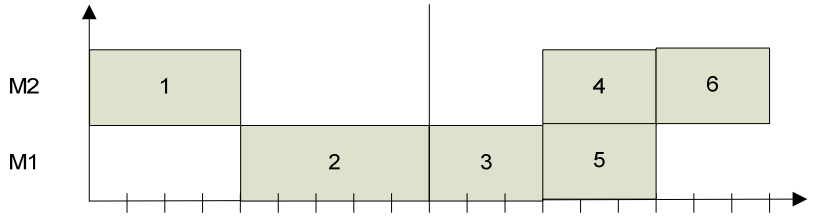

Figure 3-14. Ordonnancement canonique

Les ordonnancements canoniques peuvent être obtenus directement en utilisant comme intermédiaire de calcul la représentation par ordre total d'opération. Dans ce cas, la représentation canonique consiste à remplacer dans l'ordre des jobs chaque numéro de job par les opérations qui le compose. On obtient alors un ordre total d'opérations dont l'ordonnancement semi-actif correspondant peut être évalué (cf. §4.3.3, chapitre 2).

Propriétés

- Sur représentation : Non, tous les ordres de job mènent à un ordonnancement canonique réalisable.
- Multi représentation : Non, il est trivial de remarquer que tous les ordres de jobs mènent à des ordonnancements canoniques différents.
- Heuristique : Oui, car peu d'ordonnancements sont représentables. Entre autres, sauf cas rares, il n'est pas possible de représenter l'ordonnancement optimal.

7 Proposition d'un algorithme tabou

Dans le chapitre précédent, nous avons présenté en détail l'algorithme tabou de Nowicki et Smutnicki (1996). Puisque cet algorithme fait référence pour le problème de jobshop, il est naturel de vouloir l'étendre pour le problème de jobshop avec time lags. Ainsi, la première méthode d'optimisation que nous avons proposée est une adaptation de l'algorithme tabou au problème de jobshop avec time lags. Cette méthode a été présenté dans (Caumond *et al.*, 2004).

Pour réaliser cette adaptation, nous avons procédé en deux temps : d'abord, nous avons généralisé le module d'évaluation basé sur le graphe conjonctif-disjonctif (cf. § 5 ci-dessus) puis nous avons adapté l'algorithme. Cette adaptation de l'algorithme fait l'objet de ce paragraphe.

7.1 Présentation

Pour construire la version de l'algorithme tabou dédiée au problème de jobshop avec time lags, nous réutilisons complètement l'algorithme de principe. Pourtant, trois adaptations sont réalisées : la fonction $C_{\max}$ qui calcule le makespan à partir d'un ordre d'opérations est modifiée, le voisinage est réadapté et plusieurs points de départ sont envisagés.

La fonction $C_{\max}$ est liée à la représentation utilisée : l'algorithme tabou pour le problème de jobshop utilise la représentation semi-active du problème de jobshop et nous proposons de remplacer

cette représentation par la représentation semi-active pour le problème de jobshop avec time lags (cf. §6.1). La principale différence entre ces deux représentations est dans la réalisabilité des solutions envisagées. En effet, dans les deux versions, l'algorithme tabou n'envisage que des solutions réalisables. Mais pour le problème de jobshop avec time lags, il est possible que des voisins ne soient pas réalisables. Dans ce cas, elle renvoie un makespan infini pour que cette solution irréalisable soit exclue de la recherche. La seconde différence entre les deux représentations est la complexité des algorithmes mis en œuvre. À cause des contraintes de time lags, la complexité du plus long chemin est plus élevée dans le graphe conjonctif pour le jobshop avec time lags. Ainsi, l'algorithme proposé est moins performant en terme de temps de calcul.

De plus, nous adaptons le voisinage proposé par les auteurs. Ce voisinage est présenté dans la section 0, c'est un voisinage très guidé qui tire partie du graphe conjonctif - disjonctif et du chemin critique calculé. Dans le cas du jobshop, tous les voisins obtenus sont forcément réalisables. Ce n'est pas le cas pour le problème de jobshop avec time lags. En fait, tous les voisins obtenus respectent forcément les contraintes du sous-problème de jobshop, mais tous ne respectent pas forcément les contraintes de time lag. Dans cet algorithme tabou, nous envisageons à priori tous les voisins, mais les voisins qui se révèlent être irréalisables sont exclus de la recherche.

L'algorithme 3-5 présenté ci-dessous est l'algorithme de principe de l'algorithme tabou réadapté au problème de jobshop avec time lags. La dernière adaptation de l'algorithme consiste à envisager plusieurs points de départ. Pour générer ces points de départ, on utilise la représentation par ordonnancements canoniques (cf. §6.4). Chaque ordre de jobs permet de générer un ordre canonique différent.

```
Pour i de 1 à max_start faire
    Soit x une séquence de traitement;
    BT=∅;
    LT=∅;
    tant que non fin faire
          pour y voisin de x faire
                Evaluer Cmax(y);
          finpour
          Choisir le meilleur voisin non tabou;
          Mettre à jour LT;
          si Cmax(y)<Cmax(x*) alors
                Mettre y dans BT;
                iter=0;
          finsi
          si iter>max_iter alors
                si BT≠∅ alors
                      revenir en arrière dans BT;
                sinon
                      fin = true;
                finsi
          finsi
    fintantque
```

Algorithme 3-5. TSAB - Algorithme tabou

Notre algorithme tabou peut être exécuté sur tous les problèmes de jobshop avec time lags. En prenant des coefficients de time lag minimum nul et des coefficients de time lag maximum suffisamment grand, on obtient donc un problème de jobshop classique. Ainsi, on peut aussi utiliser notre adaptation de l'algorithme tabou pour le problème de jobshop classique. L'algorithme reste aussi performant que l'algorithme de Nowicki et Smutnicki en termes de qualité de solutions, mais il est légèrement moins performant en termes de temps de calcul.

7.2 Expérimentations numériques[6]

Pour évaluer la qualité des résultats de l'algorithme tabou, nous séparons les résultats en deux catégories : les instances dont les time lags sont lâches et les instances dont les time lags sont serrés.

Quand les time lags sont très lâches, le problème est alors un jobshop classique. L'algorithme tabou est alors particulièrement efficace (colonne ∞ du tableau 3-15). Au fur et à mesure que l'on diminue les coefficients de time lags, la durée des time lags maximum diminuent et la qualité des résultats diminue.

TL max		**2**		**3**		**5**		**10**		∞	
Instance	**OPT.**	**MS**	**Écart**	**MS**	**Écart**	**MS**	**Écart**	**MS**	**Écart**	**MS**	**Écart**
ft06	55	55	0%	55	0%	55	0%	55	0%	55	0%
car5	7702	8281	8%	8006	4%	7702	0%	7702	0%	7702	0%
car6	8313	8467	2%	8667	4%	8313	0%	8313	0%	8313	0%
car7	6558	6573	0%	6576	0%	6558	0%	6558	0%	6558	0%
car8	8264	8586	4%	8581	4%	8264	0%	8264	0%	8264	0%
la01	666	732	10%	666	0%	666	0%	666	0%	666	0%
la02	655	681	4%	663	1%	655	0%	655	0%	655	0%
la03	597	671	12%	638	7%	628	0%	597	0%	597	0%
la04	590	642	9%	615	4%	597	1%	590	0%	590	0%
la05	593	600	1%	593	0%	593	0%	593	0%	593	0%
Moyenne			5%		2%		1%		0%		0%

Tableau 3-15. Résultats de l'algorithme tabou sur des time lags lâches

Par contre, l'algorithme tabou proposé se révèle être de moins en moins efficace quand les coefficients de time lag diminuent. Les solutions trouvées sont à 30 % en moyenne de la solution optimale. La pire solution trouvée est à 72% de l'optimum pour l'instance de la01 avec un coefficient de time lag maximum à 0,25. Pourtant, ces instances sont des instances de petite taille et l'algorithme devrait trouver des solutions de bonne qualité. Ceci montre que l'algorithme tabou trouve difficilement de bonnes solutions pour les instances à time lags serrés.

Seules les instances de Carlier (car5 à car8) obtiennent globalement de bons résultats. La particularité de ces instances est que tous les jobs ont la même gamme. Les instances car5 à car8 sont

[6] La taille des instances est présentée dans le paragraphe 3.1

donc des instances de flowshop avec permutation. Ces instances sont donc particulières et nous pensons que la forme des solutions de départ est particulièrement adaptée (cf. module d'évaluation 4).

Sur des instances plus serrées (cf. tableau 3-16), l'algorithme tabou est relativement moins performant. La qualité des solutions décroit au fur et à mesure que les time lags diminuent. Pourtant, la solution obtenue pour les time lags minimum à 0 (contraintes no-wait) est de meilleure qualité que les time lags supérieurs. Ce phénomène a été observé sur d'autres méthodes.

	0			**0,25**			**0,5**			**0,75**			**1**		
Instance	**OPT**	**MS**	**Écart**	**OPT**	**MS**	**Écart**	**OPT**	**MS**	**Écart**	**OPT**	**MS**	**Écart**	**OPT**	**MS**	**Écart**
ft06	73	94	29%	64	91	42%	63	81	29%	58	77	33%	58	61	5%
car5	9159	11459	25%	8311	10838	30%	7788	10293	32%	7768	9481	22%	7755	8910	15%
car6	9690	11243	16%	8897	10080	13%	8648	9100	5%	8330	8911	7%	8323	9248	11%
car7	7705	7705	0%	6971	7363	6%	6654	6953	4%	6573	6733	2%	6573	6590	0%
car8	9372	10144	8%	8689	9135	5%	8452	8856	5%	8279	8856	7%	8279	8833	7%
la01	971	1473	52%	819	1408	72%	758	1230	62%	708	1061	50%	683	916	34%
la02	937	1436	53%	819	1217	49%	742	1153	55%	695	1082	56%	686	900	31%
la03	820	1108	35%	721	1160	61%	679	1052	55%	651	978	50%	640	847	32%
la04	887	1275	44%	760	1175	55%	703	1106	57%	673	987	47%	646	950	47%
la05	777	1128	45%	702	1031	47%	622	957	54%	615	840	37%	593	761	28%
Moyenne			31%			38%			36%			31%			21%

Tableau 3-16. Résultats de l'algorithme tabou sur des time lags serrés

7.3 Conclusion sur l'algorithme tabou

L'algorithme tabou proposé généralise bien l'algorithme de Nowicki et Smutnicki : il fournit exactement les mêmes solutions que l'algorithme original pour les instances de jobshop classique. Pour ces instances, la différence principale se situe donc au niveau des temps de calcul qui sont légèrement supérieurs pour la version avec time lags.

L'intérêt de l'algorithme tabou proposé est qu'il prend en compte les contraintes de time lags. Les expérimentations numériques montrent que l'algorithme est particulièrement efficace pour les problèmes dont les time lags sont lâches. Mais, lorsque les time lags se resserrent, la qualité des résultats diminue fortement. Nous expliquons cette perte de qualité par la raréfaction des solutions réalisables dans le voisinage (comme nous l'avons mis en évidence dans la section "difficulté de résolution").

Pour construire un algorithme plus performant pour les time lags serrés, et pour permettre une meilleure exploration de l'espace de recherche, nous proposons, dans l'algorithme mémétique proposé ci-après, d'autoriser les solutions irréalisables.

8 Proposition d'un algorithme mémétique

Cet algorithme, proposé dans (Caumond *et al.*, 2005), est spécialement dédié aux instances dont les time lags sont serrés. Pour ces instances, nous avons vu dans la section 3 que le nombre de solutions réalisables est très faible, et que la plupart des voisins d'une solution réalisable ne le sont pas. Il faut donc plusieurs applications successives du voisinage pour passer d'une solution réalisable à une autre. Ainsi, pendant le processus de recherche, la méthode d'optimisation doit envisager des solutions irréalisables de mauvaise qualité.

Pour cela, nous proposons d'utiliser le module d'évaluation "espace stratifié" qui permet d'envisager et de comparer des solutions irréalisables. Il devient donc toujours possible de passer d'une solution réalisable de bonne qualité à une autre en utilisant itérativement le voisinage. Pour permettre une bonne exploration de l'espace, nous proposons d'utiliser un algorithme mémétique car, par son mécanisme de gestion de population, il garantit la diversité des solutions envisagées.

La forme du chromosome et son décodage sont dictés par la représentation (cf. §8.1). Dans le paragraphe 8.2, nous présentons le croisement associé. Par contre, à cause de la forme particulière de l'espace de recherche nous avons ajouté une détection de double (§ 8.3) qui vise à limiter la convergence prématurée de l'algorithme. Malgré ces efforts, l'algorithme a tendance à retenir des solutions très approchantes et à mal explorer l'espace de recherche. Ainsi, des remplacements périodiques (§ 8.6) permettent l'introduction d'une nouvelle partie de la population pour aider à diversifier les individus de la population. L'opérateur de mutation (§ 8.5) tire parti des voisinages efficaces connus pour le problème de jobshop.

8.1 Définition des chromosomes

Pour le problème du jobshop, Bierwirth (1995) a proposé le chromosome à valeurs dupliquées. Ce chromosome est constitué d'une suite ordonnée de numéros de jobs. La $j^{ème}$ occurrence du nombre i dans le chromosome désigne la $j^{ème}$ opération du job i. L'avantage de cette représentation est de ne représenter que des ordres valides d'opérations (i.e. des ordres compatibles avec les contraintes de précédence).

L'ordre défini par le chromosome peut être utilisé dans la représentation stratifiée. Ainsi, la représentation permet d'évaluer toute valeur du chromosome. En résumé, on part donc d'un chromosome à valeurs dupliquées, que l'on transforme en un ordre d'opérations, cet ordre est utilisé pour orienter le graphe conjonctif, l'algorithme de comptage des time lag à supprimer est lancé, cet algorithme utilise un algorithme de plus long chemin dans le graphe.

Exemple de chromosome

Le tableau ci-dessous présente un ordonnancement semi-actif et le chromosome associé.

M1	1	7	6
M2	4	2	9
M3	5	8	3

1	2	2	1	3	3	2	3	1

Tableau 3-17. Exemple de chromosome

8.2 Croisement des chromosomes

Les chromosomes que nous manipulons sont identiques à ceux proposés par Bierwirth. Nous utilisons donc aussi l'opérateur de croisement appelé GOX. Cet opérateur est inspiré de l'opérateur OX pour le TSP et l'idée principale est de conserver l'ordre relatif des opérations des parents vers leurs fils.

L'opérateur traite les deux parents de manière asymétrique : l'un est le donneur et l'autre est le receveur. Pendant le croisement, on choisit de manière aléatoire une sous-chaîne du donneur. On impose que la longueur de la chaîne soit comprise entre un tiers et la moitié de la longueur totale du chromosome. Le fils est construit comme étant une copie du receveur. Puis, les opérations correspondant à la sous-chaîne sont recherchées et supprimées dans le fils. Enfin, la position de la

première opération de la sous-chaîne est recherchée dans le receveur. À cette position, la sous-chaîne est insérée dans le fils.

Une procédure différente est suivie lorsque la sous-chaîne va au-delà de la fin de la chaîne. Ce cas apparaît lorsque la procédure de tirage aléatoire choisit une position vers la fin de la chaîne. Dans ce cas, les opérations de la sous-chaîne sont aussi supprimées dans le fils, mais la sous-chaîne est réinsérée à la même position que celle qu'elle occupait dans le donneur.

Les exemples ci-dessous illustrent le croisement GOX. Le tableau 3-18 présente un exemple de croisement classique. Dans la première ligne, les cases grisées représentent la sous-chaîne choisie. Dans la deuxième ligne, les cases grisées représentent les cases supprimées et dans la dernière ligne, les cases grisées représentent la sous-chaîne réinsérée. Le tableau 3-19 présente le cas où la sous-chaîne dépasse la fin du chromosome.

Donneur	1	2	2	1	3	3	2	3	1
Receveur	3	1	1	2	1	2	3	3	2
Enfant	1	2	1	3	3	1	2	3	2

Tableau 3-18. Exemple de croisement GOX (cas classique)

Donneur	1	2	2	1	3	3	2	3	1
Receveur	3	1	1	2	1	2	3	3	2
Enfant	1	2	3	1	2	3	3	2	1

Tableau 3-19. Exemple de croisement GOX (cas de dépassement)

8.3 Technique de détection des doubles

On appelle "double" un chromosome représentant la même solution qu'un autre chromosome de la population. On s'intéresse à la présence de doubles dans la population, car ils appauvrissement le matériel génétique et nuisent à la convergence de l'algorithme. En particulier pour le problème de jobshop avec time lag, nous avons observé qu'il était souhaitable de limiter leur présence. En effet, des études préliminaires ont montré que de nombreuses solutions de la population deviennent prématurément identiques si aucune détection de doubles n'était présente dans l'algorithme mémétique.

Notre technique de détection de doubles consiste à calculer une signature pour chaque chromosome. La signature est une valeur scalaire agrégeant des informations. Pour calculer une signature à partir d'un chromosome, nous proposons d'utiliser la fonction de hachage quadratique : si on note t_i la date de début de l'opération i, alors la signature est $\sum_{i \in O} t_i \mod K$. Le paramètre K est un nombre choisi arbitrairement. Nous avons choisi de détecter les doubles sur l'ordonnancement plutôt que sur le chromosome car de nombreux chromosomes différents mènent au même ordonnancement. Par contre, dû à la représentation stratifiée, les ordonnancements sur lesquels on calcule les signatures peuvent ne pas respecter tous les time lags.

Les signatures permettent de discriminer rapidement si deux ordonnancements sont identiques. Par contre, comme toute fonction qui réalise une agrégation, elles perdent des informations. Ainsi, deux ordonnancements différents peuvent mener à une même signature. Pour tenter de diminuer ce phénomène, il est important de choisir le paramètre K suffisamment grand.

Lors de l'insertion d'un nouvel individu dans la population, on souhaite vérifier si celui-ci n'est pas le double d'un autre individu. Il faut donc à priori faire autant de comparaison qu'il y a d'individus dans la population. Pour accélérer ce calcul, nous proposons d'utiliser un grand tableau appeler "*DOUBLE*" donnant, dans la case "*i*", le nombre d'individus de la population dont la signature est "i". Ainsi la vérification des doubles se fait en O(1). La taille du tableau *DOUBLE* est égale à la valeur de *K*. La taille des mémoires centrales est largement suffisante pour envisager des valeurs élevées de *K*.

8.4 Population initiale

La population initiale est constituée d'ordonnancements canoniques. Lors de la création de cette population, on met à jour au fur et à mesure le tableau *DOUBLE*. Ainsi, on est en mesure d'interdire les doubles dans la population.

8.5 Mutation

La mutation proposée est une recherche locale basée sur un algorithme itératif de type *Hill Climbing*. Cet algorithme consiste à parcourir le voisinage et à retenir le premier voisin améliorant (stratégie "première amélioration").

Le voisinage retenu pour cet algorithme est basé sur la notion de blocs introduite par Grabowski *et al.* (1986). Cette notion de blocs a été utilisée avec succès pour le problème de jobshop par Nowicki et Smutnicki (1996). Nous réutilisons ce voisinage car il permet d'obtenir rapidement des optimaux locaux de bonne qualité. En effet, ce voisinage est très guidé et contient donc un nombre faible de voisins.

Dans Mattfeld (1995), l'auteur propose une étude des algorithmes de type Hill Climbing. Il compare différentes stratégies de choix du voisin utilisées en conjonction avec différents voisinages. Sa conclusion est que la stratégie "meilleure amélioration" utilisée en conjonction avec un voisinage complexe (appelé N4) donne les meilleurs résultats. Dans le contexte d'un algorithme mémétique, nous avons préféré utiliser un voisinage plus simple et une stratégie limitant le nombre d'évaluations de voisins.

En résumé, la mutation d'un chromosome *C* consiste à parcourir tous les blocs et à accepter le premier voisin améliorant. On calcule de nouveau les blocs sur ce voisin puis on choisit de nouveau le premier voisin améliorant. Cet algorithme continue itérativement jusqu'à ce qu'aucun voisin ne soit améliorant. Dans ce cas, le point courant est un optimum local et la mutation est terminée. Grâce à cette mutation, on génère donc efficacement de nouveaux optimaux locaux dans la population.

De plus, on choisit d'arrêter la recherche locale au bout de *pml* améliorations. Cette fonctionnalité est introduite pour deux raisons. D'abord, cette limitation permet de contrôler le temps passé à faire des mutations, car pour les problèmes de grande taille, le nombre d'itérations avant d'arriver à un optimum local peut être relativement élevé. Deuxièmement, quels que soient les réglages effectués, l'algorithme a tendance à converger prématurément. En limitant le nombre d'itérations dans la mutation, on introduit un levier pour contrôler cette convergence.

8.6 Méthode de remplacement et condition d'arrêt

Pour lutter contre l'appauvrissement de la population, des redémarrages périodiques sont effectués toutes les *np* itérations successives sans améliorations de la meilleure solution. Malgré les différents mécanismes introduits pour limiter la convergence prématurée, l'algorithme mémétique proposé a tendance à perdre de son matériel génétique et à converger prématurément. Ceci est particulièrement observé dans le cas où les coefficients de time lags sont serrés, nous pensons donc que cette convergence est due à la structure particulière de l'espace de recherche et au faible nombre de solutions réalisables.

Pour pouvoir continuer la recherche, et pour pouvoir exploiter les bonnes solutions trouvées, nous proposons une procédure de redémarrage qui perturbe les chromosomes pour obtenir une population voisine. La perturbation choisie consiste à permuter les jobs deux à deux. Quand deux jobs *i* et *j* sont permutés, les opérations du job *i* sont remplacées par les opérations du job *j* dans le chromosome et inversement. Rien ne garantit que cette permutation ne rende irréalisable une solution qui était réalisable.

8.7 Structure générale de l'algorithme mémétique

Les différentes parties de l'algorithme mémétique décrites ci-dessus sont utilisées conjointement dans notre algorithme mémétique. L'algorithme 3-6 présente l'algorithme de principe utilisé.

```
1) Construire la population initiale avec nc ordonnancements canoniques
2) npi :=0;
3) ni :=0;
4) Répéter
5)       Choisir P1 et P2 par tournoi binaire
6)       Appliquer le croisement GOX entre P1 et P2, on obtient C
7)       Tirer aléatoirement k dans l'intervalle [1, int(nc/2)]
8)       si random<pm alors                  // Faire une mutation
9)             Appliquer la mutation à C
10)            Stocker le résultat dans S
11)            Si S n'est pas un double alors
12)                  C :=S;
13)            finsi
14)      finsi
15)      si C n'est pas un double alors
16)            si f(C)<f(P(nc)) alors
17)                  npi :=0;
18)            sinon
19)                  npi :=npi+1;
20)            finsi
21)            P[k] := C;
22)            Retrier la population P;
23)      finsi
24)      Si npi=np alors
25)            Effectuer un redémarrage
26)            npi :=0;
27)      finsi
28)      ni :=ni+1;
29) Jusqu'à ce que (ni=nmi) ou (f(P(nc))=LB);
```

Algorithme 3-6. Algorithme de principe de l'algorithme mémétique

Le schéma d'optimisation est basé sur un algorithme mémétique incrémental et hybride. À l'initialisation, l'algorithme consiste à construire la population initiale (ligne 1). Les compteurs *npi* (pour le nombre d'itérations sans amélioration) et *ni* (nombre total d'itérations) sont initialisés à zéro (ligne 2 et 3).

Les lignes 4 à 29 forment la partie itérative de l'algorithme. D'abord, les deux parents *P1* et *P2* sont choisis par tournoi binaire (ligne 5). Puis, l'individu *C* est généré par le croisement GOX à partir des deux individus *P1* et *P2* (ligne 6). La future place de l'individu généré s'appelle *k*, elle est tirée aléatoirement dans l'intervalle $[1; \lfloor nc/2 \rfloor]$ (ligne 7). Puis, avec une probabilité *pm* (ligne 8), cet individu

subit une mutation (ligne 9). Cette mutation engendre un nouvel individu qui peut être un double. S'il est double, cet individu est abandonné, sinon il est conservé (ligne 12).

Que l'individu ait été muté ou non, on vérifie si c'est un double. Si c'est un double, l'individu est abandonné, sinon, on va l'insérer dans la population (lignes 16 à 22). Si l'individu est un nouveau record, on réinitialise le compteur *npi* du nombre d'itérations sans améliorations (ligne 17). Sinon, ce compteur est incrémenté (ligne 19). Ensuite, l'individu est inséré dans la population à l'emplacement *k* (ligne 21). La population est triée de nouveau dans l'ordre du fitness de ses individus (ligne 22).

Si le nombre d'itérations sans amélioration (*npi*) atteint le seuil *np* (ligne 24), alors un redémarrage est réalisé (ligne 25) et le compteur est réinitialisé (ligne 26). À la ligne 28, le compteur global d'itérations sans amélioration est incrémenté (ligne 28). L'algorithme se termine au bout de *mni* itération ou lorsque la borne inférieure (*LB*) a été atteinte.

8.8 Application numérique

Cet algorithme mémétique a été implanté avec Delphi sur un Pentium IV 2.8 Ghz sous Windows XP. Comme les time lags maximum constituent la difficulté majeure du problème, nous avons conduit les tests avec des time lags minimums nuls. Pour mesurer la performance de la méthode, nous avons envisagé 4 réplications avec des racines différentes pour le générateur aléatoire. Nous utilisons le réglage suivant pour les différents paramètres :

- *mni=2 000 000*, le nombre maximum d'itérations,
- *np=1 000*, le nombre maximum de transitions défavorables avant un redémarrage,
- *nc=50*, le nombre d'individus dans la population,
- *pm=0,5%*, la probabilité de mutation
- $w_3 = 0$ pour inhiber l'influence de la mesure de la violation des time lags

Notre objectif est de mettre en évidence le champ d'application très large de la méthode. En effet, suivant les coefficients de time lag envisagés, notre heuristique permet de résoudre des problèmes de jobshop classique comme des problèmes de jobshop sans attente (no-wait) ou bien des problèmes de jobshop avec time lag. Ainsi, nous testons notre algorithme mémétique sur des instances variées.

Les notations suivantes sont utilisées dans les résultats :

- Opt(pour la valeur optimale),
- Sm (solution moyenne des 4 réplications),
- Ecart (Ecart=100(Sm-Opt)/Opt),
- Im (itération moyenne de la dernière amélioration en milliers d'itérations),
- Tm (temps moyen de la dernière amélioration en milliers d'itérations),
- TTm (temps moyen total de l'algorithme en secondes),
- S* (meilleure solution trouvée au cours des 4 réplications),
- I* (dernière amélioration de la meilleure réplication en millier d'itérations).

8.8.1 Petites Instances

Les instances suivantes sont suffisamment de petite taille pour que la formalisation linéaire proposée puisse être résolue de manière directe par un solveur. La colonne Opt donne le makespan de la solution optimale. Chaque ligne décrit donc les résultats pour une instance. Les colonnes sont regroupées en deux parties : les résultats moyens sur quatre réplications ainsi que la meilleure des quatre réplications.

Comme le montre le tableau 3-21, les résultats pour les instances la01 à la05 sont en moyenne à 4,76% de l'optimum pour la meilleure des 4 réplications alors que la moyenne est à 7,69%. Le tableau met en évidence que l'algorithme fournit de moins bons résultats quand le coefficient est égal à 0,5. Ceci est relativement étonnant car la première intuition est que la difficulté des problèmes diminue quand les coefficients de time lags augmentent. Or les résultats sont meilleurs pour le problème sans attente que pour le problème dont le coefficient est 0,5. Cette observation se répète pour d'autres algorithmes d'optimisation et n'est donc pas due à l'algorithme mémétique.

Le tableau 3-22 présente les résultats obtenus pour les instances de Carlier. Pour ces instances, toutes les solutions trouvées sont optimales. Les instances proposées par Carlier sont en fait des instances de flow-shop. Par contre, ce ne sont pas des instances de flowshop avec permutation et la combinatoire du problème est identique. Ceci explique pourquoi les résultats sont exceptionnellement bons. Pour l'instance ft06, les résultats sont détaillés dans le tableau 3-20. L'écart moyen pour cette instance est 1,37% pour la meilleure des 4 réplications et de 1,71% en moyenne. Cette instance de relativement petite taille résiste tout de même dans le cas sans attente (ft06_0_0) où la meilleure des 4 réplications est à 5,48%.

		Résultats moyens sur 4 rép.					**Meilleure des 4 rép.**		
	Opt.	**Sm**	**Ecart(%)**	**Im**	**Tm**	**TTm**	**S***	**Ecart(%)**	**I***
ft06_0	73	78	6,85	459	35,5	157,25	77	5,48	4
ft06_0,5	63	63	0,00	965	72	149,75	63	0,00	63
ft06_1	58	58	0,00	19	1	144,25	58	0,00	58
ft06_2	55	55	0,00	42	2,25	133,5	55	0,00	55
	Moyenne=		1,71			**Moyenne=**		1,37	

Tableau 3-20. Résultats de l'algorithme mémétique sur les instances de ft

		Résultats moyens sur 4 rép.					Meilleure des 4 rép.		
	Opt.	**Sm**	**Écart(%)**	**Im**	**Tm**	**TTm**	**S***	**Écart(%)**	**I***
la01_0	971	1018,25	4,87	1050	149	280	971	0,00	511024
la02_0	937	968	3,31	922	131	288	937	0,00	429503
la03_0	820	873,5	6,52	1182	144	288	820	0,00	243932
la04_0	887	953,25	7,47	1374	196	285	923	4,06	900095
la05_0	777	819	5,41	1186	144	256	797	2,57	72014
Moyenne=			5,51		153		**Moyenne=**	1,33	
la01_0,5	758	881	16,23	1076	149	278	867	14,38	511636
la02_0,5	742	883,5	19,07	590	80	285	872	17,52	230630
la03_0,5	679	735,75	8,36	1452	211	293	685	0,88	585881
la04_0,5	703	792	12,66	763	104	273	769	9,39	331708
la05_0,5	622	696,25	11,94	1030	135	260	678	9,00	513089
Moyenne=			13,65		136		**Moyenne=**	10,24	
la01_1	683	762,5	11,64	1227	164	265	723	5,86	680175
la02_1	686	739	7,73	1126	150	267	723	5,39	446195
la03_1	640	669,25	4,57	1299	206	278	641	0,16	284607
la04_1	646	688	6,50	1114	151	273	662	2,48	125350
la05_1	593	623,25	5,10	760	92	244	615	3,71	235834
Moyenne=			7,11		153		**Moyenne=**	3,52	
la01_2	666	666,5	0,08	755	86	230	666	0,00	269722
la02_2	655	691,75	5,61	1394	167	238	683	4,27	795287
la03_2	597	651,25	9,09	1196	150	253	648	8,54	768426
la04_2	590	635,75	7,75	700	83	245	631	6,95	151229
la05_2	593	593	0,00	347	36	207	593	0,00	130221
Moyenne=			4,51		104		**Moyenne=**	3,95	

Tableau 3-21. Résultats de l'algorithme mémétique sur les instances de la01 à la05

			Resultats moyens sur 4 rép.						Meilleure des 4 rép.		
	Opt.		**Sm**	**Ecart(%)**	**Im**	**Tm**	**TTm**		**S***	**Ecart(%)**	**I***
car5_0	9159		9159	0,00	150	22	52		9159	0,00	86121
car6_0	9690		9690	0,00	14	6	6		9690	0,00	2624
car7_0	7705		7705	0,00	2	0	0		7705	0,00	1399
car8_0	9372		9372	0,00	6	9	2		9372	0,00	4085
	Moyenne=			0,00			**Moyenne=**			0,00	
car5_0,5	7768		7788	0,26	180	41	255		7788	0,26	2963
car6_0,5	8648		8708	0,69	122	22	467		8648	0,00	11917
car7_0,5	6645		6645	0,00	4	0	270		6645	0,00	1681
car8_0,5	8452		8452	0,00	300	65	65		8452	0,00	2810
	Moyenne=			0,24			**Moyenne=**			0,06	
car5_1	7750		7750	0,00	130	24	24		7750	0,00	37388
car6_1	8323		8323	0,00	109	12	511		8323	0,00	2402
car7_1	6573		6573	0,00	53	6	6		6573	0,00	1265
car8_1	8279		8279	0,00	26	5	5		8279	0,00	11962
	Moyenne=			0,00			**Moyenne=**			0,00	
car5_2	7702		7723,25	0,28	998	163	289		7702	0,00	148462
car6_2	8313		8313	0,00	504	81	126		8313	0,00	55468
car7_2	6558		6572,75	0,22	18	2	58		6558	0,00	5975
car8_2	8264		8267,75	0,05	524	104	191		8264	0,00	238661
	Moyenne=			0,14			**Moyenne=**			0,00	

Tableau 3-22. Résultats de l'algorithme mémétique sur les instances de Carlier

8.8.2 Instances de grande taille

Pour les instances de taille supérieure, le nombre d'itération sans amélioration est doublé (*np=20 000*). Pour ces instances, on ne dispose pas de valeurs de comparaison car le makespan des solutions optimales ne peut être calculé et parce qu'aucune autre méthode connue ne permet de traiter ce problème. Pour tout de même évaluer les solutions, nous utilisons une borne inférieure du makespan de la solution. À notre connaissance, il n'existe pas de borne inférieure du problème de jobshop prenant en compte les time lag maximum. Ainsi, nous utilisons simplement la solution optimale du problème de jobshop en tant que borne inférieure du problème. Cette borne inférieure est de mauvaise qualité, d'autant qu'elle ne dépend pas des coefficients de time lag maximum.

Sur ces instances difficiles (5 machines, 15 jobs), la borne inférieure est atteinte pour des time lags maximaux avec un coefficient égal à 10. Une telle valeur de time lag correspond à un problème de jobshop classique car les contraintes de time lag sont désactivées tant les constantes sont grandes. Pour les coefficients de time lag inférieurs, il est difficile de conclure sur la qualité des résultats car la valeur de la borne inférieure est de mauvaise qualité.

		Resultats moyens sur 4 rép.					Meilleure des 4 rép.		
	LB	**Sm**	**Écart(%)**	**Im**	**Tm**	**TTm**	**S***	**Écart(%)**	**I***
la06_0	926	1409,75	52,24	806	341	779	1392	50,32	364471
la07_0	890	1359,75	52,78	1322	599	907	1329	49,33	819075
la08_0	863	1449,25	67,93	759	327	851	1385	60,49	340100
Moyenne=			57,65		422,42	**Moyenne=**		53,38	
la06_0,5	926	1228	32,61	1402	545	773	1153	24,51	928483
la07_0,5	890	1153	29,55	1445	573	784	1132	27,19	646648
la08_0,5	863	1195,5	38,53	1231	470	772	1164	34,88	707639
Moyenne=			33,56		529,25	**Moyenne=**		28,86	
la06_1	926	1115	20,41	1103	403	735	1101	18,90	777363
la07_1	890	1044,5	17,36	1444	532	736	1009	13,37	1166890
la08_1	863	1067	23,64	1463	541	742	1013	17,38	619330
Moyenne=			20,47		491,75	**Moyenne=**		16,55	
la06_2	926	1115	20,41	1103	405	739	1101	18,90	777363
la07_2	890	1034	16,18	1352	477	710	975	9,55	782532
la08_2	863	1067	23,64	1463	544	746	1013	17,38	619330
Moyenne=			20,08		475,25	**Moyenne=**		15,28	
la06_10	926	926	0,00	47	14	14	926	0,00	29008
la07_10	890	890	0,00	169	39	39	890	0,00	95097
la08_10	863	863	0,00	70	17	17	863	0,00	19470
	Moyenne=		0,00		23,17	**Moyenne=**		0,00	

Tableau 3-23. Résultats de l'algorithme mémétique sur les instances de grande taille

8.8.3 Comparaison aux instances sans attente

Le problème de jobshop avec time lags généralise le problème de jobshop sans attente. Dans le tableau 3-24, nous avons donc comparés nos résultats avec deux articles récents dédiés à l'optimisation du problème de jobshop sans attente. (Mascis et Pacciarelli 2002) propose des heuristiques de construction pour le problème de jobshop sans attente (colonne pdrs). Dans Schuster et Framinan (2003), les auteurs proposent un algorithme génétique couplé avec un recuit simulé (colonne GASA) ainsi qu'un algorithme à voisinage variable (Variable Neighborhood Search) (colonne VNS).

Comme on pouvait s'y attendre, notre algorithme mémétique surclasse les heuristiques de construction sur toutes les instances. De plus, notre algorithme mémétique surclasse aussi l'algorithme VNS. Ceci est plus remarquable, car l'algorithme VNS proposé par Schuster et Framinan est dédié à l'optimisation du problème de jobshop sans attente.

Pour finir, GASA (qui est à notre connaissance le meilleur algorithme pour le problème de jobshop sans attente) trouve les solutions optimales pour les instances la06 à la15. Notre algorithme mémétique est en moyenne à 4,2% de la solution optimale. Cette différence s'explique par le fait que notre algorithme traite un problème plus général.

		GA				VNS			GASA				pdrs	
Instances	**Meill. heur.**	**S***	**Sm**	**Tm**	**%**	**Sm**	**Tm**	**%**	**S***	**Sm**	**Tm**	**%**	**Sm**	**%**
la06_0_0	1339	1392	1410	341	3	1431	<1	6	1339	1407	3	0	1758	23
la07_0_0	1240	1329	1360	600	6	1366	<1	9	1240	1333	2	0	1609	22
la08_0_0	1296	1385	1449	327	6	1390	<1	6	1296	1400	2	0	1580	17
la09_0_0	1447	1504	1560	327	3	1586	<1	8	1447	1525	3	0	2430	40
la10_0_0	1338	1386	1432	562	3	1527	<1	12	1338	1435	2	0	1506	11
la11_0_0	1825	1903	1937	961	4	1915	<1	4	1825	1883	6	0	2226	18
la12_0_0	1631	1705	1728	806	4	1694	<1	3	1631	1686	5	0	1935	15
la13_0_0	1766	1843	1875	877	4	1907	<1	7	1766	1848	6	0	1867	5
la14_0_0	1805	1887	1943	1159	4	2313	<1	21	1805	1916	6	0	2381	24
la15_0_0	1829	1931	1960	1188	5	1898	<1	3	1829	1932	6	0	2134	14
	Moy.				4,2			7,9				0		18,9

Tableau 3-24. Résultats de l'algorithme mémétique pour les instances de nowait

8.8.4 Instance de jobshop classique

Instances	**Opt**	**S***	**Sm**	**Tm**	**%**
la06_inf	926	926	926	0	0
la07_inf	890	890	890	51	0
la08_inf	863	863	863	10	0
la09_inf	951	951	951	0	0
la10_inf	958	958	958	0	0
la11_inf	1222	1222	1222	1	0
la12_inf	1039	1039	1039	2	0
la13_inf	1150	1150	1150	1	0
la14_inf	1292	1292	1292	0	0
la15_inf	1207	1207	1209	104	0
	Moyenne =				0,0

Instances	**Opt**	**S***	**Sm**	**Tm**	**%**
ft06_inf	55	55	55	0	0
ft10_inf	930	946	972	669	1
	Moyenne =				0,5

car5_inf	7702	7731	7799	163	0
car6_inf	8313	8313	8313	127	0
car7_inf	6558	6558	6558	6	0
car8_inf	8264	8279	8311	85	0
	Moyenne =				0,0

Tableau 3-25. Résultats de l'algorithme mémétique sur les instances de jobshop

L'algorithme mémétique proposé est aussi testé sur les instances de jobshop classique. Pour transformer une instance de jobshop avec time lag en une instance de jobshop classique. Il suffit de choisir des coefficients de time lag minimum nuls et des coefficients de time lag maximum suffisamment élevés.

On peut donc lancer notre algorithme mémétique pour ces instances. Par contre, l'évaluation d'un ordre d'opérations n'est pas adaptée au problème de jobshop classique car elle est notablement inefficace. De plus, les réglages de l'algorithme ont été conservés pour montrer les performances de cet

algorithme. Il est probable qu'avec des réglages plus adaptés à l'espace de recherche du problème de jobshop, l'algorithme puisse trouver de meilleures solutions.

Dans le tableau 3-25, les solutions de l'algorithme mémétique pour les instances de jobshop sont présentées. Toutes les solutions sont résolues de manière optimale sauf l'instance ft10 dont la solution trouvée est à 1%. Cette solution est connue pour être difficile.

8.9 Conclusion sur l'algorithme mémétique

Les résultats de l'algorithme mémétique sont satisfaisants, ils outrepassent les résultats de l'algorithme tabou pour les instances avec time lags serrés. Mais mesurer la qualité des résultats reste difficile car cet algorithme est le premier proposé pour le problème de jobshop avec time lag. Les résultats proposés ne peuvent donc être comparés ni à d'autres méthodes ni à des bornes inférieures de qualité. Ainsi, nous avons comparé les résultats de l'algorithme mémétique sur les cas particuliers que sont le jobshop sans attente et le jobshop classique. Les résultats montrent que pour ces instances, l'algorithme proposé n'est pas performant en termes de temps de calcul mais est tout à fait performant en termes de qualité de solution trouvée. Ces résultats sont remarquables d'autant que l'algorithme couvre un large spectre de problèmes.

Pendant les expérimentations, nous avons remarqué que de nombreuses solutions irréalisables sont envisagées. De plus, les solutions ayant un plus faible nombre de time lags violés sont choisies prioritairement, ce qui fait que la population est majoritairement constituée de solutions avec un faible nombre de time lags violés. Dans la section suivante, nous avons expérimenté un autre algorithme à population : un algorithme bi-objectif. La principale différence de cet algorithme réside dans la gestion de la population qui maintient une plus grande diversité en termes de makespan et de nombre de time lags violés.

9 Proposition d'un algorithme bi-objectif

Dans cette section, nous proposons d'utiliser un algorithme bi-objectif afin de prendre en compte nos deux objectifs principaux : que la solution soit réalisable en minimisant le nombre de time lags violés, et que la solution soit de bonne qualité en minimisant le makespan. L'algorithme que nous proposons est un algorithme génétique bi-objectif basé sur l'algorithme NSGA II. Les deux objectifs de cet algorithme sont de minimiser le makespan et le nombre de time lags violés. Cet algorithme a été proposé dans (Caumond *et al.*, 2005b).

Ainsi, cet algorithme optimise un ensemble de solutions réalisables et irréalisables. Le fait d'avoir et de maintenir ces deux populations a un double intérêt. D'un point de vue résolution, les solutions irréalisables permettent de garder une certaine diversité. En effet, les meilleures solutions pour l'algorithme bi-objectif sont celles qui ont le plus faible makespan pour un nombre donné de time lags violés. D'un point de vue pratique, les solutions violant des time lags maximums indiquent au décideur quel est le gain potentiel s'il relâche une ou plusieurs contraintes de time lag maximum.

9.1 Introduction à l'optimisation multi-objectif

Un problème d'optimisation multi-objectif consiste à trouver la meilleure solution pour plusieurs objectifs. On trouve trois approches principales dans la littérature pour résoudre de tels problèmes : la transformation du problème multi-objectif en un problème mono-objectif (approche d'agrégation adoptée par l'algorithme mémétique ci-dessus), l'optimisation indépendante de chacun des objectifs et l'optimisation conjointe des objectifs à l'aide d'une approche de type Pareto.

Dans cette dernière approche, plusieurs critères sont envisagés simultanément pendant l'optimisation, il n'est donc pas immédiat de comparer deux solutions. Dans l'approche de type Pareto, on utilise la notion de dominance qui se définit comme suit : on dit qu'une solution en domine une autre si elle est meilleure ou égale sur les deux critères. La dominance n'est pas une relation antisymétrique car deux solutions peuvent être incomparables (i.e. aucune des deux ne domine l'autre).

Le fait qu'il existe des solutions incomparables complexifie les méthodes de résolution car il n'existe plus une meilleure solution mais un ensemble de meilleures solutions.

Le but des optimisations de type Pareto est de construire l'ensemble des solutions non dominées, c'est-à-dire l'ensemble des solutions tel qu'il n'existe pas de solutions les dominant. L'algorithme d'optimisation construit alors cet ensemble et le fait évoluer en diversifiant les solutions non dominées. Les solutions ainsi obtenues sont appelées "front de Pareto".

L'article de Ehrgott et Gandibleux (2002) et les livres de Deb (2001), Coello *et al.* (2002) et (Gandibleux *et al.*, 2006) proposent un état de l'art pour les algorithmes génétiques multi-objectif. Parmi ces algorithmes, NSGA II est un algorithme largement utilisé et reconnu dans la littérature. NSGA II est un algorithme génétique dont la population est triée à l'aide d'un tri non dominé. A l'issue de ce tri, l'on dispose de fronts, le premier front contenant tous les points tels qu'aucune solution de la population n'est meilleure sur tous les critères.

Récemment, l'algorithme NSGA-II a été utilisé pour le problème de tournées sur arcs (CARP) proposé par Lacomme *et al.* (2006). Les auteurs ont montré que des solutions de bonne qualité peuvent être trouvées en un temps de calcul restreint à l'aide d'algorithmes de type NSGA-II. En effet, leur algorithme est compétitif avec les algorithmes mono-objectifs classiques.

9.2 Algorithme bi-objectif pour le jobshop avec time lag

L'algorithme bi-objectif proposé utilise la représentation stratifiée présentée dans la section 6.2. Parmi les trois critères utilisés par cette représentation, nous utilisons les critères de makepsan et de nombre de time lags violés. Ces deux critères sont les deux objectifs de notre algorithme.

En ce qui concerne la structure générale de l'algorithme, nous utilisons la structure proposée par NSGA-II. Une version détaillée de l'algorithme de principe est présentée dans (Lacomme *et al.*, 2006). Pour construire notre algorithme génétique, nous avons donc à définir un opérateur de croisement, un opérateur de mutation, une procédure pour la gestion des clones, une recherche locale directionnelle et un ensemble de solutions initiales choisies. L'algorithme fonctionne alors de la manière suivante : les individus de la population sont triés suivant un tri non dominé, les points extrêmes sont calculés pour pouvoir appliquer la recherche locale directionnelle, cette recherche locale est appliquée avec une certaine probabilité, l'individu obtenu est inséré dans la population s'il n'est pas un clone. La génération des fils continue jusqu'à ce que la population soit doublée, la meilleure moitié est alors conservée. Cet algorithme est itératif et se termine au bout d'un certain nombre d'itérations.

- L'opérateur de croisement que nous utilisons est l'opérateur GOX (i.e. le même que celui présenté dans l'algorithme ci-dessus (§8.2)).
- L'opérateur de mutation est appliqué de manière périodique à la population. Cet opérateur effectue l'une des trois modifications suivantes : avec une probabilité 0,5 une recherche locale directionnelle est appliquée. Comme le montre (Lacomme *et al.*, 2006), une recherche locale directionnelle peut être appliquée aux fronts pour obtenir un bon compromis entre la diversification et l'intensification. Avec la probabilité 0,25, 40 itérations d'une descente stochastique sont effectués. Les deux recherches locales précédentes utilisent le voisinage de Nowicki et Smutnicki (1996) pour le problème de jobshop avec time lags. Enfin, avec la probabilité 0,25, une procédure de réparation est effectuée. Cette procédure consiste à choisir un job parmi les jobs dont une opération de time lags maximum est violée. Ce job est choisi de manière aléatoire en pondérant le choix en fonction du nombre de time lags maximum violés. Puis, ce job est supprimé de l'ordre d'opérations et réinséré à la fin de l'ordre. Ce faisant, on obtient un ordre dans lequel aucun nouveau time lag maximum n'est violé et dans lequel le job choisi n'a plus de time lag maximum violé.
- La procédure de gestion des clones a pour objectif de détecter les solutions équivalentes de la population. L'idée principale de cette procédure consiste à calculer une signature pour chaque solution de la population. La signature est calculée en utilisant la fonction de hachage quadratique des dates de début des opérations ($\sum_{i \in O} t_i^2 \bmod K$ où K est un nombre suffisamment grand, K=499999). La signature permet de comparer deux

solutions de la population. Si les deux solutions sont identiques, leur signature est identique. Si deux solutions sont codées différemment (ordres différents des opérations), mais que ces deux solutions mènent au même ordonnancement, alors leur signature est aussi identique. Lorsque deux solutions sont détectées comme étant identiques, elles sont considérées comme des clones. En calculant et en mettant à jour un grand tableau, on peut détecter en O(1) si un individu de même signature est présent dans la population.

- Finalement, l'ensemble des solutions initiales contient les solutions heuristiques générées par les règles de priorités. Cet ensemble est complété par des solutions générées aléatoirement jusqu'à ce que la population contienne 200 individus.

9.3 Expérimentations numériques

Le but des expérimentations numériques pour l'algorithme bi-objectif est double : évaluer la qualité des solutions trouvées en les comparant avec les meilleurs algorithmes publiés. L'intérêt de l'algorithme bi-objectif est aussi de construire des fronts de bonne qualité pour disposer de solutions non dominées.

9.3.1 Analyse des fronts obtenus

Lors du déroulement de l'algorithme, on organise la population de manière à constituer des fronts. La figure 3-15 illustre graphiquement les populations initiales et finales et montre que les fronts obtenus sont satisfaisants. La population initiale est constituée de deux paquets, les solutions générées par les règles de priorité sont toutes réalisables, elles sont donc toutes sur l'axe des abscisses. Le deuxième paquet de solutions est généré de manière complètement aléatoire, ces solutions ont entre 20 et 33 time lags violés et leur makespan est supérieur à 1200. Ce graphique illustre le faible nombre de solutions réalisables, car toutes les solutions générées aléatoirement ont un grand nombre de time lags violés. Dans la partie droite de la figure, les fronts obtenus par l'algorithme bi-objectif sont représentés graphiquement. Ces fronts montrent que l'algorithme bi-objectif a réussi à générer des fronts de bonne qualité. Entre autres, l'espacement des individus le long du front non dominé est satisfaisant car il y a un individu par nombre de time lags violés.

En examinant le meilleur front, on obtient la meilleure solution trouvée pour chaque nombre de time lag maximum violé. Le meilleur front de la figure montre donc qu'une solution de makespan 926 a été trouvée pour 6 time lags maximaux violés. Or, cette valeur de makespan est la valeur optimale pour le problème de jobshop sans contrainte de time lag. Ainsi, on ne peut trouver de meilleure solution en augmentant le nombre de time lag violé. Lorsque le nombre de time lag violé diminue, le makespan de la meilleure solution augmente. La meilleure solution ne violant aucun time lag a 1017 pour makespan, ce qui est meilleur que la meilleure solution connue pour cette instance.

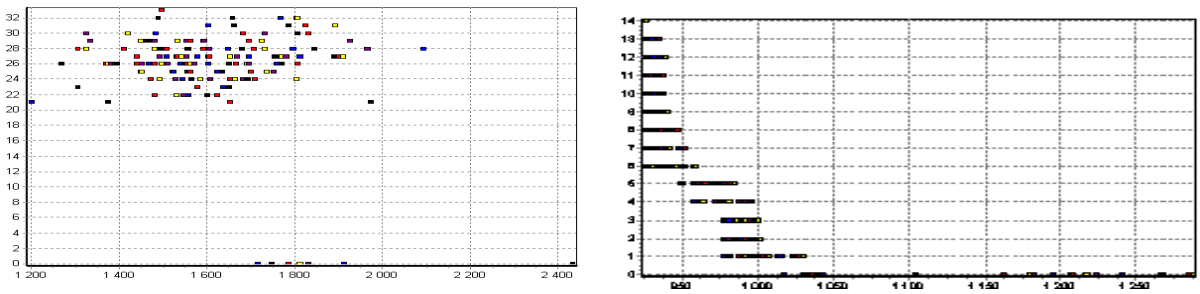

Figure 3-15. Population initiale et fronts finaux obtenus par l'algorithme bi-objectif pour la06_0_1

9.3.2 Résultats détaillés

	Contraintes Faibles				Contraintes Fortes				
Instance	**LB**	**MS**	**# TL violés**	**Écart Relatif**	**Algo. Géné.**	**MS**	**# TL violés**	**Écart Relatif**	**Temps Total**
la06_0	926	1010	19	9,07%	1049	1470	0	4,33%	811
la07_0	890	1022	12	14,83%	1359	1385	0	1,91%	873
la08_0	863	1018	19	17,96%	1449	1591	0	9,80%	810
la06_0,5	926	968	11	4,54%	1228	1167	0	-4,97%	666
la07_0,5	890	959	11	7,75%	1153	1130	0	-1,99%	784
la08_0,5	863	910	10	5,45%	1195	1118	0	-6,44%	686
la06_1	926	926	6	0,00%	1115	1017	0	-8,79%	613
la07_1	890	959	9	7,75%	1044	1039	0	-0,48%	656
la08_1	863	899	6	4,17%	1067	988	0	-7 40%	545
la06_2	926	926	6	0,00%	1115	1017	0	-8,79%	619
la07_2	890	894	8	0,45%	1034	1086	0	5,03%	654
la08_2	863	899	6	4,17%	1067	988	0	-7,40%	544
			Moy.	6,35%			**Moy**.	-2,10%	

Tableau 3-26. Résultats détaillés de l'algorithme bi-objectif pour les instances la06 à la08

Les résultats détaillés pour les instances la06 à la08 sont donnés dans le tableau 3-26. Pour chaque instance, le tableau donne les deux points extrêmes d'un front : le point "weak constraints" qui minimise d'abord le makespan puis le nombre de time lags violés, et le point "strong constraints" qui minimise le nombre de time lags violés puis le makespan. Le tableau est donc organisé en deux parties. Dans la partie "weak constraints", la colonne LB donne la valeur de la solution optimale du problème de jobshop classique, dans la partie "strong constraints", la colonne "GA" donne la meilleure solution obtenue par l'algorithme mémétique proposé dans (Caumond *et al.*, 2005a). Dans les deux parties, les colonnes "ms" donnent le makespan, les colonnes "#vtl" sont le nombre de time lags violés et les colonnes "rel. dev." donnent l'écart relatif de la solution par rapport à la valeur de référence (LB pour weak constraint et GA pour strong constraint). Pour finir, la colonne "Total time" donne la durée totale d'exécution de l'algorithme.

Même si nous l'atteignons dans quelques cas, la borne inférieure n'est pas forcément atteignable. En effet, cette borne correspond au problème sans time lags, or les contraintes de time lags ne sont supprimées que si elles mènent à une irréalisabilité. Ainsi, la borne inférieure n'est pas toujours de bonne qualité et les résultats à 17,96% (instance la08_0) peuvent être proches de la solution optimale.

Les durées d'exécution des algorithmes mémétiques et bi-objectif sont équivalentes. Le tableau montre donc que l'algorithme bi-objectif est compétitif avec l'algorithme mémétique. Pour plusieurs instances, les résultats de NSGA-II sont même meilleurs que l'algorithme mémétique. En résumé, on peut dire que l'algorithme bi-objectif obtient des meilleurs résultats sauf pour les instances à time lags nuls.

9.4 Conclusion sur l'algorithme bi-objectif

L'algorithme bi-objectif proposé est compétitif, il permet même de trouver de nouveaux records pour plusieurs instances. De plus, ces résultats valident ceux trouvés par l'algorithme mémétique car

deux algorithmes relativement différents trouvent des résultats de qualité comparable sur les mêmes instances. Ceci ne constitue pas une preuve de la qualité des solutions trouvées mais rassure sur leur pertinence.

10 Proposition d'une heuristique de construction

Les trois méthodes d'optimisation proposées ci-dessus sont des algorithmes itératifs. Ce type d'algorithme fournit des solutions de bonne qualité mais le temps requis est généralement long. Ci-après, nous proposons une heuristique de construction qui permet de calculer rapidement une solution réalisable de bonne qualité. Cette méthode complète l'éventail des algorithmes proposés. Elle permet de trouver rapidement une solution réalisable et de bonne qualité. Ceci est particulièrement utile dans un contexte temps réel, ou pour démarrer une méthode d'optimisation plus complexe.

De plus, nous allons montrer que cette heuristique fournit des résultats d'une qualité au moins égale aux heuristiques équivalentes pour le problème de jobshop et qu'elle permet aussi de trouver des solutions forcément réalisables. Puisque cette heuristique peut trouver à priori n'importe quelle solution de l'espace de recherche, elle est donc particulièrement intéressante.

Dans cette section, nous présentons une heuristique pour le problème de jobshop avec time lag, cette heuristique a été proposée dans (Caumond *et al.*, 2005c). Cette heuristique utilise la représentation basée sur les règles de priorité présentée dans le paragraphe 6.3. Les algorithmes de principe de notre heuristique sont donc présentés dans ce paragraphe. Une solution est donc calculée itérativement et à chaque étape, une opération doit être choisie dans l'ensemble des opérations "restreintes". Le choix de cette opération peut être fait suivant plusieurs critères. Dans cette heuristique, le choix d'une opération est fait suivant une règle de priorité. Ce type d'heuristique est largement répandu dans la littérature pour de nombreux problèmes d'ordonnancement. Dans cette partie, nous proposons de construire une heuristique basée sur les règles de priorité qui construit des solutions forcément réalisables.

10.1 Heuristiques de la littérature

Des heuristiques de construction ont été proposées par Deppner (2004). Ces heuristiques ont été développées pour un problème général d'ordonnancement et utilisent les règles de priorité. Dans sa thèse, Deppner étudie le cas où les time lags sont présents entre n'importe quelle paire d'opérations. Une version modifiée de Giffler et Thompson est proposée et une adaptation de l'algorithme à ordre strict est aussi développée. Le problème traité est donc plus général que le jobshop avec time lags. Pour ce problème, l'existence d'une solution réalisable n'est pas systématique et trouver une solution réalisable est NP-difficile. Ces difficultés théoriques font que l'heuristique proposée souffre des défauts suivants : les ordonnancements générés ne sont pas systématiquement actifs ni même semi-actifs, de plus, au-delà d'un certain seuil, les contraintes de time lag sont relâchées. Ainsi, même pour les problèmes sur-contraints, l'heuristique propose tout de même une solution. L'auteur propose aussi des heuristiques basées sur les grappes d'opérations. Dans cette approche, l'ensemble des opérations est décomposé en grappes dont chacune est ordonnancée seule. L'ordonnancement complet est ensuite construit en arrangeant les grappes entre elles. Dans ces travaux, aucun résultat n'est publié pour les problèmes de jobshop classique.

Pour le cas particulier du problème de jobshop sans attente, Mascis et Pacciarelli (2002) ont montré que compléter de manière réalisable toute solution partielle réalisable est un problème NP-complet. En d'autres termes, lors du déroulement d'un algorithme de construction, on ne peut assurer que toute solution partielle pourra être complétée de manière réalisable. Ce résultat a été prouvé même dans le cas d'un jobshop à trois machines et temps de traitement unitaire. C'est pourquoi nous envisageons une procédure de réparation pour transformer les solutions partielles que l'on ne peut compléter.

10.2 Détails de notre heuristique

Pour construire notre heuristique, nous utilisons la représentation par règles de priorités présentée dans le paragraphe 6.3. Cette représentation nécessite des règles pour déterminer quelle opération doit être ordonnancée. Ainsi, pour déterminer une heuristique à partir de cette représentation, il suffit de définir une règle de priorité.

L'heuristique que nous proposons est la première heuristique permettant de générer des solutions systématiquement réalisables. Pour cela, elle se base sur le module d'évaluation 3 par règles de priorité. Ce module d'évaluation est construit à partir de l'algorithme de Giffler et Thompson. Les principales modifications apportées portent sur l'évaluation des dates de début et sur la procédure de réparation.

Pour cela, nous utilisons les règles de priorité déjà publiées dans la littérature. En particulier, Haupt (1989) décrit de nombreuses règles de priorité pour les problèmes d'ordonnancement. Dans la suite, nous utilisons les règles suivantes :

- FIFO (First In First Out) : choisit la première opération à être disponible (i.e. plus faible date de début au plus tôt).
- SPT (Shortest Processing Time) : choisit l'opération de plus faible temps de traitement
- LPT (Longest Processing Time) : choisit l'opération de plus grand temps de traitement
- LWKR (Less WorK Remaining) : choisit l'opération qui nécessite le moins de temps de traitement (opération y compris)
- MWKR (Most WorK Remaining) : choisit l'opération qui nécessite le plus de temps de traitement (opération y compris)
- FOPNR (Fewest number of OPeratioNs Remaining) : choisit l'opération pour laquelle il reste le moins d'opérations à suivre dans la gamme
- GOPNR (Greatest number of OPeratioNs Remaining) : choisit l'opération pour laquelle il reste le plus d'opération à suivre dans la gamme
- TWORK (Total WORK) : choisit l'opération dont le job a la plus grande durée totale de traitement
- EDD (Earliest Due Date) : choisit l'opération dont le job aura la plus faible durée de traitement quand l'opération sera terminée
- WINQ (Work IN the Queue) : choisit l'opération pour laquelle il y a le plus de jobs à traiter sur la machine
- SRMPT (Shortest ReMaining Processing Time) : choisit l'opération pour laquelle le minimum des temps de traitement des opérations suivantes dans la gamme est le plus faible.
- LRMPT (Longest ReMaining Processing Time) : choisit l'opération pour laquelle le minimum des temps de traitement des opérations suivantes dans la gamme est le plus élevé.
- SPT_TWORK (Shortest Processing Time divided by Total Work) : choisit l'opération dont le quotient suivant est le plus faible : temps de traitement de l'opération divisé par temps total de traitement.
- FCFS (First Come First Serve) : choisit l'opération qui peut être ordonnancée le plus tôt.

Quelle que soit la règle retenue, des cas d'égalité peuvent subvenir. Lorsque cela arrive, un deuxième critère de choix doit être clairement retenu. Nous avons choisi de retenir l'opération avec le plus faible numéro identificateur. Ainsi, si les opérations *2* et *3* ont la même durée ($p_2 = p_3 = 5$) alors la règle SPT choisit l'opération *2*.

L'algorithme général des heuristiques de construction est un algorithme polynomial faisant appel à des procédures. Nous montrons que toutes ces procédures sont polynomiales :

- La procédure Evaluate est basée sur le calcul du plus long chemin que nous avons proposé. Cet algorithme est polynomial.

- La procédure Rule consiste à choisir une opération dans un ensemble. Les règles que nous utilisons sont toutes polynomiales. Remarque : il est à priori possible d'envisager des règles non polynomiales
- A procédure Update consiste en l'algorithme de plus long chemin auquel on ajoute la procédure de réparation. L'algorithme de plus long chemin est polynomial et l'algorithme de réparation est borné par $O(o)$ fois la complexité de l'algorithme de plus long chemins où o est le nombre d'opérations. Cet algorithme est donc polynomial.

Ainsi l'heuristique proposée est polynomiale pour toutes les règles de priorité polynomiales. Puisque l'heuristique permet de construire que des solutions réalisables, nous montrons donc par le fait que la découverte de solutions réalisables est un problème polynomial.

De plus, cette heuristique permet d'envisager n'importe quelle solution réalisable. Pour prouver cette affirmation, envisageons une solution réalisable quelconque S. Pour définir l'heuristique, il faut préciser une règle et nous envisageons la règle suivante : entre deux opérations, on choisit l'opération réalisée en premier dans l'ordonnancement *S*. Cette règle associée à la restriction *Semi active*, permet de reconstruire l'ordonnancement *S* à l'aide de l'heuristique. Ainsi, nous avons prouvé que l'heuristique peut trouver à priori n'importe quelle solution réalisable.

L'heuristique proposée est donc particulièrement performante pour le problème de jobshop, elle permet de trouver que et toutes les solutions réalisables.

10.3 Expérimentations numériques

Pour conduire les expérimentations numériques, nous procédons en quatre étapes : la section 10.3.1 tente de déterminer l'influence du time lag maximum sur les solutions optimales, la section 10.3.2 présente les résultats détaillés pour une petite instance dont on connaît les solutions optimales, la section 10.3.3 présente les résultats de l'heuristique et la dernière section 10.3.4 évalue les performances de l'heuristique dans le cas particulier des problèmes sans attente.

10.3.1 Influence des time lags maximums

Dans cette partie, nous résolvons des instances du problème de jobshop avec l'heuristique classique de Giffler et Thompson. On peut ramener ces instances à des instances de jobshop avec time lags minimum nuls et time lags maximum infinis. En diminuant la valeur du coefficient, on souhaite déterminer à partir de quelle valeur les solutions trouvées par l'heuristique classique ne peuvent plus être corrigées pour prendre en compte le time lag. Ces expérimentations montrent qu'il est vain de tenter d'utiliser les solutions d'un algorithme d'optimisation ne prenant pas en compte les time lags.

D'abord, les résultats du problème de jobshop classique sont présentés dans le tableau 3-27. La performance de chaque règle de priorité est la moyenne sur les 20 instances de l'écart à la solution optimale (ecart = (valeur trouvée-optimale)*100/optimale). La meilleure solution pour la meilleure règle est à environ 20%. Ce résultat corrobore les résultats précédemment publiés sur la qualité des algorithmes basés sur les règles de priorité.

Règle	Écart Moy.	Écart Max.	Écart Min.
FCFS	23,4%	39,2%	5,6%
SPT	48,6%	70,9%	33,2%
EDD	24,2%	52,7%	13,2%
LWKR	54,2%	80,1%	27,3%
Moy. :	37,6%	60,8%	19,8%

Tableau 3-27. Performances de l'heuristique pour le problème de jobshop

Règle	TLmax Moy.	TLmax Max.	TLmax Min.
FCFS	0	0	0
SPT	12	4	25
EDD	9	1	19
LWKR	6	2	11
Moy. :	7	2	14

Tableau 3-28. Valeur minimale du coefficient de time lag maximum pour obtenir une solution réalisable

Dans le tableau 3-28, on calcule la plus faible valeur du coefficient de time lag maximum (TLmax) qui mène à une solution réalisable. En d'autres termes, l'heuristique est d'abord exécutée sans prendre en compte les time lags maximum. L'ordre des opérations dans l'ordonnancement obtenu est conservé. Puis, le coefficient de time lag maximum est itérativement décrémenté jusqu'à ce que l'ordre d'opérations ne soit pas réalisable (i.e. ne soit compatible avec les contraintes de time lag maximum).

Par exemple, la plus grande valeur de TLmax est 0 pour la règle FCFS, ce qui indique que les ordonnancements construits avec cette règle sont réalisables quelle que soit l'instance considérée. Ce résultat en particulier s'explique par le fait que la règle FCFS tend à choisir en premier l'opération qui peut être commencée le plus tôt. En choisissant cette opération, on diminue les risques d'obtenir un ordonnancement irréalisable.

Le tableau 3-28 regroupe les résultats par règle plutôt que par instance. Ce tableau montre donc que toutes les règles ne sont pas équivalentes quand on traite de la contrainte de time lag. Ce résultat est spécifique à la contrainte de time lag. Pinson (1997) souligne au contraire qu'aucune règle n'est plus performante que les autres dans le cas général.

Un exemple frappant est celui de la règle SPT. Cette règle est souvent utilisée dans la littérature, mais le tableau 3-28 montre qu'elle fournit des ordonnancements particuliers dans lesquels les écarts entre opérations consécutives sont très grands. En effet, pour la pire des instances, un coefficient de time lag à 24 et moins conduit à un ordonnancement irréalisable. Cette remarque illustre les modifications profondes qui ont lieu dans l'espace de recherche.

10.3.2 Analyse des résultats pour l'instance ft06

Dans cette section, nous présentons les résultats de l'heuristique pour l'instance ft06. Cette instance est de taille relativement faible (36 opérations et 30*2 time lags). Notre objectif est de montrer que trouver des solutions réalisables n'est pas évident et que la procédure de réparation ne construit pas des solutions spécifiques. En effet, la procédure de réparation introduit des trous dans l'ordonnancement. Mais des trous semblables sont présents dans la solution optimale.

Les résultats de l'heuristique sont analysés en les comparant aux ordonnancements canoniques et à des ordonnancements générés de manière aléatoire. Les ordonnancements générés de manière aléatoire sont construits de la manière suivante : d'abord un ordre valide d'opérations est construit et le meilleur ordonnancement respectant cet ordre est construit grâce au graphe conjonctif - disjonctif.

Le tableau 3-29 fournit les résultats détaillés pour l'instance ft06 avec différentes valeurs de time lag. La colonne 2 donne pour chaque instance la valeur du makespan optimal. Le makespan de la solution de la meilleure règle et son écart à la solution optimale sont donnés dans les colonnes 3 et 4. Les colonnes 5 et 6 fournissent le résultat du meilleur ordonnancement canonique tiré aléatoirement parmi un millier. Enfin, les colonnes 7 à 9 donnent la meilleure des solutions tirées aléatoirement.

Aucune de ces solutions n'est réalisable alors qu'en supprimant les contraintes de time lag maximum, toutes le seraient. Ces résultats montrent que trouver une solution réalisable est difficile même pour une instance de taille modérée.

De plus, le tableau 3-29 montre que les résultats obtenus sont largement supérieurs aux solutions canoniques. En effet, en tirant aléatoirement mille ordonnancements canoniques, la meilleure des solutions trouvées (Colonnes 5 et 6) est à 99,9% en moyenne de l'optimum. Alors que l'heuristique que nous proposons est à 26,1%.

		règles		**Ordonnancement canonique**		**Ordonnancement stochastique**		
Instance	**Opt.**	**Meill.**	**Écart**	**Meill.**	**Écart**	**Meill.**	**Écart**	**# réalisable**
ft06_0_0	73	83	13,7%	123	68,5%	∞	∞	0%
ft06_0_0,5	63	109	73,0%	123	95,2%	∞	∞	0%
ft06_0_1	58	63	8,6%	123	112,1%	∞	∞	0%
ft06_0_2	55	60	9,1%	123	123,6%	∞	∞	0%
	Moy.		26,1%		99,9%			0,0%

Tableau 3-29. Résultats de l'heuristique pour l'instance ft06

Figure 3-16. Diagramme de Gantt pour l'instance ft06_0_1

La figure 3-16 présente deux ordonnancements pour l'instance ft06_0_1 : l'ordonnancement de la meilleure règle de priorité et l'ordonnancement optimal calculé par programmation linéaire. Ces ordonnancements montrent que les ordonnancements optimaux ne sont pas compacts et que de nombreuses fenêtres de temps d'inactivité sont présentes dans l'ordonnancement optimal. Ainsi, les temps morts insérés par la procédure de réparation semblent nécessaires pour obtenir des solutions réalisables.

10.3.3 Heuristique avec réparation

Le tableau 3-30 fournit les résultats détaillés pour les instances la01 à la05 de Laurence en considérant des coefficients de time lag de 0 à 2. Pour chaque instance (et donc sur chaque ligne), la case de la meilleure règle est grisée. Aucune colonne n'est significativement meilleure que les autres.

Toutes les règles ont entre deux et quatre records sur les instances envisagées. Seules les règles SPT, LPT GOPNR et TWORK sont toujours pires que les autres. Il est difficile de conclure que ces règles sont vraiment moins bonnes que les autres, des tests plus complets devraient être envisagés.

La deuxième colonne (opt/rec.) donne la meilleure solution trouvée pour chaque instance. Lorsque la valeur est surmontée d'une étoile, c'est que cette solution est optimale, sinon, la solution est la meilleure solution connue.

Les colonnes 4 à 17 donnent les solutions de chaque règle pour chaque instance. La synthèse de ces résultats est donnée dans la colonne 3. La colonne 3 donne l'écart à la solution optimale de la meilleure règle. En fonction, de la valeur du coefficient du time lag maximum, cet écart varie de 52,1% pour les instances de type sans attente, jusqu'à 33,6% pour les instances dont le coefficient de time lag maximum est 2. Les instances les moins bien résolues sont les instances avec un faible mais non nul coefficient de time lag maximum.

	opt/		Meilleur	Règles													
Instance	**rec.**		**écart**	**FIFO**	**SPT**	**LPT**	**LWKR**	**MWKR**	**FOPNR**	**GOPNR**	**TWORK**	**EDD**	**WINK**	**SRMPT**	**LRMPT**	**SPT TWORK**	**FCFS**
la01_0_0	971	*	54,9%	1880	2270	1837	1672	1618	1993	1880	1882	1762	1624	2020	2020	1504	1752
la02_0_0	937	*	51,1%	1416	1655	1683	1638	1511	1618	1454	1604	1835	1445	1822	1822	1652	1416
la03_0_0	820	*	45,4%	1661	1338	1417	1240	1429	1192	1584	1337	1305	1364	1361	1361	1328	1248
la04_0_0	887	*	51,7%	1416	1752	1767	1346	2007	1787	1938	1946	1593	1735	1774	1774	1602	1709
la05_0_0	777	*	57,5%	1224	1543	1671	1574	1548	1444	1423	1339	1286	1333	1464	1464	1583	1457
moy:			52,1%														
la01_0_0,5	758	*	94,5%	1824	1642	1737	1494	1823	1665	1824	2163	1486	1474	1620	1620	1846	1489
la02_0_0,5	742	*	62,7%	1239	1710	1428	1358	1734	1358	1503	1823	1207	1538	1386	1386	1436	1368
la03_0_0,5	679	*	59,8%	1328	1569	1147	1374	1460	1085	1435	1441	1113	1380	1285	1285	1280	1328
la04_0_0,5	703	*	64,4%	1223	1365	1795	1308	1623	1156	1296	1558	1196	1391	1156	1156	1511	1203
la05_0_0,5	622	*	94,2%	1266	1296	1554	1475	1208	1229	1348	1505	1353	1249	1217	1217	1274	1404
moy:			75,1%														
la01_0_1	683	*	63,1%	1606	1483	1600	1114	1615	1146	1606	1583	1322	1543	1130	1130	1404	1277
la02_0_1	686	*	65,6%	1252	1267	1391	1350	1579	1530	1210	1764	1178	1544	1505	1505	1233	1136
la03_0_1	640	*	45,5%	1355	1294	1508	1002	1424	1005	1161	1224	1039	931	1091	1091	1014	1355
la04_0_1	646	*	32,7%	1397	1027	1592	1016	1282	955	1374	1410	1102	1217	857	857	1506	972
la05_0_1	593	*	62,6%	1309	1113	1134	1027	1297	964	1156	1558	973	1426	982	982	1158	1061
moy:			53,9%														
la01_0_2	666	*	42,3%	1542	1033	957	1117	1137	1227	1545	1882	1130	948	1046	1046	1182	1452
la02_0_2	655	*	36,6%	1321	907	1256	1038	1412	1100	1436	1469	993	895	978	978	1104	906
la03_0_2	597	*	31,8%	1240	974	1403	929	1217	979	1098	1494	864	991	886	886	1101	787
la04_0_2	590	*	42,0%	1012	1098	910	1004	996	1069	1292	939	1020	899	939	939	1263	838
la05_0_2	593	*	15,2%	929	1103	1336	907	1171	861	1067	786	683	734	921	921	1142	1146
moy:			33,6%														

Tableau 3-30. Résultats détaillés de l'heuristique pour les instances de Laurence

		Ordo.	Ordo. aléatoire.		Règles
	Inst.	**canon.**	**Écart.**	**# réal.**	**Écart.**
la01-05	0	75,9%	nf	0%	52,1%
	0,5	121,4%	nf	0%	75,1%
	1	132,0%	nf	0%	53,9%
	2	145,1%	nf	0%	33,6%
Moyenne :		129,8%		0,0%	54,1%
car05-08	0	9,6%	nf	0%	15,2%
	0,5	10,3%	nf	0%	13,2%
	1	9,8%	nf	0%	13,5%
	2	7,7%	nf	0%	14,3%
Moyenne :		9,4%	nf	0,0%	14,0%
la06-08	0	71,5%	nf	0%	61,1%
	0,5	104,1%	nf	0%	63,2%
	1	126,3%	nf	0%	56,6%
	2	130,2%	nf	0%	58,5%
	10	167,2%	23,8%	10%	8,5%
Moyenne :		119,8%	nf	2,0%	49,6%

Tableau 3-31. Résultats moyens de l'heuristique pour les instances de Laurence

Le tableau 3-31 est une synthèse des résultats pour les instances *la01* à *la06*, et *car05* à *car08*. Chaque ligne dans le tableau contient les résultats moyens pour une instance obtenue sur l'ensemble des tests réalisés. La colonne "canonical schedule" contient le meilleur des ordonnancements canoniques parmi les mille ordonnancement tirés au hasard. La colonne "ordo. aléatoire" contient les meilleurs ordonnancements parmi les mille ordres valides tirés au hasard. Lorsqu'aucun ordre n'est réalisable, la colonne contient "nf". La dernière colonne, "Règles" fournit le résultat de la meilleure.

10.3.4 Application de l'heuristique au jobshop sans attente

Dans le tableau 3-32, l'heuristique proposée est comparée aux heuristiques de Mascis et Pacciarelli (2002). La meilleure des 14 règles fournies par notre heuristique est donnée dans les colonnes 3 et 4. Dans les colonnes 5 et 6, les résultats du meilleur ordonnancement canonique parmi les mille tirés aléatoirement sont fournis. Les colonnes suivantes donnent les résultats des heuristiques de Mascis et Pacciarelli. Pour chaque instance et pour chaque heuristique, la valeur de la solution et l'écart à la solution optimale sont donnés. Les heuristiques ne fournissent pas systématiquement de solutions, dans ce cas, la case du tableau contient un "x". Il est à remarquer que plus l'heuristique fournit des résultats de bonne qualité et moins elle donne de solutions réalisables. Par exemple, l'heuristique AMMC fournit seulement deux solutions réalisables, mais ces solutions sont à 12,6 et 5,7%.

En moyenne, les solutions proposées par notre heuristique sont à 69,8% de l'optimum alors que la meilleure des solutions proposées par Mascis et Pacciarelli est à 25,8%. Cet écart s'explique par le fait que notre heuristique n'est pas dédiée au problème sans attente. De plus, les heuristiques proposées ne fournissent pas systématiquement des solutions réalisables.

	opt/	Règles		Ordo. canonique		SMCP et SMBP		SMSP		AMCC		Meill. Mascis	
Instance	**rec.**	**Meil.**	**Écart**	**Meil.**	**Écart**	**ms**	**Écart**	**ms**	**Écart**	**ms**	**Écart**	**ms**	**Écart**
la06_0_0	1339	2164	61,6%	2509	87,4%	1758	31,3%	x	x	x	x	1758	31,3%
la07_0_0	1240	2119	70,9%	2247	81,2%	1837	48,1%	1609	29,8%	x	x	1609	29,8%
la08_0_0	1296	2332	79,9%	2286	76,4%	1839	41,9%	1580	21,9%	x	x	1580	21,9%
la09_0_0	1447	2435	68,3%	2789	92,7%	2430	67,9%	x	x	x	x	2430	67,9%
la10_0_0	1338	2155	61,1%	2479	85,3%	2427	81,4%	1541	15,2%	1506	12,6%	1506	12,6%
la11_0_0	1825	3024	65,7%	3456	89,4%	2226	22,0%	x	x	x	x	2226	22,0%
la12_0_0	1631	2720	66,8%	2938	80,1%	2202	35,0%	1935	18,6%	x	x	1935	18,6%
la13_0_0	1766	3179	80,0%	3414	93,3%	2197	24,4%	2221	25,8%	1867	5,7%	1867	5,7%
la14_0_0	1805	3245	79,8%	3577	98,2%	2381	31,9%	x	x	x	x	2381	31,9%
la15_0_0	1829	2993	63,6%	3418	86,9%	2720	48,7%	2134	16,7%	x	x	2134	16,7%
Moyenne :			69,8%		87,1%		43,3%		21,3%		9,1%		25,8%

Tableau 3-32. Résultats de l'heuristique sur les instances de jobshop sans attente

10.4 Conclusion sur l'heuristique

L'heuristique proposée ci-dessus est basée sur les règles de priorité. Ce type d'algorithme est très répandu dans les problèmes d'ordonnancement mais propose des solutions irréalisables dans le cas des problèmes avec time lags. Pour proposer des solutions forcément réalisables, nous nous sommes basés sur le module d'évaluation dénommé "Evaluation par règles de priorité". Ainsi, nous proposons des solutions de relativement bonne qualité en un temps de calcul très court. Cette heuristique est la première et la seule heuristique proposant des solutions forcément réalisables pour le problème de jobshop avec time lags. C'est pourquoi, nous l'utilisons pour construire des solutions initiales d'algorithmes itératifs, et c'est aussi pourquoi elle montre donc que l'on peut construire en un temps polynomial n'importe quelle solution réalisable de l'espace de recherche.

11 Conclusion

Dans ce chapitre, nous avons proposé une modélisation mathématique et une modélisation sous forme de graphe conjonctif-disjonctif du problème de jobshop avec time lags minimum et maximum. À partir de ces deux modélisations, nous avons proposé cinq méthodes d'optimisation réparties en trois catégories complémentaires. La formalisation linéaire proposée permet de trouver optimalement, par résolution directe, la solution d'instances de petites tailles. Pour trouver des solutions de bonne qualité pour les instances de tailles supérieures, nous avons recours à des heuristiques. Ces heuristiques sont réparties en deux catégories : les heuristiques de construction pour pouvoir obtenir rapidement des solutions de bonne qualité et les méthodes itératives qui permettent d'atteindre des solutions proches de l'optimum.

L'heuristique de construction proposée est la première heuristique à proposer systématiquement des solutions réalisables pour le problème de jobshop avec time lags. Elle démontre donc que la recherche d'une solution réalisable est un problème polynomial. De plus, elle permet d'envisager n'importe quelle solution réalisable. Ainsi, elle pourrait être efficacement couplée avec un algorithme de recherche opérationnel pour trouver toutes les solutions réalisables. On peut utiliser par exemple une règle qui choisit, entre deux opérations, celle qui est le plus tôt dans une permutation fournie des

opérations. Ainsi, on peut construire une métaheuristique performante, qui peut forcément trouver une solution réalisable pour le problème de jobshop avec time lags.

Les algorithmes itératifs proposés sont tous basés sur la notion de graphe conjonctif - disjonctif. Le premier, un algorithme tabou, étend efficacement et directement tous les principes de l'algorithme tabou de Nowicki et Smutnicki (1996) car c'est l'une des meilleures méthodes publiées pour le problème de jobshop. Le deuxième algorithme proposé est un algorithme mémétique envisageant conjointement des solutions réalisables et irréalisables. Cet algorithme fournit des résultats de qualité aussi bien pour les time lags serrés que pour les time lags lâches. Un deuxième algorithme dit "bi-objectif" utilise les mêmes fonctionnalités que l'algorithme mémétique mais permet en outre de conserver plus de diversité pendant la recherche.

Nos propositions pour le problème de jobshop avec time lags permettent donc d'envisager la résolution exacte des petites instances, la résolution approchée et rapide par une heuristique de construction et une résolution par un algorithme itératif adapté à chaque type d'instances.

Le tableau 3-33 est une synthèse des différents algorithmes proposés.

Algorithme d'optimisation	**Module d'évaluation**	**Commentaire**
Tabou	1 : Représentation semi active	Très efficace sur les instances de time lag lâche
Mémétique	2 : Représentation stratifiée	Bonnes solutions sur les instances de time lag serré
Bi-objectif (génétique)	2 : Représentation stratifiée	Solutions de bonne qualité pour tous les time lags
Heuristique	3 : Règles de priorité	Solutions réalisables en un temps très court

Tableau 3-33. Synthèse des algorithmes proposés

Chapitre 4 Le problème de jobshop avec transport et contraintes additionnelles

Ce chapitre est consacré à la présentation et à la résolution du problème de jobshop avec transport et contraintes additionnelles.

Sommaire

Table des figures

Table des tableaux

Table des algorithmes

1 Introduction

Dans ce chapitre, nous nous intéressons au problème de jobshop avec transport et contraintes additionnelles. En appliquant notre démarche de modélisation au problème d'ordonnancement des systèmes flexibles de production, ce problème apparaît naturellement comme un raffinement du modèle de jobshop classique lors de l'étude des systèmes flexibles de production. Or, le problème de jobshop est central en ordonnancement car c'est un problème d'atelier général et qu'il est largement étudié dans la littérature. Ainsi, de nombreuses méthodes efficaces ont été proposées pour sa résolution (cf. chapitre 2). Nous proposons donc d'étendre les méthodes existantes pour le problème de jobshop afin de prendre en compte le transport et les contraintes additionnelles.

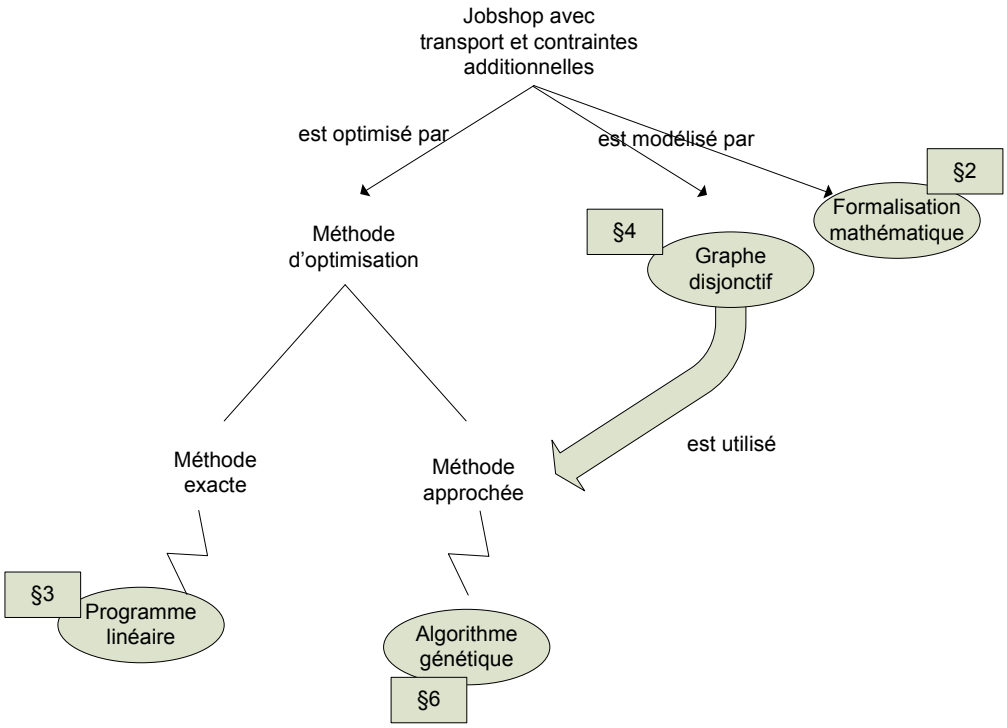

Figure 4-1. Synoptique de nos propositions pour le jobshop avec transport et contraintes additionnelles

La figure 4-1 est un synoptique de nos propositions pour le problème de jobshop avec transport et contraintes additionnelles. Ces propositions concernent principalement :

- la première modélisation mathématique prenant simultanément en compte toutes les contraintes considérées (cf. §0),
- le premier programme linéaire en nombres entiers issu de cette formalisation (cf. §3) permettant de trouver la solution optimale des instances à moins de 5 jobs,
- un modèle de graphe conjonctif - disjonctif pour la conception de méthodes efficaces d'optimisation (cf. §4),
- et un algorithme génétique pour la résolution approchée d'instances de grande taille (cf. §6).

L'étude du problème de jobshop avec transport et contraintes additionnelles est intéressante pour l'ordonnancement des système flexible de production, mais elle permet en outre de modéliser et de prendre en compte simultanément de nombreuses contraintes qui viennent enrichir le problème du jobshop. Ainsi, les contraintes prises en compte peuvent être réutilisées pour venir enrichir le problème du jobshop lors de l'étude d'autres problèmes d'ordonnancement.

Les systèmes flexibles de production sont définis par MacCarthy et Liu (1993) :

Un système flexible de production est un système de production capable de produire différents types de pièces, composé de machines à commande numérique ou à contrôle numérique et d'un système automatisé de stockage connectés par un système automatisé de manutention. Le fonctionnement du système entier est sous le contrôle et le pilotage d'un système informatique.

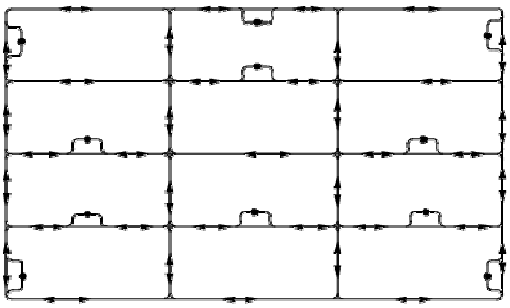

Figure 4-2. Réseau de transport

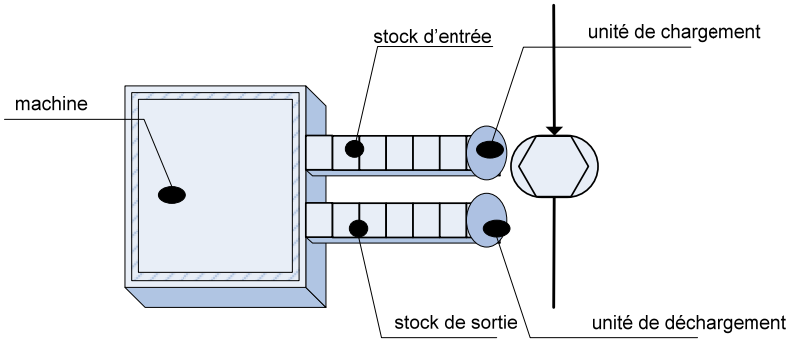

Figure 4-3. Une station

Plus précisément, un système flexible de production se compose d'un ensemble de stations reliées entre elles par un véhicule. Chaque station se compose de trois parties principales (cf. figure 4-3) : un stock d'entrée, une unité de traitement et un stock de sortie. Chaque pièce qui entre dans une station passe tout d'abord par le stock d'entrée. Lorsque l'unité de traitement est disponible, une des pièces du stock d'entrée est choisie puis celle-ci transite vers l'unité de traitement. L'opération sur l'unité de traitement commence alors. À la fin de cette opération et sous la condition qu'une place soit disponible dans le stock de sortie, la pièce quitte l'unité de traitement pour aller dans le stock de sortie. Dès son entrée dans ce stock, la pièce est détectée et un appel au véhicule est réalisé. Le véhicule se déplace d'une machine à l'autre à l'aide d'un réseau appelé réseau de guidage (cf. figure 4-2). Lorsque le véhicule est disponible, le système informatique de pilotage choisit quelle pièce va être traitée. Un déplacement commence alors entre la position courante du véhicule et le stock de sortie qui contient la pièce à déplacer. Ce déplacement est appelé déplacement à vide. À l'aide d'un système automatisé, le véhicule est capable de prendre la pièce sur le stock de sortie. Le transport de la pièce commence alors vers le stock d'entrée de la prochaine machine à visiter. Ce déplacement est appelé déplacement en charge. La pièce est alors automatiquement transférée du transporteur vers le stock d'entrée de la machine. Tous les déplacements sont réalisés suivant un trajet de durée prédéterminée. Pour des contraintes techniques, l'ordre d'entrée et l'ordre de sortie des pièces dans les stocks sont souvent PAPS (Premier Arrivé Premier Servi).

Les systèmes flexibles de production (SFP) sont soumis aux interblocages. Une situation d'interblocage a lieu lorsque le système est dans une configuration telle qu'une intervention manuelle est nécessaire. La figure 4-4 présente un exemple de situation d'interblocage : les stocks d'entrée et de sortie des stations sont pleins car leur capacité est atteinte, de plus les unités de traitement sont occupées par une pièce. Chaque machine est donc bloquée car une intervention extérieure est nécessaire pour pouvoir libérer une place en stock. Or la pièce sur l'unité de déchargement de la station de gauche doit visiter la station de droite, et inversement. Le transporteur ne peut donc commencer aucun des deux transports. Sans action particulière, le blocage de la situation perdure.

Pour éviter les interblocages, beaucoup de systèmes flexibles de production limitent le nombre de jobs simultanément autorisés dans le système. Soit N la somme des capacités des stocks d'entrée et de sortie des deux machines ayant la plus faible capacité. Si on limite le nombre de jobs simultanément autorisé à N, il est certain que la situation d'interblocage n'aura pas lieu. Le modèle présenté dans la suite prend en compte cette contrainte. Par contre, le modèle peut être utilisé pour les systèmes ne limitant pas le nombre de jobs, il suffit de supprimer la contrainte ou de prendre un nombre de jobs simultanément autorisés suffisamment grand. De nombreuses méthodes sont proposées dans la

littérature pour prendre en compte les interblocages : (Haro, 2002), (Soukhal, 2001), (Mangione, 2003), (Brauner, 1999).

Erreur ! Des objets ne peuvent pas être créés à partir des codes de champs de mise en forme.

Figure 4-4. Situation d'interblocage d'un système flexible de production

2 Définitions et notations

Pour définir notre problème de jobshop avec transport, et pour décrire les contraintes additionnelles que nous avons prises en compte, nous proposons la formalisation mathématique ci-après. Cette formalisation n'est pas une formalisation linéaire, elle ne permet pas la résolution directe par un solveur connu. La formalisation sous la forme d'un programme linéaire est décrite dans le paragraphe suivant (§ 3).

Dans la suite, nous utilisons les notations suivantes. On considère un problème $\wp$ de jobshop avec transport et contraintes additionnelles. On dispose de m machines pour réaliser n jobs, le système est muni d'un transporteur. On désigne par "opération machine" le traitement qu'un job doit subir sur une machine et on note I l'ensemble de ces opérations. Chaque job consiste en une suite d'opérations machine à réaliser dans un ordre donné, appelé la "gamme du job" : $i, SJ(i), SJ^2(i), \ldots$ (où $SJ(i)$ est l'opération suivant l'opération machine i dans la gamme). Chaque opération machine i doit être réalisée sans préemption sur la machine μ_i durant le temps p_i. Une machine ne peut réaliser qu'une opération à la fois. Chaque machine j est munie d'un stock d'entrée de capacité e_j et d'un stock de sortie de capacité s_j. Lorsqu'un job arrive sur une machine, il passe par le stock d'entrée si la capacité le permet et y réside jusqu'au début de son traitement. Le transfert entre le transporteur et le stock d'entrée a pour durée ε_c. Si la capacité ne permet pas le transfert vers le stock d'entrée, le transport du job est alors retardé. Lorsqu'un job a terminé son traitement, il passe dès que possible dans le stock de sortie jusqu'à son départ de la station. Un job quittant une station est forcément transporté vers le stock d'entrée d'une autre machine. Le transfert entre le stock de sortie et le transporteur a pour durée ε_d. S'il ne reste pas de places disponibles dans le stock de sortie, l'unité de traitement reste bloquée et aucun nouveau job ne peut commencer son traitement. La politique de gestion des stocks est premier arrivé premier servi (PAPS), ce qui signifie que l'ordre d'arrivée des jobs dans le stock d'entrée est égal à leur ordre de traitement sur la machine et est aussi égal à l'ordre de sortie des jobs du stock de sortie.

De plus, on prend en compte le transport d'un job entre deux machines. Soit une opération machine $i \in I$ et soit l'opération $SJ(i) \in I$ suivante dans la gamme du job. Entre ces deux opérations a lieu le transport du job entre le stock de sortie d'une machine et le stock d'entrée de la machine suivante. On appelle "opération transport" et on note $ST(i)$ ce transport. L'ensemble de toutes les opérations transport est noté T. La durée de l'opération transport $j = ST(i)$ est notée p_j et dépend des machines de départ et d'arrivée (μ_i à $\mu_{SJ(i)}$). De plus, l'opération transport précédent i est notée $PT(i)$. Pour une opération transport $i \in T$, on note $ST(i) \in T$ l'opération transport suivante (à déterminer). Entre ces deux opérations, le transporteur se déplace sans job de la machine d'arrivée ($\mu_{SM(i)}$) à la machine de départ de l'opération transport suivante ($\mu_{PM(ST(i))}$) (où $PM(i)$ est l'opération machine précédent l'opération transport i et $SM(i)$ est l'opération machine suivant l'opération transport i). On appelle déplacement à vide ce type de déplacement. La durée d'un déplacement à vide est fonction des machines de départ et d'arrivée : $tv(\mu_{SM(i)}, \mu_{PM(ST(i))})$. Par abus de notation, nous noterons $tv(i, j)$ le temps de déplacement à vide entre les opérations transport i et j. Enfin, on note $p(i)$ le nombre de jobs partis de la station avant l'opération transport i. De même, on note $a(i)$ le nombre de jobs arrivés sur la station avant l'opération transport i.

Finalement, on considère que l'atelier est muni d'une station d'entrée et d'une station de sortie. Ces stations permettent de prendre en compte un éventuel trajet avant la première opération et après la dernière opération.

Parmi les notations décrites ci-dessus, toutes sont des données du problème exceptées les notations suivantes qui dépendent de la solution envisagée : l'ordre des opérations transport $ST(i)$ et

$PT(i)$ pour une opération transport i, les ordres d'opérations machines $SM(i)$, $PM(i)$ pour une opération machine i, les quantités $p(i)$ et $a(i)$ et les dates de début des opérations t_i. L'objectif est de déterminer un ordonnancement réalisable minimisant la date de fin de la dernière opération.

La formalisation suivante est le modèle de référence dans la suite de cette thèse, elle contient toutes et seulement les contraintes prises en compte.

(C1) Contrainte entre les opérations machine et les opérations transport.

Cette contrainte impose qu'une opération transport ne commence que quand l'opération machine précédente a terminé.

Soit une opération transport $ST(i)$ qui suit une opération machine i. La contrainte impose que l'opération $ST(i)$ ne débute qu'après la fin du traitement de l'opération i augmentée de la durée du déplacement à vide. Ce déplacement à vide a lieu entre l'opération transport $PT(ST(i))$ traitée immédiatement avant l'opération $ST(i)$ par le transporteur et l'opération $ST(i)$. On a donc : $t_i + p_i + tv(PT(ST(i)), ST(i)) \leq t_{ST(i)}$.

Remarque : si l'anticipation est autorisée, le transport à vide peut commencer avant et la contrainte devient $t_i + p_i \leq t_{ST(i)}$.

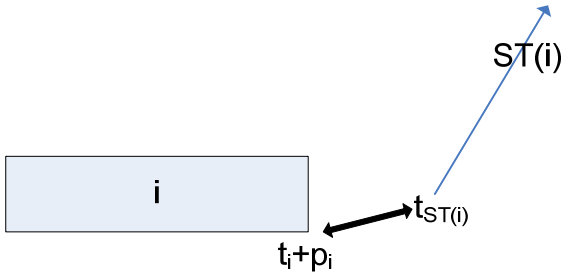

Figure 4-5. Contrainte entre les opérations machine et les opérations transport (sans anticipation)

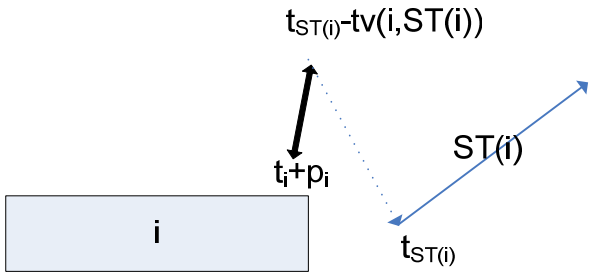

Figure 4-6. Contrainte entre les opérations machine et les opérations transport (avec anticipation)

(C2) Contrainte entre les opérations transport et les opérations machines.

Cette contrainte impose qu'une opération machine ne commence que quand l'opération transport précédente a terminé.

Soit une opération transport $i \in T$, cette opération termine son traitement avant le début de l'opération machine suivante (opération $SM(i)$). La contrainte s'écrit donc : $t_i + p_i \leq t_{SM(i)}$.

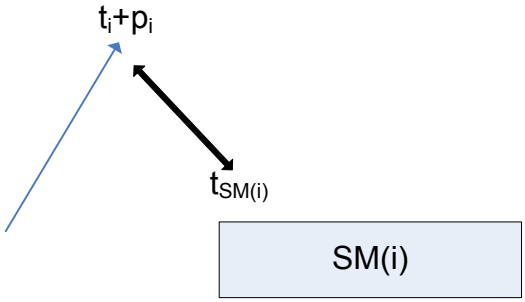

Figure 4-7. Contrainte entre les opérations transport et les opérations machines

(C3) Contrainte de disjonction sur les machines.

Cette contrainte impose qu'une seule opération ne soit traitée à un instant donné par une machine.

Étant données deux opérations $i \in I$ et $SM(i)$ traitées consécutivement sur une même machine, la contrainte s'écrit donc : $t_i + p_i \leq t_{SM(i)}$.

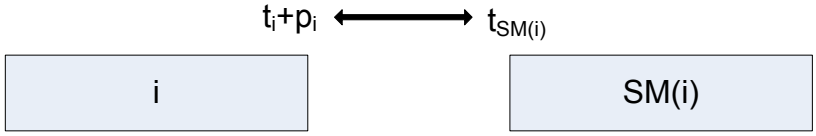

Figure 4-8. Contrainte de disjonction sur les machines

(C4) Contrainte de disjonction sur le transporteur.

Cette contrainte impose qu'une seule opération ne soit traitée à un instant donné par le transporteur.

Étant données deux opérations $i \in T$ et $ST(i)$ traitées consécutivement sur le transporteur, la contrainte s'écrit donc $t_i + p_i + tv(i, ST(i)) \leq t_{ST(i)}$ où $t_i + p_i$ est la date de fin de l'opération i et $tv(i, ST(i))$ est la durée du transport à vide entre les deux opérations transport.

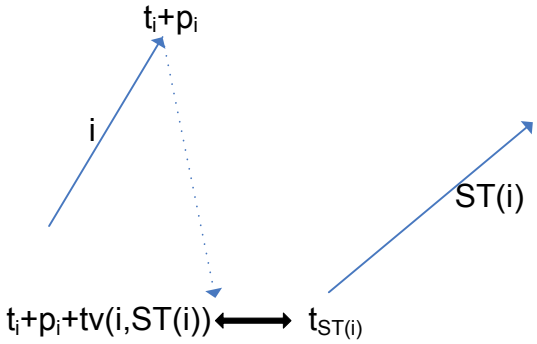

Figure 4-9. Contrainte de disjonction sur le transporteur

(C5) Contrainte PAPS sur les stocks d'entrée.

La contrainte PAPS sur les stocks d'entrée impose la politique de gestion des stocks d'entrée Premier Arrivé Premier Servi (PAPS).

Ainsi le premier job entré dans le stock est le premier à être traité. Sur chaque machine, l'ordre d'arrivée des jobs dans le stock d'entrée est égal à l'ordre de traitement de ces jobs. Or, l'ordre d'arrivée des jobs est égal à l'ordre des opérations transport amenant ces jobs sur la machine. En d'autres termes,

la contrainte PAPS sur les stocks d'entrée impose que "pour un ordre des opérations transport amenant les jobs sur une machine $\pi = (1,2,...,n)$, l'ordre des opérations machine est alors $\pi' = (SM(1), SM(2),..., SM(n))$ ".

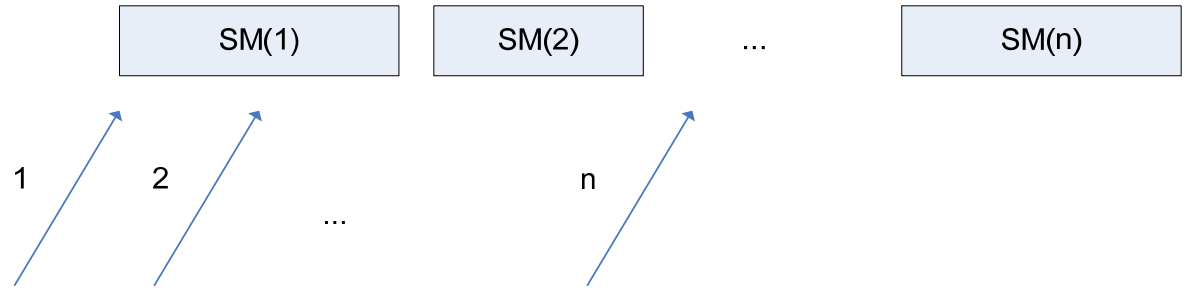

Figure 4-10. Contrainte PAPS sur les stocks d'entrée

(C6) Contrainte PAPS sur les stocks de sortie.

La contrainte PAPS sur les stocks de sortie impose la politique de gestion des stocks de sortie Premier Arrivé Premier Sauvé PAPS.

Ainsi le premier job entré dans le stock est le premier à être traité. L'ordre de traitement des jobs sur la machine est égal à l'ordre de sortie des jobs du stock de sortie. Or, l'ordre de sortie des jobs est égal à l'ordre des opérations transport amenant ces jobs hors de la machine. Soit $\pi = (1,2,...,n)$ l'ordre des opérations sur une machine donnée, $\pi' = (ST(1), ST(2),..., ST(n))$ est l'ordre des opérations transport suivant le traitement sur la machine.

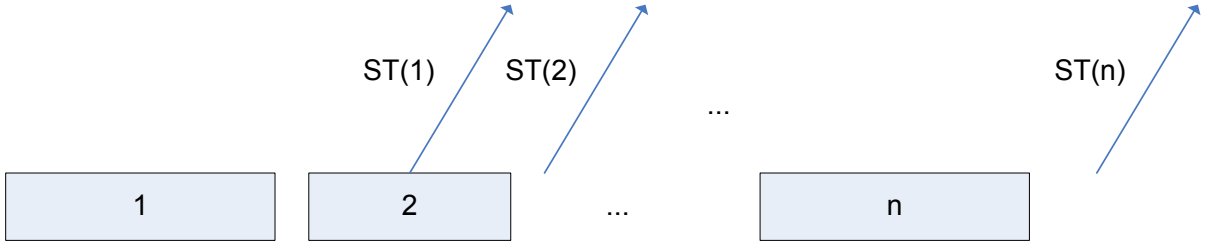

Figure 4-11. Contrainte PAPS sur les stocks de sortie

(C7) Contrainte limitant le nombre de jobs dans le stock d'entrée.

Cette contrainte limite la quantité de jobs dans le stock d'entrée de manière à respecter sa capacité.

Soit une opération machine $i \in I$ qui commence son traitement à la date t_i et libère une place dans le stock d'entrée. À cet instant, deux cas sont possibles. Soit la capacité du stock est atteinte ou non.

Si la capacité du stock d'entrée est atteinte au moment où l'opération i se termine, alors la place libérée permet à un nouveau job d'entrer dans le stock. Pour déterminer le job qui attend éventuellement cette place, on exploite la contrainte PAPS (C5). En effet, grâce à cette contrainte, on connaît l'ordre des opérations qui arrivent après i dans le stock d'entrée : $SM(i)$, $SM^2(i)$, ... Si le stock est plein au moment où l'opération i se termine, c'est qu'il contient les opérations $SM(i)$, $SM^2(i)$, ... $SM^{k-1}(i)$ où $k = e_{\mu_{SM(i)}}$ est le nombre de places dans le stock d'entrée.

Le job qui attend la place dans le stock d'entrée est donc le prochain job qui sera apporté sur cette machine, il correspond donc à l'opération $SM^k(i)$. L'opération transport qui apporte cette opération sur la machine est $STM^k(PT(i))$.

Ainsi, la contrainte C7 exprime que le transport $STM^k(PT(i))$ ne commence que lorsque l'opération i a libéré une place dans le stock d'entrée. En d'autres termes, la date de début du transport $STM^k(PT(i))$ doit être supérieure à la date de début de l'opération i.

Si la capacité du stock d'entrée est atteinte au moment où l'opération i se termine, alors toutes les opérations $SM(i)$, $SM^2(i)$, ..., $SM^{k-1}(i)$ ne sont pas simultanément dans le stock au moment où l'opération transport commence, c'est qu'au moins l'opération $SM(i)$ a débuté son traitement. Dans ce cas, l'inégalité proposée est donc trivialement vérifiée.

La contrainte C7 s'exprime donc $t_i \leq t_{STM^k(PT(i))}, \forall i \in I$. La figure 4-12 présente un ordonnancement respectant la contrainte de capacité des stocks. La figure 4-13 présente un autre ordonnancement ne respectant pas la contrainte. Dans ce deuxième ordonnancement, on observe que le nombre de places utilisées dans le stock d'entrée est supérieur à la capacité.

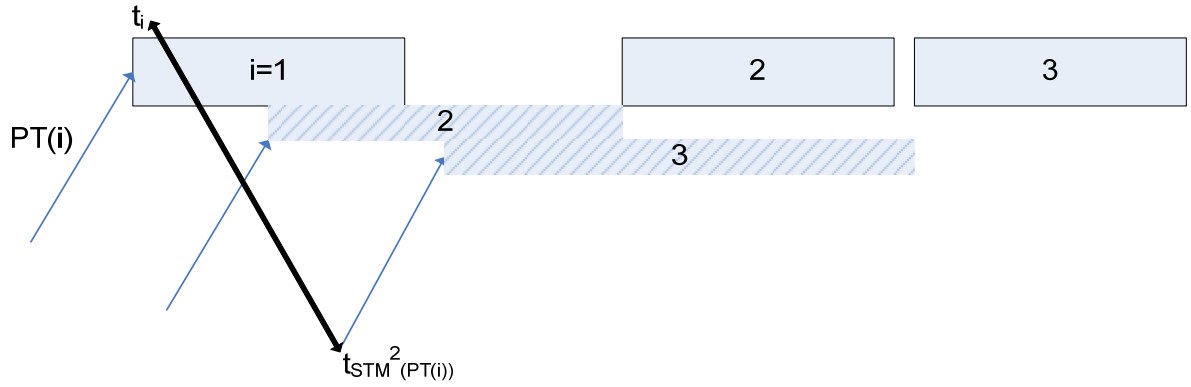

Figure 4-12. Contrainte limitant le nombre de jobs dans le stock d'entrée

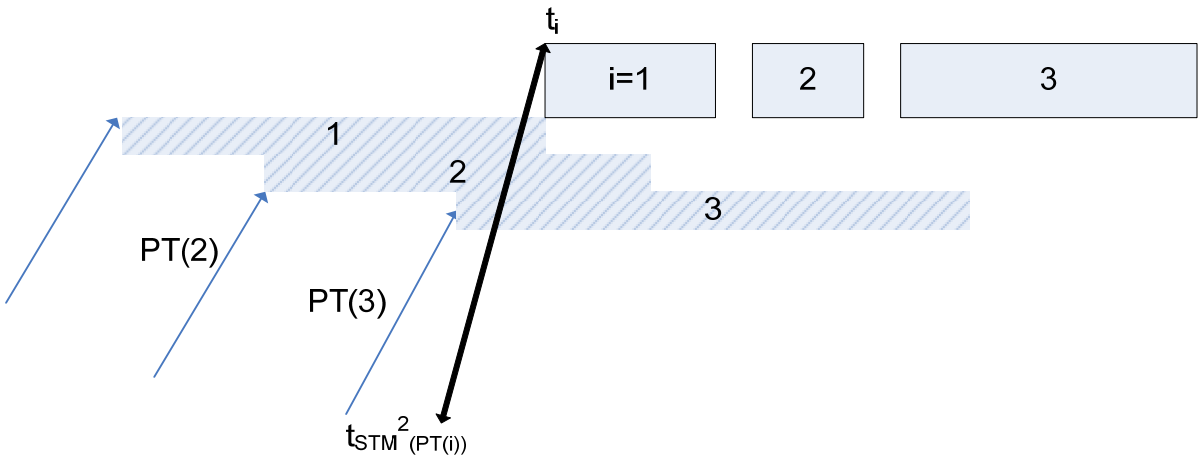

Figure 4-13. Contrainte limitant le nombre de jobs dans le stock d'entrée (ordonnancement non valide)

(C8) Contrainte de blocage de l'unité de traitement.

Cette contrainte impose qu'un job retarde le début de son traitement tant que le job précédent sur la machine n'a pas obtenu de place dans le stock de sortie.

La capacité limitée du stock de sortie est modélisée par les contraintes C8 et C9. Plus précisément la contrainte C8 empêche une opération de commencer son traitement tant que le stock a atteint sa capacité maximale. La contrainte C9 modélise quant à elle les restrictions sur l'ordre des opérations transport. Nous exprimons cette contrainte en fonction des opérations transport libérant une place dans le stock de sortie.

Soit $i \in T$ une opération transport enlevant un job du stock de sortie de la machine $m0$. Lorsque ce job quitte la station, il libère une place dans le stock de sortie. L'opération transport i commence à t_i ; ε_d unités de temps après, elle finit de libérer une place dans le stock. La place est donc libérée à la date $t_i + \varepsilon_d$.

Soit $j \in T$ l'opération transport telle que $a(j) = p(i) + s_{m_0} + 1$. Cette opération transport amène le job dont le début de l'exécution sera retardée sur la machine à cause de la contrainte de blocage. Parce que la contrainte C5 est vérifiée (contrainte PAPS sur le stock d'entrée), toutes les opérations transportées avant l'opération j sont traitées sur la machine avant elle (c'est-à-dire $p(i) + s_{m_0} + 1$ opérations). Il y a donc autant d'opérations qui sont entrées dans le stock de sortie avant le début du traitement de cette opération.

Il est possible que l'opération j n'existe pas, c'est-à-dire qu'il n'existe pas d'opération j telle que $a(j) = p(i) + s_{m_0} + 1$. Dans ce cas, il n'y a pas de contrainte de blocage de l'unité de traitement. L'inéquation proposée ci-après doit donc être supprimée dans ce cas. De même, si l'opération i apparaît avant l'opération j dans l'ordre de traitement des opérations transport, c'est qu'il y a eu $p(i)$ opérations sorties du stock avant que les $p(i) + s_{m_0} + 1$ opérations ne soient traitées par la machine. Au moment où l'opération i commence son traitement sur la machine, il y a donc $p(i)$ opérations sorties du stock avant l'opération i plus l'opération i. Au total, $p(i)+1$ opérations sont sorties du stock. Or seulement $p(i) + s_{m_0} + 1$ opérations sont entrées avant l'opération j, il y a donc au maximum $\left(p(i) + s_{m_0} + 1\right) - \left(p(i) + 1\right) = s_{m_0}$ opérations dans le stock de sortie. Celui-ci peut être saturé, mais il a la capacité suffisante pour contenir toutes ces opérations. Ainsi, l'opération j n'est donc pas bloquée par l'opération précédente.

Par contre, si l'opération i apparaît après l'opération j dans l'ordre de traitement des opérations transport, c'est que $p(i) + s_{m_0} + 1$ opérations ont été traitées par la machine avant que $p(i)$ opérations ne soient sorties du stock (cf. figure 4-14). Il y a donc au moins $\left(p(i) + s_{m_0} + 1\right) - p(i) = s_{m_0} + 1$ opérations dans le stock de sortie. Le stock n'ayant pas la capacité suffisante, la dernière opération traitée bloque l'unité de traitement. Ainsi, le traitement de l'opération j est retardé jusqu'au transport i. Lorsque l'opération j est traitée sur la machine, elle est alors une opération machine dénommée $SM(j)$.

On a donc : $t_i + \varepsilon_d \le t_{SM(j)}, \forall i \in T$ où j est l'opération telle que $a(j) = k$ (si j existe et est avant i dans l'ordre du transporteur).

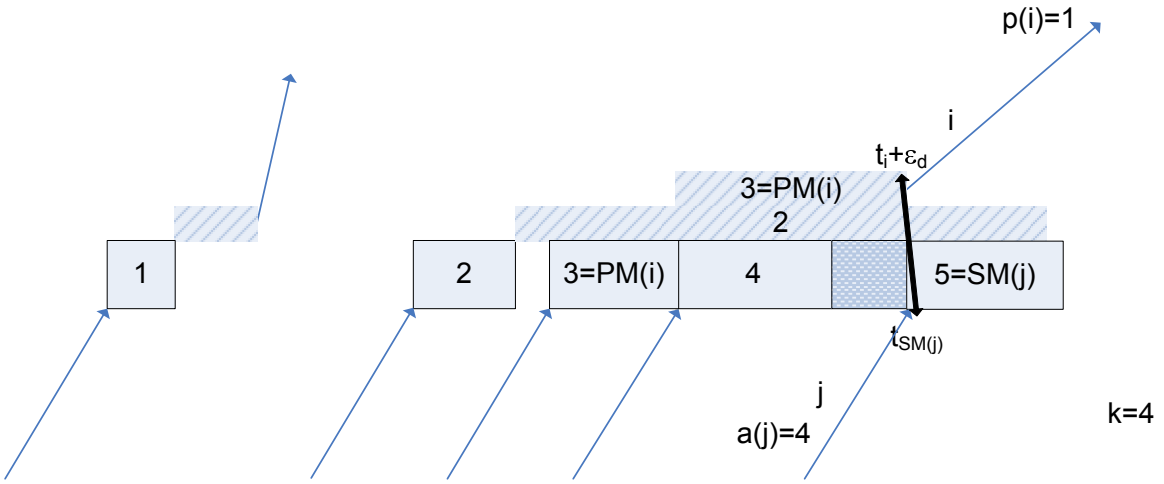

Figure 4-14. Contrainte de blocage de l'unité de traitement

Cette contrainte est illustrée par la figure 4-14. Dans cette figure, les opérations $1, 2, 3, 4$ et 5 sont traitées consécutivement sur la même machine. L'opération i est l'opération transport après l'opération 3. Pour cette opération, il y a $p(i) = 1$ opérations sortant de la machine. Le stock de sortie comporte deux places, on cherche donc l'opération transport j telle que $a(j) = 4$. Cette opération transport est

réalisée avant l'opération i, il y a donc une attente sur l'unité de traitement. L'exemple représenté sur la figure ne respecte délibérément pas la contrainte C6 (contrainte PAPS sur les stocks de sortie). La formulation proposée de la contrainte C8 a été conçue pour rester valide même si la contrainte C6 n'est pas respectée.

(C9) Contrainte limitant le nombre de jobs dans le stock de sortie.

La contrainte précédente modélise partiellement la limitation de capacité du stock de sortie en empêchant une opération de commencer son traitement si la machine et le stock de sortie sont pleins. Ce que cette contrainte ne modélise pas, c'est que l'opération bloquée sur la machine ne peut pas instantanément passer sur le transporteur. Elle doit d'abord obtenir une place sur le stock de sortie.

La contrainte s'écrit $a(i) - p(i+1) < s_{m_0}$.

Soit un stock de sortie dont la capacité maximum est atteinte sur une machine m_0 et une opération transport i telle que l'opération $SM(i)$ vient de terminer son traitement. L'opération transport suivant l'opération $SM(i)$ s'appelle $i+1$. Cette opération ne peut commencer aussitôt, et en effet la contrainte C9 est violée : la quantité $a(i)-p(i+1)$ est le nombre de jobs dans le stock de sortie quand l'opération $SM(i)$ a terminé. Puisque le stock est plein, on a $a(i)-p(i+1)=s_{m_0}$, ce qui viole la contrainte.

Si le stock de sortie n'est pas plein sur la machine m_0 au moment où l'opération $SM(i)$ vient de terminer son traitement. Alors, il y a eu moins de jobs d'amenés dans le stock ou plus de jobs partis du stock. Dans tous les cas, la quantité $a(i)-p(i+1)$ a diminué et est donc strictement inférieure à s_{m_0}. On a donc $a(i)-p(i+1)<s_{m_0}$.

Finalement, la contrainte C9 s'écrit $p(i+1)+s_{m_0} > a(i)$.

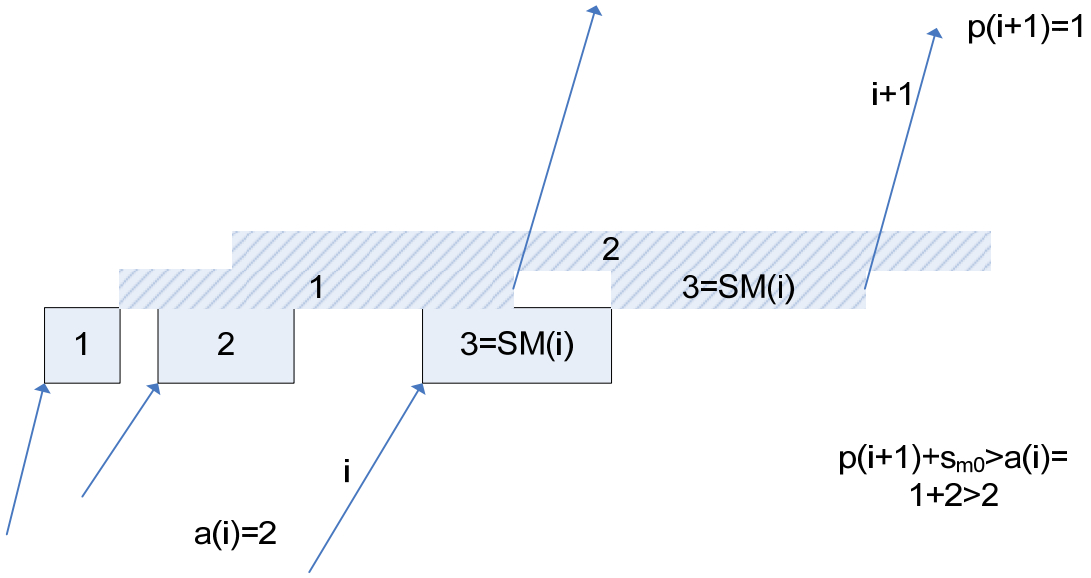

Figure 4-15. Contrainte limitant le nombre de jobs dans le stock de sortie (contrainte vérifiée)

La figure 4-15 montre un exemple d'ordonnancement vérifiant la contrainte C9. La figure 4-16 montre un exemple d'ordonnancement vérifiant la contrainte de blocage C8 mais ne vérifiant pas la contrainte C9.

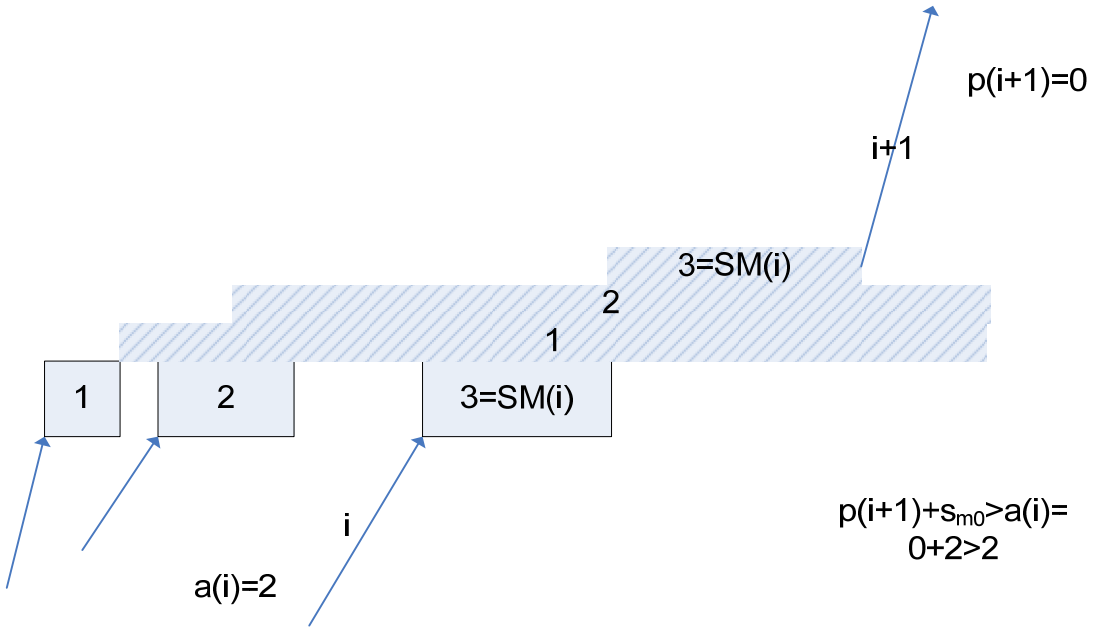

Figure 4-16. Contrainte limitant le nombre de jobs dans le stock de sortie (contrainte violée)

(C10) Contrainte du nombre de jobs simultanément autorisés dans le système.

Cette contrainte impose qu'un nombre limité de jobs soit simultanément présent dans le système.

Soit T un ordre des opérations transport. On note $e(i)$ le nombre de jobs entrés dans le système avant l'opération transport i. On note $s(i)$ le nombre de jobs sortis du système avant le début de l'opération transport i. Ces deux valeurs sont calculées à partir de l'ordre des opérations transport T. La différence entre ces deux quantités $(e(i)-s(i))$ est le nombre de jobs déjà présents dans le système. Avec le job sur le point d'entrer, il y a donc $e(i)-s(i)+1$ jobs dans le système. Ce nombre est limité par le nombre N de jobs simultanément autorisés.

La contrainte s'écrit alors, $e(i)-s(i)<N$ où N est le nombre maximum de jobs autorisés.

Récapitulatif

La formalisation ci-dessus est composée de 10 contraintes modélisant complètement le problème. Le tableau 4-1 récapitule comment les contraintes sont utilisées pour modéliser le problème.

Contrainte du système	Contrainte du modèle
Contraintes technologiques	C1+C2
Précédence opération machine / transport	C1
Précédence opération transport / machine	C2
Disjonction machine	C3
Disjonction transport	C4
Déplacement à vide	C4
Capacité du stock d'entrée	C7
Capacité du stock de sortie	C8+C9
Non anticipation	C1
Nombre de jobs	C10
PAPS sur stock d'entrée	C5
PAPS sur stock de sortie	C6

Tableau 4-1. Récapitulatif des contraintes

Comme certaines contraintes du système ont été modélisées par plus d'une contrainte, le tableau 4-1 propose un récapitulatif des liens entre ces deux types de contraintes.

Dans ce tableau figure la contrainte "technologique" entre opérations consécutives du même job. Cette contrainte est classique pour le problème de jobshop, elle impose que deux opérations consécutives i et $SJ(i)$ d'un même job soient réalisées dans cet ordre sur la machine : $t_i + p_i \leq t_{SJ(i)}$. Or d'après les contraintes (C1) et (C2) nous avons : $t_i + p_i \leq t_{ST(i)}$ et $t_{ST(i)} + p_{ST(i)} \leq t_{SM(ST(i))}$ d'où $t_{ST(i)} + p_i + p_{ST(i)} \leq t_{SM(ST(i))}$. Or l'opération $SM(ST(i))$ est par définition l'opération $SJ(i)$. On a donc la contrainte $t_{ST(i)} + p_i + p_{ST(i)} \leq t_{SJ(i)}$ qui est plus restrictive que la contrainte de conjonction. Dans la suite, nous ne considérerons plus la contrainte "technologique".

3 Formalisation linéaire

La formalisation mathématique ci-dessus ne peut être résolue directement par les solveurs. Dans cette section, nous proposons une linéarisation de cette formalisation mathématique. Cette formalisation linéaire est soumise dans (Caumond *et al.*, 2006a).

Le modèle linéaire proposé utilise trois paquets de variables continues (t_i, O_i, I_i) pour décrire les dates de début des opérations et sept types de variables binaires $(b_{ij}, T_{ij}, Y_{ij}, Z_{ij}, a_{ij}, c_{ij}, W_{ij})$ pour décrire les ordres de passage des opérations sur les différentes ressources.

Pour des raisons de simplicité, les indices i et j sont utilisés pour les opérations, les machines et les jobs. Les indices r, s et u sont utilisés pour désigner les opérations transport.

Les ensembles et notations suivants sont des données du problème :

J : Ensemble des jobs; $J = \{1; 2; ...; n\}$

M : Ensemble des stations $M = \{1; 2; ...; m\}$

$\tilde{M}$: Ensemble des stations autres que les stations d'entrée et de sortie $M = \{2; ...; m-1\}$

M_q : Ensemble des opérations machines réalisées sur la machine q $M_q = \{i \in I | \mu_i = q\}$

N : Nombre de jobs simultanément autorisés dans le système

I : Ensemble des opérations machines $I = \{1; 2; ...; |I|\}$ ordonnées par les contraintes technologiques ($i+1$ est l'opération machine suivant i si elle existe).

$\tilde{I}$: Ensemble des opérations non réalisées par les stations d'entrée/sortie; $\tilde{I} = \{ i \in I | \mu_i \neq 1, \mu_i \neq m \}$

T : Ensemble des opérations transports $T = \{1; 2; ...; |T|\}$, ordonnées suivant $(od_r)_{r \in I}$

D_M : Ensemble des paires d'opérations sujettes à disjonction $D_M = \{(i,j) \in I^2 | i < j \text{ and } \mu_i = \mu_j\}$

$\tilde{D}_M$: Ensemble des paires d'opérations de D_M réalisées sur $\tilde{M}$ $\tilde{D}_M = \{(i,j) \in D_M | \mu_i \neq 1, \mu_i \neq m\}$

ε_c : Temps de chargement sur la station

ε_d : Temps de déchargement de la station

p_i : Temps de traitement d'une opération machine, i ($i \in \tilde{I}$)

μ_i : Station où l'opération i ($i \in I$) est réalisée

l_{ij} : Temps de transport de la station i à la station j, $(i,j) \in M$

v_{ij} : Transport à vide de la machine i à la machine j, $(i,j) \in M$

od_r : Opération machine précédent l'opération transport $r \in T$

oa_r : Opération machine suivant l'opération transport $r \in T$

se_i : Capacité du stock d'entrée de la station i, $i \in \tilde{M}$

ss_i : Capacité du stock de sortie de la station i, $i \in \tilde{M}$

Ft_i : Première opération transport du job i, $i \in J$

Lt_i : Dernière opération transport du job i, $i \in J$

J_r : Numéro du job transporté par l'opération transport r, $r \in T$

Les notations ci-dessous sont des variables du problème, leurs valeurs sont à déterminer :

t_i : Date de fin de traitement de l'opération machine i, $i \in I$

O_i : Temps d'attente du job de l'opération i dans le stock de sortie de la machine μ_i, $i \in \tilde{I}$

I_i : Temps d'attente du job de l'opération i dans le stock d'entrée de la machine μ_i, $i \in \tilde{I}$

tm_i : Temps total passé sur la machine μ_i par l'opération i, $i \in \tilde{I}$

$$b_{ij} = \begin{cases} 0 \text{si l'opération } i \text{est ordonnancée avant} \\ \quad \text{l'opération } j \text{ sur la machine } \mu_i \qquad (i,j) \in I^2 \text{avec} \mu_i = \mu_j, i \neq j \\ 1 \text{sinon} \end{cases}$$

$$Y_{ij} = \begin{cases} 0 \text{si l'opération } j \text{attend l'opération } i \\ \quad \text{dans le stock d'entrée de la machine } \mu_i \quad (i,j) \in \tilde{I}^2 \text{avec} \mu_i = \mu_j, i \neq j \\ 1 \text{sinon} \end{cases}$$

$$Z_{ij} = \begin{cases} 0 \text{si l'opération } j \text{attend l'opération } i \\ \quad \text{dans le stock de sortie de la machine } \mu_i \quad (i,j) \in \tilde{I}^2 \text{avec} \mu_i = \mu_j, i \neq j \\ 1 \text{sinon} \end{cases}$$

$$a_{rs} = \begin{cases} 0 \text{si l'opération transport } r \text{est traitée avant} \\ \quad \text{l'opération transport } s \qquad (r,s) \in T^2 \text{avec } r \neq s \\ 1 \text{sinon} \end{cases}$$

$$c_{rs} = \begin{cases} 1 \text{si l'opération transport } r \text{est traité immédiatement} \\ \quad \text{avant l'opération transport } s \qquad (r,s) \in T^2 \text{où} r \neq s \\ 0 \text{sinon} \end{cases}$$

$$W_{ij} = \begin{cases} 0 \text{ si le job } i \text{ est dans le système alors que le} \\ \quad \text{job j entre dans le système} \qquad (i,j) \in N^2 \text{où} i \neq j \\ 1 \text{sinon} \end{cases}$$

Les variables O_i, I_i et tm_i modélisent le traitement du job sur la station comme le montre la figure 4-17. Quand le job entre dans la station, il séjourne dans le stock d'entrée pendant I_i unités de temps. Puis, il passe instantanément (sans attente) à la machine de la station. Le job passe tm_i unités de temps sur cette machine, ce qui correspond au minimum à la durée du traitement du job. Si le stock de sortie a une place, le job y passe instantanément (sans attente). Le job quitte la station à la date t_i (dès qu'il quitte le stock de sortie).

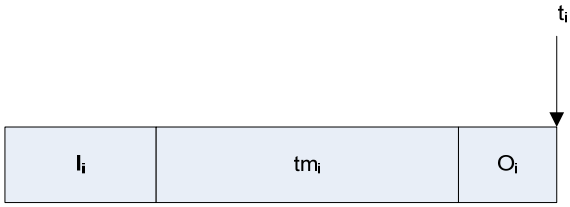

Figure 4-17. Détail du traitement d'un job sur une station

En se basant sur ces notations, nous proposons deux formulations linéaires. La première modélise les systèmes capables d'implanter la politique de gestion optimale des stocks. Ce type de système doit être muni d'un moyen de manutention capable d'identifier et de manipuler n'importe quelle pièce du stock. Ce type de moyen est onéreux et de nombreux systèmes implantent des stocks dont la politique est PAPS (Premier Arrivé Premier Servi). Ce type de stock est peu onéreux et plus facile à mettre en œuvre, il correspond à des moyens de transport de type tapis roulant. La deuxième formulation modélise donc les stocks gérés de cette manière, cette politique de gestion est implantée comme une contrainte supplémentaire sur chacun des stocks. On peut donc activer ou désactiver cette contrainte à volonté sur chaque machine pour modéliser les systèmes qui utilisent les deux types de politique de gestion.

Dans les deux programmes linéaires mixtes, l'objectif est de minimiser le makespan (date de fin de la dernière opération).

3.1 Système muni d'une politique avancée de gestion des stocks

Dans cette formalisation linéaire, nous proposons de modéliser le problème de jobshop avec transport et contraintes additionnelles. Contrairement au problème résolu dans le paragraphe suivant, il n'est pas suffisant de connaître l'ordre des opérations de transport pour déterminer l'ordre des opérations sur les machines (car l'ordre d'arrivée des opérations sur les machines, l'ordre de sortie des opérations sur les machines et l'ordre de traitement des opérations sont à priori trois ordres différents). La description du problème est réalisée en groupe de contraintes. Certains groupes sont plus dévolus aux opérations sur les machines, d'autres aux opérations sur le transporteur, les autres permettant de lier les deux. Chaque ensemble de contraintes forme un tout cohérent et est expliqué en détail.

Blocage de la machine

Le premier ensemble de contraintes impose que le temps passé par les jobs sur les machines est supérieur au temps de traitement du job. Ainsi, le temps total qu'une opération i passe sur une machine tm_i doit être supérieur au temps de traitement de l'opération p_i.

$$tm_i \geq p_i \quad (1.1)$$

où $i \in \tilde{I}$

En modélisant ainsi le séjour de l'opération sur la machine, on permet à l'opération de séjourner sur la machine plus longtemps que ne le nécessite le temps de traitement (p_i). Ceci est particulièrement utile quand le stock de sortie est plein et que l'opération ne peut libérer l'unité de traitement. Cette modélisation permet en outre de libérer la place dans le stock d'entrée au plus tôt (i.e. dès que l'unité de traitement est libre). Ainsi, on permet à la station i de contenir ses $se_i + 1 + ss_i$ pièces.

Contraintes de précédence

Les contraintes de précédence imposent que l'ordre des opérations d'un job est conforme aux contraintes technologiques.

$$t_i = t_{i-1} + \varepsilon_d + l_{\mu_{i-1},\mu_i} + \varepsilon_c + I_i + tm_i + O_i \quad (1.2)$$

où $i \in \tilde{I}$

$$t_i = t_{i-1} + \varepsilon_d + l_{\mu_{i-1},\mu_i} + \varepsilon_c \quad (1.3)$$

où $i \in I - \tilde{I}$.

L'opération qui précède l'opération *i* dans la gamme est notée *i-1*. Entre ces deux opérations machines, le job subit différentes opérations : il passe par le stock d'entrée, la machine et le stock de sortie. Toutes ces opérations sont réalisées consécutivement et sans temps d'attente. La contrainte de précédence fait donc le lien entre les dates de début des deux opérations.

La contrainte de précédence vérifie que la date de fin de l'opération *i* est égale à la date de fin de l'opération précédente *i-1* augmentée du temps de déchargement ε_d, du temps de transport l_{μ_{i-1},μ_i}, du temps de chargement de l'opération sur la machine ε_c, du temps d'attente du job dans le stock d'entrée I_i, du temps total passé sur la machine tm_i, et du temps de traitement O_i.

La contrainte (1.2) est exprimée pour toute paire d'opérations consécutives sauf pour la dernière paire d'opérations. Pour celle-ci, les variables I_i, tm_i et O_i ne sont pas définies, la contrainte se réécrit comme présenté dans (1.3). La contrainte 1.3 est un cas particulier de 1.2 dans laquelle $I_i = tm_i = O_i = 0$.

Contrainte de disjonction pour les opérations machines

Cette contrainte assure que deux opérations *i* et *j* ne soient pas traitées simultanément sur la même machine : l'opération i ne peut commencer avant que l'opération précédente sur la même machine ne soit terminée et ait obtenu une place dans le stock de sortie.

$$t_i - O_i \leq t_j - O_j - tm_j + Hb_{ij} \quad (1.4)$$

où $(i,j) \in \tilde{I}^2$ et $i \neq j, \mu_i = \mu_j$

$$b_{ij} + b_{ji} = 1 \qquad (1.5)$$

où $(i,j) \in \tilde{I}^2$ et $i < j, \mu_i = \mu_j$

Si $b_{ij} = 1$, l'opération i est ordonnancée après l'opération j sur la machine μ_i. Dans ce cas, la contrainte (1.4) est toujours vérifiée car le paramètre H est suffisamment grand.

Si $b_{ij} = 0$, l'opération i est ordonnancée avant l'opération j sur la machine μ_i. Dans ce cas, la contrainte (1.4) se réécrit $t_i - O_i \leq t_j - O_j - tm_j$. Cette contrainte exprime que l'opération j commence son traitement (date $t_j - O_j - tm_j$) après que l'opération i soit entrée dans le stock de sortie (date $t_i - O_i$).

Contrainte de disjonction pour les opérations transport

Cette contrainte exprime que deux opérations transport ne peuvent avoir lieu simultanément sur le transporteur.

$$t_i + \varepsilon_c + l_{\mu_i,\mu_{i+1}} + \varepsilon_d + v_{\mu_{i+1},\mu_j} \leq t_j + a_{rs}.H \qquad (1.6)$$

où $(r,s) \in T^2, r \neq s$, $i = od_r$ et $j = od_s$

$$a_{rs} + a_{sr} = 1 \qquad (1.7)$$

où $(r,s) \in T^2, r < s$, $i = od_r$ et $j = od_s$

$$a_{rs} = 0 \qquad (1.8)$$

où $(r,s) \in T^2, r < s, J_r = J_s$

Si $a_{rs} = 0$, l'opération transport r est traitée avant l'opération transport s. La contrainte (1.6) peut alors être réécrite $t_i + \varepsilon_c + l_{\mu_i,\mu_{i+1}} + \varepsilon_d + v_{\mu_{i+1},\mu_j} \leq t_j$.

Puisque l'opération i est définie par $i = od_r$, elle précède donc l'opération transport r. Ainsi, la date de fin t_i de l'opération i est aussi la date de début de l'opération transport r. Cette date marque le début de l'opération transport i. Augmentée des opérations de chargement de durée ε_d, transport de la station μ_i à la station μ_{i+1} de durée $l_{\mu_i,\mu_{i+1}}$, déchargement de durée ε_c et transport à vide de durée v_{μ_{i+1},μ_j}, on obtient la date au plus tôt de l'opération transport suivante.

La contrainte (1.7) indique que soit l'opération r est avant l'opération s, soit elles sont dans l'ordre inverse.

La contrainte (1.8) indique que les opérations transport sont traitées dans un ordre compatible avec les contraintes technologiques. Ainsi, si deux opérations transport r et s sont réalisées dans cet ordre dans la gamme, alors on peut imposer leur ordre de traitement a_{rs}.

Contrainte de non-anticipation

Cet ensemble de contraintes (1.9) à (1.14) impose qu'une opération transport ne commence que lorsque le job à transporter a atteint le stock de sortie. Pour exprimer cette contrainte, on utilise les opérations transport consécutives déterminées par c_{rs}. Les contraintes (1.9) à (1.13) permettent de positionner les variables c_{rs}, la contrainte (1.14) est la contrainte de non anticipation à proprement parler.

La contrainte (1.9) impose que deux opérations s et r réalisées dans cet ordre ($a_{rs} = 1$) impliquent que l'opération r ne peut être réalisée immédiatement avant s (et donc $c_{rs} = 0$). La contrainte (1.9) permet de positionner c_{rs} relativement à l'ordre des opérations r et s.

$$c_{rs} + a_{rs} \leq 1 \qquad (1.9)$$

où $(r,s) \in T^2, r \neq s$

Si $a_{rs} = 1$, l'opération transport r est réalisée après l'opération transport s. La contrainte (1.8) se réécrit alors $c_{rs} + 1 \leq 1$ qui implique $c_{rs} = 0$.

Si $a_{rs} = 0$, l'opération transport r est réalisée avant l'opération transport s. La contrainte (1.8) se réécrit alors $c_{rs} + 0 \leq 1$ qui est forcément vraie.

La contrainte (1.10) impose que pour tout triplet d'opérations r, s et v réalisées dans cet ordre, l'opération v n'est pas réalisée immédiatement après r ($c_{rv}=0$). La contrainte (1.10) code le "immédiatement" de la définition de c_{rs}.

$$a_{rs}+a_{sv}-c_{rv}\geq 0 \qquad (1.10)$$

où $(r,s,v)\in T^3$ avec $r\neq s$, $r\neq v$ et $s\neq v$

Si $a_{rs}=0$ et $a_{sv}=0$, l'opération transport r est traité avant l'opération transport s, elle-même traitée avant l'opération transport v. Dans ce cas, l'opération transport r n'est pas le prédécesseur immédiat de l'opération transport s. Ainsi, la contrainte se réécrit $0-c_{rv}\geq 0$, elle impose que $c_{rv}=0$.

Si au moins un des deux booléens a_{rs} ou a_{sv} est égal à 1, alors la contrainte se réécrit $c_{rv}\leq 1+..$ et est forcément valide.

Les contraintes (1.9) et (1.10) interdisent qu'une variable c_{rs} ne soit positionnée à 1 alors que l'opération r n'est pas immédiatement avant l'opération s, mais rien n'empêche que toutes ces variables soient positionnées à 0. Les contraintes (1.11), (1.12) et (1.13) imposent pour chaque opération transport qu'elle soit précédée du nombre adéquat d'opérations transport.

$$\sum_{r\in T\, r\neq s} a_{rs}+\sum_{r\in T\, r\neq s} c_{rs}\leq |T|-1 \qquad (1.11)$$

où $s\in T$, $od_s\in F$

La contrainte (1.11) impose que tous les c_{rs} soient nuls pour la première opération transport. La première opération transport s est une opération transport $s\in T$, qui est la première opération de son job $od_s\in F$ et telle que toutes les autres opérations transport sont traitées après elle : $\sum_{r\in T\, r\neq s} a_{rs}=|T|-1$. Pour cette opération, la contrainte se réécrit donc $\sum_{r\in T\, r\neq s} c_{rs}\leq 0$, ce qui implique que tous les c_{rs} sont nuls et qu'aucune opération n'est traitée immédiatement avant s.

Pour les autres opérations transport, il y a au moins une opération traitée avant ($\sum_{r\in T\, r\neq s} a_{rs}<|T|-1$) et la contrainte se réécrit $\sum_{r\in T\, r\neq s} c_{rs}\leq 1$. Cette inégalité est trivialement vérifiée.

La contrainte (1.12) implique que toute opération transport (autre que la première de chaque job) doit avoir une opération traitée immédiatement avant elle.

$$\sum_{r\in T\, r\neq s} c_{rs}=1 \qquad (1.12)$$

où $s\in T$, $od_s\notin F$

La contrainte (1.13) implique que parmi toutes les premières opérations transport, seule la toute première traitée n'a pas d'opération immédiatement traitée avant. Soit s la première opération transport d'un job. S'il existe une opération transport r traitée avant l'opération s ($a_{rs}=0$), la contrainte (1.13) peut être réécrite $\sum_{u\in T\, u\neq s} c_{ks}\geq 1$, ce qui assure qu'une opération transport sera traitée avant. S'il n'existe pas d'opération transport r traitée avant l'opération s, toutes les contraintes (1.13) sont inactives.

$$\sum_{v\in T\, v\neq s} c_{vs}+a_{rs}\geq 1 \qquad (1.13)$$

où $(r,s)\in D_T$, $od_s\in F$

Nombre maximum de jobs simultanément autorisés

Cet ensemble de contraintes assure qu'il n'y a pas plus de N jobs simultanément présents dans le système.

$$W_{ij}-a_{FT_iFT_j}-a_{FT_jLT_i}\leq 0 \qquad (1.14)$$

$$a_{FT_iFT_j}-W_{ij}\leq 0 \qquad (1.15)$$

$$a_{FT_jLT_i}-W_{ij}\leq 0 \qquad (1.16)$$

où $(i,j)\in J^2, i\neq j$

$$\sum_{i\in J,\, i\neq j} W_{ij}\geq n-N \qquad (1.17)$$

où $j \in J$

Les contraintes (1.14) à (1.16) permettent de positionner les booléens W_{ij}. Le booléen W_{ij} est égal à 0 si le job i est entré pendant que le job j est dans le système.

Si le job i est entré pendant que le job j est dans le système alors les opérations transport Ft_i, Ft_j et Lt_i sont traitées dans cet ordre. Dans ce cas, les booléens $a_{FT_i FT_j}$ et $a_{FT_j LT_i}$ sont nuls et la contrainte (1.14) impose que $W_{ij} = 0$. Dans les autres cas, un des deux booléens est égal à un et la contrainte est inactive.

Inversement, si $W_{ij} = 0$, les contraintes (1.15) et (1.16) se réécrivent $a_{FT_i FT_j} \le 0$, $a_{FT_j LT_i} \le 0$. Ceci implique $a_{FT_i FT_j} = 0$, $a_{FT_j LT_i} = 0$ et donc les opérations Ft_i, Ft_j et Lt_i sont traitées dans cet ordre.

La contrainte (1.17) est la contrainte de limitation du nombre de jobs simultanément autorisés à proprement parler. $n - \sum_{i \in J, i \neq j} W_{ij}$ est le nombre de jobs entrés dans le système pendant que le job j y est présent.

Ce nombre doit être inférieur à la limitation N $n - \sum_{i \in J, i \neq j} W_{ij} \le N$.

Contrainte sur la capacité du stock d'entrée

Les contraintes sur la capacité du stock d'entrée limitent le nombre de jobs simultanément présents dans le stock d'entrée. Les contraintes (1.18) à (1.20) permettent de positionner le booléen Y_{ij}, qui indique, lorsqu'il est égal à 1, que l'opération j attend l'opération i dans le stock d'entrée. La contrainte (1.21) est la contrainte limitant la capacité du stock d'entrée.

$$t_i - O_i - tm_i \le t_j - O_j - tm_j - I_j + Hb_{ij} + HY_{ij} \quad (1.18)$$

$$t_j - O_j - tm_j - I_j \le t_i - O_i - tm_i + H(1 - Y_{ij}) + Hb_{ij} - 1 \quad (1.19)$$

$$Y_{ij} + b_{ij} \le 1 \quad (1.20)$$

où $(i, j) \in \widetilde{D}_M$

$$\sum_{j \in I, \mu_i = \mu_j, j \neq i} Y_{ij} \le se_{\mu_i} - 1 \quad (1.21)$$

où $i \in \widetilde{I}$

Lemme

Les contraintes (1.18) à (1.21) expriment linéairement l'équivalence suivante :

$$Y_{ij} = 1 \Leftrightarrow \begin{cases} b_{ij} = 0 \\ t_i - O_i - tm_i > t_j - O_j - tm_j - I_j \end{cases}$$

Preuve

On suppose que $Y_{ij} = 1$.

La contrainte (1.20) se réécrit $b_{ij} \le 0$, et on a donc $b_{ij} = 0$. Les contraintes (1.18) et (1.19) se réécrivent donc $t_i - O_i - tm_i \le t_j - O_j - tm_j - I_j + H$ qui est trivialement réalisée et $t_j - O_j - tm_j - I_j < t_i - O_i - tm_i$. CQFD.

Inversement, on suppose que $b_{ij} = 0$ et $t_i - O_i - tm_i > t_j - O_j - tm_j - I_j$. Dans ce cas, la contrainte (1.20) se réécrit $Y_{ij} \le 1$ qui est trivialement vérifiée. La contrainte (1.18) se réécrit $t_i - O_i - tm_i \le t_j - O_j - tm_j - I_j + HY_{ij}$ et la contrainte (1.19) se réécrit $t_j - O_j - tm_j - I_j \le t_i - O_i - tm_i + H(1 - Y_{ij}) - 1$. Si on prend $Y_{ij} = 0$, la contrainte (1.18) se réécrit $t_i - O_i - tm_i \le t_j - O_j - tm_j - I_j$, ce qui contredit l'inéquation. Ainsi, $Y_{ij} = 1$. CQFD

Ainsi, grâce aux contraintes (1.18) à (1.21), le booléen Y_{ij} est positionné à 1 si l'opération i est traitée avant l'opération j ($b_{ij} = 0$) et si l'opération j est entrée dans le stock (date $t_j - O_j - tm_j - I_j$)

avant le début du traitement de l'opération i (date $t_i - O_i - tm_i$). Ainsi, le booléen Y_{ij} est positionné pour tous les jobs j présents dans le stock au moment où le traitement de l'opération i commence.

La contrainte (1.21) compte $\sum_{j\in I,\mu_i=\mu_j,j\neq i} Y_{ij}$ le nombre de jobs dans le stock présents en même temps que le job i.

Contrainte de capacité du stock de sortie

Cette contrainte assure qu'à chaque instant, la capacité du stock de sortie est respectée.

$$t_i \le t_j - O_j + Hb_{ij} + HZ_{ij} \quad (1.22)$$

$$t_j - O_j \le t_i + Hb_{ij} + H(1 - Z_{ij}) - 1 \quad (1.23)$$

$$Z_{ij} + b_{ij} \le 1 \quad (1.24)$$

où $(i,j) \in \widetilde{D}_M$

$$\sum_{j\in I,\mu_i=\mu_j,j\neq i} Z_{ij} \le ss_{\mu_i} - 1 \quad (1.25)$$

où $i \in \widetilde{I}$

Cette contrainte est totalement identique à la contrainte de capacité de stock d'entrée. La seule différence réside dans la définition de Z_{ij} qui est positionné à 1 si l'opération i est traitée avant l'opération j ($b_{ij} = 0$) et si l'opération j est entrée dans le stock (date $t_j - O_j$) avant le début du départ de l'opération i (date t_i).

Contraintes d'intégrité et de positivité

Toutes les variables sont positives, les variables binaires (b_{ij}, Y_{ij}, Z_{ij}, a_{rs}, c_{rs}, W_{ij}) doivent être nulles ou égales à 1.

Critères d'ordonnancement

La fonction objectif du MILP doit être écrite comme une fonction linéaire des variables. On peut donc inclure les objectifs classiques en ordonnancement. On a par exemple :

- makespan : $C_{\max}$

Pour minimiser le makespan, on doit ajouter les contraintes suivantes permettant de définir la variable $C_{\max}$. Cette contrainte et la minimisation du $C_{\max}$ exprime que $C_{MAX} = \max_{i\in L}(t_i)$:

$$\forall i \in [1,o], \mu_i = U,\ t_i \le C_{\max} \quad (1.27)$$

- Date de fin moyenne : $\overline{C} = \frac{1}{n}\sum_{i\in L} t_i$

3.2 Systèmes munis de stocks gérés en PAPS

Le modèle présenté ci-dessus concerne un stock géré de manière optimale. Il permet donc, entre autres, d'envisager les ordonnancements dans lesquels les stocks sont gérés en PAPS.

Pour modéliser les systèmes munis de stocks gérés en PAPS, il faut donc ajouter des contraintes supplémentaires qui interdisent que les jobs "se doublent" dans les stocks. Les contraintes (1.28) à (1.29) modélisent la politique PAPS sur le stock de sortie. Les contraintes (1.30) et (1.31) modélisent la politique PAPS sur le stock d'entrée.

Toutes ces contraintes assurent donc que les opérations entrent et sortent du stock d'entrée, sont traitées sur la machine, entrent et sortent du stock de sortie dans le même ordre déterminé par les booléens b_{ij}

$$t_i \le t_j + b_{ij}H \quad (1.28)$$

$$t_i - O_i \le t_j - O_j + b_{ij}H \quad (1.29)$$

$$t_i - O_i - tm_i \le t_j - O_j - tm_j + b_{ij}H \quad (1.30)$$

$$t_i - O_i - tm_i - I_i \le t_j - O_j - tm_j - I_i + b_{ij}H \quad (1.31)$$

où $(i,j) \in \widetilde{D}_M$

Si le booléen b_{ij} est positionné à 1, c'est que l'opération i est traitée après l'opération j. Dans ce cas, toutes les contraintes sont trivialement réalisées. Sinon, ($b_{ij} = 0$) les contraintes imposent que l'opération i sorte du stock de sortie avant l'opération j (contrainte 1.28), que l'opération i entre dans le stock de sortie avant l'opération j (contrainte 1.29), que l'opération i commence son traitement avant l'opération j (contrainte 1.30), que l'opération i entre dans le stock d'entrée avant l'opération j (contrainte 1.31).

Les contraintes (1.32) ne sont pas nécessaires, mais sont utiles pour la résolution du problème car elles lient les booléens b_{ij} et a_{rs}. Les contraintes (1.32) indiquent que si deux opérations transport r et s amenant deux jobs sur la même machine, alors ces deux jobs doivent être traités dans le même ordre. On a donc $a_{rs} = 0 \Rightarrow b_{ij} = 0$. De même, la contrainte (1.33) applique le même raisonnement pour des opérations transport partant de la même station.

$$b_{i,j} \le a_{r,s} \qquad (1.32)$$

où $\{r,s\} \in D_T$, $i = od_r$, $j = od_s$ et $\mu_i = \mu_j, \mu_i \neq L, \mu_j \neq U$

$$b_{i,j} \le a_{r,s} \qquad (1.33)$$

où $\{r,s\} \in D_T$, $i = oa_r$, $j = oa_s$ et $\mu_i = \mu_j$

3.3 Expérimentations numériques

Dans cette section, nous présentons des expérimentations numériques des deux programmes linéaires présentés ci-dessus. Ces programmes linéaires sont résolus directement par un solveur linéaire (CPLEX). Les expérimentations sont conduites dans deux buts principaux : présenter les premières solutions optimales pour le problème étudié et mesurer l'impact des règles de gestion du transporteur communément utilisées.

3.3.1 Généralités

Pour conduire nos expérimentations, nous utilisons les instances proposées dans Bilge et Ulusoy (1995). Elles contiennent, entre autres :

- un "layout" qui définit le réseau de transport à l'aide de la matrice des temps de transport en charge (qui incluent les temps de chargement / déchargement) et la matrice des temps de transport à vide.
- deux "jobset" définissant les jobs et leur gamme.

Les jobsets 1 et 2 et le layout 1 utilisés dans cette thèse sont décrits dans le tableau 4-2 et dans le tableau 4-3. Les deux instances ainsi décrites ont des temps de traitement comparables aux temps de transport. Pour les besoins de nos expérimentations numériques, nous avons introduit les instances Jobset1H et Jobset2H. Ces instances ont des temps de traitement hétérogènes (entre les opérations d'un job), et en moyenne supérieurs aux temps de transport. Pour construire ces instances, on ajoute 100 unités de temps aux durées de traitement d'une opération sur deux.

	L/U	M1	M2	M3	M4	L/U
L/U	0	4	6	8	6	0
M1	6	0	2	4	2	6
M2	8	12	0	2	4	8
M3	6	10	12	0	2	6
M4	4	8	10	12	0	4
L/U	0	4	6	8	6	0

Tableau 4-2. Layout 1, (Bilge et Ulusoy, 1995)

Job 1	L/U 0	M1 10	M4 18	L/U 0	
Job 2	L/U 0	M2 10	M4 18	L/U 0	
Job 3	L/U 0	M1 10	M3 20	L/U 0	
Job 4	L/U 0	M2 10	M3 15	M4 12	L/U 0
Job 5	L/U 0	M1 10	M2 15	M4 12	L/U 0
Job 6	L/U 0	M1 10	M2 15	M4 12	L/U 0

Tableau 4-3. Jobset 1, (Bilge et Ulusoy, 1995)

Dans toutes ces instances, on suppose que :

- les temps de transport et les temps de déplacement à vide sont égaux,
- la machine 1 est la station d'entrée,
- la machine 6 est la station de sortie,
- les temps de chargement et de déchargement sont égaux à 1,
- la capacité des stocks d'entrée et de sortie sont égaux à 2.

Toutes les expérimentations sont réalisées sur un "Sun Entreprise 450" muni de quatre processeurs "Ultra Sparc II" cadencé à 450 MHz, muni de 2 Go de mémoire. Tous les calculs sont limités à une journée (86400 s) de temps de calcul.

Pour conduire nos expérimentations, nous dérivons des instances à partir des instances de Bilge et Ulusoy. On nomme "instance Jobset 1 $n=4,\ N=3,\ p_{ij}\times 2,\ t_{ij}/2$", l'instance dans laquelle on réalise les modifications suivantes : on ne considère que les $n=4$ premiers jobs, seulement $N=3$ jobs sont simultanément autorisés dans le système, que les temps de traitement sont multipliés par 2 ($p_{ij}\times 2$), que les temps de transport sont divisés par 2 ($t_{ij}/2$).

La taille d'une instance est caractérisée par le nombre d'opérations (No) à ordonnancer. Ce nombre inclut : le nombre d'opérations transport (Nl), le nombre de déplacements à vide (Nd), le nombre d'opérations machine (Nm), et le nombre d'opérations modélisant les places dans les stocks (Nb) : $No = Nl + Nd + Nm + Nb$. Ce nombre d'opérations peut être calculé de la manière suivante :

- Il y a autant d'opérations transport que d'opération sur les machines de $\tilde{M}$ plus un transport pour déplacer le job sur la station de déchargement. $Nl = |\tilde{I}| + n$
- Il y a autant de déplacements à vide que d'opérations sur les machines de $\tilde{M}$ plus un déplacement de la dernière station vers la station de déchargement. $Nd = |\tilde{I}| + n$
- Le nombre d'opérations sur les machines (Nm) est égal au nombre d'opérations machine sur $\tilde{M}$. $Nm = |\tilde{I}|$
- Le nombre d'opérations modélisant les places sur les stocks (Nb) est égal au nombre de places dans les stocks pour chaque opération plus les deux opérations suivantes : une pour modéliser le job dans le stock de sortie de la station d'entrée et une pour modéliser le job dans le stock d'entrée de la station de sortie. $Nb = 2n + \sum_{i\in\tilde{I}} ss_{\mu_i} + se_{\mu_i}$. Dans de nombreuses applications, les capacités des stocks d'entrée et de sortie sont égales et constantes : $se_{\mu_i} = se$ et $ss_{\mu_i} = ss$. Dans ce cas, on calcule Nb par la formule : $Nb = (se+ss)|\tilde{I}| + 2n$.

Dans le tableau 4-4, nous indiquons le nombre d'opérations pour chaque taille d'instance. Ce nombre d'opérations est calculé pour l'instance Jobset 1 dans laquelle les stocks ont une capacité de 2. Les nombres fournis dans ce tableau sont conformes avec les valeurs observées dans le diagramme de

Gantt fourni dans la figure 4-18. Le nombre d'opérations transport *Nl* est égal au nombre de flèches en trait plein, le nombre de déplacements à vide *Nd* est égal au nombre de flèches en pointillé (note. certaines flèches sont de longueur nulle car le déplacement à vide est de durée nulle), le nombre d'opérations machine *Nm* est le nombre de boîtes grisées (dont le libellé est "job en cours de traitement"), le nombre d'opérations modélisant les places en stock (*Nb*) n'est pas représenté sur le schéma.

n	$\widetilde{I}$	$N_o = 7\left\|\widetilde{I}\right\| + 4n$
1	3	25
2	6	50
3	9	75
4	11	93
5	13	111

Tableau 4-4. Nombre d'opérations à ordonnancer

Pendant le développement du modèle, nous avons tenté de limiter sa taille. Pourtant, le modèle linéaire obtenu ne permet pas de traiter des problèmes de grande taille. Les plus grandes instances testées ont 5 jobs à l'entrée du système, ce qui représente 111 opérations à ordonnancer. Malheureusement, le temps de résolution augmente de manière significative et certaines instances ne sont pas résolues en une journée (86400 secondes) de temps de calcul.

Pour évaluer la qualité de nos résultats, nous utilisons la procédure de séparation / évaluation proposée dans (Lacomme *et al.*, 2005). Cette procédure est basée sur la décomposition du problème en deux sous-problèmes.

- un algorithme de séparation / évaluation dont le but est de déterminer l'ordre optimal d'entrée des jobs dans le système.
- une simulation à événements discrets pour évaluer le makespan qui respecte au mieux cet ordre.

Cette simulation utilise la politique de gestion PAPS pour la gestion des stocks d'entrée et de sortie. De plus, elle utilise une règle de gestion pour piloter le véhicule, cette règle détermine quelle opération transport réaliser quand plusieurs sont possibles : la règle PAPS (Premier Arrivé Premier Servi) choisit le transport du premier job qui a atteint son stock de sortie, la règle STT (Shortest Traveling Time) choisit l'opération transport de plus faible durée, la règle MOQS (Most Operation in Queue Station) choisit l'opération transport dans le stock de sortie comportant le plus de jobs.

Comme la plupart des modèles basés sur la simulation, le modèle proposé par (Lacomme *et al.*, 2005) n'envisage que des ordonnancements sans délai. Comme nous l'avons signalé dans le chapitre 2[7], cette classe d'ordonnancement est heuristique : il se peut qu'il n'existe pas d'ordonnancement optimal sans délai. Il y a donc deux raisons pour que la procédure de séparation / évaluation ne trouve pas des ordonnancements optimaux : la règle de gestion du transporteur et la simulation sont heuristiques.

Dans leur article, les auteurs proposent des résultats pour des instances supérieures à 20 jobs. Nous avons réutilisé leur algorithme pour proposer les résultats présentés.

Dans les sections suivantes, on utilise les notations :

- $C_{\max}^{MILP^{(OPT)}}$ Makespan optimal du problème avec politique optimale de gestion des stocks
- $C_{\max}^{MILP^{(PAPS)}}$ Makespan optimal du problème avec politique PAPS pour la gestion des stocks

[7] chapitre 2, section 3.3, page 60

- *UB* Borne supérieure du makespan calculé par la procédure de séparation / évaluation de (Lacomme *et al.*, 2005).

3.3.2 Systèmes avec politique de gestion optimale des stocks

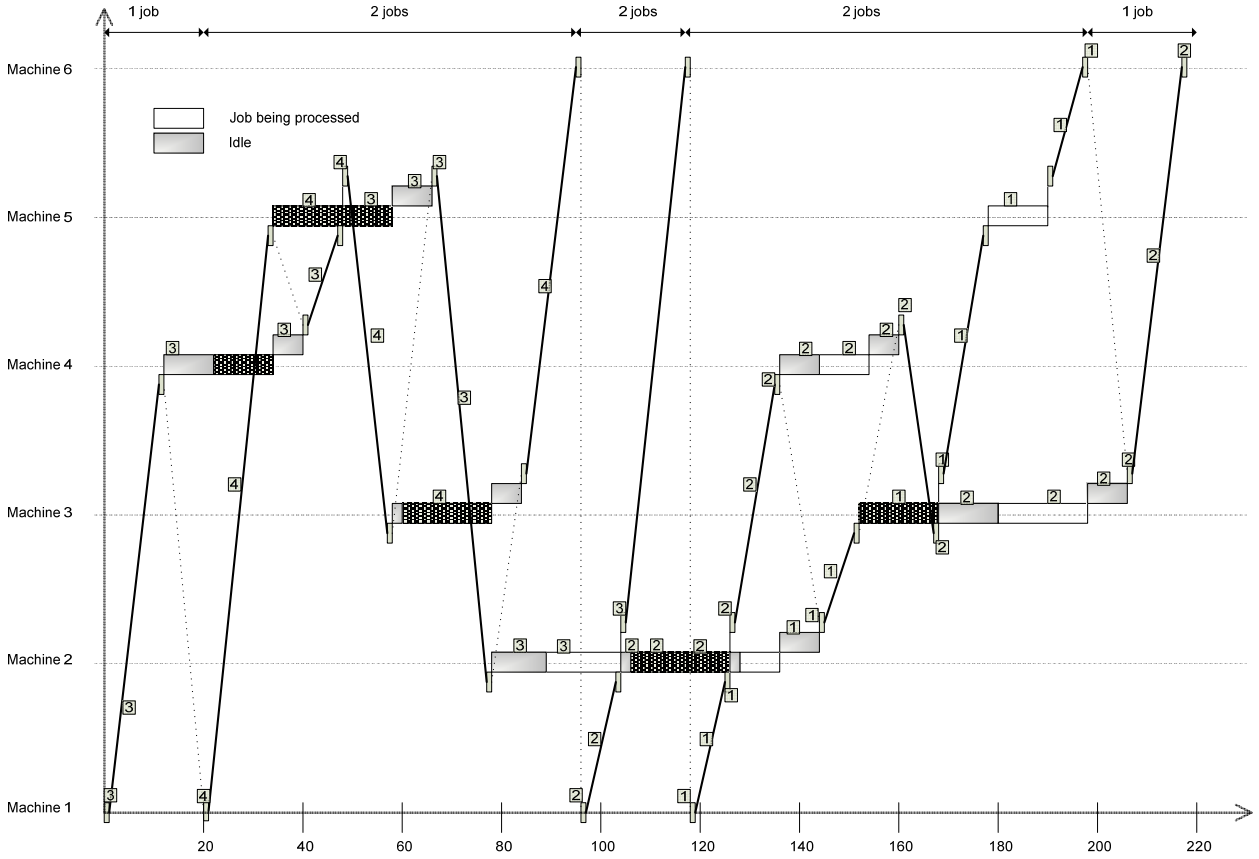

Figure 4-18. Ordonnancement optimal pour la politique optimale de gestion des stocks (N=2)

Dans cette section, on s'intéresse au problème avec politique de gestion optimale des stocks. L'ordonnancement optimal de ce problème est présenté dans la figure 4-18. Les machines sont disposées sur l'axe des ordonnées et les dates sur l'axe des abscisses. Les opérations sur les machines affichées dans des boîtes grisées, les boîtes décalées vers le bas (resp. le haut) représentent l'attente des jobs dans les stocks d'entrée (resp. de sortie). Les opérations transport sont représentées par des flèches pleines, les déplacements à vide sont représentés par des flèches en pointillés.

Dans l'ordonnancement de la figure 4-18, le second job (numéro 4) entre dans le système à la date 21. Pour aller chercher ce job, le transporteur commence par un déplacement à vide de la machine 3 à la station de chargement. Le job 4 est chargé sur le véhicule (ce temps de chargement est représenté par un rectangle blanc). Ensuite, le transport peut être réalisé de la station d'entrée vers la première machine visitée par le job 4 (i.e. machine 4). D'abord, le job est déchargé du véhicule dans le stock d'entrée de la station. Le traitement du job 4 est alors commencé et se termine à la date 48. Ensuite, la seconde opération du job 3 est commencée pour terminer à la date 58. À ce moment, une requête est émise à l'intention du transporteur qui commence son déplacement de la machine 2 vers la machine 4. Ce déplacement ne peut commencer plus tôt à cause de la contrainte de non-anticipation. Cette situation est mise en évidence dans la figure 4-19. Dans cette figure, le déplacement à vide de la machine 2 à la machine 4 démarre exactement quand le job 4 termine son traitement et entre dans le stock de sortie. Dès que ce déplacement à vide se termine, le job est chargé et l'opération transport commence.

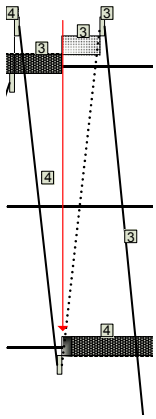

Figure 4-19. Contrainte de non-anticipation

Dans la figure 4-18, le nombre de jobs est limité à deux. On observe que cette limitation est bien respectée : entre les dates 0 et 96, seulement deux jobs sont dans le système. À chaque fois qu'un job sort du système, un nouveau peut rentrer. Dans le "layout 1", les stations d'entrée et de sortie sont confondues : le temps de transport entre ces deux stations est nul. C'est pourquoi, à chaque fois qu'un job est déposé dans la station de sortie, un autre est instantanément pris dans la station d'entrée. Cette situation est observée dans les ordonnancements optimaux proposés, mais la contrainte ne l'impose pas.

En augmentant le nombre de jobs simultanément autorisés, on permet au système de faire entrer le troisième job plus tôt dans l'ordonnancement. Ceci permet un ordonnancement optimal de plus faible makespan (=218) présenté dans la figure 4-20. Le diagramme de Gantt indique clairement que le nombre de jobs simultanément autorisés est de 3. Les deux solutions proposées sont radicalement différentes et rien ne permet à priori de passer de l'une à l'autre.

Pour évaluer l'influence du nombre de jobs simultanément autorisés, nous calculons le makespan optimal pour différentes valeurs de N. Les résultats pour l'instance jobset 1 $p_{ij} \times 1$, $t_{ij} \times 1$ sont présentés dans la figure 4-21. Sur la figure, on observe que le makespan décroît quand le nombre de jobs simultanément autorisés augmente. En effet, un ordonnancement optimal pour une valeur de N est une solution réalisable pour une valeur supérieure. Ainsi, le makespan optimal pour la valeur supérieure de N est forcément inférieur et la fonction est donc décroissante.

Ensuite, les résultats compacts pour le jobset 1, layout 1 sont fournis dans le tableau 4-5. Ces résultats sont, à notre connaissance, les premiers résultats optimaux. Le tableau montre l'influence des coefficients p_{ij}, t_{ij}, n et N. Pour chaque valeur de ces paramètres, le tableau donne les makespan optimaux pour les problèmes avec gestion optimale des stocks ($C_{\max}^{MILP^{(OPT)}}$) et pour les problèmes avec gestion des stocks en PAPS $C_{\max}^{MILP^{(PAPS)}}$.

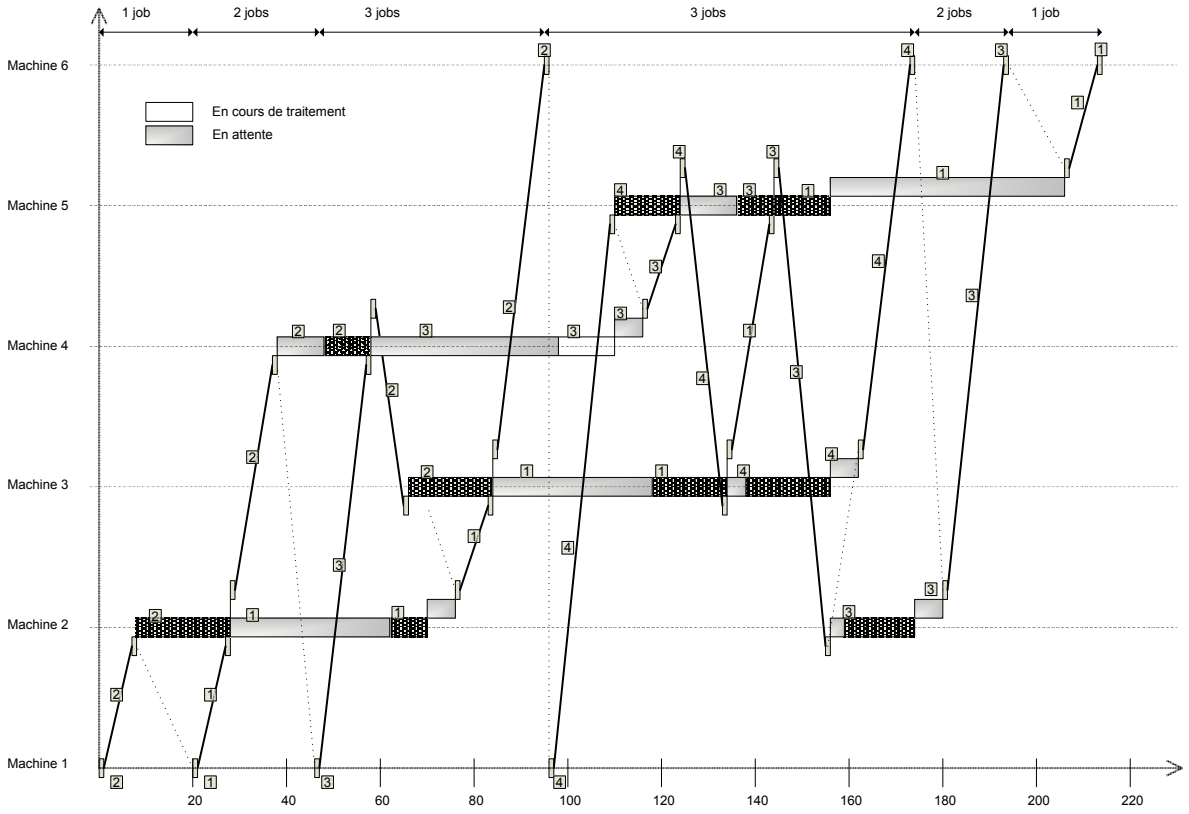

Figure 4-20. Ordonnancement optimal pour la politique optimale de gestion des stocks (N=3)

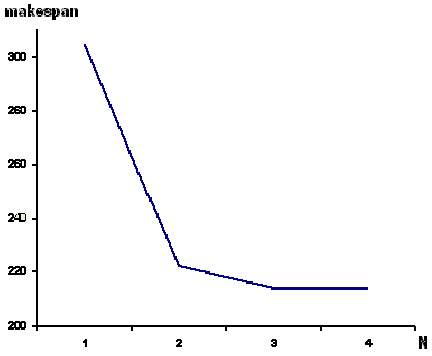

Figure 4-21. Variation du makespan en fonction de N pour le jobset 1, n=4 et $p_{ij} \times 1$, $t_{ij} \times 1$

Quand les temps de transport sont plus grands que les temps de traitement, le transporteur devient la ressource critique et les déplacements à vide pénalisent grandement l'ordonnancement optimal. Ainsi, le transporteur tend à ne traiter qu'un seul job à la fois et à le suivre depuis son entrée jusqu'à sa sortie du système. Plus précisément, quelle que soit la valeur de N, la solution optimale consiste à n'autoriser simultanément qu'un seul job dans le système. On peut vérifier ceci dans les tableaux de résultats pour les instances $p_{ij}/2$, $t_{ij} \times 2$ et $p_{ij}/2$, $t_{ij} \times 3$ pour lesquelles toutes les solutions optimales sont les mêmes quelle que soit la valeur de N. Cette observation est intuitive, mais les solutions proposées par la procédure de séparation/évaluation ne vérifient pas cette propriété. Ceci montre que la simulation ne permet pas toujours d'envisager la solution optimale. Bien sûr, dans le cas où les temps de transport ne sont pas négligeables, cette remarque ne tient plus.

p_{ij}	t_{ij}	*No*	*n*	*N*	$C_{\max}^{MILP^{(OPT)}}$	$C_{\max}^{MILP^{(PAPS)}}$	*UB*
×1	×1	52	2	1	156	156	156
				2	120	120	128
		78	3	1	237	237	237
				2	182	182	194
				3	178	178	190
		97	4	1	305	305	305
				2	218	218	226
				3	214	214	234
				4	214	214	224
		109	5	1	366	366	366
				2	260	260	271
				3	256	248	256
				4	250	248	268
				5	260	248	272
×2	/2	52	2	1	212	212	212
				2	142	142	149
		78	3	1	309	309	309
				2	199	199	202
				3	148	148	148
		97	4	1	394	394	394
				2	227	227	227
				3	193	193	202
				4	160	160	164
		109	5	1	465	465	465
				2	272	272	272
				3	213	214	217
				4	178	178	181
				5	169	169	175
×3	/2	52	2	1	296	296	296
				2	198	198	203
		78	3	1	428	428	428
				2	270	270	275
				3	199	199	211
		97	4	1	545	545	545
				2	306	306	307
				3	260	260	270
				4	205	205	216
		109	5	1	641	641	641
				2	360	360	369
				3	276	276	276
				4	241	237	241
				5	210	210	227

p_{ij}	t_{ij}	*No*	*n*	*N*	$C_{\max}^{MILP^{(OPT)}}$	$C_{\max}^{MILP^{(PAPS)}}$	*UB*
/2	×2	52	2	1	170	170	170
				2	170	170	220
		78	3	1	271	271	271
				2	271	271	318
				3	271	271	324
		97	4	1	353	353	363
				2	353	353	402
				3	353	353	397
				4	353	353	418
		109	5	1	431	431	431
				2	430	426	466
				3	431	426	465
				4	435	426	475
				5	431	426	520
/2	×3	52	2	1	226	226	226
				2	226	226	322
		78	3	1	365	365	365
				2	365	365	458
				3	365	365	474
		97	4	1	477	477	477
				2	477	477	588
				3	477	477	577
				4	477	477	612
		109	5	1	585	585	585
				2	585	585	674
				3	585	585	675
				4	585	585	691
				5	585	585	650

Tableau 4-5. Solutions optimales pour jobset 1 avec politique optimale de gestion des stocks

3.3.3 Systèmes avec politique de gestion des stocks en PAPS

Le tableau 4-5 donne les solutions optimales pour ce type de problème. En comparant ces résultats à la politique de gestion optimale, on observe que la politique PAPS a une influence

négligeable sur les makespans optimaux pour les instances de petite taille ($n \leq 4$). Cependant, on ne peut tirer aucune conclusion sur les ordonnancements optimaux, car même si les makespans sont identiques, rien n'indique que le nombre d'ordonnancements optimaux soient comparables dans les deux problèmes. Autrement dit, il est probable qu'il y ait beaucoup plus d'ordonnancements optimaux pour le problème avec stock optimaux que dans le problème avec stock géré en PAPS.

3.4 Impact de la règle PAPS sur les stocks optimaux

Dans la section précédente, nous avons montré les solutions optimales pour le problème dont les stocks sont gérés de manière optimale ou non. Dans cette section, nous présentons plus particulièrement l'influence de la règle PAPS pour les stocks d'entrée et de sortie sur les solutions. Pour observer une différence entre les deux makespans, il faut qu'il y ait au moins deux jobs présents simultanément dans un stock. La section précédente a montré que ceci arrive pour des instances avec plus de cinq jobs. Dans cette section, on compare les deux politiques de gestion des stocks sur les instances hétérogènes (Instances Jobset 1H et Jobset 2H).

Les résultats sont fournis dans le tableau 4-6 pour l'instance Jobset 1H et le tableau 4-7 pour l'instance Jobset 2H. Contrairement au tableau 4-5, les résultats montrent une différence entre $C_{\max}^{MILP^{(OPT)}}$ et $C_{\max}^{MILP^{(PAPS)}}$ dès $n=4$. Par exemple, pour $n=4$ et $N=3$, la solution optimale est de 406 pour $C_{\max}^{MILP^{(OPT)}}$ et de 410 pour $C_{\max}^{MILP^{(PAPS)}}$. La différence entre ces deux makespans optimaux est de 0,99%, ce qui est relativement faible mais tout de même non négligeable. En effet, en obtenant un écart de 0,99% alors que seulement 3 jobs sont autorisés dans le système, on peut légitimement supposer que cette différence augmente lorsque le nombre de jobs augmente.

n	N	$C_{\max}^{MILP^{(OPT)}}$	$C_{\max}^{MILP^{(PAPS)}}$	Écart %
2	1	456	456	0.00
	2	310	310	0.00
3	1	737	737	0.00
	2	468	468	0.00
	3	322	322	0.00
4	1	905	905	0.00
	2	510	510	0.00
	3	406	410	-0.97
	4	350	352	-0.57
5	1	1066	1066	0.00
	2	609	609	0.00
	3	475	476	-0.21
	4	402	402	0.00
	5	394	400	-0.01

Tableau 4-6. Évaluation de la règle PAPS pour la gestion optimale des stocks (Jobset 1H)

Le tableau 4-7 montre que pour l'instance Jobset 2H, les mêmes résultats ne sont pas observés. Ceci nous pousse à croire que l'influence de la politique de gestion des stocks dépend fortement des instances considérées.

Comme nous l'avons indiqué ci-dessus, deux solutions de même makespans peuvent être radicalement différentes. Ainsi, nous montrons qu'il est difficile de transformer une solution optimale de $C_{\max}^{MILP^{(OPT)}}$ en une solution optimale de $C_{\max}^{MILP^{(PAPS)}}$, nous montrons donc qu'il est important de prendre en compte les contraintes PAPS sur les stocks. La figure 4-22 montre un exemple d'ordonnancement optimal pour $C_{\max}^{MILP^{(OPT)}}$. Dans cet ordonnancement, la contrainte PAPS est violée au moment où le job 2 sort sur la machine 1 avant le job 1 alors qu'ils sont rentrés dans l'ordre inverse. La figure 4-23 montre un exemple d'ordonnancement optimal pour $C_{\max}^{MILP^{(PAPS)}}$, cet ordonnancement est très différent, et la transformation de l'ordonnancement $C_{\max}^{MILP^{(OPT)}}$ en l'ordonnancement $C_{\max}^{MILP^{(PAPS)}}$ n'est pas triviale. Bien

entendu, il existe peut être un ordonnancement optimal de $C_{\max}^{MILP^{(OPT)}}$ plus proche, que l'on pourrait facilement transformer en une solution de $C_{\max}^{MILP^{(PAPS)}}$.

n	N	$C_{\max}^{MILP^{(OPT)}}$	$C_{\max}^{MILP^{(PAPS)}}$	Écart %
2	1	312	312	0.00
	2	200	200	0.00
3	1	470	470	0.00
	2	324	324	0.00
	3	272	272	0.00
4	1	641	641	0.00
	2	375	375	0.00
	3	324	324	0.00
	4	284	284	0.00
5	1	912	912	0.00
	2	525	525	0.00
	3	421	421	0.00
	4	382	382	0.00
	5	382	382	0.00

Tableau 4-7. Évaluation de la règle PAPS pour la gestion optimale des stocks (Jobset 2H)

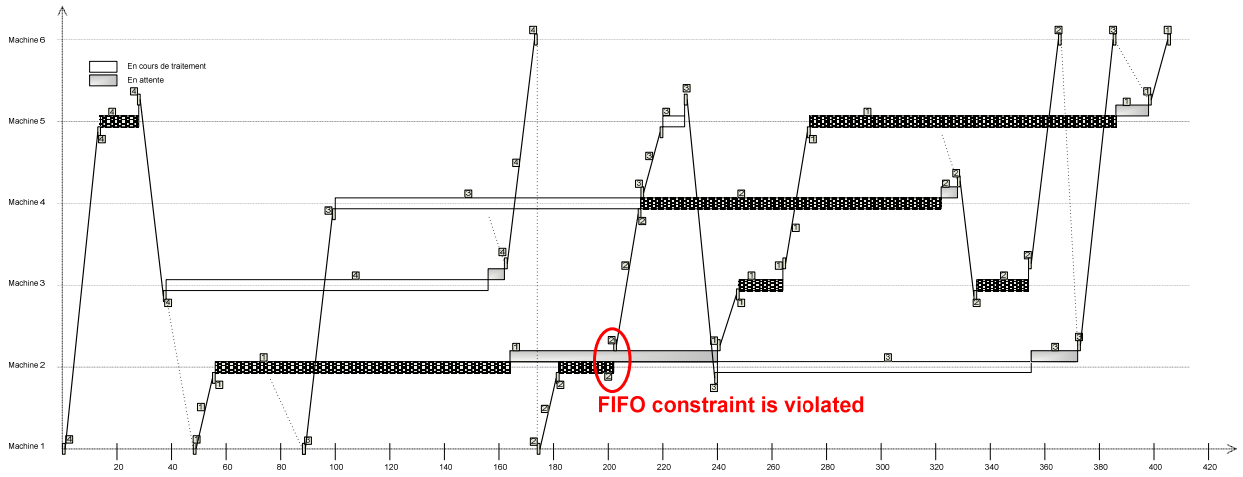

Figure 4-22. Gantt d'une solution pour le Jobset 1H, $n=4$, $N=3$ et gestion optimale des stocks

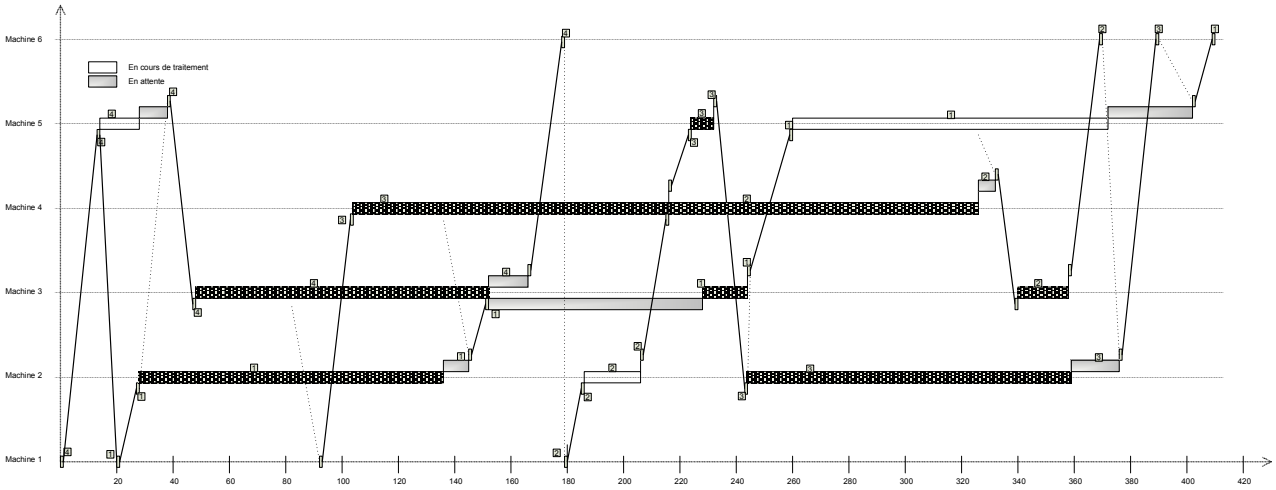

Figure 4-23. Gantt d'une solution pour le Jobset 1H, $n = 4$, $N = 3$ et gestion PAPS des stocks

3.5 Impact des règles de gestion du transporteur

Dans les systèmes flexibles de production, le transporteur est couramment géré suivant une règle : lorsque deux opérations transport sont possibles et que le transporteur est libre, le transporteur choisit l'opération transport à réaliser suivant une règle. Ceci est quelquefois imposé par un système construit de cette façon ou cela peut être un choix d'ordonnancement. Dans les deux cas, les performances de telles règles ne sont pas bien connues car il n'existe pas dans la littérature, à notre connaissance de solution optimale pour ce type de problème.

Le tableau 4-8 montre que les solutions optimales pour des règles de gestion fixées sont relativement loin de l'optimum. Le tableau fournit la meilleure règle de gestion parmi 4 : PAPS, SPT, STT et MOQS. Les solutions observées sont à 4,49% en moyenne de la solution optimale et plus le nombre de jobs augmente et plus cet écart est important. Ces résultats moyens ne sont pas particuliers à l'instance montrée, ils ont été observés pour d'autres instances. Cette étude étend celle proposée par (Lacomme *et al.*, 2005).

n	N	$C_{\max}^{MILP^{(FIFO)}}$	**Meilleure règle de gestion du véhicule**	**Écart %**
2	1	156	156	0.0
	2	120	128	6.2
3	1	237	237	0.0
	2	182	194	6.2
	3	178	190	6.3
4	1	305	305	0.0
	2	218	226	3.5
	3	214	234	8.5
	4	214	224	4.4
5	1	366	366	0.0
	2	260	271	4.1
	3	248	256	3.1
	4	248	268	7.5
	5	248	272	8.8

Tableau 4-8. Performances de la procédure de séparation / évaluation pour l'instance Jobset 1, $p_{ij} \times 1$ et $t_{ij} \times 1$

3.6 Impact des contraintes sur les solutions optimales

De plus, nous proposons d'étudier l'impact de certaines contraintes sur les solutions optimales. Pour cela, nous supprimons les contraintes dans le modèle linéaire, ce qui permet d'évaluer la qualité des modèles qui ne prennent pas en compte ces contraintes. Dans les expérimentations numériques, nous proposons de supprimer les contraintes :

- capacité limitée des stocks
- non-anticipation des déplacements du transporteur
- transport à vide
- contrainte de disjonction pour le transporteur

Nous proposons trois tableaux de résultats pour mettre en évidence l'importance des contraintes. Le tableau 4-9 traite de l'instance Jobset 1, $p_{ij} \times 2$, $t_{ij}/2$, le tableau 4-10 traite de l'instance Jobset 1, $p_{ij}/2$, $t_{ij} \times 2$, le tableau 4-11 traite de l'instance Jobset 1, $p_{ij} \times 1$, $t_{ij} \times 1$. Dans ces trois tableaux, la colonne 1 est n le nombre de jobs à ordonnancer, la colonne 2 est N le nombre de jobs simultanément autorisés et la colonne 3 est la solution optimale du problème $C_{\max}^{MILP^{(FIFO)}}$ (i.e. c'est-à-dire le problème avec toutes ses contraintes). Ensuite, dans chaque tableau, la suppression de chacune des quatre contraintes est envisagée. Pour chaque contrainte, la valeur du makespan optimal (colonne $C_{\max}$) et l'écart à la solution optimale (colonne Gap) sont donnés.

			Sans contrainte de capacité des stocks		**Sans anticipation**		**Sans déplacement à vide**		**Sans disjonction pour les transports en charge**	
n	**N**	$C_{\max}^{MILP^{(FIFO)}}$	$C_{\max}$	**Gap %**	$C_{\max}$	**Gap %**	$C_{\max}$	**Gap %**	$C_{\max}$	**Gap %**
2	1	212	212	0.00	212	0.00	212	0.00	135	-36.32
	2	142	142	0.00	135	-4.93	135	-4.93	135	-4.93
3	1	309	309	0.00	309	0.00	309	0.00	135	-56.31
	2	199	199	0.00	190	-4.52	190	-4.52	135	-32.16
	3	148	148	0.00	137	-7.43	137	-7.43	135	-8.78
4	1	394	394	0.00	394	0.00	394	0.00	137	-65.23
	2	227	227	0.00	219	-3.52	219	-3.52	137	-39.65
	3	193	193	0.00	182	-5.70	182	-5.70	137	-29.02
	4	160	160	0.00	152	-5.00	152	-5.00	137	-14.38
5	1	465	465	0.00	465	0.00	465	0.00	137	-70.54
	2	272	272	0.00	250	-8.09	250	-8.09	137	-49.63
	3	214	214	0.00	200	-6.54	200	-6.54	137	-35.98
	4	178	178	0.00	173	-2.81	173	-2.81	137	-23.03
	5	169	169	0.00	163	-3.55	163	-3.55	137	-18.93
Avg. %				0.00		-3.72		-3.72		-34.64

Tableau 4-9. Évaluation de l'influence des contraintes pour Jobset 1, $p_{ij} \times 2$, $t_{ij}/2$

			Sans contrainte de capacité des stocks		**Sans anticipation**		**Sans déplacement à vide**		**Sans disjonction pour les transports en charge**	
n	*N*	$C_{max}^{MILP^{(FIFO)}}$	c_{max}	*Gap %*	c_{max}	*Gap %*	c_{max}	*Gap %*	c_{max}	*Gap %*
2	1	170	170	0.00	170	0.00	168	-1.18	92	-45.88
	2	170	170	0.00	170	0.00	114	-32.94	92	-45.88
3	1	271	271	0.00	271	0.00	267	-1.48	101	-62.73
	2	271	271	0.00	271	0.00	202	-25.46	101	-62.73
	3	271	271	0.00	271	0.00	190	-29.89	101	-62.73
4	1	353	353	0.00	353	0.00	347	-1.70	101	-71.39
	2	353	353	0.00	353	0.00	250	-29.18	101	-71.39
	3	353	353	0.00	353	0.00	250	-29.18	101	-71.39
	4	353	353	0.00	353	0.00	250	-29.18	101	-71.39
5	1	431	431	0.00	431	0.00	423	-1.86	101	-76.57
	2	426	426	0.00	426	0.00	310	-27.23	101	-76.29
	3	426	426	0.00	426	0.00	310	-27.23	101	-76.29
	4	426	426	0.00	426	0.00	310	-27.23	101	-76.29
	5	426	426	0.00	426	0.00	310	-27.23	101	-76.29
Avg. %				0.00		0.00		-20.78		-67.66

Tableau 4-10. Évaluation de l'influence des contraintes pour Jobset 1, $p_{ij}/2$, $t_{ij} \times 2$

			Sans contrainte de capacité des stocks		**Sans anticipation**		**Sans déplacement à vide**		**Sans disjonction pour les transports en charge**	
n	*N*	$C_{\max}^{MILP^{(FIFO)}}$	$C_{\max}$	*Gap %*	$C_{\max}$	*Gap %*	$C_{\max}$	*Gap %*	$C_{\max}$	*Gap %*
2	1	156	156	0.00	156	0.00	154	-1.28	90	-42.31
	2	120	120	0.00	120	0.00	94	-21.67	90	-25.00
3	1	237	237	0.00	237	0.00	233	-1.69	90	-62.03
	2	182	182	0.00	182	0.00	149	-18.13	90	-50.55
	3	178	178	0.00	178	0.00	107	-39.89	90	-49.44
4	1	305	305	0.00	305	0.00	299	-1.97	94	-69.18
	2	218	218	0.00	218	0.00	164	-24.77	94	-56.88
	3	214	214	0.00	214	0.00	140	-34.58	94	-56.07
	4	214	214	0.00	214	0.00	126	-41.12	94	-56.07
5	1	366	366	0.00	366	0.00	358	-2.19	94	-74.32
	2	260	260	0.00	258	-0.77	199	-23.46	94	-63.85
	3	248	248	0.00	247	-0.40	161	-35.08	94	-62.10
	4	248	248	0.00	247	-0.40	156	-37.10	94	-62.10
	5	248	248	0.00	247	-0.40	156	-37.10	94	-62.10
Avg. %				0.0		-0.14		-22.86		-56.57

Tableau 4-11. Évaluation de l'influence des contraintes pour Jobset 1, $p_{ij} \times 1$, $t_{ij} \times 1$

Le premier ensemble de contraintes supprimées est l'ensemble des contraintes de capacité des stocks. Pour les trois instances testées et quel que soit le nombre de jobs, cette contrainte ne modifie jamais le makespan optimal de la solution. Encore une fois, on ne peut être sûr qu'aucune solution optimale sans la contrainte ne la viole. Pourtant, le fait qu'aucune instance ne soit différente nous pousse à croire que l'influence de la contrainte de capacité des stocks est faible. Ceci n'est pas surprenant dans la mesure où la contrainte de capacité des stocks n'est activée que lorsqu'il y a au moins autant de jobs que la capacité du stock le permet. Pour le stock d'entrée de capacité 2, la contrainte est activée lorsqu'un job est en cours de traitement, le stock et plein et qu'un quatrième job tente de se déplacer vers la station. Pour le stock de sortie, la contrainte est activée dès que le stock est plein et qu'un job a terminé son traitement sur la machine. Ainsi, seules les instances $(n=4, N=4)$, $(n=5, N=4)$ et $(n=5, N=5)$ sont susceptibles d'activer la contrainte de capacité de stock d'entrée.

Nous avons déjà observé dans les sections précédentes que les contraintes PAPS sur les stocks ont une influence non négligeable sur les résultats. Nous n'avons pas réintroduit ces résultats dans les tableaux.

Le deuxième ensemble de contraintes concerne la non-anticipation des opérations transport. La colonne correspondante mesure donc l'influence de l'hypothèse simplificatrice qui consiste à négliger une éventuelle non-anticipation. Suivant l'instance observée, cette hypothèse a une influence différente. Pour l'instance $p_{ij} \times 2$, $t_{ij}/2$ et $p_{ij} \times 1$, $t_{ij} \times 1$, l'écart est non-nul alors que pour l'instance $p_{ij} \times 2$, $t_{ij}/2$ cette hypothèse semble n'avoir aucune influence. Autrement dit, pour les instances dont les temps de transport et de traitement sont comparables, cette hypothèse ne change pas la qualité de la solution. Par contre, quand les temps de transport et de traitement ne sont pas comparables, la contrainte de non-anticipation a une influence significative.

Une hypothèse couramment admise pour les problèmes de transport est de négliger les temps de déplacement à vide. Les colonnes 8 et 9 des tableaux montrent que l'écart est significatif entre les solutions prenant cette contrainte en compte ou non. Bien sûr, plus les temps de transport sont petits (relativement au temps de traitement) et plus l'influence des déplacement à vide est faible. Ainsi, les déplacements à vide ne changent pas la valeur du makespan optimal pour l'instance $p_{ij} \times 2$, $t_{ij}/2$ (i.e. 3,72% est identique à la colonne précédente). Par contre, à partir du moment où les temps de transport et traitement sont équilibrés, négliger la contrainte permet de trouver des solutions à environ 20% en dessous de l'optimum.

Pour finir, l'influence de la disjonction du transporteur permet de montrer (colonne 10 et 11) que dans tous les cas, les solutions trouvées sont à 30% en dessous de l'optimum. Bien sûr, lorsque le temps de transport augmente (relativement au temps de traitement), la contrainte devient de plus en plus prépondérante. Ainsi, l'écart le plus important (67%) est observé pour l'instance $p_{ij}/2$, $t_{ij} \times 2$. Il est à noter que la suppression de la disjonction ne permet plus d'exprimer la contrainte du nombre limité de jobs simultanément autorisés. Ainsi, quelle que soit la valeur de N, tous les résultats sont identiques.

En conclusion, l'analyse des résultats permet de faire les remarques suivantes. Ces remarques sont vraies pour les instances étudiées, et ne sont pas directement généralisables. Pourtant, nous pensons que les remarques suivantes restent vraies pour des instances de taille supérieure :

- La capacité limitée des stocks ne modifie pas beaucoup la qualité des solutions trouvées (même si les solutions risquent d'être différentes).
- La contrainte de non-anticipation a une influence faible mais non négligeable sur la qualité des solutions.
- L'influence des déplacements à vide est importante pour les instances où les temps de transport sont supérieurs au temps de traitement. Elle est de plus en plus faible quand les temps de transport deviennent de plus en plus faibles.
- La disjonction des transporteurs est une contrainte importante sans laquelle les solutions optimales sont très différentes. Ainsi, même une instance dont les temps de transport sont quatre fois moins importants que les temps de traitement doit prendre en compte la disjonction du transporteur. Sans quoi, les solutions sont inférieures de 30% de la solution optimale.

3.7 Conclusion sur la formalisation linéaire

La formalisation linéaire proposée est la première formalisation linéaire pour le problème de jobshop avec transport. Elle contient des contraintes originales et a le mérite d'unifier dans un même programme linéaire toutes les contraintes.

Grâce à cette formalisation, nous pouvons donner les premiers résultats optimaux pour les problèmes d'ordonnancement de type FMS. Ces résultats nous ont montré quelles contraintes sont prépondérantes et dans quelle mesure.

Malheureusement, la résolution directe ne permet pas de résoudre des problèmes de grande taille (5 jobs maximums). C'est pourquoi, nous proposons de développer des méthodes approchées. Pour proposer des méthodes approchées, dans la partie suivante d'étendre le modèle de graphe conjonctif-disjonctif.

4 Modèle de graphe: proposition d'une extension

Le modèle de graphe conjonctif - disjonctif a été présenté dans le chapitre 2 pour le problème de jobshop. Les principes de ce graphe peuvent largement être adaptés à d'autres problèmes disjonctifs. Jusqu'à maintenant, le modèle de graphe a principalement été utilisé pour les problèmes classiques de jobshop et RCPSP. Mais récemment, le nombre d'articles proposant de prendre en compte des contraintes supplémentaires augmente.

Par exemple, Mascis et Pacciarelli (2002) ont proposé un modèle basé sur le graphe conjonctif - disjonctif qu'ils ont appelé "graphe alternatif". Ce graphe généralise le graphe conjonctif - disjonctif classique et permet, entre autres, de modéliser les contraintes de "sans attente" et de "blocage". Dans le modèle de graphe conjonctif - disjonctif classique, une arête modélise qu'une des deux contraintes suivante est active : $t_i + c \leq t_j$ ou exclusif $t_j + d \leq t_i$. Le modèle classique impose donc que les opérations concernées par les deux contraintes soient les mêmes. Dans le graphe alternatif, les deux opérations peuvent être complètement différentes. La représentation de la disjonction par une arête n'est donc plus adaptée et remplacée par une paire d'arcs : soit deux contraintes à modéliser $t_i + c \leq t_j$ ou exclusif $t_k + d \leq t_l$, le graphe alternatif contient donc un arc de i vers j et un arc de k vers l. Lorsque l'ordre entre les deux opérations est connu, un des deux arcs alternatif est retenu et l'autre est supprimé. Ce graphe contient des arcs positifs, négatifs et des cycles.

De même, les contraintes de stock ont été prises en compte par plusieurs auteurs. Smutnicki (1998) et Nowicki (1999) ont proposé un graphe modélisant les contraintes de stock pour les problèmes de flowshop de permutation à deux machines. Dans (Brucker et Heitmann 2003), les auteurs étendent cette formalisation pour prendre en compte les stocks dans les flowshops en général. Par rapport au graphe conjonctif - disjonctif classique, une contrainte sur l'ordre des opérations est ajoutée et des arcs sont insérés dans le graphe. Il y a donc des arcs de longueur positive et négative, ces arcs formant des cycles. Pour calculer le plus long chemin, les auteurs proposent un algorithme de plus long chemin dédié. Cet algorithme est linéaire en le nombre d'arcs.

Pour le problème avec un transporteur, Hurink et Knust (2002) ont proposé un modèle de graphe. Ce modèle contient des nœuds représentant les opérations machine et des nœuds représentant les opérations transport. Le graphe ne contient que des arcs de longueur positive et les auteurs proposent un algorithme dédié pour calculer les plus longs chemins. Dans Hurink et Knust (2002), ce modèle est utilisé dans un algorithme tabou pour optimiser les opérations transport lorsque les opérations machine sont connues. Le même modèle est utilisé dans (Brucker et Knust, 2002) pour calculer des bornes inférieures. Enfin, Hurink *et al.* (2005) proposent un algorithme complet pour optimiser tout le problème de transport.

Dans la suite, nous proposons un modèle de graphe conjonctif - disjonctif contenant toutes ces contraintes et des contraintes supplémentaires.

4.1 Schéma d'optimisation basé sur le graphe

Dans cette section, nous présentons notre schéma d'optimisation basé sur le graphe conjonctif - disjonctif. Depuis (Roy, 1964), de nombreux auteurs ont utilisés le graphe conjonctif - disjonctif en adoptant un schéma similaire : le graphe conjonctif - disjonctif est construit, puis orienté à partir d'un ordre valide des opérations, le graphe conjonctif obtenu est évalué à l'aide d'un algorithme de plus long chemin de type Bellmann. Dans la section précédente, nous avons présenté des articles utilisant le graphe conjonctif - disjonctif pour des problèmes plus complexes. Le schéma d'utilisation pour ces problèmes change légèrement, il consiste à restreindre l'ordre des opérations (cf. (Hurink *et al.* 2005) et (Brucker et Heitmann, 2003)) ou à ajouter des arcs dans le graphe. Ci-dessous, nous présentons un schéma d'optimisation basé sur la notion de graphe conjonctif - disjonctif unifiant ces différentes approches. Pour un problème général, nous décrivons quelle contrainte doit être modélisée à quel niveau dans notre schéma d'optimisation. Ce modèle est basé sur la notion d'arcs alternatifs proposée par (Mascis et Pacciarelli, 2002).

Soit un problème général décrit comme suit : l'ensemble O des opérations doit être réalisé sur l'ensemble R des ressources. Le problème contient les contraintes suivantes :

- C_O est l'ensemble des contraintes sur l'ordre des opérations. Ces contraintes peuvent être des contraintes de précédence ou de time lags, ...
- C_C est l'ensemble des contraintes de conjonction, elles s'expriment sous la forme $t_i + c \leq t_j$ où i et j sont déterminées indépendamment de la solution envisagée.
- C_D est l'ensemble des contraintes de disjonction : ces contraintes indiquent qu'une et une seule des deux contraintes suivantes sont vraies $t_i + c \leq t_j$ et $t_j + d \leq t_i$ où i et j sont dépendent de la solution envisagée.

- C_A est l'ensemble des contraintes alternatives : ces contraintes indiquent qu'une et une seule des deux contraintes suivantes sont vraies $t_i + c \leq t_j$ et $t_k + d \leq t_l$ où i, j, k, l dépendent de la solution envisagée.

Les ensembles C_D et C_A sont des contraintes disjonctives, alors que les ensembles C_O et C_C sont des contraintes conjonctives.

À partir de ces quatre paquets de contraintes, on construit le graphe conjonctif - disjonctif. La première étape consiste à construire un graphe conjonctif - disjonctif non orienté. Celui-ci contient un nœud par opération ainsi que deux nœuds modélisant deux opérations fictives : la source réalisée avant toutes les autres opérations, et le puits réalisé après toutes les autres opérations. Dans ce graphe, on insère des arcs entre les opérations. Ces arcs modélisent les contraintes de C_C : pour chaque contrainte du type $t_i + c \leq t_j$ entre les dates de début (t_i est la date de début de l'opération i), on insère un arc de longueur c allant de l'opération i à l'opération j. Les opérations concernées par ces inégalités ne doivent pas dépendre de l'ordre des opérations mais être valide dans l'absolu quelle que soit la solution envisagée. Ce type d'arcs est représenté, par exemple, par les contraintes technologiques et plus généralement les contraintes de précédences et les contraintes de time lags.

De plus, le graphe conjonctif - disjonctif non orienté contient des arêtes modélisant les contraintes de C_D. Ces contraintes expriment généralement une disjonction entre deux opérations consommant une même ressource. Il y a donc une arête entre toute paire d'opérations consommant la même ressource, ainsi une clique relie d'une arête toute paire d'opérations utilisant une ressource commune.

À terme, une arête du graphe conjonctif - disjonctif non orienté est remplacée par un arc dès que l'ordre des opérations aux extrémités de l'arête est connu. Pour orienter toutes les arêtes, on utilise un ordre des opérations. Cet ordre définit un ordre total sur chaque ressource : soit une ressource $i \in R$, l'ordre des opérations ordonne entre autres toutes les opérations utilisant i. Il y a donc autant d'ordres que de ressource. Si ces ordres ne sont pas cohérents, ils ne correspondent pas à une solution et on les appelle **"ordre non valide"**. Par exemple, un ordre non valide indiquerait que deux jobs utilisent une ressource dans un certain ordre et utilisent une autre ressource dans l'ordre inverse.

De plus, même si un ordre est valide, il peut violer les contraintes C_O. On appelle **ordre réalisable** tout ordre valide qui vérifie les contraintes sur l'ordre. Comme nous le montrons dans la suite, les ordres non réalisables correspondent aux graphes contenant des cycles absorbants. Un tel ordre est donc contradictoire avec les contraintes formulées, il ne peut mener à une solution.

À l'aide d'un ordre réalisable, on peut orienter le graphe conjonctif - disjonctif pour obtenir un graphe conjonctif. Toutes les arêtes du graphe conjonctif - disjonctif sont supprimées pour être remplacées par des arcs. Ainsi, une arête entre l'opération i et l'opération j pour la ressource r est remplacée par un arc de longueur p_i allant de l'opération i à l'opération j si l'opération i est la plus tôt dans l'ordre des opérations sur r.

Enfin, tous les arcs de C_A peuvent être insérés dans le graphe. Suivant l'ordre des opérations, une des deux inégalités est retenue pour chaque contrainte alternative.

Une fois construit, le graphe conjonctif obtenu contient uniquement des arcs. Les contraintes sur l'ordre ne sont pas exprimées directement. Par contre, les contraintes de disjonction sont quant à elles exprimées par des arcs et elles respectent ces contraintes. Chaque arc modélise une inégalité du type $t_i + c \leq t_j$. On évalue alors le graphe par un algorithme de plus long chemin. Ainsi, les marques des nœuds sont telles que les contraintes sont vérifiées. En effet, s'il y a un arc de i vers j de longueur c, alors tout chemin allant du nœud i vers le nœud j est au moins de longueur c. Ainsi, les plus longs chemins respectent toutes les contraintes. Les marques du graphe forment donc un ordonnancement.

4.2 Application au problème de transport

Nous proposons d'utiliser la démarche de la section précédente pour construire un modèle de graphe conjonctif - disjonctif pour le problème de jobshop avec transport et contraintes additionnelles.

Les dix contraintes du problème sont prises en compte dans le schéma d'optimisation présenté ci-dessus. Les contraintes C(1), C(2), C(5), C(6), C(9) et C(10) sont prises en compte sur l'ordre des opérations C_O. Les contraintes C(1) et C(2) sont prises en compte dans le graphe conjonctif - disjonctif sous forme d'arcs (C_C). Les contraintes C(3) et C(4) sont prises en compte dans le graphe conjonctif - disjonctif sous forme d'arêtes (C_D). Et pour finir, les contraintes C(7) et C(8) sont prises en compte dans le graphe conjonctif sous forme d'arcs alternatifs (C_A).

Contrainte	**Paquet de contrainte**
C(1)	C_C , C_O
C(2)	C_C , C_O
C(3)	C_D
C(4)	C_D
C(5)	C_O
C(6)	C_O
C(7)	C_A
C(8)	C_A
C(9)	C_O
C(10)	C_O

Tableau 4-12. Modélisation du problème de transport à l'aide du schéma d'optimisation

4.2.1 Ordre des opérations

Dans ce problème, les ressources sont les machines et le transporteur. Il faut donc à priori ordonner les opérations sur chacune de ces ressources. Mais, nous ne retiendrons que l'ordre des opérations transport car les ordres des opérations machines peuvent être déduits de l'ordre des opérations transport grâce à la contrainte C(5). Ainsi, à partir du seul ordre des opérations transport, on connaît l'ordre de traitement de toutes les opérations sur toutes les ressources.

Ainsi, l'ordre des opérations est formé de l'ordre des opérations transport soumis aux contraintes C(1), C(2), C(6), C(9) et C(10).

4.2.2 Graphe conjonctif - disjonctif non orienté

On note *G= (V, A, E)* le graphe conjonctif - disjonctif non orienté, 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 machine ou transport, on crée un nœud dans *V*. Pour faciliter l'explication, on confondra dans la suite une opération et le nœud qui la représente. On crée de plus deux nœuds *0* et *n+1* qui modélisent deux opérations fictives. L'opération *0* est une opération fictive réalisée avant toutes les autres opérations, on dit que ce nœud est la source du graphe. L'opération *n+1* est une opération fictive réalisée après toutes les autres opérations, on dit que ce nœud est le puits du graphe. On distingue graphiquement les opérations réalisées sur les machines (symbolisées par un cercle) et les opérations réalisées par le transporteur (symbolisées par un carré). Chaque nœud porte une marque qui est la date de début de l'opération correspondante.

- On crée un arc entre toute opération machine *i* et l'opération transport suivante (opération *ST(i)*). La longueur de cet arc est la durée de l'opération machine *i : pi*. Cet arc modélise la contrainte **C(1)** avec anticipation. La contrainte **C(1)** sans anticipation ne peut être traitée que dans le graphe conjonctif - disjonctif orienté.
- On crée un arc entre toute opération transport *i* et l'opération machine suivante *SM(i)*. La longueur de l'arc est la durée de l'opération transport *pi*. Cet arc modélise la contrainte **C(2)**.

- On crée une arête entre deux opérations machines *i* et *j* réalisées sur la même machine. Cet arc modélise la contrainte **C(3)**.
- On crée une arête entre deux opérations transport *i* et *j*. Cet arc modélise la contrainte **C(4)**.

4.2.3 Exemple de graphe non orienté

Considérons l'instance suivante de jobshop avec transport (les temps de transport inclus les temps de chargement / déchargement) :

Job 1 :	T1(8)	O1(1,8)	T2(8)	O2(2,16)	T3(12)
Job 2 :	T4(8)	O3(1,20)	T5(10)	O4(3,31)	T6(10)
Job 3 :	T7(12)	O5(3,12)	T8(8)	O6(1,8)	T9(14)

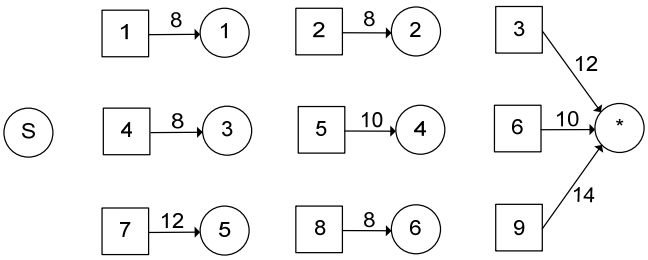

Figure 4-24. Arcs C(1) avec anticipation

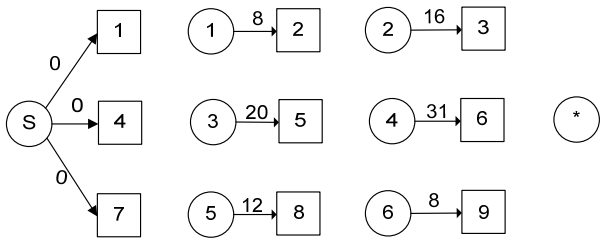

Figure 4-25. Arêtes C(3)

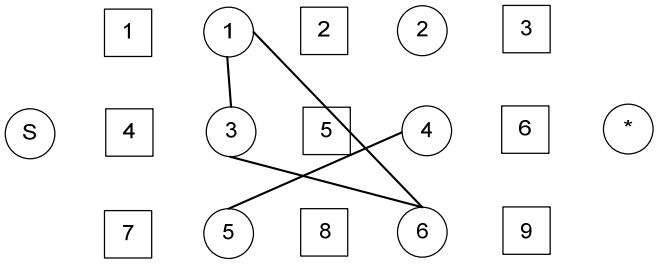

Figure 4-26. Arcs C(2)

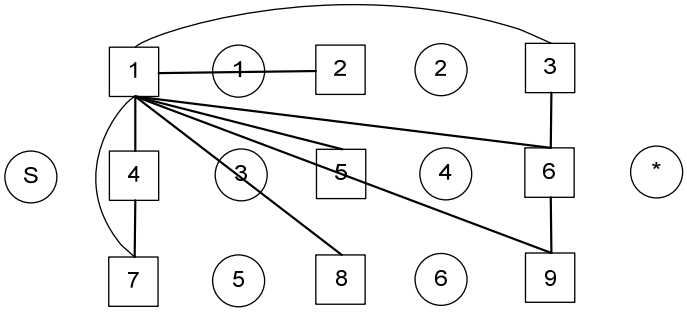

Figure 4-27. Arêtes C(4) partant de T1

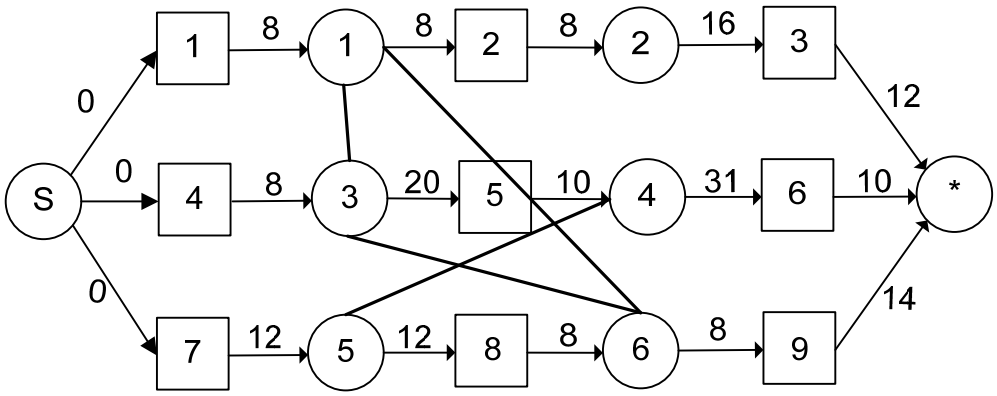

Figure 4-28. Graphes non orienté (sans les arêtes de C(4))

4.2.4 Définition d'un graphe conjonctif - disjonctif orienté

Le graphe conjonctif - disjonctif non orienté étant construit, on peut maintenant construire le graphe conjonctif à partir d'un ordre des opérations. Soit *O* un ordre des opérations (*O* définit un ordre les opérations sur le transporteur et sur chaque machine). On appelle *G(O)* le graphe conjonctif - disjonctif orienté associé. Le graphe *G(O)* est formé de tous les arcs du graphe *G*, toutes les arêtes étant supprimées pour être remplacées par les arcs suivants :

- On crée un arc entre toute opération machine *i* et l'opération transport suivante (opération *ST(i)*). La longueur de cet arc est la durée de l'opération machine *i* augmentée de la durée du déplacement à vide réalisé avant le déplacement du chariot *pi+tv(PT(ST(i)),ST(i))*. Cet arc modélise la contrainte **C(1)** sans anticipation.
- Étant données deux opérations transport consécutives *i* et *ST(i)* dans l'ordre *O*, on insère un arc entre ces deux opérations de *i* vers *ST(i)*, de longueur *pi+tv(i,j)* (i.e. temps de transport en charge *pi* augmenté de la durée du déplacement à vide *tv(i, ST(i))*). Cet arc modélise la contrainte **C(3)**.
- Soit une opération transport *i* et l'opération *j=STM(i)*. Ces deux opérations sont réalisées dans l'ordre *i* puis *j*. D'après la contrainte C(5), les opérations qui leur succèdent (*SM(i)* et *SM(j)*) sont réalisées dans le même ordre. On met donc un arc de *SM(i)* vers *SM(j)* dont la longueur est la durée de l'opération *SM(i)* (*pSM(i)*). Cet arc modélise la contrainte **C(4)**.
- Soit une opération machine *i* sur une machine *m0* et soit l'opération *j=STMk(PT(i))* où $k = e_{ss_{m_0}}$. On insère un arc allant du nœud *i* au nœud *j*, et de longueur nulle. Cet arc modélise la contrainte **C(7)**.
- Soit une opération transport *i* amenant un job vers la machine *m0* et soit l'opération transport *j* telle que $a(j) = p(i) + s_{m_0} + 1$. On insère alors un arc du nœud *i* vers le nœud *SM(j)* de longueur ε_d. Cet arc modélise la contrainte **C(8)**.

4.2.5 Exemple de graphe conjonctif - disjonctif orienté

Pour illustrer la construction du graphe conjonctif - disjonctif orienté, l'exemple précédent est complété par les données suivantes : le temps de déchargement est ε_d=1 et la capacité des stocks est *em=sm=1*, $\forall m$. La durée des transports à vide est donnée dans le tableau 4-13.

Pour construire un graphe orienté, on doit disposer d'un ordre des opérations. On construit le graphe conjonctif de la figure 4-33 est obtenu avec l'ordre suivant : *T1, T4, T2, T5, T7, T8, T9, T6, T3*. Cet ordre permet de connaître l'ordre des opérations machine, cet ordre est détaillé dans le tableau 4-14.

	L	**M1**	**M2**	**M3**	**M4**	**U**
L	0	6	8	10	12	0
M1	12	0	6	8	10	12
M2	10	6	0	6	8	10
M3	8	8	6	0	6	8
M4	6	10	8	6	0	6
U	0	6	8	10	12	0

Tableau 4-13. Temps de déplacement à vide

Machine	**Opérations transports**	**Opérations machine**
M1	0,3	0,2
M2	1	1
M3	4,6	3,4
M4	7	5

Tableau 4-14. Ordre sur les machines

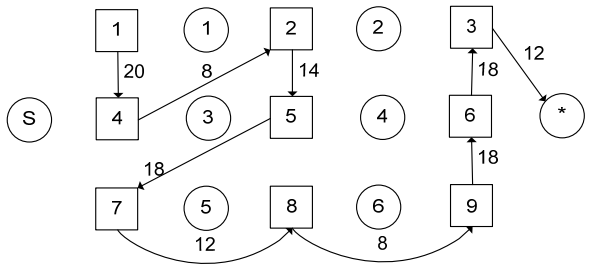

Figure 4-29. Arcs C(3)

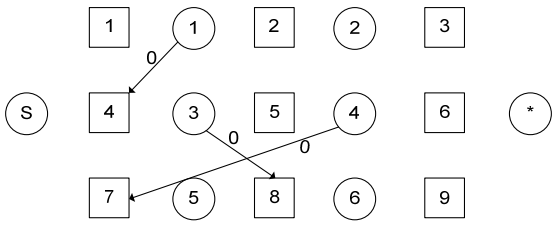

Figure 4-30. Arcs C(7)

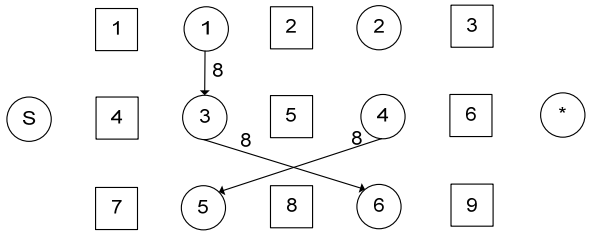

Figure 4-31. Arcs C(4)

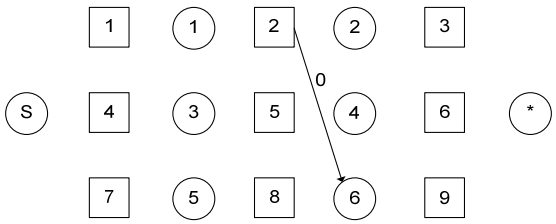

Figure 4-32. Arcs C(8)

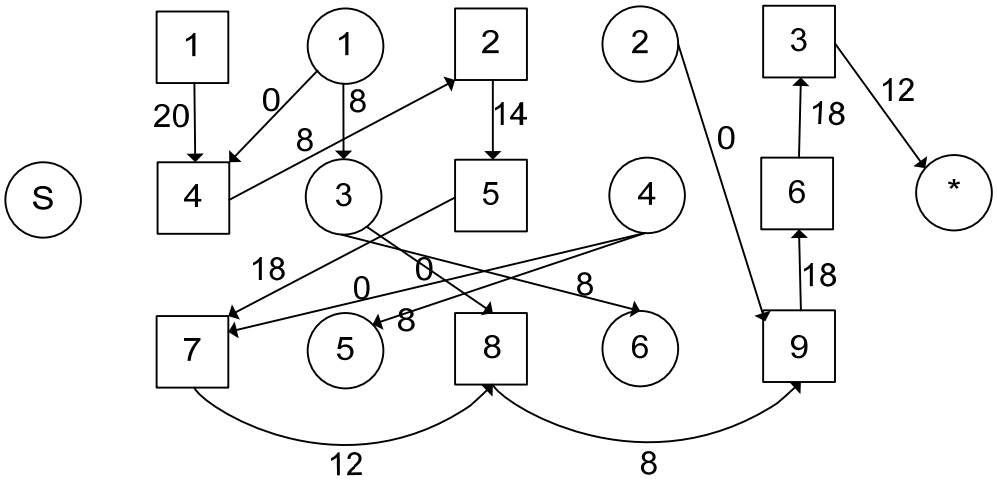

Figure 4-33. Exemple de graphe conjonctif - disjonctif orienté évalué

Le graphe conjonctif ainsi construit peut être utilisé pour calculer l'ordonnancement semi-actif correspondant à l'ordre des opérations. Tout algorithme de plus long chemin peut convenir s'il permet d'évaluer des graphes contenant des cycles, des arcs de longueur négative et éventuellement des cycles absorbants. Parmi ces algorithmes, on peut distinguer ceux qui s'arrêtent dès qu'un cycle absorbant a été détecté, avec les algorithmes qui sont capables de trouver tous les plus longs chemins élémentaires dans un graphe contenant des cycles absorbants. Ces derniers algorithmes sont largement moins performants dans la mesure où le problème qu'il résolve est NP-difficile. On préfère donc utiliser les algorithmes s'arrêtant dès qu'un cycle absorbant a été rencontré car ces algorithmes sont polynomiaux.

4.2.6 Algorithme dédié de plus long chemin

Pour un ordre des opérations donné, l'évaluation du graphe conjonctif peut être réalisée à l'aide de n'importe quel algorithme de plus long chemin supportant des cycles non absorbant et des arcs de

longueur positive ou nulle. Pourtant, . Dans le cadre d'une optimisation, le nombre de vecteurs par répétition à envisager est très élevé et le temps de calcul mis pour une évaluation est un élément critique. Ainsi, nous mettons en évidence des propriétés du graphe pour développer un algorithme de plus long chemin efficace dédié au plus long chemin.

Soit *T* un ordre réalisable d'opérations transport, on note *P(T)* le graphe partiel ne contenant pas les arcs **C(7)** et **C(8)**. Dans ce paragraphe, nous montrons que le graphe *P(T)* est acyclique, pour cela on montre que les nœuds peuvent être triés topologiquement.

Définition : Soit *T* un ordre d'opérations transport, on peut noter sans perte de généralité *T =1, 2, …, t.* On note *N(T)* l'ordre obtenu en remplaçant une opération *i* dans *T* par le couple (*i, SM(i)*) si l'opération *SM(i)* existe. Pour simplifier l'écriture, on appelle opération de rang $2k$ l'opération transport k, de même on appelle opération de rang $2k+1$ l'opération machine *SM(k)* si elle existe.

Lemme 1 : Soit *T* un ordre des opérations transport, l'ordre *N(T)* est un ordre topologique des nœuds du graphe *P(T).*

Preuve : Pour prouver que *N(T)* est un ordre topologique, nous montrons que pour toute paire d'opérations *i* vers *j*, il n'existe pas de chemin de *j* vers *i*.

Soit *i* une opération transport dans *N(T).* Soit *Hn* la propriété suivante : " il n'existe pas de chemin vers le nœud *i* qui parte des nœuds de rang *t-2n* et *t-2n+1* dans l'ordre *N(T)* ". Par récurrence finie sur *n* on peut montrer qu'il n'existe pas de chemin de *i* vers toute autre opération *j*.

L'hypothèse *H1* est trivialement vérifiée : Les opérations de rang *t-1* et de rang *t* sont les deux dernières opérations dans l'ordre *N(T).* L'opération de rang *t-1* est obligatoirement la dernière opération de sa gamme, il n'y a donc pas d'opération de rang *t*. Par construction de *P(T)*, le seul arc issu de l'opération de rang *t-1* va vers le puits. *H1* est vérifiée.

Montrons *H1, H2, … Hn⇒Hn+1*. Nous devons montrer qu'il n'existe pas de chemin des opérations de rang *t-2n* et de rang *t-2n+1* vers l'opération *i*. Par construction, l'opération suivante dans un éventuel chemin est une des trois opérations suivantes *ST(t-2n)*, *ST(t-2n+1)* ou *SM(t-2n+1)*. Nous allons montrer que ces trois opérations font partie des *2n* dernières opérations de l'ordre *N*.

Par définition de *ST*, l'opération *ST(2n-2)* est après l'opération *2n-2* dans l'ordre *T*. L'opération *ST(2n-2)* fait donc partie des *2n* dernières opérations de *N*.

Les opérations *ST(t-2n+1)* et *t-2n* sont deux opérations transport consécutives du même job. Puisque l'on envisage que des ordres d'opérations transport respectant cette contrainte, il est clair que l'opération *ST(t-2n+1)* est après l'opération t-*2n* dans l'ordre *N(T).*

L'opération *SM(t-2n+1)* est précédée d'une opération transport notée $k=PT(SM(t-2n+1))$. Par construction du graphe, l'ordre des opérations machine est égal à l'ordre des opérations transportant les jobs sur cette machine. Puisque l'opération *SM(t-2n+1)* est après l'opération *t-2n+1* sur la machine, les opérations transport les ayant amenés sur la machine sont dans le même ordre : l'opération k est donc après l'opération *t-2n* dans l'ordre *T*. L'opération *SM(t-2n)* est donc de rang supérieur dans l'ordre *N*.

Tout chemin qui part des opérations *t-2n* et *t-2n+1* passe par des opérations de rang supérieur dans l'ordre *N(T).* Or d'après la récurrence, il n'est pas possible de trouver un chemin passant par ces opérations pour aller vers *i*. Tout chemin partant des opérations *t-2n* et *t-2n+1* ne peut donc être complété en un chemin allant vers *i*. CQFD

Le lemme précédent indique que tout cycle emprunte un arc de type **C(7)** ou/et **C(8)**. Nous montrons dans de deuxième lemme que le graphe conjonctif sans les arcs **C(8)** est acyclique. On note *P'(T)* le graphe *P(T)* enrichi des arcs **C(7)**. Ce graphe modélise le problème $\wp$ avec stock de sortie de capacité infinie. Le théorème suivant formalise l'intuition suivante : " pour tout ordre π-réalisable, la capacité limitée des stocks d'entrée ne peut rendre irréalisable une solution, tout éventuel problème de capacité est résolu lorsque les opérations dans le stock d'entrée ont été traitées et sont passées dans le stock de sortie ".

Lemme 2 : Soit *T* un ordre des opérations transport. *P'(T)* est un graphe acyclique.

Preuve : Supposons par l'absurde que le graphe *P'(T)* contienne un cycle *C*. Puisque le graphe *P(T)* est acyclique, le cycle *C* contient un arc **C(7)** noté i→j. On peut donc noter le cycle *C= (j, u1, u2, ..., ur, i, j)*. Par construction, il n'y a pas d'arc possible de *l* vers *u1*, l'opération *u1* ne peut être égale à l'opération *SMc(k)*. Il y a donc forcément au moins une opération transport parmi les opérations *u1, u2, ..., ur*. On note *un* la dernière opération transport dans le cycle *C*. L'opération *un+1* est une opération machine réalisée sur la même machine et avant l'opération *i*. La séquence d'opérations transport *PT(un+1), PT(un+2), ..., PT(ur), PT(k), PT(SM(k)), ... PT(SMc(k))* est un chemin de l'opération *un* vers l'opération *l* qui n'emprunte que des arcs du graphe *G(T)*. Or la partie du cycle *l,u1, u2, ..., un* est un chemin de l'opération *l* vers l'opération *un*. Ce chemin ne contient que des arcs du graphe *G(T)*, si tel n'est pas le cas, on applique itérativement la procédure ci-dessus jusqu'à obtenir un chemin ne contenant que des arcs du graphe *G(T)*. Finalement, on a construit un cycle *l,u1,u2, ...,un, PT(un+1),PT(un+2), ..., PT(ur),PT(k), PT(SM(k)) ... PT(SMc(k))=l* qui ne contient que des arcs de *G(T)*. Absurde. CQFD.

Grâce aux deux lemmes ci-dessus, le calcul du plus long chemin dans ce graphe peut être réalisé de manière très efficace. Soit un ordre des opérations transport *T* pour lequel on veut calculer les plus longs chemins dans le graphe *G(T)*. On parcourt les opérations transport dans l'ordre *T*, et pour chaque opération, l'on met à jour les marques en empruntant les arcs **C4, C2, C3, C1, C7** et **C8** dans cet ordre. D'après les lemmes 1 et 2, cet algorithme évalue correctement le plus long chemin dans le graphe *P'(T)*. Pour prendre en compte les arcs **C8**, il faut donc potentiellement revenir en arrière dans l'évaluation du graphe. C'est ce qu'on appelle un retour arrière dans l'évaluation du graphe. Après une mise à jour le long de l'arc **C8**, on reprend le calcul en repartant du nœud de plus faible rang. Puisque tout cycle passe par un arc **C8**, il suffit de détecter les cycles lors de la mise à jour ce cet arc.

Cet algorithme est particulièrement efficace et permet d'évaluer efficacement un ordre des opérations transport.

5 Propositions de représentations

Basé sur le graphe ci-dessus, nous proposons deux représentations. La première représentation code directement une solution comme un ordre des opérations transport. Cette représentation comporte de nombreuses solutions irréalisables. C'est pourquoi nous proposons une seconde représentation capable de transformer l'ordre des opérations pour respecter un maximum de contraintes sur l'ordre des opérations.

5.1 Représentation 1 : ordre des opérations transport

Cette représentation consiste à stocker un ordre des opérations transport.

Exemple de solutions

1	4	2	5	7	8	9	6	3

Tableau 4-15. Exemple de représentation par ordre des opérations transport

Ensemble des solutions stockées

Les solutions stockées sont toutes les permutations des opérations transport.

Ensemble des solutions codées

Les solutions codées sont toutes les solutions semi-actives.

Codage/Décodage

Le codage de cette représentation consiste en la construction du graphe conjonctif présenté dans la section ci-dessus.

Propriétés

- Sur représentation : oui, car toutes les permutations d'ordres d'opération ne respectent pas les contraintes sur les ordres et donc certaines permutations ne correspondent pas à des solutions.
- Multi représentation : non, car chaque ordre correspond à un unique ordonnancement semi-actif
- Heuristique : non, car cette représentation permet de coder tous les ordonnancements semi-actifs et donc l'ordonnancement optimal.

5.2 Représentation 2 : réparation

La représentation avec réparation est identique à la représentation par ordre des opérations transport, mais elle répare les ordres qui ne respectent pas les contraintes.

Exemple de solutions

1	4	2	5	7	8	9	6	3

Tableau 4-16. Exemple de représentation par ordre des opérations transport

Ensemble des solutions stockées

Les solutions stockées sont toutes les permutations des ordres d'opérations.

Ensemble des solutions codées

Les solutions codées sont tous les ordonnancements semi-actifs.

Codage/Décodage

L'algorithme de codage est le même que la représentation par ordre d'opérations. Pourtant, quand une contrainte PAPS est violée, cet algorithme est capable de réparer cet ordre.

L'algorithme de réparation transforme un ordre d'opérations transport T en un ordre d'opérations réalisables $T2$. Pour ce faire, on parcourt les opérations dans l'ordre T, chaque opération est acceptée ou refusée. Une opération acceptée est insérée à la fin de l'ordre $T2$, une opération refusée est conservée dans T et sera insérée plus tard. À chaque opération acceptée, on envisage de nouveau d'insérer toutes les opérations conservées.

Une opération est acceptée si elle vérifie toutes les contraintes. Ainsi, pour chaque opération transport i, on vérifie les conditions suivantes :

- si i est la première opération transport, on vérifie que le nombre de jobs à insérer permet l'introduction d'un nouveau job dans le système.
- dans le stock de sortie de la machine de départ de i, le premier job disponible est le job transporté.
- si l'opération i n'est pas la prochaine opération transport pour ce job, on la remplace par la prochaine opération transport du job.

Ainsi, grâce à cet algorithme de réparation, les contraintes suivantes sont prises en compte : C(1), C(2), C(6) et C(10). Seule la contrainte C(9) n'est pas réparée.

Propriétés

- Sur représentation : oui, car certains ordres ne respectent pas la contrainte C(9) et sont représentés.

- Multi représentation : oui, car une permutation et une permutation réparée représentent le même ordonnancement.
- Heuristique : non, car cette représentation permet de coder tous les ordonnancements semi-actifs et donc l'ordonnancement optimal.

6 Proposition d'un algorithme d'optimisation

Nous proposons un algorithme génétique utilisant la représentation 1 pour le problème de jobshop avec transport et contraintes additionnelles.

6.1 Chromosomes

Un chromosome de l'algorithme génétique est un ordre des opérations transport dans lequel une opération est remplacée par son numéro de job. Ainsi, quand dans cet ordre apparaît pour la première fois un numéro de job, cela correspond à la première opération transport de ce job, ... Cette représentation est adaptée de celle proposée par (Bierwirth, 1995) pour le problème de jobshop. Les solutions que l'on peut représenter par cet ordre respectent donc tous les contraintes C(1) et C(2) sur l'ordre des opérations. Le décodage de ce chromosome est réalisé par la représentation 1.

Un chromosome C est un vecteur par répétition représentant l'ordre des opérations transport. Le fitness du chromosome se compose de :

- f_1 : le nombre d'opérations transport non ordonnancées;
- f_2 : le nombre maximal de jobs dans le système;
- f_3 : le nombre de violations de la contrainte PAPS au niveau des stocks de sortie;
- f_4 : le makespan.

Si un chromosome C mène à un graphe contenant un cycle absorbant, alors la solution proposée est irréalisable et f_1 donne le nombre d'opérations transport qui ne peuvent être ordonnancées. Si le chromosome est réalisable alors $f_1 = 0$. Le fitness du chromosome est calculé selon la formule suivante :

$f = \omega_1.f_1 + \omega_2.Max(f_2 - N;0) + \omega_3.f_3 + f_4'$ où $f_4' = 0$ si $f_1 \neq 0$ et $f_4' = f_4$ dans le cas contraire. Un croisement de type GOX à deux points de coupure (Bierwirth, 1995) est appliqué à deux parents. Le premier parent est choisi uniformément et le deuxième aléatoirement selon une loi biaisée favorisant les individus de faible fitness.

6.2 Détection de double

Un double est un chromosome représentant la même solution qu'un autre chromosome de la population. La présence de doubles conduit à un appauvrissement du matériel génétique et nuit à la convergence de l'algorithme génétique. Ainsi, nous cherchons à diminuer la présence de double dans la population. Pour cela, on introduit la notion de signature qui décrit par une valeur scalaire la solution. Nous choisissons $Sign(C) = \sum_{i \in T} t_i^2$ pour signature d'un chromosome C. Ainsi, si deux chromosomes aboutissent au même graphe conjonctif - disjonctif, leur signature est identique et ils sont détectés comme double. Il est rare que deux ordonnancements différents conduisent à la même signature.

La recherche de deux chromosomes identiques dans la population se résume alors à la recherche de deux chromosomes de même signature. Il est possible de doter la population d'un tableau nommé *DOUBLE* tel que *DOUBLE[Sign(C)]* est le nombre de chromosomes de signature *x*. La recherche de doubles peut se faire en *O(1)*.

6.3 Mutation

```
Paramètre d'entrée : C un chromosome
Paramètre de sortie : S la solution obtenues après recherche locale
Début
i :=1 ;cnt := 1 ;
Tant que (cnt < nm) faire
   cnt := cnt + 1;
   O1 := une opération choisie aléatoirement dans C;
   O2 := une opération choisie aléatoirement dans C;
   Cout_Initial := C.Fitness;
   Permuter O1 et O2 dans C;
   Si f(C) > Cout_Initial alors
         Permuter O1 et O2 dans C;     // Transition refusée
Fin Tant Que
Fin
```

Algorithme 4-1. Mutation

Avec une probabilité *Pm,* un chromosome obtenu après mutation subit une recherche locale. Cette recherche locale consiste à permuter deux valeurs différentes du chromosome et à accepter le nouveau chromosome si celui-ci est de fitness inférieur. La recherche locale s'arrête dès que le nombre maximal d'itérations *nm* est atteint ou bien dès que la borne inférieure du problème est atteinte.

6.4 Structure de la population et initialisation

La population est constituée de *n* chromosomes stockés dans un tableau *P* trié par ordre décroissant de fitness. À l'initialisation, la population se compose de solutions réalisables construites de manière heuristique. L'heuristique de construction retenue consiste à placer tous les jobs dans l'ordre lexicographique, puis toutes les opérations de chaque job suivant cet ordre. On obtient alors un chromosome de la forme : *1, 1,…, 1, 2, 2,…, 2, 3, 3…, n, n…,n.* Cette solution est insérée dans la population ainsi que la solution pour laquelle on numérote les jobs dans l'ordre inverse *n, n…,n, …, 3, 3,…, 3, 2, 2,…,2, 1, 1,…, 1.* On appelle ordre canonique un ordre tel que toutes les opérations transport d'un même job soient successives. Les deux ordres précédents sont deux ordres canoniques particuliers. Dix pourcent des chromosomes de la population initiale sont construits en choisissant de manière aléatoire un ordre canonique. Les autres chromosomes de la population sont générés de manière aléatoire.

À chaque ajout d'un chromosome dans la population, on vérifie que le chromosome n'a pas un double. Si tel est le cas, l'ajout est un échec et l'on incrémente un compteur. La génération s'arrête lorsque *nc* chromosomes ont été générés ou après *2*nc* échecs, ce qui garantit le caractère fini de l'initialisation. Il est donc possible que la population ait moins de *nc* chromosomes. On préfère une population de taille inférieure, mais qui ne contient pas de doubles.

6.5 Restarts

Pour lutter contre l'appauvrissement de la population, des restarts périodiques sont effectués toutes les *np* itérations successives sans amélioration de la meilleure solution. Nous proposons une procédure de "restart" qui consiste à introduire dans la population des chromosomes canoniques choisis aléatoirement. Notons que durant cette phase il est possible d'accepter la présence de plusieurs

chromosomes de même signe dans la population. Après un "restart" tous les chromosomes de la population sont faisables c'est-à-dire qu'ils sont tous des solutions du problème.

6.6 Schéma général de l'algorithme

Le schéma d'optimisation est basé sur un algorithme génétique incrémental et hybride. Comme le montre l'algorithme 4-2, à chaque itération un individu *C* est généré par un croisement, une évaluation puis une mutation éventuelle. La mutation consiste en une recherche locale dont l'objectif est d'améliorer l'individu. La signature de *C* est comparée à celles des individus membres de la population. Si la signature est unique, on insère *C* à la place de l'individu *k* choisi de manière aléatoire. Sinon, l'individu est considéré comme un double et son insertion est rejetée.

```
mni  : nombre maximal d'itération
np   : nombre maximal d'itérations improductives
nc   : nombre de chromosome dans la population
pm   : probabilité de recherche locale
LB   : borne inférieure du problème
nm   : nombre maximale d'itération de recherche locale
pr   : pourcentage de la population
Debut
Génération de la population initiale Pop
ni :=0;  //Numéro de l'itération courante
npi :=0; //Nombre d'itérations successives sans amiélioration
Répéter
   Choisir deux parents P1 et P2
   Appliquer le croisement GOX  : le fils est noté C
   Choisir un nombre entier uniformément dans [1,int(nc/2)]
   si (random < pm) alors
         Appliquer une recherche locale C
         Le chromosome après la recherche locale est noté S
         si S n'est pas un double alors
               C  := S;
         finsi
   finsi
   si C n'est pas un clone alors
         si f(C) < f(Pop(nc)) alors
               npi  := 0;
         sinon
               npi  := npi + 1;
         finsi;
         Pop[k]  := C; Retrier la population P;
   finsi;
   si (npi=np) alors
         Faire un restart et npi  := 0;
   finsi;
```

```
    ni  := ni + 1
Jusqu'à ce que (ni = mni) ou (f(Pop(nc)) = LB);
```

Algorithme 4-2. Algorithme génétique pour le jobshop avec transport et contraintes additionnelles

7 Expérimentations numériques

Les développements informatiques ont été effectués en Delphi 6 sous Windows XP et les tests réalisés sur un Pentium IV à 2.8 Ghz à partir d'une implémentation dédiée d'un algorithme de plus long chemin de type Bellman. Notre objectif est d'étudier la convergence de l'algorithme en mesurant :

- La stabilité des résultats obtenus par rapport à plusieurs exécutions de l'algorithme avec un ensemble de paramètres identiques ;
- La performance de l'algorithme en mettant en évidence l'écart par rapport à la solution optimale.

Les expérimentations sont réalisées en utilisant le jeu de paramètres suivant :

```
mni = 500 000   // nombre maximal d'itération de l'algorithme
np  = 10 000    // nombre maximal d'itérations improductives
nc  = 100       // nombre d'individus dans la population
pr  = 0.5       // probabilité de recherche locale
nm  = 50        // nombre d'itérations de la recherche locale
pr  = 99        // pourcentage de la population remplacée au moment des restarts
```

$\omega_1 = 1\ 000\ 000, \omega_2 = 100\ 000, \omega_3 = 1\ 000$

Le tableau 4-17 présente les résultats détaillés pour l'instance Jobset 1 $p_{ij} \times 1,\ t_{ij} \times 1$, alors que le tableau 4-18 présente les résultats détaillés pour l'instance Jobset 1 $p_{ij} \times 2,\ t_{ij} / 2$. Les deux tableaux montrent la robustesse de l'algorithme, car les réplications ont des résultats proches quelles que soient les réplications.

		Opt	C	*Dev%.*	I*	T*	T	I
n=3	N=2	182	182	0,00	1 830	0	0	1 830
		182	182	0,00	4 998	0	0	4 998
		182	182	0,00	1 768	0	0	1 768
		182	182	0,00	13 075	1	1	13 075
	N=3	178	178	0,00	22 999	3	3	22 999
		178	178	0,00	21 309	3	3	21 309
		178	178	0,00	22 891	3	3	22 891
		178	178	0,00	40 417	6	6	40 417
n=4	N=2	218	218	0,00	4 733	0	0	4 733
		218	218	0,00	166 786	19	19	166 786
		218	218	0,00	76 628	8	8	76 628
		218	218	0,00	238 727	30	30	238 727
	N=3	214	214	0,00	232 810	32	32	232 810
		214	214	0,00	122 682	15	15	122 682
		214	214	0,00	87 264	12	12	87 264
		214	214	0,00	54 646	7	7	54 646
	N=4	214	214	0,00	338 406	60	60	338 406
		214	214	0,00	245 662	43	43	245 662
		214	214	0,00	48 593	8	8	48 593
		214	214	0,00	86 911	14	14	86 911
n=5	N=2	260	263	1,15	305 809	40	69	500 000
		260	269	3,46	56 513	7	69	500 000
		260	263	1,15	108 344	14	69	500 000
		260	263	1,15	450 456	62	69	500 000
	N=3	248	252	1,61	76 261	10	71	500 000
		248	255	2,82	371 722	52	71	500 000
		248	252	1,61	134 698	19	72	500 000
		248	255	2,82	52 829	07	70	500 000
	N=4	248	250	0,81	142 485	19	71	500 000
		248	252	1,61	8 652	01	75	500 000
		248	252	1,61	469 143	69	74	500 000
		248	252	1,61	241 051	33	73	500 000
	N=5	/	252	/	184 148	26	75	500 000
		/	252	/	148 151	22	79	500 000
		/	252	/	285 176	41	76	500 000
		/	252	/	180 341	26	81	500 000
	Moy.			0,67	140 247	19.78	39.67	273 142

Tableau 4-17. Résultats de l'algorithme génétique pour le Jobset 1 $p_{ij} \times 1$, $t_{ij} \times 1$

Le tableau 4-19 donne les résultats moyens sur l'ensemble des instances. Ce tableau montre que l'algorithme génétique fournit des résultats à moins de 1% de la solution optimale et pour un temps de calcul moyen inférieur à 20s.

Ces résultats sont bien supérieurs aux résultats précédemment obtenus par le couplage d'un Branch and Bound avec un modèle de simulation. En effet, l'approche B&B permet d'obtenir un écart moyen de 9.19% alors que l'algorithme génétique fournit en moyenne des résultats à 0.31% de la solution optimale. Ceci est mis en évidence sur le tableau 4-20.

		Opt	C	*Dev%,*	I*	T*	T	I
n=3	N=2	199	199	0,00	2 004	0	0	2 004
		199	199	0,00	4 145	0	0	4 145
		199	199	0,00	3 698	0	0	3 698
		199	199	0,00	2 504	0	0	2 504
	N=3	148	148	0,00	1 095	0	0	1 095
		148	148	0,00	760	0	0	760
		148	148	0,00	531	0	0	531
		148	148	0,00	1 430	0	0	1 430
n=4	N=2	227	227	0,00	46 413	6	6	46 413
		227	227	0,00	208 679	26	26	208 679
		227	227	0,00	91 754	10	10	91 754
		227	227	0,00	248 971	30	30	248 971
	N=3	193	193	0,00	19 872	3	3	19 872
		193	195	1,04	52 294	7	73	500 000
		193	193	0,00	22 376	3	3	22 376
		193	195	1,04	48 187	6	70	500 000
	N=4	160	160	0,00	3 980	1	1	3 980
		160	160	0,00	2 733	0	0	2 733
		160	160	0,00	3 501	0	0	3 501
		160	160	0,00	3 355	0	0	3 355
n=5	N=2	272	272	0,00	46 113	6	6	46 113
		272	272	0,00	56 320	7	7	56 320
		272	272	0,00	269 464	35	35	269 464
		272	272	0,00	127 023	16	16	127 023
	N=3	214	214	0,00	57 543	8	8	57 543
		214	214	0,00	59 179	8	8	59 179
		214	214	0,00	185 710	25	25	185 710
		214	214	0,00	61 361	8	8	61 361
	N=4	178	178	0,00	175 290	26	26	175 290
		178	178	0,00	310 164	48	48	310 164
		178	178	0,00	191 519	30	30	191 519
		178	178	0,00	323 931	51	51	323 931
	N=5	169	169	0,00	2 865	0	0	2 865
		169	169	0,00	9 200	1	1	9 200
		169	169	0,00	2 885	1	1	2 885
		169	169	0,00	2 936	0	0	2 936
	Moy.			0,06	73 605	10.06	13.67	98 591

Tableau 4-18. Résultats de l'algorithme génétique pour le Jobset 1 $p_{ij} \times 2,\ t_{ij}/2$

Jobset 1	Dev%	I*	T*	T	I
$p_{ij} x1,\ t_{ij} x1$	0.67	140 247	19.78	39.67	273 142
$p_{ij} x2,\ t_{ij}/2$	0.06	73 605	10.06	13.67	98 591
$p_{ij} x3,\ t_{ij}/2$	0.51	568 18	7.97	20.34	140 480
$p_{ij}/2,\ t_{ij} x2$	0.30	1	6.00	30.28	222 222
$p_{ij}/2,\ t_{ij} x3$	0.00	1	6.00	0.00	1

Tableau 4-19. Résultats synthétiques de l'algorithme génétique

	B&B Dev %.	Algorithme génétique Dev %.	I*	T*	T	I
$p_{ij}x1,\ t_{ij}x1$	5.81	0.67	140 247	19.78	39.67	273 142
$p_{ij}x2,\ t_{ij}/2$	1.70	0.06	73 605	10.06	13.67	98 591
$p_{ij}x3,\ t_{ij}/2$	3.30	0.51	568 18	7.97	20.34	140 480
$p_{ij}/2,\ t_{ij}x2$	14.31	0.33	1	6.00	30.28	222 222
$p_{ij}/2,\ t_{ij}x3$	20.85	0.00	1	6.00	0.00	1
Moy.	9.19	0.31	53 464	10	21	146 887

Tableau 4-20. Comparaison de l'algorithme génétique et de la procédure de séparation / évaluation

En conclusion, nous avons proposé une étude numérique sur 25 instances (issues des propositions de Bilge et Ulusoy) montre que l'algorithme génétique hybride que nous proposons est une méthode efficace de résolution. Cette méthode est plus performante que l'algorithme de B&B / Simulation introduit en 2005 par (Lacomme *et al.*, 2005). Les évaluations numériques ont été réalisées en comparant les résultats aux solutions optimales obtenues en résolvant un modèle linéaire du problème (Caumond *et al.*, 2005a). Ces évaluations numériques ne concernent que des instances de taille restreinte avec 111 opérations à ordonnancer. Les résultats montrent l'efficacité de la méthode proposée et ceci dans des temps de calcul très court par rapport à une résolution du problème par le modèle linéaire (celui-ci nécessite plusieurs de calcul).

8 Conclusion

Dans ce chapitre, nous nous sommes intéressés au problème de jobshop avec transport et contraintes additionnelles. Outre la modélisation du transporteur comme une ressource, ce problème prend en compte la capacité limitée des stocks d'entrée et de sortie, la non anticipation des déplacements du transporteur, le nombre limité de jobs dans le système, le transport à vide du transporteur et le blocage de l'unité de traitement.

Pour ce problème, nous avons proposé une formalisation mathématique, un programme linéaire en nombres entiers, un modèle de graphe conjonctif-disjonctif et un algorithme génétique. La formalisation mathématique est utile pour décrire précisément le problème. Une telle description est utile pour tout développement ultérieur, car tout modèle est complet s'il prend les contraintes décrites dans la formalisation. À partir de cette formalisation, nous avons dérivé un programme linéaire en nombres entiers qui permet de résoudre jusqu'à 5 jobs (111 opérations). Pour les instances de taille supérieure, et permettre la construction d'heuristiques, nous avons proposé un modèle de graphe conjonctif - disjonctif. Ce modèle étend le modèle classique pour le problème de jobshop en prenant en compte des contraintes supplémentaires. Le modèle de graphe est à l'origine de deux représentations qui sont utilisées pour construire l'algorithme génétique.

L'étude de ce problème a un double intérêt : l'ordonnancement des systèmes flexibles de production (SFP), et la prise en compte des nombreuses contraintes.

Chapitre 5 Cadriciel orienté objet pour l'optimisation

Ce chapitre présente la BCOO, Bibliothèque et Cadriciel orienté Objet pour l'Optimisation, qui aide à la conception et au développement d'un logiciel d'optimisation.

Sommaire

Table des figures

Table des tableaux

Table des algorithmes

1 Introduction

Dans ce chapitre, nous décrivons la BCOO (Bibliothèque et Cadriciel orienté Objet pour l'Optimisation). La BCOO a pour but d'aider au développement rapide d'applications d'optimisation lors des phases de conception et d'implantation d'un logiciel d'optimisation. Les raisons qui nous ont conduit à la création d'un tel cadriciel sont doubles.

Tout d'abord, notre démarche de modélisation présentée dans le chapitre 1[8], nous a conduit à mettre en œuvre de nombreux modèles d'optimisation. Lors de ces développements, nous avons constaté que de nombreuses parties de codes étaient communes et qu'il serait donc très intéressant de pouvoir le réutiliser. Mais la réutilisation de code n'est pas une chose facile, surtout en optimisation là où le temps d'exécution joue un rôle fondamental. Le cadriciel que nous proposons est une solution élégante et performante à la réutilisation de code pour les logiciels d'optimisation.

De plus, les projets industriels sur lesquels nous avons travaillé nous ont montré que, comme le préconise l'approche RAD (Rapid Application Development), il était utile de disposer rapidement d'un prototype, plus ou moins fonctionnel, de l'application à développer. Ce prototype est alors utilisable pour discuter avec les experts afin de déterminer et / ou affiner leurs besoins. Le cadriciel que nous proposons fournit tous les outils pour mettre en œuvre une approche RAD pour l'optimisation.

Dans les paragraphes suivants, nous décrivons tout d'abord la notion de cadriciel de manière générale (§ 2). Ensuite, avant de présenter notre cadriciel à proprement parler, nous décrivons quatre nouvelles notions (§3) que nous avons proposé et qui sont largement utilisées dans la suite par le cadriciel. La présentation du cadriciel se fait alors en deux parties : tout d'abord les spécifications générales et indépendantes de la plateforme sont présentées dans le paragraphe 4, puis nous présentons des exemples de sa mise en œuvre dans le paragraphe 5.

2 Cadriciels

Littéralement, un cadriciel est un CADRre logICIEL (Application Framework en anglais). Les cadriciels sont en quelque sorte des progiciels extrêmement souples et évolutifs. Ils ont été introduits en génie logiciel pour généraliser la notion de bibliothèques de classe en y ajoutant des notions architecturales.

Dans une bibliothèque de classes, on regroupe un ensemble de classes car elles possèdent une certaine unité. Par exemple, on regroupe dans une même bibliothèque toutes les classes nécessaires à la construction d'interfaces graphiques ou toutes les classes nécessaires à la communication avec une base de données, ou bien toutes les classes ayant attrait aux communications distantes (FTP, http, RPC, ...). Toute application nécessitant une ou plusieurs fonctions disponibles dans la bibliothèque peut appeler les fonctions ou utiliser les classes présentes dans la bibliothèque. Une bibliothèque de classes sert donc à éviter simplement la réécriture du code utilisé.

Les cadriciels portent en plus une notion architecturale : un cadriciel porte une architecture (i.e. une ossature) semi-finie généralement décrite par des classes abstraites ou des interfaces et qui décrit aussi la manière dont ces classes collaborent entre elles et les responsabilités de chacune. De plus, un cadriciel est souvent muni d'un ensemble de classes préprogrammées permettant d'attribuer des comportements par défaut de toutes les parties de l'architecture de manière à couvrir au maximum les

[8] chapitre 1, section 5, page 23

besoins des utilisateurs. Le but étant de proposer un ensemble de classes flexibles et extensibles pour minimiser la quantité de classes spécifiques à développer.

Les cadriciels sont donc génériques car ils ne sont pas créés pour une application précise mais pour un type d'applications. Toute application correspondant au type peut utiliser le cadriciel en complétant les éléments architecturaux proposés et / ou en réutilisant les classes préprogrammées disponibles. La figure 5-1 présente un exemple d'utilisation d'un cadriciel pour la résolution d'un problème donné. À partir de l'analyse du problème, on en déduit deux choses : la liste des classes préprogrammées que l'on peut réutiliser et la liste des éléments d'architecture à compléter.

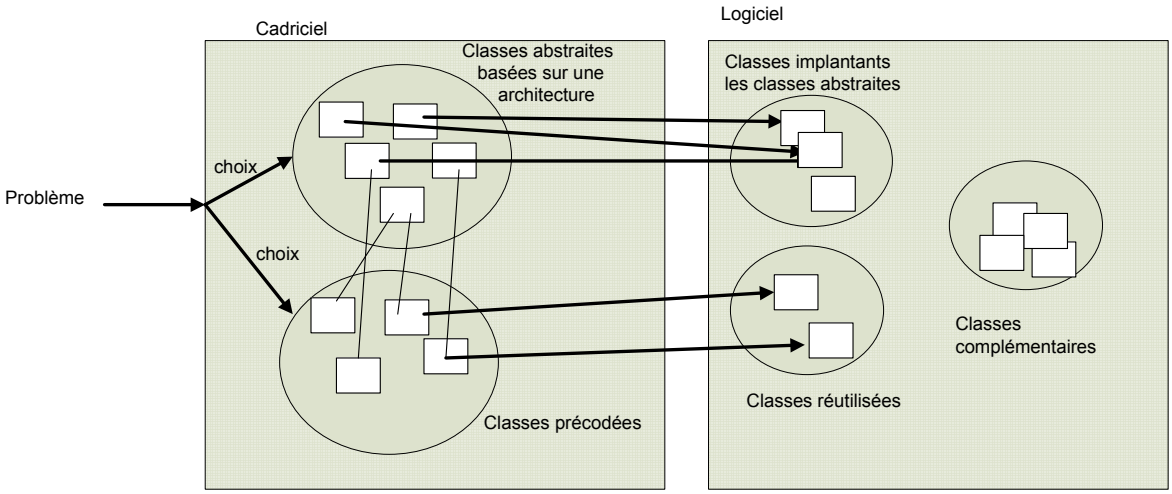

Figure 5-1. Un cadriciel en général

Les principaux avantages des cadriciels sont les suivants :

- Ils synthétisent des concepts communs à plusieurs applications. Ils fournissent ainsi un langage commun et des notions communes à tous les utilisateurs (concepteurs, développeurs de nouveaux composants du cadriciel ou développeurs d'applications), ce qui permet d'optimiser leur communication.
- Ils améliorent la productivité par leur architecture générique. Tout développement d'une nouvelle application est facilité car il nécessite simplement de compléter l'architecture, ce qui permet d'accélérer fortement la phase de conception.
- Ils permettent la réutilisation de code, car parmi les classes de l'architecture certaines peuvent être implantées une fois pour toutes et réutilisées par de nombreuses applications.
- Ils permettent d'implanter des comportements par défaut. Les classes ainsi obtenues augmentent la quantité de code réutilisé, mais ne limitent pas l'utilisateur qui prend la décision de garder ou non ce comportement par défaut.
- Ils permettent le prototypage rapide en acceptant l'architecture proposée et utilisant des comportements par défaut. On obtient un prototype qui est peut être moins efficace et / ou moins fonctionnel que l'application finale mais qui présente les principales fonctionnalités. Disposer tôt d'une maquette est un prérequis de l'approche RAD.
- Ils contiennent des "bonnes pratiques" dont les utilisateurs du cadriciel peuvent profiter. Ce sont principalement des éléments d'architecture, issus de l'expérience des développements antérieurs, et qui ont pour but d'éviter les pièges déjà rencontrés.
- Ils sont flexibles et extensibles, car ils sont conçus pour un ensemble d'applications.
- Ils centralisent les problèmes de maintenance, car mis à part les parties de programmes dédiées à une application spécifique, toutes les parties communes ne doivent être maintenues que pour le cadriciel lui-même.

Bien sûr, les cadriciels souffrent aussi de certains désavantages, qui sont la contrepartie des avancées architecturales :

- Ils contiennent une multitude de classes. Ils nécessitent donc un certain temps minimum pour leur apprentissage et sont donc difficiles d'accès à des développeurs non confirmés.
- Ils prévoient tous les cas et donc trop de cas. Un cadriciel peut traiter à la fois le cas général et les cas particuliers. Ainsi, un développeur n'ayant pas besoin des cas particuliers affronte une certaine complexité due à ces fonctionnalités.
- Un certain nombre de choix ont été réalisés dans le cadriciel (sinon, il ne guide en rien l'utilisateur). Ces choix peuvent s'avérer contraignant pour les développeurs utilisant le cadriciel. Ceci est un point délicat qui peut mener au rejet du cadriciel dans son ensemble. Si le cadriciel est de qualité, peu de problèmes risquent de subvenir dans la plupart des cas. Mais il existe toujours un risque pour qu'un cas particulier n'ait pas été envisagé et que le cadriciel devienne alors limitant.
- Lors de l'élaboration d'une application complexe, il est tout à fait possible et courant que plusieurs cadriciels soient nécessaires pour couvrir tous les domaines de l'application. Un cadriciel pour la partie système d'information, un pour la partie interface homme machine et un pour la partie optimisation par exemple. Les problèmes de compatibilité entre cadriciels ne sont pas simples et peuvent quelquefois poser des problèmes dans les phases en amont ou pire dans les phases en aval du développement.

Nom	Description
VCL	Construction d'interfaces graphiques sous Windows (Delphi/C++)
.NET	Construction d'interfaces graphiques sous Windows (nombreux langages disponibles)
Cocoa	Construction d'interfaces graphiques sous Mac (Objective C ou Java)
Zope	Serveur d'applications web orienté objet (Python)
SPIP	Gestion de contenu de site web (en PHP)

Tableau 5-1. Quelques exemples de cadriciels

Il ne faut pas confondre la notion de cadriciel et la notion de pattern. Les patterns sont une autre notion de génie logiciel relative à la structure d'une application. On distingue les patterns de conception et les patterns d'implantation, ces deux types de patterns ont en commun de proposer à leurs utilisateurs une architecture de quelques classes, connues et nommées par la communauté, pouvant être réutilisés dans de nombreuses applications dans un but précis et identifié. Un pattern est donc un élément d'architecture récurrent. La différence entre les patterns et les cadriciels est donc l'étendue de leur champ d'application : un pattern ne concerne que quelques classes et peut être utilisé dans n'importe quel type d'application alors qu'un cadriciel structure l'application complète et est dédié à un type d'applications.

3 État de l'art

Dans cette section, nous proposons un état de l'art des cadriciels et bibliothèques d'optimisation. Nous séparons ces contributions en trois parties : celles qui structurent l'évaluation d'une solution (et qui ne structure pas l'optimisation), celles qui structurent les méthodes d'optimisation (et qui ne structure pas l'évaluation) et celles qui structurent les deux.

3.1 Bibliothèques et cadriciels structurant l'évaluation d'une solution

Les bibliothèques suivantes se basent sur une représentation générique d'un problème. Ces bibliothèques se scindent donc en deux groupes, un groupe "évaluation de la solution" et un groupe "optimisation". L'évaluation de la solution se base sur une représentation générique d'un problème.

Cette représentation utilise par exemple, des contraintes linéaires, des contraintes de type Programmation Par Contraintes, ou des contraintes de type définition des invariants.

Grâce à cette représentation générique, l'utilisateur de la bibliothèque peut décrire tout nouveau problème. L'évaluation d'une solution est donc prise en charge par la bibliothèque. La partie optimisation se base sur cette représentation générique pour définir, modifier et améliorer une solution.

3.1.1 DejaVu

DÉJÀ VU (Dorn, 1999) est un cadriciel dont le but est d'aider au développement d'applications pour l'ordonnancement des systèmes de production industriels. DÉJÀ VU se base très fortement sur la notion de contraintes temporelles.

Ainsi, pour décrire un nouveau problème dans DÉJÀ VU, il faut décrire ses contraintes. Le cadriciel fournit pour cela un certain nombre de contraintes préprogrammées. L'utilisateur peut tout de même créer de nouveaux types de contraintes à l'aide des outils fournis. DÉJÀ VU est un cadriciel est donc ouvert car il permet d'écrire des contraintes définies par l'utilisateur. Pour mettre en œuvre ces mécanismes, le cadriciel utilise la notion objet de polymorphisme. Grâce au polymorphisme, les classes du cadriciel peuvent appeler une méthode d'un objet "contrainte" sans connaître l'objet qui va réellement implanter cette méthode. Une contrainte a donc une interface générale qui permet d'utiliser et de spécifier de nouvelles contraintes.

Une contrainte peut être vérifiée ou non. DÉJÀ VU propose d'affiner cette évaluation en définissant des niveaux intermédiaires de vérification. Ainsi, une valeur numérique permet de spécifier à quel degré une contrainte est vérifiée. Pour savoir si un ordonnancement est vérifié, il faut évaluer toutes les contraintes du problème. Pour chaque contrainte, on a donc une valeur numérique indiquant la réalisabilité de la contrainte. Pour évaluer la réalisabilité de l'ordonnancement, on agrège les valeurs numériques en les pondérant. Chaque contrainte se voit donc attribuer un poids indiquant son importance. Plus la contrainte a un poids élevé et plus il est important de la vérifier.

Du point de l'utilisateur du cadriciel, DÉJÀ VU est composé de deux parties principales : la première partie permet de créer de manière interactive un ordonnancement, la seconde met en œuvre des méthodes itératives d'amélioration. Ces deux parties utilisent le moteur de contraintes pour calculer une solution.

En résumé, DÉJÀ VU est un cadriciel basé sur la notion de contraintes temporelles. Grâce au polymorphisme, tout utilisateur peut définir ses propres contraintes. L'interface interactive et les algorithmes d'optimisation utilisent les contraintes temporelles pour définir et évaluer des solutions.

3.1.2 Localizer++

Localizer++ (Michel et van Hentenryck, 1997) est une bibliothèque extensible dédiée aux problèmes d'optimisation. Elle est formée de deux composants principaux : le composant déclaratif et le composant de recherche.

Le composant déclaratif est le noyau de l'architecture. Il utilise la notion d'invariants pour définir de manière déclarative les structures de données à maintenir. Par exemple, un invariant définit que les meilleurs voisins sont un sous ensemble de l'ensemble des voisins dont le coût est minimal. Pour autant, les invariants ne précisent pas la manière dont les structures de données sont mises à jour. C'est localizer++ qui propose une mise à jour incrémentale. C'est-à-dire qu'à chaque mise à jour d'une variable, toutes les contraintes utilisant cette variable répercutent l'information à leur manière.

Le composant de recherche utilise intensivement les composants déclaratifs pour mettre en œuvre les méthodes d'optimisation. Ainsi, localizer++ contient des procédures de haut niveau d'abstraction pour manipuler les composants déclaratifs. L'écriture d'une méthode d'optimisation s'en trouve alors très simplifiée.

3.1.3 SALSA

SALSA (Laburthe et Caseau, 1997) est un langage pour les algorithmes d'optimisation. Il est basé sur les concepts des problèmes de satisfaction de contraintes (CSP). SALSA généralise ces concepts pour permettre la mise en œuvre d'algorithmes hybrides, regroupant les techniques de CSP et celles des algorithmes de recherche. Pour cela, SALSA propose de séparer explicitement les deux parties classiques en CSP que sont la partie logique de la partie contrôle. SALSA est basé sur une représentation proche de localizer++.

3.2 Bibliothèques et cadriciels structurant les méthodes d'optimisation

Les bibliothèques présentées dans cette partie proposent de structurer les méthodes d'optimisation. Pour cela, chacune définit une architecture adaptée à une famille d'algorithmes d'optimisation pour laquelle elle est conçue.

3.2.1 INCOP

INCOP[9] (Neveu et Trombettoni, 2004) est une bibliothèque C++ extensible pour la mise en œuvre de méthodes d'optimisation approchées (appelées méthodes incomplètes dans l'article). Cette bibliothèque propose une architecture que l'utilisateur peut étendre en utilisant les mécanismes de polymorphisme. L'architecture proposée est basée sur six classes principales :

- La classe "Configuration" permet de spécifier le codage d'une solution,
- La classe "Move" permet de définir l'ensemble des solutions voisines,
- La classe "Neighborhood Search" permet de spécifier l'algorithme qui calcule les solutions voisines,
- La classe "IncompleteAlgorithm" permet de définir un nouvel algorithme approché,
- La classe "Metaheuristic" permet de définir une nouvelle métaheuristique,
- Et la classe "OpProblem" permet de définir un nouveau problème.

En particulier, INCOP propose une spécialisation de "OpProblem" en "CSProblem", ces problèmes permettent de modéliser les problèmes de satisfaction de contraintes. Les auteurs proposent donc aussi une spécialisation de Configuration, de Move et de Neighborhood Search adaptées à ce type de problème.

3.2.2 Hotframe

Hotframe (Fink et Voß, 2002) est un cadriciel dédié aux algorithmes d'optimisation. Il est écrit en langage C++, il a recours à la notion de template car elle est, selon les auteurs, très efficace. Hotframe structure les algorithmes de recherche, il propose donc de nombreuses classes et diagrammes de classes. Nous ne donnons pas plus de détails sur les objets proposés par hotframe car très peu d'objets sont généraux. En effet, la plupart des classes de Hotframe codent des algorithmes d'optimisation ou des morceaux d'algorithme d'optimisation.

Les trois principaux points forts de Hotframe sont :

- qu'il contient des implantations efficaces d'algorithmes classiques d'optimisation,
- qu'il est basé sur la notion de template qui permet d'allier généricité et efficacité
- que l'architecture proposée est ouverte.

[9] http://www-sop.inria.fr/coprin/neveu/incop/

Le projet Hotframe est sur le point d'intégrer un logiciel de construction de modèles d'optimisation. A l'aide de ce logiciel écrit en JAVA, un utilisateur pourra spécifier son modèle. La spécification repose sur une démarche incrémentale, à chaque étape l'interface demande à l'utilisateur de définir les propriétés des entités. L'étape suivante est construire en fonction des propriétés définies.

3.2.3 EO et ParadiseEO

EO et ParadiseEO (Keijzer, *et al.*, 2001) sont des projets de sourceforge[10]. Tous deux sont des bibliothèques de classes d'objets dédiées aux algorithmes évolutionnistes d'optimisation. ParadiseEO est la version parallèle de EO, nous concentrons donc notre description sur la version générale (EO).

Un des buts de EO est de fournir une implantation des algorithmes d'optimisation indépendante du codage de la solution.

3.3 Bibliothèques structurant à la fois l'optimisation et l'évaluation

3.3.1 HeuristicLab

HeuristicLab (Wagner et Affenzeller, 2004) est, tout comme la BCOO, un cadriciel d'optimisation indépendant de tout paradigme. Indépendant de tout paradigme signifie que le cadriciel peut être utilisé pour tout problème d'optimisation et quel que soit l'algorithme utilisé. Par exemple, le cadriciel peut être utilisé pour des problèmes traités comme des problèmes de programmation par contraintes, tout comme il peut être utilisé avec des métaheuristiques ou des algorithmes dédiés.

Les différences entre les la BCOO et HeuristicLab sont principalement :

- au niveau des modèles : l'absence de modélisation du critère,
- au niveau de l'implantation : des choix techniques différents.

En ce qui concerne les modèles, les classes "solutions" et "critères" sont confondues. Ceci implique l'inexistence de l'évaluation du critère. Ainsi, un utilisateur d'HeuristicLab souhaitant étudier un même problème sous deux objectifs différents (minimiser le makespan ou minimiser un retard) doit écrire deux évaluations distinctes.

En ce qui concerne les choix techniques, HeuristicLab propose une utilisation massive des "delegates" pour séparer les traitements des objets. Les "delegates" sont une notion proposée par Microsoft dans les langages natifs de la plateforme .NET et qui ont aussi été introduit dans leur machine virtuelle JAVA. Les delegates sont une évolution des pointeurs de fonction rendant cette notion compatible avec le paradigme objet. Cette notion n'est pas soutenue par d'autres langages objet et est très dépendante de ces plateformes.

En outre, les auteurs de HeuristicLab limitent la portée de leurs travaux en ne proposant que des modèles spécifiques aux choix techniques réalisés. Ils ne proposent pas de modèle indépendant de la plateforme.

L'architecture proposée par HeuristicLab est assez proche de celle de la BCOO, et il est nécessaire de préciser que nos développements ont été réalisés en parallèle et indépendamment. Les auteurs proposent sur leur site un historique, la première communication est une conférence en 2004, alors que nos développements ont commencé en 2002. Nos travaux n'ont pas fait l'objet d'une publication en conférence ni en revue. Par contre, Cossard (2004) a, pendant sa thèse, utilisé une

[10] http://sourceforge.net/

première version de la BCOO pour faire l'optimisation de la planification tactique. Ainsi, il présente dans ses annexes les principes de la BCOO.

3.4 Conclusion sur les bibliothèques d'optimisation

Les bibliothèques d'optimisation présentées ci-dessus ont été classées en deux catégories distinctes : les bibliothèques structurant les algorithmes de recherche et les bibliothèques structurant l'évaluation d'une solution. Chaque catégorie a ses propres avantages et inconvénients.

Les bibliothèques structurant les algorithmes de recherche sont limitées à une classe d'algorithmes : les algorithmes génétiques, les algorithmes à améliorations locales, ... Ils n'offrent très peu voire aucune fonctionnalité aux utilisateurs voulant implanter une autre sorte d'algorithmes. De plus, puisqu'ils ne connaissent pas la structure d'une solution ni de son évaluation, ils fournissent peu voir aucun outil pour la construction d'interfaces graphiques permettant de manipuler une solution. Les interfaces graphiques basées sur ces bibliothèques sont essentiellement tournées vers l'optimisation elle-même : lancement d'une optimisation, lancement d'optimisation en batch, réglage des paramètres de l'optimisation, ...

Les bibliothèques structurant l'évaluation d'une solution imposent leur structure aux utilisateurs. Ainsi, un utilisateur voulant implanter une descente stochastique pour le problème de voyageur de commerce aura recours à des concepts complexes et à une mise en œuvre non triviale alors que l'évaluation d'une solution de ce problème peut se réaliser de manière très efficace en sommant la durée de tous les arcs de la solution. Ainsi, ces bibliothèques peuvent être relativement inefficaces sur certains problèmes. De plus, leur structure impose souvent une approche de résolution : les bibliothèques basées sur le CSP sont plus adaptés aux raisonnements logiques qu'à une méthode du type recherche locale, les bibliothèques basées sur des contraintes linéaires sont évidemment plus adaptée à l'optimisation linéaire...

Bibliothèque	Extension	Vérification des contraintes	Fonctions supplémentaires
DÉJÀ VU	Polymorphisme	Pondérée	Ordonnancement interactif
Localizer++	Polymorphisme	Stricte	
INCOP	Polymorphisme	∅	
Hotframe	Template	∅	Interface de génération de modèles
EO et ParadiseEO		∅	

Tableau 5-2. Synthèse des bibliothèques d'optimisation de la littérature

4 Notions proposées

Pour structurer les algorithmes d'optimisation, nous proposons six notions sur lesquelles le cadriciel repose. Ces notions ont toutes pour objectif de décrire les transformations d'une solution durant le processus d'optimisation. Les trois premières notions concernent les trois formes qu'une solution prend pendant l'exécution d'une optimisation : solution codée, solution et critère. Les deux notions suivantes concernent les algorithmes d'évaluation qui permettent de passer d'une forme de

solution à la suivante : évaluation et évaluation du critère. Pour finir, la transformation regroupe toutes ces notions (cf. figure 5-2).

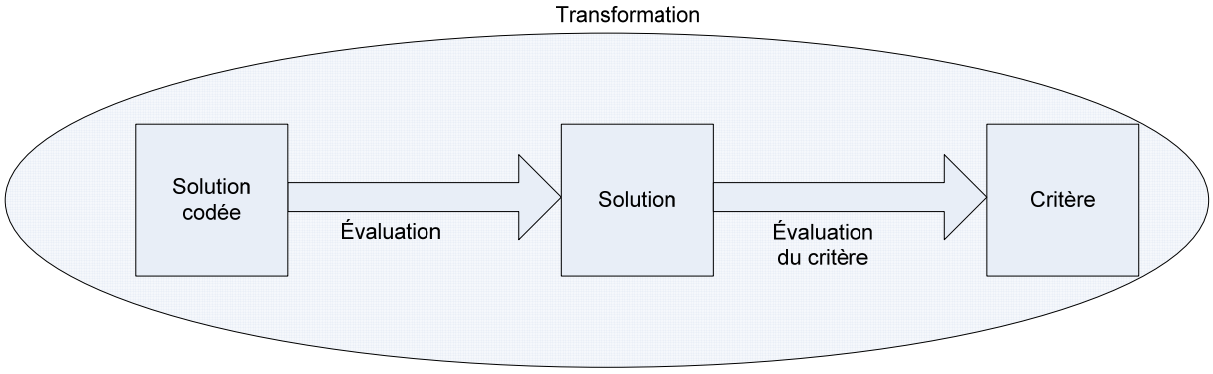

Figure 5-2. Détails d'une transformation

- Solution codée : elle est utilisée pour stocker une solution en mémoire ou sur disque. Une solution codée est la forme la plus compacte d'une solution, elle contient donc le minimum d'informations possible.
- Evaluation : permet de compléter les informations présentes dans une solution codée.
- Solution : décrit une solution complètement exprimée.
- Evaluation du critère : permet d'évaluer un critère.
- Critère : est la valeur à optimiser. La plupart des critères sont scalaires (et même entiers), mais un critère sous forme de vecteur est aussi envisageable comme dans le cas de l'optimisation multi objectif.
- Transformation : contient les 5 éléments ci-dessus. Une transformation est un regroupement cohérent qui décrit, pour un problème donné, comment passer d'une solution codée à la valeur du critère.

Pour illustrer ces notions, nous montrons comment les appliquer à la représentation semi-active pour le problème du jobshop. Dans cet exemple, une solution est codée par un ordre des opérations sur chaque machine. La solution codée est donc composée d'un vecteur de numéros d'opérations sur chaque machine, ce qui forme un vecteur de vecteur de numéros d'opérations. Une solution du problème de jobshop est un ordonnancement qui précise pour chaque opération sa date de début. Une solution est donc un vecteur de dates de début des opérations. Pour passer de l'ordre des opérations sur les machines à un ordonnancement, on utilise le graphe conjonctif-disjonctif. Ainsi, l'algorithme d'évaluation consiste à exécuter l'algorithme de plus longs chemins proposés dans le chapitre 2. Dans le problème de jobshop classique, on cherche l'ordonnancement de plus faible makespan. Ainsi, une valeur du critère est un makespan (i.e. un entier). La classe critère est donc simplement constituée d'un entier. Pour calculer le critère d'une solution, il faut calculer le makespan d'un ordonnancement. Ceci est facilement réalisé par un algorithme cherchant la plus grande valeur dans le vecteur des dates de début. Un synoptique de l'utilisation de ces notions pour la représentation semi active du jobshop est présenté dans la figure 5-3.

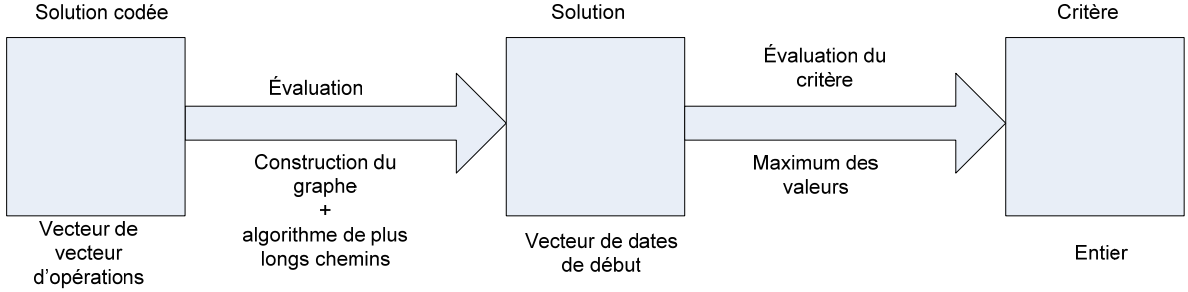

Figure 5-3. Détails de la transformation de la représentation semi active pour le jobshop

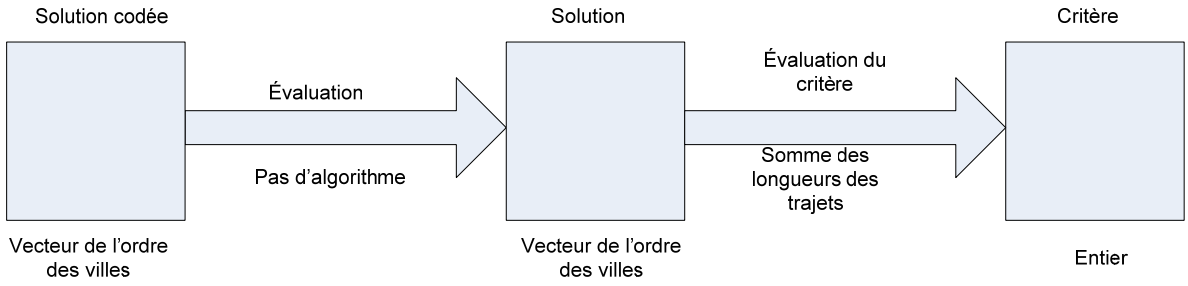

Figure 5-4. Détails de la transformation de la représentation semi active pour le TSP

Le premier exemple de transformation montre un cas classique d'utilisation d'une transformation. Dans la figure 5-4, nous présentons un second exemple qui montre que cette architecture n'est pas rigide et peut s'adapter à des cas particuliers. La transformation présentée dans cette figure concerne le problème du TSP (Traveling Salesman Problem) ou "problème de voyageur de commerce". Le problème de voyageur de commerce consiste à chercher un cycle dans le graphe formé par les villes à visiter. Le cycle recherché doit minimiser la somme des distances portées par les arcs entre les villes. Une solution codée est formée de la liste des villes à visiter dans l'ordre envisagé. L'algorithme d'évaluation consiste donc à sommer les longueurs des arcs entre les villes successives dans la solution codée. Cette évaluation fournit donc directement la valeur du critère stockée dans la solution. L'évaluation du critère n'a plus qu'à recopier la valeur stockée dans la solution pour la mettre dans le critère.

5 Spécifications de la BCOO

Dans ce paragraphe, nous présentons les spécifications générales de la BCOO. Ces spécifications sont décrites à un niveau général, elles ne présupposent donc d'aucun choix de plateforme. D'après les préconisations de MDA (Model Driven Architecture) (Blanc, 2005), les spécifications correspondent au modèle appelé PIM (Platform Independant Model). Outre les descriptions textuelles, cette spécification est décrite à l'aide du formalisme UML car c'est un formalisme orienté objet largement répandu.

Cette section contient les spécifications de la BCOO. Nous proposons de répartir les classes du cadriciel en 4 paquetages distincts : "Spécifique au Problème", "BCOO_Module_Affichage", "BCOO_Optimisation" et "BCOO_Classes de base". Un paquetage contient un ensemble de classes cohérentes, faites pour réaliser un même objectif. Ainsi, le paquetage "BCOO_Optimisation" contient toutes les classes purement optimisation alors que le paquetage "BCOO_Module_Affichage" contient toutes les classes d'affichage pour les interfaces homme / machine (IHM).

Parmi ces 4 paquetages, les trois préfixés par "BCOO_" sont des paquetages génériques qui peuvent être réutilisés dans tous les problèmes, alors que le paquetage "Spécifique au Problème" est, comme son nom l'indique, dépendant du problème.

La décomposition en quatre paquetages permet de répartir les classes de la manière suivante : le paquetage "BCOO_Classes de base" contient les interfaces de chaque élément de la transformation et une bibliothèque de classes qui archive toutes les classes préprogrammées implantant des comportements par défaut. Le paquetage "BCOO_Optimisation" contient des algorithmes d'optimisation généralistes et fournit les classes de base pour construire les algorithmes d'optimisation. En particulier, il permet de structurer les algorithmes à base de voisinage. Le paquetage "Spécifique au Problème" permet de spécifier les caractéristiques du problème comme les données, l'évaluation d'une solution et éventuellement des algorithmes d'optimisation spécifiques. Éventuellement, l'application construite peut être munie d'une interface graphique en utilisant le paquetage "BCOO_Module_Affichage.

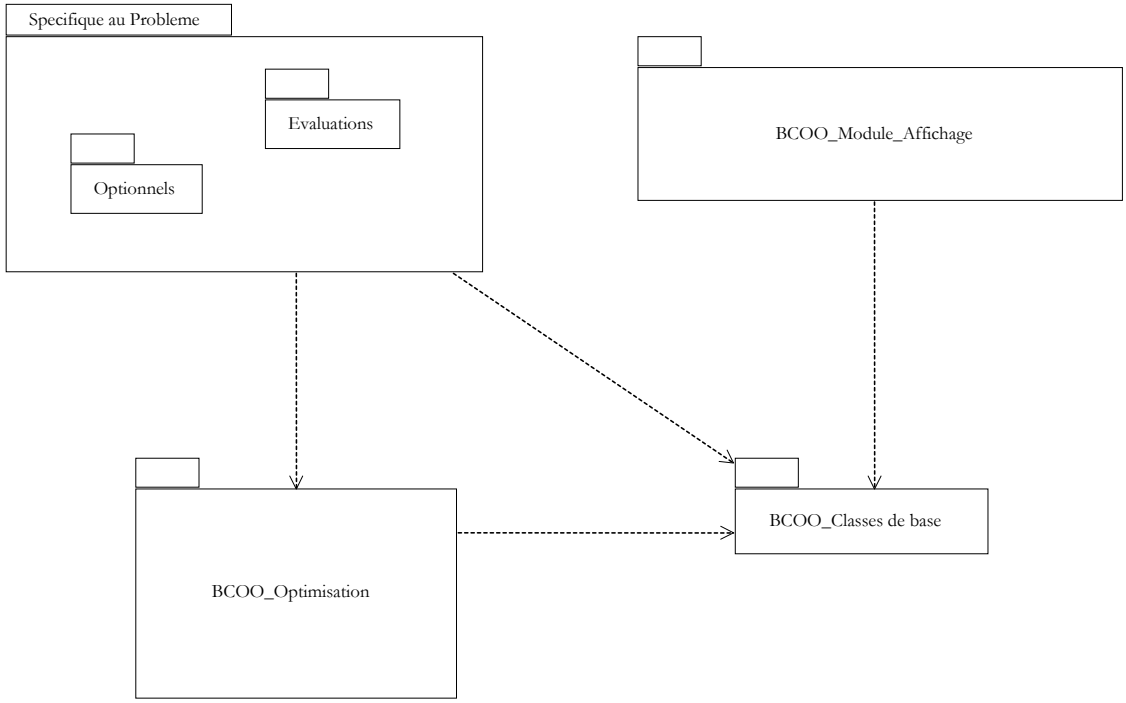

Figure 5-5. Vues globales des paquetages

Les liens entre ces paquetages sont symbolisés sur la figure 5-5, ils peuvent se résumer par les deux phrases suivantes : tous les paquetages utilisent le paquetage "BCOO_Classes de base" et le paquetage "Spécifique au Problème" utilise tous les autres paquetages.

5.1 Paquetage "BCOO_Classes de base"

Le paquetage "BCOO_Classes de base" (cf. figure 5-6) est le noyau du cadriciel que tous les autres paquetages utilisent. Il contient des classes techniques (i.e. nécessaire pour l'implantation), mais aussi des classes abstraites et interfaces qui servent de référence aux paquetages extérieurs.

Une application utilisant la BCOO doit spécialiser les trois classes abstraites dénommées "CustomProjet", "CustomOptimisation" et "CustomTransformation". La classe "CustomProjet" agrège toutes les autres classes comme les transformations et les optimisations. Elle devra être spécialisée par une classe projet dans le paquetage "Spécifique au Problème". Entre autres, la méthode Load permettant de charger une nouvelle instance doit être spécialisée. La classe "CustomOptimisation" est la classe de base servant à toute classe d'optimisation et la classe "CustomTransformation" est la classe de base de toutes les transformations.

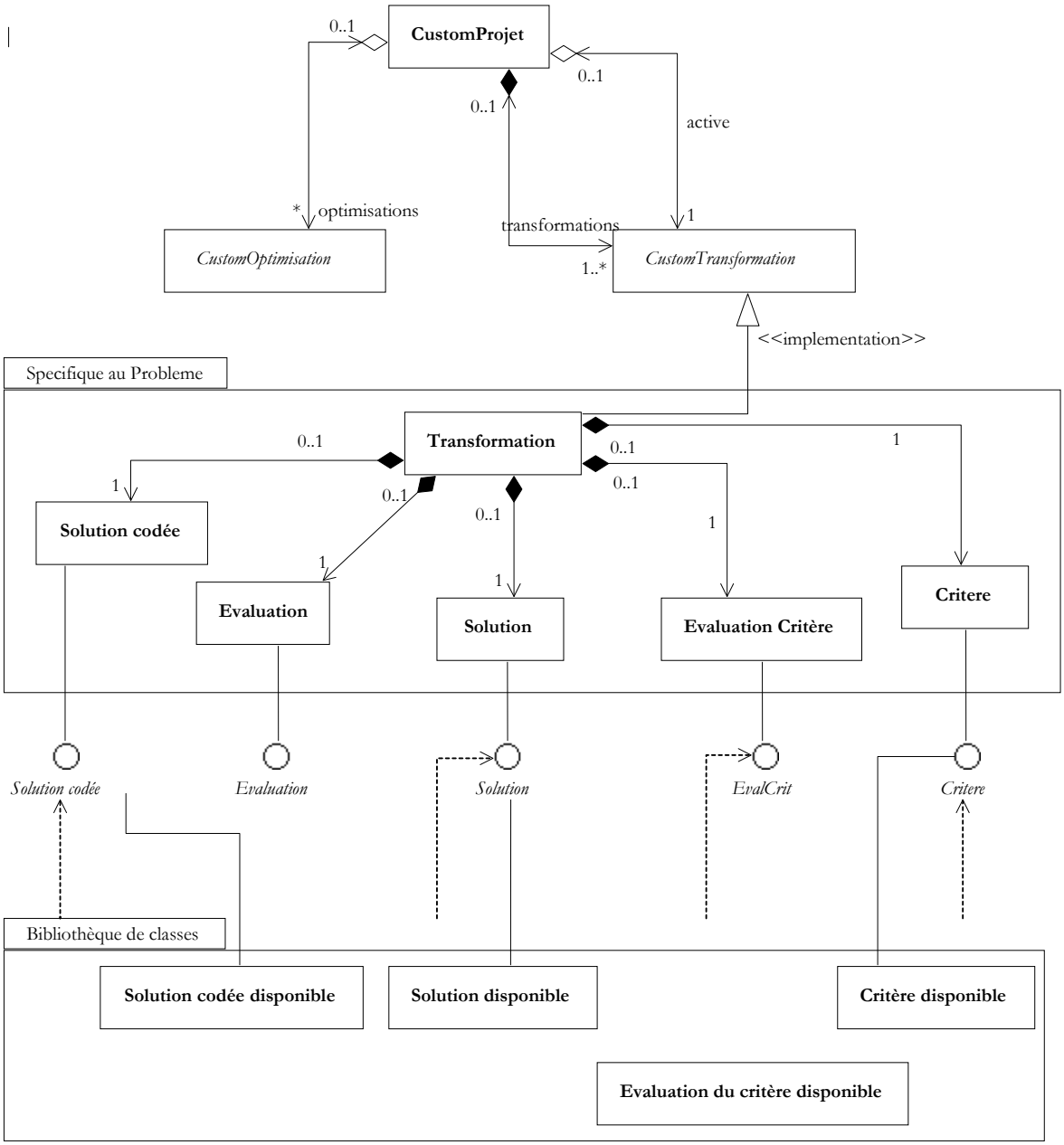

Figure 5-6. Paquetage BCOO_Classes de base

De plus, le paquetage "BCOO_Classes de base" contient 6 interfaces principales : "Solution codée", "Evaluation", "Solution", "Critère", "EvalCrit" et "Comparable" qui permettent de manipuler toutes les parties d'une transformation ou une transformation entière sans connaître celle-ci. Ceci est particulièrement utile pour pouvoir écrire du code générique quelle que soit la transformation. Par exemple, on utilise ces interfaces pour pouvoir développer des algorithmes d'optimisation indépendants de la transformation.

5.1.1 Interface "Solution codée"

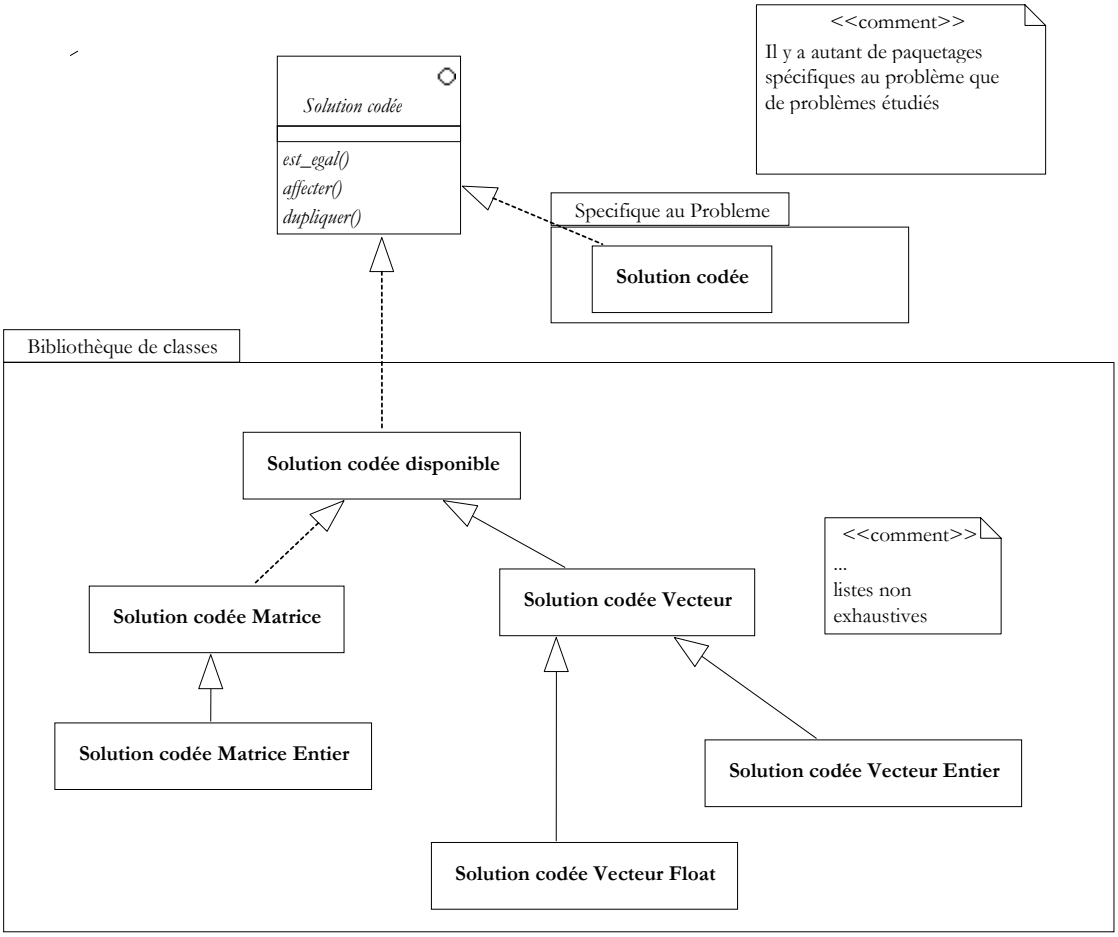

Figure 5-7. Solution codée

Dans la BCOO, une "Solution codée" est la classe représentant une solution telle qu'elle doit être sauvegardée en mémoire. Son but est de stocker le minimum d'informations nécessaires pour représenter une solution. Pour être une solution codée, une classe doit répondre à l'interface "Solution codée" du paquetage "BCOO_Classes de base". Cette interface requiert les trois méthodes suivantes :

- La méthode "est_egal" est utilisée pour pouvoir comparer deux solutions codées. Nous avons ajouté cette fonctionnalité aux objets solutions codées car elle est très souvent requise dans les algorithmes d'optimisation et car sa mise en œuvre est généralement triviale.
- La méthode "affecter" est utilisée pour pouvoir modifier une solution codée en une autre. Cette méthode est généralement utile pour recopier une solution codée, ce qui est particulièrement nécessaire lors d'une optimisation.
- La méthode "dupliquer" demande à une solution codée de créer une nouvelle instance de solution codée contenant les mêmes informations que l'instance à dupliquer. Cette méthode est aussi utile pour la duplication d'instances.

Comme l'indique la figure 5-7, les classes relatives aux solutions codées sont présentes dans trois paquetages. D'abord, le paquetage "BCOO_Classes de base" contient l'interface "Solution codée". Cette interface est largement utilisée dans tous les autres paquetages. Par exemple, à travers l'interface de "Solution codée", une interface graphique ou un algorithme d'optimisation peuvent manipuler une solution codée sans réellement connaître l'implantation utilisée. Cette propriété est utile pour écrire des interfaces graphiques ou des algorithmes d'optimisation génériques.

De plus, le paquetage "Bibliothèques de classes" contient un ensemble de "solutions codées" préprogrammées et disponibles pour les utilisateurs de la BCOO. En utilisant des solutions codées existantes, on peut ainsi réduire le temps de développement d'un nouveau projet. En effet, la plupart des algorithmes d'optimisation utilisent les mêmes "solutions codées".

L'utilisation de cette bibliothèque de classes n'interdit pas pour autant à un utilisateur de créer sa propre "solution codée" dans le paquetage "Spécifique au Problème". La seule contrainte est que la classe proposée doit répondre à l'interface "Solution codée". Dès lors, cette classe s'intègre au cadriciel et peut être utilisée dans des algorithmes d'optimisation et dans des interfaces existantes.

5.1.2 Interface "Évaluation"

L'évaluation permet de passer d'une solution codée à une solution. La fonctionnalité principale de l'évaluation est donc réalisée à travers l'unique méthode de son interface dénommée "evaluation". Pour pouvoir réaliser cette transformation, une évaluation doit donc connaître les classes "solution codée" et "solution" utilisées.

Bien sûr, une évaluation étant spécifique à un problème donné, son implantation doit être réalisée dans le paquetage "Spécifique au Problème" car elle n'est pas, à priori, réutilisable pour d'autres problèmes. En tout cas, le cadriciel ne fournit aucune aide ni pour modéliser ni pour organiser les classes "évaluation".

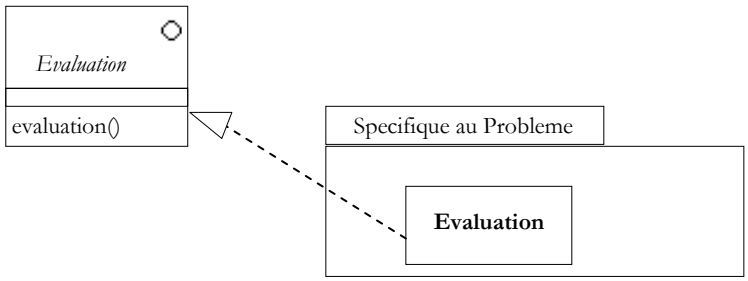

Figure 5-8. Evaluations

La classe évaluation ne comprend donc à priori qu'une seule méthode. Mais, pour des raisons techniques et pour des raisons de performance, on peut lui ajouter d'autres méthodes afin de structurer la méthode "évaluation" et des attributs afin de conserver des informations d'un calcul pour le prochain calcul. Les attributs peuvent être utilisés pour des fins techniques comme par exemple réserver des tableaux dynamiques aux dimensions du problème.

5.1.3 Interface "Solution"

La solution est le résultat d'une évaluation. Elle décrit complètement une solution du problème dans tous ses détails : dans une solution, toutes les décisions sont déjà prises et la description est complètement non ambiguë. Ainsi, une solution peut être directement fournie à un utilisateur sous forme de fichier, sous forme graphique ou autre.

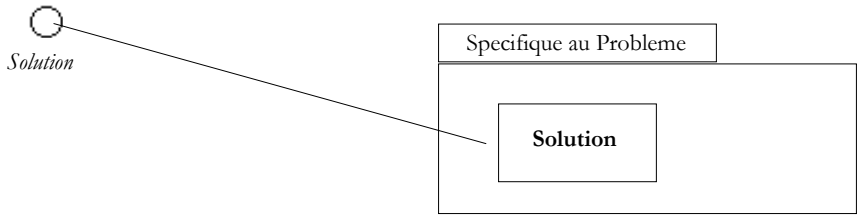

Figure 5-9. Solutions

5.1.4 Interface "Evaluation critère"

L'interface "Evaluation critère" permet d'évaluer une solution en calculant la valeur de son critère. Cette évaluation se fait directement à l'aide des informations contenues dans la solution. Suivant les problèmes étudiés et suivant le critère choisi, ce calcul peut être plus ou moins trivial. Par exemple, une solution peut donner, à l'aide de ses attributs, la dernière opération ordonnancée. Dans ce cas, la classe "evaluation du critère" se limite à la récupération de la date de fin de la dernière opération.

Certaines évaluations de critère peuvent être codées à l'avance. Par exemple, une évaluation qui consiste à chercher le plus grand élément dans une solution vecteur. Cette recherche du plus grand élément peut être programmée de manière générique en lui passant en paramètres la méthode de comparaison (i.e. sous la forme d'un foncteur). Ainsi, grâce à la méthode contenue dans le foncteur, on peut spécifier la comparaison de deux éléments.

5.1.5 Interface "Critère"

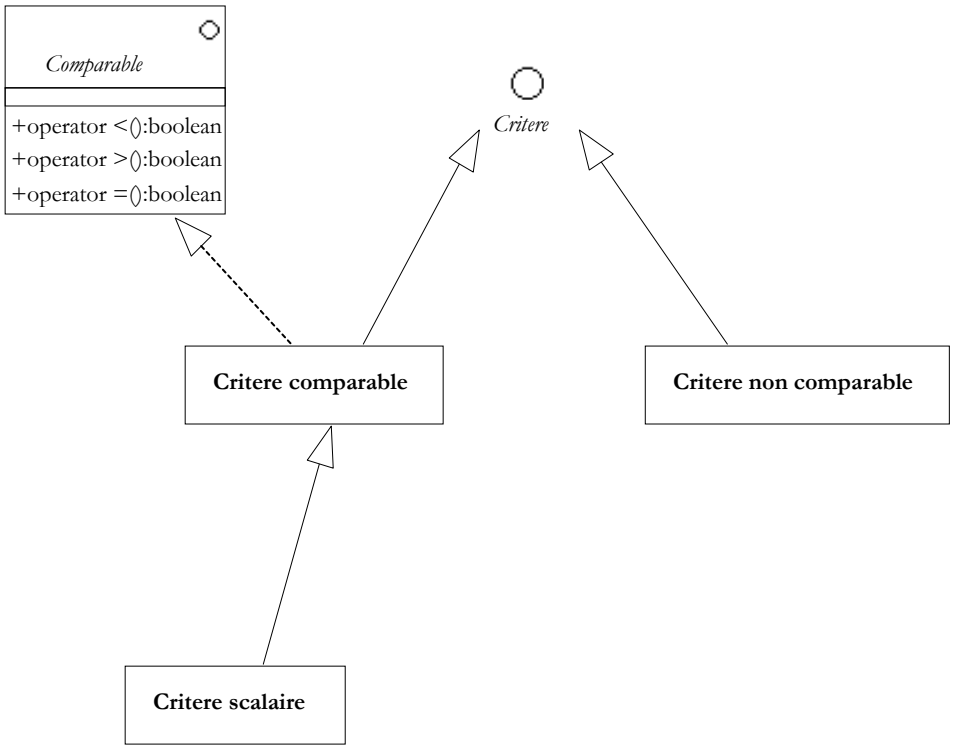

Figure 5-10. Critères

L'interface critère est proposée dans le paquetage "BCOO_Classes de base". Elle ne comporte à priori aucune méthode. Mais de nombreux algorithmes d'optimisation ont besoin de comparer deux solutions en comparant leur critère.

L'interface "Comparable" permet de comparer deux solutions à l'aide des opérateurs < et =. Ces deux opérateurs permettent de générer tous les opérateurs de comparaison nécessaires comme <=, ou >.

Parmi les classes comparables se trouvent les critères scalaires : par exemple, un critère entier ou en nombre flottant est un critère scalaire. Ces classes critères scalaires peuvent être utilement codées à l'avance car utilisée dans la plupart des algorithmes d'optimisation. Par contre, tous les critères ne sont pas scalaires. Par exemple, les problèmes multi objectif nécessitent une classe critère composée de plusieurs nombres.

5.1.6 Interfaces génériques

Une interface générique est une interface dont le contenu ne dépend pas du problème étudié. Typiquement, on peut y charger les données d'un problème, et lancer une ou plusieurs optimisations. On y manipule donc des solutions codées, des solutions et des critères sans pour autant connaître leurs implantations. Les fonctionnalités décrites ci-dessus sont suffisantes pour réaliser une interface générique basique.

Afin de rendre l'interface plus complète, il est possible d'ajouter des fonctionnalités de chargement et sauvegarde aux classes "Solution codée", "Solution" et "Critère".

5.2 Paquetage "BCOO_Optimisation"

Le paquetage BCOO_Optimisation contient les classes d'optimisation à proprement parler. On y trouve les algorithmes d'optimisation préprogrammés ainsi que les classes permettant de structurer la liste des algorithmes d'optimisation.

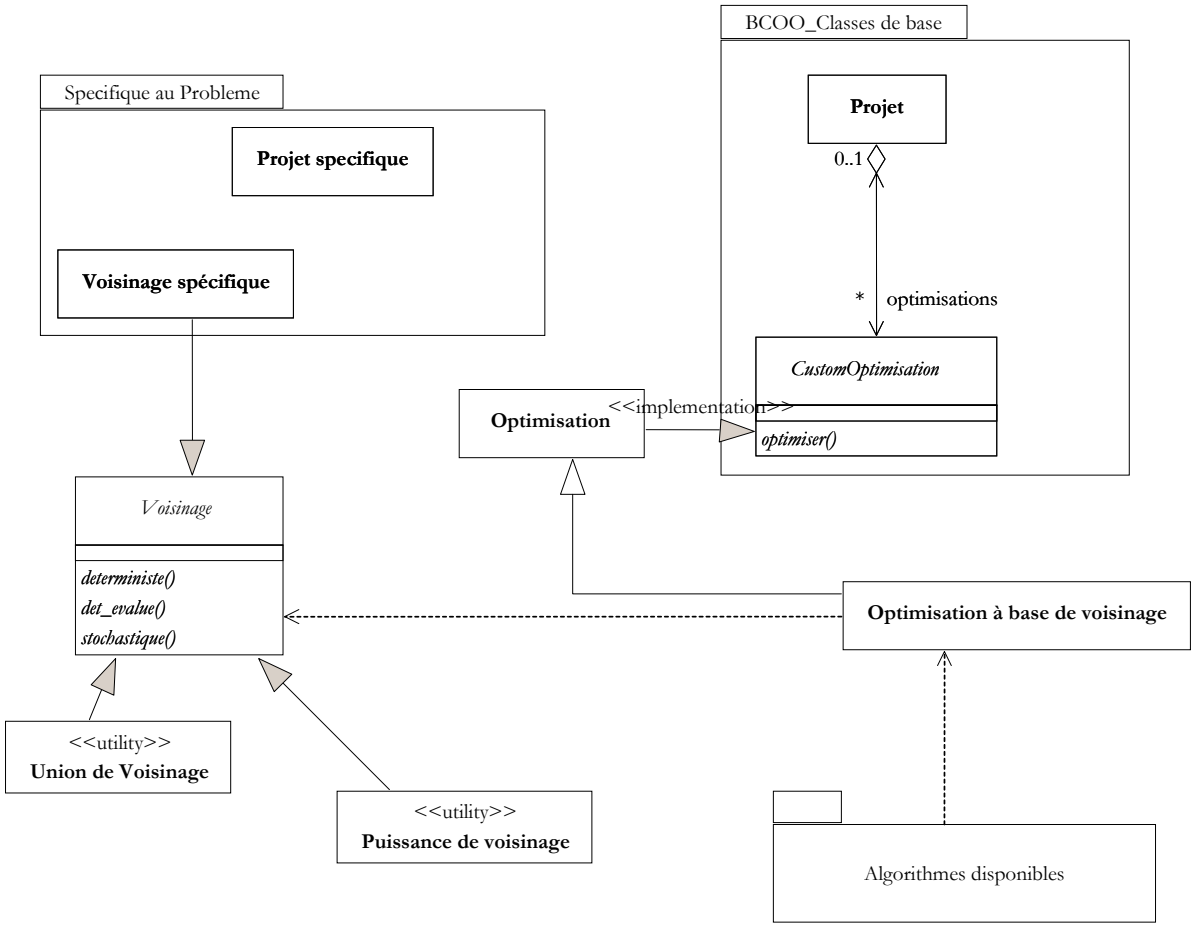

Figure 5-11. Paquetage BCOO_Optimisation

5.2.1 Classe "Optimisation"

La principale classe du paquetage est la classe "Optimisation" qui hérite de "CustomOptimisation". Peu de méthodes et d'attributs y sont disponibles car elle est commune à tous les algorithmes d'optimisation et que peu de caractéristiques leur sont communes.

Par contre, toute structuration d'un ensemble d'algorithme d'optimisation peut être prise en compte afin d'aider au développement. Pour cela, une hiérarchie de classe peut être introduite en tant que spécialisation de la classe "Optimisation". Par exemple, la classe "Optimisation à base de voisinage", qui utilise la classe "Voisinage", peut être introduite pour structurer les algorithmes itératifs que nous avons développés dans cette thèse. Même si un effort particulier a été porté pour prendre en compte les méthodes d'optimisation à base de voisinage, le cadriciel n'est absolument pas limité à celles-ci.

5.2.2 Sous paquetage "Algorithmes disponibles"

Le sous paquetage "Algorithmes disponibles" contient des algorithmes d'optimisation généralistes (i.e. qui ne contiennent pas de connaissance spécifique à un problème). Concrètement ces algorithmes sont ceux qui n'utilisent aucune classe et aucune information du paquetage "Spécifique au Problème". Les algorithmes présents dans ce paquetage sont stockés afin d'être disponibles à tout

utilisateur du cadriciel. Ainsi, un nouveau problème d'optimisation peut éventuellement être résolu par n'importe lequel de ces algorithmes.

On trouve par exemple des algorithmes de type métaheuristique comme la descente stochastique, la descente déterministe, le recuit simulé ou la méthode taboue.

5.3 Paquetage "Spécifique au Problème"

Le paquetage "Spécifique au Problème" contient toutes les classes utilisant de la connaissance spécifique au problème, comme par exemple la transformation ou les algorithmes d'optimisation.

5.3.1 Classe "Projet"

La classe projet est la classe qui agrège toutes les autres classes. Elle dérive de la classe "Custom Projet" du paquetage "BCOO_Classes de base" qui lui fournit donc les fonctionnalités d'agrégation des optimisations (classe "CustomOptimisation") et des transformations (classe "CustomTransformation"). Ainsi, toute référence à la classe Projet permet de remonter aux algorithmes et aux transformations définis dans le projet.

Outre ses fonctionnalités d'agrégation, la classe "projet" est aussi utile pour stocker toutes les données relatives à l'instance du problème en cours de résolution. Ce stockage de données ne peut être décrit de manière générique car il est complètement dépendant du problème.

5.3.2 Classe "Solution codée"

Il est relativement rare d'avoir besoin d'écrire une solution codée spécifique à un problème. La plupart du temps, on peut réutiliser des solutions codées déjà disponibles dans le cadriciel.

5.3.3 Classe "Evaluation"

Contrairement à la solution codée, on trouve systématiquement une évaluation dans le paquetage "Spécifique au problème". En effet, une évaluation définit comment on passe d'une solution codée à la solution correspondante, ce passage est donc complètement spécifique au problème et doit être dans ce paquetage.

Puisque l'évaluation est dans le paquetage "Spécifique au problème, elle connaît toutes les données du problème stockées dans la classe projet.

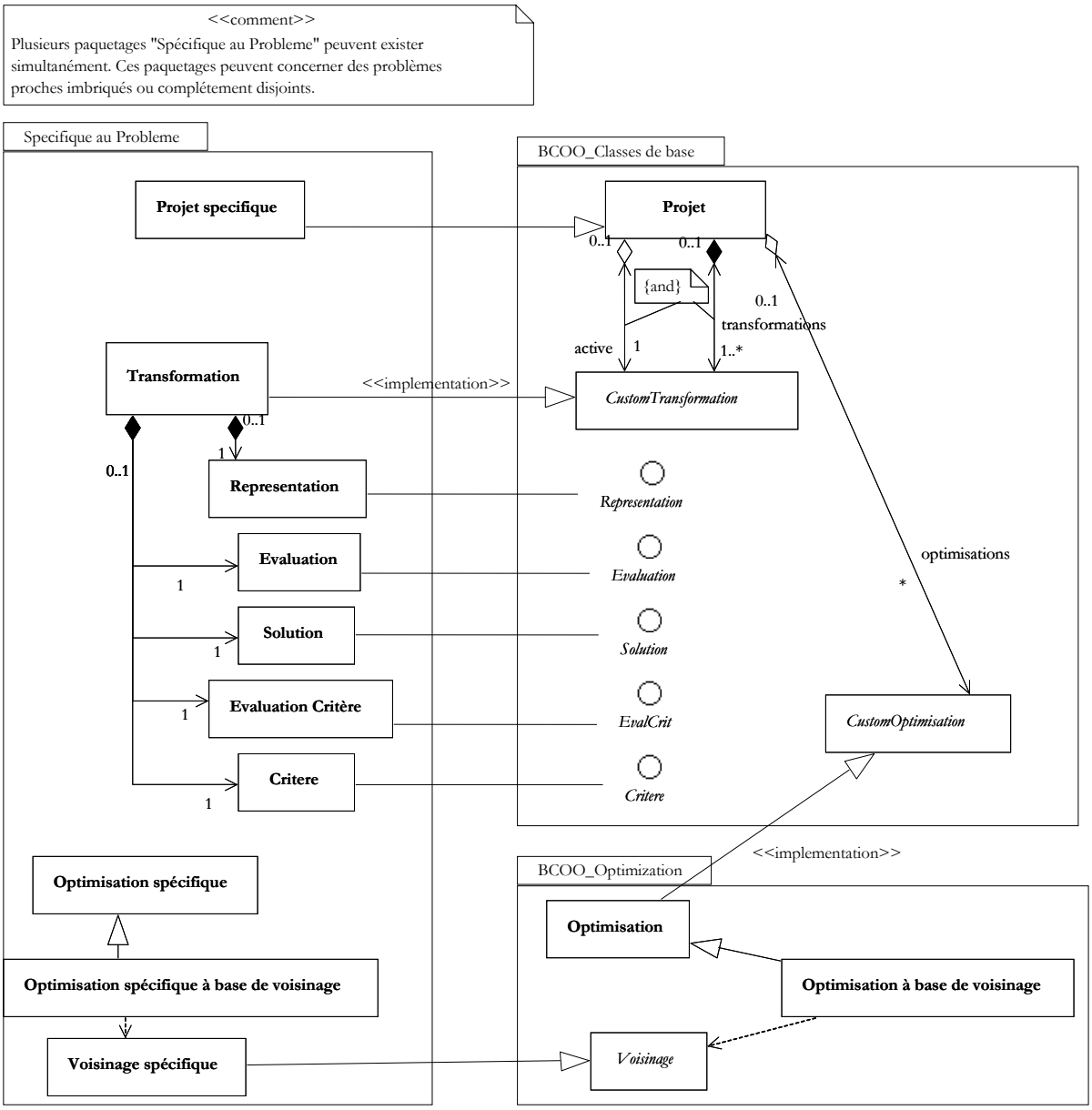

Figure 5-12. Paquetage Specifique au problème

5.3.4 Classe "Solution"

Les solutions sont, comme les solutions codées, relativement générales et il est souvent possible de réutiliser des solutions génériques du paquetage "BCOO_Classes de base". Pourtant, suivant le langage de programmation utilisé, cette remarque peut être plus ou moins pertinente. En fait, tous les langages munis de classes paramétrées permettent réellement d'écrire la plupart des solutions et de les stocker dans le paquetage "BCOO_Classes de base" : on peut alors utiliser les classes Vecteurs, Vecteurs de Vecteurs et matrices dont le type d'éléments est paramétré.

Si le langage de programmation ne permet pas l'utilisation de classes paramétrées, il est généralement nécessaire de réécrire des solutions spécifiques afin de stocker tous les éléments d'une solution. Par exemple, une solution du problème de voyageur de commerce est une liste de villes qui se code à l'aide d'un vecteur d'entiers. Une solution du problème de flowshop hybride à deux machines

sur un même étage est une liste de paire (numéro de job et numéro de machine). Dans un langage implantant les classes paramétrées, il suffit d'écrire une seule classe, appelée vecteur, et dont le type d'élément est paramétré. Dans un langage ne permettant les classes paramétrées, il faut écrire deux classes distinctes.

Même dans le cas d'un langage de programmation dédié, il peut être nécessaire de stocker des informations supplémentaires que la classe n'a pas initialement prévu. Par exemple, dans une solution du problème de jobshop, outre les dates de début stockées dans le vecteur d'entiers, il peut être intéressant de stocker le numéro de la dernière opération. Cette information est un entier qu'il faut stocker en plus des données du vecteur. Afin d'éviter à un utilisateur du cadriciel d'avoir à développer une "Solution" spécifique, nous préconisons d'ajouter dans toutes les classes "Solutions" un attribut supplémentaire de type entier (valeur tag) qui ne sert pas au cadriciel mais qui permet à l'utilisateur de stocker des valeurs.

5.3.5 Classe "Evaluation critère"

L'évaluation du critère est aussi relativement générique : par exemple, le makespan est la plus grande date de fin de traitement, ce qui revient à déterminer la plus grande valeur dans la solution stockée. Cet algorithme peut être écrit de manière générique si le langage de programmation le permet.

Pour certains problèmes, la frontière entre calcul d'une solution et calcul du critère est faible et il est artificiel d'interdire à l'algorithme d'évaluation de calculer le critère. Pour des raisons de rapidité d'exécution du code, on peut donc souhaiter que l'évaluation réalise le calcul de certains critères. Il est alors possible de stocker la valeur du ou des critères dans la solution, (l'utilisation des valeurs "tags" est tout à fait appropriée). L'algorithme d'évaluation du critère consiste alors à recopier cette valeur dans le critère. Ainsi, on dispose alors d'un algorithme d'évaluation qui permet de transformer une solution codée en une solution et qui stocke dans cette dernière quelques critères d'évaluation. L'utilisateur du cadriciel peut alors choisir quelle évaluation du critère il utilise pour savoir quelle valeur va être recopiée dans le critère. L'utilisation de l'évaluation du critère permet alors de rendre facile et souple le choix du critère à optimiser.

5.3.6 Classe "Critère"

Tout comme la solution codée, il n'y a généralement pas de classe critère dans le paquetage "spécifique au problème". En effet, en proposant une classe critère "Entier", une classe critère "Nombre à virgule flottante", puis éventuellement des classes contenant deux scalaires pour créer des critères bi-objectifs, on dispose alors d'un éventail très large de critère couvrant la plupart des besoins.

5.3.7 Classe "Optimisation spécifique"

En étant dans le paquetage "Spécifique au Problème", la classe "optimisation spécifique" connaît quelle transformation elle est en train de traiter. Ainsi, toute heuristique complètement dédiée ou toute métaheuristique dédiée peuvent être programmée car elle connaît et a accès à toutes les classes du cadriciel.

D'un point de vue technique, la classe "Optimisation spécifique" a accès au "CustomProjet" qui la contient grâce aux liens de navigations proposés par le paquetage "BCOO_Classes de base". Puisque l'algorithme est spécifique, il connaît le "Projet" utilisé. Ainsi, il est capable de remonter (par upcast) à toutes les informations contenues dans le "Projet Spécifique". De même, la transformation étant connue, on peut directement accéder à toutes les informations qui y sont contenues.

Suivant le langage de programmation utilisé, ces upcasts ne sont pas réalisés de la même manière. Si le langage dispose de fonctionnalités décrivant le type d'une donnée à l'exécution (Run Time Type Information), il suffit alors d'utiliser ces fonctionnalités. Sinon, on réalise directement des transtypages brutaux de la classe "CustomProjet" vers la classe spécialisée. Ce transtypage n'a besoin d'être réalisé qu'une seule fois au début de l'optimisation.

5.3.8 Classe "Voisinage spécifique"

Les voisinages spécifiques sont des voisinages spécialement conçus pour un problème donné. Il ne faut pas confondre les voisinages spécifiques avec les voisinages destinés à une certaine solution codée. En effet, un voisinage manipule toujours une solution codée, et il n'est pas possible d'écrire un voisinage sans connaître pour quelle "Solution Codée" il a été destiné. Par exemple, un voisinage de permutation ne peut être programmé sans savoir si l'on permute deux éléments d'un vecteur ou deux éléments dans un vecteur de vecteur. Tout voisinage est donc destiné à une certaine "Solution codée".

Les voisinages spécifiques utilisent en plus des informations sur le problème traité. Par exemple, ils utilisent des données présentes dans le "Projet spécifique" ou ils utilisent les informations contenues dans la "Solution" pour déterminer les voisins.

En général, les voisinages spécifiques sont des voisinages guidés.

5.4 Paquetage "BCOO_Module Affichage"

Le paquetage "Module Affichage" contient toutes les classes de visualisation du cadriciel. Ce paquetage n'est donc pas très général car il dépend de la technologie retenue pour la visualisation et éventuellement des plateformes qui les implantent.

5.4.1 Généralités

Un module d'affichage est forcément dédié aux éléments qu'il souhaite afficher. Les fonctionnalités suivantes sont les fonctionnalités principales souhaitées pour ce module d'affichage :

- la visualisation d'éléments de transformations,
- la modification d'une solution codée,
- l'affichage d'une solution,
- des composants de suivi de l'évolution de l'algorithme d'optimisation.

Comme nous l'avons indiqué, il n'est pas possible d'afficher une solution sans en connaître son contenu. C'est pourquoi nous ne décrivons pas plus les composants que nous avons développés afin de répondre à ces souhaits.

5.4.2 Diagramme de Gantt générique

Dans l'objectif de réduire les temps de développements, nous avons proposé un ensemble de composants réutilisables permettant de construire des interfaces graphiques.

Ces composants reposent sur la création d'un objet (hub) concentrant toutes les informations nécessaires pour l'affichage des solutions. À l'aide de ce hub, un utilisateur du cadriciel peut décrire son problème en renseignant les informations sur les jobs, les machines et les opérations. Ces classes comportent les principales informations des opérations comme la date de début, la durée d'une opération, le nom d'un job ou d'une machine, la plage d'indisponibilité d'une machine etc... L'intérêt du hub est que toutes ces informations sont complètement indépendantes de celles stockées dans le projet et utilisées pour le problème. En effet, c'est l'utilisateur du cadriciel qui doit transformer ses propres informations pour les fournir au hub.

Ainsi, tout utilisateur du diagramme de Gantt générique a décrit son problème dans le hub, sous une forme standard. Il peut alors profiter pleinement de tous les composants graphiques associés comme une liste de jobs, une liste de machines, un diagramme de Gantt dynamique.

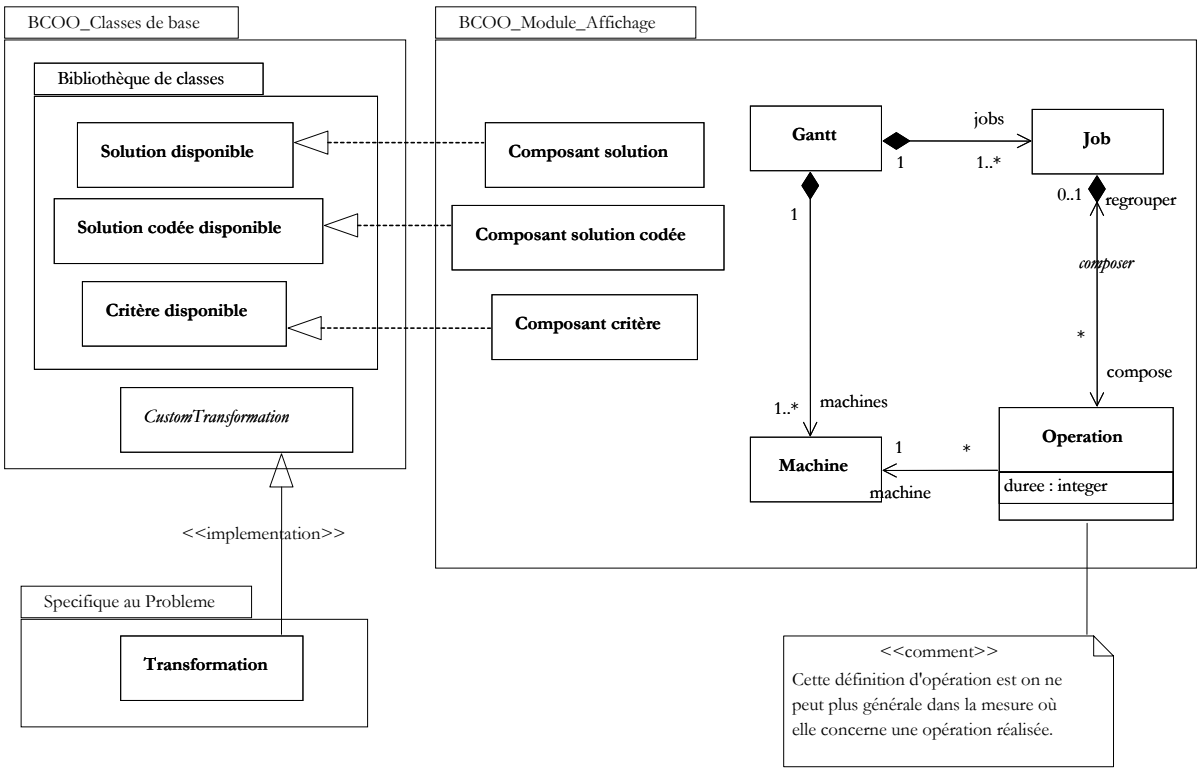

Figure 5-13. Paquetage Module Affichage

6 Mise en œuvre de la BCOO en C++ et VCL©

Dans le paragraphe précédent, nous avons décrit les spécifications générales de la BCOO. Dans ce paragraphe, nous présentons une mise en œuvre réalisée en C++. Pour implanter la BCOO, il faut (ces questions ne sont pas indépendantes) :

- Choisir un langage de programmation : C++,
- Définir comment implanter les spécifications du cadriciel : utilisation massive de classes paramétrées + architecture retenue,
- Définir les classes préprogrammées qui seront initialement présentes dans le cadriciel.

Le plan de cette partie est le suivant : outre les considérations techniques évoquées dans le paragraphe 6.1, nous présentons aussi les classes préprogrammées dans le paragraphe 6.2. Afin de mieux montrer les possibilités offertes par la BCOO, nous illustrons sa mise en œuvre sur deux scénarios possibles : la mise en œuvre d'une nouvelle méthode d'optimisation (§ 6.3) et la résolution d'un nouveau problème d'optimisation (§6.4).

6.1 Considérations techniques

Pour mettre en œuvre les spécifications de la BCOO, il faut pouvoir rendre indépendant les codes de certains objets. Par exemple, un algorithme d'optimisation doit pouvoir manipuler une "solution codée" sans savoir à quoi elle correspond ni comment elle est implantée. Pour cela, la plupart des langages objets proposent une fonctionnalité de "méthodes virtuelles". Cette fonctionnalité est largement répandue dans tous les langages et dans tous les formalismes dit "orientés objets".

Il y a trois grandes méthodes répandues aujourd'hui pour mettre en œuvre cette indépendance entre les classes : les méthodes virtuelles, les interfaces et les classes paramétrées.

Les méthodes virtuelles sont prises en charge par la totalité des langages de programmation et par les méthodes de modélisation orientées objet. Et la BCOO peut être complètement implantée à l'aide de classes virtuelles. Nous ne détaillerons pas ici cette implantation car nous n'avons pas fait ce choix technique pour des raisons de performances. En effet, chaque appel à une méthode virtuelle se traduit par une résolution de lien dynamique. Concrètement cela signifie que l'appel à une méthode virtuelle entraîne le processus suivant : le type de l'objet est déterminé, la liste des méthodes virtuelles (appelées table des méthodes virtuelles (VMT, Virtual Methods Table) est parcourue pour vérifier s'il existe une méthode virtuelle dans la table correspondant à ce type. Si ce n'est pas le cas, on cherche la classe mère de l'objet considéré et on recommence le processus. Ce processus est relativement coûteux en temps et est réalisé à chaque appel de méthode virtuelle. Cela pose un problème pour l'efficacité du code. Or en optimisation, il est important d'écrire des programmes efficaces, ce qui interdit cette édition de liens dynamique.

Il est aussi possible d'utiliser les interfaces pour rendre les classes indépendantes. Les interfaces sont connues dans les méthodes de modélisation orientées objet et elles sont implantées dans quelques langages dont le java. Pourtant, nous n'avons pas retenu cette solution technique car un appel à une méthode dans une interface met en œuvre une édition des liens.

Nous avons choisi d'utiliser les classes paramétrées. Ce choix limite quelque peu les langages de programmation que l'on peut utiliser car tous ne fournissent pas de facilités pour faire des classes paramétrées. En C++ (comme en ADA, Eiffel, les versions récentes de JAVA...), des mots clés spécifiques prévus par le langage permettent de créer des classes paramétrées.

Une classe paramétrée est une classe non complètement définie qui utilise des types non spécifiés. Une classe peut être paramétrée par certains attributs ou par certains paramètres de méthodes. Typiquement un vecteur peut être implanté comme une classe paramétrée. Un vecteur est un tableau redimensionnable au moment de l'exécution fournissant toutes les facilités pour l'ajout la suppression et la modification de ses éléments. Les éléments stockés dans le vecteur peuvent être de nature très différentes : des entiers, des nombres à virgules flottantes, des paires de nombres ... Grâce aux classes paramétrées, on peut écrire une classe vecteur dont les éléments sont de type T où T est le nom du type passé en paramètres. à l'intérieur du code de vecteur, on utilise le type T alors qu'on ne connaît explicitement ce type. C'est ensuite, au moment où l'on précise Vecteur d'entiers, que l'on sait que le type T est entier. C'est donc à ce moment que l'on connaît réellement les opérations de recopie, de tests d'égalité et toutes les opérations nécessaires à la gestion du vecteur.

Grâce aux classes paramétrées, nous avons proposé :

- une classe "Transformation" paramétrée par les "Solutions Codées", "Evaluation", "Solution", "Evaluation critère" et "Critère". Pour écrire une nouvelle transformation, il suffit donc de spécifier / modifier le paramètre correspondant, la classe transformation se crée alors automatiquement,
- une classe "Solution codée vecteur générique", cette classe est une solution codée paramétrée par le type des éléments qu'elle contient, nous avons aussi proposé les "Solution codée vecteur de vecteur générique" et "Solution codée matrice générique",
- les méthodes d'optimisation sont des classes paramétrées par le type de la transformation, ainsi une optimisation peut directement utiliser les types des éléments de transformation.

Un exemple d'utilisation typique des classes paramétrées pour la BCOO est celui d'une classe optimisation paramétrée par le type de la classe "Solution codée". Ainsi, lorsque l'algorithme a besoin de manipuler une solution codée, il en connaît déjà le type et peut manipuler directement l'instance. D'un point de vue technique, le programme ainsi généré est tout à fait identique au programme que l'on aurait écrit si l'on n'avait écrit que des classes dédiées. Autrement dit, cette solution technique a les avantages de la programmation orientée objets, mais reste complètement efficace.

6.2 Classes mises en œuvre dans le cadre de cette thèse

6.2.1 Projet

Nous avons mis en œuvre les spécialisations suivantes de la classe TCustomProjet qui permettent de charger les problèmes suivants :

- TJobshop : charge les données du problème de jobshop, peut aussi être utilisé pour le problème de flowshop
- TTSP : charge les données du problème de voyageur de commerce
- TJobshopTL : charge les données du problème de jobshop avec time lags,
- TJobshopT : charge les données du problème de jobshop avec transport.

6.2.2 Solution codée

Dans cette implantation, nous proposons des solutions codées contenant des éléments dont le type est paramétré. D'un point de vue technique, tout type peut être utilisé du moment qu'il dispose des fonctionnalités de construction par défaut et recopie.

- TSolCodVecteur<T> : décrit un vecteur d'éléments de type T,
- TSolCodMatrice<T> : décrit une matrice d'éléments de type T,
- TSolCodVecVecteur<T> : décrit un vecteur de vecteur d'éléments de type T (contrairement à une matrice, un vecteur de vecteur n'est pas rectangulaire, mais chaque élément du grand vecteur peut contenir des vecteurs de taille différente).

6.2.3 Solution

Les solutions proposées sont des classes paramétrées par le type des éléments que les structures de données contiennent. D'un point de vue technique, tout type peut être utilisé du moment qu'il dispose des fonctionnalités de construction par défaut et de recopie.

- TSolVecteur<T> : décrit un vecteur d'éléments de type T,
- TSolMatrice<T> : décrit une matrice d'éléments de type T,
- TSolVecVecteur<T> : décrit un vecteur de vecteur d'éléments de type T (contrairement à une matrice, un vecteur de vecteur n'est pas rectangulaire, mais chaque élément du grand vecteur peut contenir des vecteurs de taille différente).

6.2.4 Critère

Une seule classe de critère est proposée, mais comme elle est paramétrée, elle permet de prendre en compte la plupart des cas.

- TCritereScalaire<T> : décrit un critère scalaire dont le type est T (T est donc soit float, double, int, long, …)

6.2.5 Évaluation de critère

Une seule évaluation de critère a été proposée : la classe TEvalCritMaxValeur<T, F>. Celle-ci consiste à rechercher la plus grande valeur dans une solution. Cette classe est déclinée pour chacun des trois types de solutions, et prend en paramètre T le type de l'élément stocké dans la solution ainsi qu'un foncteur F décrivant la comparaison de deux éléments.

6.2.6 Voisinage

Les voisinages suivants sont disponibles dans notre implantation de la BCOO :

- TVoisSolCodVecteurPermu : permutation de deux éléments dans une solution codée sous la forme d'un vecteur (i.e. classe TSolCodVecteur),
- TVoisSolCodVecVecteurPermu : permutation de deux éléments dans une solution codée sous la forme d'un vecteur de vecteur (i.e. classe TSolCodVecVecteur),
- TVoisSolCodMatricePermu : permutation de deux éléments dans une solution codée sous la forme d'une matrice (i.e. classe TSolCodMatrice),
- TVoisUnion<T1,T2> : Classe paramétrée créant un voisinage stockant union des voisinages T1 et T2.
- TVoisPuiss<T,n> : Classe paramétrée créant un voisinage puissance du voisinage T. Le voisinage créé est T^n.

6.2.7 Optimisation

Nous avons programmé de manière générique les algorithmes d'optimisation suivant :

- TDescenteStoch : descente stochastique,
- TDescenteDert : descente déterministe,
- TRecuitSimule : algorithme du recuit simulé,
- TKangourou : algorithme du kangourou,
- TTaboo : algorithme tabou de Nowicki et Smutnicki (allégé des parties spécifiques).

Cette liste, comme les précédentes, n'est pas exhaustive et d'autres classes peuvent être introduites. Nous n'avons rencontré aucune limitation à l'utilisation du cadriciel et tout algorithme est envisageable. En particulier, les algorithmes génétiques sont la prochaine étape.

6.3 Application 1 : Mise en œuvre d'une nouvelle méthode

Pour illustrer l'utilisation de la BCOO, nous nous plaçons dans le cas où l'on souhaite implanter une nouvelle méthode d'optimisation. L'exemple que nous allons présenter concerne l'algorithme de la descente stochastique. Nous allons programmer cet algorithme pour qu'il soit générique (i.e. non spécifique au problème). Concrètement, cela signifie que l'algorithme n'aura connaissance d'aucune classe du paquetage "Spécifique au Problème".

6.3.1 Prérequis

Pour implanter une nouvelle méthode itérative comme l'algorithme de la descente stochastique, il est nécessaire de disposer de voisinages et d'une transformation dont la solution codée est compatible. Dans la page suivante, il est rappelé et mis en opposition avec sa traduction en BCOO.

6.3.2 Classes préprogrammées utilisées

Aucune classe préprogrammée n'est utilisée pour mettre en œuvre cet algorithme. Ce n'est qu'au moment de l'utilisation de cet algorithme que l'on utilisera des classes préprogrammées.

6.3.3 Mise en œuvre de l'algorithme

L'algorithme de la descente stochastique est un algorithme itératif à base de voisinage. Il est donc nécessaire pour son implantation de lui passer en paramètres le type de la transformation utilisée mais aussi le type du voisinage utilisé. Ainsi, la classe TDS est paramétrée par le type de la transformation TTransformation et par le type du voisinage TVoisinage. On note ainsi TDS< TTransformation, TVoisinage>.

L'algorithme 5-2 détaille l'implantation de la descente stochastique en BCOO. La première ligne consiste à créer une instance de la classe voisinage. La seconde initialise le compteur d'itérations. Les

cinq lignes suivantes créent les éléments de transformation nécessaires. Dans l'algorithme de principe (algorithme 5-1), deux variables (*x* et *y*) sont utilisées. Dans l'algorithme de la BCOO, nous avons donc aussi besoin de deux solutions codées *scx* et *scy*. Mais, par soucis d'optimisation du code, nous n'utilisons qu'une seule solution codée (*scx*). Cette solution codée est initialisée à la solution de départ, puis à chaque itération, le voisinage modifie directement la solution codée. Si la solution codée générée est améliorante, elle est conservée telle quelle. Si elle n'est pas améliorante, l'algorithme de la descente la refuse et on redéfait le voisinage (méthode *undo* du voisinage).

```
Entree :
   x : solution de départ,
   cx : cout de x

Tant que nécessaire faire

   y=V(x)

   cy=c(y)
   si cx ≥ cy alors
         cx=cy
         x=y
   finsi
FinTantque
```

Algorithme 5-1. Algorithme de principe de la descente stochastique

```
pV        = new TVoisinage(this);
int       count=0;
TSC       scX(_scX);
TSolution sX(_sX);
TCritere  cX(_cX);
TSolution sY(_sX);
TCritere  cY(_cX);
while(count<_max_iter)
{
   pV->stochastique(scX,sX)
   evaluation->eval(scX,sy);
   evalCritere->eval(sy,cy);
   si(cx ≥ cy)
   {
         cx = cy;
         Echange sX et sY;
         pV->confirme();
   }
   else
         pV->undo(scx);
   ++count;
}
```

Algorithme 5-2. Algorithme de la descente stochastique en BCOO

Les éléments de transformation à créer sont donc :

- la solution codée *scX*,
- la solution associée à *x* appelée *sX*,
- la valeur du critère pour *x* appelée *cX*,
- la solution associée à *y* (i.e. à *x* une fois qu'il a été modifié par le voisinage) appelé *sY*,
- la valeur de critère associé à *y* appelé *cY*.

Ensuite, la boucle principale comme par l'application du voisinage. Celui-ci consiste à partir de la solution codée scX et de la solution sX. Le voisinage pV modifie la solution codée scX. La variable scX contient donc alors le voisin nommé y dans l'algorithme de principe. Ensuite, la solution et le critère sont évalués. Si le critère de x est meilleur que le critère de y, alors la solution est acceptée. Pour cela, on recopie le critère cy dans cx et la solution sY dans sX. Ces deux recopies peuvent être évitées par une habile utilisation de pointeurs. La recopie théorique de sY dans sX s'implante alors comme un échange des valeurs des pointeurs. Le voisinage est alors informé que le voisin est accepté. Sinon, la solution y est rejetée. Pour cela, il faut donc transformer scX pour qu'il retrouve sa valeur originale. C'est le rôle de la méthode undo du voisinage.

6.3.4 Exemples d'utilisation

Pour utiliser l'algorithme nouvellement créé sur un nouveau problème, il suffit de créer deux classes. La première est la classe Projet qui permet de stocker les informations liées au problème étudié et la seconde est une instance de TTransformation qui détermine la manière dont on code une solution et dont on la traduit en valeur de critère.

Nous prenons donc exemples d'utilisation. Le premier exemple consiste à appliquer l'algorithme du kangourou à l'instance ft06 du problème de jobshop. Pour cela, il suffit d'implanter le code suivant.

- La première ligne consiste à créer un nouveau projet, lié au problème de jobshop, et dont on garde une référence générale cp.
- La deuxième consiste à déclarer TMyTransformation, une transformation utilisée pour le problème de jobshop.
- La ligne 3 crée une solution codée, calculée par la classe cp. Cette solution servira d'initialisation au programme.
- La ligne 4 crée une instance de la transformation. Pour ce faire, on crée une instance de solution codée, une instance de solution, …
- La ligne 5 crée l'algorithme d'optimisation basé sur la transformation et sur deux voisinages : le voisinage de permutation de deux éléments, et le même voisinage élevé à la puissance 3.
- La ligne 6 charge une instance de problème de jobshop.
- La ligne 7 lance l'optimisation

```
1) TCustomProject *cp=new TJobshop();
2) typedef TTransformation<TSolCodVecteur<int>, TEvaluationJS,
              TSolVecteur<int>, TEvalCritMaxVal<int>, TCritScalaire<int> >
              TMyTransformation;
3) TSolCodVecteur<int> *sol=cp->GetNewSol();
4) TMyTransformation::GetInstance(cp,
          sol, new TEvaluationJS(), new TSolVecteur<int>(),
          new TEvalCritMaxVal<int>(), new TCritScalaire<int>());
5) TCustomOptimisation *co =
     new TKangourou<TMyTransformation, TVoisSolCodVecteurPermu,
      TVoisPuissance<TVoisSolCodVecteurPermu,3> >();
6)cp->load("ft06");
7)co->optimize();
```

Algorithme 5-3.Utilisation de la BCOO pour un nouvel algorithme

Si l'on souhaite lancer un problème différent, comme le problème de voyageur de commerce (TSP) il suffit d'utiliser le projet ainsi que l'évaluation adéquats. Le code obtenu est détaillé ci-dessous, les seules modifications au programme sont soulignées. Les modifications concernent le nom du projet qui désigne maintenant le projet du TSP ainsi que les éléments de transformation modifiés :

```
1) TCustomProject *cp=new TTSP();
2) typedef TTransformation<TSolCodVecteur<int>, TEvaluationTSP,
          TCritScalaire<int>, TRecopie<int>, TCritScalaire<int> >
              TMyTransformation;
3) TSolCodVecteur<int> *sol=cp->GetNewSol();
4) TMyTransformation::GetInstance(cp,
          sol, new TEvaluationTSP (), new TCritScalaire<int>(),
          new TRecopie<int>(), new TCritScalaire<int>());
5) TCustomOptimisation *co =
     new TKangourou<TMyTransformation, TVoisSolCodVecteurPermu,
      TVoisPuissance<TVoisSolCodVecteurPermu,3> >();
6)cp->load("ft06");
7)co->optimize();
```

6.4 Application 2 : Résolution d'un nouveau problème

Dans ce paragraphe, on envisage le cas où l'on dispose d'un nouveau problème et où l'on souhaite tester son optimisation. Nous allons détailler l'exemple du problème de flowshop de permutation. Pour ce nouveau problème, nous allons décrire les nouvelles classes qui doivent être réécrites afin de le mettre en œuvre ainsi que les méthodes d'optimisation que l'on peut tester sur ce problème.

6.4.1 Classes préprogrammées utilisées

Le problème de flowshop est un problème disjonctif comme le problème de flowshop, mais contrairement au problème de jobshop, toutes les pièces d'un problème de flowshop passe par la même

séquence de machines. Ainsi, pour décrire une solution du problème de flowshop de permutation, il suffit de décrire l'ordre dans lequel les opérations sont traitées sur la première machine. Une solution codée de ce problème est donc un vecteur d'opération. Nous utilisons la classe préprogrammée TSolCodVecteur<int> pour stocker une solution codée.

Une solution du problème de flowshop est un ordonnancement décrit par les dates de fin des opérations. Nous utilisons donc comme solution codée, un vecteur de date de fin (TSolVecteur<int>). Cette classe est aussi une classe préprogrammée du paquetage "BCOO_Classes de base".

Le critère à optimiser est le makespan, i.e. la date de fin de traitement de la dernière opération traitée. Ce critère est donc un entier, que l'on peut mettre en œuvre à l'aide de la classe TCritereScalaire<int>.

L'évaluation d'une solution consiste à rechercher la plus grande date de fin d'une opération. Pour cela, on utilise l'évaluation de crièttre préprogrammée TEvalCritMaxValeur<int, F> où F est un foncteur comparant deux entiers.

6.4.2 Classes à mettre en œuvre

Les classes qu'il reste à mettre en œuvre sont donc :

- la classe TProjetFS, spécialisation de TCustomProjet, qui reste indispensable pour pouvoir stocker les informations du problème de flowshop de permutation comme : le nombre de jobs n, le nombre de machines m et le temps de traitement de chaque job j sur chaque machine k (tps[j,k]),
- et une classe TEvaluationFS qui est une classe évaluation et qui décrit comment passer d'une solution codée à une solution (i.e. passer d'un ordre des opérations en entrée aux dates de début des opérations).

L'algorithme d'évaluation qui permet de passer d'une solution codée à une solution est un algorithme connu dans la littérature. Cet algorithme est décrit dans l'algorithme 5-4. Les entrées de cet algorithme sont le tableau SC dont un élément SC[i] est la ième opération entrée dans le système, TPS[i,j] est la durée de l'opération i sur la machine j. En sortie, le programme fournit S[i], la date de fin de traitement de la dernière opération traitée sur la machine i.

L'algorithme 5-5 présente la méthode de l'algorithme d'évaluation, elle possède deux paramétres : La solution codée SC et la solution S. Les deux premières lignes de la méthode consiste à récupérer le projet, ce qui est possible car cette évaluation est dédiée au problème de flowshop avec permutation, elle connaît donc la classe projet. La seconde ligne consiste à récupérer les données (le tableau TPS) du projet TFSP. Ensuite, l'algorithme est traduit tel quel en C++.

```
Entree : SC, TPS

Sortie : S

Début :

S[1]:= TPS[SC[1],1];

Pour i de 2 à m faire

   S[i]=S[i-1]+TPS[SC[1],i];

finpour;

Pour j de 1 à n faire

   S[1]= S[1]+TPS[SC[j],1];

   Pour i de 2 à m faire

         S[i]=max(S[i-
1],S[i])+TPS[SC[j],i];

   finpour;

finpour;
```

Algorithme 5-4. Algorithme de principe pour l'évaluation du flowshop

```
void     TEvalFSP::evaluation(TSC     *SC,
TSolution *S)

{

   TFSP *fsp=GetProjet();

   TPS=fsp->GetTPS();



   S[1]:= TPS[SC[1],1];

   for(int i=2;i≤m;++i)

         S[i]=S[i-1]+TPS[SC[1],i];

   for(int j=1;j≤n;++j)

   {

   S[1]= S[1]+TPS[SC[j],1];

   for(int i=2;i≤m;++i)

         S[i]=max(S[i-
1],S[i])+TPS[SC[j],i];

   }

}
```

Algorithme 5-5. Évaluation en BCOO d'un vecteur d'opération pour le flowshop de permutation

6.4.3 Algorithmes d'optimisation possibles

Maintenant que l'algorithme d'évaluation est construit, on peut lancer diverses optimisations. Pour cela, il suffit d'exécuter le code suivant :

```
1) TCustomProject *cp=new TJobshop();
2) typedef TTransformation<TSolCodVecteur<int>, TEvalFSP,
              TSolVecteur<int>, TEvalCritMaxVal<int>, TCritScalaire<int> >
              TMyTransformation;
3)cp->load("ft06");
4) TSolCodVecteur<int> *sol=cp->GetNewSol();
5) TMyTransformation::GetInstance(cp,
          sol, new TEvaluationJS(), new TSolVecteur<int>(),
          new TEvalCritMaxVal<int>(), new TCritScalaire<int>());
6) TCustomOptimisation *co =
     new DS<TMyTransformation, TVoisSolCodVecteurPermu, >();
7)co->optimize();
```

Algorithme 5-6. Descente stochastique sur le flowshop de permutation

Cet algorithme est élémentaire, chacune des lignes correspond à un paramétrage de l'optimisation. Sur la première ligne, on choisit le projet que l'on utilise. Sur la deuxième, c'est la transformation que l'on définit, si l'on souhaite changer de solution codée ou changer d'évaluation, c'est cette ligne qu'il faut modifier. Sur la troisième ligne, on charge une instance du problème. Sur la ligne 4, on crée une nouvelle solution codée. Cette solution peut être éventuellement modifiée, elle sert de solution de départ aux algorithmes d'optimisation. La cinquième ligne initialise la transformation avec la solution codée. La sixième crée l'optimisation choisie. La septième et dernière ligne lance l'optimisation à proprement parler.

Afin de lancer un algorithme du kangourou sur le problème, il suffit de remplacer la ligne 5 par :

```
5) TCustomOptimisation *co =
     new TKangourou<TMyTransformation, TVoisSolCodVecteurPermu,
      TVoisPuissance<TVoisSolCodVecteurPermu,3> >();
```

Afin de lancer un algorithme du kangourou sur le problème, il suffit de remplacer la ligne 5 par :

```
5) TCustomOptimisation *co =
     new TKangourou<TMyTransformation, TVoisSolCodVecteurPermu,
      TVoisPuissance<TVoisSolCodVecteurPermu,3> >();
```

6.5 Application 3: réalisation d'une interface

Nous avons mis en œuvre le module d'affichage préconisé par la BCOO. Cette mise en œuvre a été réalisée en VCL. La VCL est un cadriciel de construction d'interfaces graphiques proposé par Borland. Complètement intégré à l'environnement de développement (IDE), la VCL permet d'utiliser ses composants graphiques afin de construire des interfaces. Elle permet aussi aux développeurs de créer leurs propres composants et de venir les intégrer à l'environnement. Nous avons donc créé nos propres composants graphiques basés sur les préconisations de la BCOO. Principalement, un composant d'affichage d'un diagramme de Gantt a été proposé. Des composants de paramétrage ont aussi été ajoutés tels qu'une liste des machines possibles ou une liste de jobs.

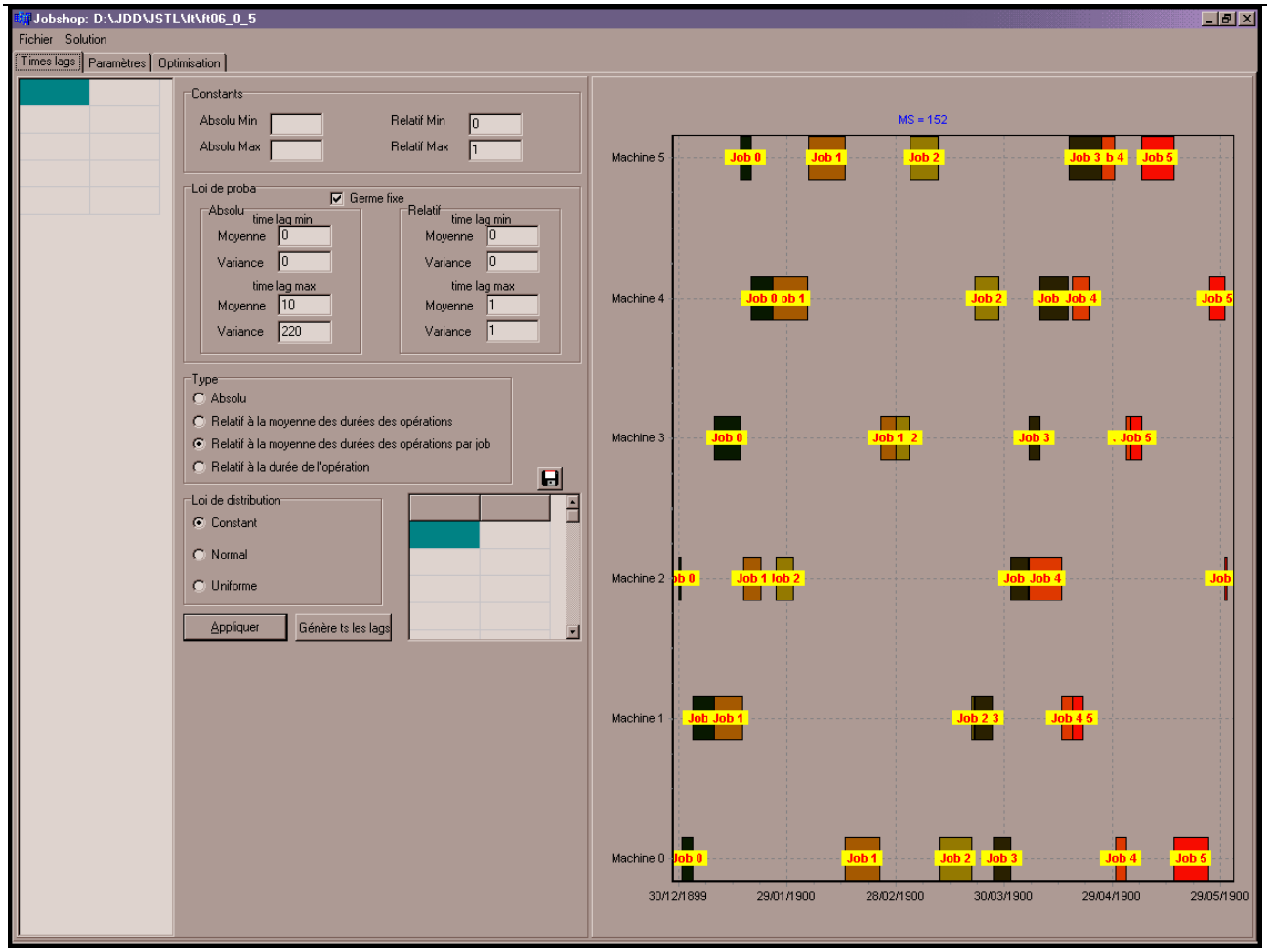

Figure 5-14. Exemple de mise en œuvre du module affichage graphique

Dans la suite, nous montrons un exemple de mise en œuvre de la BCOO sur l'algorithme tabou proposé à MOSIM. La première figure (cf. figure 5-14) montre un exemple de mise en œuvre du module d'affichage. La partie gauche de l'interface est complètement dédiée au problème étudié (i.e. au projet de jobshop avec time lags TJSTL). La partie droite est un diagramme de Gantt générique, créé à partir des informations fournies au hub. La figure 5-15 montre une mise en œuvre de paramètres génériques. Ces paramètres n'ont pas fait l'objet d'une présentation car ils sont très dédiés à l'implantation. Le but de ces paramètres génériques est de permettre au développeur d'algorithme d'optimisation ou de toute classe de la BCOO d'avoir accès à des valeurs de paramètres chargée par l'interface à l'initialisation sans que l'interface ne connaisse à l'avance les paramètres qu'elle charge ni les paramètres qui lui seront demandés. Ainsi, l'interface graphique montrée dans cette figure est construite de manière complètement dynamique à partir des informations lues dans le fichier. La figure 5-16 montre une interface de suivi de l'optimisation. Cette interface permet à l'utilisateur de suivre l'évolution du ou des meilleurs critères. Il est à noter que l'implantation proposée de la BCOO ne nécessite pas d'interface graphique, mais la possibilité est laissée à l'utilisateur de compléter des méthodes liées (bound methods) pour spécifier des mises à jour. Ainsi, au début de l'optimisation, une méthode est passée en paramètres à l'algorithme d'optimisation. Cette méthode est appelée à chaque mise à jour de critère. Elle a donc tout loisir de faire la mise à jour de l'interface graphique, qu'elle soit dédiée ou générique.

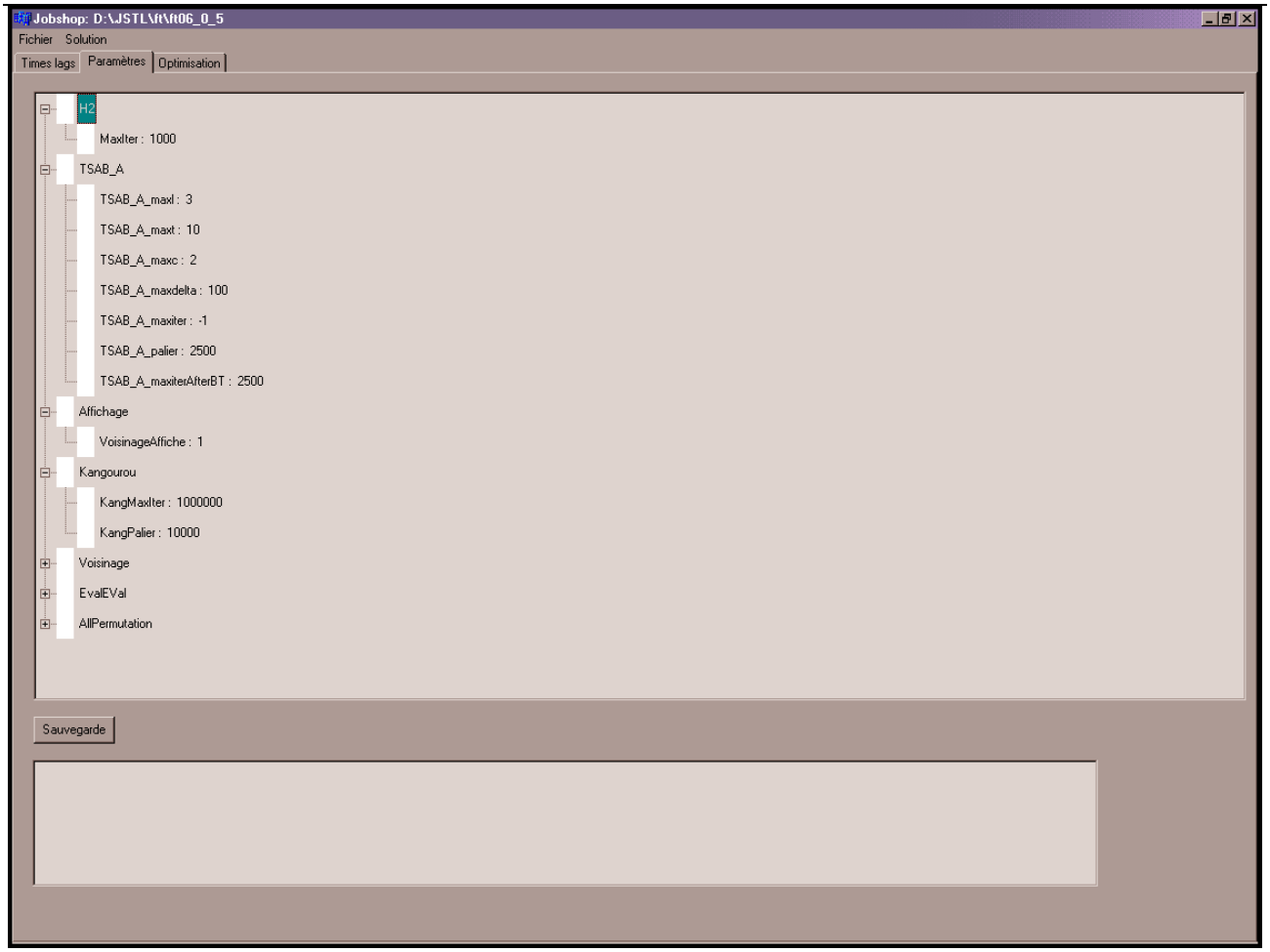

Figure 5-15. Exemple de mise en œuvre de paramètres génériques

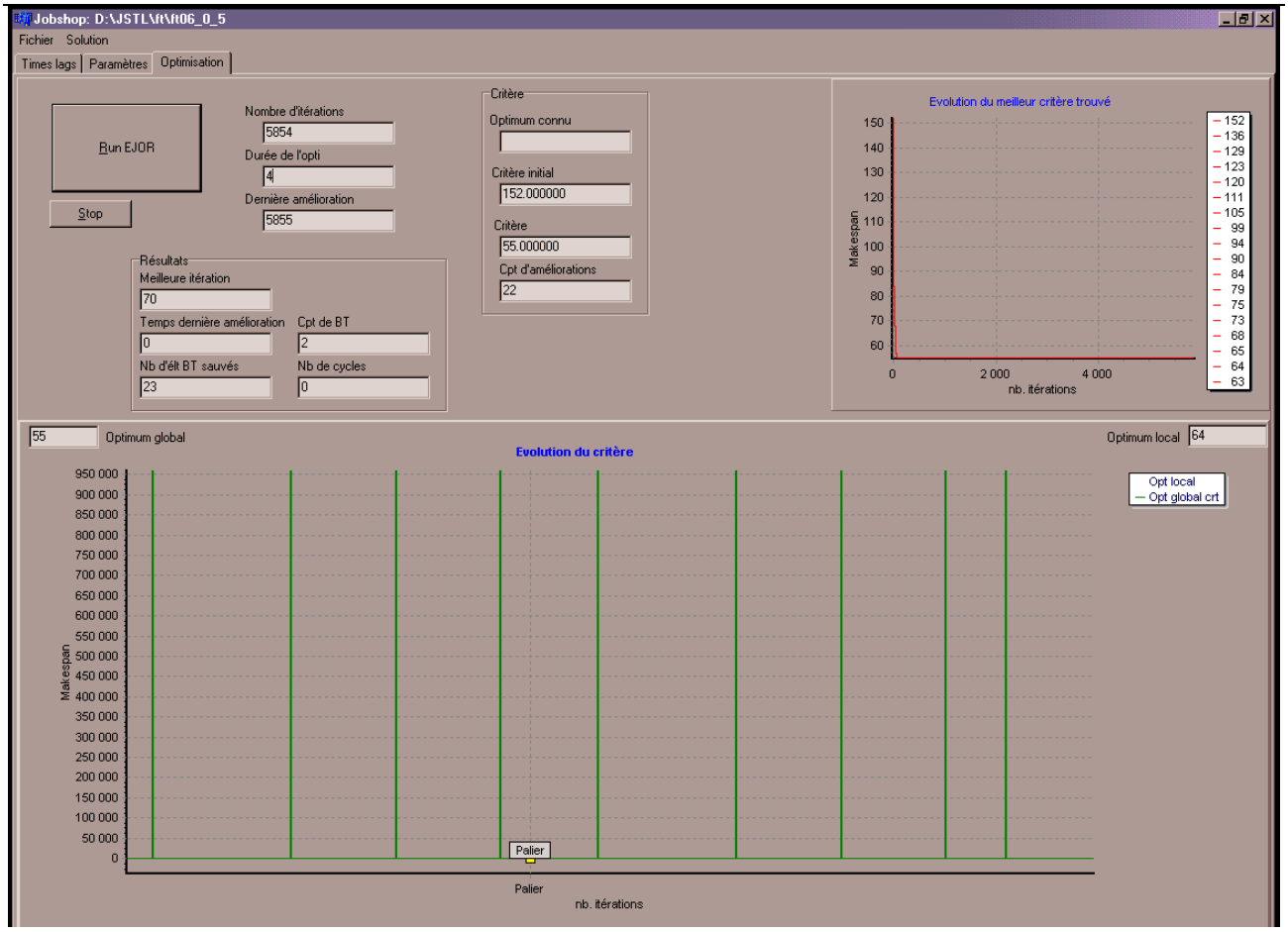

Figure 5-16. Exemple de mise en œuvre d'affichage des résultats d'optimisation.

7 Conclusion sur le cadriciel

La BCOO aide à la construction d'un modèle d'optimisation dans les étapes de conception et d'implantation. Il fournit :

- une architecture générale et indépendante de la plateforme pour la construction du logiciel,
- des propositions pour l'implantation dans de nombreux langages,
- des classes préprogrammées de l'architecture,
- des algorithmes d'optimisation généraux (qui ne contiennent pas de connaissance spécifique du problème)
- des parties d'algorithmes d'optimisation dédiés comme des voisinages, des structures de données, ...
- des outils pour la mise en œuvre de nouveaux algorithmes.

Comme nous l'avons illustré à travers des exemples de mise en œuvre, la BCOO permet au développeur de ne programmer que la partie juste nécessaire à ses développements. Cela a de nombreux avantages :

- réduire le temps de développement pour les nouvelles applications,
- permettre à tout développeur de stocker dans une bibliothèque les algorithmes génériques qu'il a développé afin de les rendre disponibles,
- augmenter la réutilisation du code et permettre ainsi de le fiabiliser,
- enrichir les interfaces graphiques, car toute nouvelle application peut réutiliser simplement les développements antérieurs en terme d'interface graphique,
- permettre l'utilisation directe, et avec une très faible quantité de code, d'interfaces graphiques génériques.

La plupart des algorithmes présentés dans cette thèse ont été programmés en BCOO. Cela nous a permis de valider tous ces concepts et de ne constater aucune limitation à l'utilisation de la BCOO. Ce point délicat, fait quelques fois défaut aux cadriciels.

Nous souhaitons maintenant développer la BCOO en proposant d'enrichir les bibliothèques de classes préprogrammées en incluant des algorithmes génétiques et d'autres problèmes de la littérature.

Conclusion générale

Dans ce mémoire de thèse, nous nous sommes intéressés aux problèmes d'optimisation des systèmes de production. Pour traiter ces problèmes, nous avons mis en place des démarches de modélisation et d'optimisation. Ces démarches complètent les démarches existantes et proposent en plus d'aider à la conception de logiciels d'optimisation. Au travers de cette thèse, nous avons appliqué ces démarches à deux problèmes industriels : le problème d'ordonnancement des ateliers de forge à chaud et le problème d'ordonnancement des systèmes flexibles de production. Ces problèmes ont peu de choses en commun si ce n'est le modèle théorique auquel ils se rattachent. En effet, par hypothèses simplificatrices successives, la plupart des problèmes industriels se ramènent à un modèle théorique de la littérature. Dans cette thèse, nous avons choisi le problème de jobshop comme modèle théorique de base pour tous les problèmes que nous avons étudiés.

Après avoir choisi un modèle théorique, notre démarche consiste à identifier les hypothèses simplificatrices à poser pour se ramener à ce problème. Puis, nous choisissons un algorithme efficace pour la résolution du modèle théorique. Cet algorithme est adapté progressivement jusqu'à prendre en compte toutes les contraintes nécessaires. Pour ce faire, nous proposons de supprimer itérativement les hypothèses simplificatrices. À chaque hypothèse simplificatrice supprimée correspond une contrainte ajoutée dans l'expression du problème ou, à défaut, à une contrainte modifiée. À la fin de ce processus, on obtient donc un algorithme d'optimisation résolvant complètement le problème étudié. C'est grâce à cette démarche que nous avons adapté les algorithmes d'optimisation dédiés au problème de jobshop à des problèmes plus complexes.

Durant cette thèse, il est apparu que de nombreux algorithmes d'optimisation efficaces pour le problème de jobshop utilisent un ensemble d'outils basés sur le graphe conjonctif - disjonctif. Ces outils passent par l'utilisation du graphe conjonctif - disjonctif, du graphe conjonctif et de l'exploitation du chemin critique en particulier à travers les voisinages. L'intérêt de la démarche est de guider l'adaptation de ces outils pour tenter de les utiliser sur des problèmes plus complexes. À travers cette thèse, nous avons montré comment prendre en compte de nouvelles contraintes dans le graphe conjonctif - disjonctif. Les types de contraintes ajoutées ont été variés : contraintes sur l'ordre des opérations, contraintes modifiant la longueur de certains arcs, contraintes ajoutant des arcs de longueur positive (qui créent ou non des cycles) et des arcs de longueur positive ou négative (qui créent ou non des cycles absorbants). Grâce à notre démarche, nous avons pu adapter ces outils à des problèmes plus complexes. Nous avons ainsi proposé des adaptations du graphe conjonctif - disjonctif pour le problème de jobshop avec time lags et pour le problème de jobshop avec transport.

L'application de cette démarche implique l'écriture et la mise au point de nombreux algorithmes d'optimisation. À chaque itération, une nouvelle contrainte est ajoutée et l'algorithme d'optimisation doit donc être adapté afin de prendre en compte cette contrainte. En général, les contraintes ajoutées ne remettent pas en cause tout l'algorithme mais seulement une partie. Afin de limiter l'effort de programmation nécessaire à l'adaptation du nouvel algorithme et pour éviter la duplication de code, nous avons proposé un cadriciel orienté objets pour l'optimisation. Ce cadriciel repose sur une architecture particulière qui permet de séparer les différents composants intervenant dans un logiciel d'optimisation. Outre l'identification de ces composants, le cadriciel fournit aussi des modalités de communication entre eux. Les principaux composants identifiés dans tous les algorithmes d'optimisation sont : une solution codée qui permet de stocker une solution en mémoire ou sur disque dur (c'est l'ensemble des valeurs de cette solution qui forme l'espace de recherche), une solution qui décrit complètement une solution, un algorithme d'évaluation qui décrit comment passer d'une solution codée à une solution, un critère qui contient l'évaluation de la qualité d'une solution et un algorithme d'évaluation du critère qui décrit comment calculer le critère à partir de la solution. L'utilisation du cadriciel permet :

- de limiter la quantité de code à écrire pour développer un nouvel algorithme d'optimisation,
- de proposer un ensemble de composants préprogrammés,

Le premier problème que nous avons traité est le problème de jobshop avec time lags. Pour ce problème, nous avons proposé les premiers algorithmes d'optimisation qui fournissent forcément une solution réalisable. Outre les algorithmes d'optimisation que nous avons proposés, nous avons surtout proposé une adaptation du modèle de graphe conjonctif - disjonctif ainsi que les modifications qui en découlent. Ainsi, nous avons adapté le graphe disjonctif-conjonctif pour prendre en compte ces contraintes, mais nous avons aussi proposé un algorithme de plus long chemin très efficace adapté à la forme particulière du graphe.

Le second problème que nous avons traité est le problème de jobshop avec transport et contraintes supplémentaires. Ce problème apparaît, par exemple, comme un sous problème du problème d'ordonnancement des systèmes flexibles de production. Pour ce problème, nous avons proposé la première modélisation linéaire en nombres entiers et les premiers résultats optimaux. De plus, notre démarche d'optimisation nous a permis d'adapter le modèle de graphe disjonctif-conjonctif et les outils associés de manière à les mettre en œuvre sur des algorithmes itératifs.

Nos propositions forment donc un ensemble cohérent allant de la démarche générale pour l'optimisation jusqu'à sa mise en œuvre sur les outils de graphe disjonctif-conjonctif en passant par une solution logicielle facilitant sa mise en œuvre. Nous avons testé cette approche sur le problème de jobshop avec time lags, sur le problème de jobshop avec transport et contraintes additionnelles mais aussi sur des problèmes industriels.

Les perspectives que nous souhaitons donner à ce travail concernent :

- la proposition d'algorithmes itératifs ne générant que des solutions réalisables pour le problème de jobshop avec time lags,
- la proposition de voisinage basé sur le chemin critique pour le problème de jobshop avec transport et contraintes additionnelles,
- le déploiement de la BCOO : création de délivrables, documentation, site Web de présentation, …

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.