La contrainte Increasing NValue

Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca, Thierry Petit

La contrainte Increasing Nvalue

Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca et Thierry Petit

Mines-Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
{Nicolas.Beldiceanu,Fabien.Hermenier,Xavier.Lorca,Thierry.Petit}@mines-nantes.fr

Résumé

Cet article introduit la contrainte INCREASING_NVALUE, qui restreint le nombre de valeurs distinctes affectées à une séquence de variables, de sorte que chaque variable de la séquence soit inférieure ou égale à la variable la succédant immédiatement. Cette contrainte est une spécialisation de la contrainte NVALUE, motivée par le besoin de casser des symétries. Il est bien connu que propager la contrainte NVALUE est un problème NP-Difficile. Nous montrons que la spécialisation au cas d'une séquence ordonnée de variables rend le problème polynomial. Nous proposons un algorithme d'arc-consistance ayant une complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, où Σ_{Di} est la somme des tailles des domaines. Cette algorithme est une amélioration significative, en termes de complexité, des algorithmes issus d'une représentation de la contrainte INCREASING_NVALUE à l'aide d'automates ou de la contrainte SLIDE. Nous utilisons notre contrainte dans le cadre d'un problème d'allocation de ressources.

1 Introduction

La contrainte NVALUE [9] exprime une restriction du nombre de valeurs distinctes affectées à un ensemble de variables. Bien que tester si NVALUE admet une solution ou pas soit un problème NP-Difficile [5], un certain nombre d'algorithmes de filtrage ont été développés ces dernières années [3, 2]. Dans le but de casser des symétries, nous présentons dans cet article la contrainte INCREASING_NVALUE, qui représente la conjonction de NVALUE avec une chaîne de contraintes binaires d'inégalité non strictes. Nous proposons un algorithme de filtrage qui réalise la consistance d'arc généralisée (GAC) pour INCREASING_NVALUE avec une complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, où Σ_{Di} est la somme des tailles des domaines. Cet algorithme est plus efficace les algorithmes issus d'une représentation de la contrainte INCREASING_NVALUE à l'aide d'automates finis déterministes [11] ou de la contrainte SLIDE [4], qui réalisent le même filtrage en, respectivement, $O(n(\cup_{Di})^3)$ et $O(nd^4)$, où n est le nombre de variables, $\cup_{Di}$ la taille de l'union des domaines, et d la taille maximale d'un domaine. Cette efficacité provient en partie d'une structure de données spécifique, une matrice creuse avec accès ordonnés aux colonnes.

Nos expérimentations sont réalisées sur un problème réel d'allocation de ressources, relatif à la gestion de grappes dans un réseau de machines. *Entropy* est une machine virtuelle (VM) pour la gestion de grappes [7], qui propose un moteur flexible et autonome pour manipuler l'état et la position de VM sur les différents nœuds composant une grappe. La programmation par contraintes intervient dans l'affectation des VM (tâches) sur un nombre réduit de nœuds (les ressources) dans la grappe. La contrainte NVALUE intervient pour maintenir le nombre de nœuds nécessaires pour couvrir toutes les VM. Cependant, en pratique on constate que les ressources consommées sont le plus souvent équivalentes d'une VM à une autre. Cette propriété induit qu'il existe un nombre limité de classes d'équivalences de VM, et on peut profiter du filtrage polynomial et linéaire de la contrainte INCREASING_NVALUE pour casser les symétries induites par les classes d'équivalence.

La section 2 rappelle quelques définitions et introduit formellement la contrainte INCREASING_NVALUE. La section 3 décrit une condition nécessaire et suffisante pour tester l'existence d'une solution pour INCREASING_NVALUE. La section 4 présente l'algorithme réalisant GAC. La section 5 évalue l'impact de la contrainte sur le problème d'allocation de ressource. Enfin, nous décrivons dans la section 6 les approches génériques pour reformuler INCREASING_NVALUE, qui s'avèrent être moins efficaces que notre algorithme.

2 Définitions

Étant donnée une séquence X de variables, le *domaine* $\mathrm{D}(x)$ d'une *variable* $x \in X$ est l'ensemble fini des valeurs entières pouvant être affectées à cette variable. $\mathcal{D}$ est l'union des domaines des variables X. On note $\min(x)$ la valeur minimale de $D(x)$ et $\max(x)$ sa valeur maximale. La somme des tailles des domaines de $\mathcal{D}$ est $\Sigma_{\mathrm{Di}} = \sum_{x_i \in X} |D(x_i)|$. $A[X]$ désigne une affectation de valeurs aux variables X. Étant donnée $x \in X$, $A[x]$ est la valeur de x dans $A[X]$. $A[X]$ est *valide* ssi $\forall x_i \in X$, $A[x_i] \in \mathrm{D}(x_i)$. Une *instantiation* $I[X]$ est une affectation complète de valeurs aux variables X. Étant donnée $x \in X$, $I[x]$ est la valeur de x dans $I[X]$. Une contrainte $C(X)$ spécifie les combinaisons de valeurs acceptées pour un ensemble de variables X. Elle définit un sous ensemble $\mathcal{R}_C(\mathcal{D})$ du produit cartésien des domaines $\Pi_{x_i \in X} \mathrm{D}(x_i)$. Toute instantiation solution de $C(X)$ appartient à $\mathcal{R}_C(\mathcal{D})$. On emploie le terme d'instantiation *faisable* ou *réalisable*, ou encore, on dira que si $I[X]$ est une solution de $C(X)$ alors $I[X]$ *satisfait* $C(X)$. Sans perte de généralité, nous considérons dans cet article que X contient au moins deux variables. Étant donnés $X = [x_0, \ldots, x_{n-1}]$ et deux entiers i et j tels que $0 \leq i < j \leq n-1$, $I[x_i, \ldots, x_j]$ est la projection de $I[X]$ sur $[x_i, \ldots, x_j]$.

Définition 1 Increasing_Nvalue(N, X) *est définie par une variable N et une séquence de n variables $X = [x_0, \ldots, x_{n-1}]$. Soit une instantiation de $[N, x_0, \ldots, x_{n-1}]$. Elle satisfait* Increasing_Nvalue(N, X) *ssi :*

1. *N est égale au nombre de valeurs distinctes affectées aux variables X,*
2. *$\forall i \in [0, n-2]$, $x_i \leq x_{i+1}$.*

3 Faisabilité de Increasing Nvalue

Cette section présente une condition de faisabilité nécessaire et suffisante pour Increasing_Nvalue. Nous montrons tout d'abord que le nombre de valeurs distinctes d'une instantiation $I[X]$ telle que $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$ est égal au nombre de stretchs dans $I[X]$: un *stretch* [10] est une séquence de variables consécutives de longueur maximale qui sont affectées à la même valeur. Pour chaque valeur v de chaque domaine $D(x)$, nous calculons les nombres minimal et maximal de stretchs parmi toutes les instantiations $I[X]$ telles que $I[x] = v$. Ensuite, étant donnée une variable $x \in X$, nous établissons les propriétés liant l'ordre naturel des valeurs dans $D(x)$ et les nombres de stretchs minimal et maximal qui peuvent être obtenus en affectant une valeur à x. À partir de ces propriétés, nous prouvons qu'il existe une instantiation satisfaisant Increasing_Nvalue(N, X) pour chaque valeur de $\mathrm{D}(N)$ comprise entre le minimum $\underline{s}(X)$ et le maximum $\overline{s}(X)$ des nombres de stretchs possibles. Cette propriété nous amène au résultat majeur de la section : Increasing_Nvalue(N, X) est réalisable ssi $\mathrm{D}(N) \cap [\underline{s}(X), \overline{s}(X)] \neq \emptyset$ (Proposition 3 de la Section 3.3).

3.1 Estimation du nombre de stretchs

Une instantiation réalisable $I[X]$ de Increasing_Nvalue(N, X) satisfait la propriété suivante : $I[x_i] \leq I[x_j]$ pour tout $i < j$. Dans la suite de l'article, une instantiation $I[x_0, x_1, \ldots, x_{n-1}]$ est dite *bien ordonnée* ssi pour tout i et tout j tels que $0 \leq i < j \leq n-1$, on a $I[x_i] \leq I[x_j]$. Une valeur $v \in \mathrm{D}(x)$ est bien ordonnée pour x ssi elle peut appartenir à au moins une instantiation bien ordonnée.

Lemme 1 *Soit $I[X]$ une instantiation. Si $I[X]$ satisfait* Increasing_Nvalue(X, N) *alors $I[X]$ est bien ordonnée.*

Preuve : D'après la Définition 1, si $I[X]$ satisfait Increasing_Nvalue alors $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$. Par transitivité de $\leq$, le lemme est correct.

Définition 2 (stretch) *Soit $I[x_0, \ldots, x_{n-1}]$ une instantiation. Soient i et j tels que $0 \leq i \leq j \leq n-1$. Un* stretch *de $I[X]$ est une séquence de variables consécutives $[x_i, \ldots, x_j]$ telle que dans $I[X]$: (1) $\forall k \in [i, j]$, $\forall \ell \in [i, j]$, $x_k = x_\ell$. (2) soit $i = 0$ soit $x_{i-1} \neq x_i$. (3) soit $j = n-1$ soit $x_j \neq x_{j+1}$.*

Lemme 2 *Soit $I[X]$ une instantiation bien ordonnée. Le nombre de stretchs dans $I[X]$ est égal au nombre de valeurs distinctes dans $I[X]$.*

Preuve : $I[X]$ est bien ordonnée, donc pour tout i et tout j tels que $0 \leq i < j \leq n-1$, on a $I[x_i] \leq I[x_j]$. En conséquence, si x_i et x_j appartiennent à deux stretchs distincts (et $i < j$) alors $I[x_i] < I[x_j]$. La réciproque est trivialement vraie.

Il est possible d'évaluer pour chaque valeur v dans chaque domaine $\mathrm{D}(x_i)$ les minima et maxima exacts du nombre de stretchs d'instantiation suffixes bien ordonnées $I[x_i, \ldots, x_n]$ telles que $I[x_i] = v$, et similairement pour des instantiations préfixes.

Notation 1 *Soit $X = [x_0, x_1, \ldots, x_{n-1}]$ une séquence de variables et soit v une valeur dans $\mathcal{D}$. Le nombre minimal (exact) de stretchs parmi toutes les instantiations bien ordonnées $I[x_i, \ldots, x_{n-1}]$ telles que $I[x_i] = v$ est noté $\underline{s}(x_i, v)$. Par convention, si $v \notin \mathrm{D}(x_i)$*

alors $\underline{s}(x_i, v) = +\infty$. *Similairement, le nombre minimal (exact) de stretchs parmi toutes les instantiations bien ordonnées* $I[x_0, \ldots, x_i]$ *telles que* $I[x_i] = v$ *est noté* $\underline{p}(x_i, v)$. *Par convention, si* $v \notin \mathrm{D}(x_i)$ *alors* $\underline{p}(x_i, v) = +\infty$.

Lemme 3 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables.* $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\underline{s}(x_i, v)$ *peut être calculé ainsi :*

1. *Si* $i = n - 1$: $\underline{s}(x_i, v) = 1$,
2. *Si* $i < n - 1$: $\underline{s}(x_i, v) = \min(\ \underline{s}(x_{i+1}, v),\ \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1\ )$.

Preuve : par induction. Si $|X| = 1$ il y a un stretch, donc quelque soit $v \in \mathrm{D}(x_i)$, on a $\underline{s}(x_i, v) = 1$. Considérons à présent une variable x_i, $i < n - 1$, et une valeur $v \in \mathrm{D}(x_i)$. Les instantiations telles que $I[x_{i+1}] < v$ ne peuvent pas être étendues avec la valeur v pour x_i pour former une instantiation bien ordonnée $I[x_i, \ldots, x_{n-1}]$. Soit $I[x_{i+1}, \ldots, x_{n-1}]$ une instantiation telle que $I[x_{i+1}] \geq v$, minimisant le nombre de stretchs dans $[x_{i+1}, \ldots, x_{n-1}]$. Soit $I[x_{i+1}] = v$ et $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$, car le premier stretch de $I[x_{i+1}, \ldots, x_{n-1}]$ est étendu lorsqu'on ajoute à $I[x_{i+1}, \ldots, x_{n-1}]$ la valeur v pour x_i, soit $I[x_{i+1}] \neq v$ et $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, I[x_{i+1}]) + 1$ car v crée un nouveau stretch. Par construction, les instantiations des variables $[x_{i+1}, \ldots, x_{n-1}]$ ne minimisant pas le nombre de stretchs ne peuvent pas aboutir sur une valeur $\underline{s}(x_i, v)$ strictement inférieure à $\min(\underline{s}(x_{i+1}, w), w > v)+1$, et ce même si $I[x_{i+1}] = v$.

Étant donnée une séquence de variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, calculer $\underline{p}(x_i, v)$ est symétrique : si $i = 0$: $\underline{p}(x_i, v) = 1$. Si $i > 0$: $\underline{p}(x_i, v) = \min(\ \underline{p}(x_{i-1}, v),\ \min_{w<v}(\underline{p}(x_{i-1}, w)) + 1\ )$.

De plus, étant donnée une variable x_i, nous évaluons pour chaque valeur v le nombre maximal (exact) de stretchs qui peuvent apparaître, parmi toutes les instantiations suffixes bien ordonnées $I[x_i, \ldots, x_{n-1}]$ avec $I[x_i] = v$, et similairement pour les instantiations préfixes.

Notation 2 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables et* v *une valeur dans* $\mathcal{D}$. *Le nombre maximal (exact) de stretchs parmi toutes les instantiations bien ordonnées* $I[x_i, \ldots, x_{n-1}]$ *avec* $I[x_i] = v$ *est noté* $\overline{s}(x_i, v)$. *Par convention, si* $v \notin \mathrm{D}(x_i)$ *alors* $\overline{s}(x_i, v) = 0$. *Similairement, le nombre maximal (exact) de stretchs parmi toutes les instantiations bien ordonnées* $I[x_1, \ldots, x_i]$ *avec* $I[x_i] = v$ *est noté* $\overline{p}(x_i, v)$. *Par convention, si* $v \notin \mathrm{D}(x_i)$ *alors* $\overline{p}(x_i, v) = 0$.

Lemme 4 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables.* $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\overline{s}(x_i, v)$ *peut être calculé ainsi :*

1. *Si* $i = n - 1$: $\overline{s}(x_i, v) = 1$,
2. *Si* $i < n - 1$: $\overline{s}(x_i, v) = \max(\ \overline{s}(x_{i+1}, v),\ \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1\ )$.

Preuve : Similaire au Lemme 3.

Étant donnée une séquence de variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, le calcul de $\overline{p}(x_i, v)$ est symétrique : si $i = 0$: $\overline{p}(x_i, v) = 1$, si $i > 0$: $\overline{p}(x_i, v) = \max(\ \overline{p}(x_{i-1}, v),\ \max_{w<v}(\overline{s}(x_{i-1}, w)) + 1\ )$.

3.2 Propriétés sur le nombre de stretchs

Nous considérons exclusivement des valeurs bien ordonnées, pouvant appartenir à une instantiation réalisable pour Increasing_Nvalue.

3.2.1 Propriétés sur une unique valeur

Les propriétés suivantes sont directement déduites, par construction, des Lemmes 3 et 4.

Propriété 1 *Toute valeur* $v \in \mathrm{D}(x_i)$ *bien ordonnée pour* x_i *est telle que* $\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$.

Propriété 2 *Soit* $v \in \mathrm{D}(x_i)$ *(*$i < n - 1$*) une valeur bien ordonnée pour* x_i. *Si* $v \in \mathrm{D}(x_{i+1})$ *et* v *est bien ordonnée pour* x_{i+1} *alors* $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$.

Propriété 3 *Soit* $v \in \mathrm{D}(x_i)$ *(*$i < n - 1$*) une valeur bien ordonnée pour* x_i. *Si* $v \in \mathrm{D}(x_{i+1})$ *et* v *est bien ordonnée pour* x_{i+1} *alors* $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$.

Preuve : D'après le Lemme 4, s'il existe une valeur $w \in D(x_{i+1})$, $w > v$ qui est bien ordonnée pour x_{i+1} et telle que $\overline{s}(x_{i+1}, w) \geq \overline{s}(x_{i+1}, v)$ alors $\overline{s}(x_i, v) > \overline{s}(x_{i+1}, v)$. Dans le cas contraire, $\overline{s}(x_i, v) = \overline{s}(x_{i+1}, v)$.

3.2.2 Ordre sur les valeurs

Les deux propriétés suivantes établissent les liens entre l'ordre naturel des valeurs dans un domaine $D(x_i)$ avec le nombre minimal et maximal de stretchs que l'on peut obtenir dans la sous-séquence $[x_i, x_{i+1}, \ldots, x_{n-1}]$.

Propriété 4 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables et* $i \in [0, n - 1]$ *un entier.* $\forall v, w \in \mathrm{D}(x_i)$ *deux valeurs bien ordonnées,* $v \leq w \Rightarrow \underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$.

Preuve : Si $v = w$ alors la propriété est vraie. Si $i = n-1$, d'après le Lemme 3, $\underline{s}(x_{n-1}, v) = \underline{s}(x_{n-1}, w) = 1$. La propriété est vérifiée. Étant donné $i < n-1$, soit v', w' deux valeurs bien ordonnées de $\mathrm{D}(x_{i+1})$ telles que $v' \geq v$ et $w' \geq w$, qui minimisent le nombre de stretchs débutant en x_{i+1} : $\forall \alpha \geq v$, $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, \alpha)$ et $\forall \beta \geq w$, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_{i+1}, \beta)$. Ces valeurs existent car v et w sont bien ordonnées. Par construction, on a $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, w')$, et d'après le Lemme 3, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_i, w)$, ce qui entraîne $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_i, w)$. D'après le Lemme 3, $\underline{s}(x_i, v) \leq \underline{s}(x_{i+1}, v') + 1$. Ainsi, $\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$.

Une propriété symétrique existe concernant le nombre maximal de stretchs.

Propriété 5 *Soit $X = [x_0, x_1, \ldots, x_{n-1}]$ une séquence de variables et $i \in [0, n-1]$ un entier. $\forall v, w \in \mathrm{D}(x_i)$ deux valeurs bien ordonnées, $v \leq w \Rightarrow \overline{s}(x_i, v) \geq \overline{s}(x_i, w)$.*

Preuve : si $v = w$ alors la propriété est vraie. Si $i = n-1$, d'après le Lemme 4, $\overline{s}(x_{n-1}, v) = \overline{s}(x_{n-1}, w) = 1$. La propriété est vérifiée. Étant donné $i < n-1$, soit $w' \in \mathrm{D}(x_{i+1})$ une valeur bien ordonnée telle que $w' \geq w$, qui maximise le nombre de stretchs débutant en x_{i+1} : $(\forall \beta \geq w,\ \overline{s}(x_{i+1}, w') \geq \overline{s}(x_{i+1}, \beta))$. D'après le Lemme 4, $\overline{s}(x_i, w) \leq \overline{s}(x_{i+1}, w') + 1$. Sachant que $v < w$ et donc $v < w'$, $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, w') + 1$. La propriété est vraie.

3.2.3 Ordre sur le nombre maximal de stretchs

L'intuition de la propriété 6 réside dans le fait que, plus une valeur bien ordonnée pour une variable x_i est petite, plus le nombre de stretchs qui peuvent être obtenus sur la séquence $[x_i, x_{i+1}, \ldots, x_{n-1}]$, à partir de cette valeur, est grand.

Propriété 6 *Soit $X = [x_0, x_1, \ldots, x_{n-1}]$ une séquence de variables et soit i un entier dans $[0, n-1]$. $\forall v, w \in \mathrm{D}(x_i)$ deux valeurs bien ordonnées, $\overline{s}(x_i, w) < \overline{s}(x_i, v) \Rightarrow v < w$.*

Preuve : Nous prouvons que si $v \geq w$ alors il existe une contradiction avec $\overline{s}(x_i, w) < \overline{s}(x_i, v)$. Si $i = n-1$, le Lemme 4 assure $\overline{s}(x_{n-1}, w) = \overline{s}(x_{n-1}, v) = 1$, une contradiction. Considérons le cas $i < n-1$. Si $v = w$ alors $\overline{s}(x_i, w) = \overline{s}(x_i, v)$, une contradiction. Sinon $(v > w)$, soit v' une valeur de $\mathrm{D}(x_{i+1})$ telle que $v' \geq v$, qui maximise $\overline{s}(x_{i+1}, \alpha)$, $\alpha \geq v$. Une telle valeur existe car v est bien ordonnée. Par construction $w < v'$. D'après le Lemme 4, $\overline{s}(x_i, w) \geq \overline{s}(x_{i+1}, v') + 1$ (1). Par construction on a aussi $v \leq v'$, qui entraîne $\overline{s}(x_{i+1}, v') + 1 \geq \overline{s}(x_i, v)$ (2). D'après (1) et (2) on a $\overline{s}(x_i, w) \geq \overline{s}(x_i, v)$, une contradiction.

3.2.4 Ordre sur le nombre minimal de stretchs

Il n'existe pas d'implication liant le nombre minimal de stretchs et l'ordre des valeurs dans les domaines. Par exemple, soit $X = [x_0, x_1, x_2]$ avec $D(x_0)$=$D(x_1)$=$\{1, 2, 3\}$ et $D(x_2) = \{1, 2, 4\}$. $\underline{s}(x_0, 1) = 1$ et $\underline{s}(x_0, 3) = 2$, alors $\underline{s}(x_0, 1) < \underline{s}(x_0, 3)$ et $1 < 3$. Supposons à présent que $D(x_2) = \{2, 3, 4\}$. $\underline{s}(x_0, 1) = 2$ et $\underline{s}(x_0, 3) = 1$, alors $\underline{s}(x_0, 3) < \underline{s}(x_0, 1)$ et $3 > 1$.

3.2.5 Résumé

Le tableau ci-dessous récapitule les relations qui existent entre des valeurs bien ordonnées v et w d'un domaine $D(x_i)$ et les estimations du nombre minimal et maximal de stretchs parmi toutes les instantiations débutant par ces valeurs (*i.e.*, $I[x_i, \ldots, x_{n-1}]$ telle que $I[x_i] = v$ ou telle que $I[x_i] = w$).

Pré-condition	Propriété	Prop.
$v \in D(x_i)$ est bien ordonnée	$\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$	Prop. 1
$v \in D(x_i)$ est bien ordonnée, $i < n-1$ et $v \in D(x_{i+1})$	$\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$ $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$	Prop. 2 Prop. 3
$v \in D(x_i)$, $w \in D(x_i)$ sont bien ordonnées et $v \leq w$	$\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$ $\overline{s}(x_i, v) \geq \overline{s}(x_i, w)$	Prop. 4 Prop. 5
$v \in D(x_i)$, $w \in D(x_i)$ sont bien ordonnées et $\overline{s}(x_i, w) < \overline{s}(x_i, v)$	$v < w$	Prop. 6

3.3 Condition nécessaire et suffisante de faisabilité

Notation 3 *Étant donnée une séquence de variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\underline{s}(X)$ est la valeur minimale de $\underline{s}(x_0, v)$, $v \in \mathrm{D}(x_0)$, et $\overline{s}(X)$ est la valeur maximale de $\overline{s}(x_0, v)$, $v \in \mathrm{D}(x_0)$.*

Proposition 1 *Étant donnée* IncreasingNvalue(N, X), *si $\underline{s}(X) > \max(\mathrm{D}(N))$ alors la contrainte n'a pas de solution. symétriquement, si $\overline{s}(X) < \min(\mathrm{D}(N))$ alors la contrainte n'a pas de solution.*

Preuve : Par construction d'après les Lemmes 3 et 4.

Sans perte de généralité, $\mathrm{D}(N)$ peut être réduit à $[\underline{s}(X), \overline{s}(X)]$. Néanmoins, on peut observer que $\mathrm{D}(N)$ peut avoir des trous ou bien être strictement inclus dans $[\underline{s}(X), \overline{s}(X)]$. Nous montrons que pour toute valeur k dans $[\underline{s}(X), \overline{s}(X)]$ il existe une valeur dans $v \in \mathrm{D}(x_0)$ telle que $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$. Ainsi, toute valeur de $D(N) \cap [\underline{s}(X), \overline{s}(X)]$ est une valeur réalisable.

Proposition 2 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables. Pour tout entier* k *dans* $[\underline{s}(X), \overline{s}(X)]$ *il existe* v *dans* $\mathrm{D}(x_0)$ *telle que* $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$.

Preuve : Soit $k \in [\underline{s}(X), \overline{s}(X)]$. Si $\exists v \in \mathrm{D}(x_0)$ telle que $k = \underline{s}(x_0, v)$ ou $k = \overline{s}(x_0, v)$, alors la propriété est vraie par construction. Supposons que $\forall v \in \mathrm{D}(x_0)$, dans ce cas soit $k > \overline{s}(x_0, v)$ soit $k < \underline{s}(x_0, v)$. Soient v', w' deux valeurs telles que v' soit la plus grande valeur de $\mathrm{D}(x_0)$ telle que $\overline{s}(x_0, v') < k$ et w' soit la plus grande valeur telle que $k < \underline{s}(x_0, w')$. Alors, $\overline{s}(x_0, v') < k < \underline{s}(x_0, w')$ (1). D'après la propriété 1, $\underline{s}(x_0, w') \leq \overline{s}(x_0, w')$. D'après la propriété 6, $\overline{s}(x_0, v') < \overline{s}(x_0, w') \Rightarrow w' < v'$. D'après les propriétés 4 et 1, $w' < v' \Rightarrow \underline{s}(x_0, w') \leq \overline{s}(x_0, v') + 1$, qui est en contradiction avec (1).

Algorithm 1: Construction d'une solution pour Increasing_Nvalue(k, X).

```
1 if k ∉ [s(X), s̄(X)] ∩ D(N) then return "no solution" ;
2 v := a value ∈ D(x_0) s.t. k ∈ [s(x_0, v), s̄(x_0, v)] ;
3 for i := 0 to n − 2 do
4 |   I[x_i] := v;
5 |   if ∀v_{i+1} ∈ D(x_{i+1}) s.t. v_{i+1} = v,
  |   k ∉ [s(x_{i+1}, v_{i+1}), s̄(x_{i+1}, v_{i+1})] then
6 |   |   v := v_{i+1} in D(x_{i+1}) s.t. v_{i+1} > v ∧
  |   |   k − 1 ∈ [s(x_{i+1}, v_{i+1}), s̄(x_{i+1}, v_{i+1})];
7 |   |   k := k − 1 ;
8 I[x_{n−1}] := v ; return I[X];
```

Lemme 5 *Étant donnée une contrainte* Increasing_Nvalue(N, X) *et un entier* k*, si* $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$ *alors l'algorithme 1 retourne une solution pour* Increasing_Nvalue(N, X) *avec* $N = k$*. Sinon, il n'existe aucune solution avec* $N = k$.

Preuve : la première ligne de l'algorithme assure que soit $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$ et $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$, soit il n'existe pas de solution (d'après les Propositions 1 et 2). À chaque itération de la boucle **for**, d'après les lemmes 3 et 4 et la proposition 2, soit la condition (ligne 6) est satisfaite et un nouveau stretch débute en $i + 1$ avec une valeur supérieure (qui garantit que $I[\{x_1, \ldots, x_{i+1}\}]$ est bien ordonnée) et k est décrémenté de 1, soit il est possible de garder la valeur courante v pour $I[x_{i+1}]$. C'est pourquoi au début de la boucle (ligne 4), $\exists v \in \mathrm{D}(x_i)$ telle que $k \in [\underline{s}(x_i, v), \overline{s}(x_i, v)]$. Lorsque $i = n - 1$, par construction $k = 1$ et $\forall v_{n-1} \in \mathrm{D}(x_{n-1})$, $\underline{s}(x_{n-1}, v_{n-1}) = \overline{s}(x_{n-1}, v_{n-1}) = 1$; $I[X]$ est bien ordonnée et contient k stretchs. D'après le Lemme 2, l'instantiation $I[\{N\} \cup X]$ with $I[N] = k$ est une solution de Increasing_Nvalue(N, X) avec k valeurs distinctes pour les variables X.

Le Lemme 5 nous amène à une condition nécessaire et suffisante de faisabilité.

Proposition 3 *Étant donnée une contrainte* Increasing_Nvalue(N, X)*, les deux propositions suivantes sont équivalentes :*

1. Increasing_Nvalue(N, X) *admet une solution.*
2. $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$.

Preuve : ($\Rightarrow$) Supposons que la contrainte Increasing_Nvalue(N, X) admette une solution. Soit $I[\{N\} \cup X]$ une telle solution. D'après le Lemme 2 la valeur k affectée à N est égale au nombre de stretchs dans $I[X]$. Par construction (Lemmes 3 et 4) $k \in [\underline{s}(X), \overline{s}(X)]$. Il s'en suit $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. ($\Leftarrow$) Soit $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. D'après le Lemme 5 il est possible de construire une solution pour la contrainte Increasing_Nvalue(N, X).

4 GAC pour Increasing Nvalue

Cette section présente un algorithme de filtrage réalisant GAC pour Increasing_Nvalue(N, X) avec une complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, où Σ_{Di} est la somme des tailles des domaines des variables X. Pour une variable $x_i \in X$ et une valeur $v \in D(x_i)$, le principe consiste à déterminer le nombre minimal et le nombre maximal de stretchs parmi toutes les instantiations $I[X]$ telles que $I[x_i] = v$, puis à comparer l'intervalle dérivé de ces deux bornes avec $D(N)$. Dans ce but, et sans perte de généralité, nous déterminons le nombre minimal et le nombre maximal de stretchs relatifs aux instantiations préfixes $I[x_0, \ldots, x_i]$ et aux instantiations suffixes $I[x_i, \ldots, x_{n-1}]$.

Définition 3 (GAC) *Soit une contrainte* $C(X)$*. Un* **support** *sur* $C(X)$ *est une instantiation* $I[X]$ *qui satisfait* $C(X)$*. Un domaine* $\mathrm{D}(x)$ *est* **arc-consistant** *sur* $C(X)$ *ssi* $\forall v \in \mathrm{D}(x)$*,* v *appartient à un support sur* $C(X)$*.* $C(X)$ *a la propriété d'***arc-consistance généralisée** *(GAC) ssi* $\forall x_i \in X$*,* $\mathrm{D}(x_i)$ *est arc-consistant.*

4.1 Condition nécessaire et suffisante de filtrage

D'après le Lemme 5, les valeurs de $D(N)$ qui ne sont pas dans $[\underline{s}(X), \overline{s}(X)]$ peuvent être supprimées de $D(N)$. D'après la Proposition 3, toutes les valeurs restant dans $D(N)$ sont réalisables. Nous fournissons à présent une condition nécessaire et suffisante pour supprimer une valeur d'un domaine $D(x_i)$, $x_i \in X$.

Proposition 4 *Considérons une contrainte* Increasing_Nvalue(N, X)*. Soit* $i \in [0, n-1]$ *un entier et* v *une valeur dans* $D(x_i)$*. Les deux propositions suivantes sont équivalentes :*

1. $v \in D(x_i)$ *est arc-consistant pour* Increasing_Nvalue
2. v *est bien ordonnée relativement à* $D(x_i)$ *et l'intersection* $[\underline{p}(x_i,v)+\underline{s}(x_i,v)-1, \overline{p}(x_i,v)+\overline{s}(x_i,v)-1] \cap \mathrm{D}(N)$ *n'est pas vide.*

Preuve : si v n'est pas bien ordonnée alors d'après le Lemme 1, v n'est pas arc-consistante sur Increasing_Nvalue. Sinon, $\underline{p}(x_i,v)$ est le nombre minimal exact de stretchs parmi les instantiations bien ordonnées $I[x_0,\ldots,x_i]$ telles que $I[x_i]=v$ et $\underline{s}(x_i,v)$ est le nombre minimal exact de stretchs parmi les instantiations bien ordonnées $I[x_i,\ldots,x_{n-1}]$ telles que $I[x_i]=v$. Donc, par construction, $\underline{p}(x_i,v)+\underline{s}(x_i,v)-1$ est le nombre minimal exact de stretchs parmi les instantiations bien ordonnées $I[x_0,\ldots,x_{n-1}]$ telles que $I[x_i]=v$. Soit $\mathcal{D}_v \subseteq \mathcal{D}$ l'ensemble de domaines tels que tous les domaines dans $\mathcal{D}_v$ soient identiques aux domaines de $\mathcal{D}$ sauf $D(x_i)$ qui est réduit au singleton $\{v\}$. Nous appelons X_v l'ensemble des variables associées aux domaines de $\mathcal{D}_v$. D'après la Definition 3, $\underline{p}(x_i,v)+\underline{s}(x_i,v)-1=\underline{s}(X_v)$. Par un raisonnement symétrique, $\overline{p}(x_i,v)+\overline{s}(x_i,v)-1=\overline{s}(X_v)$. D'après la proposition 3, la proposition est vraie.

4.2 Algorithmes

D'après la proposition 4, nous dérivons un algorithme de filtrage réalisant GAC en $O(\Sigma_{\mathrm{Di}})$. Pour une variable x_i $(0 \le i < n)$, nous avons besoin de calculer les informations préfixes et suffixes $\underline{p}(x_i,v)$, $\overline{p}(x_i,v)$, $\underline{s}(x_i,v)$ et $\overline{s}(x_i,v)$, et ce que la valeur v appartienne ou non à $D(x_i)$. Afin d'atteindre la complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, nous tirons parti de deux points :

1. L'algorithme itère toujours sur $\underline{p}(x_i,v)$, $\overline{p}(x_i,v)$, $\underline{s}(x_i,v)$ et $\overline{s}(x_i,v)$ en parcourant $D(x_i)$ en ordre croissant ou décroissant de valeurs.
2. Pour une valeur v n'appartenant pas à $D(x_i)$, 0 (resp. n) est la valeur par défaut pour $\overline{p}(x_i,v)$ et $\overline{s}(x_i,v)$ (resp. $\underline{p}(x_i,v)$ et $\underline{s}(x_i,v)$).

Dans ce cadre, nous utilisons une structure de données pour gérer ces matrices creuses, pour lesquelles les accès en lecture et en écriture sont toujours réalisés en itérant sur les lignes d'une colonne donnée, selon un ordre croissant ou décroissant d'indices sur une colonne . La partie haute du tableau ci-dessous décrit les trois primitives sur ces matrices creuses avec accès ordonné aux colonnes, et fournit leur complexité en temps. La partie basse décrit les primitives utilisées pour accéder ou modifier le domaine d'une variable. Les primitives réduisant le domaine d'une variable x retournent vrai si $D(x) \neq \emptyset$ après l'operation et faux dans le cas contraire.

Algorithm 2: BUILD_SUFFIX($[x_0,\ldots,x_{n-1}], \underline{s}[][], \overline{s}[][]$).

```
1  Allocate mins, maxs;
2  ScanInit({s, s̄}, n − 1, ↓) ; v := max(x_{n−1}) ;
3  repeat
4      Set(s, n − 1, v, 1) ; Set(s̄, n − 1, v, 1) ; w := v ;
       v :=getPrev(x_{n−1}, v);
5  until w = v ;
6  for i := n − 2 downto 0 do
7      ScanInit({s, s̄, mins, maxs}, i + 1, ↓) ; v := max(x_{i+1}) ;
8      repeat
9          if v < max(x_{i+1}) then
10             Set(mins, i + 1, v, min(Get(mins, i + 1, v +
               1),Get(s, i + 1, v)));
11             Set(maxs, i + 1, v, max(Get(maxs, i + 1, v +
               1),Get(s̄, i + 1, v)));
12         else
13             Set(mins, i + 1, v,Get(s, i + 1, v));
14             Set(maxs, i + 1, v,Get(s̄, i + 1, v));
15         w := v ; v :=getPrev(x_{i+1}, v);
16     until w = v ;
17     ScanInit({s, s̄}, i, ↓) ;
       ScanInit({s, s̄, mins, maxs}, i + 1, ↓) ; v := max(x_i) ;
18     repeat
19         if v = max(x_{i+1}) then
20             Set(s, i, v,Get(s, i + 1, v)) ;
               Set(s̄, i, v,Get(s̄, i + 1, v));
21         else
22             if v ≥ min(x_{i+1}) then
23                 Set(s, i, v, min(Get(s, i +
                   1, v),Get(mins, i + 1, v + 1) + 1));
24                 Set(s̄, i, v, max(Get(s̄, i +
                   1, v),Get(maxs, i + 1, v + 1) + 1));
25             else
26                 Set(s, i, v,Get(mins, i + 1, min(x_{i+1})) + 1);
27                 Set(s̄, i, v,Get(maxs, i + 1, min(x_{i+1})) + 1);
28         w := v ; v :=getPrev(x_i, v);
29     until w = v ;
```

Primitives (*accès aux matrices*)	**Description**	**Complex.**
ScanInit(mat, i, dir)	indique que l'on itère sur la i-ème colonne de la matrice mat en ordre croissant d'indices ($dir = \uparrow$) ou décroissant ($dir = \downarrow$)	$O(1)$
Set($mat, i, j, info$)	réalise l'affectation $mat[i, j] := info$	$O(1)$
Get(mat, i, j)	retourne le contenu de l'entrée $mat[i, j]$ ou la valeur par défaut si cette entrée n'appartient pas à la matrice creuse (q appels consécutifs à Get sur la même colonne i et en ordre croissant ou décroissant des indices des lignes coûte $O(q)$).	amortie

Primitives (*accès aux variables*)	**Description**	**Complex.**
adjust_min(x, v) :bool	ajuste le minimum de la variable x à la valeur v	$O(1)$
adjust_max(x, v) :bool	ajuste le maximum de la variable x à la valeur v	$O(1)$
remove_val(x, v) :bool	supprime v de $D(x)$	$O(1)$
instantiate(x, v) :bool	instancie x à la valeur v	$O(1)$
get_prev(x, v) :bool,int	retourne vrai et la plus grande valeur w de $D(x)$ telle que $w < v$, ou retourne faux	$O(1)$
get_next(x, v) :bool,int	retourne vrai et la lus petite valeur w de $D(x)$ telle que $w > v$, ou retourne faux	$O(1)$

L'algorithme 3 correspond à l'algorithme de filtrage principal, qui implémente la Proposition 4. Dans une première phase il réduit les valeurs minimum et maximum des domaines des variables $[x_0, x_1, \ldots, x_{n-1}]$ en fonction des contraintes d'inégalité (*i.e.*, seules les valeurs bien ordonnées sont conservées). Dans une seconde phase, il calcule les informations relatives au nombre de stretchs minimum et maximum sur les matrices de préfixes et de suffixes $\underline{p}$, $\overline{p}$, $\underline{s}$, $\overline{s}$. Enfin, en fonction de ces informations, les bornes de N sont ajustées et l'algorithme filtre les domaines des variables $x_0, x_1, \ldots, x_{n-1}$. En utilisant les Lemmes 3 et 4, l'algorithme 2 construit les matrices de suffixes $\underline{s}$ et $\overline{s}$ utilisées par l'algorithme 3 ($\underline{p}$ et $\overline{p}$ sont construites de façon similaire) :

1. Dans une première phase, les colonnes $n-1$ des matrices $\underline{s}$ et $\overline{s}$ sont initialisées à 1 (premier item des Lemmes 3 et 4).
2. Dans une deuxième phase, les colonnes $n-2$ jusqu'à 0 sont initialisées (deuxième item des lemmes 3 et 4). Afin d'éviter de recalculer les quantités $\min(\underline{s}(x_{i+1}, v), \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1)$ et $\max(\overline{s}(x_{i+1}, v), \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1)$, nous introduisons deux matrices creuses avec accès ordonné aux colonnes $mins[i, j]$ et $maxs[i, j]$. Lorsques les i-èmes colonnes des matrices $\underline{s}$ et $\overline{s}$ sont initialisées, nous calculons d'abord les $i+1$-èmes colonnes des matrices *mins* et *maxs* (premier **repeat** de la boucle **for**). Ensuite, dans le second **repeat** de la boucle **for** nous initialisons les i-èmes colonnes de $\underline{s}$ et $\overline{s}$. On peut observer que l'on balaye les colonnes $i+1$ des matrices *mins* et *maxs* par indice de ligne décroissant.

L'algorithme 2 a une complexité temporelle en $O(\Sigma_{\text{Di}})$ et l'algorithme 3 supprime toutes les valeurs qui ne sont pas arc-consistantes avec Increasing_Nvalue en $O(\Sigma_{\text{Di}})$.[1]

Algorithm 3: Increasing_Nvalue($N, [x_0, \ldots, x_{n-1}]$) : boolean.

```
1  if n = 1 then return instantiate(N, 1);
2  for i = 1 to n − 1 do if ¬adjust_min(x_i, min(x_{i−1})) then
   return false;
3  for i = n − 2 downto 0 do if ¬adjust_max(x_i, max(x_{i+1}))
   then return false;
4  Allocate p, p̄, s, s̄;
5  build_prefix p, p̄ ; build_suffix s, s̄;
6  ScanInit({s, s̄}, 0, ↑);
7  if ¬adjust_min(N, min_{v∈D(x_0)}(Get(s, 0, v))) then return
   false;
8  if ¬adjust_max(N, max_{v∈D(x_0)}(Get(s̄, 0, v))) then return
   false;
9  for i := 0 to n − 1 do
10     ScanInit({p, p̄, s, s̄}, i, ↑) ; v := min(x_i);
11     repeat
12        N_v := Get(p, i, v)+Get(s, i, v) − 1 ;
          N̄_v := Get(p̄, i, v)+Get(s̄, i, v) − 1;
13        if [N_v, N̄_v] ∩ D(N) = ∅ and ¬remove_val(x_i, v)
          then return false;
14        w := v ; v := getNext(x_i, v);
15     until w = v ;
16 return true;
```

[1] Le code source de Increasing_Nvalue est disponible à l'adresse suivante : `http ://choco.emn.fr`.

5 Expérimentations

Cette section présente une série d'expérimentations de la contrainte Increasing_Nvalue. Tout d'abord, la section 5.1 présente une reformulation de la contrainte Nvalue utilisant Increasing_Nvalue, motivée par le fait de casser des symétries. Ensuite, la section 5.2 évalue les performances de Increasing_Nvalue sur une application réelle. Toutes ces expériences ont été réalisées à l'aide de la librairie de programmation par contraintes Choco, sur un processeur Intel Core 2 Duo 2.4GHz avec 4GB de RAM, et 128Mo alloués à la machine virtuelle Java.

5.1 Amélioration de la propagation de Nvalue

Assurer GAC pour une contrainte Nvalue est un problème NP-Difficile, et les algorithmes de filtrage de cette contrainte sont connus pour être assez peu efficaces pour des domaines de variables énumérés [3]. Dans notre implémentation, nous utilisons une représentation de Nvalue qui est basée sur les contraintes d'occurrence de Choco. Nous évaluons alors l'effet de la contrainte Increasing_Nvalue utilisée comme contrainte implicite sur des classes d'équivalence : Étant donné un ensemble $\mathcal{E}(X)$ de classes d'équivalence sur l'ensemble X, le filtrage de la contrainte Nvalue(X, N) peut être renforcé ainsi :

$$Nvalue(N, X) \quad (1)$$

$$\forall E \in \mathcal{E}(X), Increasing\_Nvalue(N_E, E) \quad (2)$$

$$\max_{E \in \mathcal{E}(X)}(N_E) \leq N \leq \sum_{E \in \mathcal{E}(X)}(N_E) \quad (3)$$

où N_E est la variable d'occurrence associée à la classe $E \in \mathcal{E}(X)$ (avec $E \subseteq X$).

Les paramètres enregistrés sont le nombre de nœuds dans l'arbre de recherche, le nombre d'échecs et le temps d'une résolution nécessaires à l'obtention d'une solution. Les paramètres variables de nos expériences sont le nombre de valeurs dans les domaines des variables, le pourcentage de trous dans ces domaines, et le nombre de classes d'équivalences. Le comportement observé n'est pas dépendant du nombre de variables : des tailles de 20, 40 et 100 variables ont été testées.

Les tableaux 1 et 2 montrent les résultats de nos expérimentations pour 40 variables, avec des domaines contenant au plus 80 valeurs (des tailles de 20 et 40 sont aussi testées). Concernant le tableau 1, 50 instances ont été générées pour chaque taille de classe d'équivalence. Concernant le tableau 2, 350 instances ont été générées pour chaque densité évaluée. Une limite de temps de résolution a été fixée à 60 secondes. Un paramètre est inclus dans la moyenne si les deux modèles résolvent l'instance. Les deux modèles sont strictement comparables si le pourcentage d'instances résolues est le même. Autrement, les paramètres enregistrés ne peuvent être comparés que pour les instances résolues par les deux modèles.

Le tableau 1 illustre le fait que le nombre de classes d'équivalence ait un impact sur les performances du modèle intégrant la contrainte Increasing_Nvalue. Avec sept classes, le nombre moyen de variables dans chaque classe est suffisant pour que le gain en propagation comble le coût de calcul supplémentaire dû à l'application de l'algorithme de filtrage de Increasing_Nvalue. Ce n'est plus le cas quand on a plus 10 classes d'équivalences.

La table 2 montre que le nombre de trous dans les domaines a un impact sur les performances du modèle incluant la contrainte Increasing_Nvalue. On peut remarquer que lorsque ce nombre de trous augmente, le nombre d'instances résolues décroît. Ce phénomène est dû à la propagation de la contrainte Nvalue, moins efficace quand il y a des trous dans les domaines.

5.2 Intégration dans un problème d'allocation de ressources

Entropy[2] [7] fournit un moteur autonome et flexible pour manipuler l'état et la position de VMs (machines virtuelles) sur les différents nœuds actifs composant une grappe. Ce moteur est basé sur la programmation par contraintes. Il fournit un modèle central dédié à l'affectation de VMs à des nœuds, et permet de personnaliser cette affectation à l'aide de contraintes qui modélisent des besoins exprimés par des utilisateurs et des administrateurs.

Le modèle central définit chaque nœud (les ressources) par son CPU et sa capacité mémoire, et chaque VM (les tâches) par sa demande en CPU et en mémoire pour s'exécuter. La partie traitée via un modèle en programmation par contraintes consiste à calculer une affectation de chaque VM qui (i) satisfait la demande en ressources (CPU et mémoire) des VMs et (ii) utilise un nombre minimum de nœuds. Au final, la libération de nœuds peut faire que davantage de tâches seront acceptées dans la grappe, ou bien peut permettre d'éteindre les nœuds non utilisés pour économiser l'énergie. Dans ce problème, deux parties peuvent être distinguées : (1) l'affectation de nœuds en fonction de la capacité de ressources : il s'agit d'un problème de bin-packing 2D. Il est modélisé par un ensemble de contraintes *sac à dos* associées avec chaque nœud. La propagation est basée sur l'utilisation de la contrainte CostRegular [6], afin de traiter simultanément les deux dimensions de ressource. (2) Ré-

[2] http ://entropy.gforge.inria.fr

Nb de	Nvalue				Nvalue + Increasing_Nvalue			
cl. d'quiv.	nœuds	échecs	temps (ms)	résolus(%)	nœuds	échecs	temps (ms)	résolus(%)
1	2798	22683	6206	76	**28**	**0**	**51**	**100**
3	1005	12743	4008	76	**716**	**7143**	**3905**	**82**
5	1230	14058	**8077**	72	**1194**	**12067**	8653	72
7	850	18127	**6228**	64	**803**	**16384**	6488	**66**
10	387	3924	**2027**	58	387	**3864**	2201	58
15	1236	16033	**6518**	38	**1235**	**16005**	7930	38
20	379	7296	**5879**	58	379	7296	6130	58

Tab. 1: Évaluation de Increasing_Nvalue en fonction du nombre de classes d'équivalence sur les variables.

Trous(%)	Nvalue				Nvalue + Increasing_Nvalue			
	nœuds	échecs	temps (ms)	résolus(%)	nœuds	échecs	temps (ms)	résolus(%)
25	1126.4	13552	5563.3	63.1	**677.4**	**8965.5**	**5051.1**	**67.7**
50	2867.1	16202.6	**4702.1**	50.8	**1956.4**	**12345**	4897.5	**54.9**
75	5103.7	16737.3	**3559.4**	**65.7**	**4698.7**	**15607.8**	4345.5	65.1

Tab. 2: Évaluation de Increasing_Nvalue en fonction du pourcentage de trous dans les domaines.

duction du nombre de nœuds utilisés pour instancier toutes les VMs. Les VMs sont classées en fonction de leur consommation en CPU et en mémoire (il existe donc naturellement des classes d'équivalence entre les VMs). Les contraintes Nvalue et Increasing_Nvalue (Section 5.1) sont utilisées pour modéliser cette partie du problème.

En pratique, les résultats obtenus par un modèle incluant des contraintes Increasing_Nvalue montrent un gain modéré en terme de temps de résolution (3%), alors que le gain en nombre d'échecs est plus significatif (35% en moyenne).

6 Travaux connexes

Un filtrage GAC pour Increasing_Nvalue peut être obtenu via deux autres techniques : l'usage d'automates déterministes avec un nombre polynomial de transitions [11], et la contrainte Slide [4].

Soit C une contrainte d'arité k et soit X une séquence de n variables. La contrainte Slide(C,X) [4] est un cas particulier de la contrainte *cardpath*. Cette contrainte est satisfaite ssi $C(X_i, X_{i+1}, \ldots, X_i + k - 1)$ est vérifiée pour tout $i \in [1, n - k + 1]$. La GAC peut être établie en temps $O(nd^k)$ où d est la taille maximale d'un domaine. Une extension appelée slide$_j$(C,X) est satisfaite $C(X_{ij+1}, X_{ij+2}, \ldots, X_{ij+k})$ est vérifiée pour tout $i \in [0, \frac{n-k}{j}]$. Étant donné $X = \{x_i \mid i \in [1; n]\}$, la contrainte Increasing_Nvalue peut être codée par Slide$_2$(C,$[x_i, c_i]_{i\in[1;n]}$) où (a) $c_1, c_2, \ldots, c_n$ sont des variables prenant leurs valeurs dans $[1, n]$ avec $c_1 = 1$ et $c_n = N$, et (b) $C(x_i, c_i, x_{i+1}, c_{i+1})$ est la contrainte $b \Leftrightarrow x_i \neq x_{i+1} \wedge c_{i+1} = c_i + b \wedge x_i \leq x_{i+1}$. La complexité en temps est alors en $O(nd^4)$ pour établir GAC sur la reformulation de Increasing_Nvalue.

La reformulation basée sur des automates fini déterministes est détaillée dans le catalogue de contraintes globales [1]. En utilisant l'algorithme de Pesant [11], la complexité obtenue est $O(n{\cup_{Di}}^3)$ pour GAC, où $\cup_{Di}$ est la taille de l'union des domaines des variables.

7 Conclusion

Nous avons proposé une technique pour réaliser un filtrage complet (GAC) pour une spécialisation de la contrainte Nvalue, où les variables de décision doivent satisfaire une séquence de contraintes binaires d'inégalité. Alors que prouver si une contrainte Nvalue admet ou non une solution est un problème NP-Difficile, notre algorithme a une complexité linéaire en la somme des tailles des domaines des variables. Nous pensons que la structure de données pour gérer ces matrices creuses avec accès ordonné aux colonnes peut être utile au delà du cadre de Increasing_Nvalue, afin d'améliorer la complexité d'algorithmes de filtrage d'autres contraintes globales. Nos travaux futurs se concentreront également sur une amélioration pratique de l'implémentation de l'algorithme de filtrage de Increasing_Nvalue, en considérant des intervalles de valeurs consécutives dans un domaine donné, plutôt que chaque valeur séparément. De façon plus générale, ce travail aborde le thème de l'intégration générique de méthodes pour casser des symétries dans les systèmes

à contraintes, via des contraintes globales [8, 12].

Références

[1] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog, working version of January 2010. Technical Report T2005-08, Swedish Institute of Computer Science, 2005. Available at `www.emn.fr/x-info/sdemasse/gccat`.

[2] N. Beldiceanu, M. Carlsson, and S. Thiel. Cost-Filtering Algorithms for the two Sides of the *sum of weights of distinct values* Constraint. Technical report, Swedish Institute of Computer Science, 2002.

[3] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. Filtering Algorithms for the *nvalue* Constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 79–93, 2005.

[4] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. SLIDE : A useful special case of the CARDPATH constraint. In *ECAI 2008, Proceedings*, pages 475–479, 2008.

[5] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The Complexity of Global Constraints. In *19th National Conference on Artificial Intelligence (AAAI'04)*, pages 112–117. AAAI Press, 2004.

[6] S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4) :315–333, 2006.

[7] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy : a consolidation manager for clusters. In *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 41–50, 2009.

[8] G. Katsirelos, N. Narodytska, and T. Walsh. Combining Symmetry Breaking and Global Constraints. In *Recent Advances in Constraints, Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2008*, volume 5655 of *LNCS*, pages 84–98, 2009.

[9] F. Pachet and P. Roy. Automatic Generation of Music Programs. In *CP'99*, volume 1713 of *LNCS*, pages 331–345, 1999.

[10] G. Pesant. A filtering algorithm for the stretch constraint. In *CP'01*, volume 2239 of *LNCS*, pages 183–195, 2001.

[11] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP'04*, volume 3258 of *LNCS*, pages 482–495, 2004.

[12] M. Ågren, N. Beldiceanu, M. Carlsson, M. Sbihi, C. Truchet, and S. Zampelli. Six Ways of Integrating Symmetries within Non-Overlapping Constraints. In *CP-AI-OR'09*, volume 5547 of *LNCS*, pages 11–25, 2009.