TER - Rapport de Stage : Dérivation d'un algorithme de filtrage à partir d'une relaxation linéaire d'un modèle PLNE de la contrainte globale *noOverlap*.

Daniel Blévin

26 novembre 2017

Remerciements Tous mes remerciements vont à Monsieur Hadrien Cambazard professeur associé à G-SCOP.

Table des matières

Première partie

Contexte

1 La programmation par contraintes

La programmation par contraintes (PPC ou CP pour Constraint Programming) est un paradigme de programmation apparu à la fin des années 1980 issu de la programmation logique et de l'intelligence artificielle permettant la résolution de problèmes combinatoires complexes.

Dans ce paradigme l'on définit un problème de manière déclarative par un ensemble de relations logiques, des contraintes, qui s'appliquent à des variables. Les contraintes se différencient des primitives utilisées en programmation impérative, elles ne définissent pas une suite d'étapes à exécuter, mais les propriétés de la solution recherchée. Chaque variable possède un ensemble de valeurs possibles appelées le domaine de la variable. On considérera dans ce rapport uniquement les problèmes dont le domaine des variables est fini.

Lorsqu'on affecte une valeur de son domaine à une variable, cette variable est dite instanciée. Une affectation viole une contrainte si toutes ses variables sont instanciées et que la relation associée à la contrainte n'est pas vérifiée. Une instanciation de toutes les variables d'un problème satisfaisant l'ensemble de ses contraintes est une solution du problème [3].

Plus formellement, une instance du problème de satisfaction de contraintes, ou CSP (pour constraint satisfaction problem), se définit par un quadruplet $P = (X, C, D, R)$ 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 $(X_1, \ldots, X_{j_{n_j}})$ [1].

Résoudre un CSP consiste à exhiber une unique solution ou à montrer qu'aucune solution n'existe (le problème est irréalisable). À titre d'exemple, on considère le CSP suivant : deux variables x et y ayant pour domaine respectif $\{1, 2, 3, 4\}$ et $\{1, 2, 3\}$ et deux contraintes tel que $x < y$ et $x + y \geq 5$ soit le problème suivant :

$$x < y \tag{1}$$
$$x + y \geq 5 \tag{2}$$

Une instanciation complète (toute les variables sont instanciées) du problème qui est une solution est l'instanciation des variables x et y tel que $x \leftarrow 2$ et $y \leftarrow 3$. Une instanciation complète qui n'est pas une solution : $x \leftarrow 2$ et $y \leftarrow 2$ car $x = y$ ce qui viole la première contrainte du CSP.

Les contraintes peuvent être de différents types. Elles peuvent être définies par une équation (1) (2), une relation logique simple (tel qu'une implication) ou encore par un prédicat portant sur n variables. Ce dernier type de contrainte est appelé contrainte globale et nous aurons l'occasion d'y revenir par la suite.

1.1 Les solveurs de contraintes

Dans l'exemple précédent nous avons instancié les variables de manière intuitive. Pour résoudre des problèmes réels, on utilise un solveur de contraintes. Un solveur de contraintes résout un CSP par recherche arborescente avec une phase propagation des contraintes à chaque nœuds de l'arbre.

Prouver la satisfiabilité d'un CSP est un problème NP-complet et trouver une solution admissible est un problème NP-difficile [2]. C'est pour quoi on adjoint à l'algorithme de recherche une phase de propagation qui par des techniques de cohérence ou par l'utilisation de contraintes globales vont réduire l'espace de recherche.

Une notion importante de la phase de propagation est celle de support. L'affectation d'une valeur v à une variable x possède un support en regard d'une contrainte C s'il existe une affectation des variables restantes tel que la contrainte C est satisfaite. Si une valeur n'a pas de support en regard d'une des contraintes, elle peut être retirée du domaine de la variable (on dit que la valeur est incohérente). L'algorithme qui détermine la cohérence des valeurs est appelée algorithme de filtrage. Ces algorithmes sont itérés jusqu'à ce qu'aucune valeur supplémentaire ne puisse être détectée et retirée. Ce processus est appelé propagation des contraintes [4].

Si on reprend l'exemple précédent, la propagation de la contrainte (1) permet d'éliminer la valeur 1 du domaine de y. En effet 1 est la plus petite valeur de x et ne pourra donc jamais être inférieure à toutes valeurs affectées à y. Les valeurs 3 et 4 du domaine de x sont également incohérentes puisque 3 est la plus grande valeur de y.

De même, la propagation de la contrainte (2) permet d'éliminer du domaine de x la valeur 1 puisque qu'aucune valeur du domaine de y ne permet de satisfaire la contrainte (2) lorsque $x \leftarrow 1$.

Les domaines respectifs de x et de y sont maintenant $\{2\}$ et $\{2, 3\}$. Une nouvelle itération de la propagation de la contrainte (1) permet maintenant de retirer la valeur 2 du domaine de y. Il est maintenant impossible de continuer à propager.

Les domaine d'application de la CPP sont très variés. Leurs études et leurs classifications ont permis de faire émerger des sous-problèmes de complexité polynomiale spécifique à chacun d'eux. De nombreuses contraintes globales ont ainsi été développées. La structure des solveurs PPC à permis l'intégration de ces algorithmes de filtrage spécialisés. C'est notamment le cas pour les problèmes d'ordonnancements pour lesquels il existe toute une littérature et auquel nous allons nous intéresser plus spécifiquement [3].

2 Problèmes d'ordonnancement

L'ordonnancement est l'un des domaines les plus étudiés et les plus fructueux de la PPC. Bien que les problèmes d'ordonnancements appartiennent à une classe de problèmes complexes, leur résolution efficace est permise par l'utilisation de contraintes globales spécifiques. Celles-ci permettent de réduire considérablement l'espace de recherche par l'identification et l'encapsulation de sous-problèmes polynomiaux [3].

Deux des contraintes globales les plus fréquemment utilisées pour réduire l'espace de recherche d'un problème d'ordonnancement sont les contraintes de précédences binaires et les contraintes de ressources unaires, appelées aussi *noOverlap*.

2.1 Contrainte de ressource unaire (*noOverlap*)

En ordonnancement, une ressource unaire est la généralisation d'une machine ou d'une tâche. Une ressource unaire est définie par son horaire de commencement au plus tôt, son horaire de fin au plus tard, ainsi que sa durée.

Un modèle avec ressource unaire est un ensemble de tâches non interruptibles d'activités qui ne doivent pas se chevaucher. De plus, une fois qu'une tâche est affectée à une ressource elle s'exécute jusqu'à sa terminaison, sans changement ni interruption.

Un sous-problème du problème d'ordonnancement est un problème qui satisfait la contrainte de ressource unaire. Le problème étant NP-Difficile [4], l'objectif de la contrainte est de réduire l'espace de recherche par élimination de toutes les valeurs sans support.

En raison du caractère également NP-difficile du problème de la réduction du domaine [4], il n'est pas possible de retirer toutes les valeurs incohérentes de manière efficace. Sachant que ces algorithmes seront répétés dans chaque nœuds de l'arbre de recherche, on utilise à la place plusieurs algorithmes de filtrage incomplets mais rapides. Pour des raisons de performance, les différents algorithmes de filtrage actuels réduisent les ensembles de valeurs de départs possibles à des intervalles de valeurs. Cette simplification permet de se concentrer sur la réduction des bornes de ces intervalles.

2.2 Le *jobshop*, un exemple d'utilisation de la contrainte de ressource unaire

L'un des problème classique d'ordonnancement est l'ordonnancement d'ateliers. L'ordonnancement d'ateliers consiste à organiser dans le temps le fonctionnement d'un atelier pour utiliser au mieux les ressources disponibles. Le but étant de produire une certaine quantité de biens dans un temps imparti.

Un *jobshop* est un problème d'ordonnancement d'ateliers caractérisé par un ensemble de m ressources qui peuvent être utilisées pour accomplir n tâches. Chaque tâche consiste en une séquences de m activités requérant différentes ressources. les différentes activités d'un job s'exécutent dans un ordre déterminé. Une ressource ne peut être affectée qu'à une seule activité à la fois et ne peut être arrêtée pendant son exécution. La durée d'exécution de chaque activité est connue [4].

Le but est de trouver un ordonnancement qui réduise le *makespan*, soit le temps minimal d'exécution de tous les jobs repris de *Global Constraints in Scheduling* [4].

2.2.1 Exemple de résolution d'un *jobshop*

Le problème de *jobshop* (LA19) est issu d'un ensemble de benchmarks connus, l'OR Library. Pour modéliser ce problème comme un CSP on utilise habituellement la contrainte de ressource unaire pour s'assurer qu'aucune des ressources n'est utilisées par deux activités en même temps. Une autre contrainte utilisée est la contrainte de précédence (i s'exécute avant j, $i \ll j$).

Le problème (LA19) peut se formuler dans un pseudo langage proche de celui utilisé en CP, voir 1, page 7. Une solution optimale du problème (LA19) est 2, page 8.

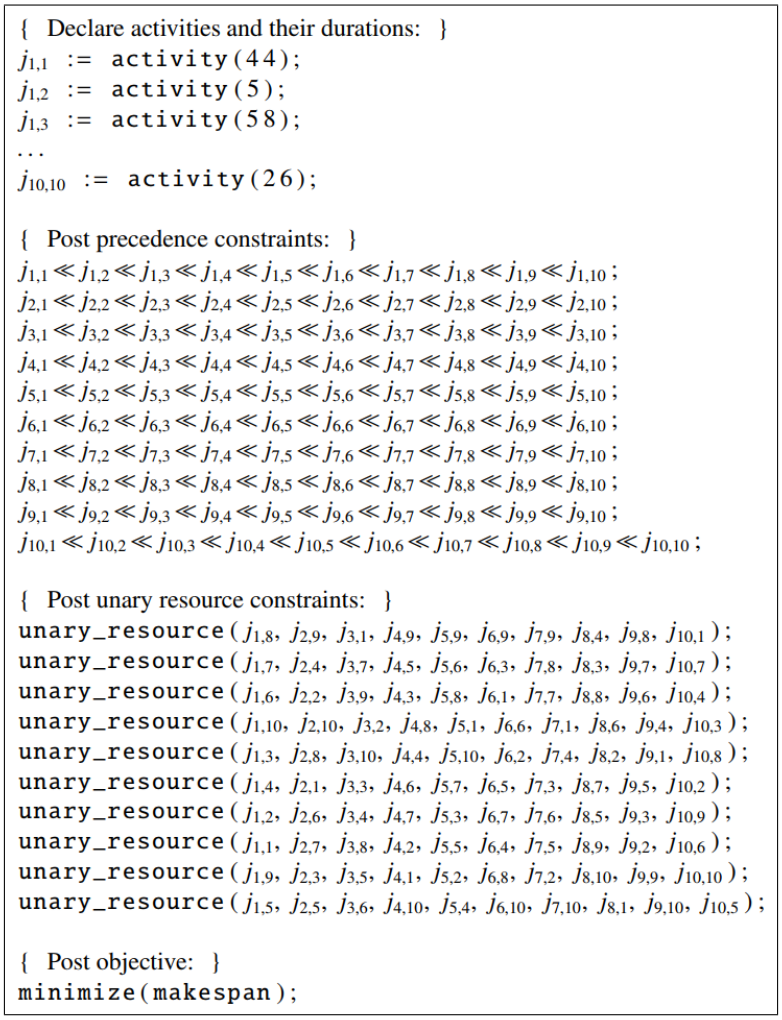


FIGURE 1 – Formulation de (LA19) dans un pseudo langage CP repris de *Global Constraints in Scheduling* [4]

.

2.3 Règles de filtrage de la contrainte de ressource unaire

La thèse *Global Constraints in Scheduling* de Petr Vilím [4], contient une liste des règles de filtrage utilisées lors de la résolution de problèmes d'ordonnancement et notamment les règles de filtrage de la contrainte dite de ressource unaire (ou *noOverlap*).

Liste des règles de filtrage de la contrainte globale *noOverlap* :
— Overload Checking
— Detectable Precedences
— Not-First/Not-Last
— Edge Finding
— Precedence Graph

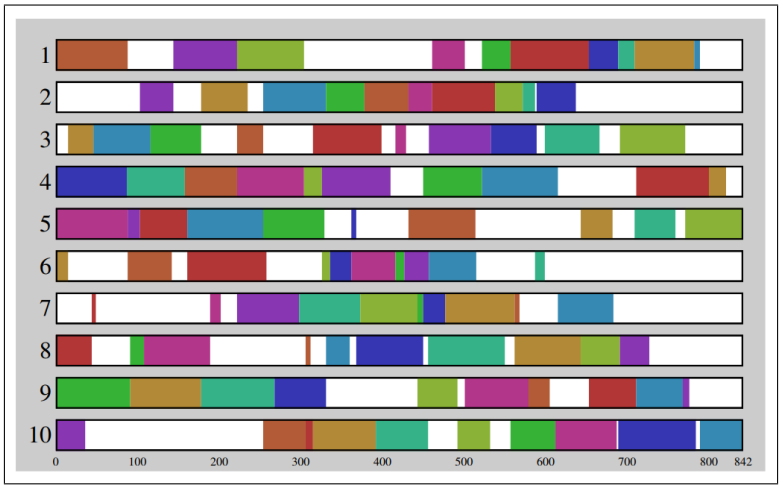


FIGURE 2 – Illustration d'une solution optimale d'un jobshop avec $n = 10$ et $m = 10$. Chaque tâche est représenté par une couleur et chaque ressource par une ligne numérotée repris de *Global Constraints in Scheduling* [4]

.

On présente dans la sous-section suivante une description succincte de ces règles. Règles actuellement implémentées dans le solveur CPLEX CP Optimizer d'IBM pour la propagation de la contrainte globale *noOverlap*.

2.4 Vocabulaires

Un certain nombre de notations présent dans la thèse de P.V. [4] sera repris dans la partie description et modélisation.

Soit un ensemble de tâches T.

Chaque tâche $i \in T$ est restreinte de la manière suivante :
— est_i *(earliest possible starting time)* :
L'horaire de commencement au plus tôt d'une tâche.
— lct_i *(latest possible completion time)* :
L'horaire de fin au plus tard d'une tâche.
— p_i *(processing time)* :
Le temps de traitement de la tâche i.

Soit $\Omega \subseteq T$ Un sous-ensemble non-vide de T.
— $est_\Omega = \min_{j \in \Omega}\{est_j\}$:
L'horaire de commencement au plus tôt d'une tâche appartenant à ω.
— $lct_\Omega = \max_{j \in \Omega}\{lct_j\}$:
L'horaire de fin au plus tard d'une tâche appartenant à ω.
— $p_\Omega = \sum_{j \in \Omega} p_j$:

La somme des durées des tâches appartenant à ω.

Souvent, il est nécessaire d'avoir le plus petit temps de complétion de l'ensemble Ω.
— ect_Ω *(earliest completion time)* $= \max_{\Omega' \subseteq \Omega}\{est_{\Omega'} + p_{\Omega'}\}$

Les contraintes de précédence sont dénotées de la manière suivante :
— $i \ll j$ *(activity i finishes before the activity j starts)* :
La tâche i termine avant que j ne commence.

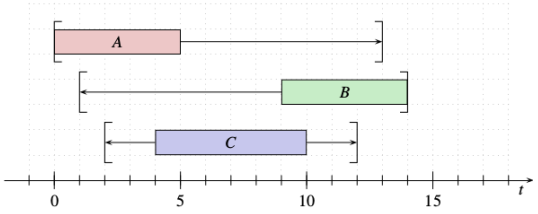
Les contraintes de précédence se propagent de cette manière :
— $i \ll j \Rightarrow est_j := \max\{est_j, est_i + p_i\}$
— $i \gg j \Rightarrow lct_i := \min\{lct_i, lct_j - p_j\}$

2.5 Descriptions des règles de filtrage

2.5.1 Overload Checking

Formulation $\forall \Omega \subseteq T, est_\Omega + p_\Omega > lct_\Omega \Rightarrow fail$

Exemple



$$est_{ABC} = \min\{est_A, est_B, est_C\} = 0$$
$$p_{ABC} = p_A + p_B + p_C = 16$$
$$lct_{ABC} = \max\{lct_A, lct_B, lct_C\} = 14$$
$$est_{ABC} + p_{ABC} > lct_{ABC} \Rightarrow \bot$$

2.5.2 Edge Finding

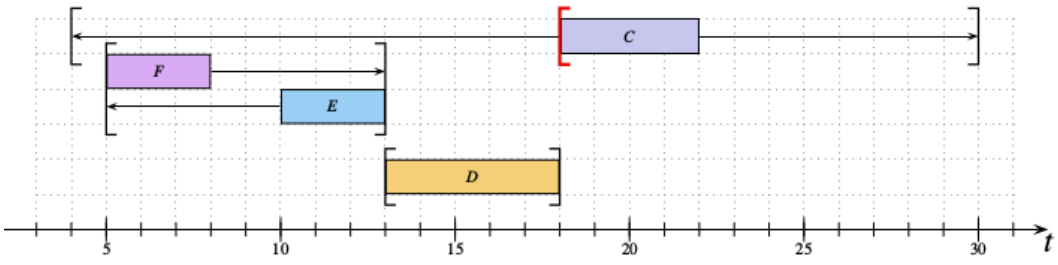
Formulation

$\forall \Omega \subseteq T, \forall i \in (T \setminus \Omega)$:

$$(est_{\Omega \cup \{i\}} + p_\Omega \cup \{i\} > lct_\Omega \Rightarrow \Omega \ll i \Rightarrow est_i := \max\{est_i, ect_\Omega\})$$

L'edge finging est une règle de filtrage qui s'appuie sur la condition de l'overload checking donné précédemment. L'Edge Finding permet de décalé la date de début d'une tâche. Une version symétrique de cette règle existe pour calculer les dates de fins.

Exemple



$\Omega = \{F, E, D\}$
$i = C$
$est_{\Omega\cup\{i\}} = 4$
$p_{\Omega\cup\{i\}} = p_i + p_D + p_E + p_i = 15$
$est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} = 19$
$lct_{DEF} = 18$

$est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} > lct_{DEF} \Rightarrow$
$est_i = \max\{est_i, ect_{DEF}\}$
$est_i = \max\{4, \max\{est_D + p_D, est_E + p_E, est_F + p_F\}\}$
$est_i = \max\{4, \max\{13 + 5, 5 + 3, 5 + 3\}\}$
$est_i = \max\{4, \max\{18, 8, 8\}\}$
$est_i = \max\{4, 18\}$
$est_i = 18$

2.5.3 Autres règles de filtrage

On trouvera dans la thèse de P.V. la description des autres règles de filtrage ainsi que leur implémentation. On retiendra surtout que chaque règle de filtrage de la *noOverlap* peut se résoudre en $\mathcal{O}(n \log n)$ [4].

Deuxième partie
Problématique

Comme explicité dans la partie 1, l'approche actuelle pour propager de manière efficace les contraintes de la *noOverlap* dans un problème d'ordonnancement consiste à se restreindre à l'utilisation d'algorithmes de filtrage polynomiaux travaillant essentiellement à réduire les fenêtres temporelles d'exécution de chaque tâche.

La raison tient au caractère NP-difficile [2] du problème et du fait que la phase de propagation intervient à chaque nœud de l'arbre de recherche de l'espace des solutions [1]. On cherche donc à minimiser son impact sur les performances.

Toutefois, la simplification du problème initial et la limitation des algorithmes de filtrage à une complexité polynomiale réduisent d'autant la quantité de raisonnements possibles lors de la phase de propagation.

L'approche proposée dans ce rapport consiste à outrepasser cette limite de complexité pour maximiser la quantité de raisonnements possibles lors de la phase de propagation. Pour cela on utilisera une méthode de résolution hybride mêlant la PPC et la PL (programmation linéaire) pour la propagation de la contrainte globale *noOverlap*.

On propose plus spécifiquement d'utiliser une relaxation linéaire d'un modèle PLNE de la noOverlap et d'en dériver un algorithme de filtrage.

La PLNE (programmation linéaire en nombre entier) est issue de la recherche opérationnelle. Bien que la distinction ne soit pas aussi claire, la recherche opérationnelle consiste en général à résoudre des problèmes d'optimisation tandis que la PPC s'attache plus à résoudre des problèmes de satisfiabilité [3].

L'accent n'est pas dans un premier temps porté sur les performances pures de l'implémentation qui pourra toujours être raffinée par la suite, mais sur la qualité des raisonnements obtenus sur les domaines. C'est-à-dire la quantité de filtrages supplémentaires obtenus lors de la phase de propagation en utilisant le modèle PLNE relâché (le LP) comparativement à ceux obtenus avec les algorithmes polynomiaux décrits précédemment.

Troisième partie
Relaxation

La relaxation d'un problème consiste en général à simplifier la satisfaction de certaines de ses contraintes les plus complexes (soit par suppression, soit par une prise en compte partielle).

La relaxation d'un problème de minimisation permet d'obtenir une borne inférieure de l'optimum. Une manière classique de relâcher un problème consiste à supprimer toutes ses contraintes d'intégralité. On passe alors d'un programme linéaire en variables entières (PLNE) à un programme linéaire en variables continues (PL). L'écart entre la borne inférieure du problème relâché d'avec le véritable optimal peut toutefois être important. Tout aussi problématique le fait que la solution du problème réduit ne ressemble potentiellement en rien à celle du modèle PLNE en raison de la possibilité pour les variables de prendre des valeurs fractionnaires.

C'est pourquoi on ajoute des *coupes* à la modélisation afin de resserrer l'écart entre la solution entière et celle issue du modèle relâché. L'idéal étant de retrouver la solution du modèle PLNE. Cela revient à obtenir l'égalité entre la borne optimale inférieure du PL et à l'optimal réel du PLNE.

3 Modélisation

3.1 Formulation

On considère pour le modèle PLNE une formulation en temps discrétisé. On rappelle qu'une solution de cette modélisation est une solution de la *noOverlap*.

La variable s_t représente le nombre de jobs s'exécutant au même instant et permet de formuler (M1) comme un problème d'optimisation plutôt qu'un problème de satisfaction permettant de tester la qualité de la relaxation.

$$x_{it} \in \{0,1\} \text{ Indique si une tâche } i \text{ s'exécute à l'instant } t \quad (7)$$
$$y_{it} \in \{0,1\} \text{ Indique si une tâche } i \text{ démarre à l'instant } t \quad (8)$$

$$Objectif : \min z = \sum_{t=1}^{H} s_t$$

$$\text{s.c.} \quad \sum_{i=1}^{n} x_{it} \leq 1 + s_t, \quad \forall t \in [1, H] \tag{1}$$

$$\sum_{t=1}^{H} y_{it} = 1, \quad \forall i \in [1, n] \tag{2}$$

$$(M1) \quad \sum_{i=1}^{n} y_{it} \leq 1, \quad \forall t \in [1, H] \tag{3}$$

$$y_{it} \leq x_{it'}, \quad \forall t' \in [t, t + p_i[, \forall i \in [1, n], \tag{4}$$

$$y_{it} = 0, \quad \forall t \notin [est_i, lct_i - p_i[\tag{5}$$

$$x_{it} = 0, \quad \forall t \notin [est_i, lct_i[\tag{6}$$

$$x_{it} \in \{0, 1\} \tag{7}$$

$$y_{it} \in \{0, 1\} \tag{8}$$

$$s_t \geq 0 \tag{9}$$

$$s_t \leq n \tag{10}$$

Au plus s_t tâches s'exécutent à l'instant t (1), i doit démarrer à un seul moment (2), au plus une tâche démarre à chaque instant (3), si i démarre à t alors i s'exécute sur intervalle $[t, p + p_i[$ (4), i ne peut s'exécuter que dans sa plage autorisée (5), i ne peut démarrer que dans sa plage autorisée (6).

3.2 Les coupes

Lorsqu'on supprime les contraintes d'intégralité de $(M1)$, on passe d'un modèle à variables entières (PLNE), à continues (PL). La solution de la modélisation $(M1)$ est appelée le z_{PL} ou z_{NE} selon que l'on considère ou non le problème relâché.

Lorsqu'on supprime les contrainte d'intégralité de $(M1)$, les contraintes restantes ne permettent plus de détecter efficacement le chevauchement des tâches.

C'est pourquoi on introduit des *coupes* à la modélisation afin de resserrer l'écart entre le z_{PL} et le z_{NE}. L'idéal étant d'obtenir l'égalité entre la borne optimale inférieure du PL et l'optimal réel du PLNE.

Les contraintes (11), (12), (13), (14) et (15) correspondent aux coupes ajoutées au modèle $(M1)$ dans ce but.

3.2.1 Exemple considéré

Pour illustrer on considère le problème (E_0) comprenant deux tâches i, j tel que :
— $est_i = 0$, $lct_i = 4$ et $p_i = 2$.
— $est_j = 0$, $lct_j = 4$ et $p_j = 3$.

Soit une représentation schématique du problème (E_0) tel que :
— $\times$: représente une variable assignable sur l'ensemble $\{0, 1\}$ dans le cas entier ou sur l'intervalle $[0, 1]$ dans le cas continu.
— $\cdot$: une variable non assignable.

x_{it} :

i \ t	1	2	3	4
1	$\times$	$\times$	$\times$	$\cdot$
2	$\times$	$\times$	$\times$	$\times$

y_{it} :

i \ t	1	2	3	4
1	$\times$	$\times$	$\cdot$	$\cdot$
2	$\times$	$\times$	$\cdot$	$\cdot$

Dans le cas d'une résolution en PLNE du problème on obtient la solution optimale suivante :

x_{it} :

i \ t	1	2	3	4
1	1	1	0	$\cdot$
2	0	1	1	1

y_{it} :

i \ t	1	2	3	4
1	1	0	$\cdot$	$\cdot$
2	0	1	$\cdot$	$\cdot$

Tandis qu'en LP, on obtient la solution suivante :

x_{it} :

i \ t	1	2	3	4
1	0.5	0.5	0.5	$\cdot$
2	0.5	0.5	0.5	0.5

y_{it} :

i \ t	1	2	3	4
1	0.5	0.5	$\cdot$	$\cdot$
2	0.5	0.5	$\cdot$	$\cdot$

Soit un $z_{PL} = 0.0$ et un $z_{NE} = 1$. Le fractionnement des affectations entraîne la non-détection d'un chevauchement pourtant détecté dans le PLNE.

3.3 Présentation des différentes coupes

3.3.1 Première coupe

$$\sum_{i=1}^{n} x_{it} = p_i, \quad \forall t \in [1, H] \tag{11}$$

La coupe (11) contraint la somme des exécutions pour une tâche i à être égale à la durée totale de cette tâche.

En ajoutant la coupe (11) à l'exemple précédent (E_0) on détecte un chevauchement des tâches avec z_{PL} ramené à 1, toutefois la solution trouvée est encore assez différente de celle du modèle PLNE :

x_{it} :

i \ t	1	2	3	4
1	0.5	0.5	1	$\cdot$
2	0.5	0.5	1	1

y_{it} :

i \ t	1	2	3	4
1	0.5	0.5	$\cdot$	$\cdot$
2	0.5	0.5	$\cdot$	$\cdot$

3.3.2 Deuxième coupe

$$\sum_{t'=\max(1,t-p_i+1)}^{t} y_{it'} \geq x_{it}, \quad \forall t \in [1,H], \forall i \in [1,n] \tag{12}$$

La coupe (12) contraint une tâche i qui s'exécute à un instant t à démarrer sur l'intervalle $[t-p_i+1, t]$.

En ajoutant encore la coupe (12) à (E_0) on ramène sa solution à celle du modèle PLNE.

3.3.3 Troisième coupe

$$\sum_{t'=t}^{\min(t+pi-1,H)} y_{jt'} \leq 1 - y_{it}, \quad \forall t \in [1,H], \forall j \neq j, j \in [1,n] \tag{13}$$

La coupe (13) contraint toute tâche j à s'exécuter après une tâche i dont l'exécution est fixée.

3.3.4 Quatrième et cinquième coupes

On considère pour ces coupes en plus de celui définie par Vilim, le vocabulaire suivant :
o_i^s : Début de la section obligatoire de la tâche i.
o_i^p : Durée de la section obligatoire de la tâche i.
o_i^e : Fin de la section obligatoire de la tâche i.

$$\sum_{t=\max(1,o_i^s-p_i+o_i^p)}^{o_i^s} y_{it} = 1 \quad \forall i \in [1,n] \tag{14}$$

$$x_{it} = 1, \quad \forall t \in [o_i^s, o_i^e[, \forall i \in [1,n] \tag{15}$$

La coupe (14) contraint une tâche i, lorsque celle-ci comprend une partie dite obligatoire (tel que définie dans la partie vocabulaire) à démarrer sur l'intervalle $[o_i^s - p_i + o_i^p, o_i^s]$.

La coupe (15), en complément de la coupe (14) contraint une tâche i à s'exécuter sur une partie dite obligatoire.

Quatrième partie
Algorithme de filtrage

4 Principes

Pour filtrer les domaines de départ d'une tâche on utilise un algorithme simple de type *Brute Force*. Pour toutes valeurs de début possible v de toutes tâches $i \in T$ on force le démarrage de

la tâche i à v et on observe la valeur de l'objectif. Cela revient dans le modèle $(M1)$ à forcer itérativement à 1 chaque t de $y_{it}, \forall i \in T$ et d'observer z. Si z_{PL} ou z_{NE} est ≥ 0, alors v est une valeur de départ incohérente pour la tâche i et on peut la retirer de son domaine.

4.1 Considération

Cet algorithme de filtrage permet potentiellement de détecter toutes valeurs de départ incohérentes du domaine de démarrage d'une tâche. Contrairement au règles de filtrages définie par P.V. qui détecte uniquement les valeurs incohérentes en bordure des intervalles de valeurs. C'est pourquoi l'on utilisera pour mesurer la qualité du filtrage obtenue deux valeurs : la taille effective du domaine résultante du filtrage, la taille du domaine résultante du filtrage sans considérer les "trous" dans les intervalles.

4.2 Exemple

On exécute l'algorithme de filtrage sur le problème (E_0) grâce au modèle $(M1)$ avec les coupes (11), (12), (14) et (15). Voici les différentes solutions obtenues :

Pour $i = 1$ et $t = 1$:

x_{it} :

i \ t	1	2	3	4
1	1	1	0	·
2	0	1	1	1

y_{it} :

i \ t	**[1]**	2	3	4
[1]	**1**	0	·	·
2	0	1	·	·

$z_{PL} = 1$.

Pour $i = 1$ et $t = 2$:

x_{it} :

i \ t	1	2	3	4
1	0	1	1	·
2	1	1	1	1

y_{it} :

i \ t	1	**[2]**	3	4
[1]	0	**1**	·	·
2	1	0	·	·

z_{PL} : 2.

Et ainsi de suite jusqu'à $i = 2$ et $t = 2$

Toutes les valeurs de départ incohérentes sont ensuite retirées du domaine. (Dans le cas du filtrage de (E_0), la taille du domaine filtré est égale à 0).

5 Réduction du domaine sur les exemples de Vilim

5.1 Valeur de l'objectif préfixé à l'optimal

On observe le filtrage des domaines sur l'exemple *Edge Finding* donné par P.V filtré sur $(M1)$ avec les coupes (11), (12) :

Coupes	zCP	$diCP$	$drCP$	zLP	zNE	$dbLP$	$dbNE$	$drLP$	Arc
	19	39	10	0	0	22	10	18	6
(11)	19	39	10	0	0	16	10	14	6
(12)	19	39	10	0	0	22	10	18	6
(11), (12)	19	39	10	0	0	10	10	6	6

Légende :
— Coupes : Coupes utilisées.
— zCP : Makespan du modèle CP.
— $diCP$: Domaine initial du modèle CP.
— $drCP$: Domaine après filtrage du modèle CP.
— zLP : Objectif du Modèle PL.
— zNE : Objectif du Modèle PLNE.
— $dbLP$: Domaine du modèle LP après filtrage par réduction des bordures uniquement.
— $dbNE$: Domaine du modèle PLNE après filtrage par réduction des bordures uniquement.
— $drLP$: Domaine du modèle PLNE après filtrage.
— Arc : Cohérence par arcs (*arc consistency*).

L'utilisation combinée des contraintes (11), (12) permet de se ramener à l'arc cohérence du problème (tous les raisonnements possibles ont été effectués).

Cinquième partie
Évaluation Expérimentale

6 Jeu de données

On teste notre filtrage sur des instances de jobshop classiques de la littérature et appartenant à l'OR-library. Il existe 82 jobshops communément cités dans la littérature. Nous en choisissons 8 sur lesquels nous testerons notre filtrage.

Lien vers la source : ORLIB (https ://www.eii.uva.es/elena/JSSP/InstancesJSSP.htm)

6.1 Descriptions des instances

instance	machines x tâches	trouver l'optimum	trouver l'optimal
la01	5 x 10	facile	facile
la02	5 x 10	facile	facile
la03	5 x 10	facile	difficile
la04	5 x 10	facile	difficile
la05	5 x 10	facile	facile
mt06	6 x 10	facile	facile
mt10	10 x 10	difficile	très difficile
mt20	5 x 20	difficile	très difficile

6.2 Format d'une instance

Une ligne contenant le nombre de taches et le nombre de machines. Ensuite une ligne pour chaque job comprenant le numéro de la machine ainsi que son temps d'exécution pour chaque

étape du job. La numérotation des machines commence à zéro.

7 Résultats

Faute de temps, l'implémentation du test de filtrage sur des instances de jobshop est encore instable et ne permet donc pas de conclure quant à la qualité de filtrage supplémentaire obtenu.

Références

[1] Sophie Demassey. Méthodes hybrides de programmation par contraintes et programmation linéaire pour le problème d'ordonnancement de projet à contraintes de ressources. modélisation et simulation. 2003.

[2] W.H.Freeman M. R. Garey, D. S. Johnson and Company. Computers and intractability : A guide to the theory of np-completeness. 1079.

[3] Arnaud Malapert. Techniques d'ordonnancement d'atelier et de fournées basées sur la programmation par contraintes. 2011.

[4] Petr Vilím. Global constraints in scheduling. 2007.

A Notes relatives à l'utilisation du solveur CPO

A.1 Notes d'installation

Lien vers la page de téléchargement : IBM ILOG CPLEX Optimization Studio 12.7

A.2 Notes de configuration

Dossier d'installation

```
/opt/ibm/ILOG/CPLEX_Studio127/cpoptimizer/bin/x86-64_linux/
/opt/ibm/ILOG/CPLEX_Studio127/cplex/bin/x86-64_linux/
```

VM Option

```
-Xmx1024m -Xms1024m -d64
```

Environnements variables

```
LD_LIBRARY_PATH=/opt/ibm/ILOG/CPLEX_Studio127/cpoptimizer/bin/x86-64_linux/:/
    opt/ibm/ILOG/CPLEX_Studio127/cplex/bin/x86-64_linux/:$DYLD_LIBRARY_PATH;
    LD_LIBRARY_PATH=/opt/ibm/ILOG/CPLEX_Studio127/cpoptimizer/bin/x86-64
    _linux/:/opt/ibm/ILOG/CPLEX_Studio127/cplex/bin/x86-64_linux/:
    $LD_LIBRARY_PATH
```

Dépendances

```
/opt/ibm/ILOG/CPLEX_Studio127/cpoptimizer/lib/ILOG.CP.jar
```

A.3 Options de la contrainte NoOverlap

A.3.1 Extrait du manuel de référence

Lien vers le pdf : CP Optimizer User's Manual
Lien vers le site : CP Optimizer Java API Reference Manual / Package ilog.cp

```
CP Optimizer provides tuning parameters which are used to adjust the
    constraint propagation inference levels.

The inference level of a constraint, that is, the strength of the domain
    reduction is achieves, is controlled by tuning parameters.

There is a parameter for each specialized constraint whose inference level
    can be changed.

The effects of changing the values of these inference level parameters will
    be
described in the following sections.
In the C++ API, the possible values of these parameters are:
- IloCP::Default,
- IloCP::Low,
- IloCP::Basic,
- IloCP::Medium and
- IloCP::Extended.
For the Java API, the default value is IloCP.ParameterValues.Default.
```

A.3.2 Exemple d'utilisation

```
IloCP cp = new IloCP();
cp.setParameter(IloCP.IntParam.NoOverlapInferenceLevel, IloCP.ParameterValues
    .Extended);
System.out.println("NoOverlap Inference level = " + cp.getParameter(IloCP.
    IntParam.NoOverlapInferenceLevel));
cp.propagate();
```