Octogones entiers pour le problème RCPSP

Pierre Talbot[1] **David Cachera**[2] **Éric Monfroy**[1] **Charlotte Truchet**[1]

[1] Université de Nantes, LS2N, 44000 Nantes, France
[2] Univ Rennes, Inria, CNRS, IRISA, 35000 Rennes, France
{prénom.nom}@univ-nantes.fr david.cachera@irisa.fr

Résumé

En programmation par contraintes, l'efficacité des solveurs est principalement due aux contraintes globales qui encapsulent des algorithmes de résolution spécifiques. Une conséquence est la prolifération de centaines de contraintes globales qui sont spécialisées pour des problèmes très particuliers. Dans cet article, nous étudions une approche basée sur les domaines abstraits de l'analyse de programmes par interprétation abstraite. Les domaines abstraits permettent de résoudre efficacement des conjonctions de contraintes primitives dans un langage fixé. Essentiellement, nous utilisons une décomposition de la contrainte globale `cumulative` vers des conjonctions de contraintes primitives pouvant être prises en charge par les domaines abstraits. Le premier avantage est que ces derniers sont moins nombreux car plus généralistes, et peuvent donc être réutilisés sur une gamme de problèmes plus large. Le deuxième est que ces domaines abstraits peuvent échanger des contraintes grâce à la technique du produit réduit issue de l'interprétation abstraite. La contribution principale est d'utiliser les contraintes réifiées afin de programmer cet échange d'information entre le domaine abstrait des intervalles et des octogones. Nous explicitons cette approche sur le problème de gestion de projet à contraintes de ressources (RCPSP). Les expérimentations préliminaires montrent que l'approche offre un bon compromis entre efficacité et expressivité.

1 Introduction

L'efficacité d'un solveur de contraintes tient principalement à la présence de contraintes globales permettant d'encapsuler des algorithmes spécifiques à une sous-structure d'un problème. Par exemple, la contrainte globale `alldifferent`$(x_1, \ldots, x_n)$ s'assure que les variables $x_1, \ldots, x_n$ sont toutes différentes. On recense presque 400 contraintes globales [2] possédant chacune plusieurs algorithmes de propagation plus ou moins efficaces. Ce qui fait à la fois la force et la faiblesse des contraintes globales est leur flexibilité : on peut toujours créer une nouvelle contrainte globale capturant un problème spécifique, au prix de passer plus de temps dans l'implémentation algorithmique que sur la partie modélisation. De plus, la plupart des solveurs de contraintes n'implémentent que quelques dizaines de contraintes globales—les plus courantes. Un dernier problème est l'absence de communication forte entre deux contraintes globales identiques. Par exemple, si un modèle contient deux contraintes `alldifferent` partageant des variables, cette information est difficilement prise en compte dans les solveurs classiques. Une solution est de passer par des algorithmes de consistance plus fortes que ceux implémentés dans les solveurs classiques [3]. Une autre solution est de créer une nouvelle contrainte globale agrégeant plusieurs `alldifferent`, qui par ailleurs existe et s'appelle `k_alldifferent`.

Dans cet article, nous étudions une approche basée sur les domaines abstraits venant du domaine de l'analyse de programme par interprétation abstraite [7]. Au lieu de capturer des sous-structures du problème, un domaine abstrait capture des sous-théories de la programmation par contraintes [20]. Une théorie représente une conjonction de contraintes dans un langage fixé. Par exemple, on peut imaginer le domaine abstrait `Diff` qui gère les contraintes de la forme $x \neq y$. Dans ce cas, toutes les contraintes $\neq$ du modèle seront propagées dans ce domaine abstrait par un algorithme efficace. Le domaine abstrait `Diff` subsume les deux contraintes `alldifferent` et `k_alldifferent`. Un premier avantage est que les domaines abstraits sont moins dépendants d'un problème en particulier, et peuvent donc être réutilisés sur des problèmes avec des sous-structures très différentes. Un deuxième avantage est que les domaines abstraits peuvent communiquer entre eux grâce à la technique du produit réduit de l'interprétation abstraite.

Il existe une multitude de produits réduits entre deux

domaines et il doivent généralement être programmés à la main. La contribution de cet article est de proposer un produit réduit générique entre deux domaines abstraits en utilisant des contraintes d'équivalence de la forme $c_1 \Leftrightarrow c_2$, également appelées *contraintes réifiées*. Soit deux domaines abstraits A et B, ces contraintes réifiées permettent de faire le pont entre les contraintes c_1 appartenant à A et c_2 appartenant à B. Afin d'obtenir ce produit réduit automatiquement, nous étendons la définition d'un domaine abstrait avec un opérateur de déduction (*entailment*). Cet opérateur est ensuite utilisé pour programmer le propagateur de la contrainte réifiée.

Nous illustrons cette méthode en proposant d'étudier le produit réduit du domaine des intervalles et des octogones afin de résoudre le problème de gestion de projet à contraintes de ressources (RCPSP) [9, 10], que nous introduisons en Section 2. L'idée principale est d'utiliser une décomposition de la contrainte globale `cumulative` pour qu'elle puisse être gérée par le domaine des octogones [22]. Les contraintes restantes, non octogonales, peuvent être prise en charge par le domaine des intervalles (également appelés « boîtes ») qui peut être vu comme le domaine de base en programmation par contraintes. Une boite discrète est un produit Cartésien d'intervalles de la forme $[l, u]_x \in \mathbb{N}^2$ tel que la variable x peut être affectée à une valeur dans l'ensemble $\{v \mid l \leq v \leq u\}$.

Les définitions nécessaires d'interprétation abstraite sont introduites en Section 3 ainsi que les octogones entiers en Section 4. Le produit réduit par contraintes réifiées est donné en Section 5. Nous fournissons une implémentation [1] de ces domaines abstraits et testons son efficacité sur une série de *benchmarks* standard pour le RCPSP (Section 6).

Finalement, afin d'éviter toute ambiguïté, notons de suite que l'idée des domaines abstraits est très proche des solveurs *Satisfiability Modulo Theories* (SMT). La différence principale est que les domaines abstraits sont basés sur la théorie des treillis plutôt que sur un fondement logique comme en SMT. Des travaux récents connectant les deux domaines sont disponibles [8].

2 Le problème RCPSP

Le problème de gestion de projet à contraintes de ressources (RCPSP) est un problème classique d'ordonnancement de tâches sous ressources. Un problème RCPSP est un triplet (T, P, R) où T est un ensemble de tâches, P est l'ensemble des priorités entre les tâches, que nous noterons $i \ll j$ pour indiquer que la tâche i doit terminer avant que j commence, et R est l'ensemble des ressources. Chaque tâche $i \in T$ a une durée $d_i \in \mathbb{N}$ et une utilisation des ressources $r_{k,i} \in \mathbb{N}$ pour toutes ressources $k \in R$. Chaque ressource $k \in R$ a une capacité $c_k \in \mathbb{N}$ décrivant la quantité de cette ressource disponible à chaque instant. Le but est de trouver un planning des tâches T respectant les priorités dans P et n'excédant pas, à chaque instant, la capacité des ressources disponibles. Généralement, on cherche également à optimiser la durée totale d'exécution du planning.

Un modèle par contraintes de ce problème consiste à représenter une tâche i par sa date de début s_i. Les constantes et variables du problème sont toutes discrètes. Les contraintes temporelles sont exprimées comme suit :

$$\forall (i \ll j) \in P,\ s_i + d_i \leq s_j \tag{1}$$

Les contraintes de ressources sont exprimées comme suit :

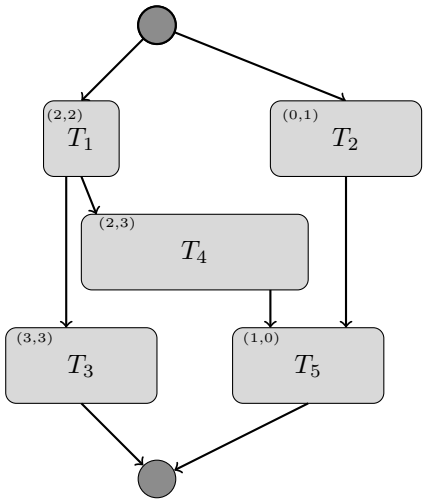
$$\forall t \in [0..h-1],\ \forall k \in R,\ \Big(\sum_{i \in T, s_i \leq t < s_i + d_i} r_{k,i}\Big) \leq c_k \tag{2}$$

où h est l'horizon du problème représentant la date maximale de fin d'une tâche. Une façon simple d'obtenir l'horizon est de faire la somme de la durée des tâches. On vérifie que l'utilisation de chaque ressource k n'excède pas sa capacité c_k à chaque instant t. Une ressource est potentiellement utilisée dans un instant t si une tâche utilisant cette ressource s'exécute à cet instant.

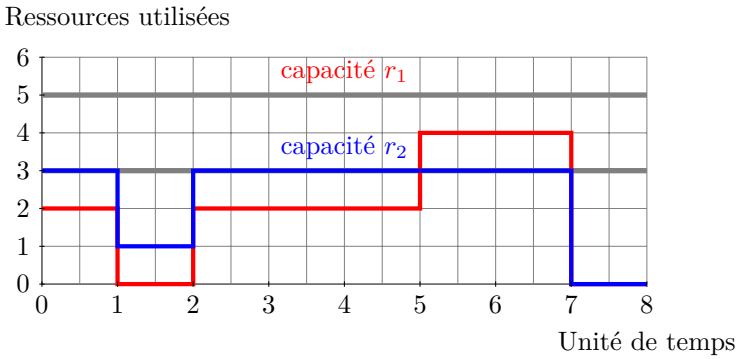
Exemple 1. Nous considérons un exemple de problème RCPSP sur la Figure 1. Ce problème comporte 5 tâches dont les priorités sont décrites sur la Figure 1a, et 2 ressources de capacité 5 et 3 dont l'usage est également donné sur la Figure 1a dans le coin supérieur gauche de chaque tâche. La durée totale minimum d'un planning pour cet exemple est de 7 unités de temps (Figure 1c). Le profil d'utilisation des ressources est représenté sur la Figure 1b : à chaque instant la somme des ressources utilisées n'excède pas leurs capacités. Par exemple, durant le deuxième instant seule la tâche T_2 est active et consomme 1 unité de la ressource r_2.

Dans la suite, nous considérons la version généralisée *RCPSP/max* où les priorités entre deux tâches sont généralisées. L'ensemble P contient des contraintes temporelles de la forme $\pm s_i - \pm s_j \leq c$ où i, j sont des tâches et $c \in \mathbb{Z}$ une constante entière. Ainsi, si nous avons $s_2 - s_1 \leq 3$, cela signifie que la tâche 2 doit commencer au plus tard 3 instants après la tâche 1. Les contraintes temporelles « au plus tard », « au plus tôt »,

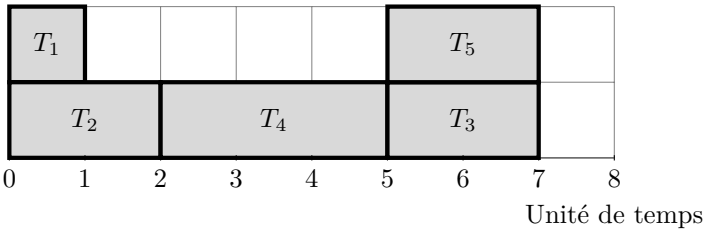
1. Le code et les *benchmarks* sont publiquement disponibles à l'adresse github.com/ptal/AbSolute.



(a) Contraintes temporelles et de ressources.



(b) Profil des ressources.



(c) Une solution minimisant la durée totale.

FIGURE 1 – Un exemple du problème RCPSP sur 5 tâches et 2 ressources de capacité 5 et 3.

« exactement », « après » et « avant » peuvent également être encodées. Cette forme de contraintes apparaît dans de nombreux travaux comme les réseaux temporels de contraintes [9], les octogones en interprétation abstraite [15], et la théorie de la logique différentielle (DL) dans les solveurs *Satisfiability Modulo Theories* (SMT) [21]. Dans cet article, nous nous intéressons aux octogones entiers.

3 Interprétation abstraite

L'interprétation abstraite est une technique d'analyse de programmes par sur-approximation de l'ensemble des valeurs concrètes que peuvent prendre les variables du programme [7]. Nous nous intéressons à un fragment particulier de cette théorie qui est les *domaines abstraits*. Ceux-ci encapsulent des conjonctions de contraintes de forme prédéfinie comme les contraintes de bornes $x \geq c$ ou $x \leq c$ (domaine des intervalles) ou les contraintes linéaires de la forme $a_1 * x_1 + \ldots + a_n * x_n \geq c$ où a_i et c sont des constantes (domaine des polyèdres).

3.1 Notations

On utilise l'ensemble $K = \{true, false, unknown\}$ pour représenter les éléments de la logique de Kleene (on notera par exemple que $false \wedge unknown = false$ et $true \wedge unknown = unknown$). Une contrainte est un prédicat logique défini sur un ensemble de variables, et nous notons C l'ensemble de toutes les contraintes logiques. Nous appellerons *domaine concret* l'ensemble des solutions d'une conjonction de contraintes $\bigwedge c \in C$.

3.2 Domaine abstrait

En interprétation abstraite, un domaine abstrait est un ensemble partiellement ordonné muni de certaines opérations utiles à l'analyse de programme. Les domaines abstraits ont été adaptés au cadre de la programmation par contraintes, avec des opérateurs spécifiques, dans [19]. Dans cet article, nous nous plaçons dans ce cadre et introduisons un domaine entier intégrant des contraintes de différences, les octogones entiers. Nous ne donnons que les opérations nécessaires dans la suite de cet article, et notamment les correspondances de Galois ne sont pas explicitées.

Définition 2 (Domaine abstrait)**.** Un domaine abs-

trait pour la programmation par contraintes est un treillis complet $\langle Abs, \leq \rangle$ dont les éléments sont représentables en machine, et muni des opérations suivantes :

- $\bot$ est le plus petit élément.
- $\sqcup$ est l'opérateur d'union (*join*) de deux éléments.
- $state : Abs \to K$ donne l'état d'un élément : *true* si l'élément satisfait toutes les contraintes du domaine abstrait, *false* si c'est un élément qui ne satisfait pas toutes les contraintes du domaine abstrait, et *unknown* si la satisfiabilité de cet élément ne peut pas encore être décidée.
- $[\![.]\!] : C \to Abs$ est une fonction partielle transformant une contrainte en un élément du domaine abstrait. Cette fonction n'est pas nécessairement définie pour toutes les contraintes puisqu'un domaine abstrait ne permet de gérer que certaines contraintes.
- $closure : Abs \to Abs$ est une fonction extensive ($x \leq closure(x)$) permettant d'éliminer des valeurs inconsistantes du domaine abstrait. Dans le jargon de la programmation par contraintes, il s'agit de la propagation.

Les *solutions* d'un élément abstrait $a \in Abs$ sont données par l'ensemble $sol(a) = \{s \in Abs \mid a \leq s \wedge state(s) = true\}$. Quand une confusion est possible, nous annotons les opérations du domaine abstrait avec le nom du domaine, par exemple $\bot_{Box}$ pour l'élément $\bot$ sur le domaine des intervalles Box. Finalement, notons que le domaine abstrait contient les contraintes que celui-ci peut propager, et l'opérateur *closure* peut être vu comme l'algorithme de propagation des contraintes à l'intérieur du domaine abstrait. Plus de détails concernant les autres opérateurs nécessaires sont disponibles dans [19]. Nous illustrons maintenant cette définition avec l'exemple du domaine abstrait des octogones entiers.

4 Octogones entiers

4.1 Définitions

Le domaine abstrait des octogones entiers est défini sur un ensemble de variables $(x_0, \ldots, x_{n-1})$ et une conjonction de contraintes octogonales de la forme suivante :

$$\pm x_i - \pm x_j \leq d$$

où $d \in \mathbb{Z}$ est une constante. En étendant le domaine à $2n$ variables $(x'_0, \ldots, x'_{2n-1})$, Miné [15] donne une transformation de ces contraintes octogonales en contraintes de potentiel de la forme $x'_i - x'_j \leq d$ (les variables ne sont que positives). Cette représentation est intéressante car un système de contraintes de potentiel peut être résolu en temps cubique par l'algorithme des plus

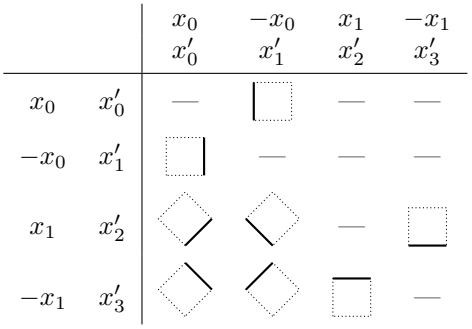


FIGURE 2 – Correspondance DBM/octogone en dimension 2.

courts chemins de Floyd-Warshall. On peut représenter ces contraintes dans une matrice à différences bornées (DBM) qui correspond à l'octogone initial [15]. Une DBM est une matrice m où un élément $m_{i,j}$ représente la constante d d'une contrainte de potentiel $x'_j - x'_i \leq d$. Nous noterons que lorsque $j/2 > i/2$ (où / est la division entière), l'élément représenté $m_{j,i}$ est redondant. Une matrice dont les éléments redondants sont égaux est appelée *cohérente* dans la littérature. Nous ne représenterons pas ces éléments dans les exemples [2] et gardons la matrice entière pour définir les opérations.

Afin de donner quelques intuitions, nous pouvons voir un octogone à n dimensions comme une intersection de boîtes à n dimensions également. Sur la Figure 2, on donne un octogone en 2 dimensions comme l'intersection de deux boîtes (dont une tournée à 45°) et les contraintes associées aux différents côtés de l'octogone. Les lignes et colonnes de la matrice sont annotées avec les variables correspondantes, et on schématise l'élément de la matrice avec la borne du côté qu'il représente. Notons qu'à partir d'une entrée (i, j) de la DBM, nous pouvons récupérer les indices du côté opposé avec $(\bar{i}, \bar{j})$ où :

$$\bar{i} = \begin{cases} i+1 \text{ si } i \text{ est pair} \\ i-1 \text{ si } i \text{ est impair} \end{cases}$$

et similairement pour $\bar{j}$.

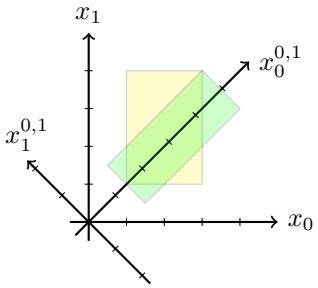
Nous pouvons passer d'un ensemble de contraintes octogonales à un ensemble de contraintes de potentiel par la réécriture suivante (avec $i \neq j$) :

$$\begin{array}{lcll} x_i \geq d & \rightsquigarrow & x'_{2i+1} - x'_{2i} & \leq -2d \\ x_i \leq d & \rightsquigarrow & x'_{2i} - x'_{2i+1} & \leq 2d \\ x_i - x_j \leq d & \rightsquigarrow & x'_{2i} - x'_{2j} & \leq d \\ x_i + x_j \leq d & \rightsquigarrow & x'_{2i} - x'_{2j+1} & \leq d \\ -x_i - x_j \leq d & \rightsquigarrow & x'_{2i+1} - x'_{2j} & \leq d \\ -x_i + x_j \leq d & \rightsquigarrow & x'_{2i+1} - x'_{2j+1} & \leq d \end{array}$$

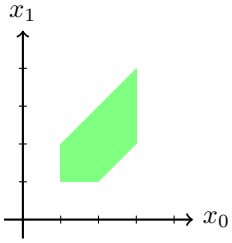
2. C'est la représentation utilisée dans les implémentations [15].

	x'_0	x'_1	x'_2	x'_3
x'_0	0	-2		
x'_1	6	0		
x'_2	1	-2	0	-2
x'_3	7	1	8	0

(a) DBM



(b) Octogone par intersection de deux boîtes



(c) Octogone

FIGURE 3 – DBM et sa représentation par intersection de deux boîtes.

Exemple 3. Afin d'illustrer la correspondance entre les contraintes octogonales, la DBM et sa représentation géométrique, on considère les contraintes suivantes :

$$x_0 \geq 1 \wedge x_0 \leq 3 \qquad x_1 \geq 1 \wedge x_1 \leq 4$$
$$x_0 - x_1 \leq 1 \qquad -x_0 + x_1 \leq 1$$

Les contraintes de borne sur x_0 et x_1 sont représentées par la boîte en jaune sur la Figure 3b, et les contraintes octogonales par la boîte en vert. L'intersection des deux boîtes est donnée sur la Figure 3c.

La contrainte de potentiel associée à $-x_0+x_1 \leq 1$ est $x'_1 - x'_3 \leq 1$, représentée dans la DBM (Figure 3a) par l'entrée $dbm_{3,1} = 1$. Nous pouvons facilement retrouver le côté de l'octogone représenté à la position $(3, 1)$ grâce à la Figure 2.

4.2 Opérations

Nous prenons la représentation matricielle d'un octogone pour définir les opérations du domaine abstrait. Pour définir ces opérations, nous considérons deux DBMs m et m' de dimension n, et $N = [0..2n-1]$. On voit une matrice m comme un ensemble d'éléments indexés $\{d^{i,j} \mid i, j \in N\}$.

Premièrement, la plus petite matrice $\perp$ est définie par $\{\infty^{i,j} \mid i, j \in N\}$.

En second, nous avons l'opérateur $\sqcup$ défini comme suit :

$$m \sqcup m' = \{min(m_{i,j}, m'_{i,j})^{i,j} \mid i, j \in N\} \qquad (3)$$

L'intersection géométrique de deux octogones se fait en prenant le minimum des coefficients des deux DBMs. En effet, si un coefficient est plus petit, cela signifie que l'intervalle entre le côté qu'il représente et le côté opposé est plus étroit.

Ensuite, nous avons l'opération *state* :

$$state(m) = \begin{cases} true & \text{si } \forall i, j \in N,\ m_{i,j} = m_{\bar{i},\bar{j}} \\ false & \text{si } \exists i, j \in N,\ m_{i,j} + m_{\bar{i},\bar{j}} < 0 \\ unknown & \text{sinon} \end{cases} \qquad (4)$$

Un octogone est dans l'état *true* si toutes ses variables sont des singletons, ce qui signifie que la DBM représente un point dans l'espace, et chaque côté est identique à son côté opposé. Un octogone est inconsistant (*false*) si un de ses côtés inférieurs a dépassé le côté supérieur correspondant, ce qui se traduit par la somme des coefficients inférieure à zéro [3]. Dans tous les autres cas, l'octogone est dans un état indéterminé.

L'opération de transfert dans les octogones n'est définie que pour les contraintes octogonales. Considérons a, b les indices des variables octogonales et i, j des variables de potentiel, on a la réécriture suivante :

$$\pm x_a - \pm x_b \leq d \rightsquigarrow x'_i - x'_j \leq d'$$

et donc le transfert suivant :

$$[\![\pm x_a - \pm x_b \leq d]\!] = \perp \sqcup \{d'_{i,j}\}$$

On utilise la transformation d'une contrainte octogonale en contrainte de potentiel et affectons le coefficient obtenu d' dans la case correspondante de la matrice. Finalement l'opérateur de clôture est obtenu grâce à l'algorithme de Floyd-Warshall. Nous ne redonnons pas l'algorithme de clôture ici qui est étudié extensivement dans [15, 1, 4]. Il est particulièrement intéressant de considérer la clôture incrémentale qui permet de mettre à jour la DBM avec une contrainte octogonale en $O(n^2)$ [15, 4] au lieu de la clôture générale qui est en $O(n^3)$.

3. Notons que le signe du coefficient d'un côté inférieur doit être inversé pour obtenir sa coordonnée dans le plan Euclidien.

5 Produit réduit via contraintes réifiées

5.1 Décomposition du problème RCPSP/max

Les contraintes temporelles du problème RCPSP/max sont des contraintes octogonales qui peuvent être gérées efficacement par les octogones entiers. L'étape non triviale est de gérer les contraintes de ressources qui ne sont pas purement octogonales (sinon la résolution du problème serait polynomiale).

En programmation par contraintes, les contraintes de ressources sont généralement représentées par la contrainte globale `cumulative` :

$$\texttt{cumulative}_k([s_1,\ldots,s_n],[d_1,\ldots,d_n],[r_{k,1},\ldots,r_{k,n}],c_k)$$

qui assure que les tâches n'excèdent pas la capacité c_k de la ressource k. Notons que cette contrainte ne gère qu'une seule ressource $k \in R$ à la fois. Afin d'extraire des contraintes octogonales de cumulative, nous considérons sa *décomposition par tâches* telle que donnée dans [22] :

$$\begin{aligned}&\forall j \in [1..n],\ \forall i \in [1..n] \setminus \{j\},\\ &\quad b_{i,j} \Leftrightarrow (s_i \leq s_j \wedge s_j < s_i + d_i)\end{aligned} \tag{5}$$

$$\forall j \in [1..n],\ r_{k,j} + (\sum_{i \in [1..n]\setminus\{j\}} r_{k,i} * b_{i,j}) \leq c_k \tag{6}$$

La décomposition (5) introduit $n^2 - n$ variables booléennes, où une variable $b_{i,j}$ est vraie ssi les tâches i et j se chevauchent. La contrainte de chevauchement $s_i \leq s_j \wedge s_j < s_i + d_i$ est réifiée dans la variable booléenne $b_{i,j}$. Pour ce qui est de la décomposition (6), l'intuition principale est que l'utilisation maximale d'une ressource k ne peut changer que lorsque une tâche commence, et pas pendant son exécution. Ainsi, pour toutes les dates de début s_j, la somme de $r_{k,j}$ et des ressources utilisées par les tâches i s'exécutant à l'instant s_j ne doit pas excéder la capacité c_k.

5.2 Opération de déduction

Les contraintes temporelles du RCPSP (en (1)) peuvent être propagées dans le domaine abstrait des octogones, tandis que les contraintes de ressources (en (6)) peuvent être propagées dans le domaine abstrait des intervalles. Il reste à faire communiquer les deux domaines et les contraintes réifiées remplissent ce rôle. Une contrainte réifiée est une contrainte de la forme $c_1 \Leftrightarrow c_2$ permettant de poser des équivalences entre contraintes. La force de la méthode proposée ici est que les contraintes c_1 et c_2 ne sont pas forcément représentées dans le même domaine abstrait. Ce mécanisme est donc particulièrement adapté pour échanger de l'information entre deux domaines abstraits.

Nous avons des contraintes réifiées en (5) où les variables booléennes $b_{i,j}$ sont représentées dans le domaine abstrait des intervalles et les contraintes $s_i \leq s_j \wedge s_j < s_i + d_i$ dans le domaine des octogones. Afin de gérer cet échange d'information, nous devons munir nos domaines abstraits de l'opération de déduction *entailment* permettant de détecter si une contrainte est déjà induite par le problème.

Définition 4 (Déduction)**.** Soit un domaine abstrait Abs et deux éléments $a, b \in Abs$, l'opération $entailment : Abs \times Abs \to K$ est un opérateur de déduction si il satisfait les propriétés suivantes :

1. Si $entailment(a,b) = true$ (noté $a \vDash b$) alors $sol(a) = sol(a \sqcup b)$; on n'obtient pas moins de solutions avec l'élément b, il est redondant dans a.
2. Si $entailment(a,b) = false$ (noté $a \nvDash b$) alors $sol(a \sqcup b) = \emptyset$; on ne pourra jamais déduire b de a dans une solution.
3. Si $entailment(a,b) = unknown$ alors $state(a) = unknown \vee state(b) = unknown$; on répond $unknown$ si un des éléments a ou b est lui-même $unknown$.

L'opérateur de déduction peut être beaucoup plus facile à calculer qu'établir la satisfiabilité du problème puisqu'on peut répondre *unknown*. Une version générique de cette opération utilisant les opérations $\sqcup$ et *state* d'un domaine abstrait peut-être définie comme suit :

$$entailment(a,b) = \begin{cases} true & \text{si } (a \sqcup b) = a \\ false & \text{si } state(a \sqcup b) = false \\ unknown & \text{sinon} \end{cases}$$

Proposition 5. Soit les DBMs m et m', $entailment(m, m')$ est un opérateur de déduction (Définition 4) pour le domaine abstrait des octogones.

Démonstration. On procède par cas :

— *true* : Si on a $m \sqcup m' = m$ alors par la Définition 3, toutes les entrées de m sont plus petites que celles de m'. La DBM m' contient donc déjà toutes les solutions présentes dans m, et ne permet pas de réduire cet ensemble.

— *false* : Si on a $state(m \sqcup m') = false$ alors par la Définition 4, il existe une dimension nulle dans $m \sqcup m'$, et donc l'octogone ne contient pas de solution.

— *unknown* : Si m ou m' est inconsistant ($state(m) = false \vee state(m') = false$), alors leur union donne également une DBM inconsistante. Si m et m' sont consistants ($state(m) =$

$true \wedge state(m') = true$) alors leur union ne peut donner qu'une DBM consistante ou inconsistante, vu que toutes les bornes inférieures sont égales aux bornes supérieures, soit $m_{i,j} = m_{\bar{i},\bar{j}}$. Par conséquent, *unknown* n'est possible que lorsqu'une des deux DBMs est *unknown*.

□

Dans le cas des octogones, nous pourrions également fournir une version plus précise de l'opérateur de déduction en utilisant $closure(a \sqcup b)$, mais cette version est plus coûteuse vu qu'elle est de complexité $O(n^3)$—la première version étant en $O(n^2)$. Finalement, notons que dans le cas du RCPSP, nous appliquons l'opérateur de déduction entre une DBM et une seule contrainte octogonale. La complexité de la première version est ainsi en temps constant, et celle de la deuxième en $O(n^2)$ avec la clôture incrémentale.

5.3 Produit réduit des intervalles et octogones

Un nouveau domaine abstrait $Box \times Oct$ issu de la combinaison du domaine des intervalles Box et des octogones Oct peut être obtenu par produit direct. Le produit direct est un produit cartésien de deux domaines abstraits avec les opérateurs redéfinis sur ce produit [16]. Nous pouvons le définir sur $Box \times Oct$ de la manière suivante :

$$\bot = (\bot_{Box}, \bot_{Oct})$$

$$(b, o) \sqcup (b', o') = (box \sqcup_{Box} b', o \sqcup_{Oct} o')$$

$$state((b, o)) = state_{Box}(b) \wedge state_{Oct}(o)$$

$$[\![c]\!] = \begin{cases} ([\![c]\!]_{Box}, [\![c]\!]_{Oct}) & \\ ([\![c]\!]_{Box}, \bot_{Oct}) & \text{si } [\![c]\!]_{Oct} \text{ n'est pas défini} \\ (\bot_{Box}, [\![c]\!]_{Oct}) & \text{si } [\![c]\!]_{Box} \text{ n'est pas défini} \end{cases}$$

$$closure((b, o)) = (closure(b), closure(o))$$

Le problème du produit direct est que les domaines abstraits n'échangent pas d'information. Le *produit réduit* augmente le produit direct avec un échange d'information [16]. Bien sûr, il existe une multitude de produits réduits entre deux domaines abstraits. Nous proposons un produit réduit entre les intervalles et octogones où les deux communiquent exclusivement via des contraintes réifiées. Nous ajoutons au produit direct un ensemble R de contraintes réifiées, et redéfinissons l'opérateur de clôture. Pour ce faire, nous introduisons d'abord un propagateur pour les contraintes réifiées.

À l'aide de l'opérateur de déduction nous pouvons donner un propagateur générique pour les contraintes d'équivalence entre deux domaines abstraits. Bien que ce propagateur soit générique, nous utilisons le domaine des intervalles Box et octogones Oct afin d'illustrer son fonctionnement. Soit $c_1 \Leftrightarrow c_2$ une contrainte d'équivalence où $[\![c_1]\!]_{Box}$ et $[\![c_2]\!]_{Oct}$ sont définis, nous avons :

$$prop_{\Leftrightarrow}(b, o, c_1 \Leftrightarrow c_2) := \begin{cases} b \vDash [\![c_1]\!]_{Box} \implies (b, o \sqcup [\![c_2]\!]_{Oct}) \\ b \nvDash [\![c_1]\!]_{Box} \implies (b, o \sqcup [\![\neg c_2]\!]_{Oct}) \\ o \vDash [\![c_2]\!]_{Oct} \implies (b \sqcup [\![c_1]\!]_{Box}, o) \\ o \nvDash [\![c_2]\!]_{Oct} \implies (b \sqcup [\![\neg c_1]\!]_{Box}, o) \end{cases}$$

Cette fonction peut être généralisée à des contraintes d'équivalence de la forme $c_1 \wedge \ldots \wedge c_n \Leftrightarrow c'_1 \wedge \ldots \wedge c'_m$ en étendant l'opération de déduction sur des conjonctions.

Nous pouvons maintenant définir l'opérateur de clôture du produit réduit $Box \times Oct$:

$$closure_R(box, oct, R) = (\bigsqcup_{r \in R} prop_{\Leftrightarrow}(box, oct, r), R)$$

$$closure((box, oct, R)) = (closure_R(closure(box'), closure(oct')), R))$$

Soit e un élément du produit réduit, l'opérateur de clôture peut être appliqué jusqu'à réaliser le point fixe de $closure(e) = e$.

6 Implémentation et expérimentations

Les fonctions introduites dans les sections précédentes nous permettent d'intégrer le domaine abstrait $Box \times Oct$ dans le solveur abstrait `AbSolute` [19] que nous améliorons pour gérer le cas discret. Nous fournissons trois domaines abstraits : les intervalles, les octogones et le produit réduit de ces deux domaines. L'implémentation et les *benchmarks* présentés dans cette section sont publiquement disponibles à l'adresse github.com/ptal/AbSolute. Les mesures sont réalisées sur un Dell XPS 13 9370, avec un processeur Intel(R) Core(TM) i7-8550U cadencé à 1.8GHz sous GNU Linux. Le solveur `AbSolute` est compilé avec la version `4.07.1+flambda` du compilateur OCaml.

6.1 Implémentation fonctionnelle

Une partie de la complexité d'implémentation d'un solveur de contraintes vient des structures de données qui doivent être *backtrackable*. Par conséquent, à chaque fois qu'une variable doit être restaurée lors d'un *backtrack*, celle-ci doit être manipulée au travers d'une mémoire dédiée qui est gérée par le solveur. Cette mémoire apparait de part et autre dans le code du solveur, complexifiant les interfaces et la programmation, alors qu'il devrait s'agir d'un aspect transversal du solveur.

Le paradigme fonctionnel fournit par défaut une mémoire non-mutable qui peut être utilisée dans des algorithmes de *backtrack* sans effort particulier. De plus, le langage `OCaml`, dans lequel est programmé

	faisable (%)	optimal (%)	insatisfiable (%)
J30	100	100	0
`AbSolute`	100	47.71	0
`GeCode`	100	72.29	0
`Chuffed`	100	**99.38**	0
UBO100	86.87	86.87	13.13
`AbSolute`	50	12.22	11.11
`GeCode`	64.44	32.22	**12.22**
`Chuffed`	**85.55**	**74.44**	8.89

FIGURE 4 – Résultats sur les instances J30 et UBO100.

notre solveur `AbSolute`, propose des traits impératifs classiques. Cela permet à tout moment de choisir si une variable est globale à l'arbre d'exploration (via une référence mutable) ou locale à une branche de l'arbre (via une structure fonctionnelle). L'avantage est que la mémoire *backtrackable* est gérée par le langage lui-même au lieu du solveur `AbSolute`.

On peut comparer la stratégie de restauration offerte par OCaml à une stratégie de *trailing* où seules les modifications sur une structure sont prises en compte, au lieu de réaliser une copie [5]. D'autre part, nous utilisons les tableaux persistants de Conchon et Filliatre [6] afin d'implémenter efficacement la DBM. La technique décrite dans leur article est très proche d'un *trailing* à gros grain, donc sans compression des modifications successives sur une même case mémoire lors de l'étape de propagation (technique appelée *time-stamping*). Nos mesures indiquent néanmoins que la compression ne réduit que de 10% la taille de la structure dans le cas général, et au maximum de 50% sur certaines instances. Dans l'implémentation, nous avons fait le choix de ne pas compresser les modifications.

6.2 Expérimentations préliminaires

On considère deux ensembles d'instances générés par l'outil ProGen/max [13] :

— `J30` pour le problème RCPSP contenant 480 instances de 30 activités [12].
— `UBO100` pour le problème RCPSP/max contenant 90 instances de 100 activités et 5 ressources [11].

Le temps maximal pour résoudre une instance est fixé à 60 secondes.

On teste et compare trois solveurs :

— `AbSolute` avec le produit réduit décrit en Section 5 et un algorithme de *branch and bound* classique.
— `GeCode 6.0.1` avec un modèle du RCPSP utilisant la contrainte globale `cumulative` et un algorithme de *timetabling* [24].
— `chuffed 0.10.0` avec un modèle du RCPSP sans contraintes cumulatives, avec la décomposition par tâches (Section 5.1). Ce solveur utilise la technique de génération de clauses paresseuses (*lazy clause generation*) [18] qui est la méthode la plus efficace pour résoudre le RCPSP et RCPSP/max à ce jour.

La stratégie d'exploration est basée sur les temps de début des tâches : on sélectionne la variable qui a la plus petite borne inférieure, et on affecte cette variable à cette borne.

Des résultats préliminaires sont disponibles sur la Figure 4 où on donne en pourcentage la proportion de problèmes faisables (au moins une solution est trouvée), optimal (preuve de la meilleure solution) et insatisfiables (preuve d'insatisfiabilité). La première ligne de chaque ensemble d'instances (commençant par `J30` et `UBO100`) contient la répartition des instances faisables et insatisfiables. On constate que le solveur `Chuffed` est largement le meilleur, sauf pour les preuves d'insatisfiabilité de `UBO100` où `AbSolute` parvient à en faire 2 de plus dans le temps imparti. On remarque que la différence entre `AbSolute` et `GeCode` est moindre mais reste quand même conséquente.

Il faudra conduire d'autres expérimentations pour tirer des conclusions plus précises sur les faiblesses et forces de ces différentes approches.

7 Conclusion

Les contraintes globales sont nécessaires à l'efficacité de la programmation par contraintes mais sont trop nombreuses. Une solution intermédiaire entre efficacité et expressivité est d'utiliser des domaines abstraits capturant des sous-théories réutilisables pour une large gamme de problèmes. Dans ce but, nous avons présenté un produit réduit de deux domaines abstraits où l'échange d'information se fait via des contraintes réifiées. En particulier, nous avons formalisé et expérimenté cette idée sur le produit réduit des intervalles et octogones afin de résoudre le problème RCPSP. Les expérimentations montrent que l'approche est suffisam-

ment efficace pour de nombreuses instances de tailles variées, mais reste en deçà des algorithmes de l'état de l'art comme la génération de clauses paresseuses.

Les perspectives futures sont variées, tant au niveau de la performance brute du domaine abstrait que des problèmes pouvant être résolus avec cette méthode. Premièrement, il est possible d'optimiser les octogones afin de cibler des instances de grandes tailles (au delà de 200 tâches), notamment par décomposition en sous-octogones [23]. Deuxièmement, de nombreuses extensions du RCPSP existent [17] et cachent certainement des contraintes propices à l'utilisation de domaines abstraits. Finalement, d'autres contraintes globales peuvent être décomposées vers des contraintes octogonales comme la contrainte `sequence` [14]. L'avantage étant qu'on pourrait supporter ces contraintes efficacement sans effort puisque le domaine des octogones existe déjà.

Références

[1] Roberto Bagnara, Patricia M. Hill et Enea Zaffanella : Weakly-relational shapes for numeric abstractions : Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[2] Nicolas Beldiceanu, Mats Carlsson et Jean-Xavier Rampon : Global Constraint Catalog, 2nd Edition (revision a), février 2011. SICS research report T2012-03, http ://soda.swedish-ict.se/5195/.

[3] Christian Bessiere, Stéphane Cardon, Romuald Debruyne et Christophe Lecoutre : Efficient algorithms for singleton arc consistency. *Constraints*, 16(1):25–53, janvier 2011.

[4] Aziem Chawdhary, Ed Robbins et Andy King : Incrementally closing octagons. *Formal Methods in System Design*, janvier 2018.

[5] Chiu Wo Choi, Martin Henz et Ka Boon Ng : Components for state restoration in tree search. *In Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, CP '01, pages 240–255, London, UK, UK, 2001. Springer-Verlag.

[6] Sylvain Conchon et Jean-Christophe Filliâtre : Semi-persistent data structures. *In European Symposium on Programming*, pages 322–336. Springer, 2008.

[7] Patrick Cousot et Radhia Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[8] Patrick Cousot, Radhia Cousot et Laurent Mauborgne : Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM (JACM)*, 59(6):31, 2012.

[9] Rina Dechter, Itay Meiri et Judea Pearl : Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[10] Thibaut Feydy, Andreas Schutt et Peter J. Stuckey : Global difference constraint propagation for finite domain solvers. *In Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 226–235. ACM, 2008.

[11] Birger Franck, Klaus Neumann et Christoph Schwindt : Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling. *OR-Spektrum*, 23(3):297–324, 2001.

[12] Rainer Kolisch et Arno Sprecher : PSPLIB-a project scheduling problem library. *European journal of operational research*, 96(1):205–216, 1997.

[13] Rainer Kolisch, Arno Sprecher et Andreas Drexl : Characterization and generation of a general class of resource-constrained project scheduling problems. *Management science*, 41(10):1693–1703, 1995.

[14] Michael Maher, Nina Narodytska, Claude-Guy Quimper et Toby Walsh : Flow-based propagators for the SEQUENCE and related global constraints. *In International Conference on Principles and Practice of Constraint Programming*, pages 159–174. Springer, 2008.

[15] A. Miné : The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1): 31–100, 2006.

[16] A. Miné : Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 4(3–4):120–372, 2017.

[17] Ernesto Nunes, Marie Manner, Hakim Mitiche et Maria Gini : A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90:55–70, avril 2017.

[18] Olga Ohrimenko, Peter J. Stuckey et Michael Codish : Propagation via lazy clause generation. *Constraints*, 14(3):357–391, septembre 2009.

[19] Marie Pelleau, Antoine Miné, Charlotte Truchet et Frédéric Benhamou : A constraint solver

based on abstract domains. *In Verification, Model Checking, and Abstract Interpretation*, pages 434–454. Springer, 2013.

[20] Marie Pelleau, Charlotte Truchet et Frédéric Benhamou : The octagon abstract domain for continuous constraints. *Constraints*, 19(3):309–337, 2014.

[21] Vaughan Pratt : Two easy theories whose combination is hard. Rapport technique, Technical report, Massachusetts Institute of Technology, 1977.

[22] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey et Mark G. Wallace : Why cumulative decomposition is not as bad as it sounds. *In International Conference on Principles and Practice of Constraint Programming*, pages 746–761. Springer, 2009.

[23] Gagandeep Singh, Markus Püschel et Martin Vechev : Making numerical program analysis fast. pages 303–313. ACM Press, 2015.

[24] Petr Vilím : *Global constraints in scheduling*. Thèse de doctorat, Charles University in Prague, 2007.