Séquencer et compter avec la contrainte multicost-regular

Julien Menana, Sophie Demassey

Séquencer et compter avec la contrainte multicost-regular

Julien Menana **Sophie Demassey**

École des Mines de Nantes, LINA CNRS UMR 6241, F-44307 Nantes, France.

{julien.menana,sophie.demassey}@emn.fr

Abstract

Ce papier introduit une contrainte globale associant une contrainte `regular` et plusieurs coûts cumulatifs. Le problème d'optimisation sous-jacent à la contrainte `multicost-regular` est NP-difficile mais peut être réduit par une relaxation lagrangienne performante. L'expressivité et l'efficacité de cette nouvelle contrainte sont évaluées sur des problèmes de planification de personnel basés sur des réglementations du travail réelles. Les résultats empiriques de la contrainte `multicost-regular` surpassent de façon significative les résultats donnés par un modèle décomposé comprenant d'une part la contrainte `regular` et d'autre part la contrainte `global-cardinality`.

1 Introduction

L'action simultanée du séquencement et du comptage d'objets intervient dans de nombreux problèmes de décision combinatoires, et notamment dans les problèmes de planification d'horaires et de tournées de véhicules. Lors de sa tournée, un véhicule visite une suite de lieux en empruntant un chemin dans le réseau routier. Ce chemin ou circuit est défini selon des critères numériques tels que la distance totale parcourue, le temps nécessaire, ou la capacité du véhicule. Si un seul attribut numérique est spécifié, trouver un circuit possible revient à résoudre un problème de plus court/long chemin. Lorsque plusieurs attributs sont renseignés, le problème de plus court/long chemin sous contraintes de ressources – ou Resource Constrained Shortest/Longest Path Problem (RCSPP) –devient NP-complet. Toutes ces conditions numériques restreignent le nombre de chemins pouvant correspondre au sein du réseau routier. C'est pourquoi il est plus efficace de propager l'ensemble de ces critères pendant la recherche de chemins, plutôt que de les prendre en compte séparément. Un parallèle peut être fait avec les problèmes de planification d'horaires. En effet, planifier le travail d'un employé revient à définir, sur une plage de temps donnée, une séquence d'activités (ou postes) respectant différentes règles, nombreuses et variées, comme par exemple : « une nuit de travail est suivie d'une matinée de repos », un travail de nuit est payé double, au moins dix jours de repos par mois, etc. Ainsi, la définition d'un planning horaire mêle étroitement des critères structurels – les séquences d'activités autorisées ou interdites – et des critères numériques – les coûts et les compteurs d'activités. La modélisation de toutes ces conditions individuellement est en soi une tâche difficile, pour laquelle l'expressivité et la flexibilité de la programmation par contraintes sont reconnues. Les modéliser de manière efficace est plus difficile encore puisque cela implique de pouvoir raisonner sur l'ensemble des conditions de manière globale. En introduisant la contrainte globale `regular`, Pesant [9] proposait une méthode élégante et efficace pour modéliser et pour résoudre les contraintes structurelles de séquencement comme une recherche de chemins dans un graphe acyclique. Les contraintes d'optimisation `soft-regular` [14] et `cost-regular` [4] étendent cette approche en considérant des bornes sur le coût total (coût de violation ou coût financier) d'une séquence d'activités. Le problème sous-jacent consiste alors à calculer les plus courts et plus longs chemins du graphe acyclique et à les filtrer selon ces bornes. La contrainte `cost-regular` a été appliquée avec succès, au sein d'une approche de génération de colonnes basée sur la programmation par contraintes, pour résoudre des problèmes réels de planification de personnel [4]. Pourtant, le modèle de programmation par contraintes utilisé révèle une trop

faible interaction entre la contrainte `cost-regular` et une contrainte externe `global-cardinality` chargée de limiter le nombre d'occurrences des activités. En effet, avec une telle décomposition, le graphe support de `cost-regular` conserve de nombreux chemins qui ne satisfont pas aux contraintes de cardinalité. Dans cet article, nous poursuivons la généralisation de cette approche en proposant de traiter plusieurs compteurs ou attributs de coûts au sein d'une seule contrainte globale `multicost-regular`. Celle-ci permet de raisonner simultanément sur les critères de séquencement, de coûts et d'occurrences. Comme évoqué ci-avant, le problème d'optimisation sous-jacent est un RCSPP et il reste NP-difficile même quand le graphe est acyclique. Ainsi, l'algorithme de filtrage que nous proposons atteint un niveau de consistance relâché. Celui-ci s'appuie sur une relaxation lagrangienne du RCSPP en suivant le principe de filtrage par relaxation lagrangienne défini par Sellmann [11]. Notre implémentation de `multicost-regular` est disponible dans la distribution du solveur de contraintes open-source *CHOCO* [1].

Ce papier est organisé de la façon suivante. Dans la Section 2, la classe des contraintes `regular` est présentée et une comparaison théorique entre l'approche de recherche par chemins de Pesant [9] et l'approche par décomposition proposée par Beldiceanu et al. [1] est développée. Nous introduirons alors la nouvelle contrainte `multicost-regular`. Dans la Section 3, une introduction à l'algorithme de filtrage par relaxation lagrangienne est présentée. Dans la Section 4, une variété de règles de travail standards sont décrites et l'étude d'une méthode systématique de création d'une instance `multicost-regular` à partir d'un ensemble de ces règles est présentée. Enfin, dans la Section 5, les résultats de calculs empiriques sur des cas tests de problèmes de planification de personnel sont comparés. Ils montrent l'atout significatif de `multicost-regular` comparée à un modèle par décomposition avec des contraintes `regular` et `globalcardinality`.

2 Contraintes d'appartenance à un langage rationnel

Cette section revient sur la définition de la contrainte `regular` et les travaux associés puis introduit la contrainte `multicost-regular`. Tout d'abord, nous rappelons les notions de base de la théorie des automates et introduisons les notations qui seront utilisées dans ce papier :

Nous considérons un ensemble non-vide Σ appelé *alphabet*. Les éléments de l'ensemble Σ sont appelés *lettres*, les séquences de lettres sont appelées *mots*, et les ensembles de mots sont appelés *langages* dans Σ. Un *automate* Π est un multigraphe orienté (Q, Δ) dont les arcs sont étiquetés par les lettres d'un alphabet Σ, et où deux sous-ensembles non-vides de noeuds I et A sont distingués. L'ensemble de noeuds Q est appelé l'ensemble d'*états* de Π, I est l'ensemble des *états initiaux*, et A est l'ensemble des*états finaux*. L'ensemble non-vide $\Delta \subseteq Q \times \Sigma \times Q$ des arcs est appelé l'ensemble des *transitions* de Π. Un mot dans Σ est dit *accepté* par Π s'il correspond à la séquence d'étiquettes des arcs d'un chemin depuis un état initial vers un état final dans Π. L'automate Π est un *automate fini déterministe (AFD)* si Δ est fini et s'il possède un unique état initial ($I = \{s\}$) et aucune paire de transitions partageant la même extrémité initiale et la même étiquette. Le langage accepté par un AFD est un *langage rationnel* ou *régulier*.

2.1 Chemins et décomposition : deux approches pour `regular`

La contrainte `regular` a été introduite par Pesant [9]. Étant donnée une séquence $X = (x_1, x_2, ..., x_n)$ de variables et un automate fini déterministe $\Pi = (Q, \Sigma, \Delta, \{s\}, A)$, la contrainte `regular`$(X, \Pi)$ est vérifiée si et seulement si X est un mot de taille n sur Σ accepté par Π. Par définition, les solutions de `regular`(X, Π) correspondent une à une aux chemins de n arcs connectant s à un sommet dans A dans le multigraphe orienté Π. Soit $\delta_i \in \Delta$ l'ensemble des transitions qui correspondent au i-ème arc d'un tel chemin, alors une valeur pour x_i est cohérente si et seulement si δ_i contient une transition étiquetée par cette valeur.

Incidemment, Pesant [9] et Beldiceanu et al [1] ont introduit deux approches orthogonales pour assurer la Consistance d'Arc Généralisée (CAG) de la contrainte `regular` (voir la figure 1). L'approche proposée par Pesant [9] est de développer Π comme un AFD acyclique Π_n qui accepte uniquement les mots de longueur n. Par construction, Π_n est un multigraphe à plusieurs couches avec l'état s dans la couche 0 (la source), les états finaux A dans la couche n (les puits), et où l'ensemble des arcs d'une couche quelconque i coïncide avec δ_i. Une première recherche étendue permet de maintenir la cohérence entre Π_n et les domaines des variables en retirant les arcs dans δ_i dont les étiquettes ne sont pas dans le domaine de x_i, puis en retirant les noeuds et arcs qui ne sont pas connectés à une source et à un puits. Dans Beldiceanu et al [1], une contrainte `regular` est décomposée en n contraintes ternaires définies en extension et modélisant les ensembles $\delta_1, \delta_2, \ldots, \delta_n$. La décomposition introduit des variables d'état $q_0 \in \{s\}, q_1, \ldots, q_{n-1} \in Q$, $q_n \in A$ et des contraintes ternaires de transition

[1] `http ://choco.emn.fr/`

$(q_{i-1}, x_i, q_i) \in \delta_i$. Le réseau des contraintes de transition étant Berge-acyclique, appliquer une consistance d'arc sur la décomposition permet d'atteindre la CAG sur `regular`.

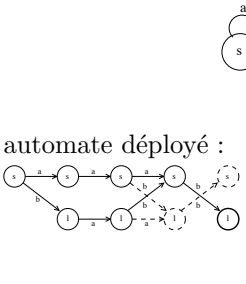


FIG. 1 – Soit l'AFD décrit ci-dessus appliqué à $X \in \{a, b\} \times \{a\} \times \{a, b\} \times \{b\}$. L'automate déployé de `regular` est présenté à gauche et le modèle décomposé sur la droite du schéma. Les transitions en pointillé sont filtrées dans les deux modèles.

Dans la première approche, un algorithme spécifique est défini pour maintenir l'ensemble des chemins support, tandis que la seconde approche permet de laisser le solveur propager les contraintes de transitions. Ces deux approches de filtrage sont orthogonales mais, selon la propagation du second modèle, elles peuvent procéder de manière rigoureusement identique.

Assumons, sans perte de généralité, que Σ est l'union de domaines variables, alors l'exécution initiale de l'algorithme de Pesant pour la construction de Π_n est réalisé en $O(n|\Delta|)$ en temps et en espace (avec $\Delta \leq |Q||\Sigma|$ si Π est un AFD). Le filtrage incrémental procède par un parcours avant/arrière de Π_n et possède cette même complexité de pire cas. En réalité, la complexité de l'algorithme dépend plus de la taille $\Delta_n|$ de l'automate déployé Π_n que de la taille $|\Delta|$ de l'automate spécifié Π. Notons par exemple que lorsque l'automate spécifié Π accepte uniquement des mots de taille n alors il est déjà déployé ($\Pi = \Pi_n$) et la première exécution de l'algorithme est en $O(|\Delta|)$. En pratique, de même que dans nos tests (Section 5), Π_n peut même être nettement plus petit que Π. Cela signifie que de nombreux états finaux dans Π ne peuvent pas être atteints en exactement n transitions. Le filtrage s'effectue alors en $O(|\Delta_n|)$ avec ici $|\Delta_n| \ll n|\Delta|$.

`regular` est une contrainte très expressive. Elle est utile pour modéliser les séquencements acceptés (motifs), contraintes fréquentes dans les problèmes de planification. Elle permet aussi de reformuler d'autres contraintes globales [1] ou encore de modéliser des contraintes définies en extension. Une autre application de `regular` est de modéliser une contrainte glissante : Récemment, Bessière et al. [2] ont introduit la méta-contrainte `slide`. Dans sa forme la plus générale, `slide` prend comme arguments une matrice de variables Y de taille $n \times p$ et une contrainte C d'arité pk avec $k \leq n$. `slide`(Y, C) est satisfaite si et seulement si $C(y^1_{i+1}, \ldots, y^p_{i+1}, \ldots, y^1_{i+k}, \ldots, y^p_{i+k})$ est satisfaite pour tout $0 \leq i \leq n - k$. En utilisant la décomposition proposée dans [1], `regular`(X, Π) peut être reformulée comme `slide`$([Q, X], C_\Delta)$, avec Q la séquence de variables d'état et C_Δ la contrainte de transition $C_\Delta(q, x, q', x') \equiv (q, x, q') \in \Delta$. Inversement [2], une contrainte `slide` peut être reformulée comme une contrainte `regular` mais cela nécessite d'énumérer tous les n-uplets valides de C. Cette reformulation peut cependant s'avérer utile dans le cadre de la planification (notamment en car-scheduling) pour modéliser une contrainte de cardinalité glissante, aussi connue sous le nom de `sequence`. De puissants algorithmes spécialisés existent pour cette contrainte (voir e.g. [7]). L'avantage de la reformulation est qu'elle permet d'intégrer l'automate résultant avec d'autres règles de motifs comme nous le montrerons dans la Section 4. Enfin, notons l'existence de travaux liés aux contraintes dans les grammaires à contexte non-contraint (voir e.g. [6]) bien que les langages rationnels soient généralement suffisants dans le cadre de la planification de personnel.

2.2 Bornes de coûts et cardinalités

Les problèmes de planification de personnel sont en général définis comme des problèmes d'optimisation. La plupart du temps, le critère à optimiser est un *coût cumulatif*, c'est à dire la somme des coûts associés à chacune des activités d'un travailleur. Un tel coût peut avoir différentes significations : il peut représenter un coût financier, une préférence, ou un compteur d'actions. Ainsi, concevoir un emploi du temps valide pour un travailleur consiste à définir une séquence d'activités respectant des motifs donnés et dont le coût est borné. Cela peut être spécifié à l'aide d'une contrainte `cost-regular` [4]. Soit $c = (c_{ia})_{i \in [1..n] \times a \in \Sigma}$ une matrice de coûts des activités et $z \in [\underline{z}, \overline{z}]$ une variable bornée ($\underline{z}, \overline{z} \in \mathbb{R}$), `cost-regular`$(X, z, \Pi, c)$ est satisfaite si et seulement si `regular`(X, Π) est satisfaite et $\sum_{i=1}^{n} c_{ix_i} = z$. On notera que la contrainte `knapsack` [13] est un cas spécial. À moins que $P = NP$, la CAG de la contrainte `knapsack` ne peut être assurée au mieux qu'en temps pseudo-polynomial (polynomial en les valeurs des bornes de z). Par conséquent, assurer la CAG de `cost-regular` est NP-difficile.

La définition de `cost-regular` révèle une décomposition naturelle en une contrainte `regular` reliée à une contrainte `knapsack`. En réalité, cette décomposition est équivalente à celle proposée par Beldiceanu

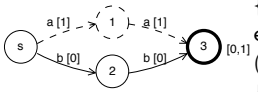
et al. [1] lorsque l'on s'intéresse à un coût cumulatif[2] : des variables de coûts k_i sont associées aux variables d'état q_i, avec $k_0 = 0$ et $k_n = z$, et plusieurs fonctions arithmétiques et contraintes `element` modélisent les `knapsack` et les contraintes de transition. Cette formulation peut être ré-écrite comme $\texttt{slide}([Q, X, K], C^c_\Delta)$, avec $C^c_\Delta(q_{i-1}, x_{i-1}, k_{i-1}, q_i, x_i, k_i) \equiv (q_{i-1}, x_{i-1}, q_i) \in \Delta \wedge k_i = k_{i-1} + c_{ix_i}$. En fonction de la taille des domaines des variables de coûts, la CAG peut être assurée sur `knapsack` dans un temps raisonnable. Cependant, comme l'hypergraphe des contraintes du modèle décomposé n'est plus Berge-acyclique mais α-acyclic, on doit assurer la consistance deux-à-deux des variables partagées – une paire (q_i, k_i) d'état et variable de coût – par les contraintes de transition afin d'atteindre la CAG du modèle complet. Une option similaire proposée pour `slide` [2] est d'assurer l'AC sur l'encodage dual de l'hypergraphe des contraintes C^c_Δ, mais là-aussi cela nécessite d'expliciter tous les n-uplets supports, rendant son utilisation non fonctionnelle.

L'algorithme de filtrage présenté dans [4] pour `cost-regular` est une légère adaptation[3] de l'algorithme de Pesant pour `regular`. Il s'appuie sur le calcul de plus court et plus long chemins dans le graphe déployé Π_n évalué par les coûts des transitions. À chaque noeud (i, q) d'une couche i de Π_n sont associés deux variables de coûts bornées k^-_{iq} et k^+_{iq}, modélisant les longueurs des chemins, respectivement depuis la couche 0 jusqu'à la couche (i, q) et depuis (i, q) jusqu'à la couche n. Les variables de coûts peuvent être initialisée de façon triviale pendant la construction de Π_n : k^-_{iq} dans le sens avant et k^+_{iq} dans le sens arrière. Les bornes de la variable z sont alors réduites selon la condition $z \subseteq k^+_{0s}$. Inversement, un arc $((i-1, q), a, (i, q')) \in \delta_i$ peut être retiré dès que :

$$\underline{k^-_{(i-1)q}} + c_{ia} + \underline{k^+_{iq'}} > \overline{z} \quad \text{or} \quad \overline{k^-_{(i-1)q}} + c_{ia} + \overline{k^+_{iq'}} > \underline{z}.$$

Comme le graphe Π_n est acyclique, surveiller les variables de coût, c'est à dire les plus court et plus long chemins, peut être réalisé par l'algorithme de parcours avec la même complexité $O(|\Delta_n|)$ que pour maintenir la connexité du graphe dans `regular`.

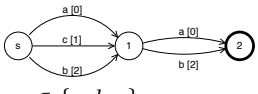
Comme évoqué précédemment, cet algorithme atteint un niveau hybride de consistance sur `cost-regular`. En fait, il assure pour le modèle décomposé une consistance entre les variables d'état et les bornes de la variable de coût associée selon la relation $q_i = (i, q) \iff k_i = k^-_{iq}$. L'algorithme supplante donc le modèle décomposé `knapsack`∧`regular` quand seule la consistance aux bornes est appliquée sur les variables de coûts. Sinon, si l'AC est assurée sur `knapsack` alors les deux approches sont incomparables, comme nous le montrent les deux exemples présentés en Figures 2 et 3.



$x_1 \in \{a, b\}$,
$x_2 \in \{a, b\}$,
$z \in [0, 1]$.

$(q_0, x_1, q_1, j_1) \in \{(s, a, 1, 1), (s, b, 2, 2)\}$,
$\texttt{element}(k_1, j_1, (k_0 + 1, k_0))$,
$(q_1, x_2, q_2, j_2) \in \{(1, a, 3, 1), (2, b, 3, 2)\}$,
$\texttt{element}(z, j_2, (k_1 + 1, k_1))$,
$q_0 \in \{s\}, q_1 \in \{1, 2\}, q_2 \in \{3\}$,
$k_0 \in \{0\}, k_1 \in \{0, 1\}$,
$x_1 \in \{a, b\}, x_2 \in \{a, b\}, z \in \{0, 1\}$.

FIG. 2 – Soit l'AFD représenté avec les coûts entre crochets appliqué à $X = (x_1, x_2) \in \{a, b\} \times \{a, b\}$ et $z \in [0, 1]$. L'algorithme `cost-regular` (sur la gauche) filtre les arcs en pointillé et atteint ainsi la CAG. L'arc-consistance sur le modèle décomposé (sur la droite) n'atteint pas la consistance globale.



$x_1 \in \{a, b, c\}$,
$x_2 \in \{a, b\}$,
$z \in [2, 2]$.

$(x_1, j_1) \in \{(a, 1), (c, 2), (b, 3)\}$,
$\texttt{element}(k_1, j_1, (k_0, k_0, k_0 + 2))$,
$(x_2, j_2) \in \{(a, 1), (b, 2)\}$,
$\texttt{element}(z, j_2, (k_1, k_1 + 2))$,
$k_0 \in \{0\}, k_1 \in \{0, 2\}$,
$x_1 \in \{a, b, c\}, x_2 \in \{a, b\}, z \in \{2\}$.

FIG. 3 – Soit maintenant l'AFD représenté ci-dessus appliqué à $X = (x_1, x_2) \in \{a, b, c\} \times \{a, b\}$ et $z \in [2, 2]$. Assurer l'AC sur le modèle décomposé (sur la droite) atteint la CAG. L'algorithme cost-regular (sur la gauche) n'atteint pas la CAG puisque les chemins minimum et maximum traversant l'arc $x_1 = c$ sont consistants avec les bornes de z.

2.3 La contrainte `multicost-regular`

Une généralisation naturelle de `cost-regular` est de traiter plus d'un coût cumulatif : soit un vecteur $Z = (z^0, ..., z^R)$ de variables bornées et $c = (c^r_{ia})^{r \in [0..R]}_{i \in [1..n], a \in \Sigma}$ une matrice de coûts associés aux activités, $\texttt{muticost-regular}(X, Z, \Pi, c)$ est satisfaite si et seulement si $\texttt{regular}(X, \Pi)$ est satisfaite et $\sum_{i=1}^n c^r_{ix_i} = z^r$ pour tout $0 \leq r \leq R$. Une telle généralisation trouve son intérêt dans le cadre de la planification de personnel. En effet, outre le coût financier et les restrictions liées aux motifs, l'emploi du temps individuel est en général assujetti à une contrainte `global-cardinality` limitant le nombre d'occurrences de chaque valeur dans la

[2] Le modèle dans [1] peut traiter non seulement les sommes mais aussi diverses fonctions arithmétiques de coûts, mais aucun exemple de cette utilisation n'est donnée.

[3] Auparavant, l'algorithme était appliqué partiellement – uniquement pour la minimisation – au cas spécial `soft-regular[hamming]` dans [1] et dans [14].

séquence. Ces bornes peuvent réduire drastiquement l'ensemble des solutions valides et donc le graphe support de la contrainte `regular`. Ainsi, traiter les coûts au sein de la contrainte `regular` permet également de réduire la taille du graphe support. Puisqu'il s'agit ici d'une généralisation de la contrainte `cost-regular` ou de la contrainte `global-sequencing` [10], nous ne pouvons pas espérer atteindre la CAG en un temps polynomial. Le modèle de Beldiceanu et al [1] – ou la contrainte `slide` – s'intéresse également à la prise en compte de plusieurs coûts, mais encore une fois le traitement proposé consiste à décomposer une contrainte `regular` en une conjonction de contraintes en extension liées à une contrainte `knapsack` pour chaque coût.

Ainsi, nous proposons d'exploiter la structure du graphe support Π_n pour avoir une bonne propagation relaxée de `muticost-regular`. Les problèmes d'optimisation sous-jacent à `cost-regular` étaient des problèmes de plus court et plus long chemins dans Π_n. Les problèmes d'optimisation sous-jacent à `multicost-regular` sont les Resource Constrained Shortest and Longest Path Problems (RCSPP and RCLPP) dans Π_n. Le RCSPP (resp. RCLPP) a pour but de trouver le plus court (resp. plus long) chemin entre une source et un puits dans un graphe orienté multi-valué, de telle sorte que les quantités de ressources accumulées sur les arcs ne dépassent pas certaines limites. Même pour des graphes acycliques, ce problème est connu pour être NP-difficile[5]. Deux approches sont le plus souvent utilisées pour résoudre RCSPP [5] : la programmation dynamique et la relaxation lagrangienne. Les méthodes basées sur la programmation dynamique étendent les algorithmes usuels de plus court chemin par enregistrement des coûts sur chaque dimension à chaque noeud du graphe. De même que dans `cost-regular`, cela peut aisément être adapté pour le filtrage en convertissant ces étiquettes de coûts en variables de coûts mais la mémoire nécessaire pour l'algorithme serait trop importante. Au lieu de cela nous avons investigué l'approche par relaxation lagrangienne, qui peut aussi être aisément adaptée au filtrage par l'algorithme `cost-regular` sans augmenter de manière drastique les besoins en mémoire.

3 Filtrage basé sur la relaxation lagrangienne

Sellmann [11] a établi le principe de l'utilisation de la relaxation lagrangienne d'un programme linéaire pour filtrer des contraintes d'optimisation. Nous appliquons ce principe au RCSPP/RCLPP pour le filtrage de `multicost-regular`. L'algorithme résultant est un simple procédé itératif dans lequel le filtrage est pris en charge par `cost-regular` dans Π_n pour différentes fonctions de coûts agrégés. Dans cette section, nous présentons le modèle usuel de relaxation lagrangienne pour le RCSPP, ainsi que la méthode de résolution du dual lagrangien basée sur l'algorithme du sous-gradient. Enfin nous montrons comment l'adapter pour le filtrage de `multicost-regular`.

Relaxation lagrangienne du RCSPP. On considère un graphe orienté $G = (V, E)$ muni d'un nœud source s et d'un nœud puits t, et un ensemble de ressources $\mathcal{R}_1, ..., \mathcal{R}_R$. Pour tout $1 \leq r \leq R$, $\overline{z_r}$ (resp. $\underline{z_r}$) dénote la consommation maximum (resp. minimum[4]) de la ressource $\mathcal{R}_\nabla$ sur un chemin de s à t dans G, et c^r_{ij} est la consommation de la ressource $\mathcal{R}_\nabla$ sur tout arc $(i, j) \in E$. Le RCSPP se modélise par le programme linéaire en nombres binaires suivant :

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij} \qquad \text{s.t.} \qquad (1)$$

$$\underline{z^r} \leq \sum_{(i,j)\in E} c^r_{ij}x_{ij} \leq \overline{z^r} \qquad \forall r \in [1..R] \quad (2)$$

$$\sum_{j\in V} x_{ij} - \sum_{j\in V} x_{ji} = \begin{cases} 1 & \text{if } i = s, \\ -1 & \text{if } i = t, \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i \in V \qquad (3)$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E. \quad (4)$$

Dans ce modèle, une variable de décision x_{ij} indique si l'arc (i, j) appartient à un chemin solution. Les contraintes (2) sont les contraintes de ressources et les contraintes (3) sont les contraintes de chemin.

Le principe de la relaxation lagrangienne repose sur le transfert de contraintes dites difficiles dans la fonction objectif où elles sont ajoutées avec un coût de violation $u \geq 0$ appelé *multiplicateur lagrangien*. Le programme linéaire résultant, appelé *sous-problème lagrangien de paramètre u* est une relaxation du problème original. Résoudre le *dual lagrangien* consiste à trouver les multiplicateurs $u \geq 0$ qui fournissent la meilleur relaxation, c.à.d. la plus grande borne inférieure possible.

Les contraintes difficiles du *RCSPP* sont les $2R$ contraintes de ressources (2). En effet, la suppression de ces contraintes produit un problème de plus court chemin qui se résout en temps polynomial. Soit P l'ensemble des solutions $x \in \{0, 1\}^E$ satisfaisant les contraintes (3) : P est l'ensemble des chemins de s à t dans G. Le sous problème lagrangien de paramètre $u = (u_-, u_+) \in \mathbb{R}^{2R}_+$ s'écrit :

$$SP(u) : f(u) = \min_{x\in P} cx + \sum_{r=1}^{R} u^r_+(c^r x - \overline{z^r}) - \sum_{r=1}^{R} u^r_-(c^r x - \underline{z^r})$$

[4] Dans la définition originale du RCSPP, il n'y a pas de borne inférieure sur la capacité : $\underline{z_r}$

Une solution optimale x^u pour $SP(u)$ correspond ainsi à tout plus court chemin dans le graphe $G(u) = (V, E, c(u))$ défini par :

$$c(u) = c + \sum_{r=1}^{R} (u_+^r - u_-^r)c^r \tag{5}$$

et son coût est égal à :

$$f(u) = c(u)x^u + \kappa_u, \tag{6}$$
$$\text{avec } \kappa_u = \sum_{r=1}^{n} (u_-^r \underline{z^r} - u_+^r \overline{z^r}).$$

Résoudre le dual lagrangien. Le dual lagrangien consiste à trouver la meilleur borne inférieure $f(u)$, c.à.d. à maximiser la fonction concave linéaire par morceaux f :

$$LD: \quad f_{LD} = \max_{u \in \mathbb{R}_+^{2R}} f(u) \tag{7}$$

Plusieurs algorithmes permettent de résoudre le dual lagrangien. Nous nous sommes intéressés ici à l'algorithme du sous-gradient [12] pour sa simplicité d'utilisation et parce qu'il ne nécessite pas l'utilisation d'un solveur linéaire. L'algorithme du sous-gradient résout de manière itérative, pour différentes valeurs de u, le sous-problème $SP(u)$. À chaque itération, le vecteur u est mis à jour suivant la direction d'un super-gradient Γ de f avec un pas de longueur μ : $u^{p+1} = max\{u^p + \mu_p \Gamma(u^p), 0\}$. Il existe plusieurs manières de choisir la longueur de pas afin de garantir la convergence vers f_{LD} de l'algorithme du sous-gradient (voir par ex. [3]). Notre implémentation repose sur une longueur de pas standard $\mu_p = \mu_0 \epsilon^p$ avec μ_0 et $\epsilon < 1$ « suffisamment » grands (nous avons choisi empiriquement $\mu_0 = 10$ et $\epsilon = 0.8$). Le super-gradient $\Gamma(u)$ est calculé à partir de la solution optimale $x^u \in P$ retournée par la résolution de $SP(u)$: $\Gamma(u) = ((c^r x^u - \overline{z^r})_{r \in [1..R]}, (\underline{z^r} - c^r x^u)_{r \in [1..R]})$.

De la relaxation lagrangienne au filtrage. Comme montré dans [11], si une valeur est inconsistante dans au moins un des sous-problèmes lagrangiens, alors elle l'est également pour le problème original. C'est la clé du filtrage par relaxation lagrangienne.

Théorème 1. *(i) Soit P un programme linéaire de valeur minimale $f^* \leq +\infty$, soit $\overline{z} \leq +\infty$ une borne supérieure de f^*, et soit $SP(u)$ un sous-problème lagrangien de P de valeur minimale $f(u) \leq +\infty$. Si $f(u) > \overline{z}$ alors $f^* > \overline{z}$.*
(ii) Soit x une variable de P et v une valeur de son domaine. Soit $P_{x=v}$ (resp. $SP(u)_{x=v}$) la restriction de P (resp. $SP(u)$) à l'ensemble des solutions telles que $x = v$, et soit $f^*_{x=v} \leq +\infty$ (resp. $f(u)_{x=v} \leq +\infty$) sa valeur minimale. Si $f(u)_{x=v} > \overline{z}$ alors $f^*_{x=v} > \overline{z}$.

Démonstration. La proposition (i) du Théorème 1 est triviale car, $SP(u)$ étant une relaxation de P, $f(u) \leq f^*$. La proposition (ii) se déduit de (i) et du fait qu'ajouter une contrainte $x = v$ à P puis appliquer une relaxation lagrangienne ou appliquer la relaxation lagrangienne à P puis ajouter la contrainte $x = v$ produit la même formulation. □

On établit la relation entre une instance de `muticost-regular`(X, Z, Π, c), avec $|Z| = R + 1$, et deux instances du RCSPP et du RCLPP comme suit : On sélectionne une variable de coût, par exemple z^0, et on crée R ressources, une par variables de coûts restantes. Le graphe considéré est $G = (\Pi_n, c^0)$. Une solution réalisable du RCSPP (resp. RCLPP) est un chemin dans Π_n depuis la source (dans la couche 0) vers un puits (dans la couche n) dont la consommation pour chaque ressource $1 \leq r \leq R$ est au moins $\underline{z^r}$ et au plus $\overline{z^r}$. De plus, nous voulons limiter par une borne supérieure $\overline{z^0}$ la valeur minimale du RCSPP (resp. une borne inférieure $\underline{z^0}$ la valeur maximale du RCLPP). Les arcs de Π_n coincident avec les variables binaires du modèle linéaire de ces deux instances.

Intéressons nous à un sous-problème lagrangien $SP(u)$ du RCSPP (l'approche est symétrique pour le RCLPP). Une légère modification de l'algorithme de `cost-regular` permet de résoudre $SP(u)$, mais aussi de filtrer des arcs de Π_n selon le Théorème 1 et de mettre à jour la borne inférieure $\underline{z^0}$. L'algorithme considère d'abord $c^0(u)$, défini par l'équation (5), comme valuation du graphe Π_n. Puis il calcule pour chaque noeud (i, q), le plus court chemin $\underline{k_{iq}^-}$ depuis la couche 0 et le plus court chemin $\underline{k_{iq}^+}$ vers la couche n. On obtient la valeur optimale $f(u) = \underline{k_{0s}^+} + \kappa_u$. Comme il s'agit d'une borne inférieure pour z^0, on peut éventuellement mettre à jour $\underline{z^0} = \max\{f(u), \underline{z^0}\}$. Les arcs sont ensuite à nouveau parcourus, afin de filter tout arc $((i-1, q), a, (i, q')) \in \delta_i$ tel que $\underline{k_{(i-1)q}^-} + c^0(u)_{ia} + \underline{k_{iq'}^+} > \overline{z^0} - \kappa_u$.

L'algorithme de filtrage complet de `multicost-regular` procède comme suit : u est initialisé à 0, l'algorithme du sous-gradient guide le choix des sous-problèmes lagrangiens auxquels est appliqué l'algorithme de filtrage décrit ci-dessus. Dans nos expérimentations, le nombre d'itérations de l'algorithme du sous-gradient est limité à 20 (il termine le plus souvent en 5 ou 6 itérations). On résout d'abord le problème de minimisation (RCSPP) puis celui de maximisation (RCLPP). Enfin, l'algorithme original de `cost-regular` est utilisé sur chaque variable de coûts de manière indépendante afin de réduire leurs bornes (Il peut également filtrer certains arcs, même si cela n'est jamais arrivé au cours de nos tests).

4 Modéliser des problèmes de planification de personnel

Dans cette section, nous montrons comment modéliser les principales règles métiers qui apparaissent dans les problèmes de planification de personnel (Personnel Scheduling Problem ou PSP) à l'aide d'une seule instance de `multicost-regular`. Nous démontrons ainsi la simplicité d'utilisation d'un tel modèle en automatisant la modélisation de PSP.

4.1 Réglementations standards

Dans un PSP, il existe plusieurs types de règles métiers. Ces dernières peuvent être regroupées en catégories : les motifs rationnels, les contraintes de cardinalité et les contraintes glissantes.

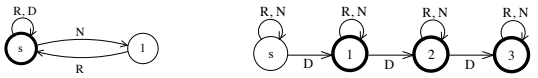
Afin d'illustrer ces catégories, considérons une planification sur 7 jours de 3 types d'activités à affecter par jour : nuit (N), jour (D) ou repos (R). Une planification s'écrit par exemple :

R	R	D	D	N	R	D

Les motifs rationnels sont modélisables directement par un AFD. Par exemple, la règle « une période de nuit est suivie par un repos » se représente par l'automate 4 (A). Une règle peut être définie aussi bien par un motif interdit ou obligatoire. Dans le premier cas, il suffira de construire l'automate complémentaire.

Les règles de cardinalité permettent de borner le nombre d'occurrences d'une ou plusieurs activités pendant une période donnée. De telles règles peuvent se modéliser à l'aide d'un automate ou d'un compteur. L'AFD de la Figure 4 (B) représente la règle « au moins 1 et au plus 3 journées chaque semaine ». Cependant, avec cette formulation, on s'aperçoit que l'état initial a été dupliqué 3 fois afin de représenter le nombre maximum de transitions D acceptables. Cela peut devenir problématique lorsque le nombre maximum d'occurrences augmente. Dans ce cas, il est plus approprié d'utiliser un compteur, ainsi il suffit de créer une nouvelle variable de coûts $z^r \in [1, 3]$ avec $c^r_{ia} = 1$ si $a = D$ et $c^r_{ia} = 0$ sinon. Plus généralement, il se peut que l'on veuille contraindre la cardinalité d'un motif. On peut également le gérer à l'aide d'un coût en isolant le motif dans l'automate décrivant les emplois du temps valides, puis d'ajouter un coût de 1 à la transition menant à ce motif.

Les contraintes glissantes peuvent également être modélisées par un AFD grâce à la reformulation proposée par Bessière et al. [2]. Cependant, la séquence glissante ne doit pas être trop longue car la reformulation impose de décrire tous les n-uplets réalisables de la contrainte glissante.



(A) Exemple de motif rationnel (B) Exemple de règle de cardinalité.

FIG. 4 – Exemples de différentes règles métiers sous forme d'automates.

4.2 Génération systématique de `multicost-regular`

Un formalisme basé sur un schéma XML et permettant de décrire des instances de PSP a été proposée dans [8]. Nous avons développé un framework capable d'interpréter ces fichiers XML afin de générer automatiquement un modèle PPC basé sur `multicost-regular`. Tout d'abord nous associons une catégorie de règles métier à chaque balise XML. Ainsi, nous pouvons pour chaque règle générer soit l'automate soit le compteur correspondant selon la catégorie à laquelle elle appartient. Par exemple, le motif interdit « pas de travail de jour après une nuit » sera défini dans le fichier XML ainsi :

```
<Pattern weight="1350"><Shift>N</Shift>
<Shift>D</Shift></Pattern>
```

Il sera automatiquement traduit en une expression rationnelle équivalente $(D|N|R) * ND(D|N|R)*$. Nous utilisons une librairie java spécialisée pour la manipulation des automates[5] afin de créer les AFD à partir des expressions rationnelles et de pouvoir par la suite les manipuler. Les fonctions complément, intersection et minimisation sont utilisées pour produire un AFD unique. Une fois cet automate construit, les règles qui produisent des compteurs sont traitées. Une instance de `multicost-regular` est générée pour chaque employé. Les dernières contraintes intégrées au modèle CSP sont les contraintes transversales. Par exemple, les demandes d'activités pour chaque jour sont représentées par une contrainte globale de cardinalité `gcc`. Notons que nous ne sommes pas capables de gérer les coûts de violation d'une règle dans `multicost-regular`. De plus, nous ne savons pas transformer de manière automatique un AFD afin d'intégrer des règles de cardinalité sur des motifs.

4.3 Deux cas de planification de personnel

Nous avons d'abord étudié un problème appelé *GPost* [8]. Ce PSP consiste à construire un emploi du

[5] http ://www.brics.dk/automaton/

temps valide de 28 jours pour 8 personnes. Chaque jour, un employé doit être affecté à une activité de jour (D), de nuit (N) ou de repos (R). Chaque employé est lié à un contrat (temps plein ou partiel) défini par un ensemble de règles. Les règles de motifs rationnels identifiées sont : « Un repos doit durer au moins deux jours », « Les WE consécutifs travaillés sont limités » et « Certaines successions de périodes sont interdites ». En utilisant notre méthode de modélisation automatique, nous construisons un AFD pour chaque type de contrat. Les contraintes de cardinalités identifiées : « Au plus n jours travaillés par mois », « Le nombre d'occurences de certains types d'activité est limité » et « Le nombre de jours travaillés par semaine est limité ». Les besoins d'activités journalières ainsi que les disponibilités des employés sont aussi définies et modélisées. Le coût de violation des règles a été ignoré, de même que la première règle de motif afin d'éviter l'irréalisabilité du problème.

Notre deuxième étude porte sur l'ensemble des problèmes de référence générés par Demassey et al. [4]. Les règles métiers sont issues d'un problème de planification de personnel réel. L'objectif est ici de construire un emploi du temps constitué de 96 périodes de 15 minutes pour une seule personne. On fait correspondre à chaque période soit une activité travaillée, une pause, un repas ou du repos. Chaque affectation possède son propre coût. On cherche à trouver l'emploi du temps de coût minimum respectant toutes les règles métiers, des motifs rationnels : « Une activité dure au moins une heure », « Changer d'activité nécessite une pause ou un repas », « Pause, repas et repos ne peuvent être consécutifs », « Les repos doivent être placés en début et en fin de journée », et « une pause dure 15 minutes ». ainsi que des contraintes de cardinalité : « Au moins 1 et au plus 2 pauses par jour », « Au moins un repas par jour » et « Entre 3 et 8 heures d'activité par jour ». À ces règles métiers s'ajoute l'interdiction de faire certains couples période-activité. Des contraintes unaires permettent de les modéliser.

5 Expérimentations

Nos expérimentations ont été effectuées sur Intel Core 2 Duo 2Ghz avec 2048Mo de RAM sous OS X. Les deux PSP présentés ont été résolus en utilisant la librairie de contraintes en java *CHOCO*. L'heuristique de sélection de valeurs par défaut, *min value*, a été appliquée.

5.1 À propos de la taille de l'automate

Comme vu dans la section 2.1, la complexité de l'algorithme de filtrage de `regular` dépend de la taille du graphe déployé, égale dans le pire des cas au produit du nombre de variables par la taille de l'automate. Si un tel algorithme peut sembler inapplicable en théorie sur des automates de trop grandes tailles, nos résultats expérimentaux mettent deux choses en évidence. Tout d'abord, les opérations utilisées pour construire automatiquement un AFD à partir de plusieurs règles tendent à générer un automate déjà partiellement déployé (suite aux intersections) et à réduire le nombre d'états redondants dans les différentes couches (suite à la minimisation). Grâce à cela, le graphe déployé généré pendant la première phase d'initialisation de la contrainte peut être largement plus petit que l'automate passé en paramètre Π. Aussi, la seconde phase d'initialisation réduit considérablement encore le graphe déployé Π_n car de nombreux états finaux ne peuvent pas être atteints en exactement n transitions. Le Tableau 1 donne le nombre de noeuds et d'arcs dans les différents automates construits lors de la construction d'une contrainte `multicost-regular` pour le problème *GPost* avec, dans l'ordre : la somme des tailles des AFD générés pour chacune des règles, la taille de l'AFD Π après intersection et minimisation, l'AFD déployé après les deux phases d'initialisation de la contrainte (phase Avant, puis phase Arrière Π_n).

Contract	Count	$\sum$AFD	Π	Avant	Π_n
Fulltime	# Nodes	5782	682	411	230
	# Arcs	40402	4768	1191	400
Parttime	# Nodes	4401	385	791	421
	# Arcs	30729	2689	2280	681

TAB. 1 – Illustration de la réduction des graphes avant résolution.

5.2 Comparaisons expérimentales

Dans la section précédente, nous avons vu la facilité de modélisation offerte par `multicost-regular`. Néanmoins, il n'y aurait pas d'intérêt à définir une telle contrainte si la rapidité de résolution était impactée. Nous avons donc comparé notre algorithme avec un modèle décomposé, liant une contrainte `regular` (ou `cost-regular` pour l'optimisation) à une contrainte `global-cardinality` (`gcc`).

Les résultats expérimentaux de l'instance *GPost* sont présentés dans le Tableau 2. Le modèle pour ce problème est composé de 8 `multicost-regular` (ou 8 `regular` et `gcc`) – une par personne – et de 28 `gcc` transversales – une par période. Deux variantes de l'instance sont considérées : la première (resp. seconde) ligne du tableau correspond au problème sans (resp. avec) la contrainte glissante limitant le nombre de week-ends consécutifs travaillés. Nous avons testé plusieurs heuristiques de choix de variables. Choisir les

	multicost-regular		regular ∧ gcc	
WE ?	Time (s)	# Fails	Time (s)	# Fails
no	1.94	24	12.6	68035
yes	16.0	1576	449.2	2867381

TAB. 2 – Résultats du problème *GPost*

variables correspondant à une même période d'abord donne les meilleurs résultats. Cette heuristique permet au solveur de contraintes de traiter plus efficacement les gcc transversales. Les deux modèles mènent aux mêmes solutions. En fait, le temps moyen passé pour chaque noeud est beaucoup plus important avec multicost-regular. Cependant grâce à un filtrage plus efficace, la taille de l'arbre de recherche et le temps global de résolution pour trouver une solution réalisable sont considérablement réduits.

Notre deuxième jeux de tests évalue le comportement de la contrainte multicost-regular (MCR) par rapport à cost-regular∧gcc (CR) lorsque l'on augmente la taille du graphe. Les tests sont effectués sur le problème d'optimisation défini dans la Section 4.3. Les modèles ne contiennent pas d'autre contrainte, cependant, le modèle décomposé CR nécessite l'utilisation de variables de channeling supplémentaires. L'ensemble de tests présenté dans le Tableau 3 compte 110 instances. Le nombre n d'activités varie entre 1 et 50. La taille de l'automate ($4n+7$ états et $10n+5$ transitions) varie ainsi de 11 à 207 états et de 15 à 505 transitions. Les coûts d'affectation ont été générés aléatoirement. Plusieurs heuristiques de choix de variables ont été testées et la meilleure pour chaque modèle a été retenue. Les résultats du modèle CR sont plus sensibles au choix de l'heuristique que le modèle MCR.

Le première partie du Tableau 3 montre que le modèle MCR permet de résoudre toutes les instances (Colonne #) en moins de 15 secondes pour les plus difficiles (Colonne t). Le nombre moyen de backtracks (Colonne bt) reste stable et bas lorsque n augmente. Le modèle CR est en revanche beaucoup plus sensible à l'augmentation de n comme le montre la seconde partie du tableau. En effet, la taille du graphe augmentant, celui-ci contient de plus en plus de chemins violant les contraintes de cardinalité. Ces chemins ne sont pas supprimés par le filtrage résultant du modèle décomposé CR. En tout, 40 instances de plus de 8 activités n'ont pas été résolues en moins de 30 minutes (Colonne #). Sur ces instances, l'optimum est atteint (mais non prouvé) dans seulement 4 cas et la déviation moyenne de la meilleure solution trouvée à l'optimum est de $4,7\%$. Si on ne s'intéresse qu'aux instances résolues, le temps de résolution (t) et le nombre de backtracks (bt) sont toujours beaucoup plus élevés avec le modèle décomposé CR.

	MCR model			CR model		
n	#	t	bt	#	t	bt
1	10	0.6	49	10	1.1	292
2	10	0.8	54	10	2.4	539
4	10	1.5	65	10	13.5	1638
6	10	1.6	44	10	53.6	4283
8	10	2.1	51	9	209.2	5132
10	10	2.4	58	7	283.5	6965
15	10	3.8	59	6	283.9	4026
20	10	4.9	49	6	311.8	4135
30	10	6.9	51	1	313.0	4303
40	10	13.4	68	0	-	-
50	10	14.4	51	1	486.0	1406

TAB. 3 – Résultats des problèmes d'optimisation

6 Conclusion

Dans ce papier, nous introduisons la contrainte globale multicost-regular et proposons une implémentation simple d'un filtrage basé sur une relaxation lagrangienne. Les expérimentations sur des problèmes de planification de personnel montrent son efficacité et son extensibilité par rapport aux modèles décomposés utilisés habituellement pour décrire des règles métiers. De plus, nous explorons une méthode systématique pour construire des instances de multicost-regular à partir d'un ensemble de règles métiers. Nous chercherons par la suite à créer un outil automatisé lié au solveur de contraintes *CHOCO* capable de modéliser et de résoudre une grande variété de problèmes d'emploi du temps.

Remerciements

Nous remercions Mats Carlsson pour avoir fourni l'exemple illustrant le défaut de propagation de l'algorithme de cost-regular. Nous remercions également Christian Schulte pour ses commentaires avisés sur le papier et ses conseils pour améliorer la contrainte.

Références

[1] N. Beldiceanu, M. Carlsson, R. Debruyne, and T. Petit. Reformulation of Global Constraints Based on Constraint Checkers. *Constraints*, 10(3), 2005.

[2] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints : The SLIDE and REGULAR constraints. In *SARA*, pages 80–92, 2007.

[3] S. Boyd, L. Xiao, and A. Mutapcic. Subgradient methods. *lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004, 2003.

[4] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4) :315–333, 2006.

[5] G. Handler and I. Zang. A dual algorithm for the restricted shortest path problem. *Networks*, 10 :293–310, 1980.

[6] Serdar Kadioglu and Meinolf Sellmann. Efficient context-free grammar constraints. In *AAAI*, pages 310–316, 2008.

[7] M. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-Based Propagators for the *sequence* and Related Global Constraints. In *Proceedings of CP'2008*, volume 5202 of *LNCS*, pages 159–174, 2008.

[8] Personnel Scheduling Data Sets and Benchmarks. `http ://www.cs.nott.ac.uk/ tec/NRP/`.

[9] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'2004*, pages 482–495, 2004.

[10] Jean-Charles Régin and Jean-Francois Puget. A filtering algorithm for global sequencing constraints. In *CP*, pages 32–46, 1997.

[11] Meinolf Sellmann. Theoretical foundations of CP-based lagrangian relaxation. *Principles and Practice of Constraint Programming –CP 2004*, pages 634–647, 2004.

[12] NZ Shor, KC Kiwiel, and A Ruszcayǹski. Minimization methods for non-differentiable functions. *Springer-Verlag New York, Inc.*, 1985.

[13] M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints, 2001.

[14] Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming : Flow-based soft global constraints. *J. Heuristics*, 12(4-5) :347–373, 2006.