ACADÉMIE D'AIX-MARSEILLE
UNIVERSITÉ D'AVIGNON ET DES PAYS DE VAUCLUSE

THÈSE

Méthodes Hybrides de
Programmation par Contraintes et Programmation Linéaire
pour le
Problème d'Ordonnancement de Projet à Contraintes de Ressources

Soutenue publiquement le 18 décembre 2003 pour l'obtention du grade de
Docteur en Sciences de l'Université d'Avignon et des Pays de Vaucluse

SPÉCIALITÉ : Informatique

Sophie Demassey

Composition du jury :

M.	Philippe Baptiste	CR Cnrs, Lix, Paris	Rapporteur
M.	Jacques Carlier	PR, Heudiasyc, Utc, Compiègne	Président
M.	Claude Le Pape	Ilog s.a., Gentilly	Examinateur
M.	Maurice Queyranne	PR, Ubc, Vancouver	Rapporteur
M.	Alain Quillot	PR, Limos, Clermont-Ferrand	Examinateur
M.	Christian Artigues	MdC, Lia, Avignon	Co-directeur de thèse
M.	Philippe Michelon	PR, Lia, Avignon	Co-directeur de thèse

Laboratoire Informatique d'Avignon

Remerciements

Tous mes remerciements vont, pour le très grand honneur qu'ils m'ont fait de rapporter cette thèse, à Monsieur Philippe Baptiste, chercheur CNRS (LIX) et professeur associé à l'École Polytechnique, et à Monsieur Maurice Queyranne, professeur à l'Université de Vancouver (UBC) et chercheur invité au Laboratoire Leibniz (IMAG) de Grenoble.

J'adresse autant de remerciements à Monsieur Jacques Carlier, professeur à l'Université de Technologie de Compiègne, Monsieur Claude Le Pape, du Laboratoire R&D de la société ILOG, et Monsieur Alain Quillot, professeur à l'Université de Clermont-Ferrand II, d'avoir accepté de participer au jury.

Si la rédaction d'un mémoire de thèse n'est pas un long fleuve tranquille, elle me fut rendue plus sereine, en aval, par Claude Le Pape pour sa lecture méthodique et ses corrections avisées. Je l'en remercie doublement. En amont, Philippe Baptiste a été l'initiateur d'une partie des travaux présentés ci-après. Je n'aurais pu souhaiter plus forte et plaisante motivation que la collaboration qu'il m'a offerte et je lui en suis grandement reconnaissante.

Merci aux matelots et éclusiers du Laboratoire Informatique d'Avignon et merci à son capitaine, Monsieur Renato De Mori : grâçe à eux, j'ai navigué durant quatre ans sous les meilleurs auspices. Je n'oublie pas Cristian Oliva et Serigne Gueye, partis, depuis, croiser le long de rivages exotiques...

Merci à Philippe Michelon de m'avoir encadré pendant mon doctorat de manière idéale, sachant répondre exactement à mes besoins d'assistance ou d'autonomie.

Pour finir, mille mercis à Christian Artigues pour sa disponibilité et son soutien permanent. Son dynamisme et ses encouragements m'ont permis de garder le cap. Nous aurons bien l'occasion d'explorer d'autres horizons !

Sommaire

Introduction

L'intelligence artificielle et la recherche opérationnelle s'intéressent à la résolution de problèmes combinatoires. Ces deux disciplines ont développé un cadre différent pour aborder ce type de problèmes, puis des techniques de résolution de plus en plus évoluées, adaptées à chacun de ces cadres. Les techniques de l'intelligence artificielle sont basées sur le raisonnement logique formelle, quand les techniques de la recherche opérationnelle adoptent un mode plus calculatoire.

La *programmation logique* de l'intelligence artificielle s'est enrichie au contact de la recherche opérationnelle en incorporant, par exemple, des principes de la théorie des graphes, de manière à traiter des problèmes numériques plus complexes. Elle s'est étendue ainsi à la *programmation (logique) par contraintes* (PPC). Les techniques de programmation par contraintes permettent de déterminer la réalisabilité d'un problème combinatoire en recherchant une solution satisfaisant chacune des contraintes du problème.

La recherche opérationnelle s'occupe principalement de résoudre des problèmes d'optimisation combinatoire au moyen de techniques souvent conjuguées à un formalisme précis du problème. En particulier, les techniques de *programmation linéaire en nombres entiers* (PLNE) sont étroitement liées à la modélisation des solutions du problème par des vecteurs d'entiers contraints de satisfaire des inégalités linéaires. Malgré cette restriction, la PLNE permet de modéliser une grande diversité de problèmes d'optimisation et les techniques de résolution associées à ce formalisme particulier sont d'autant plus raffinées et efficaces.

La distinction entre problème de décision et problème d'optimisation est en fait artificielle, et les deux approches PPC et PLNE s'appliquent indifféremment à l'un ou à l'autre. Elles présentent même un schéma général commun de résolution, l'énumération implicite des solutions, mais une démarche différente qui les rend complémentaires. Très récemment, plusieurs études ont été entreprises pour coupler les avantages des deux approches dans la conception de méthodes de résolution de problèmes d'optimisation combinatoire complexes.

L'intégration du raisonnement logique et des méthodes spécifiques de recherche opérationnelle est une idée plus ancienne, voire fondatrice, de l'ordonnancement. De nombreux problèmes d'ordonnancement offrent ainsi un cadre d'application des méthodes hybrides, du fait de leur forte complexité combinatoire, mais aussi de la quantité de travaux qui leur a été consacrée dans chacune des deux approches. L'*ordonnancement de projet à contraintes de ressources* ou *RCPSP* appartient à cette catégorie de problèmes. Il s'agit d'un problème général qui recouvre une grande variété d'autres problèmes tout aussi réputés dans la littérature.

Le travail présenté dans ce mémoire se situe à la jonction des trois domaines : programmation par contraintes, programmation linéaire en nombres entiers et ordonnancement. Plus précisément, nous nous intéressons aux méthodes hybrides PPC/PLNE et à leur application à la résolution optimale du RCPSP. Nous proposons ainsi deux nouvelles méthodes d'évaluation par défaut pour le RCPSP basées sur l'hybridation de la programmation par contraintes et de la programmation linéaire. La première est basée sur l'utilisation conjointe de techniques de propagation de contraintes

et d'une relaxation lagrangienne originale du problème. La seconde repose sur la linéarisation des règles ou bien des inférences de la programmation par contraintes pour la génération de coupes ajoutées à des relaxations continues du RCPSP. Nous mettons aussi en évidence le principe de coopération sur lequel est fondé *resolution search*, une méthode originale et peu connue, présentée dans [Chvátal 1997], pour la résolution exacte de problèmes de décision ou d'optimisation en variables binaires. Nous proposons des améliorations de cette procédure pour favoriser la coopération sous-jacente, une étude comparative avec une méthode arborescente classique sur une formulation linéaire du RCPSP, ainsi que des pistes pour une application plus avancée au RCPSP.

Nous débutons ce mémoire par une revue de la littérature sur les méthodes hybrides de programmation par contraintes et de programmation linéaire pour la résolution exacte de problèmes combinatoires (chapitre 1). Nous n'évoquons pas ici les autres types d'intégration de la recherche opérationnelle et de l'intelligence artificielle qui se sont développées parallèlement ces dix dernières années, comme la combinaison de recherche locale et de programmation par contraintes pour la résolution approchée des problèmes. En revanche, nous mettons l'accent sur les applications ; les méthodes hybrides ayant souvent été appliquées, en premier lieu, à des problèmes d'ordonnancement ou assimilés. Au préalable, pour une meilleure compréhension de ces méthodes et des procédures que nous avons nous-mêmes développées, nous revenons sur les principales techniques de résolution en programmation par contraintes et en programmation linéaire et considérons les similarités et différences des deux approches.

Le chapitre 2 est consacré à la présentation du problème d'ordonnancement de projet à contraintes de ressources. Nous définissons le problème et en présentons quelques variantes, cas particuliers et domaines applicatifs. Nous nous intéressons à la forme classique du RCPSP : minimisation de la durée totale d'ordonnancement dans le cas non préemptif avec contraintes de précédence simples. Compte tenu de la littérature abondante sur le sujet, nous n'exposerons que les techniques de résolution exacte proches de notre problématique. Nous détaillerons ainsi, les principales règles de propagation de contraintes s'appliquant au RCPSP (y compris, notre propre version de la règle globale du *shaving*), les schémas de branchement et les évaluations par défaut basées sur la programmation par contraintes ou la programmation linéaire.

Le chapitre 3 présente un premier type de bornes inférieures hybrides pour le RCPSP, issues de deux nouvelles relaxations lagrangiennes du modèle linéaire des ensembles admissibles [Mingozzi 1998]. La première borne considère la formulation linéaire de Mingozzi et al. pour le problème complet. La décomposition du sous-problème lagrangien fait apparaître des structures distinctes, à savoir : des problèmes de sac-à-dos multidimensionnels simples et un problème de coupe de capacité minimale dans un graphe (structure mise à jour par Mingozzi et al. dans une autre relaxation lagrangienne pour le RCPSP [Möhring 2003]). Nous avons calculé une borne similaire, où n'apparaît plus le problème de coupes, en ne considérant qu'une version préemptive du problème. Cette borne obtenue de manière destructive est pratiquement duale à la borne obtenue par génération de colonnes de Brucker et Knust [Brucker 2000]. Nous présentons les résultats expérimentaux de cette borne comparée aux bornes relatives de la littérature ([Mingozzi 1998, Brucker 2000, Möhring 2003]), ainsi que l'impact de notre procédure de programmation par contraintes utilisée en prétraitement de la relaxation lagrangienne.

Le chapitre 4 présente une technique hybride plus avancée pour le calcul de bornes inférieures pour le RCPSP. Ici, la programmation par contraintes n'est pas seulement utilisée en prétraitement d'une formulation linéaire, mais aussi employée pour la génération de coupes ajoutées à la relaxation continue du modèle linéaire. Nous proposons ainsi une linéarisation de règles d'inférence logique pour trois modèles distincts : un modèle en temps continu, un modèle en temps discrétisé et le modèle préemptif sur les ensembles admissibles. Pour les deux premiers modèles, nous utilisons en

particulier les déductions du shaving et la notion d'ensembles disjonctifs pour déduire de nouvelles contraintes linéaires. Pour le troisième modèle, nous nous sommes associés au travail de Philippe Baptiste sur la génération de coupes basées sur le raisonnement énergétique ou sur la prise en compte de la non-préemptivité des activités. Des résultats expérimentaux sont présentés pour les trois modèles dans une approche constructive et/ou destructive.

Enfin, dans le chapitre 5, nous nous intéressons à une méthode de résolution optimale pour les programmes linéaires en variables binaires, appelée *resolution search* [Chvátal 1997]. Cette procédure a été conçue comme une alternative aux recherches arborescentes, moins dépendante de la stratégie de branchement. Elle est peu connue en recherche opérationnelle et n'a, semble-t'il, donné lieu à aucune application pratique, mais elle nous semble néanmoins très prometteuse. Nous tentons ici d'apporter un éclairage nouveau sur cette méthode, en montrant qu'il s'agit d'une nouvelle forme de *backtracking intelligent* au sens de la PPC, pouvant être appliquée à la PL. Resolution search met aussi en œuvre une forme originale de coopération de solveurs : l'un gérant explicitement, par raisonnement logique, l'espace des solutions écartées de la recherche, et l'autre évaluant l'espace de recherche restant. Le second solveur peut alors être implémenté de diverses façons, par programmation linéaire mais aussi par programmation logique ou par contraintes, etc. Nous proposons dans un premier temps une application fidèle de la procédure de Chvátal à la formulation linéaire en temps discrétisé du RCPSP et constatons de manière expérimentale qu'il domine nettement une recherche arborescente équivalente. Nous apportons alors des améliorations à la procédure pour intensifier la coopération des solveurs et accélérer la recherche et proposons l'intégration d'un schéma de branchement spécifique au RCPSP.

Chapitre 1

Interactions entre programmation linéaire et programmation par contraintes

La programmation (logique) par contraintes (PPC), issue principalement de l'intelligence artificielle (IA), et la programmation linéaire en nombres entiers (PLNE), la branche la plus développée de la programmation mathématique en recherche opérationnelle (RO), sont deux approches permettant la résolution d'une large majorité de problèmes d'optimisation combinatoire. Leurs différences et similitudes n'ont réellement été évaluées et rapportées l'une à l'autre que depuis une dizaine d'années au plus. Depuis, des études toujours plus nombreuses se sont attachées à exploiter leur complémentarité en cherchant à hybrider ces deux approches avec pour objectif de parvenir à résoudre les problèmes les plus difficiles de la littérature. Nous montrons, dans ce chapitre, où se situent nos méthodes dans la littérature des approches coopératives PPC/PLNE pour la résolution exacte des problèmes d'optimisation combinatoire. Nous commençons par présenter, plus ou moins succinctement, les principales techniques de la PPC et de la PLNE. Certaines sont employées dans nos propres méthodes ; cette partie sert ainsi de référence aux chapitres suivants. D'autres interviennent dans les méthodes hybrides présentées en fin de ce chapitre.

1.1 Programmation par contraintes

Nous nous intéressons dans cette section à la résolution des problèmes de satisfaction de contraintes (CSP) en domaines finis (1.1.1). Plus particulièrement, nous abordons les méthodes systématiques de résolution, par opposition aux stratégies heuristiques comme la recherche locale. Ces méthodes sont basées sur deux principes, la *déduction* (1.1.2) ou renforcement de la consistance, pour rendre plus explicite, voire totalement explicite, la formulation d'un problème, et la *recherche* (1.1.3), essentiellement par backtracking, qui mène à une solution par tentatives successives. Les méthodes combinant ces deux principes (1.1.4) sont souvent les plus efficaces : les techniques de consistance servent à réduire l'espace de recherche des solutions par prédiction des tentatives infructueuses. Les *backtracking intelligents* (1.1.5) présentent une autre façon de réduire la recherche, en sachant identifier et exploiter la cause d'un échec. Plus récemment encore, des procédures hybridant backtracking intelligent et techniques de déduction ont été appliquées avec succès. On verra pourtant (1.1.6) qu'il est primordial de trouver une bonne combinaison de ces méthodes, l'efficacité de l'une pouvant être annulée par une autre. Nous terminerons par les manières de prendre en compte un critère d'optimisation au moyen de ces techniques de satisfaction de contraintes (1.1.7).

1.1.1 Problèmes de satisfaction de contraintes

Définitions

Une instance du *problème de satisfaction de contraintes*, ou CSP pour *constraint satisfaction problem*, est la donnée d'un quadruplet $P = (X, D, C, R)$ où :

- $X = \{X_1, \ldots, X_n\}$ désigne l'ensemble des variables ;
- $D = \{D_1, \ldots, D_n\}$ l'ensemble des domaines, X_i prend ses valeurs dans l'ensemble D_i ;
- $C = \{C_1, \ldots, C_m\}$ et $R = \{R_1, \ldots, R_m\}$ l'ensemble des contraintes, où la contrainte j est donnée par un ensemble $C_j = \{X_{j_1}, \ldots, X_{j_{n_j}}\}$ de variables, et par l'ensemble $R_i \subseteq D_{j_1} \times \cdots \times D_{j_{n_j}}$ des combinaisons de valeurs pouvant être prises par $(X_{j_1}, \ldots, X_{j_{n_j}})$;

On ne considère ici que les CSP en domaines finis (les ensembles D_i sont finis, pour tout i). Une variable est dite *instanciée* quand on lui assigne une valeur de son domaine. On définit ici formellement une *instanciation* par une application $\phi : X \to \cup_{i=1}^{n} D_i \cup \{*\}$ telle que $\phi(X_i) \in D_i \cup \{*\}$, pour toute variable X_i. Alternativement, on représentera ϕ par le vecteur $(\phi(X_1), \ldots, \phi(X_n))$. L'instanciation ϕ est dite *complète* si toutes les variables $X_i \in X$ sont instanciées par ϕ : $\phi(X_i) \in D_i$. L'instanciation est dite *partielle*, sinon. Une instanciation complète ϕ est une *solution* si elle satisfait l'ensemble des contraintes du problème : $(\phi(X_{j_1}), \ldots, \phi(X_{j_{n_j}})) \in R_j$, $\forall j \in \{1, \ldots, m\}$. De même une instanciation partielle ϕ est une *solution partielle* si elle satisfait l'ensemble des contraintes j ne portant que sur des variables toutes instanciées ($C_j \cap \phi^{-1}(\{*\}) = \emptyset$).

Le problème a plusieurs énoncés possibles selon qu'il s'agisse d'un problème de décision « existe-t'il une solution ? », d'un problème d'énumération ou de dénombrement de l'ensemble (ou d'un sous-ensemble) des solutions, etc. Généralement, et c'est le problème considéré ici, la résolution du problème consiste à retourner, soit une solution, soit qu'il n'existe pas de solution.

Le seul problème de décision d'un *CSP binaire* (où les contraintes ne portent au plus que sur deux variables) est NP-complet puisque le problème NP-complet de référence, SAT, le *problème de satisfiabilité* s'y réduit polynomialement.

Problème de satisfiabilité

SAT est le problème de base pour l'étude des méthodes de résolution par inférence logique en intelligence artificielle. Une instance de SAT est définie par un ensemble de variables booléennes $X = \{X_1, \ldots, X_n\}$, les variables propositionnelles, et un ensemble $C = \{C_1, \ldots, C_m\}$ de clauses. Une *clause* est une disjonction de littéraux, où un *littéral* est une variable X_i ou sa négation $\bar{X}_i$. Une solution consiste en une affectation des variables, à vrai (1) ou faux (0), telle que toutes les clauses soient satisfaites (au moins un littéral par clause est évalué à vrai). Le problème SAT est clairement un CSP booléen (les variables ne peuvent prendre que les valeurs vrai ou faux) et permet d'établir un parallèle plus rapide entre programmation logique (et donc programmation par contraintes) et programmation mathématique, comme on le verra à la dernière section (1.3.4) de ce chapitre.

CSP n-aires

Tout CSP d'arité supérieure à 2 est nécessairement NP-complet et on peut réduire un *CSP n-aire* en un CSP binaire. Cette *binarisation* passe par la création de nouvelles variables, chacune « encapsulant » une contrainte d'arité supérieure à 2. Par exemple, dans un CSP à 3 variables toutes définies sur le domaine $D = \{0, 1, 2\}$, la contrainte $X_1 + X_2 = X_3$ peut être encapsulée dans une nouvelle variable U représentant le triplet (X_1, X_2, X_3) avec pour domaine $\{(0,0,0), (0,1,1), (1,0,1), (1,1,2), (0,2,2), (2,0,2)\}$. De nouvelles contraintes, binaires cette fois, de type `element`, lient la nouvelle variable avec chacune des variables du triplet : la contrainte liant U et X_2 s'écrit `element(`X_2`,`U`,2)` (X_2 est le second élément de U) ou, plus formellement, $C_{(U,2)} = \{U, X_2\}$ et $R_{(U,2)} = \{((0,0,0),0), ((0,1,1),1), ((1,0,1),0), ((1,1,2),1), ((0,2,2),2), ((2,0,2),0)\}$.

Pour cette raison, même si la binarisation est rarement utilisée en pratique, de nombreuses techniques de résolution des CSP ont été développées dans le cadre particulier des CSP binaires, aussi appelés *réseaux de contraintes* car ils peuvent se représenter par un graphe, le *graphe de contraintes*, où les sommets sont les variables X, et les arcs entre deux sommets X_i et X_j sont les contraintes binaires liant ces deux variables. Cependant, d'autres techniques, ont été spécifiquement développées pour exploiter la globalité de certains types de contraintes n-aires, ou *contraintes globales*. Un exemple célèbre de contrainte globale est la contrainte `all-different`, pour spécifier qu'un ensemble de variables doivent prendre des valeurs toutes différentes. En ordonnancement, la contrainte globale `cumulative` [Aggoun 1993] s'emploie pour modéliser les contraintes de ressources intervenant, par exemple, dans le RCPSP (voir section 2.2). Elle indique que, à chaque instant, la quantité d'une ressource requise par l'ensemble des activités en cours d'exécution n'excède pas la quantité disponible de la ressource. Formellement, la contrainte globale

$$\texttt{cumulative}([S_1, p_1, r_1], \ldots, [S_n, p_n, r_n], R, T)$$

est satisfaite si les inégalités suivantes sont vérifiées :

$$\sum_{i : S_i \leq t < S_i + p_i} r_i \leq R, \quad \forall t \in \{0, \ldots, T\},$$

$$S_i + p_i \leq T \quad \forall i \in \{1, \ldots, n\},$$

où R désigne la capacité de la ressource et, pour chaque activité $i \in \{1, \ldots, n\}$, S_i, p_i et r_i désignent respectivement sa date de début (variable), sa durée et sa consommation sur la ressource.

1.1.2 Déduction et filtrage

Propagation de contraintes

La première approche pour résoudre un CSP est de procéder par raisonnement logique. Cette approche est appelée *propagation de contraintes* ou *renforcement de la consistance*. Pour reprendre l'exemple de la section précédente avec trois variables et la contrainte $X_1 + X_2 = X_3$, si une autre contrainte $X_1 + X_2 = 0$ entre dans la définition du problème, on en déduit que $X_3 = 0$ puis que $(0, 0, 0)$ est l'unique solution du problème. Le raisonnement consiste ainsi à inférer d'un ensemble de contraintes, d'autres contraintes plus explicites, voire totalement explicites sur les solutions du problème. Les contraintes unaires ou *contraintes de domaines* sont parmi les plus explicites puisqu'elles restreignent (ou *filtrent*) le domaine de la variable associée. Autrement dit, elles éliminent des instanciations possibles de la variable. Elles se présentent le plus souvent comme des contraintes *nogoods*, $X_i \notin D'_i$ (plutôt que $X_i \in D_i \setminus D'_i$) et sont directement traduites par l'*ajustement de domaines* des variables, $D_i \leftarrow D_i \setminus D'_i$. Supprimer les valeurs des domaines incompatibles avec les contraintes unaires se dit : effectuer la *consistance de noeuds*. Le problème est résolu en particulier dans deux cas : soit quand le domaine d'au moins une variable est vide (le CSP est insatisfiable), soit quand le domaine de toutes les variables est réduit à un singleton.

k-consistance

On considère ici un CSP binaire. Une contrainte binaire, liant deux variables X_i et X_j, est dite *consistante d'arc* si pour toute valeur d_i de X_i, il existe une valeur d_j dans le domaine de X_j telle que l'instantiation partielle (d_i, d_j) satisfait la contrainte, et de même en interchangeant X_i et X_j. Par exemple, la contrainte $X_1 > X_2$ avec les domaines $D_1 = D_2 = \{0, 1, 2\}$ est rendue consistante d'arc en supprimant les valeurs 0 de D_1 et 2 de D_2. De nombreux algorithmes (AC-1, AC-3,...) effectuent la consistance d'arc d'un CSP binaire. Ils diffèrent sur la manière de *propager* la réduction de domaine, dûe à la consistance d'une contrainte, aux autres contraintes du problème.

Dans un CSP consistant d'arcs, toutes les instanciations partielles de deux variables sont des solutions partielles, mais une instanciation complète quelconque n'est pas une solution. Dans l'exemple précédent, si on considère une troisième variable X_3 de domaine $\{0, 1\}$ et les contraintes $X_1 + X_3 = 2$ et $X_2 + X_3 = 1$, le problème est toujours consistant d'arc, mais la solution partielle $(X_1 = 2, X_2 = 0)$ ne peut pas être étendue à X_3. On peut ainsi ajouter une nouvelle contrainte liant X_1 et X_2, plus forte, $X_1 = X_2 + 1$. Pour renforcer la consistance du problème, il faut considérer les instanciations possibles de k variables simultanément. La consistance d'arc se généralise ainsi à la k-consistance. Un problème est dit *k-consistant* si toute solution partielle de $k - 1$ variables peut être étendue à une solution partielle de k variables en choisissant n'importe quelle valeur dans le domaine de la k-ème variable. Un problème est fortement k-consistant s'il est consistant pour tout $k' \leq k$. $k = 2$ correspond bien à la consistance d'arc, tandis que pour $k = 3$, le problème est dit *consistant de chemin*.

Complexité

Maintenir la k-consistance forte d'un CSP binaire est une méthode de résolution complète, dans le sens où, si le problème est fortement consistant pour un degré k suffisamment élevé, alors on peut facilement déterminer un ordre d'instanciation des variables, et une instanciation progressive des variables dans cet ordre qui mène à une solution complète [Freuder 1982]. Malheureusement, en général, même la 3-consistance ajoute tant de contraintes au problème, que le nouveau degré de consistance à atteindre pour résoudre le nouveau réseau de contraintes augmente en proportion.

Pour certaines classes de CSP cependant, la seule consistance d'arc est complète. C'est le cas en particulier des CSP modélisant des problèmes d'ordonnancement avec simplement des contraintes de précédence entre les tâches de type $S_j - S_i \geq p_i$ (la variable S_i désignant la date de début de la tâche i et la donnée p_i, sa durée). Le graphe des contraintes, sans cycle, peut en effet se décomposer en niveaux indiquant un ordre sur les arcs pour effectuer la consistance, qui se traduit alors par le calcul de plus long chemins dans le graphe (p. ex. , par l'algorithme de Bellman en $O(m)$). Plus généralement, dans les *CSP temporels simples* où les contraintes s'écrivent sous la forme $X_j - X_i \geq d_{ij}$, et où les domaines sont approximés par des intervalles, si la consistance n'est maintenue qu'aux bornes des domaines (seules les valeurs des extrémités des intervalles sont considérées), alors la consistance de chemin aux bornes est complète et peut s'effectuer en $O(n^3)$ par l'algorithme de Floyd-Warshall (voir section 2.2.1).

CSP n-aires et contraintes globales

La notion de consistance d'arc s'étend aux CSP n-aires de deux manières, à la *consistance d'arc généralisée* (pour une contrainte, toute valeur dans le domaine d'une variable est compatible avec au moins une instanciation des autres variables de la contrainte), et à la condition plus forte d'*interconsistance* (les contraintes sont rendues compatibles deux à deux). Ces deux techniques sont généralement très lourdes.

Pour certaines contraintes d'arité supérieure, des tests de consistance spécifiques ont été développés, souvent issus de la recherche opérationnelle. Ils présentent l'avantage d'être plutôt rapides mais aussi plus efficaces que la consistance d'arc sur les contraintes binarisées. La consistance de la contrainte `all-different` [Régin 1994], par exemple, est assurée par la recherche d'un couplage maximal dans un graphe. Les techniques de consistance de la contrainte `cumulative` sont incomplètes mais elles permettent d'inférer de nouvelles contraintes unaires (filtrage) et binaires, redondantes mais plus facilement traitées. Ces techniques sont présentées dans le cadre du RCPSP au chapitre suivant.

1.1.3 Recherche systématique

Énumération complète

Une autre approche pour résoudre un CSP consiste à instancier les variables ou des sous-ensembles de variables et à supprimer les instanciations qui ne sont pas des solutions, et ce jusqu'au moment où une solution complète est trouvée ou bien quand l'ensemble des instanciations possibles a été envisagé. La première méthode, *generate-and-test*, énumère toutes les instanciations complètes tant qu'aucune ne satisfait l'ensemble des contraintes. Une telle méthode n'est pas applicable en général puisque, dans le pire des cas, si par exemple le problème est insatisfiable, $|D|$ instanciations sont générées et testées.

Énumération implicite

La seconde méthode, le *backtracking*, cherche à étendre progressivement une solution partielle S en instanciant, à chaque pas, une nouvelle variable X_i à une valeur de son domaine. Seules les contraintes portant uniquement sur des variables déjà instanciées sont testées. Si la nouvelle instanciation partielle ainsi obtenue n'est pas une solution partielle, on effectue un *backtrack* sur X_i, autrement dit, X_i est instanciée à une autre valeur de son domaine, au cas où il en existe au moins une. Dans le cas contraire, si S est vide le problème est insatisfiable, sinon on effectue

un backtrack sur la dernière variable instanciée de S. Le processus se poursuit jusqu'à ce qu'une solution complète soit construite.

On parle ainsi de *recherche arborescente* où les *noeuds* de l'arbre sont des instanciations partielles, la *racine*, l'instanciation vide et les *feuilles*, soit une (des) solution(s) complète(s), soit des instanciations ne satisfaisant pas toutes les contraintes. À chaque noeud, un *branchement* est effectué sur la prochaine variable X_i à instancier, correspondant à autant de branches qu'il y a de valeurs dans le domaine de X_i. La complexité dans le pire des cas est la même que pour generate-and-test. Malgré tout l'insatisfiabilité est souvent détectée plus tôt quand seul un sous-ensemble X' de variables est instancié, ce qui permet au backtracking de supprimer de l'espace de recherche, $\prod_{i \in X \setminus X'} |D_i|$ intanciations. Le backtracking est la méthode de résolution de base la plus employée. Elle peut être adaptée de différentes façons.

Stratégies de branchement

Le premier point à considérer est l'ordre d'instanciation des variables et des valeurs des domaines. Ces choix peuvent avoir un impact substantiel sur l'efficacité du backtracking et plusieurs stratégies ont été développées et analysées. Le plus souvent, l'ordre est déterminé dynamiquement, en cherchant à effectuer, pour tout noeud, le prochain branchement le plus approprié. Parmi les stratégies les plus puissantes pour le choix de la prochaine variable à instancier, on trouve : la variable de domaine minimal (*search-rearrangement method*) ou bien la variable la plus contrainte, de sorte d'arriver rapidement à une insatisfiabilité. Une autre stratégie, proche de cette dernière, consiste à instancier en dernier des variables non liées entre elles (appartenant à un ensemble stable, le plus grand possible, du graphe des contraintes). Enfin, on peut aussi choisir en premier les variables liées entre elles par des contraintes formant des cycles dans le graphe de contraintes. Ainsi, une fois ces variables instanciées, le graphe se présente alors comme un arbre et l'instanciation courante peut être complétée sans backtrack.

De même, l'ordre des branchements pour une variable sur les différentes valeurs de son domaine, a son importance et est particulièrement spécifique aux données du problème. On peut, par exemple, préférer considérer en premier, les valeurs qui maximisent le nombre d'alternatives suivantes, ou encore les valeurs qui conduisent aux sous-problèmes estimés les plus faciles à résoudre. Enfin, on peut aussi choisir de ne pas instancier la variable à une valeur mais plutôt de séparer son domaine en sous-ensembles (éventuellement disjoints) et ainsi de créer une branche pour chaque sous-ensemble, obligeant la variable à prendre ses valeurs uniquement dans ce nouveau domaine. L'arbre obtenu est susceptible d'être binaire, par exemple, si on sépare le domaine sur une valeur : en partitionnant $D_i = \{v\} \cup D_i \setminus \{v\}$ ou bien, comme dans un CSP numérique, en posant $D_i = [a_i, b_i] = [a_i, v] \cup]v, b_i]$. En fait, cette méthode de branchement diffère du backtracking standard seulement si on est capable de détecter l'infaisabilité d'instancier une variable à un ensemble de valeurs par l'utilisation de techniques de maintien de consistance.

1.1.4 Évitement des conflits

En effet, la manière la plus simple d'accélérer un backtracking est de prévenir au plus tôt les infaisabilités futures (*look-ahead*) en alliant backtracking et propagation de contraintes. Un noeud de l'arbre s'identifie en fait à un sous-problème CSP où le domaine de certaines variables (les variables instanciées) est réduit à un singleton. Dans un sens, le backtracking utilise déjà, à chaque noeud, une forme de consistance d'arc sur les variables déjà instanciées pour prouver la violation des contraintes. Plus évolué, *forward-checking* réduit le domaine des variables non encore instanciées en renforcant la consistance des arcs les liant avec la dernière variable instanciée. On

peut ainsi apporter, pour un CSP binaire, divers degrés de consistance d'arc. On peut aller, par exemple, jusqu'à assurer la consistance d'arc de tous les sous-problèmes associés à chaque noeud (*real-full-look-ahead*, `rfl`, ou *maintaining-arc-consistency*, `mac`).

Évidemment, dans cette approche, il est nécessaire de choisir le meilleur équilibre entre filtrage et recherche, c.-à-d. entre le nombre de noeuds visités et le temps de calcul passé à chaque noeud. Si historiquement, forward-checking semblait généralement être le meilleur compromis, avec l'amélioration des algorithmes de consistance d'arc, `mac` est maintenant considéré comme l'un des meilleurs algorithmes pour résoudre les CSP [Bessière 1996].

Il est intéressant aussi de noter que, inversement, certaines techniques de consistance incorporent l'idée de backtracking. C'est le cas, par exemple de SAC, *singleton-arc-consistency* [Debruyne 1997]. Un CSP est dit SAC si, pour toute variable X_i et pour toute valeur v de D_i, le sous-problème associé à l'instanciation de X_i à v est consistant d'arc. Cette idée a été reprise en ordonnancement sous le nom de *shaving* (voir section 2.2.6).

1.1.5 Réparation des conflits

Trashing et redondance

De l'idée de base du backtracking, ont été dérivées d'autres formes de recherche qui regardent de plus près les causes de l'insatisfiabilité d'une instanciation partielle (*look-back*), c'est la classe des *backtrackings intelligents*. Leur but est de remédier aux deux principaux défauts du backtracking standard : le *trashing*, reproduire toujours le même échec dû à une instanciation ancienne, et la *redondance*, réeffectuer une partie de la recherche qui est indépendante d'un instanciation antérieure.

L'exemple suivant illustre ce comportement, avec 4 variables X_1, X_2, X_3 et X_4 de domaines $\{0, 1, 2\}$ et les contraintes $C = X_3 + 2 < 2X_2$ et $C' = X_4 < X_1$. La figure 1.1 représente une partie de l'arbre de recherche, où les noeuds sont les instanciations partielles (p. ex. , $(0, 1, *, *)$ est l'instanciation $X_1 = 0$, $X_2 = 1$ et X_3 et X_4 non encore instanciées). Dans un backtracking simple

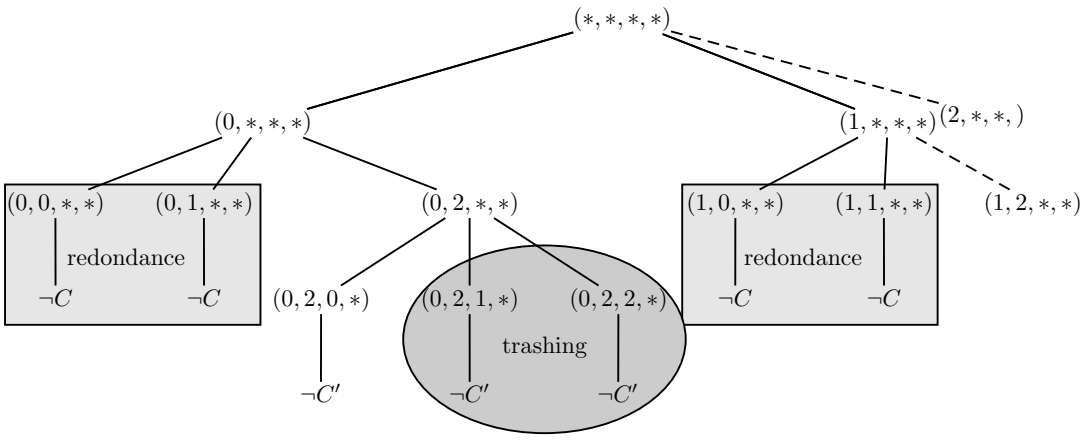

FIG. 1.1 – *Trashing et redondance.*

où les variables et valeurs sont prises dans l'ordre numérique, le phénomène de trashing apparaît au moment d'instancier X_4 car, quelles que soient les valeurs de X_2 et X_3, aucune valeur de X_4 n'est compatible avec $X_1 = 0$.

Backjumping

backjumping remédie à ce cas de figure en effectuant directement un backtrack sur une variable dont l'instanciation, en même temps que les instanciations précédentes, cause le conflit (ici on passe directement du noeud $(0, 2, 0, *)$ au noeud $(1, *, *, *)$, sans visiter $(0, 2, 1, *)$ et $(0, 2, 2, *)$). Les algorithmes de *backjumping* [Gaschnig 1979, Prosser 1993] diffèrent entre eux sur leur manière d'identifier la variable la plus haute à laquelle on peut directement remonter sans oublier de solution. Il s'agit en particulier de caractériser un *conflict-set*, un sous-ensemble de variables dont l'instanciation présente est interdite, le plus petit possible. Quand un échec intervient au moment d'instancier une nouvelle variable X_i, le conflict-set peut être, par exemple, l'ensemble des variables liées à X_i dans le graphe de contraintes ou bien l'ensemble des variables (déjà instanciées) qui interviennent dans au moins une contrainte violée par une instanciation de X_i. Pour assurer la convergence de l'algorithme, le backtrack s'effectue sur la variable du conflict-set la plus récemment instanciée.

Learning

Le backjumping est plus efficace que le backtracking, cependant il n'empêche pas la redondance de la recherche. Dans notre exemple, une partie du travail qui avait été fait sur les variables X_2 et X_3 pour l'instanciation $X_1 = 0$ sera reproduite pour $X_1 = 1$. La solution consiste à « apprendre » de la recherche passée, c'est à dire, à enregistrer les informations déduites des branchements déjà effectués, branchements qui ont donc abouti à un échec. Les approches basées sur ce principe sont regroupées sous la dénomination *learning algorithms* (ou *constraint recording*,...). *Dependency directed backtracking* [Stallman 1977], `ddb`, est le premier algorithme à avoir adopté ce principe. Comme le backjumping, il agit en identifiant, à chaque échec rencontré, la cause de l'échec, autrement dit un conflict-set pour la dernière variable instanciée. En ce sens, backjumping, mais aussi d'autres stratégies de backtracking modifié (backmarking, backchecking,... non développées ici) sont des cas particuliers de `ddb` : la raison de l'échec provient des variables dont « dépend » la dernière variable considérée. Seulement, `ddb` va en plus mémoriser cette cause sous la forme d'une contrainte *nogood* :

$$(X_{i_1} \neq v_{i_1}) \vee (X_{i_2} \neq v_{i_2}) \vee \cdots (X_{i_k} \neq v_{i_k}),$$

où $(X_{i_1}, \ldots, X_{i_k})$ sont les variables du conflict-set et $(v_{i_1}, \ldots, v_{i_k})$ leur instanciation courante. Une instanciation partielle est désormais considérée comme une solution partielle si elle satisfait les contraintes initiales du problème mais aussi l'ensemble des contraintes nogood portant sur les variables instanciées.

Le coût d'un tel algorithme peut évidemment s'avérer prohibitif, tant sur l'espace mémoire pour stocker l'ensemble Γ des nogoods, que sur le temps pris pour vérifier la satisfiabilité de ces contraintes supplémentaires. Pour remédier à cela, il suffit de restreindre l'apprentissage ou de gérer l'ensemble Γ de façon à en limiter la taille. Une première possibilité (*k-order learning* [Dechter 1990]) consiste à ne générer des nogoods contenant au plus k variables, pour un entier k donné, l'idée étant qu'un « petit » nogood a plus de probabilité d'être violé de nouveau. Pourtant, un conflict-set, même grand, généré à un noeud de l'arbre a des chances d'être rencontré de nouveau dans les noeuds voisins. Une autre technique (*relevance-bounded learning*) consiste donc à ne conserver que les nogoods *pertinents* à la position courante dans l'espace de recherche. Cette technique a été initiée par Ginsberg [Ginsberg 1993] avec le *dynamic backtracking*, `dbt`. À un instant donné, `dbt` mémorise pour toute variable X_i, l'ensemble des valeurs v_{ik} auxquelles X_i ne peut être actuellement instanciée, ainsi qu'une *justification* J_{ik} du fait que $X_i \neq v_{ik}$: J_{ik} est

un ensemble des variables dont l'instanciation courante ϕ avec $X_{ik} = v_{ik}$ viole une contrainte du problème. La contrainte nogood correspondant s'écrit sous la forme directe par :

$$\wedge_{j \in J_{ik}} (X_j = \phi(X_j)) \Rightarrow X_i \neq v_{ik}.$$

`dbt` ne conserve que les nogoods pertinents pour l'instanciation courante ϕ, c.-à-d. dont le membre de gauche est vérifié par ϕ. De plus, si toutes les valeurs d'une variable X_i donnée sont interdites par des contraintes de ce type, alors ces dernières sont supprimées et remplacées par un unique nogood, aussi écrit sous la forme directe :

$$\wedge_{j \in J \setminus \{i'\}} (X_j = \phi(X_j)) \Rightarrow X_{i'} \neq \phi(X_{i'}),$$

où $J = \cup_k J_{ik}$ et i' est la variable de J la plus récement instanciée par ϕ. Ce principe correspond exactement au *principe de résolution* en logique propositionnelle $\{ A_1 \vee B \ , \ A_2 \vee \neg B \} \vdash A_1 \vee A_2$. La contrainte obtenue est la *résolvante* des contraintes supprimées. Le principe de résolution est complet, autrement dit, on prouve l'insatisfiabilité du problème (`UNSAT`), dès que la clause vide est déduite par résolution. Il est à noter que le graphe de l'espace de recherche de `dbt` n'a pas

---

Algorithme 1 – `Dynamic-backtracking`

1° initialiser l'instanciation ϕ et l'ensemble Γ des nogoods à vide ;

2° si l'instanciation ϕ est complète, retourner ϕ, sinon choisir une variable x non instanciée. Pour toute valeur de x entraînant une violation (des contraintes originales ou des nogoods), ajouter la contrainte nogood correspondante à Γ ;

3° s'il en existe une, choisir une valeur v possible pour x, poser $\phi(x) = v$ et aller à l'étape 2° ;

4° aucune instanciation de x n'est possible. Remplacer, dans Γ, les nogoods sur x par leur résolvante dans la forme directe. Si la résolvante est vide, retourner `UNSAT`. Sinon, soit y, la variable du membre de droite de la résolvante. Poser $\phi(y) = *$ et supprimer tous les nogoods de Γ contenant la variable y dans le membre de gauche. Si chaque valeur de y viole un nogood, aller à l'étape 4° , sinon, poser $x = y$ et aller à l'étape 2° .

à proprement dit une structure arborescente puisqu'il arrive que le backtrack s'effectue sur une variable (la variable y de l'algorithme 1) tandis que d'autres variables ultérieurement instanciées (les variables intermédiaires entre y et x) conservent leurs valeurs.

1.1.6 Amélioration du backtracking ?

Look-ahead et look-back sont deux approches alternatives pour améliorer le backtracking standard, en diminuant le nombre de noeuds à explorer dans l'arbre de recherche. Avec l'objectif de réduire toujours plus l'espace de recherche, des techniques combinant ces deux approches ont été proposées plus récemment. Ainsi, de nouveaux algorithmes intègrent la propagation de contraintes à des backtracking intelligents comme, par exemple, `mac` et backjumping [Prosser 1995], forward checking et `dbt` [Schiex 1994] ou encore `mac` et `dbt` [Jussien 2000]. Bien que cette intégration ajoute un degré de complexité aux algorithmes (dû à l'enregistrement des nogoods issus d'une inconsistance et à la restauration de la consistance après la suppression d'une instanciation), elle s'avère généralement utile pour accélerer la résolution. Jussien et al. [Jussien 2000] rapportent par exemple de très bons résultats de `mac-dbt` sur des instances CSP de grande taille ou sur des instances CSP plus petites et plus structurées.

Inversement, d'autres expérimentations [Bessière 1996] ont montré que `mac` était souvent plus

rapide seul, c.-à-d. dans un backtracking standard, que combiné à une recherche de type backjumping. Bien que ces résultats puissent être attribués au fait que le backjumping, contrairement au `dbt`, n'évitent pas la redondance de la recherche, il semble néanmoins que le comportement de ces techniques peut parfois être contraire à l'effet attendu. Par exemple, Baker produit dans sa thèse [Baker 1995] un type d'instances sur lesquelles l'efficacité de `dbt` est annulée par la stratégie, pourtant usuelle, de sélection des variables. Ce comportement provient du fait que, au moment d'effectuer un backtrack sur une variable y, `dbt` conserve les instanciations ultérieures, de façon à ne pas répéter le travail déjà effectué sur ces variables. Pourtant, bien que ces instanciations peuvent elles aussi être la cause d'une infaisabilité, la recherche va se poursuivre sur la base de cette instanciation partielle. À l'inverse, backtracking et backjumping suppriment toutes les instanciations ultérieures au moment du backtrack sur y.

En réalité, stratégies de branchement, techniques de prévision ou réparation des conflits ne sont pas toujours des méthodes strictement complémentaires et adoptent même parfois des comportements similaires. Forward checking peut être vu, dans un sens, comme une stratégie de branchement particulière, où les valeurs inconsistantes des variables sont testées en premier, entraînant immédiatement un backtrack. Forward checking réduit aussi l'espace de recherche de manière analogue au backjumping, s'il détecte, par exemple, à une profondeur k de l'arbre, l'impossibilité d'instancier une variable à un niveau $k + j$.

Pour accélérer la recherche d'une solution, il est donc capital, non seulement de trouver le meilleur compromis entre le nombre de noeuds du graphe de recherche et le temps de résolution passé à chaque noeud, mais aussi d'étudier les combinaisons possibles entre stratégies de branchement, techniques de filtrage et méthodes d'apprentissage.

1.1.7 Optimisation

Toutes les méthodes exposées ci-dessus pour la recherche d'une solution d'un CSP s'appliquent également à la résolution des CSOP *constraint satisfaction optimisation problem*, quant il s'agit de rechercher une solution satisfaisant toutes les contraintes du problème et qui, de plus, minimise ou maximise un critère donné. Pour prendre l'exemple d'un problème de minimisation, il existe plusieurs façons de prendre en compte un critère $\min f(\phi(X))$, où l'*objectif* f est fonction de l'instanciation totale ϕ des variables du problème, et prend ses valeurs dans un ensemble ordonné. Soit f^* la valeur minimale de f sur l'ensemble des solutions du problème.

L'approche la plus intuitive consiste à résoudre une succession de problèmes de décision P. Le problème de décision P est initialisé avec les contraintes du problème, sans tenir compte du critère d'optimisation, puis résolu. Si P est insatisfiable, alors le problème d'optimisation n'a pas de solution, sinon, tant que P possede une solution $\phi^*(X)$, on ajoute la contrainte $f(\phi(X)) < f(\phi^*(X))$, pour toute solution ϕ de P. La dernière solution trouvée est une solution optimale du problème. Les contraintes redondantes qui ont éventuellement été générées lors d'une étape de résolution demeurent évidemment valides à l'étape suivante. Les déductions faites à une étape peuvent donc être conservées dans P ou pas. Dans une recherche arborescente, cela revient, soit à repartir de la racine de l'arbre à chaque nouvelle résolution, soit à modifier le problème de décision en cours de recherche, en ajoutant la nouvelle contrainte dès lors qu'une feuille de l'arbre est atteinte.

Pour accélerer la recherche, il est d'usage d'itérer sur des valeurs T différentes, et de résoudre successivement des instances décisionnelles P_T comprenant les contraintes du problème d'optimisation avec la contrainte supplémentaire $f(\phi(X)) < T$. Généralement, connaissant LB et UB, respectivement une borne inférieure et une borne supérieure de f^*, on recherche par dichotomie

sur l'intervalle $]LB, UB]$ la plus grande valeur T rendant P_T insatisfiable. Si pour une valeur T, P_T est insatisfiable, alors l'intervalle est réduit de moitié en posant $LB = T$. De plus, la contrainte $f(\phi(X)) \geq T$ étant redondante au problème, elle peut être ajoutée à sa définition. Inversement, si P_T possède une solution $f^*(X)$, l'intervalle est réduit en posant $UB = T$ avec $T = f(\phi^*(X))$. On remarque que, à tout moment du processus, $LB < f^* < UB$. À terme, dès que $LB = UB$, la dernière solution trouvée est encore optimale ($f(\phi^*(X)) = UB = f^*$). Plutôt que par dichotomie, la recherche peut aussi se faire par valeur croissante ou décroissante de T.

La technique s'applique à l'identique au simple calcul d'une évaluation par défaut de la valeur minimale du problème. À la place de P_T, on peut considérer un sous-problème P'_T n'incluant qu'une partie des contraintes initiales du problème. Si P'_T est insatisfiable alors P_T (plus contraint) aussi, et T est bien une borne inférieure de f^*. Dans un processus itératif, on peut, dans ce cas, tester une autre valeur de T plus grande. Inversement, on itère sur une valeur de T plus petite, si une solution existe ou bien si, pour réduire les temps d'exécution, on décide d'interrompre la résolution de P'_T. La borne inférieure retenue est évidemment la plus grande valeur de T pour laquelle on prouve l'insatisfiabilité de P'_T.

Le calcul de bornes inférieures par ce type d'approche est de plus en plus fréquent en ordonnancement où on parle alors de *bornes destructives* [Klein 1999]. Cette technique est en effet particulièrement efficace dans ce cadre car, ainsi, les contraintes « difficiles » (les contraintes de ressources) n'ont pas besoin d'être prises en compte dans leur intégralité, et surtout, les techniques de propagation de contraintes dédiées à l'ordonnancement prouvent généralement rapidement l'inconsistance pour les valeurs de T trop en-deçà de la valeur optimale du problème. On reviendra à la section 2.4.3 sur les bornes destructives pour le RCPSP. Les bornes inférieures que nous proposons aux chapitres 3 et 4 sont aussi calculées de manière destructive.

Il existe d'autres méthodes exactes d'optimisation en programmation par contraintes, basées sur des modèles plus généraux que le CSP, comme les CSP valués [Schiex 1995] ou bien les SCSP [Bistarelli 1997] (*semiring-based CSP*). Ces modèles intègrent une notion de hiérarchie entre les contraintes, différenciant les contraintes « dures », à satisfaire obligatoirement, et les contraintes « molles » (telles que le critère d'optimisation) qui doivent être satisfaites avec un certain degré d'incertitude. Initialement, ces modèles et les méthodes de résolution associées ont été développés dans un cadre (les CSP sur-contraints) plus général que le cadre habituel d'optimisation combinatoire. Le niveau de complexité qu'ils engendrent fait qu'ils sont, actuellement, rarement employés à l'optimisation « simple ». En revanche, ils peuvent éventuellement être envisagés pour l'optimisation multi-critères.

Enfin, le backtracking peut être adapté aux CSOP où la fonction objectif aide à la réduction de l'espace de recherche. Pour la minimisation de la fonction f, il s'agit de caractériser une *borne inférieure* et une *borne supérieure* de f, autrement dit, deux fonction lb et ub sur l'ensemble des instanciations partielles, telles que, si ϕ est une instanciation partielle et ϕ' une solution étendant ϕ, alors $lb(\phi(X)) \leq f(\phi'(X))$ et $f(\phi'(X)) \leq ub(\phi(X))$. Un bactrack est effectué à tout noeud de l'arbre de recherche dont l'instanciation partielle correspondante ϕ vérifie $lb(\phi(X)) > ub(\phi(X))$, puisqu'alors aucune solution ϕ' ne peut être étendue à partir de ϕ. Cette méthode appartient à la famille des PSE, *procédures par séparation et évaluation*, initialement développées pour la résolution de problèmes d'optimisation en recherche opérationnelle. En réalité, la borne supérieure est rarement explicitée en programmation par contraintes. Elle correspond en général à la valeur optimale f^* de la dernière solution trouvée et est directement ajoutée au problème sous la forme d'une contrainte $f(X) < f^*$. L'inconvénient ici est qu'il est souvent difficile d'identifier une bonne borne inférieure, ceci étant dû principalement au fait que, avec leur formalisme large, les contraintes ne peuvent être traitées que de manière « locale ». À l'inverse, en imposant un formalisme plus

restrictif, la programmation mathématique permet de considérer l'ensemble des contraintes du problème dans leur globalité, et ainsi de prendre réellement en compte le critère d'optimisation pour guider la recherche.

1.2 Programmation linéaire en nombres entiers

L'objectif de cette section est de présenter les notions et les outils de la programmation linéaire qui seront utilisés dans la suite de ce mémoire (méthodes de coupes, relaxation lagrangienne,...) ou qui présentent une similarité ou un moyen d'intégration avec la programmation par contraintes (décomposition de Benders, p. ex.). Cette section ne présente pas un état de l'art exhaustif et précis des techniques du domaine (la programmation dynamique, p. ex. , n'est pas évoquée) ; nous renvoyons donc à la lecture de [Nemhauser 1988, Wolsey 1998], par exemple, pour un exposé plus complet.

1.2.1 Programmation linéaire

Un problème d'optimisation consiste à déterminer un élément parmi un ensemble S qui minimise ou maximise un certain critère f. Si l'ensemble S est formulé comme un ensemble de variables à valeurs dans l'ensemble des réels $\mathbb{R}$ et contraintes à satisfaire des inégalités linéaires et si f est une fonction linéaire en ces variables, on parle alors d'un problème de programmation linéaire (PL). Une instance de ce problème s'exprime donc par un *programme linéaire* (PL), pouvant s'écrire de la façon suivante :

$$\min cx \tag{1.1}$$

sujet à :

$$Ax \geq b \tag{1.2}$$

$$x \in \mathbb{R}^n_+ \tag{1.3}$$

avec $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ et $A \in \mathbb{R}^{mn}$.

Sans perte de généralité, on suppose que les variables sont positives (1.3) et que la *fonction objectif* $x \mapsto cx$ doit être minimisée (1.1) (qui équivaut à maximiser $x \mapsto -cx$). Les éléments de l'ensemble $S = \{x \in \mathbb{R}^n_+ \mid Ax \geq b\}$, sont les *solutions réalisables* du problème, et parmi ceux-ci, les éléments x^* qui minimisent le critère cx^* sont les *solutions optimales*.

Dans sa *forme standard*, un programme linéaire ne contient que des contraintes sous la forme d'égalité. La forme standard du programme précédent s'obtient simplement en soustrayant, pour chacune des inégalités (1.3), une variable positive, appelée *variable d'écart*, dans le membre de gauche :

$$\min\{cx \mid Ax - I_m e = b, x \in \mathbb{R}^n_+, e \in \mathbb{R}^m_+\}$$

où I_m est la matrice identité d'ordre m.

Résolution des programmes linéaires

Un programme linéaire a une interprétation géométrique simple. L'ensemble S des solutions réalisables est par définition un *polyèdre* (un *polytope* si S est fini) de l'espace euclidien $\mathbb{R}^n$. Si S est non vide, c'est à dire si le problème (PL) possède une solution, (PL) est dit *réalisable*. Dans ce cas, toute solution optimale de (PL) se trouve nécessairement à un sommet (ou *point extrême*), ou éventuellement sur une *face* (intersection de S et d'un plan $\{x \in \mathbb{R}^n \mid A_j x = b_j\}$), du polyèdre

S. Pour notre exemple de minimisation, il s'agit des éléments de S, les plus proches de l'origine de R^n, dans la direction induite par le vecteur c.

La méthode de résolution du *simplexe* [Dantzig 1951] repose sur cette observation. Partant d'une solution réalisable à un sommet du polyèdre, on recherche par itérations succesives des solutions meilleures correspondant à d'autres sommets, en parcourant les arêtes de S. Les principaux algorithmes du simplexe peuvent, théoriquement dans le pire des cas, ne pas converger (en temps polynomial).

D'autres algorithmes ont été développés pour la résolution des programmes linéaires : la *méthode de l'ellipsoïde* [Khachiyan 1979] construit itérativement une suite d'ellipsoïdes de volume décroissant jusqu'à ce que le centre de la dernière ellipsoïde soit un point intérieur « presque optimal » du polyèdre S. Une solution optimale est facilement déterminée à partir de ce point.

Les *algorithmes projectifs* [Karmarkar 1984] aboutissent à un point intérieur proche de l'optimum, mais d'une autre manière. Brièvement, il s'agit de construire successivement des points de l'espace satisfaisant chaque fois plus de contraintes en considérant leur projection sur les plans délimitant le polyèdre S (c.-à-d. les plans supportés par $A_j x = b_j$ pour $j \in \{1, \ldots, m\}$).

Ces deux types de méthodes convergent en temps polynômial et présentent donc pour cela une importance théorique capitale. De plus, les algorithmes projectifs donnent maintenant de bons résultats calculatoires. Cependant, dans la pratique, les algorithmes du simplexe sont à ce jour encore les plus utilisés pour leur efficacité depuis longtemps éprouvée. Ces derniers sont généralement implémentés dans les logiciels de résolution linéaire.

Dualité

Les programmes linéaires sont liés deux à deux, la résolution de l'un étant complémentaire de la résolution de l'autre. Le programme linéaire *dual* (DL) du programme *primal* (PL) ci-dessus est défini par :

$$\max\{ub \mid uA \leq c, u \in \mathbb{R}_+^m\}.$$

Le théorème de dualité établit que toutes solutions réalisables x et u de (PL) et de (DL), respectivement, vérifient $ub \leq cx$. De plus, il y a égalité $ub = cx$ si x et u sont optimales.

Il y a souvent plusieurs formulations possibles d'un problème d'optimisation combinatoire en un programme linéaire. Le choix d'une bonne formulation est primordial pour la rapidité de résolution par une quelconque méthode. Ainsi, pour trouver la valeur optimale d'un programme linéaire, il est parfois préférable de résoudre son dual à la place. C'est le cas encore, si on souhaite réoptimiser après l'ajout de nouvelles contraintes, comme par exemple, dans une méthode de coupes (voir section 1.2.3). Il est facile de déduire d'une solution optimale du programme initial, une solution réalisable du dual du nouveau programme. Partant de cette solution, le simplexe appliqué au dual effectuera alors un nombre moindre d'itérations. Enfin, d'après le théorème de dualité, on prouve l'irréalisabilité d'un programme linéaire si son dual est non borné et, sinon, toute solution réalisable du dual fournit une borne inférieure de la valeur optimale, qui peut être exploitée, comme on le verra à la section suivante.

1.2.2 Contraintes d'intégralité

De nombreux problèmes d'optimisation combinatoire, et en particulier les problèmes d'ordonnancement qui nous intéressent ici, peuvent s'exprimer sous la forme de programmes linéaires, mais où les variables sont astreintes à prendre leurs valeurs dans un domaine discret. Généralement, on restreint le programme linéaire aux seules solutions à coordonnées, toutes ou en partie, entières.

On parle alors de *programme linéaire en nombres entiers* (PLNE) ou, si les variables sont à valeurs booléennes 0 ou 1, de *programme linéaire en variables binaires* (PLVB). Un programme linéaire en nombres entiers s'écrit donc de la façon suivante :

$$\min cx \tag{1.4}$$

sujet à :

$$Ax \geq b \tag{1.5}$$

$$x \in \mathbb{N}^n \tag{1.6}$$

Pour que les calculs restent dans l'ensemble des rationnels, on suppose généralement les données A, b et c à valeur dans $\mathbb{Z}$.

La condition d'intégralité ajoute un degré de complexité au problème et la version décisionnelle d'un PLNE est NP-complet [Garey 1979]. Géométriquement, l'ensemble des solutions n'est plus un espace plein, mais l'ensemble discret S des points de $\mathbb{N}^n$ contenus dans le polyèdre $P = \{x \in \mathbb{R}^n \mid Ax \geq b\}$.

Approches polyédriques

Pour certains problèmes, il est possible de caractériser exactement $conv(S)$, l'*enveloppe convexe* de S. Par convexité de P, on a nécessairement $S \subseteq conv(S) \subseteq P$ et puisque le polyèdre $conv(S)$ contient tous les points de S et a aussi tous ses sommets dans S, on a l'égalité :

$$\min\{cx \mid x \in S\} = \min\{cx \mid x \in conv(S)\}.$$

Ce second programme linéaire, à variables continues cette fois, peut être facilement résolu par les méthodes exposées en 1.2.1. Les approches polyèdriques cherchent, partant d'une formulation P, à caractériser $conv(S)$.

Le cas le plus favorable est quand la matrice A est *totalement unimodulaire* (tous les déterminants de ses sous-matrices sont égaux à 0, 1, ou -1), puisqu'alors les polyèdres P et $conv(S)$ coïncident. Certains problèmes se modélisent trivialement en PLNE avec une matrice des contraintes totalement unimodulaire, tels que de nombreux problèmes de flots dans un graphe ou encore des problèmes d'ordonnancement simples (similaires à des problèmes de flots) tels que l'ordonnancement avec contraintes de précédence (ou fenêtres de temps) mais sans contraintes de ressources (voir chapitre 3). Ces problèmes peuvent donc être résolus en temps polynômial. En pratique, ils ne sont d'ailleurs pas résolus par programmation linéaire mais par des algorithmes spécifiques.

Pour d'autres problèmes simples, il est aussi possible de décrire complètement le système d'inégalités minimal définissant $conv(S)$ (les *facettes*). Dans [Queyranne 1993], Queyranne produit, par exemple, les $2^n - 1$ inégalités définissant l'ensemble des ordonnancements réalisables de n travaux sur une seule machine et sans contraintes de précédence. Le nombre exponentiel d'inégalités rend cependant impossible la résolution directe de tels programmes. La solution consiste alors à générer progressivement les contraintes et résoudre le programme linéaire avec uniquement, à chaque itération, un sous-ensemble de contraintes. Un tel algorithme est appelé *algorithme de séparation* ou *algorithme de coupes*, puisqu'il s'agit de partir d'un point x de l'espace, généralement une solution optimale du programme linéaire $\min\{cx \mid x \in P\}$, puis, tant que x n'appartient pas à $conv(S)$, « couper » l'espace en ajoutant une ou plusieurs inégalités au programme linéaire qui ne sont pas vérifiées par la solution courante x. Le nouveau PL est résolu, x mis à jour avec une de ses solutions optimales et le processus est réitéré. Les inégalités ajoutées sont appelées des *inégalités valides*

(pour toute solution de S) ou, plus précisément, des *coupes* car les plans qu'elles supportent séparent le point x de P du polyèdre $conv(S)$ et la recherche de la nouvelle solution est effectivement restreinte à la partie de l'espace qu'elles définissent.

D'autres études polyédriques ont été menées sur des problèmes d'ordonnancement plus difficiles : avec contraintes de précédence [Queyranne 1991] ou avec fenêtres de temps [Dyer 1990], ou bien encore, sur des problèmes avec plusieurs machines comme le job-shop [Balas 1985, Applegate 1991] ou le RCPSP [Alvarez-Valdés 1993]. Pour l'ordonnancement non-préemptif, on trouvera dans [Queyranne 1994, Schultz 1996] un état de l'art complet sur les approches polyédriques, en fonction des problèmes et de la représentation des solutions : au moyen de variables entières de *temps continu* S_i pour la date de début de l'activité i ou bien de variables booléennes de *temps discrétisé* x_{it} pour la décision « i commence au temps t ».

Rappelons ici, l'importance fondamentale du choix de la formulation (et donc de la définition des variables), plus particulièrement encore pour l'efficacité des approches polyédriques. En effet, une formulation est meilleure si P n'est pas trop grand par rapport à $conv(S)$.

Pour les problèmes plus difficiles, la caractérisation de $conv(S)$ ne peut être que partielle puisque le problème de séparation sous-jacent est alors, comme le problème d'optimisation NP-complet [Grötschel 1981]. Un algorithme de coupes, seul, n'est donc plus suffisamment efficace. Quand plus aucune inégalité valide ne peut être ajoutée et que la solution n'est toujours pas entière, il est nécessaire alors d'effectuer une coupe artificielle de l'espace et de considérer séparément les deux demi-espaces ainsi obtenus. Cette séparation est appelée *branchement* et c'est l'un des principes de base d'une autre méthode de résolution des PLNE : la *recherche arborescente*.

Procédures par séparation et évaluation

Les *procédures par séparation et évaluation* (PSE ou *branch and bound*) sont le moyen générique le plus utilisé pour la résolution exacte des problèmes d'optimisation combinatoire, et en particulier pour la résolution des PLNE, pour lesquelles les PSE ont initialement été initialement conçues. Comme dans toute recherche arborescente, il s'agit d'énumérer implicitement l'ensemble S des solutions du problème en explorant progressivement les différentes parties de l'espace de recherche, et en tentant d'identifier et de supprimer les plus larges sous-espaces ne contenant pas de solution optimale. Une telle méthode peut se représenter comme la construction d'un arbre, de la racine aux feuilles, où chaque noeud correspond à une partie de l'espace de recherche, la racine correspondant à une partie incluant S.

À chaque itération de la procédure, à un noeud N donné, on tente de déterminer si l'espace E correspondant contient une solution optimale. Si on ne prouve pas le contraire et si E n'est pas réduit à un point, alors E est partitionné en sous-espaces, chacun étant représenté par un noeud fils de N ajouté à l'arbre. Dans tous les cas, le noeud N est marqué comme visité et la recherche se poursuit sur un autre noeud de l'arbre non déjà visité, tant qu'il en existe. À terme, soit il existe une feuille de l'arbre dont l'espace associé est réduit à une solution optimale du problème, soit le problème est irréalisable. Il y a donc trois ingrédients principaux qui caractérisent une PSE : la façon de séparer l'espace ou *schéma de branchement*, la méthode d'évaluation des noeuds pour tenter d'identifier ceux ne contenant pas de solution optimale et l'ordre de sélection des noeuds.

- ***Ordre de sélection***

Pour l'ordre des noeuds, il existe trois stratégies usuellement mises en oeuvre :

– *Profondeur d'abord* : quand un branchement est effectué, la recherche se poursuit sur l'un des noeuds créé. Si le noeud est une feuille (le domaine associé est réduit à un point ou bien ne contient pas de solution optimale), on effectue un *backtracking* en considérant un noeud

frère, et ainsi de suite. Cette stratégie est la plus souvent utilisée car elle présente, entre autres, l'avantage majeur d'économiser l'espace mémoire et du temps de calcul en passant d'une itération à l'autre. En effet, le travail d'évaluation d'un domaine s'applique à tout sous-domaine complété par la nouvelle information issue du branchement. Autrement dit, le travail effectué à un noeud peut être facilement réutilisé à l'itération suivante, pour l'évaluation d'un noeud fils (principe d'incrémentalité).

- *Largeur d'abord* : l'arbre est parcouru de sorte qu'un noeud n'est évalué qu'après tous les noeuds de profondeur inférieure. À un branchement, tous les noeuds fils sont examinés l'un à la suite de l'autre, puis tous les noeuds du niveau suivant, etc.
- *Meilleur d'abord* : le parcours de l'arbre est dépendant du résultat d'une estimation des noeuds restant à visiter. Selon cette estimation, on choisira alors d'examiner en premier, soit un noeud susceptible de contenir une solution optimale, soit un noeud menant rapidement à une meilleure solution réalisable. Les techniques d'estimation sont similaires, voire identiques, aux techniques d'évaluation. Elles sont aussi utilisées dans les stratégies de profondeur ou largeur d'abord, au moment de choisir l'ordre de sélection des fils d'un noeud.

Comme pour cette dernière stratégie, l'ordre de sélection peut être étroitement lié à la façon dont les noeuds sont créés, c.-à-d. selon le choix de séparation de l'espace.

• ***Schémas de branchement***

En théorie, il existe une infinité de schémas de branchement possibles : l'espace peut être séparé en deux parties ou plus ; les parties peuvent être disjointes (cas le plus fréquent) ou pas. Surtout, la séparation peut se faire le long de n'importe quel plan de l'espace de type $\sum_i \pi_i x_i = \pi_0$, avec $\pi_0 \in \mathbb{Z}$ et $\pi \in \mathbb{Z}^n$. Elle se traduit par l'ajout dans le PLNE, de l'inégalité $\sum_i \pi_i x_i \leq \pi_0$ d'une part, et de l'inégalité $\sum_i \pi_i x_i \geq \pi_0 + 1$ d'autre part (puisque, par intégralité, il n'existe pas de solution x telle que $\pi_0 < \sum_i \pi_i x_i < \pi_0 + 1$).

En pratique, pour la résolution des PLNE généraux, on créé fréquemment un arbre binaire en séparant en deux l'intervalle des valeurs possibles d'une variable, selon un plan défini par $x_i = d$. Dans le cas d'un PLVB, cela revient à fixer la coordonnée x_i à 0 dans un noeud fils et à 1, dans l'autre. Un autre schéma généralise ce cas dans un PLNE où l'intervalle des valeurs de x_i est borné ($0 \leq x_i \leq d$, $d \in \mathbb{N}$). On ajoute alors au noeud courant autant de branches qu'il y a de valeurs entières possibles pour x_i : $x_i = 0$, $x_i = 1$,..., $x_i = d$. Enfin, un dernier schéma courant s'applique aux problèmes où un ensemble de variables Q est sujet à une contrainte GUB (*generalized upper bound*) $\sum_{x \in Q} x = 1$. La séparation se fait selon un sous-ensemble de Q' de Q en posant l'alternative $\sum_{x \in Q'} x = 1$ ou $\sum_{x \in Q \setminus Q'} x = 1$.

• ***Évaluations***

Pour limiter la recherche, un noeud de l'arbre n'est exploré que si on le présume permettant de conduire à une solution optimale, ou, si une solution réalisable $\bar{x}$ est connue à cet instant, à une solution strictement meilleure que $\bar{x}$. À chaque noeud, on évalue ainsi la valeur de la meilleure solution contenue dans le sous-espace E associé au noeud (donc délimité par des plans) à savoir, ici pour notre problème de minimisation, $\min\{cx \mid x \in S \cap E\}$. Puisque ce PLNE est généralement aussi dur que le programme original, on se contente d'encadrer sa valeur optimale z_E par une borne inférieure LB_E (*évaluation par défaut*) et par une borne supérieure UB_E (*évaluation par excès*).

UB_E correspond à la valeur de la meilleure solution réalisable connue ($UB_E = +\infty$ si aucune solution n'est connue). Si on dispose d'une heuristique efficace, il peut être intéressant de commencer par rechercher une solution réalisable dans E la meilleure possible. Dans ce cas, si E ne contient pas de solution réalisable, on pose $UB_E = -\infty$ ou bien si la solution trouvée x_E est meilleure que $\bar{x}$, on met à jour la meileure solution connue $\bar{x}$ avec x_E.

On calcule alors une borne inférieure LB_E de z_E. Si $LB_E \geq UB_E$ alors E ne contient pas

de solution strictement meilleure que $\bar{x}$ et il n'est donc pas nécessaire d'explorer plus en avant ce noeud. Le rôle de l'évaluation par défaut est donc primordial, puisqu'elle permet de ne pas avoir à énumérer explicitement l'ensemble des solutions réalisables du problème. Disposer d'une méthode puissante de calcul de borne inférieure permet d'écarter, tôt dans le processus de recherche, les solutions sub-optimales. L'évaluation par défaut a une autre fonction capitale, celle de guider la recherche. En effet, obtenue en résolvant un problème proche du problème initial, elle permet éventuellement de détecter un point de l'espace, non nécessairement dans l'ensemble des solutions réalisables, mais proche d'une solution optimale. Ainsi, elle donne une indication sur le plan de séparation à introduire au prochain branchement, passant de préférence à proximité de ce point. Autrement dit, la solution du problème approché est tout aussi importante que sa valeur.

Bien que la recherche arborescente s'applique à toute forme de résolution de problèmes d'optimisation combinatoire (comme le backtraking en programmation par contraintes, voir section 1.1.7), les PSE sont particulièrement adaptées à la programmation linéaire en nombres entiers car on possède alors, dans ce cadre précis, des méthodes efficaces d'évaluation par défaut en considérant une solution réalisable du programme dual (voir 1.2.1) ou une *relaxation* du PLNE $\min\{cx \mid x \in S \cap E\}$.

1.2.3 Relaxations et décompositions

Les techniques de relaxation appliquées à un problème de minimisation fournissent une évaluation par défaut de l'optimum, en relâchant les contraintes du problème les plus difficiles à satisfaire, c'est-à-dire, en les supprimant ou bien en les prenant partiellement en compte. On étudie, dans cette section, les principales techniques de relaxation pour un programme linéaire en nombres entiers écrit sous la forme $(IP) : z = \min\{cx \mid Ax = b, x \in \mathbb{N}^n\}$, avec $A \in \mathbb{Z}^{nm}$, $b \in \mathbb{Z}^m$ et $c \in \mathbb{Z}^n$. On reprend les notations : S l'ensemble des solutions réalisables, $conv(S)$, son enveloppe convexe, P le polyèdre $\{x \in \mathbb{R} \mid Ax = b\}$ et x^* une solution optimale de (IP). Pour simplifier l'exposé, on supposera ici le problème réalisable (car si la relaxation est irréalisable alors il en est de même du problème initial), et parfois borné.

La plupart des problèmes d'optimisation combinatoire présentent des structures particulières de sorte qu'ils se modélisent intuitivement en PLNE de grande taille, dans lesquels, on peut reconnaître et isoler, soit des contraintes, soit des variables plus « difficiles » que les autres. Les méthodes de décomposition s'appliquent plus particulièrement à ces programmes en les scindant de façon à résoudre des sous-programmes plus nombreux mais plus rapides à traiter.

Relaxation continue et coupes

- ***Relaxation continue***

Une technique simple de relaxation consiste à ignorer toutes les contraintes d'intégralité. On obtient alors un programme linéaire en variables continues $(RC) : z_{RC} = \min\{cx \mid x \in P\}$, appelé la *relaxation continue* qui peut être résolu par les méthodes exposées à la section 1.2.1. Pour certains problèmes cependant, la borne inférieure ainsi obtenue (la valeur optimale z_{RC} de la relaxation continue, arrondie à la valeur entière supérieure car $z \in \mathbb{Z}$), est parfois bien en-deçà de la valeur optimale de (IP), et surtout, la solution optimale fractionnaire peut être très éloignée de la solution optimale entière. Par exemple [Wolsey 1998], le problème suivant a une unique solution optimale $(5, 0)$ et comme valeur -500, tandis que sa relaxation continue a pour solution $(376/193, 950/193)$ (proche de $(2, 5)$) et pour valeur arrondie -509 :

$$\min\{-100x_1 - 64x_2 \mid 50x_1 + 31x_2 \leq 250\ ,\ 3x_1 - 2x_2 \geq -4\ ,\ x_1 \in \mathbb{N}, x_2 \in \mathbb{N}\}.$$

Ce comportement se rencontre plus souvent encore pour les programmes en variables binaires où la solution fractionnaire contient généralement de nombreuses coordonnées égales à $0,5$.

La faiblesse de la relaxation continue est dûe à la faiblesse de la formulation du problème, à savoir que P est large par rapport à $conv(S)$. Pour améliorer la relaxation continue, on peut ajouter au programme linéaire, comme dans les approches polyédriques (voir section 1.2.2), des inégalités valides qui coupent P. Cette technique permet ainsi de resserrer la formulation du problème (on résoud alors (RCC) : $\min\{cx \mid x \in P'\}$ avec $S \subseteq conv(S) \subseteq P' \subseteq P$) en prenant implicitement en compte les contraintes d'intégralité. Idéalement, on cherche à déterminer des inégalités définissant des *facettes* de $conv(S)$, c'est à dire des plans supports de $conv(S)$ de dimension égale à dim $conv(S) - 1$.

• ***Exemples d'inégalités valides***

Des inégalités valides génériques ont été identifiées, les plus connues étant certainement celles de Chvàtal-Gomory [Gomory 1958, Chvátal 1973] : Étant donnée une inégalité $\pi x \leq \pi_0$ valide pour S (par exemple, $\pi = a_j$ et $\pi_0 = b_j$ pour toute contrainte j de P) et un réel positif u, alors l'inégalité suivante est valide pour S, puisque le membre de gauche est entier et inférieur à $u\pi_0$:

$$\sum_{i=1}^{n} \lfloor u\pi_i \rfloor x_i \leq \lfloor u\pi_0 \rfloor \tag{1.7}$$

Un résultat intéressant est que toute inégalité valide d'un PLNE peut être obtenue de cette façon.

Une autre technique de génération d'inégalités valides consiste à prendre une combinaison linéaire des contraintes :

$$\sum_{j=1}^{m} \mu_j \sum_{i=1}^{n} a_{ij} x_i \leq \sum_{j=1}^{m} \mu_j b_j, \quad \text{avec } \mu \in \mathbb{R}_+^m. \tag{1.8}$$

Une telle inégalité est appelée *contrainte surrogate*. L'efficacité de la coupe dépend évidemment du choix des coefficients μ. Un choix possible [Glover 1975] consiste à prendre pour μ_j la valeur de la j-ème coordonnée de la solution duale de la relaxation continue du programme. L'inégalité s'appelle alors *contrainte surrogate duale*.

Certains schémas apparaissent fréquemment dans les formulations PLNE de problèmes d'optimisation combinatoire. Des inégalités valides existent pour certains d'entre eux. Dans un problème linéaire à variables binaires, par exemple, il est possible souvent d'identifier des ensembles $\{x_i \mid i \in C\}$ de variables ne pouvant pas être affectées simultanément à 1. Pour un ensemble C d'indices, minimal pour cette condition, on déduit l'*inégalité de clique* suivante, imposant qu'au moins une des variables doit être nulle :

$$\sum_{i \in C} x_i \leq |C| - 1. \tag{1.9}$$

Ces méthodologies générales ne permettent pas de déterminer efficacement toutes les facettes d'un PLNE. Il est donc souvent nécessaire d'étudier la structure des problèmes au cas par cas.

• ***Inégalités disjonctives et lifting***

Souvent, il est possible de caractériser plus précisément, pour une variable x_i et un entier d donné, par exemple, les deux ensembles des solutions telles que, respectivement, $x_i \leq d$ ou bien $x_i \geq d + 1$, de sorte que l'on connaît ainsi une meilleure formulation où l'ensemble des solutions continues se présente comme l'union de deux polyèdres P_1 et P_2 disjoints. Ceci traduit en effet une condition logique qui se rencontre fréquemment dans les problèmes d'optimisation combinatoire, du type, *une solution x appartient à P_1 si $x_i \leq d$, et à P_2 sinon*. La *programmation*

disjonctive [Balas 1979] traite de ce genre de formulations en proposant des méthodes de résolution spécifiques ou en se ramenant à la programmation linéaire en recherchant l'enveloppe convexe de $P_1 \cup P_2$. En particulier, dans un programme linéaire en variables binaires, la procédure de *lifting* permet de générer les inégalités caractérisant cette enveloppe convexe en considérant celles définissant séparément P_1 et P_2. On part d'une contrainte vérifiée par l'ensemble des solutions telles que $x_i = 0$ par exemple, puis on cherche à l'étendre (lifting) à l'ensemble de toutes les solutions par projection sur le plan $x_i = 1$. Au chapitre 4, nous reviendrons plus en détail sur cette technique que nous avons employée à la génération de coupes pour le RCPSP.

- ***Génération des coupes***

Comme il a été mentionné à la section 1.2.2, il est possible de combiner approche polyèdrique (génération de coupes) et PSE, on parle alors d'une méthode de *branch and cut*. À chaque itération de la recherche arborescente, l'évaluation par défaut d'un noeud est calculée par relaxation augmentée de coupes. L'introduction des coupes artificielles (les branchements) permet généralement de déterminer de nouvelles inégalités valides plus profondes pour le sous-espace considéré à chaque noeud. Dans la pratique cependant, rechercher de nouvelles coupes à chaque noeud peut ralentir considérablement la procédure. Un compromis consiste à générer préalablement, à la racine, un certain nombre d'inégalités valides et, à un noeud donné qui n'est pas invalidé par la relaxation continue *RC* seule, d'ajouter à *RC* ces inégalités qui coupent la solution optimale de *RC*. Éventuellement, le PL peut être réoptimisé et ce, tant qu'il existe des coupes pour la solution fractionnaire courante.

Les bornes inférieures du RCPSP que nous calculons par génération de coupes (section 4) reprennent ce principe. La relaxation continue du problème est résolue et des inégalités sont itérativement ajoutées et supprimées selon qu'elles coupent ou non la solution fractionnaire. En fait, pour l'une des formulations (en variables S_i en temps continu) que nous avons étudié, nous ne relaxons pas seulement les contraintes d'intégrité, mais aussi les contraintes de ressources difficilement modélisables au moyen des variables S_i. Les coupes ont, ici, pour but de prendre aussi en compte partiellement ces contraintes ignorées. Les algorithmes de génération de coupes peuvent, en effet, être utilisés autrement que pour la relaxation continue. En particulier, ils peuvent générer des inégalités qui coupent l'ensemble des solutions pour s'approcher plus rapidement de l'optimal, ou bien générer progressivement les contraintes mêmes du programme quand elles sont en trop grand nombre.

- ***Décomposition de Benders***

La décomposition de Benders présente une méthode de résolution des problèmes pour lesquels on peut isoler des variables qui, une fois fixées, simplifient le problème. Après reformulation, on se ramène à un programme contenant de nombreuses contraintes, qui ne sont pas identifiées *a priori*, mais générées progressivement par un algorithme de coupes. Par exemple, dans un programme en variables mixtes (entières et réelles), on considère généralement les variables entières comme difficiles :

$$(MIP) : \quad z_{MIP} = \min\{cx + hy \mid Ax + Gy \geq b, x \in X \subseteq \mathbb{N}^n, y \in \mathbb{R}^p_+\}.$$

Si on fixe dans ce programme les variables x, on obtient un programme linéaire (en y) dont le dual s'écrit :

$$(D(x)) : \quad z_x = \max\{u(b - Ax) \mid u \in Q\} \quad \text{avec} \quad Q = \{u \in \mathbb{R}^m_+ \mid Gu \leq h\}.$$

D'après la décomposition de Minkowski, un point u appartient au polyèdre Q si et seulement

s'il s'écrit comme la combinaison linéaire des points extrêmes u^k, $k \in \{1,\dots,K\}$ et des rayons extrêmes v^j, $j \in \{1,\dots,J\}$ de Q : $u = \sum_{k=1}^{K} \lambda_k u^k + \sum_{j=1}^{J} \mu_j v^j$, avec $\sum_{k=1}^{K} \lambda_k = 1$. En supposant (MIP) borné alors Q est non vide et on voit facilement que, pour un x donné, s'il existe un rayon extrême v^j tel que $v^j(b - Ax) > 0$ alors $z_x = +\infty$, sinon $D(x)$ atteint son maximum en un point extrême u^k. (MIP) qui, par dualité, correspond à $\min\{cx + z_x \mid x \in X\}$, peut donc se reformuler par le programme en variables mixtes suivant :

$$(PM): \qquad z_{MIP} = \min cx + z \tag{1.10}$$

sujet à :

$$z \geq u^k(b - Ax) \qquad \forall k \in \{1,\dots,K\} \tag{1.11}$$

$$v^j(b - Ax) \leq 0 \qquad \forall j \in \{1,\dots,J\} \tag{1.12}$$

$$x \in X,\ z \in \mathbb{R} \tag{1.13}$$

Étant donné le nombre de contraintes de (PM), on optimise itérativement une relaxation (PR) en générant une à une les contraintes à la manière de coupes (*coupes de Benders*). À une itération donnée, soit $(\bar{x}, \bar{z})$ la solution optimale de (PR), on résoud alors le sous-problème $(D(\bar{x}))$ en traitant les 3 cas possibles : S'il est non borné (le simplexe, p. ex. , retourne un rayon extrême v^j tel que $v^j(b - A\bar{x}) > 0$) alors $\bar{x}$ n'entre dans aucune solution réalisable de (MIP) et la coupe (1.12) est ajoutée à (PR) interdisant cette solution. Sinon il possède une solution optimale en un point extrême u^k et sa valeur optimale $z_{\bar{x}} = u^k(b - A\bar{x})$ est nécessairement supérieure ou égale à $\bar{z}$. Si elle est strictement supérieure, alors $(\bar{x}, \bar{z})$ n'est pas réalisable et on ajoute l'inégalité (1.11) pour couper cette solution, sinon $(\bar{x}, \bar{z})$ est réalisable et donc optimale pour (PM) et l'algorithme s'arrête.

La convergence de l'algorithme est assurée puisque les coupes de Benders sont en nombre fini et qu'une nouvelle est ajoutée à chaque itération. Cependant si la convergence est trop lente, l'algorithme peut être arrêté, fournissant alors un encadrement de l'optimum (si $z_{\bar{x}} < +\infty$) puisque, à chaque itération, $c\bar{x} + \bar{z} \leq z_{MIP} \leq c\bar{x} + z_{\bar{x}}$.

On reviendra sur cette méthode à la section 1.3.3 car elle présente un cadre particulier d'intégration de la programmation par contraintes.

Génération de colonnes

La *génération de colonnes* a un principe « opposé » à la génération de coupes. Au lieu de supprimer des contraintes difficiles (ou trop nombreuses) et d'en ajouter de nouvelles plus faciles à traiter, il s'agit de partir d'un sous-ensemble de variables puis d'en considérer davantage à chaque itération de la méthode. Cette méthode s'applique essentiellement aux formulations contenant un grand nombre de variables comme, par exemple, pour les problèmes de découpe où on modélise usuellement chaque solution « possible » par une variable et où les colonnes de la matrice A sont donc associées au coût des solutions correspondantes. On commence par optimiser sur un sous-ensemble de solutions, ce qui permet de caractériser d'autres solutions améliorantes. Le programme est augmenté de ces solutions et réoptimisé, et le processus se poursuit tant qu'il existe de meilleures solutions non encore prises en compte.

Formellement, dans le cas d'un problème borné, on cherche à résoudre un programme (IP) $z = \min\{cx \mid Ax = b, x \in X\}$ où X contient un nombre élevé mais fini d'éléments de $\mathbb{Z}^n$: $X = \{x^1,\dots,x^K\}$. La *reformulation de Dantzig-Wolfe* de (IP) contient les variables binaires $\lambda_1,\dots,\lambda_K$ et est obtenue en considérant que $X = \{x \in \mathbb{R}^n \mid x = \sum_{k=1}^{K} \lambda_k x^k, \sum_{k=1}^{K} \lambda_k = 1, \lambda \in \{0,1\}^K\}$. En relâchant les contraintes d'intégrité de la reformulation, on obtient une relaxation de (IP) avec le

programme linéaire maître (LPM) suivant :

$$z_{LPM} = \min\{\sum_{k=1}^{K} \lambda_k c x^k \mid \sum_{k=1}^{K} \lambda_k A x^k = b, \sum_{k=1}^{K} \lambda_k = 1, \lambda \in \mathbb{R}_+^K\}.$$

Si ce problème contient trop de variables pour être résolu tel quel, on commence par résoudre un programme similaire restreint à un sous-ensemble de variables λ_k, disons les K' premières : $(RLPM)$ s'écrit comme (LPM) avec K' à la place de K. Toute solution réalisable de $(RPLM)$, complétée de 0 pour les coordonnées de $K'+1$ à K, est réalisable pour (LPM). En particulier, si $\lambda^* \in \mathbb{R}_+^{K'}$ est une solution optimale de $(RLPM)$ et $(\mu^*, \nu^*) \in \mathbb{R}^m \times \mathbb{R}$ une solution optimale de son dual, alors

$$\mu^* b + \nu^* = c\lambda^* \geq z_{LPM}.$$

On cherche maintenant à savoir si (μ^*, ν^*) est une solution réalisable du dual de (LPM). Par dualité, dans le cas positif, on a alors une égalité dans la relation précédente, autrement dit, $(\lambda^*, 0)$ est une solution optimale de (LPM) et fournit donc une évaluation par défaut de (IP). Pour ça, on résoud généralement un sous-problème d'optimisation $\zeta = \min\{cx - \mu^* Ax - \nu^* \mid x \in X\}$. Si $\zeta \geq 0$, alors (μ^*, ν^*) est une solution duale de (LPM). Sinon on ajoute la colonne correspondant à la solution optimale x^{k^*} $(K' < k^* \leq K)$ au programme restreint $(RPLM)$ et on recommence le processus tant que $\zeta < 0$.

Il est possible aussi d'arrêter avant puisque $(\mu^*, \nu^* + \zeta)$ étant elle-même une solution duale de (LPM), elle fournit une borne inférieure $(\mu^* b + \nu^* + \zeta)$ de (LPM) et donc de (IP). On peut vérifier aussi, que x^{k^*} est une solution réalisable de (IP) car, dans ce cas, elle est nécessairement optimale :

$$z \leq cx^{k^*} = \zeta + \mu^* Ax^{k^*} + \nu = \mu^* b + \nu^* + \zeta \leq z_{LPM} \leq z.$$

Autrement, la génération de colonnes se termine avec une solution fractionnaire (souvent, la plupart des coordonnées λ_k sont entières). Il est nécessaire alors de brancher pour trouver la solution optimale de (IP) (algorithme de *branch and price*).

Relaxation lagrangienne

- ***Relaxation des contraintes***

Pour la relaxation lagrangienne, on suppose encore que le PLNE est décomposable, c'est à dire qu'on peut isoler les contraintes difficiles $(Ax = b)$ des autres, plus faciles $(Dx \leq d)$:

$$(IP) \quad z = \min\{cx \mid Ax = b, x \in X\} \text{ avec } X = \{x \in \mathbb{Z}^n \mid Dx \leq d\}.$$

L'idée ici est de relâcher les contraintes difficiles, non pas en les supprimant totalement, mais en les *dualisant*, autrement dit, en les prenant en compte dans la fonction objectif de sorte qu'elles pénalisent la valeur des solutions qui les violent. On définit ainsi, pour tout $\mu \in \mathbb{R}^m$, la *relaxation lagrangienne de* (IP) *de paramètre* μ :

$$(LP_\mu) \quad z_\mu = \min\{cx + \mu(b - Ax) \mid x \in X\}.$$

μ est appelé *multiplicateur de lagrange*. Il s'agit maintenant de déterminer le multiplicateur μ qui fournit la meilleure relaxation lagrangienne, en résolvant le *problème dual lagrangien* :

$$(DL) \quad z_{DL} = \max\{z_\mu \mid \mu \in \mathbb{R}^m\}.$$

On a ainsi la relation $z_\mu \leq z_{DL} \leq z$, pour tout $\mu \in \mathbb{R}^m$.

• ***Résolution du dual lagrangien***

Il existe plusieurs méthodes pour résoudre le dual lagrangien. La première est la génération de coupes. Elle consiste à reformuler (DL) en considérant, comme pour Dantzig-Wolfe, X comme un ensemble fini de points $x^1, \ldots, x^K$. (DL) s'écrit alors :

$$(DL') \quad z_{DL} = \max\{u \mid u \leq cx^k + \mu(b - Ax^k)\ \forall k \in \{1, \ldots, K\}, \mu \in \mathbb{R}^m, u \in \mathbb{R}\}.$$

Les éléments de X correspondent maintenant à des contraintes qui peuvent être générées et ajoutées au programme de manière progressive, la suite des points x^k étant obtenue en résolvant itérativement (LP_{μ_k}) où μ_k est la solution optimale courante de (DL'). Il est intéressant de noter qu'en passant au dual, on reconnaît exactement le problème maître (LPM) de la section précédente et donc, que la génération de contraintes ici s'exécute de la même façon que la génération de colonnes sur le dual.

Une seconde méthode de résolution du dual lagrangien, la plus utilisée, est la *méthode du sous-gradient* pour maximiser des fonctions concaves, linéaires par morceaux, à savoir ici $\mu \mapsto z_\mu = \min\{cx + \mu(b - Ax) \mid x \in X\}$. Comme précédement, l'algorithme construit itérativement une suite $(\mu_k)_k$ convergeant vers la solution optimale du problème, μ_{k+1} étant le point se trouvant à une distance choisie, le *pas de déplacement*, de μ_k dans une direction donnée par la fonction, plus précisément, dans la direction opposée d'un sous-gradient de la fonction en μ_k. Ayant employé la génération de contraintes et le sous-gradient pour une relaxation lagrangienne du RCPSP, nous reviendrons sur ces deux méthodes au chapitre 3.

Pour pallier au problème d'instabilité de la suite $(\mu_k)_k$ inhérent à la résolution du dual lagrangien, en particulier par génération de contraintes, d'autres méthodes,à mi-chemin entre les deux précédentes techniques, ont été développées comme la *méthode de coupes du centre analytique* (ACCPM) et la *méthode des faisceaux* (voir [Lemaréchal 2001]).

Comparaison des méthodes

On a rapproché, dans leur fonctionnement, génération de coupes et génération de colonnes, ainsi que génération de colonnes et relaxation lagrangienne. En réalité, ces trois méthodes sont comparables au vu de la proposition suivante, qui découle trivialement des résultats des deux sections précédentes, par les propriétés de dualité et de convexité :

Proposition 1

$$z_{RC} \leq z_{DL} = z_{LPM} = \min\{cx \mid Ax \geq b, x \in conv(X)\} \leq z_{IP}$$

La distance $z_{IP} - z_{DL}$ est appelée le *saut de dualité*. Si elle est strictement positive, il est nécessaire alors de brancher pour trouver la solution optimale du PLNE (IP). La solution optimale du dual lagrangien (ou au problème maître (LPM)) correspond à un point $\bar{x}$ généralement proche d'une solution de (IP) et donne donc des indications sur les branchements à effectuer. Elle est aussi un point de départ intéressant d'une heuristique quelconque pour déterminer une bonne solution réalisable de (IP), en considérant uniquement les contraintes $A_j x = b_j$ non satisfaites par $\bar{x}$.

Le choix de la décomposition du problème (les contraintes de X à isoler) est déterminant puisqu'il s'agit de trouver le meilleur compromis entre la facilité de résolution des sous-problèmes et un faible saut de dualité. Encore une fois, la formulation du problème en PLNE doit être choisie avec soin car elle impose généralement la décomposition ou la relaxation particulière la mieux

adaptée. Enfin, avant de résoudre un programme de grande taille, il est important d'y effectuer un *prétraitement* de façon à réduire le nombre de variables, affiner les inégalités et les bornes des variables, supprimer les inégalités redondantes. Cette réduction est fréquemment induite par un simple raisonnement logique sur les contraintes du problème.

1.3 Approches Hybrides PPC-PLNE

Nous n'évoquons dans cette section que la partie de l'études des systèmes hybrides IA-RO consacrée aux moyens d'intégrer programmation par contraintes sur domaines finis et programmation linéaire en nombres entiers pour la résolution des problèmes d'optimisation combinatoire. Pour un traitement complet des approches hybrides de la programmation logique et des méthodes d'optimisation, on pourra se réferer, par exemple, à [Hooker 2000a] ou encore à une collection d'articles [Milano 2003] présentant des techniques plus récentes encore d'intégration. Nous commençons par une comparaison des deux approches (1.3.1) puis nous montrons ce qu'elles peuvent s'apporter et comment elles s'intégrent l'une à l'autre (1.3.2). Nous présentons ensuite les méthodes de la littérature basées sur une coopération réciproque des deux solveurs (1.3.3). Nous terminons sur la notion d'apprentissage, issue de la logique et base du backtracking intelligent en PPC, et ses liens avec la programmation linéaire (1.3.4).

1.3.1 Comparaison des approches

Modélisations

La différence la plus évidente entre les deux approches réside dans la modélisation du problème. Tandis que la programmation linéaire ne permet de traiter les contraintes que sous la forme d'égalités ou d'inégalités linéaires, la programmation par contraintes accepte tout type de relations pour formuler les contraintes d'un CSP, comprenant les inégalités linéaires, mais aussi, les contraintes logiques (p. ex. , sous forme de clause $x = 5 \vee y = 2 \vee z \neq 3$ ou d'implication $x = 0 \Rightarrow y \geq 1 \wedge y+z < 2$), ou encore les contraintes globales. Les contraintes globales sont un moyen de formuler succintement un groupe de contraintes, comme par exemple `all-different`$(x_1, \ldots, x_k)$ pour spécifier l'ensemble des contraintes $x_i \neq x_j$, pour tout $1 \leq i < j \leq k$.

Ce langage de modélisation déclaratif est un atout pour la programmation par contraintes car les problèmes d'optimisation combinatoire présentent généralement des sous-structures indépendantes facilement identifiables, chacune pouvant être formulée par une contrainte globale. Il requiert ainsi une moindre expertise au moment de déterminer une (bonne) formulation du problème. Il est aussi flexible et permet de prendre plus facilement en compte de nouvelles contraintes, sans avoir à reconsidérer entièrement la méthode. En effet, ce type de modélisation, avec en particulier les contraintes globales, induit une décomposition du problème sur laquelle est basé le mode de résolution de la PPC. À chaque contrainte globale, est associé un ou plusieurs algorithmes de filtrage spécifiques. Ces algorithmes prennent en compte le groupe de contraintes, sous-jacent à la contrainte globale, dans son ensemble et de manière plus efficace que les techniques de propagation standard appliquées aux contraintes séparées. De tels algorithmes ont fait l'objet d'études avancées du fait qu'une même contrainte globale peut modéliser une sous-structure commune à de nombreux problèmes d'optimisation. La contrainte globale `cumulative` permet, par exemple, de modéliser l'occupation d'une ressource dans n'importe quel problème d'ordonnancement, mais elle apparaît aussi dans les problèmes de tournée de véhicules. Les algorithmes associés, issus d'ailleurs principalement de la RO (voir section 2.2), s'appliquent ainsi à tous ces problèmes, ou plutôt à leurs sous-problèmes

de façon à filtrer davantage. La propagation de contraintes se charge alors de faire partager les réductions de domaines à toutes les contraintes du problème.

Le principe d'associer algorithmes et contraintes est une notion importante en PPC et qui marque sa différence fondamentale d'avec la PLNE où, hormis la condition d'intégralité, les contraintes ont toutes plus ou moins un rôle identique. À l'inverse, en imposant un langage de modélisation plus strict, la programmation linéaire a permis de développer des techniques de résolution sophistiquées et aussi moins spécifiques aux (sous-)problèmes : elles s'appliquent, plus ou moins, à tout type de programme linéaire et considèrent le problème d'une manière plus globale. La résolution d'un programme linéaire est peu dépendante de la forme des inégalités que celui-ci contient, tandis qu'en PPC, s'il n'existe pas de techniques de filtrage dédiées, la propagation standard sera peu efficace sur les contraintes portant sur plus de deux variables.

Cette distinction entre les modèles et donc entre les modes de résolution PPC et PLNE, au lieu d'être un argument opposé à l'hybridation des deux méthodes, est plutôt une preuve de leur complémentarité. Les schémas d'intégration les plus récents reposent sur des modélisations mixtes PPC/PLNE où les contraintes d'un problème sont formulées et traitées soit par PPC, soit par PL, soit par les deux méthodes à la fois.

Méthodes de résolution

Cette intégration est facilitée par le fait que les schémas de résolution des deux approches sont similaires. Tous deux sont en effet basés sur trois principes : le partitionnement de l'espace de recherche, l'étude de relaxations et l'inférence de nouvelles contraintes.

Le partitionnement de l'espace de recherche est le point de départ de la stratégie « diviser et conquérir », la plus employée à la fois en PPC (backtracking) et en PLNE (PSE). Tout l'espace est bien sûr considéré, mais à chaque moment, seule une fraction est étudiée, à la recherche d'une solution strictement meilleure que celles trouvées dans les sous-espaces précédement étudiés. Les méthodes arborescentes séparent ainsi progressivement l'espace en parties de plus en plus fines jusqu'à ce que, soit une solution optimale dans le sous-espace courant est exhibée, soit on prouve qu'aucune meilleure solution n'est contenue dans ce sous-espace.

À l'opposé du partitionnement où le problème est simplifié en étant sur-contraint, une relaxation considère le problème sans tenir compte des contraintes les plus dures, élargissant ainsi la recherche, pour la faciliter, à des solutions non-réalisables. Si le terme « relaxation » est clairement défini en PLNE, avec en premier lieu la relaxation continue, il est sous-jacent au maintien de la consistance en PPC où il désigne simplement l'ensemble des contraintes de domaines (*constraint store*). Une solution de la relaxation en PPC consiste en une instanciation des variables à une valeur quelconque de leurs domaines, mais, à moins que l'espace ne soit réduit à cette solution uniquement, elle n'est pas explicitée, contrairement à la PLNE où la relaxation est effectivement résolue. Dans les deux approches, on cherche à prouver que la relaxation (et donc le problème original) ne contient pas de (meilleure) solution : en PPC, si le domaine d'une variable est vide ; en PLNE, si la solution optimale de la relaxation est moins bonne que la solution courante du problème.

Enfin, pour resserrer la formulation de la relaxation, de nouvelles contraintes peuvent être inférées des contraintes initiales du problème et ajoutées à la relaxation. Cela se traduit, en PLNE, par la génération de coupes pour le programme linéaire relâché, et en PPC, par les techniques de filtrage et la propagation de contraintes induisant une réduction des domaines des variables.

Ce schéma modulaire commun présente une voie d'unification des deux approches puisque, par exemple, dans une méthode arborescente, la relaxation à chaque noeud peut être résolue alternativement ou simultanément par propagation de contraintes ou par programmation linéaire. De

même, les contraintes inférées par l'une des deux méthodes peuvent éventuellement être traduites puis utilisées par l'autre. Ainsi, la plupart des méthodes coopératives emploient le pouvoir d'inférence de la PPC pour déduire les coupes d'une relaxation formulée par un programme linéaire. À son tour, le programme linéaire renvoie alors une instanciation complète des variables proche d'une solution optimale, permettant de guider la recherche mais aussi de supprimer le noeud s'il est sub-optimal.

La dualité est une autre notion commune aux deux approches. En effet, la formulation du dual en programmation linéaire a l'interprétation logique suivante [Hooker 2000b] : Si f est une fonction définie au moins sur un ensemble S et à valeurs dans un ensemble ordonné, disons $\mathbb{R}$, alors le dual du problème $\min\{f(x) \mid x \in S\}$ s'écrit $\max\{\beta \mid x \in S \Rightarrow f(x) \geq \beta\}$. Cette remarque permet d'appliquer à la PPC certaines techniques linéaires considérant une formulation duale du problème, et en particulier, la décomposition de Benders.

Optimisation

Même si la PPC permet de résoudre des problèmes d'optimisation, le rôle du critère d'optimisation y est moins important qu'en PL. En effet, pour un problème de minimisation par exemple, la fonction objectif $f(x)$ en PPC est généralement formulée au moyen d'une *contrainte de borne* $f(x) < ub$ où ub est la valeur de la meilleure solution connue et est donc graduellement diminuée au cours de la recherche. Cette contrainte est simplement prise en compte par la propagation comme une contrainte de réalisabilité. De plus, si f porte sur de nombreuses variables, les techniques de réduction de domaines s'avérent parfois inefficaces sur ce type de contraintes.

Au contraire, la résolution d'un PLNE repose sur une utilisation intensive du critère d'optimisation par le biais des relaxations. Déjà, la solution du problème relâché, proche d'une solution réalisable optimale, sert à guider la recherche en fournissant une bonne indication sur la prochaine séparation de l'espace à effectuer. En fait, elle fournit une information plus précise encore, car toute solution optimale d'un programme linéaire renseigne, par l'intermédiaire des *coûts réduits*, sur la variation de son coût si la valeur de ses coordonnées hors-base est modifiée. En particulier, dans un PLVB, si on met à 1 une variable hors-base (égale à 0 dans la solution relâchée) et que, de ce fait, le coût de la solution relâchée dépasse le coût de la meilleure solution entière connue alors on peut conclure que cette variable est égale à 0 dans toute solution optimale de l'espace courant. On supprime ainsi la valeur 1 du domaine de cette variable, autrement dit on fixe la variable à 0. Cette technique (*reduced cost fixing*), courante en PLVB et qui se généralise en PLNE aux bornes inférieures et supérieures des variables, est clairement une technique de réduction de domaines au sens de la PPC. La propagation de contraintes basée sur les coûts réduits a récemment été étudiée et appliquée avec succès (voir section suivante).

1.3.2 Modes d'intégration

La comparaison des deux approches met en valeur leur complémentarité et leurs avantages respectifs : l'inférence pour la PPC, la relaxation et le raisonnement global sur le critère d'optimisation pour la PL. Nous détaillons dans cette section comment les techniques PPC permettent de resserrer un programme linéaire et, inversement, comment la résolution d'un PL peut être employée comme technique de filtrage de contraintes globales en PPC. Les formulations PPC et PL d'un même problème peuvent communiquer trivialement par la réduction des bornes des domaines des variables qu'elles ont en commun. Une collaboration plus poussée requiert une traduction entre les deux modèles, à savoir la linéarisation des contraintes logiques et globales.

Linéarisation

- ***Contraintes symboliques***

La linéarisation automatique des contraintes symboliques a fait l'objet de travaux plus anciens puisqu'elle était déjà invoquée pour l'intégration du raisonnement de la logique propositionnelle en programmation linéaire, comme dans les méthodes booléennes ou en programmation disjonctive.

Il existe différentes manières de modéliser, par un ensemble d'inégalités linéaires, une contrainte disjonctive de la forme

$$(A^1 x \leq b^1) \vee \ldots \vee (A^k x \leq b^k), \tag{1.14}$$

où $x \in \mathbb{R}^n_+$, A^i est une matrice réelle $m^j \times n$ et b^j un vecteur de $\mathbb{R}^{m^j}$. Celle-ci peut s'écrire par la *formulation disjonctive* [Balas 1979], dont la relaxation continue recouvre chacun des polyèdres définis par $A^j x \leq b^j$:

$$\{y_j \in \{0,1\}, x_j \in \mathbb{R}^n_+ \forall j \in \{1,\ldots,k\} \mid A^j x_j \leq b^j y_j \ \forall j \in \{1,\ldots,k\},\ x = x_1 + \ldots, x_k,\ y_1 + \ldots + y_k \geq 1\ \},$$

ou, alternativement, au moyen de constantes M^j suffisamment grandes :

$$\{y \in \{0,1\}^k \mid A^j x \leq b^j + M^j(1 - y_j)\ \forall j \in \{1,\ldots,k\},\ y_1 + \ldots + y_k \geq 1\ \}.$$

La première modélisation pêche par le nombre de variables additionnelles à prendre en compte, tandis que la seconde remédie à cela au détriment de contraintes plus lâches, induisant généralement une faible relaxation continue (par projection sur x, dans ce dernier cas, les contraintes peuvent éventuellement être resserrées [Beaumont 1990]).

Le principe d'associer une variable binaire à chacun des termes d'une disjonction se généralise à tout type de disjonctions. Le problème de linéarisation se reporte alors sur la représentation de la relation entre la variable et le littéral correspondant. Par exemple, la disjunction $A_1 \vee \neg A_2 \vee \neg A_3$ se traduit par $y_1 + (1 - y_2) + (1 - y_3) \geq 1$ où la variable binaire $y_i = 1$ si A_i est *vrai*. Pour spécifier le « ou exclusif », l'inégalité est remplacée par l'égalité.

La procédure de linéarisation automatique présentée dans [Rodosek 1997] repose essentiellement sur la relaxation basée sur les constantes M. Dans [Refalo 2000], Refalo utilise plutôt la formulation disjonctive, introduisant ainsi un grand nombre de variables additionnelles mais qui sont communes aux linéarisations de chacune des contraintes du problème, y compris les contraintes globales : pour chaque variable x d'un CSP et pour chaque valeur d dans son domaine D_x, la relaxation linéaire de ce CSP contient une variable binaire $y_{x=d}$, avec la correspondance $y_{x=d} = 1 \leftrightarrow x = d$. En particulier, dans cette formulation, la contrainte de domaine portant sur x s'écrit :

$$(x = \sum_{d \in D_x} d y_{x=d}), \quad \sum_{d \in D_x} y_{x=d} = 1, \quad y_{x=d} \in \{0,1\}\ \forall d \in D_x. \tag{1.15}$$

- ***Contraintes globales***

Comme pour les contraintes disjonctives, toute contrainte globale possède plusieurs relaxations linéaires. En fait, on peut généralement réécrire une contrainte globale sous la forme (1.14) et ainsi, associer une variable binaire à chaque « atome » de la contrainte.

La contrainte globale `element`$(x, [d_1, \ldots, d_n], z)$, par exemple, se rencontre fréquemment car elle permet de modéliser les variables en indice : elle est satisfaite si la variable z prend la valeur d_x où x est une variable à valeurs dans un sous-ensemble D_x de $\{1,\ldots,n\}$. Elle s'écrit donc encore

$\bigvee_{k \in D_x}(z = d_k)$ et, dans sa formulation disjonctive :

$$x = \sum_{k \in D_x} k y_{x=k}, \quad \sum_{k \in D_x} y_{x=k} = 1, \quad z = \sum_{k \in D_x} d_k y_{x=k}.$$

En particulier, dans la linéarisation de Refalo, seule la dernière inégalité sera ajoutée au programme puisque les deux précédentes font partie de la définition de la contrainte de domaine de x. La linéarisation de la contrainte `element` et de ses extensions est aussi étudiée dans [Thorsteinsson 2001b, Hooker 2000a].

Avec le même type de variables, la contrainte `all-different`$(x_1, \ldots, x_n)$ se linéarise trivialement par :

$$\sum_{i=1}^{n} y_{x_i=d} \leq 1 \quad \forall d \in D = D_1 \cup \ldots \cup D_n.$$

Au cas où $D_1 = \ldots = D_n$ contiennent exactement n valeurs distinctes, ces inégalités avec les contraintes de domaine (1.15) des variables x_i forment l'ensemble des solutions réalisables d'un problème d'affectation [Focacci 2002]. Dans ce même cas, une autre formulation est présentée dans [Williams 2001], consistant en les inégalités définissant l'enveloppe convexe des points $(x_1, \ldots, x_n)$ vérifiant la contrainte `all-different`.

Les relaxations linéaires d'autres contraintes globales ont été étudiées récemment comme, par exemple, `piecewise linear` modélisant une fonction linéaire par morceaux [Refalo 1999, Ottosson 2002] ou `global cardinality` qui généralise `all-different` en imposant le nombre exact de variables devant être affectées à une même valeur [Milano 2002, Williams 2001].

La linéarisation de la contrainte `cumulative` a été utilisée dans des méthodes hybrides pour des problèmes particuliers d'ordonnancement et de transport et sous une forme sévèrement relâchée [Jain 2001, Thorsteinsson 2001a]. Ces relaxations, bien que très faibles, se sont avérées de première importance dans l'efficacité des deux méthodes. À la suite de ces résultats ou encore des travaux de Heipcke [Heipcke 1999], une étude de la relaxation linéaire de `cumulative` a été menée par Hooker et Yan [Hooker 2002]. Les auteurs identifient des inégalités valides pour l'ensemble des solutions S satisfaisant une contrainte `cumulative`(S, p, r, R) ($\sum_{i \in \mathcal{A}_t} r_i \leq R$ à tout temps t où $\mathcal{A}_t$ est l'ensemble des activités i en cours d'exécution au temps $t : S_i \leq t < S_i + p_i$). Ces inégalités sont de la forme $\sum_{i \in J} S_i \geq h$ où J est un sous-ensemble d'activités. En fait, les travaux menés en ordonnancement sur la programmation linéaire (voir p. ex. les formulations du RCPSP à la section 2.3) et, plus précisément, les approches polyédriques [Queyranne 1994, Schultz 1996] peuvent aussi être utilisés pour la linéarisation de la contrainte `cumulative` dans le but de développer des approches hybrides. Les coupes linéaires, basées sur les techniques de filtrage de cette même contrainte, que nous avons développées pour le RCPSP entrent dans ce cadre de linéarisation et hybridation (voir chapitre 4).

Propagation pour resserrer la relaxation linéaire

- ***Prétraitement et réduction des domaines***

Dan un degré basique d'hybridation, on peut considérer l'exemple [Hajian 1995] de la résolution d'un problème d'affectation des vols d'une compagnie aérienne où la PPC sert d'heuristique pour déterminer rapidement une solution réalisable de départ, à la racine uniquement d'une PSE classique. On entend généralement par méthodes hybrides un degré supérieur d'intégration comme une recherche arborescente où les relaxations sont traitées à la fois par PPC et par PL, ou bien une méthode de décomposition où le problème maître et les sous-problèmes sont résolus, chacun par une méthode précise.

Une manière d'intégrer PPC et PLNE dans la recherche d'une solution optimale consiste à modéliser le problème dans chacun des deux formalismes et, dans une méthode arborescente, de résoudre à chaque noeud les deux relaxations correspondantes. Cette stratégie est intéressante pour traiter des problèmes auxquels s'appliquent des techniques de PPC efficaces mais où la relaxation linéaire est mauvaise. À chaque noeud, on commence alors par réduire la relaxation PPC par les règles de filtrage. Si l'irréalisabilité n'est pas prouvée, on poursuit en résolvant la relaxation linéaire qui peut, à son tour, prouver l'irréalisabilité ou, sinon, retourner une solution relâchée aidant à déterminer la prochaine séparation à effectuer.

Une amélioration évidente de cette procédure consiste à utiliser le résultat de la réduction des domaines par la PPC en *prétraitement* de la relaxation linéaire. Le raisonnement logique de la PPC permet de resserrer dynamiquement le modèle linéaire en déduisant des inégalités plus fortes ou en fixant la valeur de certaines variables.

Quand les formulations par contraintes et linéaire partagent les mêmes variables, par exemple, la réduction des bornes des domaines s'appliquent immédiatement au PL [Beringer 1994]. La linéarisation automatique de [Refalo 2000] permet en plus de prendre facilement en compte les valeurs à l'intérieur des domaines en fixant à 1 la variable binaire $y_{x=d}$ si le domaine de x est réduit à d par PPC, et à 0 si la valeur d est supprimée du domaine de x. L'avantage de cette méthode est qu'elle maintient la structure du PL même après réduction des domaines, permettant ainsi une réoptimisation efficace après branchement.

- ***Coupes Logiques***

Outre la réduction de domaines, toute contrainte redondante inférée par propagation est valide pour la relaxation linéaire du problème. Une première application de la linéarisation consiste ainsi à traduire ces contraintes en inégalités linéaires en les variables du PL. Ces inégalités sont alors, soit ajoutées en prétraitement du modèle linéaire, soit générées de manière itérative pendant sa résolution. La linéarisation automatique de [Refalo 2000], par exemple, identifie et isole les inégalités de base de la relaxation, de celles, plus difficiles, à ajouter ultérieurement en tant que coupes.

Dans [Bockmayr 2003], une procédure hybride de branch and cut est proposée pour résoudre un PLNE augmenté d'une contrainte non-linéaire de type $f(x) = 0$ où f est une fonction monotone à valeurs binaires. Cette *contrainte monotone*, en même temps que la solution $\bar{x}$ de la relaxation continue du PLNE, est traitée séparément par PPC de façon à générer deux types de coupes pour séparer $\bar{x}$. Le principe est appliqué à un problème d'ordonnancement où les contraintes de ressources sont modélisées sous la forme d'une contrainte monotone.

Toutes nos techniques de résolution reposent en partie sur ce principe de prétraiter, par PPC, une relaxation linéaire du RCPSP. Plus particulièrement encore, les bornes inférieures du chapitre 4 sont obtenues en ajoutant au PL des inégalités valides générées à partir des inférences de la PPC.

Filtrage par programmation linéaire

Utiliser la programmation linéaire en PPC pour l'optimisation d'un CSP (CSOP) vise à davantage tenir compte de la fonction objectif puisque, dans une approche PPC pure pour un problème de minimisation, seule est considérée une borne supérieure de la solution optimale, par l'intermédiaire d'une contrainte $f(x) < ub$ et que, de plus, cette contrainte est généralement mal propagée, faute d'algorithmes spécifiques.

Dans une recherche arborescente hybride où, à chaque noeud, deux relaxations PL et PPC sont invoquées en parallèle (ou plus rarement PL avant PPC), le modèle linéaire peut aider à guider la recherche ; il peut aussi communiquer au modèle PPC sa valeur optimale lb (ajoutant une contrainte de borne inférieure $lb \leq f(x)$) ou encore, symétriquement à la section précédente,

lui communiquer l'infaisabilité ou des variables fixées à une valeur. En programmation linéaire, les variables peuvent être fixées par calcul des coûts réduits.

Plus généralement, les coûts réduits sont la base d'une nouvelle technique de filtrage en PPC pour les problèmes d'optimisation. Pris en compte, ils permettent d'éliminer, non plus seulement des solutions irréalisables, mais aussi des solutions sub-optimales. Focacci et al. [Focacci 2002] proposent d'intégrer un tel algorithme de filtrage dans deux contraintes globales : `all-different` et sa généralisation `path`. L'algorithme consiste à résoudre le problème d'affectation (AP) équivalent à `all-different`, et donc aussi une relaxation de `path`, sous l'objectif de minimiser la somme des coûts ; le coût associé à une variable de (AP) étant le coût de l'instanciation d'une variable à une valeur, dans le CSOP initial. Les coûts réduits renseignent alors sur des instanciations sub-optimales et réduisent ainsi les domaines des variables. D'autres algorithmes de filtrage basés sur les coûts réduits ont été développés comme par exemple pour la contrainte globale `element` [Thorsteinsson 2001b]. Dans [Milano 2002], on trouvera une étude plus complète du rôle de la programmation linéaire dans les contraintes globales.

Les bornes inférieures destructives (voir [Brucker 2000] et chapitres 3 et 4) entrent aussi dans ce schéma, en considérant la résolution d'un PL comme une contrainte globale, non pas pour filtrer les domaines, mais pour prouver directement l'infaisabilité.

1.3.3 Schémas de coopération

Les approches basées sur une réelle collaboration entre les deux solveurs gèrent en parallèle une formulation PPC et une formulation PLNE et utilisent tour à tour, comme indiqué ci-avant, les inférences d'un modèle pour resserrer la relaxation de l'autre. Une interaction efficace consiste à faire partager des informations qui sont facilement déduites par l'une des deux méthodes. Ainsi, la plupart des approches coopératives étudient les sous-structures des problèmes pour isoler celles qui seront mieux traitées par PPC de celles qui seront mieux traitées par PL.

Branch and infer

Le schéma général d'intégration *branch and infer* proposé par Bockmayr et Kasper [Bockmayr 1998] met en exergue ce principe en identifiant, pour chacune des deux approches, le type de contraintes pouvant être qualifiées de *primitives*, s'il existe un algorithme efficace (complet et au moins polynomial) de résolution, ou bien de *non-primitives*, sinon. En PPC, la classe des contraintes primitives est plutôt réduite (les contraintes de domaine $x \leq d, x \geq d, x \neq d, x \in \mathbb{Z}$ et l'égalité entre deux variables $x = y$) tandis qu'en PLNE, seules les contraintes d'intégrité sont non-primitives. Branch and infer est une méthode arborescente théorique où les relaxations sont formées des contraintes primitives des deux modèles et l'inférence consiste à déduire (de manière incomplète donc) des contraintes primitives (PPC et PL) à partir des contraintes non-primitives.

En pratique, plusieurs méthodes reposent ce principe où les relaxations sont formulées de manière explicite ([Beringer 1994, Heipcke 1999]) ou par traduction systématique ([Rodosek 1997, Refalo 1999]) et où les contraintes primitives inférées sont toutes ou en partie échangées entre les modèles.

Programmation logique/linéaire mixte (MLLP)

Hooker et al. [Hooker 1999, Hooker 2000b] ont proposé une méthode coopérative (MLLP) basée sur une modélisation mixte logique/linéaire de la forme :

$$\begin{array}{lll} \min & cx & \\ \text{s.à :} & h_i(y) \rightarrow A^i x \geq b^i, & i \in I, \\ & g_k(y), & k \in K, \\ & y \in D,\ x \in \mathbb{R}^n. & \end{array}$$

où une solution $(y, x) \in D \times \mathbb{R}^n$ est réalisable si y satisfait les contraintes logiques $g_k(y)$ et si pour toute contrainte logique $h_i(y)$ satisfaite alors x vérifie le système d'inégalités $A^i x \geq b^i$.

La méthode de résolution proposée est une recherche arborescente où la séparation s'effectue sur les variables discrètes y. L'évaluation d'un noeud (caractérisant un sous-espace de recherche $D' \times \mathbb{R}^n$ avec $D' \subseteq D$) s'effectue en trois temps. On commence par effectuer un filtrage PPC sur les contraintes $g_k(y)$ pour tout $k \in K$ avec $y \in D'$ et le domaine D' est éventuellement réduit à D''. On résoud alors le programme linéaire $\min\{cx \mid A^i x \geq b^i\ \forall i \in I',\ x \in \mathbb{R}^n\}$ où I' est l'ensemble des indices $i \in I$ tels que $h_i(y)$ est satisfaite pour tout $y \in D''$. Enfin, soit x^* la solution optimale de ce PL, alors on ajoute une nouvelle contrainte $\neg h_i(y)$ au modèle MLLP pour tout $i \in I \setminus I'$ tel que $A^i x^* < b^i$ et on réeffectue un filtrage par PPC sur les contraintes $g_k(y)$ augmentées de ces nouvelles contraintes. L'infaisabilité peut être détectée par la PPC ou la PL et si aucune contrainte n'est ajoutée à la troisième étape de l'évaluation, alors toute instanciation de y dans D'' correspond à une solution optimale (y, x^*) du noeud.

Un programme disjonctif peut facilement se modéliser en MLLP. Par exemple, en ordonnancement, le fait que deux activités i et j ne peuvent s'exécuter simultanément faute de ressources disponibles, se traduit en programmation disjonctive par une contrainte $(S_j \geq S_i + p_i) \vee (S_i \geq S_j + p_j)$ où S_i est la date de début de l'activité i et p_i sa durée. La linéarisation d'une telle contrainte se fait généralement au moyen d'une variable binaire x_{ij} (égale à 1 si l'exécution de l'activité j débute après la fin de l'exécution de i) et de contraintes « grand-M » : $S_j \geq S_i + p_i - M(1 - x_{ij})$ et $S_i \geq S_j + p_j - M x_{ij}$. En MLLP, on pourrait écrire plutôt $x_{ij} = 1 \rightarrow S_j \geq S_i + p_i$ et $x_{ij} = 0 \rightarrow S_i \geq S_j + p_j$.

La linéarisation partielle d'un modèle PPC permet aussi de reformuler le problème en un modèle MLLP. Si le modèle PPC contient une contrainte globale $h(y)$ dont on connait une relaxation linéaire $\mathcal{L}_{h(y)}(x)$ alors, en y ajoutant les variables x de la linéarisation et la contrainte conditionnelle $h(y) \rightarrow \mathcal{L}_{h(y)}(x)$, on forme ainsi un modèle MLLP. Pour un problème de configuration, la procédure a ainsi été étendue par [Thorsteinsson 2001b] en linéarisant la contrainte globale `element` puis en utilisant les coûts réduits pour réduire les domaines, à la fois des variables discrètes et des variables de la linéarisation.

Génération de colonnes, relaxation lagrangienne et PPC

L'efficacité de la MLLP est conditionnée par le choix du modèle et donc de la décomposition du problème. Dans un sens, elle s'apparente à un *branch and cut* où les « coupes » sont déterminées par PL et ajoutées à la PPC. Les méthodes de décomposition hybrides se présentent généralement dans le contexte inverse, plus intuitif, où le problème maître, résolu par PL, retourne une solution optimale à un sous-problème chargé de vérifier, par PPC, qu'elle satisfait toutes les contraintes. Pour certains problèmes complexes très structurés et se modélisant en PLNE de grande taille, la génération de colonnes ou la relaxation lagrangienne, sont des outils de résolution privilégiés.

Un tel schéma de génération de colonnes basée sur la programmation par contraintes est présenté dans [Junker 1999]. Pour reprendre les notations utilisées dans la présentation de la génération de colonnes à la section 1.2.3, le sous-problème n'est plus un problème d'optimisation mais un CSP ayant pour contraintes $x \in X$ et $cx - \mu^* Ax - \nu^* < 0$. Si le CSP est insatisfiable, le problème est résolu, sinon toute solution x est une colonne améliorante du programme linéaire maître restreint.

La technique duale, la relaxation lagrangienne où les sous-problèmes lagrangiens sont résolus par programmation par contraintes, a été appliquée par exemple dans [Sellmann 2003, Benoist 2001] (bien que dans ces deux articles, des algorithmes spécifiques d'optimisation sont appliqués à la place d'un programme linéaire).

Décomposition de Benders hybride

Un cadre d'hybridation plus prometteur encore est la décomposition de Benders et une majorité des travaux d'intégration PPC/PLNE porte sur ce sujet. Au-delà du formalisme mathématique, la méthode de Benders a en effet une interprétation pratique plus générale : On cherche à résoudre un problème (supposé borné) $z = \min\{f(x,y) \mid (x,y) \in S,\ x \in X,\ y \in Y\ \}$, de manière itérative sur les instanciations possibles des variables x les plus difficiles, en partant d'une instanciation $x^1 \in X$ et de l'évaluation par défaut $z^1 = -\infty$ de z. À une itération k, on résoud le problème sur le sous-espace de recherche défini par l'instanciation courante x^k de x qui s'écrit $z(x^k) = \min\{f(x^k,y) \mid (x^k,y) \in S,\ y \in Y\ \}$. En résolvant plutôt le dual de ce sous-problème, on déduit en fait une fonction $\beta^k(x)$ telle que $\beta^k(x^k) = z(x^k)$ et qui borne inférieurement la valeur optimale $z(x)$ des sous-problèmes pour toutes instanciations de x. La condition $z(x) \geq \beta^k(x)$, appelée *coupe de Benders*, peut, quand $z(x^k) = +\infty$, s'écrire alternativement par tout autre contrainte interdisant les instanciations de x telles que $\beta^k(x) = +\infty$. Si $z^k < z(x^k)$, alors on résoud le problème maître $z^{k+1} = \min\{z \mid z \geq \beta^k(x),\ x \in X,\ z \in \mathbb{R}\ \}$ (z^{k+1} est bien encore une évaluation par défaut de z), on pose x^{k+1} la solution courante et on réitère avec $k = k+1$; sinon, puisqu'on a l'encadrement $z^k \leq z \leq z(x^k)$, la procédure s'arrête.

Cet algorithme est assuré de terminer si D_x est fini ou bien, comme en PLNE, s'il n'existe qu'un nombre fini de coupes de Benders correspondant chacune à un point ou rayon extrême du polyèdre dual des sous-problèmes. La rapidité de convergence est clairement dépendante de la force des coupes de Benders, autrement dit, du nombre d'instanciations possibles pour x supprimées de la recherche à chaque ajout de coupes.

Le principe de dualité en PL appliqué aux sous-problèmes permet d'identifier des coupes fortes, et les coupes de Benders dans le cas général ci-dessus jouent clairement le rôle de *nogoods* en PPC. Une décomposition de Benders hybride faisant appel à la PPC pour la résolution du programme maître et à la PL pour les sous-problèmes a ainsi été appliquée, par exemple dans [Eremin 2001] à un problème d'ordonnancement.

Cependant, il est tout à fait possible d'inverser les rôles des deux solveurs, pour plusieurs raisons : Premièrement, la dualité n'est pas une notion propre à la PL (voir section 1.3.1) et s'applique à tout problème d'optimisation. De plus, si on assure la convergence autrement, il n'est pas besoin dans l'algorithme précédent de résoudre le dual à l'optimum : une solution duale réalisable fournit aussi une borne inférieure et on peut donc résoudre un problème de satisfiabilité pour déterminer une coupe de Benders. Enfin, la puissance des techniques d'inférence en PPC permet aussi de déduire de bonnes coupes.

Dans [Jain 2001], par exemple, Jain et Grossmann ont utilisé une décomposition de Benders hybride pour un problème d'ordonnancement à machines parallèles et fenêtres de temps, où chaque activité doit être exécutée sur une seule machine avec un coût et une durée dépendants de la

machine. Dans leur procédure, le problème d'affectation des activités sur les machines est séparé des problèmes d'ordonnancement sur chacune des machines. Le premier, le problème maître, est résolu par PLVB en minimisant la somme des coûts d'affectation puis la PPC vérifie que cette affectation optimale permet un ordonnancement des activités dans leurs fenêtres de temps sur chacune des machines. Si l'ordonnancement sur au moins une machine est irréalisable, des coupes sont ajoutées au PLVB excluant l'affectation courante et d'autres similaires, et le processus est réitéré.

Branch and check

En étudiant cette dernière méthode, Thorteinsson a observé que son efficacité n'était pas seulement due à la qualité des coupes générées mais surtout à la prise en compte dans le programme maître d'une relaxation linéaire du sous-problème, ce qui constitue une amélioration de l'algorithme de Benders classique. Il a ainsi développé un schéma de résolution, *branch and check* [Thorsteinsson 2001a], qui généralise à la fois la méthode de Benders (dans branch and check, le programme maître est non contraint à la première itération et contient une relaxation du sous-problème) et la PSE classique (dans branch-and-check, la résolution d'une partie du problème est retardée). Ce schéma s'applique, en particulier, à des problèmes sous un modèle mixte de la forme :

$$\begin{array}{ll} \min & cx + f(y) \\ \text{s.à :} & Ax \leq b, \\ & x \sim y, \\ & F(y), \\ & x \in X, y \in Y. \end{array}$$

où les relations $x \sim y$ établissent une correspondance entre les variables x de la partie linéaire du problème et les variables y de la partie « symbolique », de sorte que la fixation d'une variable dans un modèle se traduit par la fixation, ou au moins la réduction de domaine, de la (des) variable(s) associée(s) dans l'autre modèle. Connaissant une relaxation linéaire, éventuellement générée dynamiquement, $\mathcal{L}_{F(y)}(x)$ et $\mathcal{L}_{f(y)}(x)$ de la partie symbolique, le problème est décomposé en un problème maître :

$$\begin{array}{ll} \min & cx + z \\ \text{s.à :} & Ax \leq b, \\ & z \geq \mathcal{L}_{f(y)}(x), \\ & \mathcal{L}_{F(y)}(x), \\ & x \in X, \end{array}$$

résolu par PSE en supposant X discret, et d'un sous-problème, résolu p. ex. par PPC :

$$\begin{array}{ll} \min & f(y) \\ \text{s.à :} & F(y), \\ & y \in Y. \end{array}$$

La résolution du sous-problème peut être invoquée à tout moment *pendant* la résolution du programme maître, c.-à-d. à n'importe quel noeud de l'arbre de recherche, après lui avoir communiqué (par $x \sim y$) les nouvelles fixations des variables x. Le sous-problème retourne et ajoute alors au programme maître, soit un nogood $N(x)$ s'il est infaisable, soit sinon une contrainte de borne inférieure $z \geq lb(x)$.

La caractéristique principale de branch and check, à savoir la génération de coupes de Benders

pendant la résolution du programme maître, a aussi été étudiée dans [Hooker 2000a] où les variables x non encore instanciées sont prises en compte dans le sous-problème. Ces méthodes se différencient d'un branch and cut dans le sens où les coupes générées à un noeud sont encore valides dans le reste de l'arborescence et permettent de couper d'autres branches à la manière des backtracking intelligents en PPC.

1.3.4 Backtracking intelligent, principe de résolution et programmation linéaire

Sans compter ces formes de décomposition de Benders généralisée, il existe peu d'applications du backtracking intelligent et des techniques d'apprentissage en PLNE. Ceci provient certainement du fait que, contrairement à la PPC où on identifie facilement la cause d'un échec en considérant l'instanciation partielle des variables violant une contrainte, l'évaluation globale du problème qui est faite en PLNE à un noeud de l'arbre de recherche ne permet pas de déterminer quel est le sous-ensemble des décisions prises rendant ce noeud irréalisable ou sub-optimal.

En ordonnancement, on trouve une forme d'apprentissage avec les *règles de dominance* : on enregistre les ordonnancements partiels construits par branchement, de façon à couper l'arbre dès qu'on accède à un ordonnancement que l'on sait dominé par à un autre déjà visité et enregistré. Cette technique, bien qu'ayant démontré son efficacité, n'est généralement pas optimisée : d'une part, en terme d'espace mémoire car les nogoods (les ordonnancements partiels) sont généralement accumulés et parfois supprimés quand ils sont eux-même dominés, et d'autre part, ils ne servent qu'à couper l'arbre et ne sont pas utilisés pour guider la recherche comme dans un backtracking dynamique par exemple. Guéret, Jussien et Prins ont proposé une amélioration d'une PSE pour le problème d'open-shop par un backtracking intelligent [Guéret 2000]. Les décisions responsables des modifications des fenêtres de temps des activités sont mémorisés de façon à effectuer un backtrack, quand la fenêtre de temps d'une activité devient vide, directement sur la dernière décision responsable. Bien que, dans ces deux cas, une relaxation linéaire puisse être résolue à chaque noeud de l'arborescence, le raisonnement d'apprentissage n'est effectué ici que sur le résultat d'un filtrage PPC.

Les stratégies de recherche pour un PLNE général ne font pas intervenir l'apprentissage, pourtant il existe une analogie fondamentale entre le *principe de résolution* en logique propositionnelle et l'algorithme de coupes de Chvatal-Gomory, qui offre autant de relation entre le backtracking intelligent en PPC (qui utilise le principe de résolution pour la génération et la mise à jour des nogoods) et le branch and cut en PLNE.

Cette analogie a récemment été étudiée en intelligence artificielle dans le cadre du problème SAT de satisfiabilité d'un ensemble de clauses sous forme normale conjonctive (CNF) : existe-t'il $(x_1, \ldots, x_n) \in \{0,1\}^n$ satisfaisant chacune des clauses $c_1, \ldots, c_m$, une clause étant une disjonction de littéraux, et un littéral l_i correspondant à une variable x_i ou à sa négation $\bar{x}_i$? Le principe de résolution est une technique de preuve, basée sur l'inférence, *complète*, c.-à-d. elle résoud SAT en un nombre fini d'étapes. À chaque étape, une nouvelle clause est inférée à partir d'une ou deux clauses du problème puis ajoutée au problème. L'inférence s'effectue par résolvante $(l \vee \bigvee_i l_i) \wedge (\bar{l} \vee \bigvee_j l_j) \vdash (\bigvee_i l_i \vee \bigvee_j l_j)$ ou par factorisation $(l \vee l \vee \bigvee_i l_i) \vdash (l \vee \bigvee_i l_i)$.

Ce principe est un cas particulier de l'algorithme de Chvatal-Gomory quand une clause $(\bigvee_i x_i \vee \bigvee_j \bar{x}_j)$ est traduite par l'inégalité $\sum_i x_i + \sum_j (1 - x_j) \geq 1$. En effet, la résolvante est l'équivalent d'une coupe surrogate (la somme de $x + \sum_i x_i \geq 1$ et $(1-x) + \sum_j x_j \geq 1$ s'écrit $\sum_i x_i + \sum_j x_j \geq 1$) et la factorisation s'obtient aussi par combinaison linéaire ($x + x + \sum_i x_i \geq 1$ et $\sum_i x_i \geq 0$ implique $x + \sum_i x_i \geq \frac{1}{2}$) puis par arrondi ($x + \sum_i x_i \geq \lceil \frac{1}{2} \rceil = 1$). Ainsi, l'existence d'une preuve par

résolution en un nombre polynomial d'étapes implique l'existence d'une preuve polynomiale par cet algorithme de coupes. En revanche, l'inverse n'est pas vrai du fait que l'on peut traiter avec Chvatal-Gomory des inégalités de forme plus générale $\sum a_i x_i \geq b$ et donc inférer des coupes plus fortes. On trouve par exemple dans [Dixon 2000], une preuve de longueur n^2 pour le problème des pigeons (placer n pigeons dans k trous, un pigeon par trou) alors qu'une preuve par résolution de ce même problème est nécessairement exponentielle.

Dans ce même article, et à la suite de [Hooker 2000b], Dixon et Ginsberg proposent ainsi un backtracking intelligent où les nogoods ne sont pas générés par résolution sous la forme de clauses, mais sous la forme d'inégalités surrogate. Dans leur approche, les clauses sont linéarisées comme indiqué ci-dessus et un branchement standard (instanciation d'une variable à 0 ou 1) est effectué. Un simple filtrage par *propagation unitaire* vérifie la satisfaction des contraintes : avant d'instancier une variable x à 0 ou 1, on regarde s'il existe une contrainte linéaire c_0 (resp. c_1), dont toutes les variables sauf x sont instanciées, et qui est violée par $x = 0$ (resp. $x = 1$). Si c_0 et c_1 sont identifiées, on génère une contrainte nogood comme la combinaison linéaire de c_0 et c_1 supprimant x. Par exemple, si y, z et w sont actuellement instanciées à 0, et comme $x + y + z \geq 1$ est violée par $x = 0$ et $2\bar{x} + w \geq 2$ est violée par $x = 1$, le nogood généré est alors $2y + 2z + w \geq 2$ qui interdit dans la suite de la recherche, pour (y, z, w), l'instanciation partielle $(0, 0, 0)$ mais aussi $(0, 0, 1)$. La restriction de l'apprentissage est aussi adaptée à ce type de contraintes pour limiter la mémorisation aux seuls nogoods pertinents à un état donné de la recherche. Pour une valeur de pertinence r, on supprime les nogoods $\sum_i a_i x_i \geq b$ présentant le moins de chances d'être violés, c.-à-d. (si $a_i \geq 0$) tels que $\sum_{i \in V} a_i x_i + \sum_{i \notin V} a_i - b \geq r$ où V est l'ensemble des variables instanciées au noeud courant.

Ici encore, puisqu'il s'agit de résoudre un problème de satisfiabilité, la méthode ne fait évidemment pas appel au critère d'optimalité qui est la qualité principale des techniques PLNE. Seule est utilisée la plus grande expressivité des inégalités par rapport aux clauses CNF puisque les nogoods sont identifiés comme en PPC classique.

On verra au chapitre 5 comment la procédure *resolution search* de Chvatal développe l'idée d'apprentissage et de backtracking intelligent pour les PLVB, en mémorisant les parties de l'espace de recherche ne contenant pas de solution réalisable, ni non plus de solution optimale, et surtout en gérant ces nogoods suivant le principe de résolution de manière à limiter la mémorisation et à guider la recherche.

Chapitre 2

Le problème d'ordonnancement de projet à contraintes de ressources

---

Ce chapitre présente en détail le problème d'ordonnancement de projet à contraintes de ressources*, aussi appelé* problème d'ordonnancement de projet à moyens limités*, ou encore* RCPSP *pour «*resource-constrained project scheduling problem*». Cette dénomination couvre, en réalité, une variété de problèmes telle, que plusieurs schémas de classification ont été développés [Herroelen 1999, Brucker 1999] mais pas encore unifiés. Puisque nous traitons, dans cette thèse, de la forme classique du problème, c'est celle-ci que l'on désigne ici par RCPSP. Dans une première partie (2.1), nous décrivons formellement ce problème et présentons brièvement ses applications, cas particuliers et généralisations. Puis nous abordons les recherches qui ont été menées sur le sujet en nous penchant plus particulièrement sur les différentes règles d'ajustement qui s'appliquent au RCPSP dans le cadre de la programmation par contraintes (2.2), les principaux programmes linéaires en nombres entiers qui modélisent ce problème (2.3) et enfin, les schémas de branchement et les bornes inférieures de la littérature (2.4).*

2.1 Description du RCPSP

2.1.1 Définition

La réalisation d'un *projet* consiste en l'exécution, par des *ressources*, d'un ensemble d'*activités* (ou *tâches*) de *durées* données et liées entre elles par des *contraintes de précédence*. Dans sa forme classique, une instance du RCPSP est la donnée d'un ensemble $\mathcal{A}$ de n activités d'un projet et d'un ensemble $\mathcal{R}$ de m ressources. Les activités du projet sont non-interruptibles (ou *non-préemptives*). Ainsi, une activité i termine son exécution à l'instant $S_i + p_i$, où S_i est la date de début d'exécution de i et p_i, sa durée. Les activités sont liées par des contraintes de précédence *simples*, de la forme $i \rightarrow j$, interdisant de débuter l'exécution de la seconde activité j avant la fin de la première, i. On modélise couramment le projet par un graphe valué, orienté et sans circuit $G = (V, E, p)$, le *graphe potentiel-tâches*, formé du graphe associé à la relation formée par les contraintes de précédence, auquel sont ajoutées deux activités fictives de durées nulles, représentant respectivement, le début et la fin du projet : 0 pour le début du projet dont on commence l'exécution à l'instant de référence $S_0 = 0$, et $n + 1$ succédant à toutes les activités du projet.

Enfin, les m ressources sont renouvelables, cumulatives, (elles peuvent exécuter plusieurs activités simultanément), et disponibles, à tout instant, en quantité limitée. On appelle *consommation*, et on note r_{ik}, la quantité de la ressource k allouée à l'activité i durant tout le temps de son exécution et, *capacité*, R_k, la disponibilité totale constante de la ressource k. Les ressources étant renouvelables, la quantité r_{ik} utilisé par une activité i est à nouveau disponible au terme de l'exécution de i. On suppose que toutes ces données, durées, consommations et capacités, sont des entiers positifs. Ainsi, les dates de début des activités peuvent aussi être supposées entières et on identifiera par la suite, l'instant t avec l'intervalle de temps $[t, t + 1[$.

L'objectif du RCPSP est de réaliser l'ensemble du projet en un temps minimal, autrement dit, de déterminer un ordonnancement $S = (S_0, S_1, \ldots, S_{n+1})$ de durée S_{n+1} minimale, à la fois, en respectant les contraintes de précédence, c'est à dire les inégalités de potentiels $S_j \geq S_i + p_i$, pour tout couple d'activité $(i, j) \in E$, et en résolvant à tout instant t, les conflits dans l'utilisation de chaque ressource k : $\sum_{i \in \mathcal{A}_t} r_{ik} \leq R_k$, où $\mathcal{A}_t$ est l'ensemble des activités i en cours d'exécution au temps t ($S_i \leq t < S_i + p_i$). Un tel ordonnancement est dit *optimal*, tandis qu'un vecteur S vérifiant les contraintes de précédence et les contraintes de ressources sans nécessairement minimiser S_{n+1} est appelé *ordonnancement réalisable*.

À partir des données du problème, d'autres valeurs, qui seront utilisées tout au long de ce document, peuvent être initialisées à la toute première étape du processus d'ordonnancement :

- $T \in \mathbb{N}$, l'*horizon* est une évaluation par excès de la durée minimale d'un ordonnancement : $S_{n+1} \leq T$. Elle peut être initialisée p. ex. avec la somme des durées des activités ou par toute heuristique ;
- $ES_i \in \mathbb{N}$ (resp. $EF_i \in \mathbb{N}$), la date de début (resp. de fin) au plus tôt de l'activité i pour « *earliest starting time* » (resp. « *earliest finish time* »), avant laquelle i ne peut commencer (resp. finir) : $ES_i \leq S_i$ et $EF_i \leq S_i + p_i$. Elle est initialisée à 0 (resp. p_i) pour toute activité i ;
- $LS_i \in \mathbb{N}$ (resp. $LF_i \in \mathbb{N}$), la date de début (resp. de fin) au plus tard de l'activité i pour « *latest starting time* » (resp. « *latest finish time* »), après laquelle i ne peut commencer (resp. finir) : $S_i \leq LS_i$ et $S_i + p_i \leq LF_i$. Elle est initialisée à $T - p_i$ (resp. T) pour toute activité i ;
- $b_{ij} \in \mathbb{Z}$, la *distance* minimale entre la date de début de l'activité i et la date de début de l'activité j : $S_j - S_i \geq b_{ij}$. Puisque $S_0 = 0$, toutes les notions précédentes ont leurs équivalent dans la matrice des distances $B = (b_{ij})_{(i,j) \in V \times V}$: $T = b_{0(n+1)}$, $ES_i = b_{0,i}$, $EF_i = b_{0,i} + p_i$,

$LS_i = -b_{i,0}$, $LF_i = -b_{i,0} + p_i$.

À d'autres moments, T sera aussi un entier positif quelconque donné en entrée du problème dans le cas où l'on souhaite résoudre une instance décisionnelle du RCPSP (« existe-t'il un ordonnancement réalisable de durée totale inférieure ou égale à T ? »).

2.1.2 Complexité

Le RCPSP appartient à la classe des problèmes combinatoires les plus difficiles. Garey et Johnson [Garey 1979] ont montré par une réduction au problème de 3-partition que le problème d'ordonnancement à contraintes de ressources, et sans contraintes de précédence, avec une unique ressource, est NP-difficile au sens fort. On peut se faire une idée plus évidente de la réelle difficulté de ce problème par l'observation suivante reprise par [Schäffter 1997, Uetz 2001], sachant qu'il est généralement admis que, comme $P \neq NP$, $NP \not\subseteq ZPP$, la classe *zero-probability polynomial* des problèmes de décision résolubles par un algorithme de type *Las Vegas* :

Théorème 2 *À moins que $NP \subseteq ZPP$, pour toute constante $\epsilon > 0$, il n'est pas possible d'approcher, en temps polynomial, par un facteur de $n^{1-\epsilon}$ la valeur minimale d'un ordonnancement à contraintes de ressources.*

Ce théorème est un corrolaire du théorème de Feige et Kilian [Feige 1998] sur l'impossible approximation en temps polynomial du nombre chromatique dans le problème de coloration de graphes non-orientés, problème polynomialement équivalent au problème d'ordonnancement à contraintes de ressources.

2.1.3 Exemple

Un exemple simple d'une instance à 7 activités réelles et 3 ressources est donné par Emmanuel Néron dans sa thèse [Néron 1999] et peut se représenter sur la figure 2.1, par le graphe potentiel-tâches, avec à droite, une solution optimale modélisée par les diagrammes de Gantt d'occupation des ressources.

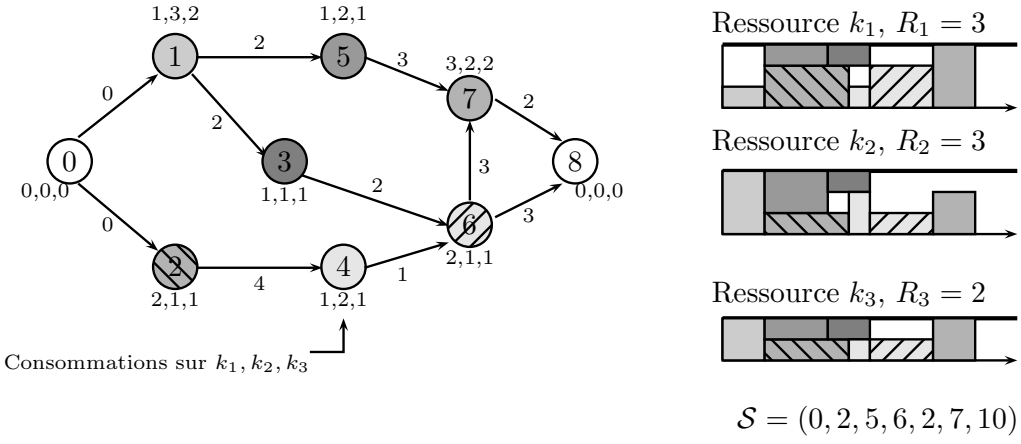

FIG. 2.1 – *Exemple d'instance du RCPSP.*

2.1.4 Applications et cas particuliers

« En dépit » de sa nature fortement combinatoire, le RCPSP a fait l'objet de nombreuses recherches. Principalement, car il possède un large champ d'applications industrielles, d'un point de vue technologique comme :

- l'ordonnancement des processus en informatique distribuée, où il est question d'exécuter des processus (les activités) sur des machines parallèles ou multi-processeurs (les ressources) en gérant les exclusions mutuelles et la synchronisation ;
- les problèmes de découpes [Dyckhoff 1990] : dans le textile ou la métallurgie par exemple, il s'agit de débiter de larges rouleaux de taille standard R, en rouleaux de moindres largeurs de façon à minimiser le nombre de rouleaux initiaux. Ce problème se modélise en un RCPSP avec une seule ressource de capacité R et où, à chaque rouleau à découper, correspond une activité de durée unitaire et de consommation égale à la largeur du rouleau ;
- le processus de production dans l'industrie chimique [Heipcke 1999] : le processus qui consiste à produire des quantités données de produits chimiques se modélise, après simplification, en un LCSP (*labor-constrained scheduling problem*) où, à chaque produit, correspond une suite d'activités identiques dont les consommations, c.-à-d. le nombre, évidemment limité, de travailleurs assignés à leur production, varient en fonction du temps ;
- les problèmes d'ateliers courant [Esquirol 1999] : des travaux, décomposés en séquences (déterminées pour le *job-shop* et le *flow-shop* ou pas pour l'*open-shop*) d'opérations, doivent être exécutées par des machines *disjonctives* n'exécutant qu'une opération à la fois, chaque opération ayant sa machine dédiée. Tous les problèmes d'ateliers sont des cas particuliers du RCPSP, où les activités sont les opérations et où une ressource, unitaire, correspond soit à un travail, soit à une machine. La consommation d'une activité-opération est de 1 sur la ressource-machine dédiée correspondante et sur la ressource-travail contenant l'opération, elle est de 0 sur toutes les autres ressources ;
- le *flow-shop hybride* [Néron 1999] : à mi-chemin entre le *flow-shop* et le RCPSP, il modélise lui-même de nombreux cas industriels. Il étend le flow-shop (problème d'atelier à cheminement unique où tous les travaux J_i possèdent le même nombre K d'opérations traitées dans le même ordre $O_{i,1}, O_{i,2}, \ldots, O_{i,K}$) en permettant, à chaque étage j, que les opérations $O_{i,j}$ soient chacunes exécutées sur l'une des m_j machines identiques disponibles. C'est évidemment un RCPSP où, à chaque étage j, correspond une ressource de capacité m_j, et où les activités sont les opérations $O_{i,j}$ de consommation unitaire sur la j-ème ressource, nulle sur les autres, et liées par les contraintes de précédence en chaînes $O_{i,j} \rightarrow O_{i,j+1}$. Une définition équivalente existe pour le job-shop, on parle alors de *job-shop généralisé*.

Le RCPSP a aussi des applications dans le domaine organisationnel comme :

- les problèmes d'emploi du temps [Brucker 2001] : il s'agit d'organiser dans le temps un certain nombre d'activités impliquant différents groupes de personnes et nécessitant du matériel ou de l'espace. Dans un exemple simple d'emploi du temps à l'université, où il est question de répartir les cours par enseignants et par salles, le problème se modélise en un RCPSP, pratiquement de la même manière qu'un problème d'atelier, mais où les ressources (classes et enseignants) ont des capacités dépendantes du temps, les enseignants n'étant disponibles qu'à certaines périodes.
- la planification du personnel, dans l'aviation par exemple.

2.1.5 Variantes et extensions du modèle classique

On voit, p. ex. pour les problèmes d'emploi du temps ou de production chimique, que la formulation classique du RCPSP de la section 2.1.1 ne décrit pas assez précisément certaines situations de la vie réelle. Ainsi, il s'est avéré nécessaire de travailler sur des variantes de ce problème, en acceptant la préemption des activités ou l'ordonnancement simultané de plusieurs projets, en faisant varier les données, en généralisant les contraintes ou encore en optimisant sur d'autres critères.

Les données peuvent être variables en fonction du mode ou du temps. Dans le cas *multi-mode*, différentes alternatives sont envisagées, à chacune correspondant des durées et consommations fixes. Il peut s'agir d'alternatives temps/ressources (1 machine pendant 8 heures ou 4 machines pendant 2 heures) ou ressources/ressources (2 unités sur une ressource ou 1 unité sur 2 ressources). Capacités, consommations et durées peuvent aussi varier en fonction du temps. Si les capacités varient, on se ramène à considérer le cas des ressources *partiellement renouvelables*, c.-à-d. non-renouvelables sur des intervalles de temps donnés, ce qui généralise les ressources renouvelables et/ou non-renouvelables. Si les consommations varient en fonction du temps, le problème s'apparente alors au LSCP.

Le RCPSP avec contraintes de précédence généralisées, $S_j - S_i \geq d_{ij}\ \forall (i,j) \in V \times V$ où d_{ij} est un entier positif ou négatif, autorise la modélisation de délais entre les activités, s'il est nécessaire par exemple de retarder l'exécution d'une tâche j de d unités de temps après la fin de la tâche i ($S_j - S_i \geq p_i + d$) ou, au contraire, si l'exécution de deux tâches doit être entièrement réalisée dans une période de longueur d ($S_j - S_i \geq p_i - d$ et $S_i - S_j \geq p_j - d$). Elles permettent aussi de modéliser, pour toute activité i, la *date de disponibilité* r_i après laquelle i peut commencer et la *date échue* d_i avant laquelle i doit être achevée, en posant $d_{0i} = r_i$ et $d_{i0} = p_i - d_i$. On peut évoquer, au passage, une autre relaxation du RCPSP, le CuSP pour *cumulative scheduling problem* [Baptiste 1998], avec une unique ressource et où les contraintes de précédence sont remplacées par les seuls intervalles $[r_i, d_i]$. La précédence généralisée intègre encore les contraintes de *parallélisme* ($S_j - S_i \geq 1 - p_j \wedge S_i - S_j \geq 1 - p_i$) et, si liées par le « ou » logique $\vee$, les contraintes *disjonctives* ($S_j - S_i \geq p_i \vee S_i - S_j \geq p_j$). Respectivement, ces deux types de contraintes forcent ou interdisent que deux activités soient en cours d'exécution à un même instant.

La fonction objectif, c.-à-d. le critère d'optimisation, est un attribut discriminant du modèle. Outre la minimisation de la durée totale du projet, d'autres critères relatifs au temps peuvent être considérés, tels que la minimisation de la somme pondérée des dates de fin $S_i + p_i$, la minimisation de la somme pondérée, du maximum ou de la moyenne des retards $T_i = \max(0, S_i + p_i - EF_i)$ ou du nombre d'actvités en retard, etc. D'autres critères, encore, sont basés sur les coûts (production, stockage,...) ou liés à la charge ou au nombre de ressources utilisées.

Enfin, le RCPSP déterministe a sa problématique dans le domaine stochastique (où les données, telles que la durée des activités, sont connues de manière probabiliste) et plus généralement en flexibilité et robustesse (réagir aux aléas venant perturber le problème initial). Les revues de la littérature [Demeulemeester 2002, Brucker 1999, Kolisch 2001] offrent plus de précision sur la large classe des problèmes d'ordonnancement de projet à contraintes de ressources et sur les recherches associées.

Par la suite, comme il a été dit au début de ce chapitre, nous ne nous intéresserons qu'au RCPSP classique. Cependant, l'usage intensif de la notion de *matrice de distances* pour la propagation des contraintes (cf. section 2.2), fait que les méthodes présentées dans ce mémoire s'appliquent également, et pratiquement toutes à l'identique, au RCPSP avec contraintes de précédence généralisées.

2.2 Règles d'ajustement

Cette section est maintenant consacrée aux principaux tests de consistance et techniques de filtrage (ou règles d'ajustement), développés, ou souvent adaptés du cas disjonctif, pour la résolution du RCPSP dans une approche de satisfaction de contraintes. Cet état de l'art n'est pas exhaustif. On trouvera d'autres règles ou des références à d'autres algorithmes de mise en oeuvre dans [Baptiste 2001, Esquirol 2001, Dorndorf 1999].

Les techniques proposées ont pour but de résoudre partiellement une instance décisionnelle du RCPSP, en supprimant des dates de début des activités ou des séquencements entre activités, qui sont inadmissibles dans tout ordonnancement réalisable de durée totale inférieure ou égale à T. La détection d'une inconsistance globale (p. ex. une activité sans date possible de début ou deux activités sans séquencement), prouve que l'ensemble $\mathcal{S}_T$ de ces ordonnancements est vide. Autrement, le problème est réduit et on possède alors une caractérisation plus fine de l'espace $\mathcal{S}_T$ de recherche des solutions, ce qui peut être exploité pour la recherche d'une solution réalisable ou optimale ou encore pour le calcul d'une borne inférieure.

Nous présentons tout d'abord (2.2.1) la formulation du RCPSP en un CSP et comment les ajustements sont propagés, puis, de (2.2.2) à (2.2.5), nous décrivons les règles d'ajustement locales, qui peuvent être implémentées les unes indépendemment des autres. Enfin (2.2.6), nous présentons la technique globale du *shaving*. Toutes, techniques locales et globale, permettent de déduire de nouvelles contraintes temporelles ou de nouveaux conflits entre paires d'activités, qui, à leur tour, infèrent d'autres contraintes temporelles et donc un ajustement des domaines.

2.2.1 Contraintes temporelles et propagation

Le RCPSP (non-préemptif) se formule comme une extension d'un *CSP temporel simple* (TCSP, voir section 1.1) auquel on ajoute, pour chaque ressource k, une contrainte globale appelée *contrainte cumulative* et qui modélise la relation $\sum_{i \in \mathcal{A}_t} r_{ik} \leq R_k$, pour tout instant t. Les variables de décision sont donc les dates de début S_i des activités et leurs domaines sont, pour des raisons évidentes de complexité, approximés par des intervalles simples $[ES_i, LS_i]$, sans « trous ». De fait, le maintien de la consistance n'est effectué qu'aux bornes des domaines (*arc-B-consistance*) et l'inconsistance globale peut être détectée à partir du moment où la borne inférieure d'un intervalle devient plus grande que sa borne supérieure. Les domaines des variables sont initialisés à $[0, T]$.

Dans ce modèle, les contraintes de précédence s'écrivent sous la forme de *contraintes temporelles simples*, $S_j - S_i \in [p_i, +\infty[$, $\forall (i,j) \in E$, de même que les *contraintes de domaine*, $S_i - S_0 \in [ES_i, LS_i]$. Ainsi, un simple algorithme (de type `bellman`) en $O(|E|)$ permet d'assurer l'arc-B-consistance de l'ensemble des contraintes de domaine et des contraintes de précédence, par le calcul, dans le graphe de ces contraintes, de plus longs chemins : de 0 à i pour ajuster ES_i et de i à $n+1$ pour ajuster LS_i.

Il est possible aussi de maintenir un plus grand degré de consistance (consistance de chemin aux bornes ou 3-consistance) en considèrant les contraintes temporelles généralisées : $S_j - S_i \in [-b_{ji}, b_{ij}]$, $\forall (i,j) \in \mathcal{A}$, $i \leq j$. En effet, l'algorithme 2 de `floyd-warshall` effectue en $O(n^3)$ la propagation *complète* de toutes les contraintes temporelles généralisées, en considérant la propriété de transitivité de $b_{ij} \geq b_{il} + b_{lj}$.

Pour maintenir cette consistance, on utilise comme structure de donnée, la *matrice des distances* $B = (b_{ij}) \in \mathbb{Z}^{V \times V}$. Cette structure modélise les bornes des domaines des variables ($b_{0i} = ES_i$, $b_{i0} = -LS_i$) et l'horizon ($b_{(n+1)0} = -T$). L'ajustement des bornes des domaines des contraintes

Algorithme 2 – Algorithme de `floyd-warshall` $O(n^3)$

pour l **de** 0 **à** $n+1$ **faire**
  pour i **de** 0 **à** $n+1$ **faire**
    pour j **de** 0 **à** $n+1$ **faire**
      $b_{ij} = \max\{b_{ij}, b_{il} + b_{lj}\}$
      si $b_{ij} < -b_{ji}$ **alors**
        STOP (*inconsistance*)
      fin si
    fin pour
  fin pour
fin pour

temporelles se traduit par une augmentation des valeurs b_{ij}. De plus, à tout moment, la valeur $b_{0(n+1)}$ est une borne inférieure de la valeur optimale du RCPSP.

La notion de distance permet de décrire plus explicitement le séquencement relatif de deux activités, à savoir : i s'exécute entièrement avant j ($i \rightarrow j$), j avant i ($j \rightarrow i$) ou bien i et j s'exécutent simultanément à au moins un instant ($i \parallel j$). Ainsi, dans tout ordonnancement réalisable de durée inférieure à T, on a :

- $i \rightarrow j$ (*précédence*) si $b_{ij} \geq p_i$;
- $i \nrightarrow j$ (« *non-précédence* » ou précédence interdite) si $b_{ji} \geq 1 - p_i$;
- $i \parallel j$ (*parallèlisme*) si $b_{ji} \geq 1 - p_j$ et $b_{ij} \geq 1 - p_i$.

Par exemple, si une nouvelle précédence $i \rightarrow j$ est ajoutée à l'ensemble des contraintes du problème alors l'ajustement des fenêtres de temps s'établit en posant $ES_j = \max(ES_j, ES_i + p_i)$ et $LS_i = \min(LS_i, LS_j - p_i)$. Cet ajustement est dominé par l'ajustement des distances $b_{ij} = \max(b_{ij}, p_i)$ puisque, après propagation au moyen de l'algorithme 2, on a $ES_j = b_{0j} \geq b_{0i} + b_{ij} \geq ES_i + p_i$ et $LS_i = -b_{i0} \leq -b_{ij} - b_{j0} \leq LS_j - p_j$. D'ailleurs la propagation d'une unique contrainte de séquencement ($i \rightarrow j$, $i \nrightarrow j$ ou $i \parallel j$) ou, plus généralement, d'une unique contrainte temporelle $S_l - S_h \in [-d_{lh}, d_{hl}]$, avec $b_{hl} \leq d_{hl}$ et $b_{lh} \leq d_{lh}$ peut être propagée à l'ensemble de la matrice en seulement $O(n^2)$ par l'algorithme 3 de `floy-warshall modifié`. Cette notion de distance est

Algorithme 3 – Algorithme de `floyd-warshall modifié` $O(n^2)$

$b_{hl} = d_{hl}, b_{lh} = d_{lh}$
pour i **de** 0 **à** $n+1$ **faire**
  pour j **de** 0 **à** $n+1$ **faire**
    $b_{ij} = \max\{b_{ij}, b_{il} + b_{lh} + b_{hj}, b_{ih} + b_{hl} + b_{lj}, \}$
    si $b_{ij} < -b_{ji}$ **alors**
      STOP (*inconsistance*)
    fin si
  fin pour
fin pour

nécessaire aussi si l'on souhaite modéliser le RCPSP avec contraintes de précédence généralisées. La majorité des algorithmes de programmation par contraintes en ordonnancement se base, cependant, sur les fenêtres de temps des activités. Ainsi de nombreux tests de consistance cherchent à déduire des ajustements sur les ES_i et LS_i.

Contrairement aux contraintes de précédence, il est extrêmement difficile d'assurer la consistance de l'ensemble des contraintes cumulatives de ressources. Les règles d'ajustement consistent donc à générer de nouvelles contraintes temporelles, en prenant en considération un sous-ensemble

de contraintes de ressources du type $\sum_{i \in C} r_{ik} \leq R_k$. Elles cherchent ainsi à caractériser les *ensembles critiques* C qui violent ces contraintes pour déduire de nouveaux séquencements obligatoires (ou des dates de débuts impossibles) par l'observation que les activités d'un ensemble critique C ne peuvent être toutes exécutées simultanément.

En particulier, certaines techniques de filtrage permettent de détecter des paires d'activités en *disjonction* $(i - j)$, c.-à-d. qui doivent être exécutées l'une à la suite de l'autre. Il s'avère utile donc de mémoriser l'ensemble D des paires d'activités en disjonction $i - j$ dont le séquencement relatif n'est pas encore connu ($i \rightarrow j$ ou $j \rightarrow i$). D est initialisé avec l'ensemble des paires d'activités $\{i, j\}$ qui entrent en conflit sur l'occupation d'au moins une ressource :

$$(\exists k \in R,\ r_{ik} + r_{jk} > R_k) \quad \Rightarrow \quad \{i, j\} \in D.$$

Inversement, d'autres techniques utilisent les disjonctions pour déduire des précédences. C'est le cas des règles présentées à la section suivante.

2.2.2 Ensembles disjonctifs

Les techniques de filtrage présentées ici ont été employées avec succès à la résolution de problèmes disjonctifs (essentiellement le job-shop) mais elles s'appliquent également à tout *ensemble disjonctif maximal* (EDM) dans un problème cumulatif tel que le RCPSP. Un ensemble disjonctif maximal C est un sous-ensemble d'activités en disjonction deux à deux, maximal pour l'inclusion. Si on considère le graphe des paires d'activités en disjonction ou en précédence, les ensembles disjonctifs correspondent aux cliques du graphe, et certaines techniques de déduction associées, *sélection immédiate* [Carlier 1989] ou *edge-finding* [Applegate 1991], consistent à déterminer l'orientation des arêtes du graphe, autrement dit à déduire de nouvelles précédences. D'autres techniques, *not-first/not-last*, concluent qu'une activité $i \in C$ ne doit être placée avant (resp. après) toutes les activités d'un ensemble $\Omega \subseteq C$: $i \nrightarrow \Omega$ (resp. $\Omega \nrightarrow i$).

Une recherche exhaustive de tous les EDM d'une instance du RCPSP, et surtout l'application des règles d'ajustement à chacun d'entre eux, se révéle trop coûteuse. On se contente en général d'un sous-ensemble d'EDM calculés, soit par une méthode exacte comme la recherche d'une clique de capacité (la somme des durées des activités) maximale [Baptiste 2004] ; soit de manière heuristique : Baptiste et Le Pape [Baptiste 2000] privilégient la capacité de la clique : pour chaque ressource, un EDM est généré en sélectionnant l'ensemble C_k des activités requérant plus de la moitié de la capacité de la ressource, puis complété progressivement par des activités classées par ordre de durée décroissante. Expérimentalement, il semble plus avantageux de rechercher de manière exacte, par programmation linéaire, une clique de capacité maximale qui étend C_k [Baptiste 2004]. Brucker et al. [Brucker 1998] proposent une autre heuristique en $O(n^2)$ en classant les activités dans l'ordre inverse du nombre de disjonctions et de précédences où elles apparaissent. Pour chaque activité i non encore présente dans un EDM, un nouvel EDM est généré en incluant d'abord i puis suivant l'ordre, les autres activités incompatibles.

Dans cette section, C dénote un EDM quelconque, i une activité de C et Ω tout sous-ensemble de $C \setminus \{i\}$. Ω est assimilé à une seule activité de durée $p_\Omega = \sum_{j \in \Omega} p_j$ devant être exécutée dans l'intervalle de temps $[ES_\Omega, LF_\Omega]$ où $ES_\Omega = \min_{j \in \Omega} ES_j$ et $LF_\Omega = \max_{j \in \Omega} LF_j$.

Paires disjonctives

En considérant simplement deux activités i et j de $\mathcal{A}$, on détecte si i ne peut précéder j par l'une ou l'autre des règles suivantes (la seconde domine la première puisque, après propagation,

$b_{ji} \geq b_{j0} + b_{0i} = -LS_j + ES_i$) :

$$(ES_i + p_i > LS_j \quad \Rightarrow \quad i \nrightarrow j) \quad \text{ou} \quad (b_{ji} > -p_i \quad \Rightarrow \quad i \nrightarrow j) \tag{2.1}$$

Appliquée à une paire d'activités en disjonction $i - j \in D$, cette règle infère le séquencement obligatoire $j \rightarrow i$ et donc, dans la matrice des distances, l'ajustement $b_{ji} = p_i$.

Not-first/not-last

La règle précédente se généralise si on considère une activité $i \in C$ vis-à-vis, non plus d'une activité, mais d'un ensemble $\Omega \subseteq C$ d'activités. Dans l'ordonnancement partiel consistant à caler l'exécution de i au plus tôt dans sa fenêtre de temps ($S_i = ES_i$) et à sa suite toutes les activités de Ω, si une activité de Ω se retrouve alors ordonnancée au-delà de sa fenêtre, alors on déduit que i ne peut être placée avant toutes les activités de Ω ($i \nrightarrow \Omega$) :

$$ES_i + p_i + p_\Omega > LF_\Omega \quad \Rightarrow \quad i \nrightarrow \Omega. \tag{2.2}$$

C'est la règle *not-first* [Pinson 1988, Carlier 1990], et elle permet de mettre à jour $ES_i = \max(ES_i, \min_{j\in\Omega} ES_j + p_j)$. Symétriquement, la règle *not-last* détecte si i ne peut être placée après Ω et, dans ce cas, entraîne la mise à jour de $LF_i = \min(LF_i, \max_{j\in\Omega} LS_j)$:

$$LF_i - p_i - p_\Omega < ES_\Omega \quad \Rightarrow \quad \Omega \nrightarrow i. \tag{2.3}$$

La détection et la mise à jour des domaines de toutes les activités $i \in C$ peuvent être menées en $O(|C|^2)$ (voir p. ex. [Baptiste 1996]).

FIG. 2.2 – *Règles not-first/not-last*

Edge-finding

Les déductions du edge-finding [Pinson 1988, Carlier 1990] sont autrement plus fortes, puisqu'il s'agit de nouvelles précédences entre une activité $i \in C$ et toutes les activités d'un ensemble $\Omega \subseteq C \setminus \{i\}$. Les deux règles du edge-finding détectent si i ne peut être placée ni après (resp. avant), ni entre les activités de Ω :

$$ES_\Omega + p_\Omega + p_i > \max(LF_\Omega, LF_i) \quad \Rightarrow \quad i \rightarrow \Omega, \tag{2.4}$$

$$LF_\Omega - p_\Omega - p_i < \min(ES_\Omega, ES_i) \quad \Rightarrow \quad \Omega \rightarrow i. \tag{2.5}$$

Comme les activités de Ω s'exécutent l'une après l'autre, la condition $i \rightarrow \Omega'$, $\forall \Omega' \subseteq \Omega$ inférée par (2.4) p. ex. , s'avère plus forte que $i \rightarrow j$, $\forall j \in \Omega$ et autorise un meilleur filtrage du domaine de S_i :

$$i \rightarrow \Omega \quad \Rightarrow \quad LF_i = \min(LF_i, \min\{LF_{\Omega'} - p_{\Omega'} \mid \emptyset \neq \Omega' \subseteq \Omega\}) \tag{2.6}$$

$$\Omega \rightarrow i \quad \Rightarrow \quad ES_i = \max(ES_i, \max\{ES_{\Omega'} + p_{\Omega'} \mid \emptyset \neq \Omega' \subseteq \Omega\}). \tag{2.7}$$

Détections et ajustements peuvent être réalisés en $O(|C|^2)$ (voir p. ex. [Carlier 1990]) ou, au moyen de structures de données plus complexes en $O(|C|\log|C|)$ [Carlier 1994].

Borne inférieure basée sur les EDM

Nous proposons un autre type de règle basée sur les EDM. Celle-ci permet uniquement de filtrer le domaine de la dernière activité fictive. En ce sens, elle ne se propage pas mais permet, par exemple, dans une approche destructive (comme aussi dans le shaving) de détecter l'inconsistance d'une valeur maximale d'ordonnancement T donnée. Nous avons implémenté en $O(|C|^2)$ la règle suivante :

$$ES_{n+1} \geq \max\{\min_{i\in\Omega} ES_i + p_\Omega + \min_{i\in\Omega} q_i \mid \Omega \subseteq C\}, \tag{2.8}$$

où $q_i = ES_{n+1} - LF_i$ correspond à la durée minimale séparant la fin de l'activité i de la fin du projet.

Cette règle indique simplement que toutes les activités d'une clique doivent s'exécuter séquentiellement et donc que la durée d'ordonnancement est au moins égale au membre de droite de 2.8.

2.2.3 Contraintes cumulatives

Les règles présentées à la section précédente se généralisent à tout ensemble d'activités (non plus exclusivement disjonctif) pour la résolution de problèmes cumulatifs, si on prend en compte une dimension supplémentaire : la consommation des ressources. On considère ici l'*énergie* $e_{ik} = r_{ik}p_i$ utilisée par une activité i durant son exécution sur une ressource k. Dans sa thèse [Nuijten 1994], Nuijten étend les règles edge-finding et not-first/not-last au cas cumulatif sous cette notion d'énergie.

Schématiquement, l'énergie $e_{\Omega k} = \sum_{j\in\Omega} e_{jk}$ utilisée par un ensemble quelconque $\Omega \subseteq \mathcal{A}$ d'activités pendant leur exécution sur une ressource k est comparée avec la quantité $R_k(f-s)$ de ressouce k disponible dans un intervalle $[s,f]$ strictement inclu dans la fenêtre d'exécution $[ES_\Omega, LS_\Omega]$ de Ω, où $ES_\Omega = \min_{j\in\Omega} ES_j$ et $LF_\Omega = \max_{j\in\Omega} LF_j$ (on note aussi $LS_\Omega = \max_{j\in\Omega} LF_j - p_j$ et $EF_\Omega = \min_{j\in\Omega} ES_j + p_j$). Si l'énergie est strictement supérieure à la quantité disponible, alors il existe au moins une activité j de Ω dont l'exécution doit être reportée partiellement en-dehors de l'intervalle $[s,f]$. Par exemple, la règle not-first au cas cumulatif s'écrit comme suit :

$$\begin{aligned}(\forall i \in \mathcal{A}\setminus\Omega,\ & ES_\Omega \leq ES_i < EF_\Omega \wedge \\ & ES_\Omega + r_{ik}(\min(EF_i, LF_\Omega) - ES_\Omega)\ ) > R_k(LF_\Omega - ES_\Omega) \quad \Rightarrow \quad ES_i \geq EF_\Omega. \end{aligned} \tag{2.9}$$

La règle not-last lui est symétrique. Les conditions d'application du edge-finding cumulatif et les ajustements déduits ont une formulation plus complexe. Nous renvoyons à la lecture de la thèse [Nuijten 1994] de l'auteur et à [Baptiste 2001], pour les énoncés, preuves et algorithmes de ce type de règles d'ajustements.

2.2.4 Raisonnement énergétique

Le raisonnement énergétique [Erschler 1991, Lopez 1991] compare pour toute ressource cumulative k et sur des intervalles temporels $\Delta = [\underline{t}_\Delta, \overline{t}_\Delta]$ choisis, l'énergie $E_k(\Delta) = R_k(\overline{t}_\Delta - \underline{t}_\Delta)$ fournie par la ressource et l'énergie consommée par une activité i. En tenant compte de la fenêtre de temps de l'activité i, on détermine (selon la position i recouvrant le moins l'intervalle Δ, c.-à-d. quand i est ordonnancée au plus tôt ou bien au plus tard), la durée minimale $\underline{p_i}(\Delta)$ d'exécution de i sur

Δ, et donc l'*énergie minimale consommée* $\underline{e_{ik}}(\Delta)$ par i sur la ressource k et sur l'intervalle Δ :

$$\underline{p_i}(\Delta) = \min(\overline{t}_\Delta - \underline{t}_\Delta \ , \ p_i \ , \max(0, ES_i + p_i - \underline{t}_\Delta) \ , \ \max(0, \overline{t}_\Delta - LS_i)), \quad \underline{e_{ik}}(\Delta) = r_{ik}\underline{p_i}(\Delta).$$

Inversement, on peut considérer la durée maximale $\overline{p_i}(\Delta)$ d'exécution de i sur Δ et l'*énergie maximale consommée* $\overline{e_{ik}}(\Delta)$ par i sur la ressource k et sur l'intervalle Δ :

$$\overline{p_i}(\Delta) = \min(\overline{t}_\Delta - \underline{t}_\Delta \ , \ p_i \ , \ \max(0, LS_i + p_i - \underline{t}_\Delta) \ , \ \max(0, \overline{t}_\Delta - ES_i)), \quad \overline{e_{ik}}(\Delta) = r_{ik}\overline{p_i}(\Delta).$$

- ***Inconsistance globale***

Le bilan énergétique, c.-à-d. l'énergie fournie moins l'énergie consommée par l'ensemble des activités, doit être positif sur tout intervalle de temps. On a donc la condition suffisante d'inconsistance globale suivante :

$$\left(\exists \Delta \subseteq [0,T], \ \exists k \in \mathcal{R}, \ \sum_{i \in \mathcal{A}} \underline{e_{ik}}(\Delta) > E_k(\Delta) \right) \quad \Rightarrow \quad \mathcal{S}_T = \emptyset. \tag{2.10}$$

- ***Ajustement des fenêtres de temps***

Au cas où cette condition n'est pas satisfaite, les valeurs énergétiques associées à un intervalle Δ donné sont utiles à l'ajustement des fenêtres de temps des activités i dont l'énergie maximale $\overline{e_{ik}}(\Delta)$ dépasse l'énergie maximale disponible $\overline{E_{ik}}(\Delta) = E_k(\Delta) - \sum_{j \in \mathcal{A} \setminus \{i\}} \underline{e_{jk}}(\Delta)$ compte tenu de la consommation minimale requise par les autres activités. En effet, cette condition fournit une meilleure estimation de la durée maximale d'exécution de i sur l'intervalle Δ, en posant $\overline{p_i}(\Delta) = \lfloor \overline{E_{ik}}(\Delta)/r_{ik} \rfloor$. À son tour, cette valeur permet de déterminer un intervalle Δ_i de dates de début de i interdites :

$$S_i \notin \Delta_i =]\underline{t}_\Delta + \overline{p_i}(\Delta) - p_i, \overline{t}_\Delta - \overline{p_i}(\Delta)[. \tag{2.11}$$

Ainsi, la fenêtre des dates de début possibles de i peut être mise à jour par $[ES_i, LS_i] \setminus \Delta_i$. Comme on ne maintient la consistance qu'aux bornes, seuls les cas où ce nouveau domaine est un intervalle sont traités.

- ***Intervalles d'étude***

Pour un intervalle donné, le test de consistance globale et l'ajustement des bornes s'exécutent clairement en $O(n)$. Il s'agit maintenant d'identifier l'ensemble O des intervalles qu'il est nécessaire et suffisant de considérer pour dériver toutes les inférences de ces règles [Lopez 1991, Baptiste 1998] :

$$O = (O_1 \times O_2) \bigcup \cup_{t \in O_1}(O_1 \times O(t)) \bigcup \cup_{t \in O_2}(O(t) \times O_2), \tag{2.12}$$

avec :

$$O_1 = \{ES_i, LS_i, EF_i \mid i \in \mathcal{A}\}, \ O_2 = \{LF_i, EF_i, LS_i \mid i \in \mathcal{A}\}, \ O(t) = \{ES_i + LF_i - t \mid i \in \mathcal{A}\}.$$

Ainsi, l'algorithme complet de consistance et d'ajustement peut être implémenté en $O(n^3)$ (voir p. ex. [Baptiste 2001]). Dans la pratique, il est parfois préférable de limiter l'étude à un sous-ensemble d'intervalles, comme par exemple aux seuls intervalles de la forme $[ES_i, EF_j]$, $[ES_i, LF_j]$, $[LS_i, EF_j]$ et $[LS_i, LF_j]$.

- ***Déduction de séquencements***

Enfin, le raisonnement énergétique détecte en $O(n^2)$ des précédences interdites $i \nrightarrow j$, en considérant deux activités i et j et l'énergie maximale disponible pour l'exécution de ces deux activités

sur l'intervalle particulier $\Delta = [ES_i, LF_j]$. La règle [Esquirol 2001] suivante généralise au cas non-disjonctif la règle (2.1) :

$$\left(\exists\, k \in \mathcal{R},\ \overline{e_{ik}}(\Delta) + \overline{e_{jk}}(\Delta) > E_k(\Delta) - \sum_{l \in \mathcal{A} \setminus \{i,j\}} \underline{e_{lk}}(\Delta) \right) \quad \Rightarrow \quad i \nrightarrow j. \tag{2.13}$$

2.2.5 Triplets symétriques

Brucker et al. [Brucker 1998] se sont intéressés aux ensembles critiques de trois activités dont l'une activité doit être exécutée en parallèle avec chacune des deux autres. Un tel ensemble est appelé *triplet symétrique*. Une première condition nécessaire de réalisabilité, évidente, permet de déduire de nouvelles disjonctions :

$$\forall \text{ ensemble critique } (i,j,h),\ (\ h \parallel i \text{ et } h \parallel j\ ) \quad \Rightarrow \quad i - j. \tag{2.14}$$

D'autres règles fixent le séquencement relatif (disjonctions, précédences et parallélismes) entre des paires d'activités. Nous les reportons ici sans démonstration.

Soit un triplet symétrique (i,j,h) avec $h \parallel i$ et $h \parallel j$, et une quatrième activité l.
Si,

$$l \parallel i \text{ et } l \parallel j,\ \text{alors } l \parallel h. \tag{2.15}$$

Si, au contraire, j,h, l ne peuvent être exécutées simultanément, et si, alternativement,

- $l \parallel i,\ \text{alors } j - l$ (2.16)
- $l \parallel i \text{ et } i \rightarrow j,\ \text{alors } l \rightarrow j$ (2.17)
- $l \parallel i \text{ et } j \rightarrow i,\ \text{alors } j \rightarrow l$ (2.18)
- i,h,l ne peuvent s'exécuter simultanément et $p_h - 1 \leq \min(p_i, p_j, p_l),\ \text{alors } h - l$ (2.19)
- $l \rightarrow i, j \rightarrow i \text{ et } p_h - 1 \leq \min(p_j, p_l),\ \text{alors } l \rightarrow h$ (2.20)
- $i \rightarrow l, i \rightarrow j \text{ et } p_h - 1 \leq \min(p_j, p_l),\ \text{alors } h \rightarrow l$ (2.21)
- $i \rightarrow l, j \rightarrow l \text{ et } p_h - 1 \leq \min(p_i, p_j),\ \text{alors } h \rightarrow l$ (2.22)
- $l \rightarrow i, l \rightarrow j \text{ et } p_h - 1 \leq \min(p_i, p_j),\ \text{alors } l \rightarrow h$ (2.23)
- $l \rightarrow i, l \rightarrow j \text{ et } p_h - 1 \leq \min(p_i, p_j),\ \text{alors } l \rightarrow h$ (2.24)

L'ensemble des triplets symétriques peut être calculé en $O(nm|P|)$ où P est l'ensemble des paires d'activités en parallèle (c.-à-d. vérifiant $b_{ij} \geq 1 - p_j$ et $b_{ji} \geq 1 - p_i$). Ainsi l'ensemble des tests s'effectuent en $O(n^2 m|P|)$.

2.2.6 Shaving

Le *shaving* a aussi pour but la réduction des domaines, mais tandis que les règles précédentes consistaient en la réécriture de contraintes de ressources, le shaving, quant à lui, génère des contraintes temporelles redondantes au problème par un tout autre moyen : une contrainte de domaine c est générée, si la propagation de la contrainte opposée $\neg c$ au problème détecte une inconsistance globale. Le shaving (on parle aussi d'*opération globale* [Carlier 1994]) est en fait le nom générique donné en ordonnancement ([Martin 1996]) aux techniques de consistance basées sur la réfutation en satisfaction de contraintes (voir SAC à la section 1.1.4). Les premières implémentations du shaving ont été faites pour les problèmes disjonctifs comme

le job-shop [Carlier 1994, Martin 1996, Péridy 1996], le flow-shop [Rivreau 1999] ou l'open-shop [Dorndorf 2001]. Malgré leur coût en espace et surtout en temps d'exécution, elles ont prouvé leur grande efficacité en résolvant des instances jusque là ouvertes.

Les techniques de shaving se différencient entre elles par le type de contraintes qu'elles tentent de réfuter et aussi par l'algorithme de consistance utilisé pour inférer la réfutation. L'inconsitance peut être prouvée en implémentant n'importe quel sous-ensemble de règles locales présentées ci-dessus. Parmi celles-ci, on choisira de préférence les moins coûteuses puisque la propagation sera appelée pour chaque contrainte c postée. Les contraintes testées sont, comme toujours pour le RCPSP, de deux sortes selon que l'on cherche à réduire les fenêtres de temps ou à résoudre les séquencements entre paires d'activités. Pour le RCPSP, le premier cas est considéré dans [Caseau 1996] (aussi dans [Néron 2001] pour le problème proche du flow-shop hybride) : les contraintes propagées sont du type $S_i \geq t$ et $S_i \leq t$ avec $t \in [ES_i, LS_i]$. La réfutation de ces contraintes entraîne, pour la première, l'ajustement de la date au plus tard de i, $LS_i = t-1$, et pour la seconde, la date au plus tôt $ES_i = t+1$. Les valeurs de t sont généralement prises par dichotomie ou selon une partition plus complexe de la fenêtre de temps.

Dans [Péridy 1996], Péridy prouve que, dans le cas, disjonctif, le shaving sur les fenêtres de temps domine le shaving sur les séquencements. La démonstration ne s'applique pas au cas cumulatif, puisqu'alors, non plus deux, mais trois positionnements relatifs sont possibles pour une paire quelconque d'activités : $i \rightarrow j$, $j \rightarrow i$ et $i \parallel j$. C'est pourquoi nous proposons un shaving sur les séquencements [Demassey 2003] consistant à ajouter temporairement au problème chacune des contraintes respectives $S_j - S_i \geq p_i$, $S_i - S_j \geq p_j$ et $S_j - S_i \in [1-p_j, 1-p_i]$. Cette technique est plus appropriée à la matrice des distances B qu'aux fenêtres de temps. La propagation de la contrainte $i \rightarrow j$, p. ex. , permet de mieux caractériser l'ensemble $\mathcal{S}_T^{i \rightarrow j}$ des solutions S de $\mathcal{S}_T$ telles que $S_j - S_i \geq p_i$. Si on note $B^{i \rightarrow j}$ la matrice des distances et $D^{i \rightarrow j}$ les disjonctions ainsi obtenus par propagation, alors :

$$D \subseteq D^{i \rightarrow j}, \quad \left(\forall\ (h,l) \in \mathcal{A}^2,\ b_{hl}^{i \rightarrow j} \geq b_{hl} \right) \quad \text{et} \quad \left(\forall\ S \in \mathcal{S}_T^{i \rightarrow j},\ S_h - S_l \geq b_{hl}^{i \rightarrow j} \right).$$

En particulier, si une inconsistance est détectée, on sait alors que l'ensemble $\mathcal{S}_T^{i \rightarrow j}$ est vide et donc que $i \nrightarrow j$ peut être ajoutée à la définition du problème initial. Si c'est la contrainte $i \parallel j$ qui est détectée comme inconsistante, alors on ajoute sa négation $i - j \in D$.

En fait, il y a plus rapide et surtout plus efficace que d'ajouter une contrainte dont la négation a été réfutée si on tient compte du fait que l'ensemble des solutions du problème se partitionne pour toute paire d'activités $\{i, j\}$ de la façon suivante :

$$\mathcal{S}_T = \mathcal{S}_T^{i \rightarrow j} \overset{\circ}{\bigcup} \mathcal{S}_T^{j \rightarrow i} \overset{\circ}{\bigcup} \mathcal{S}_T^{i \parallel j}$$

Si les trois conditions sont inconsistantes alors $\mathcal{S}_T$ est vide et le problème est résolu. Dans le cas contraire, nous dépassons le cadre usuel du shaving en effectuant, pour toute paire d'activité $\{i, j\}$ testée, la mise à jour de B et de D par :

$$\forall\ (h,l) \in \mathcal{A}^2,\ b_{hl} = \min(b_{hl}^{i \rightarrow j}, b_{hl}^{j \rightarrow i}, b_{hl}^{i \parallel j}) \tag{2.25}$$

$$D = D^{i \rightarrow j} \bigcap D^{j \rightarrow i} \bigcap D^{i \parallel j} \tag{2.26}$$

en ayant préalablement posé, si une relation $i \sim j$ est inconsistante, $b_{hl}^{i \sim j} = +\infty$ et $D^{i \sim j} = V \times V$.

Dans nos expérimentations, on verra que, bien que cette technique soit lourde et coûteuse, elle est particulièrement puissante (voir p. ex. les résultats des tables 4.1 et 4.2, pages 100 et 101). De

plus, dans une approche destructive pour le calcul de bornes, sa capacité à détecter les infaisabilités permet d'accroître fortement les résultats avec un faible surcoût de temps de calcul moyen (voir table 3.2, p. 81 ; en fait, le temps de calcul est même largement réduit pour de nombreuses instances parmi les plus dures). Pour les plus grandes instances, nous proposerons aussi une manière de limiter l'exécution du shaving à un certain nombre de paires d'activités pour économiser de l'espace et du temps de calcul sans trop dégrader les déductions.

Pour terminer, on notera que le shaving peut s'étendre, pour assurer un plus grand degré de consistance, en testant plusieurs contraintes à la fois (il s'apparente alors encore davantage au *maintien de consistance* (MAC) [Sabin 1994]). Parmi ces extensions, testées pour le job-shop, se trouvent le *double-shaving* [Péridy 1996] (contraintes sur les fenêtres de temps de deux activités) et le shaving sur les ensembles disjonctifs de k éléments [Torres 2000] ($k-1$ contraintes de type $i \rightarrow j$).

2.3 Formulations linéaires

Les travaux les plus anciens menés sur la résolution exacte du RCPSP font appel à la programmation linéaire en nombres entiers (voir, p. ex. , [Pritsker 1969, Balas 1970, Fisher 1973, Patterson 1976, Talbot 1978, Stinson 1978]). Dans ce problème, ce sont évidemment les contraintes de ressources qu'il est difficile de modéliser au moyen d'inégalités linéaires, puisqu'il est nécessaire de garder une trace de l'ensemble des activités en cours d'exécution à tout instant. La majorité des auteurs de modèles linéaires pour le RCPSP y remédient en discrétisant le temps au détriment d'une augmentation du nombre de variables, dépendant de la valeur d'un horizon d'ordonnancement T. On distinguera donc les modèles *en temps continu*, où les dates de début des activités sont modélisées par des variables réelles (et le séquencement entre les activités par des variables binaires), des modèles *en temps discrétisé*, où les variables binaires liées aux activités sont aussi indicés par les instants.

2.3.1 Temps continu

Formulation conceptuelle

$$\min S_{n+1} \tag{2.27}$$

sujet à :

$$S_j - S_i \geq p_i \qquad \forall\ (i,j) \in E \tag{2.28}$$

$$\sum_{j \in \mathcal{A}_t} r_{jk} \leq R_k \qquad \forall\ k \in \mathcal{R}, \forall\ t \in [0,T] \tag{2.29}$$

$$S_i \geq 0 \qquad \forall i \in V$$

où $\mathcal{A}_t = \{j \in \mathcal{A} \mid S_j \leq t < S_j + p_j\}$.

Les variables S_i représentent les dates de début des activités. L'objectif (2.27) est la minimisation de la date de début de la dernière activité fictive. (2.28) sont les contraintes de précédence. Les contraintes de ressources (2.29) signifient qu'à tout instant t et pour toute ressource k, la somme des consommations de k sur l'ensemble $\mathcal{A}_t$ des activités en cours à l'instant t est inférieure à la capacité de k.

Nous avons inclu dans la liste des modèles en temps continu, cette formulation conceptuelle du RCPSP. Bien qu'elle ne soit pas à proprement dit un programme linéaire ($\mathcal{A}_t$ n'est pas identifiable), cette formulation pose clairement le problème, et comme ici seules les contraintes de ressources (2.29) ne sont pas correctement modélisées, le programme obtenu après leur suppression est bien une relaxation linéaire du RCPSP et ses solutions optimales sont entières. Malheureusement, la valeur optimale de cette relaxation est faible puisque elle est égale à la longueur du *chemin critique* : le plus long chemin de 0 à $n+1$ dans le graphe des précédences $G = (V, E)$.

Ensembles critiques minimaux

Alvarez-Valdès et Tamarit [Alvarez-Valdés 1993] se basent sur la formulation disjonctive de Balas [Balas 1970] pour le RCPSP en modélisant les contraintes de ressources (2.29) au moyen des ensembles critiques d'activités minimaux $C \in \mathcal{C}m$ définis formellement par : (1) $C \times C \cap E = \emptyset$, (2) $\exists k \in \mathcal{R}$, $\sum_{i \in C} r_i k > R_k$ et (3)$\forall C' \subsetneq C$, $\forall k \in \mathcal{R}$, $\sum_{i \in C'} r_i k \leq R_k$.

$$\min S_{n+1} \tag{2.27}$$

sujet à :

$$x_{ij} = 1 \qquad \forall\ (i,j) \in E \tag{2.30}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall\ (i,j) \in V^2, i < j \tag{2.31}$$

$$x_{ik} \geq x_{ij} + x_{jk} - 1 \qquad \forall\ (i,j,k) \in V^3 \tag{2.32}$$

$$S_j - S_i \geq -M + (p_i + M)x_{ij} \qquad \forall\ (i,j) \in V^2 \tag{2.33}$$

$$\sum_{(i,j) \in C^2} x_{ij} \geq 1 \qquad \forall\ C \in \mathcal{C}m \tag{2.34}$$

$$x_{ij} \in \{0,1\},\ x_{ii} = 0 \qquad \forall\ (i,j) \in V^2$$

$$S_i \geq 0 \qquad \forall i \in V$$

Cette formulation introduit un second type de variables, binaires, pour modéliser les précédences dans une solution : x_{ij} est égal à 1 si l'activité i précède l'activité j, 0 sinon. (2.30) modélisent les contraintes de précédence initiales du problème, et les contraintes (2.34) imposent qu'au moins deux activités d'un ensemble critique minimal soient exécutées l'une après l'autre. Les contraintes (2.31) et (2.32) garantissent respectivement la transitivité et l'absence de cycles dans le graphe de précédence. Enfin, les contraintes (2.33) lient les deux types de variables du modèle : pour une valeur de M suffisament large (par exemple $M = T$), les contraintes imposent que, pour toute précédence $x_{ij} = 1$, l'activité j doit débuter son exécution après la complétion de i. Autrement, si $x_{ij} = 0$, la distance $S_j - S_i$ n'est pas contrainte ($S_j - S_i \geq -M$). Cette formulation contient un nombre exponentiel de contraintes (2.34) et l'ensemble des $\mathcal{C}m$ est donc rarement totalement énuméré.

Flots de ressources

Artigues [Artigues 2003] propose de remplacer, dans le modèle précédent, les contraintes de ressources (2.34) par :

$$f_{ijk} \leq \min(r_{ik}, r_{jk}) x_{ij} \qquad \forall (i,j) \in V^2, \forall k \in \mathcal{R} \tag{2.35}$$

$$\sum_{j \in V} f_{ijk} = r_{ik} \qquad \forall i \in V, \forall k \in \mathcal{R} \tag{2.36}$$

$$\sum_{i \in V} f_{ijk} = r_{jk} \qquad \forall j \in V, \forall k \in \mathcal{R} \tag{2.37}$$

$$f_{ijk} \in \mathbb{N} \qquad \forall (i,j) \in V^2, \forall k \in \mathcal{R}$$

Cette alternative est basée sur la notion de flots de ressources entre les activités : le flot d'unités d'une ressource k arrive en quantité r_{ik} à une activité i lorsque celle-ci démarre, et en repart lorsque i termine son exécution. Les activités 0 et $n+1$ sont respectivement la source et le puits de ce flot. Ici, on suppose donc que $r_{0k} = r_{(n+1)k} = R_k$ pour toute ressource k. Les variables de flots $f_{ijk} \in \mathbb{N}$, désignant le nombre d'unités de la ressource k libérées par i à sa complétion et envoyées à j au début de son exécution, sont liées par les contraintes usuelles de flots (2.32), (2.34) et (2.33). Cette modélisation contient davantage de variables mais seulement un nombre polynomial de contraintes et il ne nécessite pas le calcul préalable de l'ensemble $\mathcal{C}m$.

2.3.2 Temps discretisé

Les formulations en temps discretisé contiennent un nombre de variables dépendant de l'horizon T qu'on suppose donc connu. On pose ici $\mathcal{T} = \{0, \ldots, T\}$ l'ensemble des instants possibles de début des activités.

Instants de début

La première modélisation du RCPSP en un programme linéaire a été donnée par Pritsker et al. [Pritsker 1969] et ne contient qu'un seul type de variables, binaires pour figurer la date de fin des activités. Nous reportons ici le modèle équivalent avec les dates de début : $y_{it} = 1$ si i débute à l'instant t et 0 sinon. Au moyen de ces variables, on peut désormais caractériser $\mathcal{A}_t = \{i \in \mathcal{A} \mid \sum_{s=t}^{t+p_i-1} y_{is} = 1\}$ à tout instant t et donc modéliser les contraintes de ressources (2.29) par (2.41). Les contraintes de précédence (2.40) et de non-préemption (2.39) et l'objectif (2.38) se traduisent facilement avec la correspondance $S_i = \sum_{t \in \mathcal{T}} y_{it}$.

$$\min \sum_{t \in \mathcal{T}} t y_{(n+1)t} \tag{2.38}$$

sujet à :

$$\sum_{t=0}^{T} y_{it} = 1 \qquad \forall\ i \in V \tag{2.39}$$

$$\sum_{t=0}^{T} t(y_{jt} - y_{it}) \geq p_i \qquad \forall\ (i,j) \in E \tag{2.40}$$

$$\sum_{i \in V} r_{ik} \sum_{\tau=t-p_i+1}^{t} y_{i\tau} \leq R_k \qquad \forall\ k \in \mathcal{R}, \forall\ t \in \mathcal{T} \tag{2.41}$$

$$y_{it} \in \{0,1\} \qquad \forall\ i \in V, \forall\ t \in \mathcal{T}$$

Kaplan [Kaplan 1988] et Klein [Klein 2000] ont développé des formulations similaires à celle-ci. Kaplan définit les variables de décision y_{it} égales à 1 si i est en cours d'exécution au temps t. Dans sa formulation, les contraintes de ressource s'expriment plus simplement ($\mathcal{A}_t = \{i \in \mathcal{A} \mid y_{it} = 1\}$) mais par autant d'inégalités, et il y a alors n contraintes supplémentaires pour la durée des activités, et les contraintes de non-préemption et de précédence sont spécifiées par T fois plus d'inégalités. La formulation de Klein utilise les variables de décision $y_{it} = 1$ si $S_i + p_i < t$. Elle simplifie aussi les contraintes de ressources (2.41), ainsi que les contraintes de non-préemption et de précédence par rapport au modèle de Kaplan.

Christofides et al. [Christofides 1987] proposent une autre formulation des contraintes de précédence dans le modèle de Pritsker et al. :

$$\sum_{\tau=t}^{T} y_{i\tau} + \sum_{\tau=0}^{t+p_i-1} y_{j\tau} \leq 1 \qquad \forall\ (i,j) \in E, \forall\ t \in \mathcal{T} \tag{2.42}$$

Elles modélisent les relations $S_i \geq t \Rightarrow S_j \geq t + p_i\ \forall (i,j) \in E, \forall t \in \mathcal{T}$. On appelle ces inégalités, les contraintes de précédence *désagrégées* par opposition aux contraintes de précédence *agrégées* (2.40). Ces contraintes sont plus nombreuses que les contraintes originales mais la relaxation continue du programme linéaire correspondant est théoriquement plus forte. Uetz [Uetz 2001] présente un exemple d'instance où la relaxation désagrégée est une fois et demie meilleure que la relaxation agrégée en terme de bornes. Cependant, expérimentalement, le temps supplémentaire de calcul engendré par les inégalités désagrégées ne semble pas toujours contre-balancé par une amélioration significative de la borne (voir p. ex. [Cavalcante 2001, Möhring 2003] ou nos propres résultats, section 4.5).

Ensembles admissibles

Enfin, Mingozzi et al. [Mingozzi 1998] ont construit leur modèle sur la notion d'*ensemble admissible*. Un ensemble F d'activités de $\mathcal{A}$ est dit admissible si toutes les activités qui le composent sont autorisées à s'exécuter simultanément au vu des contraintes de précédence et de ressources. Tous les ensembles admissibles sont indicés par $l \in \mathcal{F}$ et ceux, parmi eux, contenant une activité i

donnée, par $l \in \mathcal{F}_i$.

$$\min \sum_{t=0}^{T} t y_{(n+1)t} \tag{2.38}$$

sujet à :

$$\sum_{t=0}^{T} y_{it} = 1 \qquad \forall\, i \in V \tag{2.39}$$

$$\sum_{t=0}^{T} t(y_{jt} - y_{it}) \geq p_i \qquad \forall\, (i,j) \in E \tag{2.40}$$

$$\sum_{l \in \mathcal{F}_i} \sum_{t=0}^{T} x_{lt} = p_i \qquad \forall\, i \in \mathcal{A} \tag{2.43}$$

$$\sum_{l \in \mathcal{F}} x_{lt} \leq 1 \qquad \forall\, t \in \mathcal{T} \tag{2.44}$$

$$y_{it} \geq \sum_{l \in \mathcal{F}_i} x_{lt} - \sum_{l \in \mathcal{F}_i} x_{lt-1} \qquad \forall\, t \in \mathcal{T}, \forall\, i \in \mathcal{A} \tag{2.45}$$

$$x_{lt} \in \{0,1\},\ x_{l(-1)} = 0 \qquad \forall\, l \in \mathcal{F}, \forall\, t \in \mathcal{T} \tag{2.46}$$

$$y_{it} \in \{0,1\}, \qquad \forall\, i \in V, \forall\, t \in \mathcal{T} \tag{2.47}$$

Ce modèle reprend les variables y_{it} et les contraintes de non-préemption (2.39) et de précédence (2.40) de Pritsker. Sont ajoutées à cela un nombre exponentiel de variables de décision x_{lt} définies pour tout ensemble admissible F_l et pour tout temps t : $x_{lt} = 1$ si F_l est l'ensemble des activités en cours d'exécution au temps t. Les contraintes (2.44) n'autorisent au plus qu'un ensemble admissible à être en cours d'exécution à un temps t. Les contraintes (2.43) garantissent que toute activité i s'exécute pendant p_i unités de temps et, (2.45), que i commence au temps t si i appartient à l'ensemble admissible en cours au temps t mais pas à celui en cours à $t-1$. Comme la durée du projet est la somme des durées des ensembles admissibles, la fonction objectif peut alternativement s'écrire :

$$(2.38) = \sum_{l \in \mathcal{F}} \sum_{t \in \mathcal{T}} x_{lt} \tag{2.48}$$

2.4 Revue de la littérature

De la vaste littérature consacrée au RCPSP et à ses formes dérivées, nous ne présentons dans cette section que les approches en rapport direct avec ce qui nous intéresse ici : l'utilisation de méthodes générales d'optimisation combinatoire pour la résolution exacte du RCPSP. Nous décrivons ainsi rapidement les principaux schémas de branchement 2.4.2 puis les utilisations faites de la programmation par contraintes 2.4.3 ou de la programmation linéaire 2.4.4 pour réduire la recherche, en particulier pour le calcul de bornes inférieures. Nous laissons de côté, les méthodes approchées ([Hartmann 2000, Kolisch 1999] sont deux bons états de l'art des heuristiques pour le RCPSP) ou encore les bornes inférieures plus « spécifiques » au problème (pour reprendre le terme de Klein et Scholl qui présentent quelques unes de ces bornes dans [Klein 1999]), comme les extensions de la borne du chemin critique[Stinson 1978] ou les bornes des problèmes à m-machines [Carlier 1991]. Nous commençons par présenter les principaux jeux d'instances de test de la littérature.

2.4.1 Benchmarks

Les instances de Patterson, bien qu'ayant été communément utilisées un temps, se sont avérées très faciles à résoudre avec les méthodes plus récentes. Elles ne sont donc plus utilisées. Aujourd'hui, les expérimentations sont le plus souvent menées sur les instances PSPLIB [PSPLIB] générées par le générateur de projets ProGen développé par Kolisch, Sprecher et Drexl [Kolisch 1998]. On note aussi ces jeux d'instances KSD30, KSD60, KSD90 et KSD120 contenant respectivement 480 instances de 30, 60 et 90 activités et 4 ressources, et 600 instances de 120 activités et 4 ressources. Les instances sont générées par séries de 10 selon 3 critères NC, RS et RF :

- *network complexity*, $NC \geq 1$, est une mesure de la densité du graphe de précédence puisqu'il indique le nombre moyen de successeurs directs des activités. Dans les instances KSD, NC vaut 1.5, 1.8 ou 2.1 ;
- *resource factor*, $RF \in [0, 1]$, indique le nombre moyen normalisé de ressources utilisées par une activité. Ici, RF vaut 0.25, 0.5, 0.75 ou 1 ;
- *resource strength*, $RS \in [0, 1]$, décrit la disponibilité normalisée des ressources de sorte que, $RS = 0$ correspond au cas où les capacités des ressources sont égales au maximum des consommations, et $RS = 1$ au cas où il n'y a aucun conflit de ressources. Pour les instances à 30, 60 et 90 activités, RS vaut 0.2, 0.5, 0.7, 1.0, et pour KSD120, RS vaut 0.1, 0.2, 0.3, 0.4, 0.5.

Les durées et consommations des activités sont prises aléatoirement entre 1 et 10. Les séries généralement considérées comme étant les plus difficiles sont celles correspondant à un RS faible et un RF fort, en particulier (pour $n < 120$) les séries 13, 29 et 45 ($RS = 0.2$, $RF = 1$), 9, 25 et 41 ($RS = 0.2$, $RF = 0.75$), ou encore 21 et 37 ($RS = 0.2$, $RF = 0.50$). À ce jour, les instances à 30 activités sont toute résolues optimalement mais il reste environ, 120 instances ouvertes dans KSD60, autant dans KSD90 et plus de 350 dans KSD120.

Baptiste et Le Pape [Baptiste 2000] ont remarqué que les instances non-triviales sont plutôt *hautement disjonctives*, autrement dit beaucoup de paires d'activités ne peuvent s'exécuter en parallèle, ce qui correspond à un fort *ratio de disjonctions* (à l'inverse on parle d'instances *hautement cumulatives*). Comme ce type d'instances n'est pas représentatif des problèmes réels du RCPSP, ils proposent dans ce même article, un nouveau jeu de tests « BL » contenant 40 instances de 20 ou 25 activités et 3 ressources, chaque ressource étant requise par toutes les activités avec une consommation comprise entre 0 et 60% de la ressource. Ces instances présentent en moyenne un ratio de disjonctions de 0.33 (contre 0.56 pour KSD) et sont donc qualifiées de hautement cumulatives. Il est intéressant de voir que ce critère peut être déterminant sur l'utilisation de telle ou telle autre méthode. Par exemple, les règles d'ajustement basées sur les ensembles disjonctifs sont nécessairement moins efficaces sur les instances hautement cumulatives.

2.4.2 Schémas de branchement

Les méthodes exactes qui ont été développées à ce jour pour le RCPSP sont toutes des procédures de recherche arborescente : PSE en recherche opérationnelle ou backtracking en programmation par contraintes. Toutes font donc le choix d'un schéma de branchement. On peut retrouver un état de l'art récent des procédures arborescentes pour le RCPSP dans [Néron 1999, Brucker 1999, Demeulemeester 2002]. Nous étudions ces méthodes suivant le classement établit dans [Néron 1999], en adoptant un point de vue moins intuitif mais plus proche du backtracking pour les CSP : la stratégie de branchement est présentée, quitte à remodéliser le problème, comme la séparation du domaine d'une unique variable. Dans les schémas chronologiques par exemple, cette séparation est dissimulée par la prise en compte implicite des contraintes de pré-

cédence au moment du branchement ou l'utilisation de règles de dominance telles que *left-shift* (un ordonnancement optimal est calé au plus tôt). Notre objectif est de mieux comprendre comment il serait possible d'adapter les méthodes du RCPSP aux procédures basées sur l'apprentissage, de type backtracking dynamique ou resolution search (chapitre 5).

Nous utilisons pour cela la notation formelle des CSP avec X l'ensemble des variables et D les domaines associés. Un branchement se traduit par la séparation du domaine D_k d'une variable X_k en deux ou plusieurs sous-domaines. Un backtrack intervient si tous les domaines sont réduits à un singleton ou si le domaine d'une variable est vide.

Schémas chronologiques

Les schémas chronologiques construisent progressivement des ordonnancements partiels, en fixant à chaque noeud de l'arbre de recherche, la date de début d'une ou de plusieurs activités le plus tôt possible par respect des contraintes de précédence et des contraintes de ressources. En réalité, la séparation ne se fait pas sur une décision du type ($S_i = t$ ou $S_j = t$) mais plutôt sur ($S_i \leq S_j$ ou $S_i \leq S_j$). Par inférence logique (consistance des contraintes de précédence,...), la décision impose que i ou j doit commencer à la date t. En général, les arbres chronologiques sont explorés en profondeur d'abord en choisissant d'exécuter à un instant t des activités qui « remplissent » la ressource ou encore des activités qui doivent être exécutées au plus vite compte tenu de leurs fenêtres de temps.

- ***Séquencements réalisables***

Le schéma de branchement proposé initialement par Patterson et al. [Patterson 1990] puis amélioré par Sprecher [Sprecher 1996] consiste à ajouter une par une les activités à un ordonnancement partiel. En fait, il s'agit d'une énumération implicite de l'ensemble des *séquencements réalisables* : les permutations $(X_0, \ldots, X_{n+1})$ du vecteur $(0, 1, \ldots, n+1)$ telles qu'il existe un ordonnancement réalisable S avec $S_{X_0} \leq S_{X_1} \leq \ldots \leq S_{X_{n+1}}$. On part de la séquence $(X_0 = 0)$ avec les domaines $D_i = \{0\}$ si $i = 0$, $D_i = \{1, \ldots, n+1\}$ sinon. On étend progressivement cette séquence en branchant dans l'ordre sur X_1, X_2, jusque X_{n+1}. On branche sur une variable X_k en l'instantiant à chacune des valeurs j du domaine D_k, réduit préalablement par consistance des contraintes de précédence et de ressources avec les décisions déjà prises pour $(X_1, \ldots, X_{k-1})$. Comme plusieurs séquencements peuvent correspondre à un même ordonnancement, des règles de dominance sont aussi propagées pour éliminer ces séquencements redondants. Enfin, à partir d'un séquencement réalisable, on détermine un ordonnancement réalisable par calcul de plus long chemin.

Baptiste et Le Pape [Baptiste 1999] emploient un schéma similaire mais développent un arbre binaire en séparant le domaine d'une variable X_k en deux sous-domaines $\{j\}$ et $D_k \setminus \{j\}$.

Stinson [Stinson 1978] et Mingozzi et al. [Mingozzi 1998] étendent un ordonnancement partiel à chaque noeud en y intégrant plusieurs activités à la fois. Le schéma de branchement est semblable au précédent puisqu'il construit un séquencement, non plus d'activités, mais d'ensembles admissibles (section 2.3.2) d'activités. Les activités sont donc ordonnancées par « tronçons ». Le branchement sur une variable se fait en l'instantiant à chacune des valeurs de son domaine, autrement dit, à chaque ensemble admissible possible compte tenu des contraintes de durée et de non-préemption des activités déja placées. Ce schéma est en relation directe avec la formulation linéaire de Mingozzi 2.3.2 pour le calcul d'une borne inférieure.

- ***Décalages minimaux et cut-set***

Les schémas chronologiques par blocs peuvent être considérés d'une autre manière au moyen des variables $X_0, X_1, \ldots, X_T$ instanciées dans cet ordre, où chaque X_t représente l'ensemble admissible en cours d'exécution au temps t. Pour représenter le schéma de branchement de Christofides

et al. [Christofides 1987] sur lequel est basée la méthode performante de Demeulemeester et Herroelen [Demeulemeester 1997], on définit une variable X_t pour modéliser à la fois l'ensemble I_t des activités terminées avant ou au temps t et un certain ensemble DA_t (éventuellement vide) d'activités à décaler, c.-à-d. devant débuter après le temps $t+1$. À chaque noeud de l'arbre est associé un temps t tel que toutes les variables X_s, $s < t$, sont instanciées ($D_s = \{I_s\} \times \{DA_s\}$) et X_t est partiellement instancié dans le sens où I_t est défini mais il existe plusieurs alternatives DA_t avec $DA_t \cap I_t = \emptyset$. De nouveau, les contraintes de précédence, de ressources ou encore la règle du left-shift permettent de réduire le nombre d'alternatives : on ne considère que les ensembles DA_t d'activités ayant tous leurs prédécesseurs dans I_t et dont l'ordonnancement au plus tôt (par respect des précédences et des DA_s pour $s < t$) produit un conflit de ressources. D'après une autre règle de dominance [Demeulemeester 1992], on peut encore restreindre l'ensemble des alternatives aux seules alternatives minimales pour cette définition. S'il existe plusieurs alternatives (donc non vides), on branche sur chacune d'elles. La contrainte de décaler les activités de DA_t au moins au temps $t+1$ permet d'ordonnancer (au plus tôt) d'autres activités, n'appartenant pas à I_t donc se terminant après le temps $t+1$. Soit J_t les nouvelles activités ordonnancées qui se terminent le plus tôt et t' leur date de fin, on pose alors $D_s = \{I_t\} \times \{\emptyset\}$, pour tout temps $s = t, \ldots, t'-1$ et on itère la recherche sur la variable $X_{t'}$ avec $I_{t'} = I_t \cup J_t$.

Une règle de dominance particulière est utilisée dans [Demeulemeester 1992], la règle du *cut-set*, basée sur les règles de [Stinson 1978, Talbot 1978]. Par son efficacité, elle a souvent été mise en oeuvre depuis pour améliorer les procédures arborescentes pour le RCPSP. Elle s'apparente, en fait, à une technique d'apprentissage où les ordonnancements partiels sont mémorisés. Un ordonnancement partiel est alors éliminé de la recherche dès qu'il est identifié comme étant dominé par un autre précédement construit. La gestion des ordonnancements partiels est généralement assez sommaire et il serait intéressant de la comparer avec des techniques de backtrackings intelligents plus évolués.

Fenêtres de temps

Carlier et Latapie [Carlier 1991] ont proposé un tout autre schéma de branchement en séparant sur les fenêtres de temps des activités. Il s'agit d'un backtracking sur un modèle CSP où les variables X sont les dates de début des activités et les domaines D, les fenêtres de temps $[ES, LS]$. Un branchement est effectué pour une activité i en séparant sa fenêtre par dichotomie : $[ES_i, LS_i - \lfloor M_i/2 \rfloor]$ et $[ES_i + \lceil M_i/2 \rceil, LS_i]$, avec $M_i = LS_i - ES_i$. Cette méthode dépend fortement du choix de l'activité à un noeud donné. L'activité est sélectionnée de façon à maximiser l'amélioration de la borne inférieure basée sur des problèmes à m machines.

Plus récemment, Dorndorf et al. [Dorndorf 2000] ont utilisé un schéma de branchement binaire similaire mais en séparant la fenêtre de temps d'une activité i selon la décision $S_i = ES_i$ ou $S_i > ES_i$.

Schémas d'ordonnancement

Un autre schéma non-chronologique est proposé par Brucker et al. [Brucker 1998]. Il s'appuie aussi sur une formulation CSP basée sur le séquencement relatif des paires activités : les paires en précédence ou *conjonction* (C), en disjonction (D), ou en parallèle (P). Il est ainsi possible de créer un arbre binaire ou ternaire. Dans la version binaire, on branche sur la décision $i - j \vee i \parallel j$, tandis que dans la version ternaire, on branche sur $i \rightarrow j \vee j \rightarrow i \vee i \parallel j$. Comme pour le schéma précédent, le choix de la paire d'activités à considérer est important et est déterminé selon l'amélioration d'une borne inférieure.

À chaque noeud de l'arbre correspond un *schéma d'ordonnancement* (C, D, P), et à chaque feuille, un schéma d'ordonnancement tel que $C \cup D \cup P = V$. En effet, quand toutes les paires d'activités sont liées par une relation (précédence, disjonction ou parrallélisme), les auteurs montrent qu'il est possible, en temps polynomial, soit de construire un ordonnancement réalisable dominant tout autre ordonnancement représenté par le même schéma (C, D, P), soit de prouver qu'il n'existe pas de tels ordonnancements réalisables. L'algorithme construit une orientation transitive de (C, D, P), en orientant arbitrairement toutes les disjonctions de sorte que le graphe de précédence ainsi obtenu soit transitif. Si une telle orientation existe, alors elle correspond à un unique ordonnancement dominant obtenu par calcul des plus longs chemins à partir de 0.

2.4.3 Évaluation par propagation de contraintes

Toutes les méthodes présentées précédemment utilisent la propagation de contraintes pour l'évaluation des noeuds, implicitement (les contraintes de précédence dans les schémas chronologiques) ou explicitement. Les bornes élémentaires du RCPSP, comme les extensions de la borne du chemin critique, relèvent toutes aussi d'un raisonnement logique sur les contraintes. Nous présentons ici les méthodes pour le RCPSP employant des techniques PPC plus avancées, en particulier les tests de consistance présentés à la section 2.2, pour évaluer la réalisabilité des noeuds dans un backtracking ou indirectement pour calculer des bornes inférieures.

Utilisation des tests de consistance

Des algorithmes de filtrage et tests de consistance sont employés presque systématiquement dans les méthodes de résolution actuelles pour le RCPSP. Il est cependant primordial de mettre en balance le gain apporté par l'utilisation de techniques avancées et le coût, en mémoire et en temps de calcul, de leur exécution.

La règle (2.1) sur les paires en disjonction, est de loin la plus fréquemment employée dans la littérature mais on trouve aussi des applications du edge-finding disjonctif [Brucker 1998, Dorndorf 2000] ou cumulatif [Baptiste 1996], du raisonnement énergétique[Caseau 1996, Baptiste 2000, Dorndorf 2000, Néron 2001, Carlier 2003] ou encore du shaving [Caseau 1996, Néron 2001].

En particulier, Caseau et Laburthe [Caseau 1996] ont étendu le concept d'*intervalles de tâches* au cas cumulatif. Pour tout couple $(i, j) \in V^2$, l'intervalle de tâches I_{ij} est l'ensemble des activités k telles que $ES_i \leq ES_k$ et $LF_k \leq LF_j$. Des règles du edge-finding cumulatif s'appliquent directement aux intervalles de tâches comme par exemple, la condition d'inconsistance $E_k(I_{ij}) > R_k(LF_j - ES_i)$. L'algorithme de propagation proposé effectue tous les ajustements en $O(n^3)$ dans le pire des cas, mais surtout la structure de données utilisée est rapidement recalculée à chaque noeud d'un arbre de recherche par un algorithme incrémental.

Une autre structure de données, *time-table* [Le Pape 1988], permet aussi de propager les contraintes de ressources de façon incrémentale. Il s'agit de conserver la trace de l'occupation des ressources à tout instant t de manière à maintenir la consistance des contraintes de ressources avec chaque activité nouvellement (même partiellement) ordonnancée. Par exemple, sachant qu'une activité i doit nécessairement être en cours d'exécution dans un intervalle de temps donné, on pourra alors supprimer r_{ik} unités à la capacité de la ressource k sur cet intervalle. Couplé à ce mechanisme, le edge-finding cumulatif a un plus grand pouvoir de déduction.

Enfin, Baptiste et al. [Baptiste 2001] ont mis à jour des dominances entre différentes règles d'ajustement applicables au RCPSP. Ils proposent aussi une étude expérimentale des règles plus

complexes du edge-finding cumulatif et du raisonnement énergétique, sur un schéma de branchement non-chronologique (pour une activité i, un branchement est effectué sur chacune des valeurs possibles de $S_i : ES_i, \ldots, LS_i$). Tandis que le edge-finding cumulatif ne semble rien apporter en général, le comportement du raisonnement énergétique dépend lui clairement du type d'instances testées. En effet, son coût en temps de calcul se ressent fortement sur les instances hautement disjonctives KSD. Au contraire, il prouve son efficacité sur les instances hautement cumulatives BL.

Comme il a été vu à la section 1.1.7, la résolution du RCPSP uniquement par des techniques de programmation par contraintes (en particulier, sans calcul explicite de borne inférieure) peut s'effectuer de plusieurs façons. La procédure la plus fréquement employée dans la littérature consiste à calculer initialement une borne inférieure LB et une solution réalisable de durée totale UB. On résoud alors successivement, par backtracking et pour différentes valeurs de T comprises entre LB et UB, des instances décisionnelles du RCPSP sous la forme d'un CSP avec la contrainte additionnelle $S_{n+1} < T$. Dans [Baptiste 2000], par exemple, la plus grande valeur T pour laquelle le CSP correspondant est insatisfiable, est recherchée par dichotomie sur l'intervalle $[LB, UB]$. Dans [Dorndorf 2000], le CSP considéré est modifié pendant la recherche arborescente : dès qu'une meilleure solution est trouvée à une feuille de l'arbre, la contrainte $S_{n+1} < T$ est raffinée en mettant à jour T avec la durée de la nouvelle solution. À noter enfin, que la procédure proposée dans [Caseau 1996] emploie une forme basique de *nogood recording* (voir section 1.1.5) pour résoudre des instances simples du RCPSP.

Bornes destructives

Les techniques de filtrage permettent aussi de déterminer seules des bornes inférieures. En effet, comme précédement, on peut tester la consistance du problème (ou d'une relaxation du problème) pour différentes valeurs de T prises, par exemple, par dichotomie sur un intervalle. Une borne inférieure est donnée par la plus grande valeur entière T pour laquelle on prouve qu'il n'existe pas d'ordonnancement de durée strictement inférieure.

Les bornes calculées de cette manière sont appelées *bornes destructives* par Klein et Scholl [Klein 1999]. La procédure qu'ils proposent consiste à tester dans un premier temps s'il existe un ordonnancement réalisable de durée au plus T, par réduction des fenêtres de temps en ne considérant que quelques règles de base (certaines contraintes sont donc relâchées). Dans un deuxième temps, si l'inconsistance n'a pas été détectée après réduction, différentes bornes inférieures classiques LB sont calculées. Si $LB > T$ alors la valeur T est réfutée et le processus est itéré avec $T + 1$. Dans le cas contraire, T est une borne inférieure.

À ce jour, les meilleures bornes inférieures connues pour le RCPSP (du moins sur les instances KSD) sont des bornes destructives : la borne de Brucker et Knust [Brucker 2000] et celles proposées dans cette étude (voir chapitres 3 et 4). Brucker et Knust utilisent un schéma destructif dichotomique. La réfutation de la valeur T se fait en deux temps, par propagation de contraintes pour le filtrage des domaines, puis par résolution d'un programme linéaire (voir section 2.4.4 et chapitre 3). Leur algorithme de propagation de contraintes comprend la consistance de chemin sur les précédences (`floyd-warshall`), les règles des triplets symétriques, la règle sur les paires disjonctives et les règles du edge finding appliqué à des EDM calculés comme indiqué section 2.2.2.

Avec cette procédure, Brucker et Knust appliquent une forme de coopération entre les deux solveurs PPC-PL. La programmation par contraintes est utilisée pour prétraiter le programme linéaire et à l'inverse, le programme linéaire peut être considéré comme une contrainte globale permettant de détecter l'inconsistance de T.

L'approche destructive est efficace mais elle implique un effort de calcul plus important en itérant plusieurs fois le processus. Néanmoins, une borne peut aussi être, à la fois, meilleure et calculée plus rapidement dans un schéma destructif plutôt que directement de manière *constructive*, comme le montrent les résultats de la section 4.5. Ce comportement est similaire à celui du shaving qui est aussi basé sur le principe de réfutation.

2.4.4 Bornes inférieures issues de la programmation linéaire

Les procédures exactes pour le RCPSP basées uniquement sur un backtracking PPC sont moins nombreuses que les PSE où une borne inférieure est calculée à chaque noeud. Les bornes inférieures sont généralement calculées de manière « spécifique » comme les bornes du chemin critique ou la relaxation à des problèmes à m machines. D'autres font appel à la relaxation d'un programme linéaire en nombres entiers modélisant le problème.

Générations de coupes

Christofides et al. [Christofides 1987] proposent une borne obtenue par relaxation continue de la formulation en temps discrétisé de Pritsker (2.3.2). Cette relaxation est resserrée (ou modifiée) par quatre types de coupes. Les contraintes de précédence désagrégées (2.42) sont un premier type de coupes. Elles ne sont pas ajoutées au modèle mais elles remplacent les contraintes agrégées (2.40). Les contraintes de ressources (2.41) sont aussi modifiées pour prévenir les activités réelles de s'exécuter simultanément à $n+1$ (en posant $r_{(n+1)k} = R_k$) :

$$\sum_{i \in V} r_{ik} \sum_{\tau=t-p_i+1}^{t} y_{i\tau} + R_k \sum_{s=0}^{t} y_{(n+1)s} \leq R_k \qquad \forall\, k \in \mathcal{R}, \forall\, t \in \mathcal{T} \tag{2.49}$$

Des *coupes de cliques* basées sur des ensembles disjonctifs C sont ajoutées, signifiant qu'au plus une activité de C est en cours au temps t :

$$\sum_{i \in C} y_{it} \leq 1, \quad \forall t \in \mathcal{T} \tag{2.50}$$

Les résultats présentés indiquent une certaine amélioration apportée par ces coupes (en particulier les coupes sur les ensembles disjonctifs et les précédences désagrégées), mais l'augmentation du temps de calcul généré n'est pas rapportée.

Sankaran et al [Sankaran 1999] proposent des coupes pour ce même modèle avec les contraintes de précédence désagrégées. Ils génèrent à la volée des inégalités de cliques et des inégalités de recouvrement minimal, toutes déterminées de manière heuristique. Aussi, dans un premier temps, ils diminuent le nombre des variables en calculant des fenêtres de temps des activités.

Nous reprenons le principe de génération de coupes au chapitre 4.

Relaxation lagrangienne

Christophides et al. [Christofides 1987], et avant eux Fischer [Fisher 1973], ont présenté une seconde borne à partir du modèle discrétisé en considérant la relaxation lagrangienne des contraintes de ressources (2.41). En fait, cette borne est nécessairement moins bonne que la relaxation continue du programme avec contraintes désagrégées puisque le polytope formé des inégalités (2.39), (2.42) et (2.41) est entier (voir p. ex. [Sankaran 1999, Möhring 2003]).

Cependant, pour cette formulation, la relaxation lagrangienne est bien plus rapide à calculer que la relaxation continue. Möhring et al. [Möhring 2003] se sont ainsi intéressé à cette même relaxation lagrangienne pour obtenir un bon compromis entre la valeur de la borne et le temps de calcul. La rapidité de leur procédure est principalement dûe à la résolution en temps polynomial des sous-problèmes lagrangiens par un calcul de coupe de capacité minimale dans un graphe. À noter aussi qu'ils incluent des coupes de cliques et les contraintes de ressources (2.49) de Christophides. Comme nous utilisons pleinement leur procédure, nous y reviendrons plus en détail au chapitre 3.

Ensembles admissibles

Mingozzi et al. [Mingozzi 1998] dérivent trois nouvelles bornes en relaxant leur formulation basée sur les ensembles admissibles d'activités. $LB1$ est la valeur optimale du PLNE : (2.48) sujet à (2.43) et (2.46) (voir section 2.3.2). Soit x_l une variable entière représentant la durée d'exécution de l'ensemble admissible l. Ce problème s'écrit alors :

$$\min \sum_{l \in \mathcal{F}} x_l \tag{2.51}$$

sujet à :

$$\sum_{l \in \mathcal{F}_i} x_l = p_i \qquad \forall\, i \in \mathcal{A} \tag{2.52}$$

$$x_l \in \mathbb{N} \qquad \forall\, l \in \mathcal{F} \tag{2.53}$$

$LB1$ correspond à la relaxation des contraintes de préemption et de précédence. $LB2$ est obtenue en relâchant, dans ce programme, les contraintes d'intégrité (2.53) et l'égalité dans les contraintes (2.52) en une inégalité $\geq$. De cette façon, les auteurs montrent qu'il suffit alors de considérer uniquement, à la place de $\mathcal{F}$, l'ensemble $\mathcal{F}m$ des ensembles admissibles minimaux (ou irréductibles). Ils prouvent aussi que $LB2$ est au moins aussi bonne que la borne du chemin critique. Malgré tout, le nombre de variables des programmes linéaires à résoudre pour obtenir $LB1$ et $LB2$ reste trop important.

Pour la borne $LB3$, le dual du programme correspondant à $LB2$ est considéré :

$$\max \sum_{i \in \mathcal{A}} p_i u_i \tag{2.54}$$

sujet à :

$$\sum_{i \in F_l} u_i \leq 1 \qquad \forall\, l \in \mathcal{F}m \tag{2.55}$$

$$u_i \geq 0 \qquad \forall\, i \in \mathcal{A} \tag{2.56}$$

$LB3$ est le coût d'une solution heuristique du problème de *set-packing* suivant : (2.54) sujet à (2.55) et $u_i \in \{0,1\}$, $\forall i \in \mathcal{A}$. Ce problème se réduit en fait à résoudre un problème de *weighted node packing* sur le graphe de la relation de parallélisme entre les paires d'activités. Cette dernière borne a souvent été utilisée depuis. Elle a permis en particulier d'améliorer la PSE de [Demeulemeester 1997].

Brucker et al. [Brucker 1998] utilisent, dans leur PSE, la génération de colonnes pour parer au nombre de variables du programme correspondant à $LB2$. Dans la version destructive de cette borne, Brucker et Knust [Brucker 2000] prennent aussi en compte les fenêtres de temps des activités, préalablement ajustées par PPC, pour renforcer la relaxation. Leur procédure est détaillée au

chapitre 3.

Linear lower bounds

Carlier et Néron [Carlier 2000] ont récemment produit une borne inférieure originale, MEPB, pour le RCPSP basée sur l'énumération explicite de toutes les consommations pouvant être satisfaites simultanément pour des ressources de petite capacité. Cette borne présente le très net avantage d'utiliser du temps de calcul uniquement à la racine de l'arbre de recherche, puisqu'elle dépend linéairement des durées des activités. Le RCPSP est tout d'abord relâché en autant de *CuSP* qu'il y a de ressources, en relâchant les contraintes de précédence en fenêtres de temps d'exécution des activités et, pour chaque $CuSP_k$, toutes les contraintes de ressources sauf pour une ressource k. Une *linear lower bound* (LLB) pour le $CuSP_k$ est schématiquement une borne inférieure de la durée optimale d'ordonnancement se présentant comme une fonction linéaire sur les durées des activités $\sum_{i\in\mathcal{A}} a_i p_i$. Par exemple, la borne du chemin critique est une LLB ($a_i = 1$ si i appartient au chemin critique, et 0 sinon), mais aussi la *borne basique* avec $a_i = r_{ik}/R_k$ ou encore, d'après Mingozzi (voir section précédente 2.4.4), toute borne $\sum_{i\in\mathcal{A}} u_i p_i$ telle que u vérifie (2.55) et (2.56).

La borne MEPB présentée dans [Carlier 2000], est aussi une LLB, basée cette fois sur la relaxation *multi-élastique préemptive* du CuSP où plusieurs parties d'une activité i peuvent être exécutées simultanément. Elle a la particularité d'être précalculée par programmation linéaire en prenant en compte les consommations des activités. Dans [Carlier 2003], Carlier et Néron proposent un schéma général de calcul de bornes inférieures pour le RCPSP à partir des LLB. Ce schéma se présente comme une nouvelle relaxation du RCPSP en un programme linéaire dans lequel, autant de LLB désirées peuvent être prises en compte sous forme de coupes. L'horizon d'ordonnancement est découpé en L intervalles $I_l = [t_{l-1}, t_l]$, où les variables de décisions sont les t_l ainsi que les x_{il} correspondant à la durée pendant laquelle s'exécute l'activité i sur l'intervalle I_l :

$$\min t_L \tag{2.57}$$

sujet à :

$$\sum_{l=1}^{L} x_{il} = p_i \qquad \forall\, i \in \mathcal{A} \tag{2.58}$$

$$x_{il} \leq t_l - t_{l-1} \qquad \forall\, i \in \mathcal{A},\ \forall\, l \in \{1,\ldots,L\} \tag{2.59}$$

$$\sum_{s=1}^{L} (p_i x_{js} - p_j x_{is}) \geq 0 \qquad \forall\, (i,j) \in E,\ \forall\, l \in \{1,\ldots,L\} \tag{2.60}$$

$$t_{l-1} \leq t_l \qquad \forall\, l \in \{1,\ldots,L\} \tag{2.61}$$

$$x_{il} \geq 0 \qquad \forall\, i \in \mathcal{A},\ \forall\, l \in \{1,\ldots,L\} \tag{2.62}$$

$$t_l \geq 0 \qquad \forall\, l \in \{1,\ldots,L\} \tag{2.63}$$

$$\tag{2.64}$$

Trivialement, le modèle précédent produit une borne inférieure au RCPSP qui n'est d'ailleurs pas nécessairement meilleure que la borne du chemin critique. La fonction linéaire $x \mapsto \sum_{i\in\mathcal{A}} a_i x_i$ associée à toute LLB d'un $CuSP_k$ déduit du RCPSP définit pour ce modèle, l'ensemble des inégalités valides suivantes :

$$\sum_{i\in\mathcal{A}} a_i x_{il} \leq t_l - t_{l-1}, \quad \forall l \in \{1,\ldots,L\} \tag{2.65}$$

Par exemple, la valeur optimale de ce programme incluant les coupes de la LLB associée à la borne de Mingozzi et al. est plus forte que la borne-même. Dans [Carlier 2003], plusieurs applications sont proposées en choisissant différentes LLB et différentes valeurs de L.

Chapitre 3

Calcul de Bornes Inférieures par Relaxation Lagrangienne

Dans ce chapitre, nous présentons une première méthode hybride alliant une relaxation « contraintes » et une relaxation linéaire du RCPSP pour le calcul de bornes inférieures constructives et destructives. Dans l'approche constructive, une relaxation PPC classique est utilisée en prétraitement d'une relaxation linéaire dont la résolution retourne une borne inférieure de la durée d'un ordonnancement optimal. Dans l'approche destructive, la coopération est plus poussée puisqu'alors la relaxation linéaire, encore prétraitée par le filtrage PPC, est assimilée à une règle de consistance dans le but de prouver (si le filtrage seul ne l'a déjà fait) l'infaisabilité d'une durée maximale d'ordonnancement T donnée. La borne inférieure correspond alors à la plus petite valeur T pour laquelle on ne prouve pas l'infaisabilité. La technique de programmation linéaire que nous avons employée ici est la relaxation lagrangienne appliquée à la formulation sur les ensembles admissibles [Mingozzi 1998], ou bien, appliquée à une relaxation préemptive de cette formulation [Brucker 2000]. De cette façon, le sous-problème lagrangien se décompose en plusieurs petits problèmes de sac-à-dos multidimensionnel*, ainsi que, dans le premier cas, un problème d'ordonnancement avec fenêtres de temps et sans contraintes de ressources, qui, suivant [Möhring 2003], se résoud en temps polynomial par calcul d'une coupe minimale dans un graphe orienté. Nous commençons par présenter ce principe de relaxation lagrangienne pour la formulation complète, puis pour la formulation préemptive. Nous n'avons expérimenté que la borne destructive appliquée à la formulation préemptive, mais une implémentation proche est envisageable pour la version constructive et pour la formulation complète. Nous détaillons ensuite notre implémentation : la résolution du dual lagrangien par génération de contraintes ou par l'algorithme du sous-gradient et l'algorithme de filtrage PPC utilisé. Nous terminons ce chapitre par les résultats expérimentaux en les comparant à la borne lagrangienne de Möhring et al. ou encore à la borne « quasi- »duale obtenue par génération de colonnes par Brucker et Knust. Ce travail a été mené en collaboration avec Philippe Baptiste, qui nous a premièrement suggérer la décomposition des sous-problèmes dans le cas préemptif en problèmes de sac-à-dos.*

3.1 Relaxation lagrangienne du modèle des ensembles admissibles

Pour une instance du RCPSP, un ordonnancement réalisable de durée inférieure ou égale à T se caractérise, de manière unique et propre à cet ordonnancement, par la séquence $B_0, B_1, \ldots, B_T$, où B_t est l'ensemble, éventuellement vide, des activités réelles en cours d'exécution au temps t (c.-à-d. entre t et $t+1$). Pour la réalisabilité, ces blocs sont nécessairement constitués d'activités dont l'exécution simultanée à un instant t ne viole ni les contraintes de précédence, ni les contraintes de ressources du problème. Autrement dit, pour déterminer un ordonnancement réalisable, il suffit d'associer à chaque instant un ensemble d'activités de ce type, appelé *ensemble admissible*, en s'assurant que

1° les ensembles contenant une activité i donnée sont actifs à exactement p_i instants consécutifs (non-préemptivité et durée des activités) et,

2° si l'activité i doit précéder l'activité j alors aucun ensemble admissible contenant i ne peut être actif plus tard qu'un ensemble contenant j (contraintes de précédence).

L'ordonnancement optimal $\mathcal{S} = (0, 2, 6, 2, 7, 10, 12)$ de la figure 2.1, page 41 par exemple, est spécifié par la suite des ensembles admissibles : $\{1\}$ actif sur l'intervalle de temps $[0, 2[$, $\{2, 5\}$ sur $[2, 5[$, $\{2, 3\}$ sur $[5, 6[$, $\{3, 4\}$ sur $[6, 7[$, $\{6\}$ sur $[7, 10[$, $\{7\}$ sur $[10, 12[$ et enfin $\emptyset$ « actif » à partir de l'instant 12.

Exploitant ce résultat, Mingozzi, Maniezzo, Ricciardelli et Bianco [Mingozzi 1998] ont introduit la formulation linéaire en nombres entiers (EA) pour le RCPSP, qui est présentée au chapitre précédent (section 2.3.2, p. 55). On reprend dans ce chapitre les notations utilisées ci-avant :

- $\mathcal{A} = \{1, \ldots, n\}$ est l'ensemble des activités réelles, $V = \mathcal{A} \cup \{0, n+1\}$ avec les activités fictives de début et de fin, E la relation de précédence sur V ;
- $T \in \mathbb{N}^*$ est la durée maximale des ordonnancements réalisables considérés et $\mathcal{T} = \{0, \ldots, T\}$;
- les ensembles admissibles F_l sont indicés sur $\mathcal{F}$ et ceux contenant en particulier une activité réelle $i \in \mathcal{A}$ sont indicés sur $\mathcal{F}_i$;
- dans un ordonnancement réalisable (de durée inférieure ou égale à T), la date de début S_i d'une activité réelle ou fictive $i \in V$ est modélisée par des variables binaires $y_{i0}, \ldots, y_{iT}$ avec la correspondance $y_{it} = 1$ si et seulement si $S_i = t$;
- les dates d'exécution d'un ensemble admissible F_l, sont modélisées par les variables binaires $x_{l(-1)}, x_{l0}, \ldots, x_{lT}$ avec $x_{l(-1)} = 0$ et la correspondance $x_{lt} = 1$ si et seulement si les activités de F_l (et seulement ces activités) sont en cours d'exécution au temps t.

La modélisation nécessite l'identification préalable de tous les blocs potentiels et donc de tous les ensembles admissibles du problème. Cependant, il n'est généralement pas possible de tous les déterminer en un temps raisonnable, même pour des instances parmi les plus petites de la littérature, et ce modèle linéaire contient un nombre exponentiel de variables. Mingozzi et al. ont donc été amenés à considérer des relaxations fortes de ce modèle (autorisation de la préemption et relaxation partielle des contraintes de précédence), obtenant par dualité, un programme linéaire contenant un moindre nombre (toujours important) de variables mais pouvant être traité efficacement de manière heuristique (voir section 2.4.4).

Nous proposons dans cette section de traiter la formulation linéaire initiale (EA) en parant à sa taille par relaxation lagrangienne. Cette relaxation consiste à dualiser à la fois les contraintes $y_{it} \geq \sum_{l \in \mathcal{F}_i} (x_{lt} - x_{lt-1})$ (2.45) liant les deux types de variables du modèle (on associe à chacune de ces inégalités un multiplicateur $\lambda_{it} \in \mathbb{R}_+$, $i \in \mathcal{A}$ et $t \in \mathcal{T}$) et les contraintes $\sum_{l \in \mathcal{F}_i} \sum_{t=0}^{T} x_{lt} p_i$ (2.43) de durée des activités (égalités associées à des multiplicateurs $\mu_i \in \mathbb{R}$, $i \in \mathcal{A}$).

Les contraintes du sous-problème lagrangien ne faisant plus intervenir chacune qu'un seul type de variables y_{it} ou x_{lt}, cela induit aisément une séparation de ce sous-problème selon ces variables. Ainsi, après séparation, le dual lagrangien de (EA) s'écrit :

$$(DEA) \ : \quad \max\{\sum_{i\in\mathcal{A}} \mu_i p_i + \phi_{\lambda\mu} + \psi_\lambda \mid \lambda_{it} \in \mathbb{R}_+, \mu_i \in \mathbb{R}, \forall i \in \mathcal{A}, t \in \mathcal{T}\}$$

où $\phi_{\lambda\mu}$ et ψ_λ sont respectivement les valeurs optimales des PLVB $(P_{\lambda\mu})$ et (P_λ) suivants :

$$(P_{\lambda\mu}) \quad \min \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{A}} \lambda_{it}(\sum_{l\in\mathcal{F}_i} x_{lt} - \sum_{l\in\mathcal{F}_i} x_{l(t-1)}) - \sum_{i\in\mathcal{A}} \mu_i \sum_{l\in\mathcal{F}_i}\sum_{t\in\mathcal{T}} x_{lt} \tag{3.1}$$

$$\text{sujet à :} \quad (2.44),\ (2.46)$$

où n'interviennent que les variables sur les ensembles admissibles x_{lt}, et :

$$(P_\lambda) \quad \min \sum_{t\in\mathcal{T}} t y_{(n+1)t} - \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{A}} \lambda_{it} y_{it} \tag{3.2}$$

$$\text{sujet à :} \quad (2.39),\ (2.40),\ (2.47)$$

où n'interviennent que les variables de dates de début des activités y_{it}.

On regarde maintenant comment ces deux problèmes, pour tout vecteur (λ, μ) des multiplicateurs de lagrange, peuvent facilement être résolus à l'optimum, le premier se reformulant en T problèmes simples de *sac-à-dos multidimensionnels* et le second étant un *problème d'ordonnancement de projet avec coûts dépendant des dates de début*.

3.1.1 Problèmes de sac-à-dos multidimensionnels

Simplifiant la notation, en posant, pour toute activité réelle $i \in \mathcal{A}$ et pour tout temps $t \in \mathcal{T}$:

$$\nu_{it} = \begin{cases} \lambda_{it} - \lambda_{i(t+1)} - \mu_i & \text{si } t \in \{0, \dots, T-1\}, \\ \lambda_{it} - \mu_i & \text{si } t = T, \end{cases}$$

alors le premier problème $(P_{\lambda\mu})$ se réécrit comme suit :

$$\min \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{A}} \nu_{it} \sum_{l\in\mathcal{F}_i} x_{lt} \tag{3.3}$$

$$\text{sujet à :}$$

$$\sum_{l\in\mathcal{F}} x_{lt} \leq 1 \qquad \forall\ t \in \mathcal{T} \tag{2.44}$$

$$x_{lt} \in \{0,1\} \qquad \forall\ l \in \mathcal{F}, \forall\ t \in \mathcal{T} \tag{2.46}$$

Le problème se décompose encore clairement en T sous-problèmes indépendants où il s'agit de déterminer, à chaque temps t, l'unique ensemble admissible, éventuellement vide, $l_t \in \mathcal{F}$ actif qui minimise le coût $\sum_{i\in F_{l_t}} \nu_{it}$. Plus formellement, la proposition suivante est vérifiée :

Proposition 3

$$\phi_{\lambda\mu} = \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{A}} \nu_{it}\xi^i_t$$

où, pour chaque temps $t \in \mathcal{T}$, ξ^i_t est une solution optimale du problème de sac-à-dos multidimensionnel suivant :

$$
\begin{aligned}
(KS_{(\lambda\mu,t)}) \quad & \min \sum_{i\in\mathcal{A}} \nu_{it}\xi^i \\
& \textit{sujet à :} \\
& \sum_{i\in\mathcal{A}} r_{ik}\xi^i \le R_k \qquad \forall\ k \in \{1,\dots,m\} \\
& \xi^i + \xi^j \le 1 \qquad \forall\ i,j \in \mathcal{A}\ \textit{tels que}\ (i,j)\in E\ \textit{ou}\ (j,i)\in E \\
& \xi^i \in \{0,1\} \qquad \forall\ i \in \mathcal{A}
\end{aligned}
$$

De plus, si on connaît pour toute activité i une fenêtre de temps $[ES_i, LS_i + p_i]$ durant laquelle i doit être exécutée, le programme $(KS_{(\lambda\mu,t)})$ peut être resseré en supprimant les variables suivantes :

$$\xi^i = 0 \qquad \forall\ i \in \mathcal{A} \mid ES_i > t\ \textit{ou}\ LS_i + p_i \le t \tag{3.4}$$

Preuve. En séparant le problème $(P_{\lambda\mu})$ suivant les groupes de variables $\{x_{lt},\ l \in \mathcal{F}\}$ correspondants à chaque temps t, la valeur optimale $\phi_{\lambda\mu}$ du problème est la somme sur les instants $t \in \mathcal{T}$ des $z_{(\lambda\mu,t)}$, où

$$z_{(\lambda\mu,t)} = \min\{\ \sum_{i\in\mathcal{A}} \nu_{it} \sum_{l\in\mathcal{F}_i} x_{lt} \mid \sum_{l\in\mathcal{F}} x_{lt} \le 1,\ x_{lt} \in \{0,1\}\ \forall\ l \in \mathcal{F}\ \}. \tag{3.5}$$

Pour tout instant t, les solutions réalisables du problème (3.5) sont le vecteur identiquement nul et les vecteurs unités de $\{0,1\}^{\mathcal{F}}$, correspondant chacun à un ensemble admissible $l \in \mathcal{F}$. De plus, le vecteur nul et le vecteur unité associé à l'ensemble admissible vide $l_\emptyset \in \mathcal{F}$ étant tous deux de coût 0 (car $l_\emptyset$ n'appartient à aucun $\mathcal{F}_i$), on peut remplacer, dans la contrainte du programme, l'inégalité par une égalité. Une solution réalisable x de (3.5) correspond donc à un unique ensemble admissible F_l tel que $x_{lt} = 1$ et $x_{l't} = 0$ pour tout autre $l' \in \mathcal{F}$, et son coût est égal à $\sum_{i\in F_l} \nu_{it}$.

On a donc $z_{(\lambda\mu,t)} = \min\{\ \sum_{i\in F_l} \nu_{it} \mid l \in \mathcal{F}\ \}$ et il s'agit de déterminer un seul ensemble admissible l_t solution optimale de ce problème. Par définition des ensembles admissibles et en identifiant chacun d'entre eux par le vecteur $\xi \in \{0,1\}^{\mathcal{A}}$ défini par $\xi_i = 1$ si et seulement si i appartient à l'ensemble admissible, on voit facilement que $z_{(\lambda\mu,t)}$ est égale à la valeur optimale du problème de sac-à-dos $(KS_{(\lambda\mu,t)})$.

Dans la formulation initiale des ensembles admissibles (EA), on peut évidemment fixer à 0 toutes les variables x_{lt} correspondant à un ensemble admissible l constitué d'au moins une activité ne pouvant être en cours d'exécution au temps t. Les contraintes (3.4), utilisées en prétraitement du programme, sont bien valides en ce sens. ■

La première partie $(P_{\lambda\mu})$ du sous-problème lagrangien se décompose donc en T problèmes de sac-à-dos multidimensionnel de petite taille que l'on peut rapidement résoudre à l'optimum par une PSE classique, par exemple. De cette façon, à chaque itération de la résolution du dual lagrangien (DEA), pour des multiplicateurs donnés, au plus T ensembles réalisables sont effectivement calculés.

3.1.2 Ordonnancement de projet avec coûts dépendant des dates de début

La seconde partie (P_λ) du sous-problème lagrangien est résolvable en temps polynômial (pour un T donné). Posant, pour toute activité $i \in V$ et pour tout temps $t \in \mathcal{T}$

$$\omega_{it} = \begin{cases} 0 & \text{si } i = 0 \\ -\lambda_{it} & \text{si } i \in \{1, \dots, n\} \\ t & \text{si } i = n+1 \end{cases}$$

alors (P_λ) s'écrit :

$$(P_\lambda) \quad \min \sum_{i \in V} \sum_{t \in \mathcal{T}} \omega_{it} y_{it} \tag{3.6}$$

sujet à :

$$\sum_{t \in \mathcal{T}} y_{it} = 1 \qquad \forall\, i \in V \tag{2.39}$$

$$\sum_{t \in \mathcal{T}} t(y_{jt} - y_{it}) \geq p_i \qquad \forall\, (i,j) \in E \tag{2.40}$$

$$y_{it} \in \{0,1\} \qquad \forall\, i \in V,\ \forall\, t \in \mathcal{T} \tag{2.47}$$

Ce problème est un problème d'ordonnancement de projet sans contraintes de ressources et avec des coûts dépendant des dates de début des activités. Il a été particulièrement étudié par Möhring, Schulz, Stork et Uetz [Mohring 2001, Möhring 2003] et dans la thèse de Uetz [Uetz 2001]. Nous décrivons en partie ici leurs résultats et comment ils s'appliquent à notre problème (P_λ).

Complexité

L'*ordonnancement de projet avec coûts dépendant des dates de début* est un problème intéressant en lui-même, mais il est aussi un cas particulier de problèmes d'ordonnancement plus généraux. Il a ainsi fait l'objet de plusieurs études, exhibant des résultats de complexité pour des caractéristiques restreintes de la fonction objectif ou du graphe de précédence. Plus spécifiquement, son intérêt pour l'étude du RCPSP vient du fait que ce problème dérive de la relaxation lagrangienne de n'importe quelle formulation linéaire en temps discrétisé du RCPSP, en dualisant les contraintes de ressources. Une telle relaxation lagrangienne a été appliquée au modèle de Pritsker et al. dans [Christofides 1987, Möhring 2003]. Dans le premier article, Christofides, Alvarez-Valdés et Tamarit utilisent alors une PSE pour résoudre ce problème. Pourtant ce problème est de complexité polynômiale dans tous les cas. Différentes preuves de ce résultat ont été apportés dans la littérature.

La manière la plus évidente de le voir consiste à modéliser les contraintes de précédence (2.40) sous la forme désagrégée proposée (et employée d'ailleurs) par Christofides et al.

$$\sum_{\tau=t}^{T} y_{i\tau} + \sum_{\tau=0}^{t+p_i-1} y_{j\tau} \leq 1 \qquad \forall\, (i,j) \in E, \forall\, t \in \mathcal{T} \tag{2.42}$$

On vérifie facilement que la matrice correspondant aux inégalités (2.39) et (2.42) est totalement unimodulaire, et donc qu'une solution de la relaxation continue de ce programme est nécessairement entière.

Cette observation vaut encore si les contraintes de précédence sont remplacées par des contraintes de précédence généralisées $S_j - S_i \geq b_{ij}$. En effet, les inégalités (2.42) sont alors remplacées par les inégalités suivantes de forme identique :

$$\sum_{\tau=t}^{T} y_{i\tau} + \sum_{\tau=0}^{t+b_{ij}-1} y_{j\tau} \leq 1 \qquad \forall\ (i,j) \in V^2, \forall\ t \in \mathcal{T} \tag{3.7}$$

Réduction à un problème de coupe de capacité minimale

Le programme (P_λ) peut donc être résolu optimalement en temps polynômial en résolvant, par l'algorithme des points intérieurs par exemple, la relaxation continue avec contraintes de précédence désagrégées (3.7) sujet à : (2.39), (2.42) et $y_{it} \in \mathbb{R}_+$, $\forall\ i \in V$, $\forall\ t \in \mathcal{T}$. Möhring et al. utilisent un algorithme de résolution alternatif de complexité $O(nkT^2 \log(T))$ où k est le nombre de contraintes de précédence du problème (égal donc à $|E|$ dans le cas des précédences simples ou à n^2 dans le cas des précédences généralisées). Cet algorithme calcule en fait une coupe de capacité minimale dans un graphe orienté et valué positivement, c'est à dire un ensemble d'arcs C associé à deux sommets a et b tels que tout chemin reliant a à b dans le graphe contienne au moins un arc de C. On présente brièvement comment (P_λ) se réduit à un tel problème, les preuves sont exposées dans [Möhring 2003].

Pour traiter du cas général, on suppose que les contraintes de précédence sont généralisées. On suppose aussi que les coefficients de coûts ω_{it} sont tous positifs. En effet, on peut s'y ramener, par exemple, en retranchant aux coûts initiaux ω_{it} la valeur minimale $\Omega = \min_{i,t} \omega_{it}$. Pour obtenir z_λ, il suffit alors, grâce aux contraintes (2.39), d'aditionner $|V|\Omega$ à la valeur optimale du nouveau problème ; la solution optimale ne change pas.

Proposition 4 *[Möhring 2003] Le problème d'ordonnancement de projet à coûts $\omega_{it} \geq 0$ dépendant des dates de début :*

$$\min \sum_{i \in V} \sum_{t \in \mathcal{T}} \omega_{it} y_{it} \tag{3.6}$$

sujet à :

$$\sum_{t \in \mathcal{T}} y_{it} = 1 \qquad \forall\ i \in V \tag{2.39}$$

$$\sum_{\tau=t}^{T} y_{i\tau} + \sum_{\tau=0}^{t+b_{ij}-1} y_{j\tau} \leq 1 \qquad \forall\ (i,j) \in V^2,\ \forall t \in \mathcal{T} \tag{3.7}$$

$$y_{it} \in \{0,1\} \qquad \forall\ i \in V,\ \forall\ t \in \mathcal{T} \tag{2.47}$$

se réduit au problème de détermination d'une coupe C de capacité finie minimale séparant les sommets a et b dans le graphe orienté (V', E') défini par :

- *l'ensemble des sommets $V' = \{v_{i,t} \mid i \in V,\ t \in \{0, \ldots, T+1\}\} \cup \{a, b\}$;*
- *les arcs de E' sont de trois types, définis pour tout $i, j \in V$ et pour tout $t \in \mathcal{T}$: les arcs $(v_{i,t}, v_{i,t+1})$ de capacité ω_{it}, les arcs $(v_{i,t}, v_{j,t+b_{ij}})$ de capacité infinie et les arcs $(a, v_{i,0})$ et $(v_{i,T+1}, b)$ de capacité infine.*

S'il n'existe pas de telle coupe, alors le problème d'ordonnancement est irréalisable, sinon on peut construire à partir de C une autre coupe, de capacité identique, contenant exactement $n+2$ arcs : $(v_{0,t_0}, v_{0,t_0+1}), \ldots, (v_{n+1,t_{(n+1)}}, v_{n+1,t_{(n+1)}+1})$. (Si les coûts ω_{it} sont strictement positifs, la coupe minimale C est nécessairement de cette forme.) La valeur optimale du problème d'ordonnancement

est égale à la capacité de la coupe C et une solution optimale est donnée par :

$$y_{it} = \begin{cases} 1 & si\ t = t_i \\ 0 & sinon. \end{cases}$$

Möhring et al. démontrent en fait un résultat plus fort encore puisqu'ils établissent la correspondance une à une entre les coupes de capacité finie de $n+2$ arcs de ce type et les solutions réalisables du problème d'ordonnancement correspondant.

Comme précédemment, les fenêtres de temps $[ES_i, LS_i + p_i]$ des activités i peuvent aussi être prises en compte pour resserrer la formulation linéaire et donc le problème de coupe minimale. Dans ce cas, les sommets $v_{i,t}$ ne sont plus définis pour les temps $t < ES_i$ et $t > LS_i + 1$ (les variables y_{it} correspondantes ainsi que $y_{i(LS_i+1)}$ sont fixées à 0) et le graphe (V', E') s'obtient comme dans la proposition mais en considérant v_{i,ES_i} à la place de $v_{i,0}$ et v_{i,LS_i+1} à la place de $v_{i,T+1}$.

3.1.3 Saut de dualité

La dualisation que nous proposons ici possède la propriété suivante :

Proposition 5 *Si on considère la formulation linéaire sur les ensembles admissibles avec contraintes de précédence (généralisées ou non) désagrégées, alors la valeur optimale du dual lagrangien, obtenu par dualisation des contraintes de durées (2.43) et des contraintes (2.45) liant les deux types de variables x_{lt} et y_{it}, est égale à la valeur optimale de la relaxation continue.*

Preuve. Suite à la section ci-dessus, puisque la matrice correspondant aux contraintes en les variables y_{it} de non préemption (2.39) et de précédence (3.7) est totalement uni-modulaire, alors il en est de même pour la matrice correspondant aux contraintes linéaires définissant, avec la contrainte d'intégralité, l'ensemble X des solutions réalisables du sous-problème lagrangien :

$$\sum_{l \in \mathcal{F}} x_{lt} \leq 1 \qquad \forall\ t \in \mathcal{T} \tag{2.44}$$

$$\sum_{t \in \mathcal{T}} y_{it} = 1 \qquad \forall\ i \in V \tag{2.39}$$

$$\sum_{\tau=t}^{T} y_{i\tau} + \sum_{\tau=0}^{t+b_{ij}-1} y_{j\tau} \leq 1 \qquad \forall\ (i,j) \in V^2,\ \forall t \in \mathcal{T} \tag{3.7}$$

Ainsi, l'enveloppe convexe de X correspond exactement à l'ensemble des solutions fractionnaires satisfaisant ces trois ensembles de contraintes et, d'après la proposition 1, le dual lagrangien ne peut donc être strictement meilleur que la relaxation continue. ■

Cependant, contrairement à la relaxation continue contenant un nombre exponentiel de variables, le dual lagrangien ne nécessite pas d'énumérer *a priori* tous les ensembles admissibles et, de plus, chaque sous-problème lagrangien peut être résolu plutôt rapidement par la décomposition présentée ci-dessus.

3.2 Relaxation lagrangienne du modèle préemptif

La relaxation lagrangienne précédente peut être, malgré tout, lourde à traiter en pratique. Nous nous sommes donc aussi intéressés à la dualisation analogue, appliquée à une relaxation de la formulation sur les ensembles admissibles. Cette relaxation est similaire à celle proposée par Brucker

et Knust [Brucker 2000], elle-même basée sur la relaxation de Mingozzi et al. [Mingozzi 1998] pour le calcul de $LB2$ (présentée section 2.4.4, p. 63).

Mingozzi et al. ont donc proposé un PL, obtenu à partir de leur formulation en ignorant les contraintes de non-préemption et en relâchant les contraintes de précédence en contraintes de disjonctions, interdisant simplement à deux activités liées par une contrainte de précédence d'appartenir à un même ensemble admissible, autrement dit, d'être en cours d'exécution à un même instant. Ce programme n'est pas résolu tel quel, du fait de sa taille car il contient une variable pour chaque ensemble admissible minimal. Les auteurs proposent d'utiliser une heuristique pour résoudre le problème dual et ainsi fournir une borne inférieure $LB3$ de de la valeur optimale de la relaxation. Théoriquement au moins aussi forte que la borne du chemin critique, $LB3$ est en pratique souvent bien meilleure et rapidement calculable ; elle est utilisée dans plusieurs PSE pour le RCPSP.

Plus récemment, Brucker et Knust ont renforcé cette relaxation en prenant en compte les fenêtres de temps d'exécution des activités. Ce faisant, ils réduisent le nombre de variables du programme et, surtout, en réduisant au préalable les fenêtres de temps par un algorithme de filtrage PPC, ils réintroduisent un peu plus les contraintes de précédence du problème. Pour parer au nombre toujours élevé de variables, ils résolvent le programme par génération de colonnes. Enfin, en considérant seulement la réalisabilité du programme, ils incluent leur procédure dans une approche destructive. La borne inférieure ainsi calculée figure parmi les meilleures de la littérature.

Nous proposons d'utiliser la méthode duale de relaxation lagrangienne à la place de la génération de colonnes, mais pour nous ramener, comme précédement, à une décomposition simple du programme, nous ne considérons pas exactement le même problème de réalisabilité que Brucker et Knust. De façon à replacer les deux techniques l'une par rapport à l'autre, nous revenons brièvement sur la génération de colonnes de Brucker et Knust 3.2.2 avant de présenter la relaxation lagrangienne 3.2.3.

3.2.1 Formulation préemptive

Dans le schéma destructif, pour une valeur T donnée, on commence par calculer des fenêtres de temps $[ES_i, LF_i]$ durant lesquelles les activités sont exécutées, dans tout ordonnancement pour le RCPSP de durée totale inférieure ou égale à T. On cherche alors à prouver que l'ensemble plus général $\mathcal{S}'$ des ordonnancements préemptifs qui vérifient les contraintes de ressources et ces fenêtres de temps, est vide. Dans ce cas, $T+1$ est une borne inférieure du problème initial.

Tenant compte des fenêtre de temps, l'horizon d'ordonnancement (l'intervalle $[0, T[$) est découpé en sous-intervalles $I_s = [t_{s-1}, t_s[$ de longueur $\delta_s = t_s - t_{s-1}$, suivant la suite ordonnée $t_0, \ldots, t_\sigma$ des différentes valeurs ES_i et LF_i pour toute activité $i \in \mathcal{A}$. Par construction, on peut donc considérer, pour chaque intervalle I_s, l'ensemble $\mathcal{A}_s$ des activités pouvant s'exécuter à tout moment dans cet intervalle : $\mathcal{A}_s = \{i \in \mathcal{A} \mid ES_i \leq t_{s-1} < t_s \leq LF_i\}$. Cette décomposition permet enfin de caractériser plus précisément les ensembles admissibles qui peuvent effectivement être actifs, à un instant t donné et donc à tout instant dans l'intervalle I_s contenant t. Les ensembles admissibles de la sorte sont indicés par $l \in \mathcal{F}^s$; autrement dit, pour tout $s \in \{1, \ldots, \sigma\}$, $\mathcal{F}^s = \{l \in \mathcal{F} \mid F_l \subseteq \mathcal{A}_s\}$. Par définition, $\mathcal{A}_s$ et donc $\mathcal{F}^s$ sont non vides quel que soit l'intervalle s (du moins pour le RCPSP standard). À un ordonnancement préemptif de $\mathcal{S}'$, on fait correspondre la durée x_{ls}, avec

$s \in \{1, \ldots, \sigma\}$ et $l \in \mathcal{F}^s$, pendant laquelle l'ensemble admissible l est actif dans l'intervalle I_s. L'ensemble $\mathcal{S}'$ est vide si l'ensemble suivant est vide aussi :

$$\{x_{ls} \in \{0,1\}\, \forall s \in \{1,\ldots,\sigma\}, l \in \mathcal{F}^s \mid \sum_{s=1}^{\sigma} \sum_{l \in \mathcal{F}_i \cap \mathcal{F}^s} x_{ls} \geq p_i,\ \forall i \in \mathcal{A},\ \sum_{l \in \mathcal{F}^s} x_{ls} \leq \delta_s,\ \forall s \in \{1,\ldots,\sigma\}\ \}.$$

En effet, pour tout ordonnancement préemptif, les activités i sont exécutées exactement, donc au moins, pendant p_i unités de temps (premières contraintes), et la somme des durées des ensembles admissibles actifs dans l'intervalle I_s est inférieure ou égale à la longueur de l'intervalle (secondes contraintes). Introduisant des variables artificielles « d'écart » pour les premières contraintes, l'ensemble précédent est vide si et seulement si le programme suivant a une valeur optimale strictement positive.

$$(EAP) \quad \min \sum_{i \in \mathcal{A}} e_i \tag{3.8}$$

sujet à :

$$\sum_{s=1}^{\sigma} \sum_{l \in \mathcal{F}_i \cap \mathcal{F}^s} x_{ls} + e_i \geq p_i \qquad \forall\ i \in \mathcal{A} \tag{3.9}$$

$$\sum_{l \in \mathcal{F}^s} x_{ls} \leq \delta_s \qquad \forall\ s \in \{1,\ldots,\sigma\} \tag{3.10}$$

$$x_{ls} \geq 0 \qquad \forall\ s \in \{1,\ldots,\sigma\},\ l \in \mathcal{F}^s \tag{3.11}$$

$$e_i \geq 0 \qquad \forall\ i \in \mathcal{A} \tag{3.12}$$

Nous traitons ce programme linéaire par relaxation lagrangienne, en dualisant les contraintes de durée des activités (3.9), de sorte que le sous-problème lagrangien se décompose de nouveau en problèmes de sac-à-dos multidimensionnel.

3.2.2 Génération de colonnes

La différence d'avec la procédure de Brucker et Knust [Brucker 2000] est qu'ils traitent un programme linéaire analogue à (EAP), mais avec des variables artificielles $u_s \geq 0$ pour les contraintes (3.10) à la place des variables e_i des contraintes (3.9). Ce programme est résolu par génération de colonnes. À chaque itération de la procédure, une solution duale $(y_1, \ldots, y_n, w_1, \ldots, w_\sigma)$ du programme restreint à un sous-ensemble de colonnes est considérée pour déterminer des colonnes améliorantes, à savoir, pour tout intervalle I_s, des ensembles admissibles $l \in \mathcal{F}^s$ tels que $\sum_{i \in F_l} y_i - w_s > 0$. Brucker et Knust proposent alors une recherche arborescente spécifique permettant d'identifier de tels ensembles admissibles ou de prouver qu'il n'en existe pas.

Bien que notre procédure ne soit pas exactement duale à celle-ci, nous nous ramenons de même, à chaque itération et dans tout intervalle I_s, à déterminer un ensemble admissible qui vérifie ou plutôt qui optimise un certain critère.

3.2.3 Relaxation lagrangienne

Avec les multiplicateurs de lagrange $\lambda \in \mathbb{R}^n_+$, le sous-problème obtenu en dualisant les contraintes (3.9) dans (EAP) s'écrit :

$$(EAP_\lambda) \quad \min \sum_{i\in\mathcal{A}} \lambda_i (p_i - \sum_{s=1}^{\sigma} \sum_{l\in\mathcal{F}_i\cap\mathcal{F}^s} x_{ls}) + \sum_{i\in\mathcal{A}} (1-\lambda_i) e_i \tag{3.13}$$

sujet à :

$$\sum_{l\in\mathcal{F}^s} x_{ls} \leq \delta_s \qquad \forall\ s \in \{1,\dots,\sigma\} \tag{3.14}$$

$$x_{ls} \geq 0 \qquad \forall\ s \in \{1,\dots,\sigma\},\ l \in \mathcal{F}^s \tag{3.15}$$

$$e_i \geq 0 \qquad \forall\ i \in \mathcal{A} \tag{3.16}$$

Soit z_λ la valeur optimale de (EAP_λ) et $f_\lambda(x,e)$ la fonction objectif. Puisque $l \in \mathcal{F}_i$ si et seulement si $i \in F_l$, cette dernière s'écrit aussi :

$$f_\lambda(x,e) = \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} \sum_{l\in\mathcal{F}^s} (\sum_{i\in F_l} \lambda_i) x_{ls} + \sum_{i\in\mathcal{A}} (1-\lambda_i) e_i.$$

La résolution du dual lagrangien consiste à déterminer le vecteur λ maximisant z_λ. L'identification des λ pour lesquels (EAP_λ) est borné permet de restreindre l'étude à ces seuls vecteurs et de plus, dans ce cas, le programme (EAP_λ) se simplifie.

Proposition 6 *Une borne inférieure de (EAP) est donnée par la solution optimale de son dual lagrangien :*

$$(DEAP) \qquad \max_{\lambda\in[0,1]^n} \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} \delta_s \sum_{i\in F^\lambda_{l_s}} \lambda_i.$$

où, pour tout $0 \leq \lambda \leq 1$ et pour tout intervalle I_s, l^λ_s est un ensemble admissible qui maximise $\sum_{i\in F_l} \lambda_i$ parmi $l \in \mathcal{F}^s$.

Preuve. On commence par montrer que la valeur optimale de (EAP_λ), z_λ, est égale à $-\infty$ si $\lambda \not\leq 1$. On choisit donc un $\lambda \geq 0$ tel que $\lambda_j > 1$ pour au moins une activité j. Une solution réalisable du programme (EAP_λ) correspondant consiste à placer toutes les coordonnées à 0, exceptée e_j affectée à une valeur positive M arbitrairement grande. Le coût d'une telle solution est alors un nombre négatif dont la valeur décroît proportionnellement à $-M$ et le problème est donc non-borné :

$$z_\lambda \leq \sum_{i\in\mathcal{A}} \lambda_i p_i + M(1-\lambda_j) \to -\infty \ \text{ quand } \ M \to +\infty.$$

Si, au contraire, $\lambda_i \leq 1$ pour toute activité $i \in \mathcal{A}$, alors il existe une solution optimale de (EAP_λ) telle que $e = 0$ car, pour toute solution réalisable (x,e), $(x,0)$ est aussi réalisable et on a alors :

$$f_\lambda(x,e) \geq \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} \sum_{l\in\mathcal{F}^s} (\sum_{i\in F_l} \lambda_i) x_{ls} = f_\lambda(x,0).$$

Ainsi, (EAP_λ) est borné ($\sum_{i\in\mathcal{A}} \lambda_i p_i \geq z_\lambda \geq \sum_{i\in\mathcal{A}} \lambda_i (p_i - \sum_{s=1}^{\sigma} \delta_s)$). On peut supprimer les variables e du programme et remplacer la fonction objectif par $f_\lambda(x,0)$.

Soit alors $(l^\lambda_0,\dots,l^\lambda_s)$ défini dans la proposition. La solution x^λ définie par $x^\lambda_{ls} = \delta_s$ si $l = l^s$ et $x^\lambda_{ls} = 0$ sinon, est évidemment réalisable pour (EAP_λ), et son coût est égal à $\sum_{i\in\mathcal{A}} \lambda_i p_i -$

$\sum_{s=1}^{\sigma}(\sum_{i\in F_{l_{\lambda}^s}} \lambda_i)\delta_s$.

Soit x une autre solution réalisable, alors :

$$\begin{aligned}
f_\lambda(x,0) &= \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} \sum_{l\in\mathcal{F}^s} (\sum_{i\in F_l} \lambda_i) x_{ls} \\
&\geq \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} \sum_{l\in\mathcal{F}^s} (\sum_{i\in F_{l_s^\lambda}} \lambda_i) x_{ls} \\
&\geq \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} (\sum_{i\in F_{l_s^\lambda}} \lambda_i)(\sum_{l\in\mathcal{F}^s} x_{ls}) \\
&\geq \sum_{i\in\mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} (\sum_{i\in F_{l_s^\lambda}} \lambda_i)\delta_s \\
&\geq f_\lambda(x^\lambda, 0)
\end{aligned}$$

x^λ est donc bien une solution optimale de (EAP_λ) et son coût est bien égal au terme dans la maximisation de $(DEAP)$. ■

Une solution du sous-problème lagrangien (EAP_λ) consiste donc à ordonnancer, dans chaque intervalle I_s, un unique ensemble admissible l_s^λ qui minimise la somme des coûts $-\lambda_i$ associé à chaque activité i de l'ensemble. Pour déterminer l_s^λ, il suffit de résoudre un problème de sac-à-dos multidimensionnel identique à celui de la proposition 3, p. 69, avec les coûts $\nu_i = -\lambda_i$ et sur un nombre réduit de variables puisque, dû aux fenêtres de temps, on peut fixer $\xi_i = 0,\ \forall i \notin \mathcal{A}^s$.

3.3 Détails d'implémentation

Actuellement, nous avons implémenté la borne destructive basée sur la formulation préemptive.

3.3.1 Bornes constructives et destructives

Nous avons présenté chacune des deux relaxations lagrangiennes dans une approche différente : la première pour la formulation PLNE complète du RCPSP dans une approche constructive, la seconde sur la relaxation préemptive dans une approche destructive. Les méthodes sont cependant interchangeables car on peut calculer une borne inférieure destructive à partir de la formulation complète en considérant à la place de la fonction objectif $\sum_{t=0}^{T} t y_{(n+1)t}$, la somme des variables d'écarts des contraintes sur les durées (en les relâchant par une inégalité), et on peut calculer une borne inférieure constructive à partir de la relaxation préemptive en considérant le problème d'optimisation avec le critère $\min \sum_{s=1}^{\sigma} \sum_{l\in\mathcal{F}^s} x_{ls}$ à la place du problème de réalisabilité.

Dans l'approche constructive, la borne inférieure est simplement la valeur optimale du dual lagrangien, ou du moins, la plus grande valeur optimale d'un sous-problème lagrangien précédemment calculé, si la résolution du dual n'est pas menée à son terme (un temps et un nombre d'itérations maximaux sont accordés à la résolution du dual). Un algorithme de filtrage PPC est utilisé en prétraitement du programme linéaire pour resserrer les fenêtres de temps des activités.

Dans l'approche destructive, il s'agit de résoudre, pour différentes valeurs de T, une succession de problèmes de satisfiabilité, consistant à déterminer si un ensemble E_T, incluant l'ensemble $\mathcal{S}_T$ des ordonnancements réalisables du RCPSP de durée totale inférieure ou égale à T, est vide ou non. Les valeurs T sont ici testées par dichotomie sur l'intervalle $[LB, UB]$ où $LB = 0$ et UB est une borne supérieure du RCPSP, calculée pour nos expérimentations par l'heuristique rapide de

Baar et al. [Baar 1998]. Partant de $T = \lfloor \frac{UB-LB}{2} \rfloor$, si on prouve que l'ensemble E_T est vide, on poursuit la recherche sur l'intervalle $[T+1, UB]$, sinon on étudie l'intervalle $[0, T]$. À tout moment, la valeur minimale de l'intervalle de recherche est bien une borne inférieure du problème initial. On réitère donc le processus, jusqu'à ce que l'intervalle de recherche ne contienne plus qu'une valeur qui est alors une borne inférieure destructive du RCPSP.

Le problème de satisfiabilité correspondant à une valeur T donnée est tout d'abord traité par PPC. Il s'agit de la relaxation du RCPSP induite par l'algorithme de filtrage présenté ci-dessous. Si l'irréalisabilité est prouvée, alors $T+1$ est une borne inférieure et on teste une valeur de T supérieure. Sinon, on considère le dual lagrangien du modèle linéaire correspondant à la formulation décisionnelle complète ou préemptive, qui surcontraint, dans les deux cas, la relaxation PPC. Pour cela, on résoud itérativement des sous-problèmes lagrangiens pour différentes valeurs des multiplicateurs, suivant la méthode du sous-gradient ou de la génération de coupes. En particulier, on s'arrête dès que la valeur optimale d'un sous-problème est strictement positive car l'irréalisabilité est alors prouvée pour la valeur T. Sinon, le processus itératif continue jusqu'à ce que, soit le dual est résolu, soit il atteint un nombre limite d'itérations ou un temps limite d'exécution ; L'irréalisabilité n'est donc pas prouvée, on teste alors une nouvelle valeur de T strictement inférieure.

3.3.2 Algorithme de filtrage et prétraitement

Pour la relaxation du RCPSP par PPC, nous avons à disposition toutes les règles de consistance et de propagation, qui sont présentées au chapitre précédent, section 2.2. Dans nos expérimentations, nous avons utilisé un algorithme de filtrage proche de celui que Brucker et Knust ont proposé pour le calcul de leur borne inférieure. De fait, cet algorithme est plutôt rapide et puissant sur les instances « hautement disjonctives » KSD. Les règles d'ajustement incluent la consistance de chemin aux bornes sur les contraintes de précédence par l'algorithme de `floyd-warshall`, les règles sur les triplets symétriques, la règle de sélection immédiate sur les paires disjonctives, et les règles du edge-finding sur des ensembles disjonctifs. Pour le calcul des ensembles disjonctifs maximaux, nous avons implémenté les heuristiques en $\mathcal{O}(n^2)$ de Brucker et al. et de Baptiste et Le Pape, et pour l'application du edge-finding primal/dual à un EDM, l'algorithme de complexité $\mathcal{O}(|EDM|^2)$ de [Nuijten 1994]. Nous appliquons aussi la règle des bornes inférieures basées sur les EDM. On fera référence à cet algorithme par `LCP` (propagation de contraintes *locale*, par opposition à la technique *globale* du shaving).

En plus de l'ajustement des fenêtres de temps, l'algorithme détecte un ensemble D de paires d'activités en disjonction. Ce résultat est aussi utilisé en prétraitement du programme linéaire en interdisant à deux activités en disjonction d'appartenir à un même ensemble admissible. Il suffit pour cela de considérer, dans les problèmes de sac-à-dos, une contrainte additionnelle $\xi_i + \xi_j \leq 1$ pour tout $(i,j) \in D$. Enfin, comme l'algorithme de filtrage est basé sur la représentation des contraintes de domaine par la matrice des distances (b_{ij}), nous considérons, dans la partie programmation linéaire, les contraintes de précédence généralisées ainsi inférées par PPC, qui incluent et renforcent les précédences simples du problème. Ceci est pris en compte dans le calcul de la coupe de capacité minimimale comme indiqué section 3.1.2, ainsi que dans les problèmes de sac-à-dos avec des contraintes $\xi_i + \xi_j \leq 1$ si $b_{ij} \geq p_i$ ou si $b_{ji} \geq p_j$. De cette façon, les méthodes proposées ici s'appliquent à l'identique à la classe plus générale des RCPSP avec délais.

Enfin, nous avons aussi expérimenté la technique de shaving sur les séquencements que nous proposons pour le RCPSP (voir section 2.2.6). On verra dans les résultats expérimentaux que le surcoût de temps de calcul du filtrage engendré par le shaving est compensé par une amélioration substantielle de la qualité de la borne et même par une accélération de la procédure destructive.

L'algorithme de filtrage complet consiste donc à appliquer dans un premier temps l'algorithme LCP de filtrage local puis, pour toute paire d'activité $\{i, j\}$ dont le séquencement relatif n'est pas encore fixé, de tester séparément la consistance de $i \rightarrow j$, $j \rightarrow i$ et $i \| j$, par LCP. Si des déductions sont inférées pour une paire $\{i, j\}$, la matrice des distances B et l'ensemble des disjonctions D sont mis à jour comme indiqué section 2.2.6. Le processus de shaving est itéré tant que le problème est consistant et tant que des déductions sur B et D sont faites. Dans la pratique, le shaving est rarement itéré plus de 4 ou 5 fois. Pour l'approche constructive, la résolution s'arrête aussi dès que la borne inférieure $b_{0(n+1)}$ de la PPC atteint la borne supérieure réalisable T.

3.3.3 Résolution du dual lagrangien

La résolution du dual lagrangien se fait de manière itérative, en traitant à chaque itération, un sous-problème lagrangien associé à une valeur des multiplicateurs λ. Avec la décomposition des sous-problèmes, nous sommes amenés à résoudre différentes instances de problèmes de sac-à-dos multidimensionnel et, pour la formulation complète, d'un problème de coupe de capacité minimale. Les problèmes de sac-à-dos sont résolus à l'optimum par le branch and bound par défaut de ILOG CPLEX.

Nous avons suivi deux types de méthode itérative de résolution du dual lagrangien : la génération de coupes et l'algorithme du sous-gradient. Pour expliquer le principe des deux méthodes, on prend l'exemple de la résolution du dual lagrangien ($DEAP$) du problème préemptif dans l'approche destructive.

Génération de coupes

Pour la génération de coupes, on considère ($DEAP$) sous la formulation suivante qui procède trivialement de la proposition 6 :

$$\max u \tag{3.17}$$

sujet à :

$$u \leq \sum_{i \in \mathcal{A}} \lambda_i p_i - \sum_{s=1}^{\sigma} \delta_s \Big(\sum_{i \in F_{l_s}} \lambda_i \Big) \qquad \forall\, l_s \in \mathcal{F}^s,\ \forall s \in \{1, \ldots, \sigma\} \tag{3.18}$$

$$\lambda \in [0,1]^n,\ u \in \mathbb{R} \tag{3.19}$$

Ce programme est résolu itérativement, en générant et ajoutant les contraintes une à une. À chaque itération, on récupère la solution optimale (λ^*, u^*) du programme comptant seulement un sous-ensemble de contraintes et on résoud le sous-problème lagrangien associé à λ^*. Dans l'approche destructive, on teste dans un premier temps si la valeur optimale z_{λ^*} du sous-problème est strictement positive, prouvant dans ce cas l'infaisabilité. Autrement, deux cas sont à considérer selon que $u^* \leq z_{\lambda^*}$ ou $u^* > z_{\lambda^*}$. Dans le premier cas, le dual lagrangien est complétement résolu et l'optimum est atteint pour le multiplicateur λ^* et est égal à z_{λ^*}. En effet, pour toute valeur de λ, soit u la valeur mimimale du sous-problème lagrangien correspondant, sur les seuls ensembles admissibles déjà définis dans le programme courant. Alors, u est un majorant de z_λ et (λ, u) est une solution réalisable du programme courant, et donc, $z_\lambda \leq u \leq u^* \leq z_{\lambda^*}$ (en particulier, $u^* = z_{\lambda^*}$). Dans le second cas, $u^* > z_{\lambda^*}$, on coupe la solution du programme courant en ajoutant la contrainte correspondant à la solution optimale de (EAP_{λ^*}) avec les ensembles admissibles $l_1^{\lambda^*}, \ldots, l_\sigma^{\lambda^*}$, puis on réitère.

Algorithme du sous-gradient

Pour la résolution du dual ($DEAP$) par l'algorithme du sous-gradient, nous avons utilisé une construction standard de la suite des vecteurs des multiplicateurs de Lagrange assurant la convergence, en posant à chaque itération k, $\lambda^{k+1} = \lambda^k + \theta^k \gamma^k$, où γ^k est un sous-gradient de la fonction $\lambda \mapsto z_\lambda$ en λ^k, et θ^k est le pas de déplacement. Comme on peut restreindre l'étude des multiplicateurs aux vecteurs compris entre 0 et 1, on pose en fait $\lambda_i^k = \max(0, \min(1, \lambda_i^k))$.

Soient $F^1, \ldots, F^\sigma$, les ensembles admissibles actifs dans la solution optimale du sous-problème lagrangien associé à λ^k. On définit alors facilement un sous-gradient de l'itération k avec $g_i^k = p_i - \sum_{s=1}^{\sigma} \sum_{i \in F^s} \delta_s$, pour toute activité $i \in \mathcal{A}$. Le pas de déplacement est donné par $\theta^k = \theta(\bar{z} - z_{\lambda^k})/\|g^k\|^2$, avec $\bar{z}$ une borne supérieure de ($DEAP$) et θ un paramètre strictement compris entre 0 et 2 et qui est ajusté en fonction de l'amélioration de la borne inférieure.

3.4 Résultats expérimentaux

Nos expériences ont été menées sur un Pentium III 800MHz avec 384Mb de RAM sous Debian/Linux et g++ 2.95.4. Les programmes ont été écrits en C++ en utilisant ILOG CPLEX 7.0 pour la résolution des programmes linéaires.

La table 3.1 donne les résultats obtenus par relaxation lagrangienne sur les instances KSD à 30 activités, dont les solutions optimales sont toutes connues. Nous avons limité nos expériences au modèle préemptif des ensembles admissibles suivant le schéma destructif. Tout programme linéaire est prétraité par application de l'algorithme de shaving complet, c'est-à-dire appliqué à toute paire d'activités non séquencées.

Dans les deux premières colonnes, les méthodes `GC/heur` et `GC/opt` correspondent à la résolution du dual lagrangien par génération de coupes mais avec des bornes supérieures de départ différentes : la valeur de la solution obtenue par la méthode tabou de Baar et al. [Baar 1998] (`heur`) et la valeur de la solution optimale (`opt`). La colonne `SG/opt` correspond à la résolution du dual lagrangien par la méthode du sous gradient, avec une borne supérieure égale à la valeur optimale. La colonne `BK` donne pour comparaison les résultats de la méthode de génération de colonnes de Brucker et Knust.

Pour chacune de ces bornes, nous donnons l'écart moyen et maximal par rapport à l'optimum, le temps de calcul moyen et maximal en secondes, le nombre d'instances vérifiées et enfin le nombre d'instances où la borne inférieure a été déterminée par la programmation linéaire.

TAB. 3.1 – Borne destructive de relaxation lagrangienne sur le modèle préemptif des ensembles admissibles pour les instances KSD 30

#activités		GC/heur	GC/opt	SG/opt	BK
	moy Δ opt	0,8%	0,8%	0,65%	1,5%
	max	18,2%	18,2%	14,13%	11,1%
KSD30	moy CPU(1)	10,6	6,7	27,9	**0,4**
(480)	max	289,8	301,7	827,7	4,3
	#LB=opt	393	393	400	318
	#LB par LP	28	28		92

(1) la borne `BK` est calculée sur une station Sun Ultra 2 à 167MHz.

Les bornes basées sur la relaxation lagrangienne obtiennent de très bons résultats sur les instances à 30 tâches. Elles sont toutes en moyenne à moins de 1% de la solution optimale et surpassent

en cela la borne destructive basée sur la génération de colonnes de Brucker et Knust. Elles dominent également cette borne pour le nombre d'instances vérifiées. En contrepartie, elles sont moins bonnes dans le pire des cas et surtout elles ont des temps de calcul beaucoup plus élevés. Un des intérêts fondamentaux de l'utilisation de bornes destructives est également illustré. La borne obtenue par résolution du dual lagrangien au moyen de la génération de coupes est indépendante de la borne supérieure (voir colonnes `GC/heur` et `GC/opt`). Seul le temps de calcul augmente en fonction de la borne supérieure. Les meilleurs résultats sont obtenus par la méthode du sous gradient avec 400 instances vérifiées sur 480 mais les temps de calcul sont alors très élevés. Il est toutefois possible qu'un meilleur paramétrage de l'algorithme permette d'accélérer la recherche.

La table 3.2 donne les résultats sur les instances KSD à 60 tâches pour la méthode basée sur la génération de coupes avec une borne supérieure égale à la meilleure valeur connue. La colonnne `CCP+GC` désigne les résultats avec l'algorithme de shaving complet. La colonne `LCP+GC` désigne les résultats sans l'algorithme de shaving, l' algorithme de filtrage local `LCP` étant appliqué seul. Nous comparons nos résultats avec la meilleure borne inférieure connue sur ce groupe d'instances et proposée par Brucker et Knust (colonne `BK`), avec la relaxation lagrangienne de Möhring et al. (colonne `MSSU`) ainsi qu'avec la meilleure borne inférieure connue pour chacune des instances (colonne `LB*`). Suivant le schéma expérimental de Möhring et al., nous donnons les résultats sur les 360 instances dont le paramètre RS (Resource Strength) est strictement inférieur à 1. En effet, toutes les instances telles que RS=1 sont triviales.

Pour chacune des bornes, nous donnons l'écart moyen par rapport à la borne inférieure basique du chemin critique, le temps de calcul moyen et maximal, le nombre d'optima trouvés, le nombre de fois ou la borne proposée dépasse la meilleure borne connue précédemment, le nombre de fois ou la borne est déterminée par la programmation linéaire, le nombre d'instances nouvellement fermées par la borne proposée et enfin le nombre d'itérations de l'algorithme (correspondant au nombre total de coupes générées par notre méthode et le nombre d'itérations du sous-gradient pour la méthode de Möhring).

TAB. 3.2 – Borne destructive de relaxation lagrangienne sur le modèle préemptif des ensembles admissibles pour les instances KSD 60

#activités		CCP+GC	LCP+GC	MSSU	BK	LB*
	moy Δ LB0	10,6%	9,9%	7,46%	10,34%	10,56%
	moy CPU(1)	462	410	2,4	5	-
	max	7880	3342	32	3720	-
KSD60	#LB=UB	241	227	ND	221	236
(360)	#LB>LB*	43	18	-	-	-
	#LB par LP	47	57	-	92	-
	#New opt	10	2	-	-	-
	moy #it	144	190	49	-	-
	max	1982	2184	172	-	-

(1) `BK` est calculé sur une station Sun Ultra 2 à 167MHz. `MSSU` est calculé sur une station Sun Ultra 2 à 200 Mhz

Les résultats confirment la qualité des nouvelles bornes proposées basées sur la relaxation lagrangienne car sur les 360 instances, nous déterminons 43 nouvelles bornes inférieures et nous fermons 10 nouvelles instances (j601_7, j603_8, j6021_2, j6021_3, j6033_6, j6037_3, j6037_5, j6037_8, j6037_10 et j6038_8). Avec le shaving, nous obtenons ainsi un meilleur écart par rapport à la borne inférieure du chemin critique et un plus grand nombre d'optima trouvés que les meilleures bornes connues. Notre temps de calcul est par contre beaucoup plus élevé que celui des bornes BK et MSSU.

La dernière ligne montre que notre algorithme comporte beaucoup plus d'itérations que celui de Möhring et al, ce qui est dû en partie au schéma destructif, provoquant la résolution de davantage de programmes linéaires. Cela nous montre encore la nécessité de développer un algorithme de sous-gradient aussi peu coûteux que celui de Möhring et al. En supprimant l'algorithme couteux de shaving, le temps de calcul diminue faiblement alors que la qualité des résultats diminue fortement. La PPC étant moins forte, la programmation linéaire se déclenche plus souvent, le nombre de coupes générées augmentant d'autant. Notre borne reste alors meilleure que celle de Möhring et al. mais devient inférieure à la meilleure borne connue. Nous reviendrons au chapitre 4 sur l'impact des algorithmes de propagation de contraintes sur la qualité de la borne.

Nous proposons maintenant une analyse plus profonde de ces résultats en les évaluant en fonction des caractéristiques NC (la densité du graphe du projet), RF (le nombre moyen de ressources requises par chaque activité) et RS (indice sur la capacité des ressources) des instances KSD (voir section 2.4.1). Parmi les instances de KSD30, Mingozzi et al. [Mingozzi 1998] ont identifié plusieurs séries que la PSE de [Demeulemeester 1992] ne parvenait pas à résoudre. Dans la table 3.3, nous donnons, pour chacune de ces séries, la déviation moyenne en pourcentage de notre borne (relaxation lagrangienne avec sous-gradient) par rapport à l'optimum, comparée à la borne du chemin critique `LB0` et à la borne `LB1` de [Mingozzi 1998].

TAB. 3.3 – Relaxation lagrangienne : résultats comparés sur les séries « dures » de KSD30

série	NC	RF	RS	`LB0`	`LB1`	`SG`
21	1,8	0,50	0,20	25,67	13,55	**0,00**
25	1,8	0,75	0,20	36,68	9,75	**1,07**
29	1,8	1,00	0,20	41,56	11,66	**4,55**
30	1,8	1,00	0,50	8,88	5,34	**3,20**
31	1,8	1,00	0,70	2,66	**0,74**	0,95
41	2,1	0,75	0,20	37,24	8,21	**0,11**
45	2,1	1,00	0,20	36,52	8,26	**0,36**

Excepté pour la série 31, `SG` est meilleure que LB1 pour chacune de ces instances. En fait, il semble que les séries identifiées comme difficiles par Mingozzi et al. ne correspondent pas à celles pour lesquelles notre méthode est moins efficace. Par exemple, `SG` résoud toutes les instances de la série 21 mais est éloignée en moyenne de $6,25\%$ de l'optimum pour la série 13. La borne est en réalité très bonne pour les instances correspondant à un facteur de ressources bas ($RF = 0,25$ ou $0,5$), comme pour la série 21, mais beaucoup moins performante si RF est élevé et RS faible, comme pour les séries 13 et 29 ($RF = 1$ et $RS = 0,2$).

Pour mieux apprécier le comportement de notre approche en fonction de ces caractéristiques, nous donnons dans la table 3.4, les résultats expérimentaux sur les instances de KSD60 correspondant à $RS = 0,2$. On rappelle que ce sont les instances considérées souvent comme les plus difficiles [Brucker 2000] (parmi les instances KSD) et qui possèdent peu de ressources disponibles et donc de nombreuses disjonctions. Ici, les séries sont regroupées, pour chaque ligne, en fonction de leur facteur RF. Les colonnes 4, 5, et 6 donnent la déviation moyenne par rapport à la meilleure borne supérieure connue pour, respectivement, les bornes `LB0`, `BK` et `GC` (relaxation lagrangienne par génération de coupes). La colonne 7 reporte le nombre d'instances pour lesquelles `GC` améliore la meilleure borne connue, parmi celles qui sont encore ouvertes (colonne 8).

Il apparaît clairement que le groupe des instances difficiles n'est pas homogène et que la qualité de notre borne dépend aussi de RF. On détermine par exemple un grand nombre de nouvelles meilleures bornes inférieures pour les instances pour des RF faibles (3 sur 3 instances ouvertes

TAB. 3.4 – Relaxation lagrangienne : résultats comparés sur les instances KSD60 avec RS=0.2

	séries	RS	RF	LB0	BK	GC	#GC>LB*	#UB>LB*
	1,17,33	0,20	0,25	8,82%	0,99%	**0,08%**	**3**	3
60	5,21,37	0,20	0,50	23,40%	8,23%	**3,17%**	**27**	30
	9,25,41	0,20	0,75	31,51%	**9,38%**	6,17%	7	30
	13,29,45	0,20	1,00	39,90%	**7,36%**	10,60%	0	30

avec $RF = 0,25$ et 27 sur 30 pour $RF = 0,5$), mais la borne est bien moins performante pour les instances avec $RF = 1$ ou $RF = 0,75$, où les activités nécessitent en moyenne toutes ou les trois-quarts des ressources pour s'exécuter. La propagation de contraintes, qui est cruciale dans notre schéma, s'avère moins efficace dans ce cas.

Chapitre 4

Calcul de Bornes Inférieures par Génération de Coupes

Nous présentons dans ce chapitre, un autre type de bornes inférieures pour le RCPSP. La méthode générale est de nouveau basée à la fois sur la programmation par contraintes et sur la programmation linéaire, mais ici elle applique un degré supérieur d'intégration entre les deux solveurs. En effet, nous proposons d'utiliser les méthodes ou les résultats de la PPC pour définir de nouvelles inégalités valides pour différentes formulations linéaires du RCPSP. En ce sens, les coupes générées peuvent être assimilées à des linéarisations de la contrainte globale `cumulative`*. Nous calculons de la sorte différentes bornes inférieures, dans une approche constructive ou destructive, et pour trois formulations linéaires du RCPSP : le modèle en temps continu, le modèle classique en temps discretisé et le modèle préemptif basé sur les ensembles admissibles. Dans le premier modèle issu de la formulation disjonctive de Balas, le prétraitement et les coupes, tous deux basés sur la PPC, sont primordiaux pour compenser la relaxation complète des contraintes d'intégralité et des contraintes de ressources. Nous définissons ainsi de nouvelles inégalités, générées par lifting et exploitant les résultats du shaving sur les séquencements, qui est appliqué en prétraitement du programme linéaire. À la suite des linéarisations de Applegate et Cook [Applegate 1991] pour le problème du job-shop, nous générons aussi des coupes reprenant le principe du edge-finding et qui s'appliquent directement aux ensembles disjonctifs précalculés par PPC. Nous identifions de la même façon des coupes basées sur le shaving et les ensembles disjonctifs pour la relaxation continue du modèle en temps discretisé. L'étude de ces deux modèles est basée sur [Demassey 2002, Demassey 2003]. Pour le troisième modèle, il s'agit d'une association au travail de Philippe Baptiste sur l'amélioration de la borne de Brucker et Knust [Brucker 2000] par génération de coupes basées sur le raisonnement énergétique et la prise en considération des précédences et de la non-préemptivité. Les résultats présentés ici sont issus de [Baptiste 2004].*

Nous décrivons par la suite les inégalités valides que nous avons développées pour chacun des trois modèles linéaires : modèle en temps continu, modèle en temps discretisé et modèle préemptif basé sur les ensembles admissibles. Nous analysons ensuite nos résultats expérimentaux pour chacune des bornes inférieures ainsi calculées. Comme précédemment les bornes sont calculées par une approche constructive ou destructive, on se réferera donc au chapitre précédent pour les méthodes générales de calcul. Pour les deux premiers modèles, seul change le programme linéaire et son mode de résolution, effectuée, non plus par relaxation lagrangienne, mais par relaxation continue et génération de coupes. Pour le troisième modèle, la borne est calculée de façon destructive par génération de colonnes (voir section 3.2.2) mais avec un algorithme de filtrage différent.

On reprend les notations utilisées depuis le début de ce mémoire, en particulier :

- T l'horizon : à tout moment, on ne considère que des ordonnancements de durée inférieure ou égale à T ;
- b_{ij} une distance minimale, éventuellement négative, entre la date de début S_i de l'activité i et la date de début S_j de l'activité j. La matrice des distances B est calculée à partir de T ($b_{(n+1)0} = -T$) par la PPC. Elle est fermée transitivement ($b_{ij} \geq b_{il} + b_{lj}$) et supposée consistante ($b_{ij} + b_{ji} \geq 0$). La fenêtre d'exécution de l'activité i est ainsi définie par $ES_i = b_{0i}$ et $LF_i = p_i - b_{i0}$;
- D est un ensemble de paires d'activités en disjonctions $i - j$, et C est une clique d'activités ou « ensemble disjonctif » composé d'activités réelles toutes en disjonction deux à deux ;
- $B^{i\sim j}$ et $D^{i\sim j}$ sont la matrice des distances et l'ensemble des disjonctions retournés par le shaving après avoir imposé le séquencement $i \sim j$, avec $\sim \in \{\rightarrow, \|\}$ (voir section 2.2.6). Par exemple, si le séquencement relatif $i \rightarrow j$ n'est ni fixé ($b_{ij} < p_i$), ni interdit ($b_{ji} \leq -p_i$) dans la relaxation PPC, alors la distance $b_{hl}^{i\rightarrow j}$ est au moins aussi grande que b_{hl} et correspond à une borne inférieure de $S_l - S_h$ pour tout ordonnancement réalisable S dans lequel l'activité i précède l'activité j.

Comme nous avons utilisé cette technique pour la génération de certaines inégalités valides, nous revenons, pour commencer, sur la procédure générique de *lifting*.

4.1 Génération d'inégalités valides par lifting

Le lifting est une technique plutôt intuitive pour inférer des inégalités valides pour l'ensemble S des solutions d'un programme linéaire à partir d'une inégalité vérifiée par seulement une partie des solutions, $S^0 = S \cap \{x = 0\}$ ou $S^1 = S \cap \{x = 1\}$, où x est une variable binaire du programme. Elle est ainsi d'un usage assez fréquent et, en particulier pour la linéarisation de modèles disjonctifs, elle permet de déterminer, du moins théoriquement, les valeurs *grand M* les plus serrées.

Pour illustrer cette technique, on prend l'exemple d'un programme linéaire sur $x = (x_1, \ldots, x_n)$ dans $\mathbb{R}^n$, la première variable x_1 étant une variable binaire sur laquelle est effectuée le lifting :

Proposition 7 *Soit*

$$\sum_{i=2}^{n} \pi_i x_i \leq \pi_0 \tag{4.1}$$

une inégalité valide pour S^0. Si $S^1 = \emptyset$ alors $x_1 \leq 0$ est valide pour S, sinon

$$\alpha x_1 + \sum_{i=2}^{n} \pi_i x_i \leq \pi_0 \tag{4.2}$$

est une inégalité valide pour S pour tout $\alpha \leq \pi_0 - \zeta^0$, où $\zeta^0 = \max\{\sum_{i=2}^{n} \pi_i x_i | x \in S^1\}$. De plus, si

$\alpha = \pi_0 - \zeta^0$ et si (4.1) définit une facette de S^0, alors (4.2) définit une facette de S.

Symétriquement, si (4.1) est maintenant une inégalité valide pour S^1, alors : Si $S^0 = \emptyset$ alors $x_1 \geq 1$ est valide pour S, sinon

$$\beta(1 - x_1) + \sum_{i=2}^{n} \pi_i x_i \leq \pi_0 \tag{4.3}$$

est une inégalité valide pour S pour tout $\beta \leq \pi_0 - \zeta^1$, où $\zeta^1 = \max\{\sum_{i=2}^{n} \pi_i x_i | x \in S^0\}$. De plus, si $\beta = \pi_0 - \zeta^1$ et si (4.1) définit une facette de S^1, alors (4.3) définit une facette de S.

La difficulté repose évidemment sur le calcul de ζ qui correspond au meilleur coefficient possible pour la variable liftée. Cependant, il est souvent facile de trouver une approximation α ou β, suffisament bonne pour que l'inégalité liftée coupe effectivement la solution fractionnaire que l'on souhaite séparer.

Le schéma 4.1 présente un exemple pour $n = 2$ où la facette $\{(0, 2), (1, 4)\}$ de S est déduite de la facette $\{(0, 2)\}$ de S^0, par lifting sur la variable x_1.

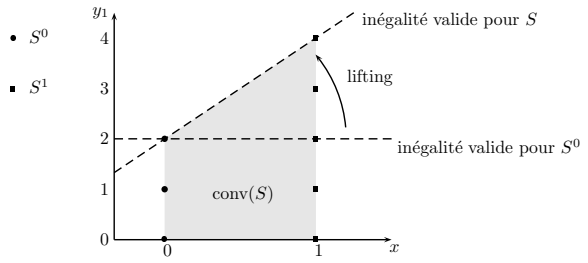

FIG. 4.1 – *Exemple de coupe générée par lifting.*

Sur ce même principe, il est possible de lifter simultanément sur plusieurs variables à la fois. Partant, par exemple, d'une inégalité valide $\sum_{i=3}^{n} \pi_i x_i \leq \pi_0$ pour l'ensemble des solutions vérifiant $x_1 = 0$ et $x_2 = 1$, il s'agit de déterminer les « plus gros » coefficients α et β faisant de $\alpha x_1 + \beta(1 - x_2) + \sum_{i=3}^{n} \pi_i x_i \leq \pi_0$ une inégalité valide pour S, c.-à-d. qui satisfont :

$$\begin{aligned} \alpha &\leq \pi_0 - \max\{p_i x | x \in S, x_1 = 1, x_2 = 1\} \\ \beta &\leq \pi_0 - \max\{p_i x | x \in S, x_1 = 0, x_2 = 0\} \\ \alpha + \beta &\leq \pi_0 - \max\{p_i x | x \in S, x_1 = 1, x_2 = 0\} \end{aligned}$$

L'inégalité résultante peut alors *dominer* strictement celles obtenues en liftant succesivement sur x_1 puis sur x_2 (de $\{x \in S | x_1 = 0, x_2 = 1\}$ à $\{x \in S | x_2 = 1\}$ puis à S) ou inversement sur x_2 puis x_1. D'ailleurs, puisque déterminer les valeurs ζ de la proposition est un problème généralement aussi difficile que le problème initial et qu'on se contente à la place d'approximations α et β, le lifting successif ne génère souvent pas les mêmes inégalités selon l'ordre choisi des variables. Parmi les coupes que nous proposons, plusieurs sont calculées par lifting simultané.

4.2 Coupes pour le modèle disjonctif en temps continu

4.2.1 Formulation linéaire et relaxation

Nous étudions dans un premier temps la formulation linéaire en temps continu pour le RCPSP, issue de la formulation disjonctive de Balas. Bien que l'ordonnancement soit décrit par les variables S_i du programme, les formulations en temps continu reposent essentiellement sur les variables binaires de séquencement x_{ij} définies par $x_{ij} = 1$ si i précède j dans l'ordonnancement. En effet, la seule affectation de ces variables suffit à déterminer l'ordonnancement de durée minimale correspondant, cet ordonnancement résultant du calcul des plus long chemins à partir de 0 dans le graphe des précédences définies par les arcs $i \to j$ tels que $x_{ij} = 1$.

Ce type de formulation n'est pas la mieux adaptée pour la modélisation des contraintes de ressources générales du RCPSP car elle nécessite de nombreuses variables (modèle sur les flots de ressources) ou contraintes (modèle sur les ensembles critiques) additionnelles. Pour cette raison, dans ce modèle, nous relaxons toutes les contraintes de ressources en même temps que les contraintes d'intégralité. Il s'agit alors d'exploiter au maximum les résultats de la PPC pour prétraiter et générer des coupes pour cette relaxation, de façon à réintroduire en partie les contraintes de ressources (et d'intégralité) dans le programme linéaire. Nous considérons donc des inégalités valides pour la relaxation linéaire suivante :

$$\min S_{n+1} \tag{2.27}$$

sujet à :

$$x_{ij} = 1 \qquad \forall\ (i,j) \in E \tag{2.30}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall\ (i,j) \in V^2, i < j \tag{2.31}$$

$$x_{ik} \geq x_{ij} + x_{jk} - 1 \qquad \forall\ (i,j,k) \in V^3 \tag{2.32}$$

$$S_j - S_i \geq -M + (p_i + M)x_{ij} \qquad \forall\ (i,j) \in V^2 \tag{2.33}$$

$$x_{ij} \in [0,1],\ x_{ii} = 0,\ S_i \geq 0 \qquad \forall\ (i,j) \in V^2$$

La formulation PPC basée sur la matrice des distances et le shaving sur les séquencements s'avèrent particulièrement appropriés dans ce cas, les contraintes inférées se linéarisant facilement au moyen des variables x_{ij}.

4.2.2 Prétraitement par PPC

Grâce aux distances minimales calculées par PPC, on peut réduire de manière significative le nombre de variables du programme, en fixant, pour tout couple de variables $(i,j) \in V^2$:

$$x_{ij} = 1 \qquad \text{si } b_{ij} \geq p_i$$
$$x_{ij} = 0 \qquad \text{si } b_{ji} \geq 1 - p_i$$

De la même façon, on peut évidemment utiliser $-b_{ij}$ comme valeur « grand M » dans la contrainte (2.33) liant les variables x_{ij} et $S_j - S_i$. Il est possible aussi de renforcer cette contrainte, en lui préférant l'une des trois inégalités suivantes, selon la valeur du séquencement $i \sim j$ dans la

relaxation PPC :

$$S_j - S_i \geq b_{ij} \qquad \text{si } b_{ij} \geq p_i \tag{4.4}$$

$$S_j - S_i \geq b_{ij} + (p_i - b_{ij})x_{ij} \qquad \text{si } 1 - p_j \leq b_{ij} < p_i \tag{4.5}$$

$$S_j - S_i \geq (1 - p_j) + (p_i + p_j - 1)x_{ij} + (b_{ij} + p_j - 1)x_{ji} \qquad \text{si } b_{ij} < 1 - p_j. \tag{4.6}$$

Dans les deux premiers cas, il s'agit de la contrainte (2.33) compte tenu des fixations précédentes. La troisième inégalité, obtenue par lifting sur les variables binaires et correspondant au cas où aucune information n'est donnée sur le séquencement relatif entre i et j, domine strictement (2.33). Elle se vérifie simplement en considérant les valeurs entières possibles du couple (x_{ij}, x_{ji}) : $S_j - S_i \geq p_i$ si $i \rightarrow j$, $S_j - S_i \geq b_{ij}$ si $j \rightarrow i$, $S_j - S_i \geq 1 - p_j$ si $i \| j$.

Avec les disjonctions déduites de la PPC (shaving et règles sur les triplets symétriques), on détermine des inégalités de cliques à deux éléments. En mémorisant aussi les ensembles disjonctifs minimaux de trois activités qui sont calculés pour l'application des règles sur les triplets symétriques, on génère aussi des inégalités de cliques de cardinalité 3 :

$$x_{ij} + x_{ji} = 1 \qquad \forall (i,j) \in D \tag{4.7}$$

$$\sum_{u,v \in \{i,j,k\}} x_{uv} \geq 1 \qquad \forall\ (i,j,k) \in EDM_3 \tag{4.8}$$

Par ce prétraitement, on assure que la relaxation linéaire est au moins aussi bonne que la relaxation PPC. Pour améliorer la borne, on ajoute de manière itérative à cette relaxation, des inégalités valides qui coupent la solution fractionnaire de ce programme. Ci-dessous, les trois premiers types de coupes exploitent les déductions du shaving, tandis que les dernières utilisent les ensembles disjonctifs précalculés par PPC et reposent sur un principe similaire au edge-finding.

4.2.3 Coupes de séquencement issues du shaving

Les coupes de séquencement basées sur le shaving considèrent le séquencement relatif d'une paire d'activités $\{i, j\}$ par rapport au séquencement d'une autre paire d'activités $\{h, l\}$. Ce lien est représenté de manière implicite dans les matrices de shaving associées à ces paires d'activités. Par exemple, la condition $b_{ij}^{h \rightarrow l} \geq p_i$ implique la relation $h \rightarrow l \ \Rightarrow \ i \rightarrow j$. Si cette condition est satisfaite, l'inégalité linéaire $x_{ij} \geq x_{hl}$ est clairement valide.

Suivant ce principe, nous avons identifié l'ensemble des coupes dominantes liant au plus trois des variables x_{ij}, x_{ji}, x_{hl} et x_{lh}, compte tenu des valeurs des distances entre i et j et entre h et l dans les matrices B, $B^{i \rightarrow j}$, $B^{h \rightarrow l}$, $B^{i \| j}$, $B^{h \| l}$, $B^{j \rightarrow i}$ et $B^{l \rightarrow h}$.

Pour un aperçu théorique de cette approche, nous considérons un second exemple : On suppose ici que l'ordre relatif des activités i et j est libre dans la relaxation PPC (c.-à-d. , si $b_{ji} < 1 - p_i$ et $b_{ij} < 1 - p_j$), et que, au contraire, les algorithmes de consistance ont permis de détecter que l'activité h ne peut être ordonnancée après l'activité l ($l \nrightarrow h$ si $b_{hl} \geq 1 - p_l$ et $b_{lh} < 1 - p_h$). Dans ce cas, la variable x_{lh} est fixée à 0 dans la relaxation linéaire tandis que x_{ij}, x_{ji} et x_{hl} sont libres entre 0 et 1. Puisque $x_{ij} + x_{ji} \leq 1$, la projection P' de l'espace des solutions fractionnaires de la relaxation sur le plan $\{(x_{ij}, x_{ji}, x_{hl})\} \subseteq [0,1]^3$ est inclue dans l'enveloppe convexe de $X = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right\}$. Comme chacun de ces vecteurs est associé à une condition nécessaire de réalisabilité, cet ensemble peut être réduit en analysant les matrices de shaving. Les coupes proposées sont alors les facettes de l'enveloppe convexe du sous-

ensemble obtenu après suppression d'un ou plusieurs des éléments de X.

Par exemple, le vecteur $\begin{pmatrix}1\\0\\1\end{pmatrix}$ peut être supprimé de P' si la condition suivante est satisfaite :

$$b_{lh}^{i\to j} \geq 1 - p_h \text{ ou } b_{ji}^{h\to l} \geq 1 - p_i. \tag{a}$$

En effet, elle correspond à l'implication $i \to j \Rightarrow \neg(h \to l)$. $x_{hl} \leq 1 - x_{ij}$ est une facette de $\text{conv}(X \setminus \left\{\begin{pmatrix}1\\0\\1\end{pmatrix}\right\})$; par conséquent, c'est une inégalité valide de P' si la condition est satisfaite.

Si, de plus, l'une des trois conditions suivantes est vérifiée :

$$b_{lh}^{i||j} \geq 1 - p_h \text{ ou } b_{ij}^{h\to l} \geq p_i \text{ ou } b_{ji}^{h\to l} \geq p_j. \tag{b}$$

alors, on ne peut avoir simultanément $i||j$ et $h \to l$. Dans ce cas, P' est inclu dans l'ensemble convexe $X' = \text{conv}(X \setminus \left\{\begin{pmatrix}1\\0\\1\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}\right\})$. Plutôt que générer les deux coupes correspondant à chacune des conditions, on génère la coupe dominante $x_{ji} \leq x_{hl}$, qui est une facette de X'. La figure 4.2 illustre ces inégalités et comment l'exploitation conjointe des déductions (a) et (b) donne de meilleures inégalités qu'un traitement séparé.

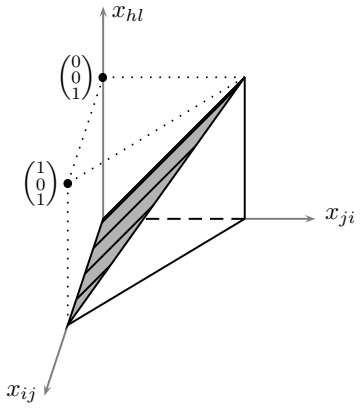

FIG. 4.2 – *Exemple de coupe de séquencement issue du shaving*

Le schéma général pour identifier les coupes dominantes de ce type est le suivant : Pour deux paires $\{i, j\}$ et $\{h, l\}$ d'activités dont l'ordre relatif n'est pas fixé par PPC, on considère le polytope de la figure 4.2, autrement dit la projection sur trois des variables x_{ij}, x_{ji}, x_{hl} et x_{lh}, et on en supprime le maximum de points extrêmes en analysant les matrices de shaving. On en déduit alors une ou deux facettes correspondant aux coupes de séquencement.

4.2.4 Coupes de distance issues du shaving

Une coupe de distance issue du shaving est définie pour toutes activités i, j, h et l telles que $i \neq j$ et $h < l$, et impose une distance minimale entre les activités i et j en fonction du séquencement relatif des activités h et l. Elle s'obtient facilement par lifting sur les variables x_{hl} et x_{lh} en utilisant des coefficients déduits des matrices de shaving :

$$S_j - S_i \geq b_{ij}^{h||l} + (b_{ij}^{h\to l} - b_{ij}^{h||l})x_{hl} + (b_{ij}^{l\to h} - b_{ij}^{l||h})x_{lh}. \tag{4.9}$$

La validité de cette coupe se vérifie en testant les trois valeurs admissibles, $\binom{0}{0}$, $\binom{1}{0}$ et $\binom{0}{1}$, pour le couple de variables (x_{hl}, x_{lh}). On note que, dans le cas général, il n'existe pas de dominance entre ces inégalités et les inégalités précédentes sur les séquencements.

4.2.5 Coupes de chemin issues du shaving

La valeur optimale du RCPSP correspond à la longueur d'un chemin constitué d'arcs (i, j) tels que $x_{ij} = 1$. Partant de cette observation, nous avons identifié des coupes de chemin avec trois activités, de la forme :

$$S_l - S_i \geq \alpha + \beta x_{ij} + \gamma x_{jl} \quad \forall (i, j, l) \in V^3 \tag{4.10}$$

On note $C(\alpha, \beta, \gamma)$ cette inégalité. Pour qu'elle soit valide, les coefficients α, β et γ doivent correspondre chacun à une évaluation par défaut de la distance entre S_i et S_l selon les différentes valeurs de x_{ij} et x_{jl}. De nouveau, les matrices de shaving offrent de bonnes évaluations de ce type, permettant de générer des coupes profondes.

Nous détaillons ci-dessous l'application d'un lifting simultané sur les variables binaires de l'inégalité permettant de définir les meilleurs coefficients (α, β, γ) possibles pour les évaluations données par les matrices de shaving.

Soient,

$$\begin{aligned}
b_{il}^{00} &= \min\{S_l - S_i | S \in \mathcal{S}(x_{ij} = 0, x_{jl} = 0)\}, \\
b_{il}^{10} &= \min\{S_l - S_i | S \in \mathcal{S}(x_{ij} = 1, x_{jl} = 0)\}, \\
b_{il}^{01} &= \min\{S_l - S_i | S \in \mathcal{S}(x_{ij} = 0, x_{jl} = 1)\}, \\
b_{il}^{11} &= \min\{S_l - S_i | S \in \mathcal{S}(x_{ij} = 1, x_{jl} = 1)\}.
\end{aligned}$$

où $\mathcal{S}(x_{ij} = 1, x_{jl} = 0)$ désigne, par exemple, l'ensemble des ordonnancements réalisables pour lesquels i précède j ($x_{ij} = 1$) et j ne précède pas l ($x_{jl} = 0$). En énumérant simplement les quatres valeurs possibles du couple de variables (x_{ij}, x_{jl}), on vérifie le lemme suivant :

Lemme 8 *$C(\alpha, \beta, \gamma)$ est une inégalité valide si et seulement si $\alpha \leq b_{il}^{00}$, $\alpha + \beta \leq b_{il}^{10}$, $\alpha + \gamma \leq b_{il}^{01}$ et $\alpha + \beta + \gamma \leq b_{il}^{11}$.*

Malheureusement, le calcul de b_{il}^{00}, b_{il}^{10}, b_{il}^{01} et b_{il}^{11} est en soi aussi difficile que la résolution du problème initial. En supposant cependant que l'on connaisse des minorants de ces valeurs, disons A, B, C et D respectivement, alors il existe une relation de dominance entre les inégalités valides $C(\alpha, \beta, \gamma)$ correspondantes :

Proposition 9 *Toute inégalité $C(\alpha, \beta, \gamma)$ satisfaisant $\alpha \leq A$, $\alpha + \beta \leq B$, $\alpha + \gamma \leq C$ et $\alpha + \beta + \gamma \leq D$ est valide et est dominée par la conjonction de deux d'entre elles $C1$ et $C2$ définies par :*
(i) si $A + D \geq B + C$, alors $C1 = C(A, B - A, C - A)$ et $C2 = C(B + C - D, D - C, D - B)$
(ii) si $A + D \leq B + C$, alors $C1 = C(A, D - C, C - A)$ et $C2 = C(A, B - A, D - B)$.

Preuve. Dans les deux cas *(i)* et *(ii)*, $C1$ et $C2$ sont valides puisqu'elles satisfont les conditions du lemme 8. Soit (α, β, γ) satisfaisant les conditions de la proposition, alors $C(\alpha, \beta, \gamma)$ est aussi valide pour la même raison. On démontre ici la première assertion *(i)* ; la seconde *(ii)* se prouve de manière symétrique. On suppose donc que $A + D \geq B + C$, $C1 = C(A, B - A, C - A)$ et $C2 = C(B + C - D, D - C, D - B)$ et on cherche à montrer qu'une solution quelconque $(\bar{S}, \bar{x})$ de la relaxation linéaire vérifiant $C1$ et $C2$, vérifie aussi $C(\alpha, \beta, \gamma)$.

On pose $\delta_\alpha = A - \alpha$, $\delta_\beta = B - A + \delta_\alpha - \beta$ et $\delta_\gamma = C - A + \delta_\alpha - \gamma$; ce sont des valeurs positives telles que $\delta_\alpha - \delta_\beta - \delta_\gamma \leq A + D - B - C$.

Si la solution $(\bar{S}, \bar{x})$ est telle que $\bar{x}_{ij} + \bar{x}_{jl} \leq 1$. Comme elle est supposée vérifier l'inégalité $C1$, alors :

$$\begin{aligned} \alpha + \beta\bar{x}_{ij} + \gamma\bar{x}_{jl} &= A + (B-A)\bar{x}_{ij} + (C-A)\bar{x}_{jl} \\ &\quad + \delta_\alpha(\bar{x}_{ij} + \bar{x}_{jl} - 1) - \delta_\beta\bar{x}_{ij} - \delta_\gamma\bar{x}_{jl} \\ &\leq A + (B-A)\bar{x}_{ij} + (C-A)\bar{x}_{jl} \leq \bar{S}_l - \bar{S}_i. \end{aligned}$$

Sinon $\bar{x}_{ij} + \bar{x}_{jl} > 1$ et puisque $C2$ est vérifiée par $(\bar{S}, \bar{x})$, alors :

$$\begin{aligned} \alpha + \beta\bar{x}_{ij} + \gamma\bar{x}_{jl} &= (B + C - D) + (D-C)\bar{x}_{ij} + (D-B)\bar{x}_{jl} \\ &\quad + (\delta_\alpha - \delta_\beta - \delta_\gamma - (A + D - B - C))(\bar{x}_{ij} + \bar{x}_{jl} - 1) \\ &\quad + \delta_\beta(\bar{x}_{jl} - 1) + \delta_\gamma(\bar{x}_{ij} - 1) \\ &\leq (B + C - D) + (D-C)\bar{x}_{ij} + (D-B)\bar{x}_{jl} \leq \bar{S}_l - \bar{S}_i. \end{aligned}$$

Dans les deux cas, $(\bar{S}, \bar{x})$ satisfait bien l'inégalité $C(\alpha, \beta, \gamma)$. ■

Connaissant les minorants A, B, C et D, il n'est donc besoin de générer qu'au plus deux coupes pour un chemin (i, j, l). Pour que ces coupes soient les plus profondes possibles, il s'agit de calculer des minorants proches des coefficients b_{il}^{00}, b_{il}^{01}, b_{il}^{10}, and b_{il}^{11}. Une méthode pour cela, serait d'effectuer un shaving simultané sur les deux décisions de séquencements associées en posant, par exemple pour B, $B = b_{il}^{(i\to j)\wedge j\nrightarrow l}$. Pour préserver du temps de calcul, nous approximons ces valeurs en utilisant les distances déjà calculées dans la phase PPC par le shaving simple sur un séquencement :

$$\begin{aligned} A &= \max\{\ \min\{b_{il}^{i||j}, b_{il}^{j\to i}\},\ \min\{b_{il}^{j||l}, b_{il}^{l\to j}\},\ \min\{b_{ij}^{j||l}, b_{ij}^{l\to j}\} + \min\{b_{jl}^{i||j}, b_{jl}^{j\to i}\}\ \} \\ B &= \max\{\ b_{il}^{i\to j},\ \min\{b_{il}^{j||l}, b_{il}^{l\to j}\},\ \min\{b_{ij}^{j||l}, b_{ij}^{l\to j}\} + b_{jl}^{i\to j}\ \} \\ C &= \max\{\ \min\{b_{il}^{i||j}, b_{il}^{j\to i}\},\ b_{il}^{j\to l},\ b_{ij}^{j\to l} + \min\{b_{jl}^{i||j}, b_{jl}^{j\to i}\}\ \} \\ D &= \max\{\ b_{il}^{i\to j},\ b_{il}^{j\to l},\ b_{ij}^{i\to j} + b_{jl}^{j\to l},\ b_{ij}^{j\to l} + b_{jl}^{i\to j}\ \}. \end{aligned}$$

Par des arguments similaires, nous identifions aussi des inégalités valides dominantes pour des chemins de quatre activités commençant à 0 ou terminant en $n+1$:

$$S_l(-S_0) \geq \alpha + \beta x_{ij} + \gamma x_{jl} \qquad \forall (i,j,l) \in V^3 \tag{4.11}$$

$$S_{n+1} - S_i \geq \alpha + \beta x_{ij} + \gamma x_{jl} \quad \forall (i,j,l) \in V^3 \tag{4.12}$$

Finalement, nous générons une coupe pour un chemin de quatre activités quelconques, sans preuve complète de dominance, de la forme :

$$S_l - S_i \geq \alpha + \beta x_{ij} + \gamma x_{jh} + \delta x_{hl} \quad \forall (i,j,h,l) \in V^4. \tag{4.13}$$

4.2.6 Coupes de clique et edge-finding

Les inégalités valides présentées dans cette section sont définies pour tout ensemble disjonctif (ou clique C) et permettent de préciser la date de début d'une des activités j en fonction de son séquencement parmi les autres activités de la clique. En ce sens, elles sont inférées par des principes similaires aux règles de edge-finding sur les ensembles disjonctifs, pour le traitement de la contrainte `cumulative` en PPC. Comme pour le edge-finding donc, chacune de ces inégalités a deux expressions symétriques. L'une correspond à une borne inférieure de la distance de la date de

début S_j de j par rapport à la date de début de l'ordonnancement $S_0 = 0$, et l'autre correspond à une borne inférieure de la distance entre la date de fin S_{n+1} du projet et la date de fin $S_j + p_j$ de j.

Par la suite, C dénote une clique d'activités en disjonction deux à deux, et j et l deux activités distinctes de C.

La première coupe que nous avons implémentée, *half cut* est proposée par [Applegate 1991] pour le problème du job-shop. Dans la forme directe, elle établit le fait qu'une activité j de C doit être ordonnancée après toutes les activités i de C telles que $x_{ij} = 1$:

$$S_j \geq \min_{i \in C} b_{0i} + \sum_{i \in C \setminus \{j\}} p_i x_{ij} \quad \forall j \in C. \tag{4.14}$$

En posant $q_i = b_{i(n+1)} - p_i$, la distance minimale entre la fin d'une activité i et la fin du projet alors, alors l'inégalité symétrique s'écrit :

$$S_{n+1} - S_j \geq p_j + \sum_{i \in C \setminus \{j\}} p_i x_{ji} + \min_{i \in C} q_i. \tag{4.15}$$

Pour les autres coupes, nous omettons l'inégalité symétrique, facilement dérivable.

La seconde coupe, *late job cut*, a été introduite par [Dyer 1990], toujours pour le problème du job-shop. Elle modifie la coupe précédente en supposant qu'une autre activité $l \in C$ est séquencée en première position, une pénalité étant ajoutée si ce n'est pas réellement le cas :

$$S_j \geq b_{0l} + \sum_{i \in C \setminus \{j\}} p_i x_{ij} + \sum_{i \in C \setminus \{l\}} \min\{0, b_{0i} - b_{0l}\} x_{il} \quad \forall j, l \in C. \tag{4.16}$$

Ces coupes reposent en fait sur les bornes des fenêtres de temps des activités ($ES_l = b_{0l}$). Utilisant les distances entre les activités, nous proposons une version alternative de cette coupe en introduisant la date de début S_l de l à la place de l'approximation de sa date de début au plus tôt b_{0l}. Dans ce cas, pour les pénalités, si une activité i est ordonnancée avant l, la date de début de la clique (S_i) n'est plus égale à S_l mais minorée par $S_l + b_{li}$.

$$S_j \geq S_l + \sum_{i \in C \setminus \{j\}} p_i x_{ij} + \sum_{i \in C \setminus \{l\}} b_{li} x_{il} \quad \forall j, l \in C. \tag{4.17}$$

Aucune relation de dominance ne peut être établie entre cette nouvelle coupe et *late job cut* du fait que $S_l \geq b_{0l}$ et $b_{li} \leq b_{0i} - b_{0l}$. (À moins d'avoir $b_{il} = b_{0i} - b_{0l} \geq 0$ pour tout $i \in C \setminus \{l\}$, dans ce cas, la nouvelle coupe est plus forte.)

Enfin, nous identifions une dernière sorte de coupe qui domine (4.17) au cas où l est prouvée devant précéder toutes les activités de C. Une telle condition est détectée dans la phase PPC par la règle duale du edge-finding.

$$S_j \geq S_l + \sum_{i \in C \setminus \{j\}} p_i x_{ij} + \min_{i \in C \setminus \{l\}} (b_{li} - p_l) \quad \forall j, l \in C \mid l \rightarrow C \setminus \{l\}. \tag{4.18}$$

Pour générer l'ensemble de ces coupes, il nous suffit de considérer les ensembles disjonctifs maximaux EDM calculées à la dernière itération du processus de propagation de la PPC. Nous en identifions alors les sous-ensembles C et les activités j et l, pour lesquels on calcule ces inégalités, de la même manière que l'algorithme de edge-finding de [Nuijten 1994] que nous avons implémenté pour la PPC.

4.3 Coupes pour le modèle en temps discrétisé

4.3.1 Formulation linéaire et relaxation

Nous avons mené une étude similaire de génération de coupes basées sur la PPC pour la formulation linéaire du RCPSP en temps discretisé (voir section 2.3.2). Ce modèle, initialement conçu par Pritsker et al. [Pritsker 1969], est le plus étudié dans la littérature (p. ex. parmi les études les plus récentes, [Christofides 1987, Sankaran 1999, Möhring 2003]). On rappelle qu'il est uniquement basé sur des variables binaires y_{it} représentant si oui ou non la date de début de l'activité i est égale au temps t. Bien que cette formulation puisse contenir théoriquement un nombre beaucoup plus élevé de variables que la précédente, il est rare cependant de devoir ordonnancer des activitée de durées très dissimilaires. De sorte, on peut généralement se ramener à des problèmes dont la durée d'ordonnancement est relativement courte. En particulier, les instances de tests (BL, KSD) pour le RCPSP tiennent compte de cette remarque, où la valeur optimale est souvent de l'ordre de la taille n du projet (c'est le cas pour les instances hautement cumulatives, les plus difficiles) et dépasse rarement $4n$.

Au moyen de ces variables, les contraintes de ressources se modélisent aisément et les contraintes de précédence se modélisent même de deux façons différentes : agrégées ou désagrégées. Dans notre algorithme de génération de coupes, nous partons donc simplement de la relaxation continue, soit, avec les contraintes de précédence agrégées (souvent appelée la relaxation faible, *weak LP-relaxation*) :

$$\min \sum_{t=0,\dots,T} t y_{(n+1)t} \tag{2.38}$$

subject to :

$$\sum_{t=0}^{T} y_{jt} = 1 \qquad \forall\, j \in V \tag{2.39}$$

$$\sum_{t=0}^{T} t(y_{jt} - y_{it}) \geq p_i \qquad \forall\, (i,j) \in E \tag{2.40}$$

$$\sum_{j \in V} r_{jk} \sum_{\tau=t-p_j+1}^{t} y_{j\tau} \leq R_k \qquad \forall\, k \in \mathcal{R}, \forall\, t \in \{0,\dots,T\} \tag{2.41}$$

$$y_{jt} \in [0,1] \qquad \forall\, j \in V, \forall\, t \in \{0,\dots,T\}$$

soit, avec les contraintes de précédence désagrégées de Christophides et al. (*strong LP-rexation*), en remplaçant ci-dessus (2.40) par :

$$\sum_{\tau=t}^{T} y_{i\tau} + \sum_{\tau=0}^{t+p_i-1} y_{j\tau} \leq 1 \qquad \forall\, (i,j) \in E, \forall\, t = 0,\dots,T \tag{2.42}$$

Comme précédemment, nous utilisons les résultats de la PPC pour prétraiter ces relaxations continues ainsi que pour définir des inégalités valides à y ajouter de façon à couper leurs solutions fractionnaires.

4.3.2 Prétraitement par PPC

La matrice des distances calculée par PPC permet de réduire drastiquement le nombre de variables du programme linéaire. En effet, les variables y_{it} n'ont à être définies que pour des temps

t auxquels l'activité i peut effectivement commencer en accord avec sa fenêtre de temps. Cela revient à fixer à 0 toutes les variables y_{it} telles que $t < ES_i = b_{0i}$ ou $t > LS_i = -b_{i0}$.

On peut aussi définir, à la place des contraintes de précédence simples, les contraintes de précédence généralisées inférées par PPC et qui s'écrivent de manière agrégée ou désagrégée :

$$\sum_{t=ES_j}^{LS_j} t y_{jt} - \sum_{t=ES_i}^{LS_i} t y_{it} \geq b_{ij} \quad \forall\ (i,j) \in V^2 \tag{4.19}$$

$$\sum_{\tau=t}^{LS_i} y_{i\tau} + \sum_{\tau=ES_j}^{t+b_{ij}-1} y_{j\tau} \leq 1 \qquad \forall\ (i,j) \in V^2\ \forall t \in \{ES_j - b_{ij} + 1, \ldots, LS_i\} \tag{4.20}$$

Avec ce modèle, notre méthode s'applique aussi à la classe des RCPSP avec délais entre les activités.

4.3.3 Coupes de cliques

Avec cette formulation, les coupes de cliques, comme on les entendait pour la formulation précédente en temps continu, rejoignent les inégalités de cliques de variables, bien connues en programmation linéaire en variables binaires. En effet, à un ensemble disjonctif maximal C correspond $T+1$ inégalités, signifiant qu'au plus une activité de la clique ne peut être en cours d'exécution à chaque temps t :

$$\sum_{(i,\tau) \in C_t} y_{i\tau} \leq 1 \quad \forall\ t \in \{0, \ldots, T\}, \tag{4.21}$$

où $C_t = \big\{(i,\tau) \in C \times \{\max\{ES_i, t - p_i + 1\}, \ldots, \min\{LS_i, t\}\} \mid ES_i \leq t < LS_i + p_i\big\}$.

Dans [Christofides 1987], une version plus simple de ces coupes (interdisant seulement aux activités d'une clique de commencer à un même temps t) est utilisée, mais les auteurs n'explicitent pas le calcul des ensembles disjonctifs maximaux. Dans [Sankaran 1999], les ensembles C_t sont générés à la volée par une heuristique considérant la solution fractionnaire courante et les contraintes de ressources violées au temps t. Sans calcul supplémentaire, nous considérons pour C les ensembles disjonctifs maximaux générés durant la phase PPC.

4.3.4 Coupes de distance issue du shaving

Dans l'esprit d'utiliser les déductions du shaving pour la génération de coupes, nous avons cherché à traduire l'implication

$$S_j - S_i \geq p_i \ \Rightarrow\ S_l - S_h \geq b_{hl}^{i \to j} \tag{4.22}$$

pour des paires distinctes d'activités $\{i,j\}$ et $\{h,l\}$ en termes d'inégalités linéaires en les variables indexées par le temps y_{it}. Pour s'assurer que cette relation n'est pas dominée par une autre contrainte de la relaxation, on suppose que $b_{hl}^{i \to j} > b_{hl}$ et que $b_{ij} \leq p_i - 1 < -b_{ji}$ (le séquencement relatif de (i,j) n'est pas connu ; au mieux, on sait que j ne peut précéder i si éventuellement $b_{ij} \geq 1 - p_j$).

Bien que les déductions du shaving sur les séquencements soient moins trivialement linéarisables que pour la formulation en temps continu, la relation (4.22) va pouvoir se modéliser en remarquant qu'il s'agit d'une implication entre deux contraintes de précédence, qui se réécrivent de deux façons selon le formalisme agrégé ou désagrégé.

Pour simplifier l'écriture, on note z_{ij} le terme $\sum_{t=ES_j}^{LS_j} t y_{jt} - \sum_{t=ES_i}^{LS_i} t y_{it}$; autrement dit, $z_{ij} = S_j - S_i$.

Linéarisation agrégée

Puisque $z_{hl} \geq b_{hl}$ dans tout ordonnancement, la relation (équivalente à $z_{ij} > p_i - 1 \;\Rightarrow\; z_{hl} \geq b_{hl}^{i \to j}$) se linéarise par l'inégalité suivante :

$$(-b_{ji} - p_i + 1)(z_{hl} - b_{hl}) \geq (z_{ij} - p_i + 1)(b_{hl}^{i \to j} - b_{hl}). \tag{4.23}$$

Elle s'obtient par projection de l'espace des solutions sur le plan (z_{ij}, z_{hl}) puis par une sorte de lifting sur la « variable » z_{hl} comme le montre la figure 4.3. Sur ce schéma, l'ensemble des solutions entières (après projection sur le plan) du programme linéaire est contenu dans l'espace hachuré et les solutions fractionnaires se situent dans la zone grisée.

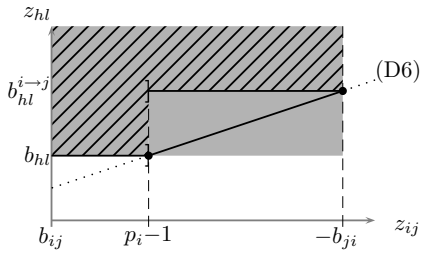

FIG. 4.3 – *Projection de $\mathcal{S}$ sur le plan (z_{ij}, z_{hl})*

Linéarisations désagrégées

La relation (4.22) se réécrit aussi en considérant la contrainte de précédence impliquée de manière désagrégée :

$$z_{ij} > p_i - 1 \;\Rightarrow\; \sum_{\tau=t}^{LS_h} y_{h\tau} + \sum_{\tau=ES_l}^{t+b_{hl}^{i \to j}-1} y_{l\tau} \leq 1 \quad \forall t \in \{0, \ldots, T\},$$

et se linéarise par l'ensemble des inégalités :

$$-b_{ji} - z_{ij} \geq (-b_{ji} - p_i + 1)\left(\sum_{\tau=t}^{LS_h} y_{h\tau} + \sum_{\tau=ES_l}^{t+b_{hl}^{i \to j}-1} y_{l\tau} - 1\right)$$
$$\forall t \in \{\max\{ES_h, ES_l - b_{hl}^{i \to j} + 1\}, \ldots, \min\{LS_h, LS_l - b_{hl}^{i \to j} + 1\}\}. \tag{4.24}$$

Comme une contrainte de précédence s'écrit, sous la forme désagrégée, par un ensemble d'inégalités, on comprend pourquoi, pour la linéarisation, seul le terme de droite de l'implication peut être traduit de cette manière. On doit donc considérer la relation équivalente à (4.22), $S_l - S_h < b_{hl}^{i \to j} \Rightarrow S_i - S_j \geq 1 - p_i$, pour désagréger l'autre contrainte de précédence :

$$z_{hl} < b_{hl}^{i \to j} \;\Rightarrow\; \sum_{\tau=t}^{LS_j} y_{j\tau} + \sum_{\tau=ES_i}^{t-p_j} y_{i\tau} \leq 1 \quad \forall t \in \{0, \ldots, T\}.$$

Par conséquent, une autre linéarisation possible est donnée par l'ensemble des inégalités valides

suivantes :

$$z_{hl} - b_{hl} \geq \left(\sum_{\tau=t}^{LS_j} y_{j\tau} + \sum_{\tau=ES_i}^{t-p_i} y_{i\tau} \right) (b_{hl}^{i \to j} - b_{hl})$$
$$\forall t \in \{\max\{ES_j, ES_i + p_i\}, \ldots, \min\{LS_j, LS_i + p_i\}\}. \quad (4.25)$$

Comme pour les contraintes de précédence, les coupes désagrégées (4.24) ou (4.25) sont théoriquement plus profondes que les coupes agrégées (4.23).

Cas particulier

Au cas où h (ou symétriquement l) est l'activité fictive 0, on connaît alors sa date début ($S_h = 0$) ce qui permet de simplifier l'écriture. Ainsi, on montre facilement que les inégalités correspondantes (4.23) et (4.24) sont dominées par l'inégalité valide suivante :

$$-b_{ji} - z_{ij} \geq (-b_{ji} - p_i + 1) \left(\sum_{\tau=ES_l}^{ES_l^{i \to j} - 1} y_{l\tau} + \sum_{\tau=LS_l^{i \to j}+1}^{LS_l} y_{l\tau} \right). \quad (4.26)$$

En effet, cette dernière modélise l'implication plus forte : $S_j - S_i \geq p_i \;\Rightarrow\; ES_l^{i \to j} \leq S_l \leq LS_l^{i \to j}$.

4.4 Coupes pour le modèle préemptif sur les ensembles admissibles

L'approche du modèle préemptif sur les ensembles admissibles est différente des deux approches précédentes. Il s'agit ici de renforcer la borne inférieure calculée de manière destructive par PPC et génération de colonnes, proposée par Brucker et Knust [Brucker 2000] et présentée à la section 3.2.2. Cette méthode est basée sur le modèle préemptif des enssembles admissibles décrite p. 55. L'amélioration de cette méthode, apportée par Philippe Baptiste et à laquelle nous avons participé, repose sur deux points : une propagation de contraintes plus poussée et la génération de coupes pour la relaxation linéaire préemptive.

Pour le filtrage, sont utilisées, les règles de *time-tabling*, la consistance de chemin sur les contraintes de précédence et les règles du edge-finding sur des ensembles disjonctifs maximaux. L'originalité de cette procédure réside en partie sur la génération des ensembles disjonctifs par la résolution de programmes linéaires en variables binaires : des variables ξ_i sont associées aux activités i ($\xi_i = 1$ si i est dans l'ensemble disjonctif) et contraintes par $\xi_i + \xi_j \leq 1$ si, i et j ne sont ni en précédence, ni en disjonction. L'objectif est la maximisation de la somme des durées des activités de la clique. Avec cette technique, dont l'utilité est validée par les expérimentations par rapport à la résolution de manière heuristique, on rejoint de nouveau l'idée d'intégration des approches, ici de la PL pour la PPC. La procédure PPC est présentée en détail dans [Baptiste 2004], nous insistons ici plus particulièrement sur les inégalités valides qui peuvent être ajoutées à la formulation préemptive de Brucker et Knust. Trois types de coupes ont été identifiées : les premières sont basées sur le raisonnement énergétique, les secondes prennent en compte la non-préemptivité, et les dernières, les contraintes de précédence.

4.4.1 Coupes énergétiques

La décomposition de l'horizon d'ordonnancement en intervalles de temps offre un moyen d'adapter le raisonnement énergétique, à l'inférence de contraintes linéaires pour ce modèle. La linéarisation proposée considère le temps minimal d'exécution $\underline{p_i}(d, f)$ d'une activité i dans un intervalle de temps $[d, f]$, compte tenu de la fenêtre de temps de i. Comme évoqué à la section 2.2.4, $\underline{p_i}([d, f])$ est défini par :

$$\underline{p_i}(d, f) = \min(f - d\ ,\ p_i\ , \max(0, ES_i + p_i - d)\ ,\ \max(0, f - LS_i)).$$

On génère alors des contraintes, imposant ce temps minimal d'exécution, pour les intervalles définis par des instants t_s, $s \in \{0, \ldots, \sigma\}$:

$$\sum_{s=d}^{f} \sum_{l \in \mathcal{F}_i \cap \mathcal{F}^s} x_{ls} \geq \underline{p_i}(t_d, t_f) \qquad \forall i \in \mathcal{A},\ \forall d, f \in \{1, \ldots, \sigma\}, d < f \tag{4.27}$$

4.4.2 Coupes non-préemptives

Comme les contraintes de non-préemption des activités sont relâchées dans la formulation linéaire, il est intéressant de voir comment elles peuvent être prises partiellement en compte et ajoutées au modèle sous forme de coupes. La non-préemption implique qu'une activité i ne peut s'exécuter à la fois dans deux intervalles I_s et $I_{s'}$, si ces intervalles sont éloignés de plus de p_i unités de temps.

Soit $\Psi \subseteq \{1, \ldots, \sigma\}$, les indices d'intervalles I_s, séparés deux à deux par une distance $t_{s'-1} - t_s$ $(s < s')$ supérieure ou égale à p_i, l'activité i ne peut alors être ordonnancée que dans au plus un intervalle indicé par Ψ. Par conséquent, le temps d'exécution de i sur l'ensemble des intervalles de Ψ est inférieur à la longueur maximale de ces intervalles, ce qui s'écrit :

$$\sum_{s \in \Psi} \sum_{l \in \mathcal{F}_i \cap \mathcal{F}^s} x_{ls} \leq \max\{\delta_s \mid s \in \Psi\}. \tag{4.28}$$

Un algorithme est proposé dans [Baptiste 2004] pour calculer des ensembles Ψ auxquels appliquer ces inégalités. Pour une activité i et un intervalle I_s, un ensemble Ψ est généré de manière itérative en deux phases : Ψ est initialisé à $\{s\}$ puis étendu en ajoutant, à chaque itération k, le premier intervalle $s^k > s^{k-1}$ se trouvant à une distance supérieure à p_i de s^{k-1}. Dans la seconde phase, Ψ est étendu de la même manière mais avec des intervalles entre $s - 1$ et 1.

4.4.3 Coupes de précédence

Comme pour la non-préemption, il s'agit ici de réintroduire les contraintes de précédence, relâchées dans le modèle linéaire. Pour cela, on considère, pour chaque activité i, la date d'exécution de la moitié de l'activité i. On ajoute ainsi au programme linéaire une variable m_i pour représenter cette date. Pour un ordonnancement non-préemptif, les contraintes de précédence s'expriment facilement au moyen de ces variables :

$$m_j - m_i \geq \frac{p_i + p_j}{2} \qquad \forall (i, j) \in E. \tag{4.29}$$

Il convient maintenant de lier les variables m_i aux variables initiales x_{ls}. Une activité i se découpe en σ morceaux A_{is}, correspondant chacun à la partie de i exécutée dans un intervalle I_s. Dans ces conditions, A_{is} a une durée p_{is} égale à $\sum_{l \in \mathcal{F}_i \cap \mathcal{F}^s} x_{ls}$, et la date m_{is}, à laquelle A_{is} est à

moitié exécuté, est comprise entre $t_{s-1}+\frac{1}{2}$ et $t_s-\frac{1}{2}$ (car sa durée est soit nulle, soit au moins égale à 1). Avec la non-préemption de l'activité i, alors m_i est le barycentre des morceaux d'activités A_{is}, autrement dit :

$$m_i = \frac{m_{i1}p_{i1} + \cdots + m_{i\sigma}p_{i\sigma}}{p_i}.$$

Ces observations se traduisent, pour toute activité $i \in \mathcal{A}$, par les inégalités linéaires suivantes :

$$\sum_{s=1}^{\sigma}(t_{s-1}+\frac{1}{2}) \sum_{l\in\mathcal{F}_i\cap\mathcal{F}^s} x_{ls} \leq m_i p_i \leq \sum_{s=1}^{\sigma}(t_s-\frac{1}{2}) \sum_{l\in\mathcal{F}_i\cap\mathcal{F}^s} x_{ls} \tag{4.30}$$

4.5 Résultats expérimentaux

Pour nous comparer aux bornes inférieures de la littérature, nous avons, comme précédemment, utilisé les instances de test de la librairie PSPLIB, générées par Kolisch, Sprecher et Drexl (KSD) pour 30, 60, 90 ou 120 activités. Nous avons mené séparément les expérimentations des deux premières méthodes et les expérimentations de la dernière méthode sur le modèle préemptif.

4.5.1 Modèles en temps continu et en temps discrétisé

Les algorithmes de calcul de bornes inférieures, basés sur les deux premiers modèles ont entièrement (PPC, shaving, génération de coupes) été écrits en C++ en utilisant ILOG CPLEX 7.0 pour la résolution des programmes linéaires. Les expériences ont été menées sur un Pentium III 800MHz avec 384Mb de RAM sous Debian/Linux et g++ 2.95.4.

4.5.2 Bornes inférieures constructives

Dans un premier temps, nous avons testé les deux méthodes dans une approche constructive. Nous présentons, dans cette section, une comparaison expérimentale des bornes ainsi obtenues : Partant d'une borne supérieure réalisable T, l'algorithme de filtrage PPC décrit à la section 3.3.2 est invoqué pour resserrer les distances entre les activités et en particulier la distance $b_{0(n+1)}$ qui correspond à la borne inférieure du problème calculée par PPC. Si $b_{0(n+1)} = T$, le problème est résolu, sinon la matrice des distances, l'ensemble des disjonctions, les matrices de shaving et les ensembles disjonctifs maximaux sont alors stockés. Les deux premiers sont utilisés pour prétraiter la relaxation linéaire du problème. La relaxation linéaire est alors résolue de manière itérative par l'algorithme dual du simplexe, en ajoutant à chaque itération l'ensemble des inégalités valides qui coupent la solution fractionnaire courante, et en supprimant les inégalités qui ne sont plus actives dans cette solution. La procédure s'arrête quand aucune amélioration de la borne n'est apportée durant un certain nombre d'itérations, quand aucune inégalité n'est violée à une itération, ou encore quand la limite de temps de calcul est dépassée.

Dans la table 4.1, nous reportons les résultats expérimentaux sur les 264 instances KSD *non-triviales* à 30 activités, c.-à-d. les instances dont la valeur optimale n'est pas égale à la borne du chemin critique (les contraintes de ressources sont actives). Les lignes 1 et 2 donnent les déviations Δ_{opt}, moyenne et maximale, de la borne inférieure à l'optimum (connu pour toutes les instances KSD30). Les lignes 3 et 4 donnent les temps CPU, moyen et maximal. La ligne 5 donne le nombre d'instances pour lesquelles l'optimum est atteint et la ligne 6, le nombre d'instances pour lesquelles la borne de la programmation linéaire améliore la borne de la PPC. Chaque colonne correspond à une borne inférieure calculée par :

- colonnes 2 et 3 : la programmation par contrainte seule, sans shaving (filtrage local `LCP`) ou bien avec shaving (filtrage complet `CCP`) ;
- colonnes 4 et 5 : la relaxation continue de la formulation en temps discrétisé et contraintes de précédence agrégées, prétraitée par filtrage local `LCP+PL` ou bien par filtrage complet `CCP+PL` ;
- colonnes 6 et 7 : la même relaxation continue prétraitée par filtrage complet et augmentée des coupes de cliques et des coupes de shaving agrégées `CCP+ag.` ou bien désagrégées `CCP+désag.` ;
- colonne 8 : la relaxation du programme linéaire en temps continu, prétraitée par filtrage complet et augmentée des coupes `CCP+coupes`.

TAB. 4.1 – Génération de coupes : bornes constructives sur les instances non triviales de KSD30

KSD30	PPC		discret (agrégé)				continu
264 instances	`LCP`	`CCP`	`LCP+PL`	`CCP+PL`	`CCP+ag.`	`CCP+désag.`	`CCP+coupes`
moy. Δ_{opt}	5,8%	3,6%	5,3%	3,2%	3,1%	3,0%	3,2%
max. Δ_{opt}	33,7%	31,3%	25,0%	25,0%	21,8%	21,8%	29,7%
CPU moy. (s.)	0,0	2,3	1,0	3,0	10,2	35,6	4,9
CPU max. (s.)	0,0	17,3	49,5	31,1	601	1296	37,6
# BInf=opt	95	155	96	157	159	160	160
# PL>PPC	-	-	24	17	35	42	47

Pour chacune de ces instances, T est la valeur optimale. En terme de qualité de la borne, les résultats sont très bons pour les deux modèles, prouvant l'optimalité pour 160 instances parmi les 264. Comparée à la formulation en temps continu, la formulation discrète avec les coupes obtient une déviation à l'optimum légèrement meilleure mais requiert aussi beaucoup plus de temps de calcul.

Dans la PPC, le shaving induit aussi un coût de calcul supplémentaire par rapport au filtrage local, mais il permet une amélioration significative de la borne, qui se répercute dans les résultats de l'hybridation avec la PL.

Le bénéfice des coupes pour les deux modèles linéaires, s'il apparaît moins dans la déviation moyenne, est plus visible dans le nombre d'amélioration apportées. En effet, parmi les 109 instances non résolues par le filtrage complet seul, les coupes du modèle en temps continu améliorent la borne pour 47 instances (car, sans les contraintes de ressources, la borne de la relaxation du modèle en temps continu est égale à la borne PPC). Tandis que les coupes agrégées et désagrégées améliorent la borne de, respectivement, 18 et 25 instances parmi les 107 instances non résolues par le modèle en temps discrétisé prétraité par la PPC complète.

L'intérêt de l'hybridation est aussi mis en valeur dans ces expérimentations puisque, en terme de déviation moyenne, la borne est bien meilleure pour les approches coopératives que pour la PL avec un simple filtrage local en prétraitement.

Nous avons aussi établit une comparaison entre les deux modèles sur les 184 instances KSD non triviales avec 60 activités (table 4.2). Parmi celles-ci, certaines sont encore ouvertes. Nous avons donc pris pour T la meilleure borne supérieure connue à ce jour et nous comparons nos bornes, non plus avec l'optimum, mais avec la valeur de la borne inférieure triviale du chemin critique LB0 (ligne 1). Ici, on considère aussi une version réduite du shaving (`RCP`) où seules les paires d'activités en disjonction sont considérées. Enfin, le temps de calcul est limité à 30 minutes.

Pour les instances à 60 activités, la PPC est nettement moins performante. L'augmentation de la taille des problèmes confère un moins grand pouvoir de déduction au filtrage, ce qui se reflète sur les résultats du shaving et des coupes basées sur la PPC. L'application d'un shaving réduit permet néanmoins d'améliorer la relaxation PPC de manière équivalente au shaving complet mais

TAB. 4.2 – Génération de coupes : bornes constructives sur les instances non triviales de KSD60

KSD60	PPC			discret (agrégé)			continu
184 instances	LCP	RCP	CCP	RCP+PL	RCP+ag.	RCP+désag.	RCP+coupes
moy. Δ_{LB0}	7,7%	9,5%	9,6%	17,5%	17,7%	17,7%	10,0%
CPU moy. (s.)	0,0	27,7	62,1	81,8	243	771	257
CPU max. (s.)	0,1	130,8	297	904	1800	1800	919
# BInf=opt	41	58	59	58	58	58	59
# PL>PPC	-	-	-	57	62	64	51

pour un temps de calcul plus raisonnable.

Pour la formulation continue, bien que les coupes améliorent encore la borne PPC pour 51 instances, les résultats en moyenne ne sont pas réellement meilleurs. Au contraire, la résolution du modèle linéaire en temps discretisé est particulièrement efficace ici par rapport à la PPC, mais l'impact des coupes est alors, assez faible sur la qualité de la borne, et plutôt néfaste pour le temps de calcul. En particulier et bien qu'elles se déclenchent, les coupes désagrégées, théoriquement meilleures que les coupes agrégées, n'aident pas ou peu à resserrer davantage la relaxation linéaire, certainement aussi car leur nombre ralentit considérablement la procédure entière.

Néanmoins, ne serait-ce que par notre principe de prétraitement, le gain de l'approche hybride PPC/PL est encore souligné ici par rapport aux deux méthodes séparées. Nous améliorons par exemple les bornes présentées dans [Möhring 2003] et basées sur la formulation en temps discretisé mais sans prétraitement.

4.5.3 Bornes inférieures destructives

Dans la suite de nos expérimentations, nous avons testé dans une approche destructive, la borne qui offrait le meilleur compromis qualité/temps, à savoir le modèle en temps discrétisé prétraité par le shaving réduit et augmenté de coupes sous la forme agrégée.

Dans la table 4.3, nous donnons les résultats de cette borne inférieure (`destr` : colonne 6) comparée à la borne inférieure du chemin critique (`LB0` : colonne 3), à la borne inférieure de Brucker et Knust (`BK` : colonne 4) et à la meilleure borne inférieure, disponible pour chaque instance (`LB*` : colonne 5) sur les instances KSD à 30, 60, 90 et 120 activités.

Pour les groupes d'instances KSD60, KSD90 et KSD120, nous donnons à chaque ligne de la table 4.3 de haut en bas : la déviation moyenne par rapport à la borne du chemin critique LB0, la déviation moyenne et maximale par rapport à la meilleur borne supérieure UB disponible pour chaque instance, le temps CPU moyen et maximal en secondes (en notant au passage que les bornes BK et destr n'ont pas été calculées sur des machines de puissance comparable), le nombre d'instances pour lesquelles la borne excède la meilleure borne connue précédemment et le nombre de nouveaux optima prouvés. Comme la valeur optimale est connue pour toutes les instances à 30 activités, les résultats sur KSD30 sont donnés pour la déviation par rapport à l'optimum, le temps CPU et le nombre d'instances pour lesquelles l'optimum est atteint.

Finalement, l'algorithme est adapté en fonction de la taille des instances de façon à réduire les temps de calcul sans trop dégrader les solutions : Pour les 480 instances à 30 activités, sont appliquées la PPC complète (avec shaving), les coupes de cliques et les coupes de shaving agrégées. Pour les 480 instances à 60 activités, on utilise l'algorithme de PPC réduit (RCP : le shaving est appliqué à au plus 500 paires d'activités en disjonction) et les coupes dans une limite de temps de calcul de 30 minutes. Pour les 480 instances à 90 activités et les 600 instances à 120 activités, on

TAB. 4.3 – Borne destructive sur le modèle discret pour les instances KSD

#activités		LB0	BK	LB*	destr
	moy Δ opt	-	1,5%	-	**0,7%**
	max	-	11,1%	-	15,2%
30	moy CPU(1)	-	**0,4**	-	3,2
(480)	max	-	4,3	-	229,9
	#LB=opt	-	318	-	**403**
	moy Δ LB0	-	7,8%	7,9%	7,7%
	moy Δ UB	7,1%	1,9%	1,8%	**1,8%**
	max	50,0%	14,7%	13,7%	17,9%
60	moy CPU(1)	-	**5**	-	168
(480)	max	-	3720	-	1963
	#LB=UB	296	341	356	**360**
	#LB>LB*	-	-	-	43
	#New opt	-	-	-	**9**
	moy Δ LB0	-	7,2%	7,2%	7,0%
	moy Δ UB	6,6%	**1,8%**	1,8%	**1,8%**
	max	50,0%	12,7%	12,7%	23,4%
90	moy CPU(1)	-	**72**	-	379
(480)	max	-	9900	-	3606
	#LB=UB	334	350	351	**364**
	#LB>LB*	-	-	-	28
	#New opt	-	-	-	**13**
	moy Δ LB0	-	21,4%	21,4%	19,1%
	moy Δ UB	16,2%	**3,8%**	3,8%	4,8%
	max	66,1%	17,4%	17,4%	33,2%
120	moy CPU(1)	-	21300(2)	-	**1388**
(600)	max	-	259200	-	3836
	#LB=UB	178	208	208	**229**
	#LB>LB*	-	-	-	60
	#New opt	-	-	-	**21**

(1) BK est calculé sur une station Sun Ultra 2 à 167MHz et destr sur un Pentium III à 800MHz.

(2) moy CPU = 355 s. pour les 481 instances traitées en un temps en deçà de la limite de 72 heures.

n'exécute seulement le prétraitement RCP et la relaxation linéaire, sans génération de coupes, et dans un temps limité à 1 heure.

En dépit d'un temps de calcul élevé, notre borne destructive est comparable à, et même surpasse parfois pour KSD30, KSD60 et KSD90, la borne la plus serrée sur ces instances (BK) en termes, à la fois, du nombre d'instances résolues et de la déviation moyenne. Pour les plus grosses instances à 120 activités, on prouve aussi l'optimalité pour plus d'instances mais notre borne est en moyenne nettement moins bonne avec des temps de calcul aussi moins élevés.

Nous démontrons une fois de plus à la suite de [Klein 1999, Brucker 2000], la puissance de l'approche destructive, en comparant pour chaque critère, qualité et temps de calcul, les résultats du même algorithme utilisé de manière constructive (table 4.1, colonne 6 pour KSD30 et table 4.2, colonne 6 pour KSD60) ou destructive (table 4.3, colonne 6).

Enfin, on notera que pour les instances non encore résolues à 60, 90 et 120 activités, nous améliorons la meilleure borne inférieure connue pour, respectivement, 43 sur 124 instances, 28 sur 129 et 60 sur 392, et parmi celles-ci nous fermons 9, 13 et 21 instances.

4.5.4 Modèle préemptif sur les ensembles admissibles

Les coupes pour le modèle préemptif ont été testées dans un autre cadre d'expérimentation. Les expérimentations ont été menées sur un HP Omnibook Pentium III à 720MHz, sur les instances KSD à 60 activités.

L'algorithme de propagation de contraintes, décrit à la section 4.4, est traité par Ilog Solver. La génération des ensembles disjonctifs maximaux par résolution optimale de programmes linéaires en variables binaires est peu coûteuse (calculés en 2,4 secondes en moyenne et 8,8 secondes dans le pire des cas) mais particulièrement performante. En effet, l'algorithme de propagation seul avec une recherche arborescente standard permet, en un temps limité à 90 secondes, de résoudre 373 instances sur les 480, en un temps de calcul moyen de 0,7 secondes, parvenant même à fermer 19 instances.

Sur les 107 instances restantes, nous avons comparé la borne `BK2` calculée par génération de colonnes, identique à celle de Brucker et Knust mais avec cet algorithme PPC, et cette même borne augmentée des coupes énergétiques, non-préemptives et de précédence `BK2+C`. Encore une fois, l'algorithme PPC utilisé ici prouve son efficacité puisque la borne `BK2` domine très largement la borne initiale de Brucker et Knust (`BK` voir table 4.4) et surpasse la meilleure borne inférieure connue pour 49 instances.

Les coupes améliorent peu en moyenne les résultats, mais permettent d'augmenter la borne `BK2` pour 22 instances. L'apport des coupes énergétiques est aussi souligné par le fait que `BK2` est moins longue à calculer en moyenne si augmentée uniquement de ces coupes (141,2 s. contre 172,7 s. sur ces 107 instances). Cela signifie que, avec les coupes énergétiques, la procédure sous une approche destructive requiert moins d'itérations, tout en améliorant la borne (strictement pour 7 instances).

4.5.5 Comparaison des bornes destructives

Pour terminer, nous présentons un tableau récapitulatif des résultats de nos bornes destructives sur les 480 instances de KSD60. Pour chacune des lignes de la table 4.4, sont indiqués : la déviation moyenne par rapport à la borne du chemin critique, le nombre d'optima prouvés, le nombre d'amélioration de la meilleure borne inférieure connue précédemment et le nombre de nouveaux optima prouvés. Pour les colonnes sont comparées les valeurs de la meilleure borne inférieure `LB*`, la borne de Brucker et Knust `BK`, la borne `BK2`, la même borne avec les coupes `BK2+C`, la borne

de la relaxation lagrangienne obtenue par génération de contraintes `GClag` (chapitre 3) et enfin, la borne destructive décrite à la section précédente `coupes`.

Nous avons volontairement omis les temps de calcul, du fait que les expérimentations n'ont pas été menées sur des calculateurs comparables, ni avec les mêmes valeurs T de départ. Il est clair néanmoins, au vu des résultats précédents, que les procédures proposées dans ce mémoire sont plus coûteuses que la borne `BK`, du moins sur ces instances à 60 activités. En contrepartie, toutes les bornes que nous proposons améliorent significativement la meilleure borne connue sur ces instances. Avec un même prétraitement, la borne basée sur la relaxation lagrangienne est légèrement supérieure à la borne basée sur la génération de coupes mais est plus coûteuse en temps de calcul. La procédure basée sur le modèle des ensembles admissibles augmenté de coupes obtient les meilleurs résultats. Pour une comparaison plus fiable des méthodes, il serait intéressant d'appliquer aux coupes du modèle en temps discrétisé et à la relaxation lagrangienne la procédure de prétraitement par programmation par contraintes, basée entre autre sur une meilleure génération des EDM, qui s'est avérée particulièrement efficace.

TAB. 4.4 – Comparaison des bornes destructives sur KSD60

480 instances	`LB*`	`BK`	`BK2`	`BK2+C`	`GClag`	`coupes`
moy Δ LB0	7,9%	7,8%	8,5%	8,6%	8,0%	7.7%
#LB=UB	356	341	380	380	361	360
#LB>LB*	-	-	67	85	43	43
#New opt	-	-	24	24	10	9

Chapitre 5

Résolution optimale du RCPSP par resolution search

En 1997, Vašek Chvátal présente une alternative aux procédures par séparation et évaluation classiques pouvant s'appliquer, en pqrticulier, à la résolution exacte des programmes linéaires en variables binaires ; il la nomme resolution search *[Chvátal 1997]. La méthode est proposée dans le but de rendre la résolution du problème moins largement dépendante de la stratégie de branchement. Ce faisant, Chvátal conçoit, dans un cadre plus général, une nouvelle forme de* backtracking intelligent *au sens de la programmation par contraintes (voir section 1.1.5). Resolution search est en effet basée sur le principe d'apprentissage : il s'agit, quand un échec est révélé à un noeud de l'« arbre » de recherche, d'identifier un sous-ensemble des contraintes associées au noeud responsable de l'échec, c.-à-d. l'instanciation de certaines des variables entrant en conflit. Éliminer cette instanciation partielle revient alors à générer une contrainte (un* nogood*) additionnelle au problème pour couper la recherche en amont du noeud considéré. Resolution search présente une manière élégante de gérer ces nogoods en limitant l'apprentissage, de façon à sauvegarder l'espace mémoire et à réduire le temps de traitement des nogoods et surtout, de façon à déterminer rapidement comment poursuivre la recherche à la suite d'un échec, tout en assurant la convergence de l'algorithme. En ce sens, il s'agit aussi d'une méthode optimale collaborative originale où la séparation de l'espace de recherche est gérée par un solveur logique et l'évaluation est traitée par un second solveur de programmation linéaire, comme décrit dans la méthode originale, ou par tout autre solveur (par exemple, un algorithme de filtrage PPC). À notre connaissance, cette procédure n'a jamais encore été expérimentée sur des problèmes théoriques ni pratiques. Ce chapitre est consacré à l'étude de resolution search et de ses applications à la résolution exacte du RCPSP. Nous commençons par présenter en détail cette procédure et ses spécificités, comparée aux PSE pour les programmes linéaires et aux backtrackings intelligents de la programmation par contraintes (5.1). Nous étudions ensuite son comportement en l'appliquant à une formulation linéaire du RCPSP en temps discretisé (5.2). Nous montrons alors comment resolution search peut aussi être utilisée adaptée à un schéma de branchement spécifique du RCPSP et proposons quelques améliorations de la procédure originale (5.3). Nous concluons ce chapitre par une comparaison de resolution search avec d'autres méthodes optimales de la PL ou de la PPC et par nos perspectives de recherche sur ce sujet (5.4).*

5.1 Resolution search

On cherche à résoudre un problème d'optimisation combinatoire *en variables binaires* de la forme :

$$z^* = \min\{f(x) \mid x \in X \subseteq \{0,1\}^n\}, \quad \text{avec } X \subseteq [0,1]^n. \tag{5.1}$$

Autrement dit, on recherche une instanciation complète des variables $x_1, \ldots, x_n$ à 0 ou 1 telle que le vecteur ainsi obtenu appartient à X et minimise la fonction f. On note $\bar{z}$ une évaluation par excès de z^*, la meilleure possible (éventuellement $\bar{z} = +\infty$).

La résolution de ce problème par énumération implicite consiste à montrer, de manière itérative, que toutes les sous-parties de l'espace de recherche initial (incluant l'ensemble des solutions) ne contiennent aucune solution du problème de valeur strictement inférieure à z^*. Il s'agit de « construire » une suite de sous-espaces, pour lesquels on a, soit prouvé directement par évaluation que les solutions qu'ils contiennent ne sont pas strictement meilleures que z^*, soit prouvé que tous leurs sous-espaces ont eux-mêmes cette propriété. La preuve d'optimalité de z^* est faite dès que l'espace de recherche entier est inclu dans cette suite.

Dans une recherche arborescente classique, la séparation de l'espace se fait sur les plans $x_i = 0$ et $x_i = 1$ de sorte que chaque sous-espace (ou chaque noeud de l'arbre de recherche) s'identifie à une instanciation partielle des variables binaires. La preuve d'optimalité s'obtient alors comme un cas particulier de la *preuve par résolution et réfutation* de la logique propositionnelle. Resolution search tire son nom de cette méthode car elle utilise plus en avant son principe général. Avant de poursuivre, nous introduisons ci-après les notations et définitions relatives à ce principe et qui seront utilisées tout au long de ce chapitre.

5.1.1 Notations et Préliminaires

Une instanciation partielle des variables binaires du problème 5.1 est représentée par un vecteur $u = (u_1, \ldots, u_n)$ de $\{0, 1, *\}^n$, où u_i correspond à l'instanciation de la variable x_i avec $u_i = *$ si la variable x_i est libre. La notion de sous-instanciation définit une relation d'ordre partielle sur $\{0, 1, *\}$, notée $\sqsubseteq$:

$$(u_1, \ldots, u_n) \sqsubseteq (v_1, \ldots, v_n) \text{ si } v_i = u_i \ \forall i \text{ tel que } u_i \neq *.$$

u est une *sous-instanciation* de v ; on dit encore que v est une *extension* de u (l'espace de recherche associé à v est inclu dans celui associé à u).

On considère uniquement les instanciations partielles dont l'espace associé est éliminé de la recherche. Une telle instanciation u peut donc être assimilée à une *clause*, puisque toutes les solutions x considérées dans la suite de la recherche sont telles que :

$$\bigvee_{i \mid u_i \neq *} x_i \neq u_i.$$

Pour faciliter l'écriture, on notera alternativement u, comme :

– une clause sous forme normale disjonctive :

$$\bigvee_{i \in I_0} x_i \vee \bigvee_{i \in I_1} \bar{x}_i$$

où les symboles x_i et $\bar{x}_i$, appelés *littéraux*, correspondent, dans leur forme positive x_i, aux indices $i \in I_0$ tels que $u_i = 0$ et, dans leur forme négative $\bar{x}_i$, aux indices $i \in I_1$ tels que

$u_i = 1$;
- un ensemble de littéraux :

$$\{x_i \mid i \in I_0\} \cup \{\bar{x}_i \mid i \in I_1\}.$$

Par exemple, l'instanciation $(*, 1, *, 0, 1, *)$ sera notée $\bar{x}_2 \vee x_4 \vee \bar{x}_5$ ou $\{\bar{x}_2, x_4, \bar{x}_5\}$ (ou encore parfois $\bar{x}_2 x_4 \bar{x}_5$), signifiant que, dans toute solution x de valeur strictement inférieure à $\bar{z}$, $x_2 = 0$ ou $x_4 = 1$ ou $x_5 = 0$.

La clause vide $(*, \ldots, *)$ est notée $\emptyset$ et correspond à l'espace de recherche initial. Pour tout littéral l, on note $\bar{l}$ sa *négation* avec $\bar{l} = \bar{x}_i$ si $l = x_i$ et $\bar{l} = x_i$ si $l = \bar{x}_i$.

L'opérateur de *résolvante* s'applique aux instanciations partielles. On rappelle que, si pour deux clauses u et u', il existe un unique littéral l tel que $l \in u$ et $\bar{l} \in u'$ alors on peut définir la clause résolvante $u \bigtriangledown u' = (u \setminus \{l\}) \cup (u' \setminus \{\bar{l}\})$.

Soit v une extension de $u \bigtriangledown u'$; si $l \in v$ alors v étend u, si $\bar{l} \in v$ alors v étend u', sinon v étend à la fois u et u'. On a donc la relation suivante, pour toute clause v :

$$u \bigtriangledown u' \sqsubseteq v \;\Rightarrow u \sqsubseteq v \;\vee\; u' \sqsubseteq v. \tag{5.2}$$

Mais la réciproque n'est pas nécessairement vérifiée. Par exemple, les clauses $u = (*, 1, *, 0, 1, *)$ et $u' = (1, 1, *, 1, *, 0)$ diffèrent sur la décision portant sur la variable x_4, leur résolvante est égale à $(1, 1, *, *, 1, 0)$. On vérifie facilement que toute extension de cette dernière est l'extension de u et/ou de u'. En revanche, $(0, 1, *, 0, 1, 1)$ est une extension de u mais pas de $u \bigtriangledown u'$. Dans cet exemple, la résolvante a la signification suivante : si u et u' sont éliminées de la recherche, alors toute solution x ne peut vérifier $x_2 = 1$ et $x_5 = 1$ simultanément à $x_1 = 1$, $x_2 = 1$ et $x_6 = 0$, car dans le premier cas $x_4 = 0$ d'après u, et dans le second, $x_4 = 1$ d'après u'.

Enfin, dans une recherche arborescente, on a à disposition une borne inférieure calculable à chaque noeud de l'arbre et croissante à chaque branchement. On note `oracle` une fonction qui associe à une instanciation partielle u une évaluation par défaut de la valeur de la meilleure solution réalisable pour 5.1 et contenue dans l'espace associé à u. On suppose que `oracle`(u) retourne la valeur $f(u)$ si u est une instanciation complète et que `oracle` est croissante sur l'espace partiellement ordonné $(\{0, 1, *\}^n, \sqsubseteq)$, c.-à-d. , $u \sqsubseteq v \Rightarrow$ `oracle(u)` $\leq$ `oracle(v)`.

5.1.2 Preuve par résolution en logique propositionnelle

Le principe de résolution est utilisé en logique propositionnelle pour prouver, compte tenu d'autres « contraintes », la véracité d'une proposition en réfutant celle-ci et, par résolvantes successives, en aboutissant à une contradiction. Ce principe s'étend à notre problème d'optimisation de la manière suivante :

Proposition 10 *Une* preuve par résolution et réfutation *[Robinson 1965] qu'une valeur z est la valeur optimale du problème 5.1 consiste en une suite $\mathcal{N} = N^1, \ldots, N^K$ de clauses telles que :*
- *$N^K = \emptyset$;*
- *pour tout indice $k \in \{1, \ldots, K\}$, soit `oracle`$(N^k) \geq z$, soit il existe deux indices i et j strictement inférieurs à k tels que $N^k = N^i \bigtriangledown N^j$.*

Preuve. Par un raisonnement par récurrence et par l'absurde, on vérifie que, connaissant une telle suite de clauses, on prouve qu'aucune solution du problème n'a de valeur strictement inférieure à z. Autrement dit, $z = z^*$ la valeur optimale de 5.1. En effet, supposons qu'il existe une solution x avec $f(x) =$ `oracle`$(x) < z$, alors x n'est l'extension d'aucune clause N^k telle que `oracle`$(N^k) \geq z$ (d'après la croissance de `oracle`). En particulier $N^1 \not\sqsubseteq x$. À une étape $k > 1$, en supposant que

$N^i \not\sqsubseteq x$ pour tout $i < k$, si $N^k = N^i \bigtriangledown N^j$ pour $i < k$ et $j < k$ alors, d'après (5.2), x ne peut être non plus une extension de N^k. Par récurrence, x n'est l'extension d'aucune clause de la suite, en particulier $N^K = \emptyset \not\sqsubseteq x$. De cette contradiction, on déduit que z est optimal. ■

5.1.3 Principe de résolution et méthodes d'énumération implicite

Pour plus de clarté, on suppose, dans les exemples donnés ci-dessous, que la séparation (branchement) s'effectue selon l'ordre lexico-graphique des variables.

Dans une PSE, les noeuds de l'arborescence qui sont coupés (les *noeuds terminaux*), l'un après l'autre et pris dans l'ordre chronologique, vérifient les conditions précédentes de la suite $\mathcal{N}$. En effet, un noeud u est coupé si $\texttt{oracle}(u) \geq \bar{z}$, où $\bar{z}$ est un majorant de z^*, ou bien si les deux noeuds fils de u ont précédemment été coupés. Dans ce dernier cas, u est bien la résolvante de ses noeuds fils. (Par exemple, $(1, 0, 1, *, *)$ est la résolvante de $(1, 0, 1, 0, *)$ et de $(1, 0, 1, 1, *)$.) Mais c'est un cas particulier car la sous-arborescence de u recouvre exactement l'union de la sous-arborescence de ses deux noeuds fils (il y a équivalence dans la relation (5.2)).

Les différentes formes de backtracking de la programmation par contraintes fonctionnent aussi sur le même principe, bien qu'ils s'appliquent, à l'origine, non pas à des problèmes d'optimisation mais de décision, et qu'ils utilisent des algorithmes de consistance pour couper l'espace de recherche. En fait, pour se ramener au cas décisionnel dans la preuve par résolution, il suffit de poser $\bar{z} = 1$ et faire correspondre $\texttt{oracle}(u)$ à 1 si l'instanciation partielle u est inconsistante avec les contraintes du problème, et à 0 sinon.

Pour les backtrackings intelligents (présentés à la section 1.1.5), les clauses ajoutées à la suite ne coupent pas seulement l'instanciation partielle courante mais une sous-instanciation de celle-ci ; il s'agit alors de nogoods. Ici le caractère local du maintien de la consistance joue en faveur de la programmation par contraintes pour l'identification des nogoods. En effet, en PPC, un échec se produit lorsqu'une contrainte est violée par l'instanciation courante. Il suffit alors de considérer les variables sur lesquelles porte la contrainte pour identifier un nogood. Par exemple, soit l'instanciation partielle courante $(1, 0, 1, 0, *)$. L'instanciation de x_5 à 0 rend inconsistante une contrainte $x_2 + x_5 \geq 1$ du problème. $(*, 0, *, *, 0)$ est donc un nogood. En supposant que x_5 ne peut être non plus instanciée à 1 à cause d'une contrainte $x_3 > x_5$, on génère alors le nogood $(*, *, 1, *, 0)$. Par résolvante sur x_5, on déduit que $(*, 0, 1, *, *)$ et toutes ses extensions sont des nogoods. Le backjumping considère uniquement l'extension $(1, 0, 1, *, *)$ et l'ajoute à la suite $\mathcal{N}$ des clauses, et poursuit la recherche sur $(1, 0, 0, *, *)$.

Le backjumping, comme le backtracking simple, ne nécessite pas d'expliciter les clauses de $\mathcal{N}$ car, à tout moment dans un parcours en profondeur, les informations précédemment inférées sont contenues dans la dernière clause N^k construite et donc dans l'instanciation couramment étudiée, obtenue à partir de N^k. Ceci est dû au fait que la résolvante s'effectue, comme en PSE, entre clauses pour lesquelles il y a équivalence dans la relation (5.2).

Au contraire, les backtrackings intelligents basés sur l'apprentissage nécessitent d'expliciter et de mémoriser, du moins partiellement, ces clauses car l'instanciation courante ne prend pas en compte toutes les clauses de $\mathcal{N}$ déjà construites. Dans l'exemple précédent, le nogood généré sera effectivement $(*, 0, 1, *, *)$, plus précise que $(1, 0, 1, *, *)$, mais la recherche se poursuivra aussi sur $(1, 0, 0, *, *)$. Le principe de résolvante est mieux utilisé puisqu'appliqué à des clauses plus dissymétriques encore, p. ex. , $(*, 0, 1, *, *) \bigtriangledown (*, *, 0, *, *) = (*, 0, *, *, *)$. Pour limiter le coût de cette apprentissage, le backtracking dynamique ne mémorise que les clauses pertinentes avec l'instanciation courante.

Resolution search procède de manière similaire aux backtrackings intelligents basés sur l'ap-

prentissage, en prenant aussi en compte le critère d'optimisation. Ce faisant, il n'y a plus moyen alors d'identifier aussi rapidement une sous-instanciation réellement en cause lors d'un échec $\texttt{oracle}(v) \geq \bar{z}$. Chvàtal propose de désinstancier progressivement chacune des variables de v (exceptée, naturellement, la dernière instanciée) et d'évaluer la borne inférieure à chaque étape. Partant de $u = v$, u est remplacé par $u \setminus \{l\}$, où l est un littéral correspondant à une variable instanciée, seulement si $\texttt{oracle}(u \setminus \{l\}) \geq \bar{z}$. On appelle ce processus, la *phase de remontée* (*waning phase* chez Chvátal), par opposition à la *phase de descente* (*waxing phase*) pour le processus de branchement. Au terme de cette remontée, $\texttt{oracle}(u) \geq \bar{z}$, et u est un nogood ajouté à la suite $\mathcal{N}$.

Comme le backtracking dynamique, resolution search se restreint, à tout instant, à la mémorisation d'une sous-famille F de clauses de $\mathcal{N}$ et gérée indépendemment. La différence fondamentale entre les deux méthodes réside dans le fait que resolution search ne garde pas les clauses en fonction de leur pertinence, mais à l'inverse, oriente la recherche en fonction des clauses mémorisées, permettant ainsi une plus grande mobilité encore dans l'espace de recherche.

Resolution search suit un processus itératif, où à chaque itération, la recherche s'effectue à partir d'un noeud $v = u_F$ obtenu à partir de F de sorte qu'aucune extension de u_F ne soit l'extension d'une clause de F (et donc déjà supprimée de la recherche). Comme dans une PSE classique, les branchements sont effectués progressivement sur les variables non-instanciées de v tant que $\texttt{oracle}(v) < \bar{z}$ (si v est totalement instanciée $\bar{z}$ est mise à jour avec $f(v) = \texttt{oracle}(v)$). On recherche alors, par remontées successives, une sous-instanciation u de v telle que $\texttt{oracle}(u) \geq \bar{z}$. u est alors utilisée pour mettre à jour la famille F, soit en étant simplement ajoutée à la suite des autres clauses de la famille, soit, si $v \sqsubseteq u_F$, en effectuant la résolvante entre u et certaines clauses de F puis en supprimant dans F, des clauses partiellement redondantes. Dans ce second cas, certaines solutions précédemment ignorées dans la recherche grâce à F, pourront de nouveau être reconsidérées aux itérations suivantes. Cependant, la structure très particulière imposée à F, à chaque étape, permet d'assurer la convergence de l'algorithme tout en permettant de recalculer rapidement u_F à la suite d'une mise-à-jour de F.

Cette structure, appelée *path-like*, sur laquelle repose l'essentiel de la procédure resolution search, ne se présente pas sous un formalisme très intuitif. Pour mieux appréhender la procédure, avant de la détailler, nous proposons de montrer son déroulement sur un exemple simple.

5.1.4 Premier exemple

On suppose que le problème possède 4 variables binaires x_1, x_2, x_3 et x_4, on connait une borne supérieure $\bar{z} = 1$ et une borne inférieure `oracle` retournant 0 tant que les variables x_2 et x_4 ne sont pas toutes deux instanciées, et 1 sinon. Autrement dit, toute solution réalisable a pour valeur 1. Pour prouver l'optimalité de 1, on branche sur les variables dans l'ordre naturel, en les fixant à 0 puis à 1. L'originalité de resolution search tient aussi dans le fait que la convergence est assurée quel que soit le point de départ de la recherche.

1ère itération : On débute par exemple au noeud $u = (1, 0, *, *)$ et le premier appel à `oracle` retourne donc 0. Comme dans une PSE classique, on branche sur les variables non instanciées tant que $\texttt{oracle}(u) < 1$. On considère donc $u = (1, 0, 0, *)$ puis $u = (1, 0, 0, 0)$. Ici $\texttt{oracle}(u) = 1$, on cherche à identifier une sous-instanciation de u, minimale pour la relation $\sqsubseteq$, telle que $\texttt{oracle}(u) \geq 1$. Pour cela, on désinstancie chacune des variables (sauf la dernière x_4) dans l'ordre inverse d'instanciation. On calcule donc $\texttt{oracle}(1, 0, *, 0) = 1$, puis $\texttt{oracle}(1, *, *, 0) = 0$ (x_2 doit être réinstanciée à 0), puis $\texttt{oracle}(*, 0, *, 0) = 1$. Ne pouvant plus remonter, on mémorise ce nogood dans la famille F initialement vide,

$F = \{C^1 = (*, 0, *, 0)\}$, en lui associant un littéral l^1, disons $x_4 = 0$ la dernière instanciation qui a causé l'échec.

2nde itération : On débute la seconde itération du processus à partir de ce noeud C^1 mais avec la négation de l^1 (c.-à-d. en inversant la décision sur x_4) : $u = u_F = (*, 0, *, 1)$. Comme $\texttt{oracle}(u) = 1$, on remonte directement en désinstanciant x_4 ($\texttt{oracle}(*, 0, *, *) = 0$) puis x_2 ($\texttt{oracle}(*, *, *, 1) = 0$). u est donc minimal pour la condition, il est alors ajouté à F. En fait, par construction, comme aucun branchement n'a été effectué à cette itération ($u \sqsubseteq u_F$), on peut calculer la résolvante entre u et C^1 sur le littéral l^1 : $(*, 0, *, 1) \bigtriangledown (*, 0, *, 0) = (*, 0, *, *)$. Par définition de la résolvante, toute solution étendant $(*, 0, *, *)$ est nécessairement de valeur égale à 1, puisqu'elle étend soit u, soit C^1. Dans ce cas particulier, la résolvante est même une sous-instanciation de C^1, elle la remplace donc dans F : $F = \{C^1 = (*, 0, *, *)\}$ et le littéral associé est naturellement $l^1 = (x_2 = 0)$.

3ème itération : On prend la négation de l^1, en partant de $u = u_F = (*, 1, *, *)$, puis on branche jusqu'à $(0, 1, 0, 0)$. On remonte alors en défixant successivement, dans l'ordre inverse, x_3, x_1, puis x_2. On obtient ainsi le nogood $u = (*, 1, *, 0)$ qui est simplement ajouté à F : $F = \{C^1 = (*, 0, *, *), C^2 = (*, 1, *, 0)\}$, avec les littéraux associés $l^1 = (x_2 = 0)$ et $l^2 = (x_4 = 0)$. (On ne peut pas choisir pour la seconde clause C^2 le littéral associé à x_2 car celui-ci est déjà associé à C^1.)

4ème itération : u_F est obtenu en prenant la négation de l^1 dans C^1 et l^2 dans C^2 et en effectuant la réunion des deux clauses : $u_F = (*, 1, *, 1)$. De même qu'à la seconde itération, il est inutile de brancher à ce noeud et la remontée indique que u_F est un nogood minimal. On met à jour F de la façon suivante : Soit $u = u_F$, comme $\bar{l}^2 \in u$, par construction, on peut effectuer la résolvante entre u et C^2 et on pose $u = u \bigtriangledown C^2 = (*, 1, *, *)$. Comme $\bar{l}^1 \in u^1$, on fait de même avec C^1 : $u = u \bigtriangledown C^1 = (*, *, *, *)$. À ce moment, on a prouvé que toute solution réalisable (étendant $(*, *, *, *)$) est de valeur au mieux égale à 1 et l'algorithme se termine.

La suite des nogoods construits, par évaluation ou par résolvante, vérifie bien les conditions de $\mathcal{N}$ et fournit donc une preuve de l'optimalité de 1 :

$(*, 0, *, 0)$,	$(*, 0, *, 1)$,	$(*, 0, *, *)$,	$(*, 1, *, 0)$,	$(*, 1, *, 1)$,	$(*, 1, *, *)$,	$(*, *, *, *)$
N^1	N^2	$N^1 \bigtriangledown N^2$	N^4	N^5	$N^4 \bigtriangledown N^5$	$N^6 \bigtriangledown N^3$

Ici, 17 appels à `oracle` ont été nécessaires à la preuve d'optimalité : 9 en phase de descente ($3+1+4+1$ pour chaque itération) et 8 en remontée ($3+2+2+0$). À titre indicatif, il en aurait fallu 19 en partant de la racine ou 14 en partant directement de N^1 à la première itération. En comparaison, la PSE avec le même branchement, aurait dû évaluer tous les noeuds de l'arborescence (31 appels) et une recherche type backtracking dynamique aurait nécessité 27 appels à `oracle` (15 sans les remontées). À noter d'ailleurs, que la preuve par backtracking dynamique consiste ici en exactement la même suite de clauses $\mathcal{N}$ que pour la preuve par resolution search.

5.1.5 Gestion de la famille des nogoods

Dans cette section, nous définissons formellement la structure de la famille F, la clause u_F associée, et comment F est mise à jour après la découverte d'un nouveau nogood u.

Définition.

– Soit F une famille de clauses ordonnées $C^1, C^2, \ldots, C^M$ et $l^1, l^2, \ldots, l^M$ des littéraux associés. La famille F est dite *path-like* si les conditions suivantes sont vérifiées :

- $l^i \in C^j$ si et seulement si $i = j$;
- si $\bar{l}^i \in C^j$ alors $j > i$;
- pour tout littéral l, si $l \in C^i$ et $\bar{l} \in C^j$ alors $l = l^i$ ou $l = l^j$.

– À une telle famille, on peut associer une clause u_F définie par :

$$u_F = \big(\bigcup_{k=1}^{M} (C^k \cup \bar{l}^k \setminus \{l^k\}) \big).$$

En d'autres termes, les conditions sur F spécifient que le littéral associé à chaque clause n'apparaît que dans cette clause ; que sa négation n'appartient pas aux clauses précédentes ; et, pour toute variable x_i ne correspondant pas à un littéral l^k, on ne peut avoir $x_i = 0$ dans une clause et $x_i = 1$ dans une autre.

On vérifie facilement avec ces propriétés que la clause u_F est bien définie, autrement dit, qu'il n'existe pas de littéral l tel que $l \in u_F$ et $\bar{l} \in u_F$. De plus, aucune extension de u_F ne peut être une extension d'une clause C^k (car $\bar{l}^k \in u_F$ et $l^k \in C^k$).

La famille F qui est gérée tout au long de l'algorithme de resolution search vérifie cette propriété. Elle n'est constituée que de clauses C, telle que toute solution x étendant C a une valeur $f(x) = oracle(x)$ supérieure ou égale à la meilleure borne supérieure $\bar{z}$ connue au même instant (on rappelle que $\bar{z}$ est mise à jour dès que l'on connaît une solution réalisable x avec $\texttt{oracle}(x) < \bar{z}$). Pour cela, comme pour la proposition 10, seules les opérations suivantes sont autorisées sur F :

- une clause C peut être ajoutée à F si $\texttt{oracle}(C) \geq \bar{z}$;
- si on peut effectuer la résolvante de deux clauses C et C' de F, alors on peut ajouter $C \bigtriangledown C'$ à F ;
- une clause C peut être supprimée de F.

Plus précisément, soit $\bar{z}$ la borne supérieure courante et soit u la clause obtenue au terme d'une itération de resolution search. Comme il a été expliqué précédemment, il existe alors une clause v telle que $u_F \sqsubseteq v$ (après la phase de descente à partir de u_F) et $u \sqsubseteq v$ (après la phase de remontée) et $\texttt{oracle}(v) \geq \texttt{oracle}(u) \geq \bar{z}$. La structure de F est maintenue principalement grâce à l'observation suivante :

$$l \in u_F \Rightarrow \bar{l} \notin u, \text{ et inversement.}$$

En effet, comme $u_F \sqsubseteq v$ et $u \sqsubseteq v$ alors toute variable intanciée dans v est, dans u_F ou dans u, soit instanciée à la même valeur, soit non instanciée.

On distingue deux cas pour la mise à jour de F à partir de u, selon que u est une sous-instanciation de u_F (aucun littéral n'a été ajouté dans la phase de descente : $u \sqsubseteq u_F = v$), ou non.

Après une phase de descente

Dans le second cas, $u \not\sqsubseteq u_F$, il existe un littéral $l \in u \setminus u_F$. La famille F est alors simplement étendue :

$$M = M + 1, \quad C^M = u \quad \text{et } l^M = l.$$

Lemme 11 *À la suite de cette mise à jour, F est encore* path-like.

Preuve. Les trois points suivants prouvent que F avec la nouvelle clause C^M vérifie chacune des trois conditions de la définition :

- Pour tout $k < M$, $\bar{l}^k \in u_F \sqsubseteq v$ et $C^M = u \sqsubseteq v$ donc $l^k \notin C^M$. Inversement, par construction ($l^M \in v \setminus u_F$), l^M ne correspond à aucune variable instanciée dans une clause C^k, pour tout $k < M$, donc $l^M \notin C^k$.
- Pour cette même raison $\bar{l}^M$ n'appartient pas non plus à un C^k tel que $k < M$.
- Enfin, soit l un littéral d'une clause C^k, $k < M$, différent de l^k. Alors, $l \in v$ donc $\bar{l} \notin C^M$.

■

Sans phase de descente

L'autre cas, $u \sqsubseteq u_F$ se produit si $\texttt{oracle}(u_F) \geq \bar{z}$. On peut à ce moment là réduire la famille F quitte à supprimer, éventuellement, des informations. La mise à jour de F suivante garantit va cependant garantir la convergence de l'algorithme :

Par définition de u_F et comme $u \sqsubseteq u_F$, si la variable associée à un littéral l^k n'a pas été désinstanciée dans la phase de remontée, elle correspond alors à $\bar{l}^k$ dans la clause u. De plus, les autres variables sont, soit instanciées à la même valeur dans u et dans C^k, soit non instanciées dans l'une des deux clauses. On peut donc « réduire » u par résolvante avec C^k et

$$u \bigtriangledown C^k \sqsubseteq \bigcup_{i=1}^{M} (C^i \setminus \{l^i\}) \cup C',$$

où C' est l'ensemble des littéraux $\bar{l}^i$, $i \neq k$, qui n'ont pas été supprimés lors de la remontée. De plus, toute extension de $u \bigtriangledown C^k$ ne peut conduire à une solution de valeur strictement meilleure que $\bar{z}$.

Pour maintenir la structure de F, on procède itérativement sur les clauses de F dans l'ordre inverse $C^M, \ldots, C^1$: On initialise $C = u$, et pour tout k de M à 1, si $\bar{l}^k \in C$, on pose $C = C \bigtriangledown C^k$. Au terme de ce processus, on a donc désinstancié, dans C, toutes les variables associées aux littéraux $\bar{l}^k$, et tout littéral de C apparaît au moins dans une clause C^k et donc :

$$C \sqsubseteq \bigcup_{i=1}^{M} (C^i \setminus \{l^i\}). \tag{5.3}$$

Si $C = \emptyset$, la procédure complète de resolution search s'arrête puisqu'on prouve ainsi que $\bar{z}$ est la valeur optimale du problème. Sinon, on recherche le plus petit indice k telle que C est incluse dans la réunion des k premières clauses, et l un littéral de C appartenant à C^k mais à aucune autre clause précédente :

$$C \sqsubseteq \bigcup_{i=1}^{k} (C^i \setminus \{l^i\}),\ l \in C^k,\ l \notin C^i,\ \forall i \in \{1, \ldots, k-1\}.$$

On remplace alors la clause C^k de F, en posant $C^k = C$ et en lui associant le littéral $l^k = l$. Les conditions 2 et 3 de la définition de la structure *path-like* sont alors encore vérifiées par F car d'après (5.3), pour tout $i \neq k$: C^k ne contient aucun littéral l^i ou $\bar{l}^i$, $\bar{l}^k$ n'appartient à aucune clause C^i, et $l \in C^k \setminus \{l^k\} \Rightarrow \bar{l} \notin C^i \setminus \{l^i\}$. Enfin, pour conserver la condition 1, il est nécessaire de supprimer toutes les clauses C^i avec $i > k$ et qui contiennent le littéral l^k.

La famille F est donc encore *path-like*. En recalculant u_F la clause associée, il est éventuellement possible de continuer à mettre à jour F avec u au cas où $l^k \notin u$ (l^k a été ajouté à C par résolvante avec une autre clause C^i). En effet, sous cette condition, $u \sqsubseteq v$ où v est l'ancienne clause u_F dans laquelle le littéral l^k est remplacé par $\bar{l}^k$. On montre que tout littéral l dans la nouvelle clause u_F

appartient à v : Toute clause de la nouvelle famille F appartient à l'ancienne, exceptée C^k qui est incluse dans la réunion des $k-1$ premières clauses et de l'ancienne clause C^k. Autrement dit, tout littéral dans $u_F \setminus \{\bar{l}^k\}$ appartient à l'ancienne clause u_F. Donc $u_F \sqsubseteq v$ et comme $u \sqsubseteq v$, les conditions sont réunies pour réitérer le processus de mise à jour de F.

L'algorithme 4 présente formellement ce processus de mise à jour de la famille F *path-like* compte tenu de la découverte d'une nouvelle clause u, telle qu'il existe v avec $u_F \sqsubseteq v$, $u \sqsubseteq v$ et $\texttt{oracle}(v) \geq \bar{z}$.

Algorithme 4 – Resolution search : mise à jour de la famille *path-like* F avec u

continue=VRAI
while continue **faire**
  si $u \not\sqsubseteq u_F$ **alors**
    choisir $l \in u \setminus u_F$
    $M = M+1, C^M = u, l^M = l$
  sinon
    $C = u$
    pour k **de** M **à** 1 **faire**
      si $l^k \in C$ **alors**
        $C = C \bigtriangledown C^k$
      fin si
    fin pour
    si $C = \emptyset$ **alors**
      STOP ($\bar{z}$ *optimal*)
    fin si
    $M' = M$, $M = \min\{k \mid C \subseteq C^1 \cup \cdots \cup C^k\}$
    choisir $l \in C \setminus C^1 \cup \cdots \cup C^{M-1}$
    $C^M = C$, $l^M = l$
    pour k **de** $M+1$ **à** M' **faire**
      si $l^M \notin C^k$ **alors**
        $M = M+1$, $C^M = C^k$, $l^M = l^k$
      fin si
    fin pour
    si $l \in u$ **alors**
      continue=FAUX
    fin si
  fin si
fin while

Exemple

Pour illustrer le fonctionnement de cet algorithme, nous reprenons ci-dessous un des exemples proposés dans [Chvátal 1997].

Soit F la famille *path-like* suivante :

$$\begin{array}{ll} C^1 = x_3 & l^1 = x_3, \\ C^2 = \bar{x}_2\bar{x}_4\bar{x}_6 & l^2 = \bar{x}_4, \\ C^3 = \bar{x}_2\bar{x}_5\bar{x}_6 & l^3 = \bar{x}_5, \\ C^4 = \bar{x}_2\bar{x}_6x_7 & l^4 = x_7, \\ C^5 = \bar{x}_1\bar{x}_2\bar{x}_3 & l^5 = \bar{x}_1, \\ C^6 = x_1x_5x_8 & l^6 = x_8. \end{array}$$

Soit $u = \bar{x}_3\bar{x}_6\bar{x}_8$ la clause considérée pour mettre à jour F.

Comme u est incluse dans $u_F = x_1\bar{x}_2\bar{x}_3x_4x_5\bar{x}_6\bar{x}_7\bar{x}_8$, on commence par calculer la clause C à ajouter à la famille F :

$$\begin{aligned} C &= u \bigtriangledown C^6 = x_1\bar{x}_3x_5\bar{x}_6, \\ C &= C \bigtriangledown C^5 = \bar{x}_2\bar{x}_3x_5\bar{x}_6, \\ C &= C \bigtriangledown C^3 = \bar{x}_2\bar{x}_3\bar{x}_6, \\ C &= C \bigtriangledown C^1 = \bar{x}_2\bar{x}_6. \end{aligned}$$

Comme $C \subseteq C^1 \cup C^2$, on remplace C^2 par C et on choisit le littéral associé, par exemple $\bar{x}_2$. Ainsi, la nouvelle famille F ne contient plus que trois clauses : $C^1 = x_3, C^2 = \bar{x}_2\bar{x}_6, C^3 = x_1x_5x_8$, avec les littéraux $l^1 = x_3$, $l^2 = \bar{x}_2$ et $l^3 = x_8$.

Bien que $\bar{x}_6$ convenait aussi comme littéral associé à C^2, il est plus judicieux de choisir $\bar{x}_2$ qui n'appartient pas à u. On peut donc poursuivre la mise à jour de F. Comme $u \sqsubseteq u_F = x_1x_2\bar{x}_3x_5\bar{x}_6\bar{x}_8$, de nouveau, on peut réduire u par résolvantes successives :

$$\begin{aligned} C &= u \bigtriangledown C^3 = x_1\bar{x}_3x_5\bar{x}_6, \\ C &= C \bigtriangledown C^1 = x_1x_5\bar{x}_6, \end{aligned}$$

et on remplace C^3 par C avec le littéral associé $l^3 = x^1$ par exemple (x_5 convient aussi, mais pas $\bar{x}_6 \in C^2$). La famille F contient maintenant les trois clauses $C^1 = x_3, C^2 = \bar{x}_2\bar{x}_6, C^3 = x_1x_5\bar{x}_6$, avec les littéraux associés $l^1 = x_3$, $l^2 = \bar{x}_2$ et $l^3 = x_1$.

On peut mettre à jour F une dernière fois car $x_1 \notin u$ et $u \not\sqsubseteq u_F = \bar{x}_1x_2\bar{x}_3x_5\bar{x}_6$. On ajoute donc u comme la quatrième clause de F en y associant le littéral $x_8 \in u \setminus u_F$.

Au terme de l'algorithme, la famille F est donc égale à :

$$\begin{array}{ll} C^1 = x_3 & l^1 = x_3, \\ C^2 = \bar{x}_2\bar{x}_6 & l^2 = \bar{x}_2, \\ C^3 = x_1x_5\bar{x}_6 & l^3 = x_1, \\ C^4 = \bar{x}_3\bar{x}_6\bar{x}_8 & l^4 = \bar{x}_8. \end{array}$$

5.1.6 Preuve de convergence

On remarque dans l'exemple précédent, que l'ensemble des solutions étendant $\bar{x}_1\bar{x}_2\bar{x}_3x_6$ étaient précédemment interdites par les clauses de F (plus précisément par C^5). Ce n'est plus le cas après la mise à jour. Il arrive ainsi que l'on regénère plusieurs fois certaines clauses de la preuve $\mathcal{N}$. Cependant, la procédure est assurée de converger.

Théorème 12 *Pour la recherche et la preuve d'optimalité du problème 5.1 par resolution search, la famille F est mise à jour au plus 2^n fois.*

On trouvera une démonstration plus formelle de ce théorème dans [Chvátal 1997]. Nous indiquons brièvement les principaux éléments de la preuve.

Le nombre $N(F)$ de vecteurs de $\{0,1\}^n$ qui étend au moins une clause de F n'est pas croissant. En revanche, il existe une borne inférieure de $N(F)$ qui, elle, croît strictement à chaque itération de la boucle principale de l'algorithme 4. On considère n_k le nombre de variables non instanciées dans aucune des k premières clauses de F. Pour toute famille F de clauses non vides, la suite des entiers $n_1, \ldots, n_M$ est strictement décroissante et varie entre $n-1$ et 0. La *force* de F, $\sigma(F) = \sum_{k=1}^{M} 2^{n_k}$

est donc bornée par 2^n, le cardinal de $\{0,1\}^n$ (en fait, $\sigma(F) \leq N(F)$). Il suffit donc de montrer que $\sigma(F)$ augmente strictement à chaque mise à jour de F pour prouver le théorème. C'est évident au cas où la nouvelle clause u est ajoutée à F (si $u \not\sqsubseteq u_F$). Dans l'autre cas ($u \sqsubseteq u_F$), une clause de F, disons C^k, est remplacée par C. Le nombre de variables non instanciées par les nouvelles k premières clauses $(C^1, \ldots, C^{k-1}, C)$ de F est strictement supérieur à l'ancienne valeur n_k puisque $C \sqsubseteq \cup_{i=1}^{k} C^i$ et que la variable associée au littéral $l^k \in C^k$ n'est plus instanciée dans aucune de ces clauses. Ainsi, la force de de ces clauses est, seule, strictement supérieure à la force de l'ancienne famille F :

$$\sigma(C^1, \ldots, C^{k-1}, C) \geq \sum_{i=1}^{k-1} 2^{n_i} + 2.2^{n_k} > \sigma(F).$$

Bien que l'ordre des clauses a une influence sur la force de F, Chvátal fait remarquer que, sans même considérer si la structure *path-like* de F est conservée ou non, déterminer l'ordre qui maximise la force est en général un problème $NP-$difficile.

5.2 Application basique à la formulation du RCPSP en temps discrétisé

Plusieurs travaux citent la procédure resolution search de Chvàtal, en comparaison théorique avec d'autres nouvelles méthodes de programmation linéaire [Hanafi 2002, Codato 2003] ou dans l'étude des backtrackings intelligents pour la résolution de problèmes de satisfiabilité [Bayardo 1996]. Néanmoins, nous n'avons rencontré dans la littérature aucune application de resolution search.

Nous avons donc décidé, dans un premier temps, de tester la méthode telle que décrite par Chvàtal en la comparant à une PSE strictement équivalente. Nous l'avons ainsi appliquée à la formulation linéaire du RCPSP en temps discrétisé de Pritsker et al. [Pritsker 1969] présentée à la section 2.3.2. On rappelle que ce programme linéaire contient des variables binaires y_{it}, pour toute activité $i \in V$ et pour tout temps $t \in \mathcal{T}$, définies par $y_{it} = 1$ si l'exécution de l'activité i débute au temps t, et $y_{it} = 0$ sinon.

5.2.1 Schéma d'application

Les deux procédures, resolution search et PSE, que nous avons implémenté à partir de ce modèle reposent sur un schéma strictement identique. Nous détaillons ci-dessous les différents ingrédients que nous avons mis en oeuvre.

Séparation de l'espace de recherche

On considère l'espace $\{0,1\}^{(n+2)(T+1)}$, l'ensemble des instanciations possibles pour les variables y_{it}. Chaque branchement correspond à l'instanciation d'une unique variable y_{it} à 0 ou 1. Un noeud du graphe de recherche correspond donc à une instanciation partielle :

$$(u_{00}, \ldots, u_{0T}, u_{10}, \ldots, u_{1T}, \ldots, u_{(n+1)0}, \ldots, u_{(n+1)T}),$$

où, pour toute activité $i \in V$ et pour tout temps $t \in \mathcal{T}$, le littéral $u_{it} \in \{0, 1, *\}$ correspond à l'instanciation courante de la variable y_{it}.

Évaluation par défaut

À chaque noeud u du graphe de recherche, nous obtenons une borne inférieure de la meilleure solution contenue en résolvant, par l'algorithme dual du simplexe, la relaxation continue du PLVB correspondant. En d'autres termes, la fonction `oracle` est définie comme associant à tout noeud u, soit $+\infty$ si le programme suivant est irréalisable, soit, sinon, la valeur entière arrondie par excès de :

$$\min \sum_{t \in \mathcal{T}} t y_{(n+1)t}$$

sujet à :

$$\begin{aligned}
&\sum_{t \in \mathcal{T}} y_{it} = 1 && \forall\, i \in V \\
&\sum_{t \in \mathcal{T}} t(y_{jt} - y_{it}) \geq p_i && \forall\, (i,j) \in E \\
&\sum_{i \in V} r_{ik} \sum_{\tau = t - p_i + 1}^{t} y_{i\tau} \leq R_k && \forall\, k \in \mathcal{R}, \forall\, t \in \mathcal{T} \\
&y_{it} \in \{0,1\} && \forall\, i \in V, \forall\, t \in \mathcal{T} \\
&y_{it} = u_{it} && \text{si } u_{it} \neq *
\end{aligned}$$

Cette borne inférieure est comparée alors avec la valeur $\bar{z}$ de la meilleure solution connue. Si $\texttt{oracle}(u) \geq \bar{z}$ ou si u est une instanciation complète, on effectue un backtrack sur la dernière variable instanciée dans le cas de la PSE (parcours en profondeur d'abord), ou on procède à la phase de remontée dans resolution search. Sinon, on instancie une nouvelle variable y_{it} telle que $u_{it} = *$. À noter que l'on considère aussi le cas où la solution de la relaxation continue est entière pour directement mettre à jour la meilleure solution connue et donc effectuer la remontée. Il s'agit pour resolution search, d'une légère amélioration de la procédure telle que décrite par Chvátal.

Stratégies de branchement (instanciation) et de désinstanciation

Nous ne prenons pas en compte la structure du problème pour le choix de la future variable à instancier, puisque nous sélectionnons simplement la variable la plus fractionnaire dans la solution de la relaxation continue et branchons sur la valeur entière la plus proche (à 1 si la valeur fractionnaire est à $0,5$).

Dans resolution search, l'identification d'un nogood minimal u coupant le noeud v courant peut s'effectuer de multiples façons. Nous avons implémenté l'heuristique proposée par Chvátal : Partant de v ($\texttt{oracle}(v) \geq \bar{z}$), on défixe l'une après l'autre les variables instanciées dans v, exceptée la dernière ajoutée dans la phase de descente, qui appartient nécessairement à tout nogood $u \sqsubseteq v$. On construit ainsi u en posant $u = v$, puis itérativement, pour chaque littéral $l \in v$ (sauf le dernier), $u = u \setminus \{l\}$ si $\texttt{oracle}(u \setminus \{l\}) \geq \bar{z}$. Selon l'ordre choisi de suppression des littéraux, la remontée n'aboutit pas nécessairement à un même nogood u. Comme indiqué par Chvátal, il peut être préférable que les clauses de la famille recouvrent un maximum de variables communes. De cette façon, moins de littéraux sont ajoutés à C par les résolvantes successives $C \bigtriangledown C^k$, le nombre de clauses dans la famille est plus certainement limité, et surtout, les clauses toutes ensemble recouvrent un nombre moindre de variables et donc maximisent la force, $\sigma(F)$ définie section 5.1.6, de la famille. Pour cela, les littéraux de v, ajoutés lors de la phase de descente, sont testés en premier dans l'ordre inverse de branchement, et sont ainsi plus probablement supprimés. Les autres littéraux

(appartenant à u_F) sont évalués dans l'ordre lexico-graphique inverse.

En fait, un choix de branchement implicite intervient au moment de sélectionner le littéral associé à la nouvelle clause entrante dans F, puisque le parcours du graphe de recherche se poursuit alors le long de la décision inverse. Au cas où au moins un littéral est ajouté lors de la phase de descente, on mémorise le dernier, l cause de l'échec et non supprimé dans la remontée, car il satisfait la condition $l \in u \backslash u_F$. Autrement, on choisit trivialement le premier littéral de la clause qui satisfait aux conditions requises. Dans notre implémentation actuelle, il s'agit du premier rencontré dans l'ordre lexico-graphique.

Prétraitement

Pour accélerer les deux procédures, PSE et resolution search, nous utilisons un simple prétraitement du programme linéaire au préalable de la recherche en calculant les fenêtres de temps des activités par un algorithme de consistance d'arcs aux bornes sur les contraintes de précédence du projet (algorithmes de `bellman`, voir section 2.2.1). De cette façon, on diminue de manière importante le nombre de variables du programme ($y_{it} = 0$ si $t < ES_i$ ou si $t > LS_i$) et donc le nombre de branchements possibles. Pour resolution search, cette technique a un autre avantage puisqu'elle diminue d'autant la taille des clauses de la famille F.

Solution réalisable initiale

Enfin, pour initialiser la valeur T du programme linéaire (et donc pour appliquer le prétraitement), on calcule au tout début de chacune des deux procédures, une solution réalisable S de l'instance du RCPSP au moyen de l'heuristique tabou de [Baar 1998]. La durée de cet ordonnancement est aussi utilisée comme borne supérieure initiale $\bar{z}$. Puisque l'on recherche une solution strictement meilleure que S, on pose $T = \bar{z} - 1$ et on effectue le prétraitement à partir de cette valeur T. Comme il a été dit précédemment et contrairement à une PSE, il est possible avec resolution search de démarrer la recherche à partir de n'importe quel noeud. Dans cette procédure, nous faisons partir la recherche directement du noeud terminal S. Plus précisément, nous commençons au noeud correspondant à la seule instanciation des variables y_{iS_i} à 1, pour toute activité réelle $i \in \mathcal{A}$.

5.2.2 Résultats expérimentaux

L'expérimentation comparative des deux procédures a été menée sur un Pentium III 800MHz avec 384Mb de RAM sous Debian/Linux et g++ 2.95.4. Les programmes ont été écrits en C++ en utilisant ILOG CPLEX 7.0 pour la résolution des relaxations continues, et les parties communes des méthodes sont implémentées par un même code.

Nous avons comparé la PSE et resolution search sur les 287 instances du RCPSP à 30 activités de la PSPLIB, pour lesquelles la solution réalisable de départ, obtenue par l'heuristique, n'est pas prouvée optimale par le prétraitement.

La table 5.1 fournit une comparaison des meilleures solutions réalisables déterminées par chacune des méthodes dans un temps de calcul limité à 30 minutes. Ici, BSup désigne la meilleure valeur trouvée par l'heuristique tabou (ligne 1), la PSE (ligne 2) et resolution search (ligne 3), comparée à la valeur optimale (opt) du problème. Chaque colonne représente, dans l'ordre :

– le nombre d'instances pour lesquelles la méthode détermine une solution optimale (sans prouver nécessairement son optimalité) ;

– le nombre d'instances pour lesquelles la PSE ou resolution search détermine une solution réalisable meilleure que la solution initiale donnée par l'heuristique tabou ;
– la déviation moyenne et maximale de la meilleure solution trouvée par rapport à l'optimum.

TAB. 5.1 – Étude comparative de resolution search et de PSE : solutions réalisables

	#BSup=opt	#BSup<tabou	Dev BSup/opt moy	(max)
tabou	86	-	3,67%	(16,67%)
PSE	118	42	3,27%	(16,67%)
RS	**157**	**89**	**2,67%**	(16,67%)

L'efficacité de resolution search, comparativement à la PSE, est évidente au vu de ces résultats. Sur les 201 instances pour lesquelles la solution initiale n'est pas optimale, resolution search permet de déterminer une meilleure solution pour 89 instances, soit 47 de plus que la PSE. Ce résultat n'apparaît pas dans cette table, mais pour ces 201 instances, la meilleure solution trouvée par resolution search est toujours au moins aussi bonne que celle trouvée au terme de la PSE, et même strictement meilleure dans 56 cas.

La table 5.2 présente maintenant les résultats obtenus sur les instances pour lesquelles les méthodes fournissent une preuve d'optimalité dans un temps de calcul limité à 30 minutes. Les

TAB. 5.2 – Étude comparative de resolution search et de PSE : preuve d'optimalité

	#opt	CPU moy	(max)	# noeuds moy	(max)
PSE	82	219,6	(1723)	70110	(616045)
RS	**141**	101,7	(1395)	45956	(654826)
RS	82	**5,7**	(92)	**3 413**	(68695)

colonnes désignent, dans l'ordre : le nombre d'optima prouvés, le temps de calcul moyen et maximal (en secondes) sur les instances dont l'optimum est prouvé et le nombre de noeuds visités (appels à `oracle`) moyen et maximal sur ces mêmes instances. La première ligne correspond aux résultats de la PSE, la seconde aux résultats de resolution search, et la troisième aux résultats de resolution search sur les 82 instances prouvées optimales par la PSE.

Resolution search domine, de nouveau, très largement la PSE et ce, sur tous les critères. En effet, dans la limite de temps accordée, resolution search prouve l'optimalité pour davantage d'instances que la PSE (141 contre 82) et ce, en moyenne, en moitié moins de temps (102 s. contre 220 s.) et en beaucoup moins d'appels à la résolution linéaire (1/3 de noeuds évalués en moins).

Sur les 82 instances pour lesquelles la PSE parvient à prouver l'optimalité en 30 minutes, resolution search résoud de même toutes ces instances en un temps et un nombre d'itérations près de 20 fois moindre.

L'effort supplémentaire fourni par resolution search pour la recherche des nogoods et pour la gestion explicite de l'espace de recherche est donc clairement rentable. Nous attribuons aussi la puissance de resolution search au fait que, avec cette méthode contrairement à la PSE, la solution initiale, et non seulement sa valeur, est totalement exploitée. Dans cet exemple d'application au RCPSP, partant d'une solution « quasi-optimale », à la première remontée, resolution search

recherche en fait les seules activités du projet dont la date de début actuelle influe sur la durée d'ordonnancement. Elle permet ainsi de déterminer rapidement une solution voisine, à la manière d'une recherche locale, mais avec l'assurance d'atteindre et de prouver l'optimalité. En cela, elle présente des similitudes avec l'approche *local branching* de Fischetti et Lodi [Fischetti 2003].

Nous présentons ces résultats comparatifs principalement dans le but de montrer le potentiel global de la procédure resolution search par rapport à une PSE classique, ce qui n'avait, à notre connaissance, jamais été fait avant. Dans ces expérimentations, nous n'avons pas exploité les particularités du RCPSP. Resolution search semble donc réellement avantageuse, du moins, en se basant sur ces résultats, quant il s'agit de résoudre des problèmes peu ou pas structurés ou bien sur lesquels on ne dispose d'aucune information. Cette méthode pourrait ainsi intéresser la recherche sur la résolution de programmes linéaires en variables binaires complexes, tels que, par exemple, les problèmes de sac-à-dos de grande taille.

5.3 Proposition d'application avancée au RCPSP

Les résultats pour le RCPSP obtenus par la méthode basique de la section précédente sont bien évidemment en-deça des résultats des nombreuses méthodes exactes qui ont spécifiquement été développées pour ce problème. Ces méthodes exactes sont pour la plupart basées sur des PSE où, à chaque noeud, une évaluation par défaut est donnée, rarement par programmation linéaire mais plutôt par des calculs de bornes aussi spécifiques au problème.

Resolution search peut s'appliquer à de nombreuses formes de résolution, autres que la programmation linéaire. L'unique limitation est ici, la modélisation du problème au moyen de variables binaires et la connaissance d'un `oracle`, permettant de déterminer si un sous-espace de recherche contient ou non une (meilleure) solution. Nous donnons dans cette section une piste d'intégration de resolution search à un schéma de branchement connu pour le RCPSP basé sur les *schémas d'ordonnancement* [Brucker 1998].

5.3.1 Branchement basé sur les schémas d'ordonnancement

Le schéma de branchement proposé par Brucker et al. [Brucker 1998] est certainement l'un des plus évidents à adapter à resolution search. En effet, le branchement s'effectue de façon binaire sur les paires d'activités $\{i, j\}$ en prenant la décision de placer les activités en disjonction $(i - j)$ ou en parallèle $(i \| j)$. Ainsi, les ingrédients de la PSE peuvent être utilisés sans modification dans une méthode de type resolution search.

Dans la PSE originale, la borne inférieure correspond à la borne $LB2$ de Mingozzi et al. [Mingozzi 1998], calculée par génération de colonnes (voir section 2.4.4). Avant le calcul de cette borne, un algorithme de filtrage PPC est invoqué de manière à fixer le séquencencement relatif de certaines paires d'activités, accélérant ainsi la descente, et permettant éventuellement de détecter l'infaisabilité du noeud considéré. Pour resolution search, il suffit donc de définir la fonction `oracle` de façon à ce qu'elle retourne $+\infty$ si l'infaisabilité est constatée, ou $LB2$ sinon.

Une solution heuristique est aussi calculée à chaque noeud. Ce principe peut aussi être utilisé dans resolution search, constituant en cela, une légère amélioration de la procédure de Chvátal. En effet, de la même façon que si la solution retournée par `oracle` est réalisable, on peut aussi mettre à jour $\bar{z}$, dès qu'une solution réalisable strictement meilleure est détectée par l'heuristique. La remontée s'effectue alors comme précédemment.

5.3.2 Améliorations de resolution search

L'utilisation de la PPC fournit un moyen plus intéressant encore d'améliorer resolution search dans le cadre général, en accélérant la gestion de la famille de clause. En effet, quand $\bar{z}$ est mis à jour, l'algorithme de filtrage peut permettre de fixer définitivement certaines variables du problème (ici, le séquencement de paires d'activités). Soit x_i une variable fixée à 0 par exemple. On peut alors, sans modifier la structure *path-like* de la famille F, supprimer toutes les clause de F qui contiennent le littéral $x_i = 1$. De la même façon, on peut supprimer toute occurence du littéral $x_i = 0$ dans chacune des clauses de F sauf s'il s'agit d'un littéral l^k associé à la clause C^k. Dans ce dernier cas, on peut en fait aller plus loin en invoquant l'algorithme de mise à jour de F avec la clause $u = \bar{l}^k$ qui est bien un nogood valide indiquant que la variable x_i ne peut être instanciée à 1 : $\bar{l}^k = (x_i = 1)$. De plus, comme $u \sqsubseteq u_F$ et u ne contient qu'un unique littéral, F sera alors réduite sans perte d'information. Si enfin, la famille F contient la clause vide à la suite de ces transformations, le problème est résolu.

Ce principe utilise la redéfinition du problème pour mettre à jour la famille F. Inversement, les clauses de la famille F permettent d'inférer une redéfinition du problème. En effet, si une instancation partielle u est interdite par les clauses de F, le problème peut être redéfini par :

$$\text{Existe-t'il } x \in X \cup \{0,1\}^n \text{ tel que } u \not\sqsubseteq x \text{ et } f(x) \leq \bar{z} \,?$$

Si le problème se présente sous la forme d'un programme linéaire en variables binaires, la contrainte $u \not\sqsubseteq x$ peut trivialement se linéariser. Par exemple, si $u = (0, 1, *, 1)$ l'inégalité correspondante est $x_1 + (1 - x_2) + (1 - x_4) \geq 1$, signifiant qu'on ne peut avoir à la fois $x_1 = 0$, $x_2 = 1$ et $x_4 = 1$.

Un avantage de resolution search par rapport aux backtrackings intelligents ou à l'algorithme de Benders est justement de ne pas ralentir l'évaluation à chaque noeud en n'ajoutant pas les nogoods sous forme de contraintes additionnelles à la relaxation. Pour conserver ce bénéfice, on peut simplement appliquer cette technique aux clauses de F ne contenant qu'un unique littéral. Par exemple, si la clause $C^k = l^k$ avec $l^k = (x_i = 1)$ appartient à F, on peut fixer dans la définition du problème la variable x_i à 0. Ce faisant, on peut alors supprimer cette clause C^k ainsi que toutes les occurences du littéral $x_i = 0$ dans chacune des autres clauses C^j de F (par la propriété de F, nécessairment, $j > k$ et le littéral $x_i = 0$, autrement dit $\bar{l}^k$ est différent de l^j).

Dans ces deux améliorations de resolution search que nous proposons, la convergence est toujours assurée, puisque les suppressions effectuées dans F deviennent en réalité explicites dans le problème et donc implicites dans F.

5.4 Discussion autour de resolution search

Resolution search développe une idée générale de la programmation par contraintes (l'amélioration du backtracking par l'apprentissage) pouvqnt s'appliquer, par exemple, la programmation linéaire. Plus encore, les exemples d'amélioration possible présentés ci-dessus illustrent un autre aspect de cette méthode. En effet, il est intéressant de remarquer comment resolution search peut être considérée comme une technique originale de collaboration de deux solveurs, avec d'une part l'algorithme de mise à jour de la famille de nogoods gérant les solutions interdites dans la recherche, et d'autre part `oracle` qui étudie les solutions possibles restantes.

Les solveurs communiquent réellement dans les deux directions, `oracle` inférant de nouvelles solutions interdites à F et F imposant l'espace de recherche de `oracle`. En ce sens, les améliorations que nous apportons contribuent à une meilleure collaboration de ces solveurs.

5.4.1 Avantages et Inconvénients

Nous discutons dans cette section, des avantages et inconvénients de resolution search comparée à la fois aux méthodes d'apprentissage et aux PSE et backtrackings classiques.

Comme il a déjà été précisé, la procédure resolution search ne s'applique pas uniquement à l'optimisation de programmes linéaires. Ainsi, le solveur `oracle` peut indifféremment employer des techniques de programmation linéaire tout autant que des algorithmes de consistance en programmation par contraintes ou en programmation purement logique.

Dans le cas de la programmation linéaire, sa prise en considération plus globale du critère d'optimisation rend plus délicate l'identification des nogoods. Ainsi, resolution search, comme tout backtracking intelligent, permet d'éviter la redondance dans la phase de descente par rapport à un backtracking simple, mais, telle qu'appliquée à un programme linéaire, elle reproduit généralement plusieurs fois les mêmes schémas dans la phase de remontée. En particulier, tant qu'un littéral non pertinent apparaît dans au moins une clause de F, il sera testé à chaque remontée. Resolution search est donc moins dépendante de la stratégie de branchement qu'une PSE classique, mais elle dissimule aussi le choix de branchement implicitement effectué au moment d'associer un littéral à une clause entrante de F.

La décomposition de Benders ne présente pas cet inconvénient puisque les nogoods (les coupes ajoutées au programme maître) sont plus rapidement identifiés par résolution du dual du sous-problème. Néanmoins, resolution search a un caractère plus général (il n'est pas besoin d'identifier un sous-problème facile à résoudre) et surtout, la résolution du programme linéaire dans `oracle` n'est pas ralentie par l'ajout de coupes, mais au contraire, simplifiée par la fixation de nouvelles variables. Il s'agit aussi d'un atout de resolution search vis-à-vis des backtrackings intelligents de la PPC, où les nogoods sont ajoutés sous forme de contraintes redondantes au problème.

Comparée à ces méthodes ou aux règles de dominance, telle que la règle du *cut-set* en ordonnancement, resolution search présente en plus l'avantage de restreindre l'apprentissage des nogoods, quitte à en « oublier », mais tout en guidant la recherche pour assurer la convergence. Resolution search présente ainsi un mode de fonctionnement similaire au *backtracking dynamique* tout en en adoptant un parcours du graphe de recherche moins statique. Contrairement au *backtracking dynamique*, resolution search se restreint cependant à des modèles en variables binaires du problème. Enfin, un intérêt majeur de resolution search par rapport à une PSE réside probablement dans une meilleure exploitation de la solution réalisable initiale comme indiqué section 5.2.

5.4.2 Perspectives

Pour conclure ce chapitre, nous avançons quelques perspectives d'améliorations ou d'applications de resolution search que nous avons en vue.

Dans le cadre de la programmation linéaire, la procédure générale bénificierait essentiellement d'une méthode plus efficace d'identification des nogoods, autrement que par défixations et évaluations successives de toutes les variables instanciées. La théorie de la dualité en programmation linéaire fournit justement, par l'intermédiaire des coûts réduits, une indication sur ces variables que l'on peut défixer sans appel supplémentaire à la fonction d'évaluation. Pour des problèmes spécifiques, il peut aussi être intéressant d'étudier la dépendance entre les variables pour détecter si une variable peut être désinstanciée, en fonction d'une autre, sans impact sur la borne inférieure retournée par `oracle`.

Une généralisation de resolution search aux problèmes en variables entières ou mixtes ouvrirait un très large champ d'applications à la méthode. Ceci nécessite de redéfinir la notion de famille

path-like tout en conservant la preuve de convergence, mais aussi la facilité d'implémentation et de gestion des clauses.

Avant cela, nous souhaitons étudier d'autres applications de resolution search à des schémas de branchement existants pour le RCPSP comme les schémas basés sur la séparation des fenêtres de temps [Carlier 1991] ou encore les schémas chronologiques. Dans ce dernier cas, l'utilisation de resolution search soulèverait un problème intéressant à savoir la réoptimisation à partir d'un ordonnancement partiel « à trous » autrement dit, un RCPSP avec des capacités variables au cours du temps.

Conclusion et Perspectives

L'hybridation de la programmation par contraintes et de la programmation linéaire est une approche prometteuse pour la résolution des problèmes d'optimisation combinatoire les plus difficiles. Il existe d'ores et déjà de nombreuses formes d'hybridation mais d'autres, plus encore, restent à découvrir, et l'approche coopérative est certainement aujourd'hui à un stade émergeant.

L'ordonnancement est un domaine privilégié d'application et d'essai de ces méthodes. Il existe, en effet, des problèmes d'ordonnancement particulièrement difficiles, connus et étudiés depuis longtemps, et auxquels une vaste littérature est consacrée. Aussi, différentes techniques issues, ou pouvant être assimilées à, de la programmation par contraintes, ainsi que des méthodes complexes de programmation linéaire ont été développées spécifiquement pour ce type de problèmes, puis améliorées au cours du temps.

L'ordonnancement de projet à contraintes de ressources, ou RCPSP, figure en bonne place parmi ces problèmes de choix, puisqu'il généralise même un grand nombre d'entre eux. Une littérature abondante de méthodes optimales et approchées est dédiée à ce sujet, sans parvenir encore à ce jour à résoudre de manière efficace des instances de test relativement faciles et de petite taille comparées aux problèmes équivalents en pratique. La difficulté principale du RCPSP, et son intérêt aussi, réside dans le fait qu'il s'agit d'un problème faiblement structuré sur lequel les méthodes basiques d'inférence logique ou de programmation linéaire se comportent mal. Il possède néanmoins un minimum de caractéristiques pouvant être exploitées par des méthodes plus complexes, et de nombreuses techniques et modélisations ont été proposées pour ce problème, dans les deux approches.

Nous pensons que la complémentarité de ces méthodes de programmation par contraintes et de programmation linéaire peut être réellement profitable à la résolution du RCPSP. Nous avons ainsi présenté dans ce mémoire, un certain nombre d'applications hybrides au calcul d'évaluations par défaut pour le RCPSP, dans la continuité de l'approche de Brucker et Knust [Brucker 2000]. Le chapitre 3 présente une relaxation lagrangienne d'un modèle linéaire, prétraité efficacement par PPC et servant à l'inverse de preuve d'inconsistance dans une approche destructive. Au chapitre 4, nous proposons un emploi plus poussé de l'inférence logique dans la formulation linéaire, en prétraitement mais aussi pour la génération de coupes, obtenues en linéarisant des informations supplémentaires déjà déduites par PPC.

Ces bornes relativement lourdes à calculer comparativement aux bornes plus classiques de la littérature obtiennent cependant des résultats nettement meilleurs. À moins de suivre l'idée avancée par Carlier et Néron [Carlier 2000] de concentrer une partie du travail à la racine de l'arbre de recherche, la difficulté du problème exige certainement cet effort supplémentaire pour parvenir à le résoudre de manière optimale. Il est intéressant d'ailleurs de voir, dans nos expérimentations sur les bornes destructives, comment une technique coûteuse comme le shaving se révèle capable d'améliorer la méthode générale tout en économisant parfois du temps de calcul. Il s'agit maintenant d'étudier le comportement de ces bornes dans une énumération implicite basée sur un schéma de

séparation adapté.

Il serait aussi avantageux de mettre en oeuvre dans ces bornes d'autres techniques de programmation par contraintes plus spécifiques aux problèmes d'ordonnancement cumulatifs comme le raisonnement énergétique. Nous avons en effet mené nos expérimentations sur les instances de test les plus courantes de la littérature, les instances KSD de la PSPLIB [Kolisch 1995], mais les plus difficiles de ces instances possèdent la particularité d'être faiblement cumulatives, comme le font remarquer Baptiste et Le Pape [Baptiste 2000]. Les techniques de propagation de contraintes que nous employons sont plutôt efficaces sur ce type d'instances mais elles sont beaucoup moins adaptées au cadre fortement cumulatif.

Enfin, les calculs de bornes que nous proposons entrent dans une seule catégorie de collaboration entre les solveurs PPC et PL. Une poursuite de nos travaux peut consister en l'étude de formes de coopération plus évoluées, inspirées par exemple des travaux de Hooker, Thorsteinsson, Ottosson et al. sur la modélisation linéaire/logique mixte [Hooker 1999] et sa résolution par décomposition de Benders hybride [Hooker 2003] ou par *branch-and-check* [Thorsteinsson 2001a].

L'étude de resolution search que nous comptons poursuivre va aussi en ce sens. Au chapitre 5, nous avons cherché à apporter un éclairage nouveau sur la méthode exacte *resolution search* de Chvátal [Chvátal 1997]. En plus de prouver expérimentalement son efficacité vis-à-vis d'une recherche arborescence classique tout à fait comparable, et de proposer deux techniques d'amélioration faciles à implémenter, nous avons souhaité décrire cette procédure en mettant l'accent sur son aspect coopératif, à la fois dans l'idée générale et dans son application pratique. En effet, si resolution search se présente dans un cadre général de résolution, y compris celui de la programmation linéaire, il s'agit en fait d'un *backtracking intelligent* au sens de la programmation par contraintes. Nous effectuons ainsi une comparaison entre cette méthode et les autres méthodes de la littérature basées sur la notion d'*apprentissage* et de réparation des conflits. De plus, cette énumération implicite emploie une forme originale de collaboration de deux solveurs, le premier pour gérer l'espace de recherche et le second, chargé d'explorer et de réduire l'espace restant. Le second solveur, procédant par évaluation de l'espace de recherche courant, peut tout aussi bien consister en une technique de programmation linéaire d'optimisation qu'en un algorithme de filtrage PPC.

Nous sommes persuadés de l'originalité et de la grande efficacité de ce type de méthodes, et il est possible d'y apporter de nombreuses améliorations et domaines d'application. Nous souhaitons donc, en priorité, nous atteler à cette tâche en cherchant à appliquer de manière plus avancée au RCPSP, ou plus généralement à tout programme linéaire, ce principe d'apprentissage.

Liste des algorithmes

Table des figures

Liste des tableaux

Bibliographie

[Aggoun 1993] Aggoun A. et Beldiceanu N., Extending chip in order to solve complex scheduling and placement problems, *Mathematical Computing and Modelling*, 17(7) :57–73, 1993. 7

[Alvarez-Valdés 1993] Alvarez-Valdés R. et Tamarit J. M., The project scheduling polyhedron : dimension, facets and lifting theorems, *European Journal of Operational Research*, 67 :204–220, 1993. 19, 53

[Applegate 1991] Applegate D. et Cook W., A computational study of job-shop scheduling, *ORSA Journal on Computing*, 3(2) :149–156, 1991. 19, 46, 85, 93

[Artigues 2003] Artigues C., Insertion techniques for static and dynamic resource constrained project scheduling, *European Journal of Operational Research*, 149(2) :247–274, 2003. 54

[Baar 1998] Baar T., Brucker P. et Knust S., Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem, dans Voss S., Martello S., I.Osman et C.Roucairol, éditeurs, *Meta-heuristics : Advances and Trends in Local Search Paradigms for Optimization*, pages 1–18, Kluwer, 1998. 78, 80, 117

[Baker 1995] Baker A. B., *Intelligent Backtracking on Constraint Satisfaction Problems : Experimental and Theoretical Results*, Thèse de doctorat, University of Oregon, 1995. 14

[Balas 1970] Balas E., Project scheduling with resource constraints, dans Beale E., éditeur, *Applications of Mathematical Programming Techniques*, American Elsevier, 1970. 52, 53

[Balas 1979] Balas E., Disjunctive programming, *Annals of Discrete Mathematics*, 5 :3–51, 1979. 23, 30

[Balas 1985] Balas E., On the facial structure of scheduling polyhedra, *Mathematical Programming Study*, 24 :179–218, 1985. 19

[Baptiste 1998] Baptiste P., *Une étude théorique et expérimentale de la propagation de contraintes de ressources*, Thèse de doctorat, Université Technologique de Compiègne, 1998. 43, 49

[Baptiste 2004] Baptiste P. et Demassey S., Tight LP bounds for resource constrained project scheduling, *OR Spectrum*, 26 :251–262, 2004. 46, 85, 97, 98

[Baptiste 1996] Baptiste P. et Le Pape C., Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, dans *Proceedings of the 15th Workshop of the UK Planning Special Interest Group*, 1996. 47, 60

[Baptiste 2000] Baptiste P. et Le Pape C., Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems, *Constraints*, 5 :119–139, 2000. 46, 57, 60, 61, 124

[Baptiste 1999] Baptiste P., Le Pape C. et Nuijten W., Satisfiability tests and time-bound adjustments for cumulative scheduling problems, *Annals of Operations Research*, 92 :305–333, 1999. 58

[Baptiste 2001] Baptiste P., Le Pape C. et Nuijten W., *Constraint-based Scheduling*, tome 39 de *International Series in Operations Research and Management Science*, Kluwer Academic Publishers, 2001. 44, 48, 49, 60

[Bayardo 1996] Bayardo R. et Schrag R., Using csp look-back techniques to solve exceptionally hard sat instances, dans *Proceeding of the International Conference on Principles and Practice of Constraint Programming, CP'96*, pages 46–60, 1996. 115

[Beaumont 1990] Beaumont N., An algorithm for disjunctive programs, *European Journal of Operations Research*, 48(3) :362–371, 1990. 30

[Benoist 2001] Benoist T., Laburthe F. et Rottembourg B., Lagrange relaxation and constraint programming collaborative schemes for travelling tournament problems, dans *Proceedings of the 3th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'01*, 2001. 35

[Beringer 1994] Beringer H. et de Backer B., *Logic Programming : Formal Methods and Practical Applications*, chapitre Combinatorial problem solving in constraint logic programming with cooperating solvers, Elsevier Science Publishers, 1994. 32, 33

[Bessière 1996] Bessière C. et Régin J.-C., MAC and combined heuristics : two reasons to forsake FC (and CBJ ?), dans Freuder E. et Jampel M., éditeurs, *Principles and Practice of Constraint Programming*, tome 1118 de *Lecture Notes in Computer Science*, Springer, 1996. 11, 13

[Bistarelli 1997] Bistarelli S., Montanari U. et Rossi F., Semiring-based constraint satisfaction and optimization, *Journal of the ACM*, 44(2) :201–236, 1997. 15

[Bockmayr 1998] Bockmayr A. et Kasper T., Branch-and-infer : A unifying framework for integer and finite domain constraint programming, *INFORMS Journal on Computing*, 10(3) :287–300, 1998. 33

[Bockmayr 2003] Bockmayr A. et Pisaruk N., Detecting infeasibility and generating cuts for MIP using CP, dans *Proceedings of the 5th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'03*, Montreal, 2003. 32

[Brucker 1999] Brucker P., Drexl A., Möhring R., Neumann K. et Pesch E., Resource-constrained project scheduling problem : Notation, classification, models and methods, *European Journal of Operational Research*, 112(1) :3–41, 1999. 39, 43, 57

[Brucker 2000] Brucker P. et Knust S., A linear programming and constraint propagation-based lower bound for the RCPSP, *European Journal of Operational Research*, 127 :355–362, 2000. 2, 33, 61, 63, 67, 74, 75, 82, 85, 97, 103, 123

[Brucker 2001] Brucker P. et Knust S., Resource-constrained project scheduling and timetabling, dans E.Burke W., éditeur, *The Practice and Theory of Automated Timetabling III*, tome 2079 de *Lecture Notes in Computer Science*, pages 277–293, Springer Verlag, 2001. 42

[Brucker 1998] Brucker P., Knust S., Schoo A. et Thiele O., A branch and bound algorithm for the resource-constrained project scheduling problem, *European Journal of Operational Research*, 107 :272–288, 1998. 46, 50, 59, 60, 63, 119

[Carlier 1991] Carlier J. et Latapie B., Une méthode arborescente pour résoudre les problèmes cumulatifs, *RAIRO-Recherche Opérationnelle*, 25(3) :311–340, 1991. 56, 59, 122

[Carlier 2000] Carlier J. et Néron E., A new LP based lower bound for the cumulative scheduling problem, *European Journal of Operational Research*, 127(2) :363–382, 2000. 64, 123

[Carlier 2003] Carlier J. et Néron E., On linear lower bounds for the resource constrained project scheduling problem, *European Journal of Operational Research*, 149 :314–324, 2003. 60, 64, 65

[Carlier 1989] Carlier J. et Pinson E., An algorithm for solving the job-shop problem, *Management Science*, 35 :164–176, 1989. 46

[Carlier 1990] Carlier J. et Pinson E., A practical use of Jackson's preemptive schedule for solving the job-shop problem, *Annals of Operations Research*, 26 :269–287, 1990. 47, 48

[Carlier 1994] Carlier J. et Pinson E., Adjustment of heads and tails for the job-shop problem, *European Journal of Operational Research*, 78 :146–161, 1994. 48, 50, 51

[Caseau 1996] Caseau Y. et Laburthe F., Cumulative scheduling with task intervals, dans Maher M., éditeur, *Proceedings of the Joint International Conference and Symposium on Logic Programming, JCPSLP'96*, pages 363–377, The MIT Press, Cambridge, MA, 1996. 51, 60, 61

[Cavalcante 2001] Cavalcante C., de Souza C., Savelsbergh M., Wong Y. et Wolsey L., Scheduling projects with labor constraints, *Discrete Applied Mathematics*, 112 :27–52, 2001. 55

[Christofides 1987] Christofides N., Alvarez-Valdés R. et Tamarit J. M., Project scheduling with resource constraints : a branch and bound approach, *European Journal of Operational Research*, 29(3) :262–273, 1987. 55, 59, 62, 71, 94, 95

[Chvátal 1973] Chvátal V., Edmonds polytopes and a hierarchy of combinatorial problems, *Discrete Mathematics*, 4 :305–337, 1973. 22

[Chvátal 1997] Chvátal V., Resolution search, *Discrete Applied Mathematics*, 73 :81–99, 1997. 2, 3, 105, 113, 114, 124

[Codato 2003] Codato G. et Fischetti M., Combinatorial Benders' cuts, Rapport technique, DEI, University of Padova, Italy, 2003. 115

[Dantzig 1951] Dantzig G., *Activity Analysis of Production and Allocation*, chapitre Maximization of a linear function of variables subject to linear inequalities, pages 339–347, Wiley, New-York, 1951. 17

[Debruyne 1997] Debruyne R. et Bessière C., From restricted path consistency to max-restricted path consistency, dans *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'97*, pages 312–326, 1997. 11

[Dechter 1990] Dechter R., Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition, *Artificial Intelligence*, 41 :273–312, 1990. 12

[Demassey 2002] Demassey S., Artigues C. et Michelon P., A hybrid constraint propagation-cutting plane algorithm for the rcpsp, dans *Proceedings of the 4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'02*, pages 321–331, 2002. 85

[Demassey 2003] Demassey S., Artigues C. et Michelon P., Constraint-propagation-based cutting planes : an application to the resource-constrained project-scheduling problem, *INFORMS Journal on Computing*, 2003, à paraître. 51, 85

[Demeulemeester 1992] Demeulemeester E. et Herroelen W., A branch-and-bound procedure for the multiple-resource constrained single project scheduling problem, *Management Science*, 38(12) :1803–1818, 1992. 59, 82

[Demeulemeester 1997] Demeulemeester E. et Herroelen W., New benchmark results for the resource-constrained project scheduling problem, *Management Science*, 43(11) :1485–1492, 1997. 59, 63

[Demeulemeester 2002] Demeulemeester E. L. et Herroelen W. S., *Project Scheduling : a Research Handbook*, tome 49 de *International Series in Operations Research and Management Science*, Kluwer Academic Publishers, 2002. 43, 57

[Dixon 2000] Dixon H. E. et Ginsberg M. L., Combining satisfiability techniques from AI and OR, *The Knowledge Engineering Review*, 15 :31–45, 2000. 38

[Dorndorf 2000] Dorndorf U., Pesch E. et Phan-Huy T., A branch-and-bound algorithm for the resource constrained project scheduling problem, *Mathematical Methods of Operations Research*, 52 :413–439, 2000. 59, 60, 61

[Dorndorf 2001] Dorndorf U., Pesch E. et Phan-Huy T., Solving the open-shop scheduling problem, *Journal of Scheduling*, 4 :157–174, 2001. 51

[Dorndorf 1999] Dorndorf U., Phan-Huy T. et Pesch E., *Project Scheduling – Recent Models, Algorithms and Applications*, chapitre 10, A Survey of Interval Capacity Consistency Tests for Time- and Resource-Constrained Scheduling, pages 213–238, Kluwer Academic Publishers, 1999. 44

[Dyckhoff 1990] Dyckhoff H., A typology of cutting and packing problems, *European Journal of Operational Research*, 44 :145–159, 1990. 42

[Dyer 1990] Dyer M. E. et Wolsey L. A., Formulating the single machine sequencing problem with release dates as a mixed integer program, *Discrete Applied Mathematics*, 26 :255–270, 1990. 19, 93

[Eremin 2001] Eremin A. et Wallace M., Hybrid benders decomposition algorithms in constraint logic programming, dans Walsh T., éditeur, *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'01*, tome 2239 de *Lecture Notes in Computer Science*, pages 1–15, Springer, 2001. 35

[Erschler 1991] Erschler J., Lopez P. et Thuriot C., Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement, *Revue d'Intelligence Artificielle*, 5 :7–32, 1991. 48

[Esquirol 1999] Esquirol P. et Lopez P., *L'Ordonnancement*, Economica, Paris, 1999. 42

[Esquirol 2001] Esquirol P., Lopez P. et Huguet M.-J., *Ordonnancement de la production*, chapitre Propagation de contraintes en ordonnancement, Hermès Science Publications, Paris, 2001. 44, 50

[Feige 1998] Feige U. et J.Kilian, Zero-knowledge and the chromatic number, *Journal of Computer and System Sciences*, 57 :187–199, 1998. 41

[Fischetti 2003] Fischetti M. et Lodi A., Local branching, *Mathematical Programming*, 98 :23–47, 2003. 119

[Fisher 1973] Fisher M., Optimal solution of scheduling problems using lagrange multipliers, Part I, *Operations Research*, 21(5), 1973. 52, 62

[Focacci 2002] Focacci F., Lodi A. et Milano M., Optimization-oriented global constraints, *Constraints*, 7(3-4) :351–365, 2002. 31, 33

[Freuder 1982] Freuder E., A sufficient condition for backtrack-free search, *Journal of the Association for Computing Machinery*, 29 :24–32, 1982. 8

[Garey 1979] Garey M. R. et Johnson D. S., *Computers and Intractibility. A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979. 18, 41

[Gaschnig 1979] Gaschnig J., *Performance Measurement and Analysis of Certain Search Algorithms*, Thèse de doctorat, Carnegie-Mellon University, Pittsburgh, 1979, rapport Technique CMU-CS-79-124. 12

[Ginsberg 1993] Ginsberg M., Dynamic backtracking, *Journal of Artificial Intelligence Research*, 1 :25–46, 1993. 12

[Glover 1975] Glover F., Surrogate constraint duality in mathematical programming, *Operations Research*, 23(3) :434–451, 1975. 22

[Gomory 1958] Gomory R. E., Outline of an algorithm for integer solutions to linear programs, *Bulletin of the American Mathematical Society*, 64 :275–278, 1958. 22

[Grötschel 1981] Grötschel M., Lovász L. et Schrijver A., The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica*, 1(2) :169–197, 1981. 19

[Guéret 2000] Guéret C., Jussien N. et Prins C., Using intelligent backtracking to improve branch-and-bound methods : an application to open-shop problems, *European Journal of Operational Research*, 127 :344–354, 2000. 37

[Hajian 1995] Hajian M., El Sakkout H., Wallace M., Lever J. et Richards E., Towards a closer integration of finite domain propagation and simpled-based algorithms, dans *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 1995. 31

[Hanafi 2002] Hanafi S. et Glover F., Resolution search and dynamic branch-and-bound, *Journal of Combinatorial Optimization*, 6(4) :401–423, 2002. 115

[Hartmann 2000] Hartmann S. et Kolisch R., Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem, *European Journal of Operational Research*, 127 :394–407, 2000. 56

[Heipcke 1999] Heipcke S., *Combined modelling and problem solving in mathematical programming and constraint programming*, Thèse de doctorat, School of Business, University of Buckingham, U.K., 1999. 31, 33, 42

[Herroelen 1999] Herroelen W., Demeulemeester E. et Reyck B. D., An integrated classification scheme for ressource scheduling, Rapport technique, Department of Applied Economics, K.U.Leuven, Belgium, 1999. 39

[Hooker 1999] Hooker J. et Osorio M., Mixed logical / linear programming, *Discrete Applied Mathematics*, 96–97 :395–442, 1999. 34, 124

[Hooker 2000a] Hooker J. N., *Logic-based methods for optimization : Combining Optimization and Constraint Satisfaction*, Wiley, New-York, 2000. 27, 31, 37

[Hooker 2003] Hooker J. N. et Ottosson G., Logic-based benders decomposition, *Mathematical Programming*, 96 :33–60, 2003. 124

[Hooker 2000b] Hooker J. N., Ottosson G., Thorsteinsson E. S. et Kim H.-J., A scheme for unifying optimization and constraint satisfaction methods, *Knowledge Engineering Review, special issue on AI/OR*, 15(1) :11–30, 2000. 29, 34, 38

[Hooker 2002] Hooker J. N. et Yan H., A relaxation for the cumulative constraint, dans Hentenryck P. V., éditeur, *Proceedings of the International Conference on Principles and Practice of Constraint Programming CP'02*, tome 2470 de *Lecture Notes in Computer Science*, pages 686–690, Springer, 2002. 31

[Jain 2001] Jain V. et Grossmann I. E., Algorithms for hybrid MILP/CP models for a class of optimization problems, *INFORMS Journal on Computing*, 13(258–276), 2001. 31, 35

[Junker 1999] Junker U., Karisch S. E., Kohl N., Vaaben B., Fahle T. et Sellmann M., A framework for constraint programming based column generation, dans *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'99*, pages 261–274, 1999. 35

[Jussien 2000] Jussien N., Debruyne R. et Boizumault P., Maintaining arc-consistency within dynamic backtracking, dans *Principles and Practice of Constraint Programming*, pages 249–261, 2000. 13

[Kaplan 1988] Kaplan L., *Resource-constrainted project scheduling with preemption of jobs*, Thèse de doctorat, University of Michigan, États-Unis, 1988. 55

[Karmarkar 1984] Karmarkar N., A new polynomial-time algorithm for linear programming, *Combinatorica*, 4 :373–395, 1984. 17

[Khachiyan 1979] Khachiyan L., A polynomial algorithm in linear programming, *Soviet Mathematics Doklady*, 20 :191–194, 1979. 17

[Klein 2000] Klein R., Project scheduling with time-varying resource constraints, *International Journal of Production Research*, 38(16) :3937–3952, 2000. 55

[Klein 1999] Klein R. et Scholl A., Computing lower bound by destructive improvement : An application to resource-constrained project scheduling, *European Journal of Operational Research*, 112 :322–346, 1999. 15, 56, 61, 103

[Kolisch 1999] Kolisch R. et Hartmann S., Algorithms for solving the resource-constrained project scheduling problem : Classification and computational analysis, dans Weglarz J., éditeur, *Handbook on Recent Advances in Project Scheduling*, chapitre 7, Kluwer Academic Publishers, 1999. 56

[Kolisch 2001] Kolisch R. et Padman R., An integrated survey of deterministic project scheduling, *Omega*, 29(3) :249–272, 2001. 43

[Kolisch 1998] Kolisch R., Schwindt C. et Sprecher A., Benchmark instances for project scheduling problems, dans Weglarz J., éditeur, *Handbook on Recent Advances in Project Scheduling*, Kluwer, 1998. 57

[Kolisch 1995] Kolisch R., Sprecher A. et Drexl A., Characterization and generation of a general class of resource-constrained project scheduling problems, *Management Science*, 41 :1693–1703, 1995. 124

[Le Pape 1988] Le Pape C., *Des systèmes d'ordonnancement flexibles et opportunistes*, Thèse de doctorat, Université Paris XI, 1988. 60

[Lemaréchal 2001] Lemaréchal C., Lagrangian relaxation, dans *Computational Combinatorial Optimization : Optimal or Provably Near-Optimal Solutions*, tome 2241 de *Lecture Notes in Computer Science*, pages 112–156, Springer-Verlag, 2001. 26

[Lopez 1991] Lopez P., *Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources*, Thèse de doctorat, Université Paul Sabatier, Toulouse, 1991. 48, 49

[Martin 1996] Martin P. et Shmoys D. B., A new approach to computing optimal schedules for the job-shop scheduling problem, dans Cunningham W. H., McCormick S. T. et Queyranne M., éditeurs, *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization IPCO'96*, pages 389–403, Vancouver, British Columbia, Canada, 1996. 50, 51

[Milano 2003] Milano M., éditeur, *Constraint and Integer Programming - Toward a Unified Methodology*, Kluwer, 2003. 27

[Milano 2002] Milano M., Ottosson G., Refalo P. et Thorsteinsson E., The role of integer programming techniques in constraint programming's global constraints, *INFORMS Journal on Computing, Special Issue on "The Merging of Mathematical Programming and Constraint Programming"*, 14(4), 2002. 31, 33

[Mingozzi 1998] Mingozzi A., Maniezzo V., Ricciardelli S. et Bianco L., An exact algorithm for the multiple resource-constrained project scheduling problem based on a new mathematical formulation, *Management Science*, 44 :714–729, 1998. 2, 55, 58, 63, 67, 68, 74, 82, 119

[Mohring 2001] Mohring R., Schulz A., Stork F. et Uetz M., On project scheduling with irregular starting time costs, *Operations Research Letters*, 28 :149–154, 2001. 71

[Möhring 2003] Möhring R. H., Schultz A., Stork F. et Uetz M., Solving project scheduling problems by minimum cut computations, *Management Science*, 49 :330–350, 2003. 2, 55, 62, 63, 67, 71, 72, 94, 101

[Nemhauser 1988] Nemhauser G. L. et Wolsey L. A., *Integer and Combinatorial Optimization*, Wiley, New York, 1988. 16

[Néron 1999] Néron E., *Du flow-shop hybride au problème cumulatif*, Thèse de doctorat, Université Technologique de Compiègne, 1999. 41, 42, 57

[Néron 2001] Néron E., Baptiste P. et Gupta J. N., Solving hybrid flow shop problem using energetic reasoning and global operations, *Omega*, 29(6) :501–511, 2001. 51, 60

[Nuijten 1994] Nuijten W., *Time and resource constrained scheduling : A constraint satisfaction approach*, Thèse de doctorat, Eindhoven University of Technology, 1994. 48, 78, 93

[Ottosson 2002] Ottosson G., Thorsteinsson E. S. et Hooker J. N., Mixed global constraints and inference in hybrid CLP–IP solvers, *Annals of Mathematics and Artificial Intelligence, Special Issue on Large Scale Combinatorial Optimisation and Constraints*, 34(4) :271–290, 2002. 31

[Patterson 1990] Patterson J., Slowinski R., Talbot F. et Weglarz J., Computational experience with a backtracking algorithm for solving a general class of precedence and resource constrained scheduling problems, *European Journal of Operational Research*, 49 :68–79, 1990. 58

[Patterson 1976] Patterson J. H. et Roth G. W., Scheduling a project under multiple resource constraints : a zero-one programming approach, *AIIE Transactions*, 8(449–455), 1976. 52

[Péridy 1996] Péridy L., *Le problème de job-shop : arbitrages et ajustements*, Thèse de doctorat, Université Technologique de Compiègne, 1996. 51, 52

[Pinson 1988] Pinson E., *Le problème de job-shop*, Thèse de doctorat, Université Paris VI, 1988. 47

[Pritsker 1969] Pritsker A., Watters L. et Wolfe P., Multi-project scheduling with limited resources : a zero-one programming approach, *Management Science*, 16 :93–108, 1969. 52, 54, 94, 115

[Prosser 1993] Prosser P., Hybrid algorithms for the constraint satisfaction problem, *Computational Intelligence*, 9(3) :268–299, 1993. 12

[Prosser 1995] Prosser P., MAC-CBJ : maintaining arc consistency with conflict-directed backjumping, Rapport technique Research Report/95/177, Dept. of Computer Science, University of Strathclyde, 1995. 13

[PSPLIB] PSPLIB, Project scheduling problem library, http. 57

[Queyranne 1993] Queyranne M., Structure of a simple scheduling polyhedron, *Mathematical Programming*, 58 :263–285, 1993. 18

[Queyranne 1994] Queyranne M. et Schulz A., Polyhedral approaches to machine scheduling, Rapport technique 408/1994, Technischen Universität Berlin, 1994. 19, 31

[Queyranne 1991] Queyranne M. et Wang Y., Single-machine scheduling polyhedra with precedence constraints, *Mathematics of Operations Research*, 16 :1–20, 1991. 19

[Refalo 1999] Refalo P., Tight cooperation and its application in piecewise linear optimization, dans Jaffar J., éditeur, *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'99*, tome 1713 de *Lecture Notes in Computer Science*, pages 373–389, Springer, 1999. 31, 33

[Refalo 2000] Refalo P., Linear formulation of constraint programming models and hybrid solvers, dans Dechter R., éditeur, *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'00*, tome 1894 de *Lecture Notes in Computer Science*, pages 369–383, Springer, 2000. 30, 32

[Régin 1994] Régin J.-C., A filtering algorithm for constraints of difference in CSPs, dans *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994. 9

[Rivreau 1999] Rivreau D., *Problèmes d'Ordonnancement Disjonctifs : Règles d'Elimination et Bornes Inférieures*, Thèse de doctorat, Université Technologique de Compiègne, 1999. 51

[Robinson 1965] Robinson J., A machine-oriented logic based on the resolution principle, *Journal of the ACM*, 12(1) :23–41, 1965. 107

[Rodosek 1997] Rodosek R., Wallace M. et Hajian M., A new approach to integrating mixed integer programming and constraint logic programming, *Baltzer Journals*, 1997. 30, 33

[Sabin 1994] Sabin D. et Freuder E. C., Contradicting Conventional Wisdom in Constraint Satisfaction, dans Borning A., éditeur, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, tome 874, pages 10–20, 1994. 52

[Sankaran 1999] Sankaran J. K., Bricker D. L. et Juang S.-H., A strong fractionnal cutting-plane algorithm for resource-constrained project scheduling, *International Journal of Industrial Engineering : Applications and Practice*, 6(2) :99–111, 1999. 62, 94, 95

[Schäffter 1997] Schäffter M., Scheduling with respect to forbidden sets, *Discrete Applied Mathematics*, 72 :141–154, 1997. 41

[Schiex 1995] Schiex T., Fargier H. et Verfaillie G., Valued constraint satisfaction problems : Hard and easy problems, dans Mellish C., éditeur, *IJCAI'95 : Proceedings International Joint Conference on Artificial Intelligence*, Montreal, 1995. 15

[Schiex 1994] Schiex T. et Verfaillie G., Nogood Recording for Static and Dynamic Constraint Satisfaction Problem, *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994. 13

[Schultz 1996] Schultz A. S., *Polytopes and Scheduling*, Thèse de doctorat, Technischen Universität Berlin, 1996. 19, 31

[Sellmann 2003] Sellmann M. et Fahle T., Constraint programming based lagrangian relaxation for the automatic recording problem, *Annals of Operations Research*, 118 :17–33, 2003. 35

[Sprecher 1996] Sprecher A. et Drexl A., Minimal delaying alternatives and semi-active timetabling in resource-constrained project scheduling, Rapport technique 426, BWL Universität Kiel, 1996. 58

[Stallman 1977] Stallman R. et Sussman G., Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence*, 9 :135–196, 1977. 12

[Stinson 1978] Stinson J. P., Davis E. W. et Khumawala B. M., Multiple resource-constrained scheduling using branch-and-bound, *AIIE Transactions*, 10(3) :252–259, 1978. 52, 56, 58, 59

[Talbot 1978] Talbot F. et Patterson J. H., An efficient integer programming algorithm with network cuts for solving RCSP, *Management Science*, 24(11) :1163–1174, 1978. 52, 59

[Thorsteinsson 2001a] Thorsteinsson E., Branch-and-check : A hybrid framework integrating mixed integer programming and constraint logic programming, dans Walsh T., éditeur, *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP'01*, tome 2239 de *Lecture Notes in Computer Science*, pages 16–30, Springer, 2001. 31, 36, 124

[Thorsteinsson 2001b] Thorsteinsson E. S. et Ottosson G., Linear relaxations and reduced-cost based propagation of continuous variable subscripts, *Annals of Operations Research, Special Issue on Integration of Constraint Programming, Artificial Intelligence and Operations Research Methods*, 2001, accepted for publication. 31, 33, 34

[Torres 2000] Torres P. et Lopez P., Overview and possible extensions of shaving techniques for job-shop problems, dans *Proceedings of the Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'00*, pages 181–186, Paderborn, Germany, 2000. 52

[Uetz 2001] Uetz M., *Algorithms for deterministic and stochastic scheduling*, Thèse de doctorat, Technischen Universität Berlin, 2001. 41, 55, 71

[Williams 2001] Williams H. et Yan H., Representations of the all different predicate of constraint satisfaction in integer programming, *INFORMS Journal on Computing*, 13(2) :96–103, 2001. 31

[Wolsey 1998] Wolsey L., *Integer Programming*, Wiley, New York, 1998. 16, 21