The Increasing Nvalue Constraint

Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca, and Thierry Petit

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

Abstract. This paper introduces the Increasing_Nvalue constraint, which restricts the number of distinct values assigned to a sequence of variables so that each variable in the sequence is less than or equal to its successor. This constraint is a specialization of the Nvalue constraint, motivated by symmetry breaking. Propagating the Nvalue constraint is known as an NP-hard problem. However, we show that the chain of non strict inequalities on the variables makes the problem polynomial. We propose an algorithm achieving generalized arc-consistency in $O(\Sigma_{\mathrm{Di}})$ time, where Σ_{Di} is the sum of domain sizes. This algorithm is an improvement of filtering algorithms obtained by the automaton-based or the Slide-based reformulations. We evaluate our constraint on a resource allocation problem.

1 Introduction

The Nvalue constraint was introduced by Pachet *et al.* in [10] to express a restriction on the number of distinct values assigned to a set of variables. Even if finding out whether a Nvalue constraint has a solution or not is NP-hard [6], a number of filtering algorithms were developed over the last years [4, 3]. Motivated by symmetry breaking, this paper considers the conjunction of an Nvalue constraint with a chain of non strict inequalities constraints, that we call Increasing_Nvalue. We come up with a filtering algorithm that achieves general arc-consistency (GAC) for Increasing_Nvalue in $O(\Sigma_{\mathrm{Di}})$ time, where Σ_{Di} is the sum of domain sizes. This algorithm is more efficient than those obtained by using generic approaches such as encoding Increasing_Nvalue as a finite deterministic automaton [12] or as a Slide constraint [5], which respectively require $O(n(\cup_{Di})^3)$ and $O(nd^4)$ time complexities for achieving GAC, where n denotes the number of variables, $\cup_{Di}$ is the total number of potential values in the domains, and d the maximum size of a domain. Part of its efficiency relies on a specific data structure, i.e. a *matrix of ordered sparse arrays*, which allows multiple ordered queries (*i.e.*, Set and Get) to the columns of a sparse matrix.

Experiments proposed in this paper are based on a real-life resource allocation problem related to the management of clusters. Entropy is a Virtual Machine (VM) manager for clusters [8], which provides an autonomous and flexible engine to manipulate the state and the position of VMs on the different working nodes composing the cluster. The constraint programming part affects the VMs (the tasks) on a reduced number of nodes (the resources) in the cluster. It uses a Nvalue constraint maintaining the number of nodes required to host all the VMs. However, in practice, the resources consumed are most often equivalents from one VM to another. This leads to a limited number of equivalence classes among the VMs, and Increasing_Nvalue is used for breaking such symmetries.

Section 2 recalls some definitions and formally introduces the Increasing_Nvalue constraint. Next, Section 3 describes a necessary and sufficient condition for the feasibility of the Increasing_Nvalue constraint. Section 4 presents an algorithm enforcing GAC for Increasing_Nvalue(N,X). Section 5 evaluates the impact of our method on a resource allocation problem. Finally, Section 6 describes generic approaches for reformulating Increasing_Nvalue that are less efficient.

2 Preliminaries

Given a sequence of variables X, the *domain* $\mathrm{D}(x)$ of a *variable* $x \in X$ is the finite set of integer values that can be assigned to variable x. $\mathcal{D}$ is the union of all domains in X. We use the notations $\min(x)$ for the minimum value of $D(x)$ and $\max(x)$ for the maximum value of $D(x)$. The sum of domains sizes over $\mathcal{D}$ is $\Sigma_{\mathrm{Di}} = \sum_{x_i \in X} |D(x_i)|$. $A[X]$ denotes an assignment of values to variables in X. Given $x \in X$, $A[x]$ is the value of x in $A[X]$. $A[X]$ is *valid* iff $\forall x_i \in X$, $A[x_i] \in \mathrm{D}(x_i)$. An *instantiation* $I[X]$ is a valid assignment of X. Given $x \in X$, $I[x]$ is the value of x in $I[X]$. A *constraint* $C(X)$, specifies the allowed combinations of values for a set of variables X. It defines a subset $\mathcal{R}_C(\mathcal{D})$ of the cartesian product of the domains $\Pi_{x_i \in X} \mathrm{D}(x_i)$. A *feasible instantiation* of $C(X)$ is an instantiation which is in $\mathcal{R}_C(\mathcal{D})$. If $I[X]$ is a feasible instantiation of $C(X)$ then $I[X]$ *satisfies* $C(X)$. W.l.o.g., we consider that X contains at least two variables. Given $X = [x_0, x_1, \ldots, x_{n-1}]$ and i, j two integers such that $0 \leq i < j \leq n-1$, $I[x_i, \ldots, x_j]$ is the projection of $I[X]$ on the sequence $[x_i, \ldots, x_j]$.

Definition 1. *The constraint* Increasing_Nvalue(N, X) *is defined by a variable* N *and a sequence of* n *variables* $X = [x_0, x_1, \ldots, x_{n-1}]$. *Given an instantiation of* $[N, x_0, x_1, \ldots, x_{n-1}]$, Increasing_Nvalue$(N, X)$ *is satisfied iff:*

1. *N is equal to the number of distinct values assigned to the variables in X.*
2. *$\forall i \in [0, n-2]$, $x_i \leq x_{i+1}$.*

3 Feasibility of the Increasing Nvalue Constraint

This section presents a necessary and sufficient condition for the feasibility of the Increasing_Nvalue constraint. We first show that the number of distinct values of any instantiation $I[X]$ such that $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$, is equal to the number of stretches in $I[X]$. A *stretch* [11] is defined as a maximum length sequence of consecutive variables assigned to the same value. For any variable $x \in X$ and any value $v \in D(x)$, we compute the minimum and maximum number of stretches among all possible instantiations $I[X]$ such that $I[x] = v$.

Next, given a variable $x \in X$, we provide the properties linking the natural ordering of values in $D(x)$ and the minimum and maximum number of stretches that can be obtained by assigning a value to x. From these properties, we prove that there exists an instantiation satisfying the constraint for any value of $\mathrm{D}(N)$ between the minimum $\underline{s}(X)$ and the maximum $\overline{s}(X)$ of possible numbers of stretches. This leads to the main result of this section: Increasing_Nvalue(N, X) is feasible iff $\mathrm{D}(N) \cap [\underline{s}(X), \overline{s}(X)] \neq \emptyset$ (Proposition 3 in Section 3.3).

3.1 Estimating the number of stretches

Any feasible instantiation $I[X]$ of Increasing_Nvalue(N, X) satisfies $I[x_i] \leq I[x_j]$ for all $i < j$. In the following, an instantiation $I[x_0, x_1, \dots, x_{n-1}]$ is said to be *well-ordered* iff for i and j s.t. $0 \leq i < j \leq n-1$, we have $I[x_i] \leq I[x_j]$. A value $v \in \mathrm{D}(x)$ is said to be *well-ordered* with respect to x iff it can be part of at least one well-ordered instantiation.

Lemma 1. *Let $I[X]$ be an instantiation. If $I[X]$ satisfies* Increasing_Nvalue(X, N) *then $I[X]$ is well-ordered.*

Proof. From Definition 1, if $I[X]$ satisfies the constraint then $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$. By transitivity of $\leq$, the Lemma holds. □

Definition 2 (stretch). *Let $I[x_0, x_1, \dots, x_{n-1}]$ be an instantiation. Given i and j such that $0 \leq i \leq j \leq n-1$, a* stretch *of $I[X]$ is a sequence of consecutive variables $[x_i, \dots, x_j]$ such that in $I[X]$: (1) $\forall k \in [i, j]$, $\forall \ell \in [i, j]$, $x_k = x_\ell$. (2) either $i = 0$ or $x_{i-1} \neq x_i$. (3) either $j = n-1$ or $x_j \neq x_{j+1}$.*

Lemma 2. *Given a well-ordered instantiation $I[X]$, the number of stretches in $I[X]$ is equal to the number of distinct values in $I[X]$.*

Proof. $I[X]$ is well-ordered then, for any i and j s.t. $0 \leq i < j \leq n-1$, we have $I[x_i] \leq I[x_j]$. Consequently, if x_i and x_j belong to two distinct stretches and $i < j$ then $I[x_i] < I[x_j]$. □

It is possible to evaluate for each value v in each domain $\mathrm{D}(x_i)$ the exact minimum and maximum number of stretches of well-ordered suffix instantiations $I[x_i, \dots, x_n]$ such that $I[x_i] = v$, and similarly for prefix instantiations. This evaluation is performed w.r.t. the domains of variables x_j such that $j > i$.

Notation 1 *Let $X = [x_0, x_1, \dots, x_{n-1}]$ be a sequence of variables and let v be a value of $\mathcal{D}$. The exact minimum number of stretches among all well-ordered instantiations $I[x_i, \dots, x_{n-1}]$ such that $I[x_i] = v$ is denoted by $\underline{s}(x_i, v)$. By convention, if $v \notin \mathrm{D}(x_i)$ then $\underline{s}(x_i, v) = +\infty$. Similarly, the exact minimum number of stretches among all well-ordered instantiations $I[x_0, \dots, x_i]$ such that $I[x_i] = v$ is denoted by $\underline{p}(x_i, v)$. By convention, if $v \notin \mathrm{D}(x_i)$ then $\underline{p}(x_i, v) = +\infty$.*

Lemma 3. *Let $X = [x_0, x_1, \dots, x_{n-1}]$ be a sequence of variables. $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\underline{s}(x_i, v)$ can be computed as follows:*

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

Proof. By induction. When $|X| = 1$ there is one stretch. Thus, if $i = n-1$, for any $v \in \mathrm{D}(x_i)$, we have $\underline{s}(x_i, v) = 1$. Consider now, a variable x_i, $i < n-1$, and a value $v \in \mathrm{D}(x_i)$. Instantiations s.t. $I[x_{i+1}] < v$ cannot be augmented with value v for x_i to form a well-ordered instantiation $I[x_i, \ldots, x_{n-1}]$. Thus, let $I[x_{i+1}, \ldots, x_{n-1}]$ be an instantiation s.t. $I[x_{i+1}] \geq v$, which minimizes the number of stretches in $[x_{i+1}, \ldots, x_{n-1}]$. Either $I[x_{i+1}] = v$ and $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$ since the first stretch of $I[x_{i+1}, \ldots, x_{n-1}]$ is extended when augmenting $I[x_{i+1}, \ldots, x_{n-1}]$ with value v for x_i, or $I[x_{i+1}] \neq v$ and $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, I[x_{i+1}]) + 1$ since value v creates a new stretch. By construction, instantiations of $[x_{i+1}, \ldots, x_{n-1}]$ that do not minimize the number of stretches cannot lead to a value $\underline{s}(x_i, v)$ strictly less than $\min(\underline{s}(x_{i+1}, w), w > v) + 1$, even if $I[x_{i+1}] = v$. □

Given a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, computing $\underline{p}(x_i, v)$ is symmetrical: If $i = 0$: $\underline{p}(x_i, v) = 1$. If $i > 0$: $\underline{p}(x_i, v) = \min(\ \underline{p}(x_{i-1}, v),\ \ \min_{w<v}(\underline{p}(x_{i-1}, w)) + 1\ )$.

Moreover, for a given variable x_i, we evaluate for each value v the exact maximum number of stretches that may appear among all well-ordered instantiations $I[x_i, \ldots, x_{n-1}]$ with $I[x_i] = v$, and similarly for prefix instantiations.

Notation 2 *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and let v be a value of $\mathcal{D}$. The exact maximum number of stretches among all well-ordered instantiations $I[x_i, \ldots, x_{n-1}]$ with $I[x_i] = v$ is denoted by $\overline{s}(x_i, v)$. By convention, if $v \notin \mathrm{D}(x_i)$ then $\overline{s}(x_i, v) = 0$. Similarly, the exact maximum number of stretches among all well-ordered instantiations $I[x_1, \ldots, x_i]$ with $I[x_i] = v$ is denoted by $\overline{p}(x_i, v)$. By convention, if $v \notin \mathrm{D}(x_i)$ then $\overline{p}(x_i, v) = 0$.*

Lemma 4. *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables. $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\overline{s}(x_i, v)$ can be computed as follows:*

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

Proof. Similar to Lemma 3. □

Given a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, computing $\overline{p}(x_i, v)$ is symmetrical: If $i = 0$: $\overline{p}(x_i, v) = 1$, If $i > 0$: $\overline{p}(x_i, v) = \max(\ \overline{p}(x_{i-1}, v),\ \ \max_{w<v}(\overline{s}(x_{i-1}, w)) + 1\ )$.

3.2 Properties on the number of stretches

This section enumerates the properties that link the natural ordering of values in a domain $D(x_i)$ with the minimum and maximum number of stretches that can be obtained in the sub-sequence $x_i, x_{i+1}, \ldots, x_{n-1}$. We consider only well-ordered values, which may be part of a feasible instantiation of Increasing_Nvalue.

Properties on a single value The next three properties are directly deduced, by construction, from Lemmas 3 and 4.

Property 1. Any value $v \in \mathrm{D}(x_i)$ well-ordered w.r.t. x_i is such that $\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$.

Property 2. Let $v \in \mathrm{D}(x_i)$ $(i < n-1)$ be a value well-ordered w.r.t. x_i. If $v \in \mathrm{D}(x_{i+1})$ and v is well-ordered w.r.t. x_{i+1} then $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$.

Property 3. Let $v \in \mathrm{D}(x_i)$ $(i < n-1)$ be a value well-ordered w.r.t. x_i. If $v \in \mathrm{D}(x_{i+1})$ and v is well-ordered w.r.t. x_{i+1} then $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$.

Proof. From Lemma 4, if there exists a value $w \in D(x_{i+1})$, $w > v$, which is well-ordered w.r.t. x_{i+1} and s.t. $\overline{s}(x_{i+1}, w) \geq \overline{s}(x_{i+1}, v)$ then $\overline{s}(x_i, v) > \overline{s}(x_{i+1}, v)$. Otherwise, $\overline{s}(x_i, v) = \overline{s}(x_{i+1}, v)$. □

Ordering on values The two following properties establish the links between the natural ordering of values in $D(x_i)$ and the minimum and maximum number of stretches in the sub-sequence starting from x_i.

Property 4. Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and let $i \in [0, n-1]$ be an integer. $\forall v, w \in \mathrm{D}(x_i)$ two well-ordered values, $v \leq w \Rightarrow \underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$.

Proof. If $v = w$ the property holds. If $i = n-1$, by Lemma 3, $\underline{s}(x_{n-1}, v) = \underline{s}(x_{n-1}, w) = 1$. The property holds. Given $i < n-1$, let v', w' be two well-ordered values of $\mathrm{D}(x_{i+1})$ such that $v' \geq v$ and $w' \geq w$, which minimize the number of stretches starting at x_{i+1}: $\forall \alpha \geq v$, $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, \alpha)$ and $\forall \beta \geq w$, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_{i+1}, \beta)$. Such values exist because v and w are well-ordered values. Then, by construction we have $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, w')$, and, from Lemma 3, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_i, w)$, which leads to $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_i, w)$. By Lemma 3, $\underline{s}(x_i, v) \leq \underline{s}(x_{i+1}, v') + 1$. Thus, $\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$. □

A symmetrical property holds on the maximum number of stretches.

Property 5. Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and $i \in [0, n-1]$ an integer. $\forall v, w \in \mathrm{D}(x_i)$ two well-ordered values, $v \leq w \Rightarrow \overline{s}(x_i, v) \geq \overline{s}(x_i, w)$.

Proof. If $v = w$ the property holds. If $i = n-1$, by Lemma 4, $\overline{s}(x_{n-1}, v) = \overline{s}(x_{n-1}, w) = 1$. The property holds. Given $i < n-1$, let $w' \in \mathrm{D}(x_{i+1})$ be well-ordered, s.t. $w' \geq w$, and maximizing the number of stretches starting at x_{i+1} ($\forall \beta \geq w$, $\overline{s}(x_{i+1}, w') \geq \overline{s}(x_{i+1}, \beta)$). By Lemma 4, $\overline{s}(x_i, w) \leq \overline{s}(x_{i+1}, w') + 1$. Since $v < w$ and thus $v < w'$, $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, w') + 1$. The property holds. □

Ordering on the maximum number of stretches The intuition of Property 6 stands from the fact that, the smaller a well-ordered value v w.r.t. a variable x_i is, the more stretches one can build on the sequence $[x_i, x_{i+1}, \ldots, x_{n-1}]$ with $x_i = v$.

Property 6. Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and let i be an integer in interval $[0, n-1]$. $\forall v, w \in \mathrm{D}(x_i)$ two well-ordered values, $\overline{s}(x_i, w) < \overline{s}(x_i, v) \Rightarrow v < w$.

Proof. We show that if $v \geq w$ then, we have a contradiction with $\overline{s}(x_i, w) < \overline{s}(x_i, v)$. If $i = n-1$, Lemma 4 ensures $\overline{s}(x_{n-1}, w) = \overline{s}(x_{n-1}, v) = 1$, a contradiction. Now, let us consider the case where $i < n-1$. If $v = w$ then $\overline{s}(x_i, w) = \overline{s}(x_i, v)$, a contradiction. Otherwise ($v > w$), let v' be a value of $\mathrm{D}(x_{i+1})$ such that $v' \geq v$ which maximizes $\overline{s}(x_{i+1}, \alpha)$, $\alpha \geq v$. Such a value exists because v is well-ordered. By construction $w < v'$. By Lemma 4, $\overline{s}(x_i, w) \geq \overline{s}(x_{i+1}, v') + 1$ (1). By construction we have also $v \leq v'$, which implies $\overline{s}(x_{i+1}, v') + 1 \geq \overline{s}(x_i, v)$ (2). From (1) and (2) we have $\overline{s}(x_i, w) \geq \overline{s}(x_i, v)$, a contradiction. $\square$

Ordering on the minimum number of stretches There is no implication from the minimum number of stretches to the ordering of values in domains. Let $X = [x_0, x_1, x_2]$ with $D(x_0)$=$D(x_1)$=$\{1, 2, 3\}$ and $D(x_2) = \{1, 2, 4\}$. $\underline{s}(x_0, 1) = 1$ and $\underline{s}(x_0, 3) = 2$, thus $\underline{s}(x_0, 1) < \underline{s}(x_0, 3)$ and $1 < 3$. Consider now that $D(x_2) = \{2, 3, 4\}$. $\underline{s}(x_0, 1) = 2$ and $\underline{s}(x_0, 3) = 1$, thus $\underline{s}(x_0, 3) < \underline{s}(x_0, 1)$ and $3 > 1$.

Summary Next table summarizes the relations between well-ordered values v and w in $D(x_i)$ and the estimations of the minimum and maximum number of stretches among all instantiations starting from these values (that is, $I[x_i, \ldots, x_{n-1}]$ such that $I[x_i] = v$ or such that $I[x_i] = w$).

Precondition	Property	Proposition
$v \in D(x_i)$ is well-ordered	$\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$	Prop. 1
$v \in D(x_i)$ is well-ordered, $i < n-1$ and $v \in D(x_{i+1})$	$\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$ $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$	Prop. 2 Prop. 3
$v \in D(x_i)$, $w \in D(x_i)$ are well-ordered and $v \leq w$	$\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$ $\overline{s}(x_i, v) \geq \overline{s}(x_i, w)$	Prop. 4 Prop. 5
$v \in D(x_i)$, $w \in D(x_i)$ are well-ordered and $\overline{s}(x_i, w) < \overline{s}(x_i, v)$	$v < w$	Prop. 6

3.3 Necessary and Sufficient Condition for Feasibility

Notation 3 *Given a sequence of variables* $X = [x_0, x_1, \ldots, x_{n-1}]$*,* $\underline{s}(X)$ *is the minimum value of* $\underline{s}(x_0, v)$*,* $v \in \mathrm{D}(x_0)$*, and* $\overline{s}(X)$ *is the maximum value of* $\overline{s}(x_0, v)$*,* $v \in \mathrm{D}(x_0)$*.*

Proposition 1. *Given an* Increasing_Nvalue(N, X) *constraint, if* $\underline{s}(X) > \max(\mathrm{D}(N))$ *then the constraint has no solution. Symmetrically, if* $\overline{s}(X) < \min(\mathrm{D}(N))$ *then the constraint has no solution.*

Proof. By construction from Lemmas 3 and 4. □

W.l.o.g., $\mathrm{D}(N)$ can be restricted to $[\underline{s}(X), \overline{s}(X)]$. However, observe that $\mathrm{D}(N)$ may have holes or may be strictly included in $[\underline{s}(X), \overline{s}(X)]$. We prove that for any value k in $[\underline{s}(X), \overline{s}(X)]$ there exists a value $v \in \mathrm{D}(x_0)$ such that $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$. Thus, any value in $D(N) \cap [\underline{s}(X), \overline{s}(X)]$ is feasible.

Proposition 2. *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables. For any integer k in $[\underline{s}(X), \overline{s}(X)]$ there exists v in $\mathrm{D}(x_0)$ such that $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$.*

Proof. Let $k \in [\underline{s}(X), \overline{s}(X)]$. If $\exists v \in \mathrm{D}(x_0)$ s.t. $k = \underline{s}(x_0, v)$ or $k = \overline{s}(x_0, v)$ the property holds. Assume $\forall v \in \mathrm{D}(x_0)$, either $k > \overline{s}(x_0, v)$ or $k < \underline{s}(x_0, v)$. Let v', w' be the two values such that v' is the maximum value of $\mathrm{D}(x_0)$ such that $\overline{s}(x_0, v') < k$ and w' is the minimum value such that $k < \underline{s}(x_0, w')$. Then, we have $\overline{s}(x_0, v') < k < \underline{s}(x_0, w')$ (1). By Property 1, $\underline{s}(x_0, w') \leq \overline{s}(x_0, w')$. By Property 6, $\overline{s}(x_0, v') < \overline{s}(x_0, w') \Rightarrow w_0 < v'$. By Properties 4 and 1, $w_0 < v' \Rightarrow \underline{s}(x_0, w') \leq \overline{s}(x_0, v') + 1$, a contradiction with (1). □

Algorithm 1: Building a solution for INCREASING_NVALUE(k, X).

```
1 if $k \notin [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$ then return "no solution" ;
2 $v$ := a value $\in \mathrm{D}(x_0)$ s.t. $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$ ;
3 for $i := 0$ to $n-2$ do
4     $I[x_i] := v$;
5     if $\forall v_{i+1} \in \mathrm{D}(x_{i+1})$ s.t. $v_{i+1} = v$, $k \notin [\underline{s}(x_{i+1}, v_{i+1}), \overline{s}(x_{i+1}, v_{i+1})]$ then
6         $v := v_{i+1}$ in $\mathrm{D}(x_{i+1})$ s.t. $v_{i+1} > v \wedge k-1 \in [\underline{s}(x_{i+1}, v_{i+1}), \overline{s}(x_{i+1}, v_{i+1})]$;
7         $k := k-1$ ;
8 $I[x_{n-1}] := v$; return $I[X]$;
```

Lemma 5. *Given an* INCREASING_NVALUE(N, X) *constraint and an integer k, if $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$ then* Algorithm 1 *returns a solution of* INCREASING_NVALUE(N, X) *with $N = k$. Otherwise,* Algorithm 1 *returns "no solution" since no solution exists with $N = k$.*

Proof. The first line of Algorithm 1 ensures that either $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$ and k belongs to $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$, or there is no solution (from Propositions 1 and 2). At each new iteration of the **for** loop, by Lemmas 3 and 4 and Proposition 2, either the condition (line 6) is satisfied and a new stretch begins at $i+1$ with a greater value (which guarantees that $I[\{x_1, \ldots, x_{i+1}\}]$ is well-ordered) and k is decreased by 1, or it is possible to keep the current value v for $I[x_{i+1}]$. Therefore, at the start of a **for** loop (line 4), $\exists v \in \mathrm{D}(x_i)$ s.t. $k \in [\underline{s}(x_i, v), \overline{s}(x_i, v)]$. When $i = n-1$, by construction $k = 1$ and $\forall v_{n-1} \in \mathrm{D}(x_{n-1})$, $\underline{s}(x_{n-1}, v_{n-1}) = \overline{s}(x_{n-1}, v_{n-1}) = 1$; $I[X]$ is well-ordered and contains k stretches. From Lemma 2, instantiation $I[\{N\} \cup X]$ with $I[N] = k$ is a solution of INCREASING_NVALUE(N, X) with k distinct values in X. □

Lemma 5 leads to a necessary and sufficient feasibility condition.

Proposition 3. *Given an* INCREASING_NVALUE(N, X) *constraint, the two following propositions are equivalent:*

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

Proof. $(\Rightarrow)$ Assume INCREASING_NVALUE(N, X) has a solution. Let $I[\{N\} \cup X]$ be such a solution. By Lemma 2 the value k assigned to N is the number of stretches in $I[X]$. By construction (Lemmas 3 and 4) $k \in [\underline{s}(X), \overline{s}(X)]$. Thus, $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. $(\Leftarrow)$ Let $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. From Lemma 5 it is possible to build a feasible solution for INCREASING_NVALUE(N, X). $\square$

4 GAC Filtering Algorithm for Increasing Nvalue

This section presents an algorithm enforcing GAC for INCREASING_NVALUE(N, X) in $O(\Sigma_{\mathrm{Di}})$ time complexity, where Σ_{Di} is the sum of domain sizes of the variables in X. For a given variable $x_i \in X$ and a value $v \in D(x_i)$, the principle is to estimate the minimum and maximum number of stretches among all instantiations $I[X]$ with $I[x_i] = v$, to compare the interval derived from these two bounds and $D(N)$. In order to do so, w.l.o.g. we estimate the minimum and maximum number of stretches related to prefix instantiations $I[x_0, \ldots, x_i]$ and suffix instantiations $I[x_i, \ldots, x_{n-1}]$.

Definition 3 (GAC). *Let $C(X)$ be a constraint. A* **support** *on $C(X)$ is an instantiation $I[X]$ which satisfies $C(X)$. A domain* $\mathrm{D}(x)$ *is* **arc-consistent** *w.r.t. $C(X)$ iff $\forall v \in \mathrm{D}(x)$, v belongs to a support on $C(X)$. $C(X)$ is* **(generalized) arc-consistent** *(GAC) iff $\forall x_i \in X$, $\mathrm{D}(x_i)$ is arc-consistent.*

4.1 Necessary and Sufficient Condition for Filtering

From Lemma 5, values of $D(N)$ which are not in $[\underline{s}(X), \overline{s}(X)]$ can be removed from $D(N)$. By Proposition 3, all remaining values in $D(N)$ are feasible. We now give a necessary and sufficient condition to remove a value from $D(x_i)$, $x_i \in X$.

Proposition 4. *Consider an* INCREASING_NVALUE(N, X) *constraint. Let $i \in [0, n-1]$ be an integer and v a value in $D(x_i)$. The two following propositions are equivalent:*

1. *$v \in D(x_i)$ is arc-consistent w.r.t.* INCREASING_NVALUE
2. *v is well-ordered w.r.t. $D(x_i)$ and* $[\underline{p}(x_i, v) + \underline{s}(x_i, v) - 1, \overline{p}(x_i, v) + \overline{s}(x_i, v) - 1] \cap \mathrm{D}(N) \neq \emptyset$.

Proof. If v is not well-ordered then from Lemma 1, v is not arc-consistent w.r.t. INCREASING_NVALUE. Otherwise, $\underline{p}(x_i, v)$ is the exact minimum number of stretches among well-ordered instantiations $I[x_0, \ldots, x_i]$ such that $I[x_i] = v$ and $\underline{s}(x_i, v)$ is the exact minimum number of stretches among well-ordered instantiations $I[x_i, \ldots, x_{n-1}]$ such that $I[x_i] = v$. Thus, by construction $\underline{p}(x_i, v) +$

$\underline{s}(x_i, v) - 1$ is the exact minimum number of stretches among well-ordered instantiations $I[x_0, x_1, \ldots, x_{n-1}]$ such that $I[x_i] = v$. Let $\mathcal{D}_v \subseteq \mathcal{D}$ be the set of domains such that all domains in $\mathcal{D}_v$ are equal to domains in $\mathcal{D}$ except $D(x_i)$ which is reduced to $\{v\}$. We call X_v the set of variables associated with domains in $\mathcal{D}_v$. From Definition 3, $\underline{p}(x_i, v) + \underline{s}(x_i, v) - 1 = \underline{s}(X_v)$. By a symmetrical reasoning, $\overline{p}(x_i, v) + \overline{s}(x_i, v) - 1 = \overline{s}(X_v)$. By Proposition 3, the proposition holds. $\square$

4.2 Algorithms

From Proposition 4, we derive a filtering algorithm achieving GAC in $O(\Sigma_{\mathrm{Di}})$. For a given variable x_i ($0 \leq i < n$), we need to compute the prefix and suffix information $\underline{p}(x_i, v)$, $\overline{p}(x_i, v)$, $\underline{s}(x_i, v)$ and $\overline{s}(x_i, v)$, no matter whether value v belongs or not to the domain of x_i. To reach an overall complexity of $O(\Sigma_{\mathrm{Di}})$, we take advantage of two facts:

1. Within our algorithm we always iterate over $\underline{p}(x_i, v)$, $\overline{p}(x_i, v)$, $\underline{s}(x_i, v)$ and $\overline{s}(x_i, v)$ by scanning the value of $D(x_i)$ in increasing or decreasing order.
2. For a value v that does not belong to $D(x_i)$, 0 (resp. n) is the default value for $\overline{p}(x_i, v)$ and $\overline{s}(x_i, v)$ (resp. $\underline{p}(x_i, v)$ and $\underline{s}(x_i, v)$).

For this purpose we create a data structure for handling such sparse matrices for which write and read accesses are always done by iterating in increasing or decreasing order through the rows in a given column. The upper part of next table describes the three primitives on *ordered sparse matrices* as well as their time complexity. The lower part gives the primitives used for accessing or modifying the domain of a variable. Primitives which restrict the domain of a variable x return true if $D(x) \neq \emptyset$ after the operation, false otherwise.

Primitives (*access to matrices*)	**Description**	**Complexity**
SCANINIT(*mats*, i, *dir*)	indicates that we will iterate through the i^{th} column of matrices in *mats* in increasing order (*dir* $=\uparrow$) or decreasing order (*dir* $=\downarrow$)	$O(1)$
SET(*mat*, i, j, *info*)	performs the assignment *mat*$[i, j]$:= *info*	$O(1)$
GET(*mat*, i, j):int	returns the content of entry *mat*$[i, j]$ or the default value if such entry does not belong to the sparse matrix (a set of q consecutive calls to GET on the same column i and in increasing or decreasing row indexes is in $O(q)$)	amortized

Primitives (*access to variables*)	**Description**	**Complexity**
ADJUST_MIN(x, v):boolean	adjusts the minimum of var. x to value v	$O(1)$
ADJUST_MAX(x, v):boolean	adjusts the maximum of var. x to value v	$O(1)$
REMOVE_VAL(x, v):boolean	removes value v from domain $D(x)$	$O(1)$
INSTANTIATE(x, v):boolean	fix variable x to value v	$O(1)$
GET_PREV(x, v):int	returns the largest value w in $D(x)$ such that $w < v$ if it exists, returns v otherwise	$O(1)$
GET_NEXT(x, v):int	returns the smallest value w in $D(x)$ such that $w > v$ if it exists, returns v otherwise	$O(1)$

Algorithm 2: BUILD_SUFFIX($[x_0, x_1, \ldots, x_{n-1}], \underline{s}[][], \overline{s}[][]$).

```
1  ALLOCATE $mins, maxs$;
2  SCANINIT($\{\underline{s}, \overline{s}\}, n-1, \downarrow$); $v := \max(x_{n-1})$;
3  repeat
4      SET($\underline{s}, n-1, v, 1$); SET($\overline{s}, n-1, v, 1$); $w := v$; $v :=$getPREV($x_{n-1}, v$);
5  until $w = v$ ;
6  for $i := n-2$ downto $0$ do
7      SCANINIT($\{\underline{s}, \overline{s}, mins, maxs\}, i+1, \downarrow$); $v := \max(x_{i+1})$;
8      repeat
9          if $v < \max(x_{i+1})$ then
10             SET($mins, i+1, v, \min($GET($mins, i+1, v+1$),GET($\underline{s}, i+1, v$)));
11             SET($maxs, i+1, v, \max($GET($maxs, i+1, v+1$),GET($\overline{s}, i+1, v$)));
12         else
13             SET($mins, i+1, v$,GET($\underline{s}, i+1, v$));
14             SET($maxs, i+1, v$,GET($\overline{s}, i+1, v$));
15         $w := v$; $v :=$getPREV($x_{i+1}, v$);
16     until $w = v$ ;
17     SCANINIT($\{\underline{s}, \overline{s}\}, i, \downarrow$); SCANINIT($\{\underline{s}, \overline{s}, mins, maxs\}, i+1, \downarrow$); $v := \max(x_i)$;
18     repeat
19         if $v = \max(x_{i+1})$ then
20             SET($\underline{s}, i, v$,GET($\underline{s}, i+1, v$)); SET($\overline{s}, i, v$,GET($\overline{s}, i+1, v$));
21         else
22             if $v \geq \min(x_{i+1})$ then
23                 SET($\underline{s}, i, v, \min($GET($\underline{s}, i+1, v$),GET($mins, i+1, v+1) + 1$));
24                 SET($\overline{s}, i, v, \max($GET($\overline{s}, i+1, v$),GET($maxs, i+1, v+1) + 1$));
25             else
26                 SET($\underline{s}, i, v$,GET($mins, i+1, \min(x_{i+1})) + 1$);
27                 SET($\overline{s}, i, v$,GET($maxs, i+1, \min(x_{i+1})) + 1$);
28         $w := v$; $v :=$getPREV($x_i, v$);
29     until $w = v$ ;
```

Algorithm 3 corresponds to the main filtering algorithm that implements Proposition 4. In a first phase it restricts the minimum and maximum values of variables $[x_0, x_1, \ldots, x_{n-1}]$ w.r.t. to all the inequalities constraints (i.e. it only keeps well-ordered values). In a second step, it computes the information related to the minimum and maximum number of stretches on the prefix and suffix matrices $\underline{p}$, $\overline{p}$, $\underline{s}$, $\overline{s}$. Finally, based on this information, it adjusts the bounds of N and does the necessary pruning on each variable $x_0, x_1, \ldots, x_{n-1}$. Using Lemmas 3 and 4, Algorithm 2 builds the suffix matrices $\underline{s}$ and $\overline{s}$ used in Algorithm 3 ($\underline{p}$ and $\overline{p}$ are constructed in a similar way):

1. In a first step, column $n-1$ of matrices $\underline{s}$ and $\overline{s}$ are initialised to 1 (i.e. see the first item of Lemmas 3 and 4).
2. In a second step, columns $n-2$ down to 0 are initialised (i.e. see the second item of Lemmas 3 and 4). In order to avoid recomputing from scratch the quantities $\min(\underline{s}(x_{i+1}, v), \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1)$ and $\max(\overline{s}(x_{i+1}, v), \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1)$ we introduce two sparse ordered matrices $mins[i,j]$ and $maxs[i,j]$. When initialising the i^{th} columns of matrices $\underline{s}$ and $\overline{s}$ we first compute the $i+1^{th}$ columns of matrices $mins$ and $maxs$ (i.e. see the first **repeat** of the **for** loop). Then, in the second **repeat** of the **for** loop we initialise the i^{th} columns of $\underline{s}$ and $\overline{s}$. Observe that we scan columns $i+1$ of matrices $mins$ and $maxs$ in decreasing rows indices.

Consequently, Algorithm 2 takes $O(\Sigma_{\mathrm{Di}})$ time and Algorithm 3 prunes all the values that are not arc-consistent in INCREASING_NVALUE in $O(\Sigma_{\mathrm{Di}})$.[1]

Algorithm 3: INCREASING_NVALUE($N, [x_0, x_1, \ldots, x_{n-1}]$) : boolean.

```
1  if $n = 1$ then return INSTANTIATE($N, 1$);
2  for $i = 1$ to $n - 1$ do if $\neg$ADJUST_MIN($x_i, \min(x_{i-1})$) then return false;
3  for $i = n - 2$ downto 0 do if $\neg$ADJUST_MAX($x_i, \max(x_{i+1})$) then return false;
4  ALLOCATE $\underline{p}, \overline{p}, \underline{s}, \overline{s}$;
5  BUILD_PREFIX $\underline{p}, \overline{p}$; BUILD_SUFFIX $\underline{s}, \overline{s}$;
6  SCANINIT($\{\underline{s}, \overline{s}\}, 0, \uparrow$);
7  if $\neg$ADJUST_MIN($N, \min_{v \in D(x_0)}$(GET($\underline{s}, 0, v$))) then return false;
8  if $\neg$ADJUST_MAX($N, \max_{v \in D(x_0)}$(GET($\overline{s}, 0, v$))) then return false;
9  for $i := 0$ to $n - 1$ do
10     SCANINIT($\{\underline{p}, \overline{p}, \underline{s}, \overline{s}\}, i, \uparrow$); $v := \min(x_i)$;
11     repeat
12         $\underline{N}_v$ :=GET($\underline{p}, i, v$)+GET($\underline{s}, i, v$) $- 1$; $\overline{N}_v$ :=GET($\overline{p}, i, v$)+GET($\overline{s}, i, v$) $- 1$;
13         if $[\underline{N}_v, \overline{N}_v] \cap D(N) = \emptyset$ and $\neg$REMOVE_VAL($x_i, v$) then return false;
14         $w := v$; $v :=$ GETNEXT($x_i, v$);
15     until $w = v$ ;
16 return true ;
```

[1] The source code of the INCREASING_NVALUE constraint is available at `http://choco.emn.fr`.

5 Using Increasing Nvalue for Symmetry Breaking

This section provides a set of experiments for the INCREASING_NVALUE constraint. First, Section 5.1 presents a constraint programming reformulation of a NVALUE constraint into a INCREASING_NVALUE constraint to deal with symmetry breaking. Next, Section 5.2 evaluates the INCREASING_NVALUE on a real life application based on constraint programming technology. In the following, all experiments were performed with the Choco constraint programming system [1], on an Intel Core 2 Duo 2.4GHz with 4GB of RAM, and 128Mo allocated to the Java Virtual Machine.

5.1 Improving NVALUE Constraint Propagation

Enforcing GAC for a NVALUE constraint is a NP-Hard problem and existing filtering algorithms perform little propagation when domains of variables are sparse [3, 4]. In our implementation, we use a representation of NVALUE which is based on occurrence constraints of Choco. We evaluate the effect of the INCREASING_NVALUE constraint when it is used as an implied constraint on equivalence classes, in addition to the NVALUE. Thus, given a set $\mathcal{E}(X)$ of equivalence classes among the variables in X, the pruning of the global constraint NVALUE(X, N) can be strengthened in the following way:

$$Nvalue(N, X) \tag{1}$$

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

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

where N_E denotes the occurrence variable associated to the set of equivalent variables $E \in \mathcal{E}(X)$ and $E \subseteq X$.

Parameters recorded are the number of nodes in the tree search, the number of fails detected during the search and the solving time to reach a solution. Variables of our experiments are the maximum number of values in the variable domains, the percentage of holes in the variable domains and the number of equivalence classes among the variables. The behavior of our experiments is not related to the number of variables: sizes 20, 40 and 100 have been evaluated.

Tables 1 and 2 report the results of experiments for 40 variables and domains containing at most 80 values (size 20 and 40 are also tested). For Table 1, 50 instances are generated for each size of equivalence classes. For Table 2, 350 instances are generated for each density evaluated. A timeout on the solving time to a solution is fixed to 60 seconds. A recorded parameter is included in the average iff both approaches solve the instance. Then, two approaches are strictly comparable if the percentage of solved instances is equal. Otherwise, the recorded parameters can be compared for the instances solved by both approaches.

Table 1 illustrates that equivalence classes among the variables impact the performances of the INCREASING_NVALUE constraint model. We observe that the

Number of	NVALUE model				INCREASING_NVALUE model			
equivalences	nodes	failures	time(ms)	solved(%)	nodes	failures	time(ms)	solved(%)
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

Table 1: Evaluation of the INCREASING_NVALUE constraint according the number of equivalence classes among the variables.

holes(%)	NVALUE model				INCREASING_NVALUE model			
	nodes	failures	time(ms)	solved(%)	nodes	failures	time(ms)	solved(%)
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

Table 2: Evaluation of the INCREASING_NVALUE constraint according the percentage of holes in the domains.

performances (particularly the solving time) are impacted by the number of equivalence classes. From one equivalence class to 7, the average number of variables involved in each equivalence class is sufficient to justify the solving time overhead which is balanced by the propagation efficiency. From 10 to 20, the size of each equivalence class is not significant (in the mean, from 4 to 2 variables involved in each INCREASING_NVALUE constraint). Thus, we show that the propagation gain (in term of nodes and failures) is not significant while the solving time overhead could be important.

Unsurprisingly, Table 2 shows that the number of holes in the variable domains impact the performances of the INCREASING_NVALUE constraint model. However, we notice when the number of holes in the domains increases the number of solved instances decreases. Such a phenomenon are directly related with the fact that propagation of NVALUE is less efficient when there exist holes in the variable domains.

5.2 Integration in a Resource Scheduling Problem

Entropy[2] [8] provides an autonomous and flexible engine to manipulate the state and the position of VMs (hosting applications) on the different working nodes composing the cluster. This engine is based on Constraint Programing. It provides a core model dedicated to the assignment of VMs to nodes and some dedicated constraints to customize the assignment of the VMs regarding to some users and administrators requirements.

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

The core model denotes each node (the resources) by its CPU and memory capacity and each VM (the tasks) by its CPU and memory demands to run at a peak level. The constraint programming part aims at computing an assignment of each VM that (i) satisfies the resources demand (CPU and memory) of the VMs, and (ii) uses a minimum number of nodes. Finally, liberating nodes can allow more jobs to be accepted into the cluster, or can allow powering down unused nodes to save energy. In this problem two parts can be distinguished: (i) VMs assignment on nodes w.r.t. resource capacity: this is a bidimensional bin-packing problem. It is modeled by a set of knapsack constraints associated with each node. Propagation algorithm is based on CostRegular propagator [7] to deal with the two dimensions of the resource; (ii) Restriction on the number of nodes used to assign all the VMs. VMs are ranked according to their CPU and memory consumption (this means there is equivalence classes among the VMs). Nvalue and Increasing_Nvalue are used (Section 5.1) to model this part.

In practice, the results obtained by the Increasing_Nvalue constraint evaluation, within the constraint programming module of Entropy, point out a short gain in term of solving time (3%), while the gain in term of nodes and failures is more significant (in the mean 35%). Such a gap is due to the tradeoff between the propagation gain (filtered values) and solving time induced by the algorithm.

6 Related Work

GAC for the Increasing_Nvalue constraint can be also obtained by at least two different generic techniques, namely by using a finite deterministic automaton with a polynomial number of transitions or by using the Slide constraint.

Given a constraint C of arity k and a sequence X of n variables, the Slide(C,X) constraint [5] is a special case of the *cardpath* constraint. The *slide* constraint holds iff $C(X_i, X_{i+1}, \ldots, X_i + k - 1)$ holds for all $i \in [1, n - k + 1]$. The main result is that GAC can be enforced in $O(nd^k)$ time where d is the maximum domain size. An extension called $slide_j(C,X)$ holds iff $C(X_{ij+1}, X_{ij+2}, \ldots, X_{ij+k})$ holds for all $i \in [0, \frac{n-k}{j}]$. Given $X = \{x_i \mid i \in [1;n]\}$, the Increasing_Nvalue constraint can be encoded as $\text{Slide}_2(C,[x_i, c_i]_{i\in[1;n]})$ where (a) $c_1, c_2, \ldots, c_n$ are variables taking their value within $[1, n]$ with $c_1 = 1$ and $c_n = N$, and (b) $C(x_i, c_i, x_{i+1}, c_{i+1})$ is the constraint $b \Leftrightarrow x_i \neq x_{i+1} \wedge c_{i+1} = c_i + b \wedge x_i \leq x_{i+1}$. This leads to a time bound of $O(nd^4)$ for achieving GAC on the Increasing_Nvalue constraint.

The reformulation based on finite deterministic automaton is detailed in the global constraint catalog[2]. If we use Pesant's algorithm [12], this reformulation leads to a worst-case time complexity of $O(n{\cup_{Di}}^3)$ for achieving GAC, where $\cup_{Di}$ denotes the total number of potential values in the variable domains.

7 Conclusion

Motivated by symmetry breaking, we provide a filtering technique that achieves GAC for a specialized case of the Nvalue constraint where the decision variables are constrained by a chain of non strict inequalities. While finding out whether a Nvalue constraint has a solution or not is NP-hard, our algorithm has a linear time complexity w.r.t. the sum of the domain sizes. We believe that the data structure on matrices of ordered sparse arrays may be useful for decreasing the time worst-case complexity of other filtering algorithms.

Future work may also improve the practical speed of the Increasing_Nvalue constraint by somehow merging consecutive values in the domain of a variable. More important, this work follows the topic of integrating common symmetry breaking constraints directly within core global constraints [9, 13].

References

1. Choco: An open source Java CP library, documentation manual. `http://choco.emn.fr/`, 2009.
2. 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`.
3. 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.
4. 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.
5. 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.
6. 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.
7. S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006.
8. 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.
9. 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.
10. F. Pachet and P. Roy. Automatic Generation of Music Programs. In *CP'99*, volume 1713 of *LNCS*, pages 331–345, 1999.
11. G. Pesant. A filtering algorithm for the stretch constraint. In *CP'01*, volume 2239 of *LNCS*, pages 183–195, 2001.
12. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP'04*, volume 3258 of *LNCS*, pages 482–495, 2004.
13. 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.