A GAC Algorithm for a Class of Global Counting Constraints

Nicolas Beldiceanu, Xavier Lorca, and Thierry Petit

Mines-Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
TR 10/1/INFO, École des Mines de Nantes, 2010.
{Nicolas.Beldiceanu,Xavier.Lorca,Thierry.Petit}@emn.fr

Abstract. This paper presents the constraint class SEQ_BIN(N, X, C, B) where N is an integer variable, X is a sequence of integer variables and C and B are two binary constraints. A constraint of the SEQ_BIN class enforces the two following conditions: (1) N is equal to the number of times that the constraint C is satisfied on two consecutive variables in X, and (2) B holds on any pair of consecutive variables in X. Providing that B satisfies the particular property of neighborhood-substitutability, we come up with a filtering algorithm that achieves generalized arc consistency (GAC) for SEQ_BIN(N, X, C, B). This algorithm can be directly used for the constraints CHANGE, SMOOTH, INCREASING_NVALUE, AMONG and INCREASING_AMONG, in time linear in the sum of domain sizes. For all these constraints, this time complexity either improves the best known results, or equals those results.

1 Introduction

Some constraints are such that a *counting* variable is equal to the number of times a given property is satisfied in a sequence of variables. Most of them can be reformulated thanks to the SEQ_BIN(N, X, C, B) constraint class, where N is an integer variable, X is a sequence of integer variables and, C and B are two binary constraints. A constraint of the SEQ_BIN class holds if and only if the two following conditions are both satisfied: (1) N is equal to the number of times that the constraint C is satisfied on two consecutive variables in the sequence X, and (2) B holds on any pair of consecutive variables in X.

Constraints that can be expressed with SEQ_BIN are, for instance, the constraint AMONG [3, 7], the constraints CHANGE [9] and SMOOTH [2], which were introduced in the context of time tabling problems, and the constraint INCREASING_NVALUE [4], which was introduced in the context of resource allocation problems. We also consider a new constraint INCREASING_AMONG, which is a specialization of AMONG for breaking variable symmetry.

In practice, we show that all these constraints correspond to an instance of SEQ_BIN where the binary constraint B satisfies the particular property of *neighborhood substitutability*. In this context, the main contribution of the paper is a generic filtering algorithm that enforces GAC in $O(\Sigma_{\mathrm{Di}})$, where Σ_{Di} is the

sum of domain sizes. This algorithm is based on a generalization of the definition of *stretches* introduced by Pesant [13]. In each case, the time complexity improves or equals the best known results of the literature. For instance, thanks to SEQ_BIN, GAC can be enforced in $O(\Sigma_{\text{Di}})$ for all the versions of CHANGE and SMOOTH, while existing algorithms either have a higher complexity or do not enforce GAC. W.r.t. INCREASING_NVALUE, the algorithm presented in [4] is a particular case of our algorithm where we consider the classical notion of stretches.

Section 2 recalls basic definitions used in this paper. Section 3 defines the class SEQ_BIN and shows how to express well-known constraints and combinations of constraints with SEQ_BIN, namely CHANGE, SMOOTH, INCREASING_NVALUE, AMONG and INCREASING_AMONG. Section 4 provides the main contribution, a necessary and sufficient condition for achieving GAC. First, Section 4.1 evaluates the minimum (resp. maximum) number of time the constraint C is satisfied. Next Sections 4.2 and 4.3 highlight the fundamental properties leading to the GAC proof. Then, Sections 4.4 and 4.5 respectively provide two propositions for checking feasibility and for achieving GAC for SEQ_BIN. Section 5 details the corresponding GAC filtering algorithm. Section 6 presents implementation details w.r.t. the constraints mentioned in Section 3. Finally, Section 7 concludes.

2 Background

A *Constraint Network* $\mathcal{N}$ is defined by a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$, a sequence of domains $\mathcal{D} = D(x_0), D(x_1), \ldots, D(x_{n-1})$, where each domain $D(x_i)$ corresponds to the finite set of values that variable x_i can take, and a set of constraints $\mathcal{C}$ that specifies the allowed combinations of values for given subsets of variables. We use the notations $\min(x)$ for the minimum value of $D(x)$, $\max(x)$ for the maximum value of $D(x)$, and d for $\max_{x_i \in X}(|D(x_i)|)$.The sum of domains sizes over $\mathcal{D}$ is $\Sigma_{\text{Di}} = \sum_{x_i \in X} |D(x_i)|$. $A[X]$ 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 $\forall x_i \in X$, $A[x_i] \in D(x_i)$. An *instantiation* $I[X]$ is a valid assignment of X. Given $x \in X$, $I[x]$ is the value of x in $I[X]$. Given $X = [x_0, x_1, \ldots, x_{n-1}]$ and i, j two integers s.t. $0 \leq i \leq j \leq n-1$, $I[x_i, \ldots, x_j]$ is the projection of $I[X]$ on the sequence $[x_i, \ldots, x_j]$. A *constraint* $C(X) \in \mathcal{C}$ specifies the allowed combinations of values for a sequence of variables X. We also use the simple notation C, and for this constraint, its set of variables is denoted by $vars(C)$. $C(X)$ defines a subset $\mathcal{R}_C(\mathcal{D})$ of the Cartesian product of the domains $\Pi_{x_i \in X} D(x_i)$. If X is a pair of variables, then $C(X)$ is a *binary* constraint. We denote by vCw a pair of values (v, w) that satisfies a binary constraint C. We denote by $\neg C$ the *opposite* of C, that is, $\neg C$ defines the relation $\mathcal{R}_{\neg C}(\mathcal{D}) = \Pi_{x_i \in X} D(x_i) \setminus \mathcal{R}_C(\mathcal{D})$. A *feasible instantiation* of C 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* $C(X)$. Otherwise, $I[X]$ *violates* $C(X)$. A *solution* of a constraint network is an instantiation of all the variables satisfying the constraints of $\mathcal{C}$.

Definition 1 (GAC for a constraint). *Let $C(X)$ be a constraint. A* **support** *on $C(X)$ is an instantiation $I[X]$ which satisfies $C(X)$. A value $v \in D(X)$ is*

(generalized) **arc-consistent** *(GAC) w.r.t.* $C(X)$ *iff* v *belongs to a support of* $C(X)$*. A domain* $D(x)$ *is GAC w.r.t.* $C(X)$ *iff* $\forall v \in D(x)$*,* v *is GAC w.r.t.* $C(X)$*.* $C(X)$ *is GAC iff* $\forall x_i \in X$*,* $D(x_i)$ *is arc-consistent w.r.t.* $C(X)$*.*

Definition 2 (Closure [6]). *Let* $\mathcal{N}(X, \mathcal{D}, \mathcal{C})$ *be a constraint network and* $\mathcal{C}$ *a set of constraints.* $\text{GAC}(\mathcal{D}, \mathcal{C})$ *is the* **closure** *of* $\mathcal{D}$ *for GAC on* $\mathcal{C}$*, i.e., the set of domains obtained from* $\mathcal{D}$ *where* $\forall X$*, all values* $a \in D(X)$ *that are not GAC w.r.t. a constraint in* $\mathcal{C}$ *have been removed. A* GAC *constraint network is a constraint network which is closed for* GAC*.*

3 The SEQ_BIN Constraint Class

In order to define the SEQ_BIN constraint, we first introduce a generalized definition of stretches [13], which corresponds to a sequence of consecutive variables where the same binary constraint is satisfied.

Definition 3 (unary-true, unary-false). *A constraint* $C(X)$ *is unary-true (resp. unary-false) iff it can be interpreted on a singleton variable* $X = \{x\}$ *and, in this case, this interpretation is the universal constraint (resp. the complementary of the universal constraint).*

For instance, the binary constraint $=$ is unary-true because $\forall v \in D(x)$ we have always $x = x$. On the contrary, the binary constraint $\neq$ is unary-false because $\forall v \in D(x), x \neq x$. Some constraints, *e.g.*, "contains 0", are neither unary-true nor unary-false.

Definition 4 (*C*-stretch). *Let* $I[X]$ *be an instantiation of the variable sequence* $X = [x_0, \ldots, x_{n-1}]$*. Given* i *and* j*,* $0 \leq i \leq j \leq n-1$*, a binary constraint* C*, a* C-stretch *of* $I[X]$ *is a sequence of consecutive variables* $[x_i, \ldots, x_j]$ *s.t. the three following conditions are all satisfied:*

1. $(i = 0) \vee$ *(*$I[x_{i-1}]\, C\, I[x_i]$ *does not hold).*
2. $(j = n-1) \vee$ *(*$I[x_j]\, C\, I[x_{j+1}]$ *does not hold).*
3. $((i = j) \wedge$ *((*C *is unary-false)* $\vee$ *(*C *unary-true)* $\vee$ *(*$I[x_i]\, C\, I[x_i]$ *holds))* $\vee$ $((i \neq j) \wedge$ *(*$\forall k \in [i, j-1]$*:* $I[x_k]\, C\, I[x_{k+1}]$ *holds)).*

Note that, when C is the equality constraint between two variables, the number of C-stretches corresponds to the classical definition of stretches, introduced by Pesant [13]. Thanks to this generalized definition of stretches, we can define the constraint class SEQ_BIN.

Definition 5. *The constraint class* $\text{SEQ\_BIN}(N, X, C, B)$ *is defined by a variable* N*, a sequence of* n *variables* $X = [x_0, x_1, \ldots, x_{n-1}]$ *and two binary constraints* C *and* B*. Given an instantiation* $I[N, x_0, x_1, \ldots, x_{n-1}]$*,* $\text{SEQ\_BIN}(N, X, C, B)$ *is satisfied iff for any* $i \in [0, n-2]$*,* $I[x_i]\, B\, I[x_{i+1}]$ *holds, and* $I[N]$ *is equal to the number of* C-stretches *in* X*.*

Let us now present some well-known constraint that can be represented thanks to SEQ_BIN. The constraint CHANGE was initially introduced in CHIP [9] in the context of timetabling problems, in order to put an upper limit on the number of changes of job types during a given period. The relation between classical stretches and CHANGE was initially stressed in [11, page 64].

Definition 6. *The* CHANGE *constraint is defined by a variable N, a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$, and a binary constraint $C \in \{=, \neq, <, >, \leq, \geq\}$. Given an instantiation,* CHANGE(N, X, C) *is satisfied iff N is equal to the number of times the constraint C holds on consecutive variables of X.*

Lemma 1. *Given an instantiation $I[X]$, the number of pairs of variables (x_i, x_{i+1}) s.t. $I[x_i]\, C\, I[x_{i+1}]$, is equal to the number of $\neg C$-stretches in $I[X]$ less one.*

Proof. By induction on the number of stretches. If there is one single $\neg C$-stretch, all the consecutive pairs of values in $I[X]$ violate C. Otherwise, let s be the number of $\neg C$-stretches. Let $[x_k, \ldots, x_{n-1}]$ be the last $\neg C$-stretch in X according to $I[X]$, and assume that the number of pairs of consecutive variables (x_i, x_{i+1}) in $I[x_0, \ldots, x_{k-1}]$ s.t. $I[x_i]\, C\, I[x_{i+1}]$ is equal to $s-2$. By construction, $I[x_{k-1}]\, C\, I[x_k]$ and $\forall i \geq k$, $I[x_i] \neg C I[x_{i+1}]$. The number of pairs of instantiated variables s.t. C is satisfied is $s-1$. The lemma holds. □

Without hindering propagation,[1] the CHANGE constraint can be reformulated as:

$$\text{CHANGE}(N, X, C) \Leftrightarrow \text{SEQ\_BIN}(N', X, \neg C, \texttt{true}) \wedge [N' = N - 1]$$

where `true` is the universal constraint. The SMOOTH constraint [2] was introduced in order to restrict the number of drastic variations on a cumulative profile. Given an integer *cst*, the constraint SMOOTH(N, X, cst) can be seen as a CHANGE(N, X, C) constraint, where $x_i\, C\, x_{i+1}$ is equivalent to $|x_i - x_{i+1}| > cst$.

The INCREASING_NVALUE constraint is a specialized version of NVALUE [1] that was introduced for breaking variable symmetry in the context of resource allocation problems [4].

Definition 7. *The* INCREASING_NVALUE *constraint is defined by a variable N and a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$. Given an instantiation,* INCREASING_NVALUE(N, X) *is satisfied iff N is equal to the number of distinct values assigned to the variables in X, and $\forall i \in [0, n-2]$, $x_i \leq x_{i+1}$.*

Any feasible instantiation $I[X]$ of INCREASING_NVALUE(N, X) satisfies $I[x_i] \leq I[x_j]$ for all $i < j$. We say that an instantiation $I[x_0, x_1, \ldots, x_{n-1}]$ is *well-ordered* iff for i and j s.t. $0 \leq i < j \leq n-1$, we have $I[x_i] \leq I[x_j]$. If $I[X]$ satisfies INCREASING_NVALUE(X, N) then $I[X]$ is well-ordered.

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

[1] The corresponding constraint network is obviously Berge-acyclic [5].

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 $I[x_i] < I[x_j]$. □

Selecting only well-ordered instantiations while defining INCREASING_NVALUE by SEQ_BIN is done by defining B as the binary inequality constraint $\leq$. From Lemma 2, we can reformulate the constraint INCREASING_NVALUE:

$$\text{INCREASING\_NVALUE}(N, X) \Leftrightarrow \text{SEQ\_BIN}(N, X, =, \leq)$$

We consider the AMONG global constraint [3, 7], initially introduced in the context of car sequencing [3].

Definition 8. *The* AMONG *constraint is defined by a variable N, a sequence of variables $X = [x_0, \ldots, x_{n-1}]$ and a set of values $\mathcal{V}$. Given an instantiation,* $\text{AMONG}(N, X, \mathcal{V})$ *is satisfied iff N is equal to the number of variables $x_i \in X$ s.t. $I[x_i] \in \mathcal{V}$.*

To represent AMONG with SEQ_BIN, we consider the following specific binary constraint, which is the unary operator $\notin$. We use a binary constraint to be consistent with Definition 5.

Definition 9. *Given an array of values $\mathcal{V}$, the binary constraint* $\text{NOTIN\_BIN}(\mathcal{V})$ *is defined on a pair of variables* $vars(\text{NOTIN\_BIN}(\mathcal{V})) = [x_i, x_{i+1}]$. *Given an instantiation $I[x_i, x_{i+1}]$,* $\text{NOTIN\_BIN}(\mathcal{V})$ *is satisfied iff $I[x_i] \notin \mathcal{V}$.*

Then, we can reformulate the constraint AMONG:

$$\text{AMONG}(N, X, \mathcal{V}) \Leftrightarrow \text{SEQ\_BIN}(N', X, \text{NOTIN\_BIN}(\mathcal{V}), \texttt{true}) \wedge [N' = N - 1]$$

Motivated by variable symmetry breaking, we also introduce the INCREASING_AMONG constraint which we can reformulate in the following way:

$$\text{INCREASING\_AMONG}(N, X, \mathcal{V}) \Leftrightarrow \text{SEQ\_BIN}(N', X, \text{NOTIN\_BIN}(\mathcal{V}), \leq) \wedge [N' = N - 1]$$

4 Necessary and Sufficient Filtering Condition

In this section, we first present how to compute, for any value in a given domain, the minimum and maximum possible number of C-stretches within a sequence of variables satisfying a chain of binary constraints of type B. Then, we introduce three properties useful to obtain a feasibility condition for SEQ_BIN. Finally, we derive from this feasibility condition a necessary and sufficient condition w.r.t. filtering, which leads to the GAC filtering algorithm presented in Section 5.

4.1 Computation of the Number of C-stretches

Definition 10. *An instantiation $I[x_0, x_1, \ldots, x_{n-1}]$ is said to be* B-coherent *iff either $n = 1$ or given $i \in [0, n-1[$, we have $I[x_i]\, B\, I[x_{i+1}]$. A value $v \in D(x_i)$ is said to be* B-coherent *with respect to x_i iff it can be part of at least one B-coherent instantiation.*

Such a definition directly leads to the following property:

Property 1. Given $X = [x_0, x_1, \ldots, x_{n-1}]$ a sequence of variables, and an integer $i \in [0, n-1[$, if $v \in D(x_i)$ is B-coherent w.r.t. x_i then $\exists w \in D(x_{i+1})$ s.t. $v\, B\, w$.

Within a given domain $D(x_i)$, values that are not B-coherent can be removed since they cannot be part of a solution of SEQ_BIN. Now our aim is to compute for each B-coherent value v in the domain of any variable x_i the minimum and maximum number of *C-stretches* on sequence $[x_0, x_1, \ldots, x_{n_1}]$.

Notation 1 *We denote by $\underline{s}(x_i, v)$ (resp. $\overline{s}(x_i, v)$) the minimum (resp. maximum) number of C-stretches within the sequence $[x_i, x_{i+1}, \ldots, x_{n-1}]$ under the hypothesis that $x_i = v$. Similarly, we denote by $\underline{p}(x_i, v)$ (resp. $\overline{p}(x_i, v)$) the minimum (resp. maximum) number of C-stretches within the sequence $[x_0, x_1, \ldots, x_i]$ under the hypothesis that $x_i = v$.*

Lemma 3. *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and one instance of the* SEQ_BIN(N, X, C, B) *constraint. Assume the domains in X contain only B-coherent values. Given $i \in [0, n-1]$ and $v \in D(x_i)$,*

- *Assume $i = n-1$: If vCv then $\underline{s}(x_{n-1}, v) = 1$, otherwise $\underline{s}(x_{n-1}, v) = 0$.*
- *Otherwise:*

$$\underline{s}(x_i, v) = \min_{w \in D(x_{i+1})} \left(\min_{[vBw] \wedge [vCw]} (\underline{s}(x_{i+1}, w)), \min_{[vBw] \wedge [v \neg Cw]} (\underline{s}(x_{i+1}, w)) + 1 \right)$$

Proof. By induction. From Definition 4, for any $v \in D(x_{n-1})$, we have $\underline{s}(x_{n-1}, v) = 1$ iff vCv, and 0 otherwise (i.e., a *C-stretch* of length 1). Consider now a variable x_i, $0 \leq i < n-1$, and a value $v \in D(x_i)$. Consider the set of instantiations $I[x_{i+1}, \ldots, x_{n-1}]$ that are B-coherent, and that minimize the number of *C-stretches* in $[x_{i+1}, \ldots, x_{n-1}]$. We denote this minimum number of *C-stretches* by δ. At least one B-coherent instantiation exists since all values in the domains of $x_{i+1}, \ldots, x_{n-1}$ are B-coherent. For each such instantiation, let us denote by w the value associated with $I[x_{i+1}]$.

– *Assume $i = n-1$: If vCv then $\overline{s}(x_{n-1}, v) = 1$, otherwise $\overline{s}(x_{n-1}, v) = 0$.*
– *Otherwise:*

$$\overline{s}(x_i, v) = \max_{w \in D(x_{i+1})} \left(\max_{[vBw] \wedge [vCw]} (\overline{s}(x_{i+1}, w)), \max_{[vBw] \wedge [v\neg Cw]} (\overline{s}(x_{i+1}, w)) + 1 \right)$$

Given a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$ s.t. domains in X contain only B-coherent values, $\forall x_i \in X$, $\forall v \in D(x_i)$, computing $\underline{p}(x_i, v)$ (resp. $\overline{p}(x_i, v)$) is symmetrical to $\underline{s}(x_i, v)$ (resp. $\overline{s}(x_i, v)$). Indeed, we only have to substitute min by max (resp. max by min), x_{i+1} by x_{i-1}, and vXw by wXv for any $X \in \{B, C, \neg C\}$.

4.2 Properties on the Number of *C*-stretches

This section gives the properties that link the values in a domain $D(x_i)$ with the minimum and maximum number of C-stretches obtained in the sequence $[x_i, \ldots, x_{n-1}]$, which will be useful to establish a necessary and sufficient condition for achieving GAC on SEQ_BIN. We consider only B-coherent values, which may be part of a feasible instantiation of the SEQ_BIN constraint. First of all, the following property can be naturally deduced from Lemmas 3 and 4.

Property 2. For any B-coherent value v in $D(x_i)$, w.r.t. x_i, $\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$.

Property 3. Consider an instance of SEQ_BIN(N, X, C, B), a variable $x_i \in X$ $(0 \leq i \leq n-1)$, and two B-coherent values $v_1 \in D(x_i)$, $v_2 \in D(x_i)$. If $i = n-1$ or if there exists $w \in D(x_{i+1})$ s.t. $v_1 B w$ and $v_2 B w$, then $\overline{s}(x_i, v_1) + 1 \geq \underline{s}(x_i, v_2)$.

Proof. The result is obvious if $i = n-1$. If $v_1 = v_2$, by Property 2 the property holds. Otherwise, assume that there exist two values v_1 and v_2 s.t. $\exists w \in D(x_{i+1})$ for which $v_1 B w$ and $v_2 B w$, and $\overline{s}(x_i, v_1) + 1 < \underline{s}(x_i, v_2)$ (hypothesis). By Lemma 4, $\overline{s}(x_i, v_1) \geq \overline{s}(x_{i+1}, w)$. By Lemma 3, $\underline{s}(x_i, v_2) \leq \underline{s}(x_{i+1}, w) + 1$. Thus, by hypothesis, $\overline{s}(x_{i+1}, w) + 1 < \underline{s}(x_{i+1}, w) + 1$, which leads to $\overline{s}(x_{i+1}, w) < \underline{s}(x_{i+1}, w)$, which is, by Property 2, not possible. □

Property 4. Consider an instance of SEQ_BIN(N, X, C, B), a variable $x_i \in X$ $(0 \leq i \leq n-1)$, and two B-coherent values $v_1 \in D(x_i)$, $v_2 \in D(x_i)$. If either $i = n-1$ or there exists $w \in D(x_{i+1})$ s.t. $v_1 \, B \, w$ and $v_2 \, B \, w$ then, for any $k \in [\min(\underline{s}(x_i, v_1), \underline{s}(x_i, v_2)), \max(\overline{s}(x_i, v_1), \overline{s}(x_i, v_2))]$, either $k \in [\underline{s}(x_i, v_1), \overline{s}(x_i, v_1)]$ or $k \in [\underline{s}(x_i, v_2), \overline{s}(x_i, v_2)]$.

Proof. The result is obvious if $i = n-1$ or $v_1 = v_2$. If $[\underline{s}(x_i, v_1), \overline{s}(x_i, v_1)] \cap [\underline{s}(x_i, v_2), \overline{s}(x_i, v_2)]$ is not empty, then the property holds. Assume that $[\underline{s}(x_i, v_1), \overline{s}(x_i, v_1)]$ and $[\underline{s}(x_i, v_2), \overline{s}(x_i, v_2)]$ are disjoint. Assume w.l.o.g. that $\overline{s}(x_i, v_1) < \underline{s}(x_i, v_2)$. By Property 3, $\overline{s}(x_i, v_1) + 1 \geq \underline{s}(x_i, v_2)$, thus $\overline{s}(x_i, v_1) = \underline{s}(x_i, v_2) - 1$. Either $k \in [\underline{s}(x_i, v_1), \overline{s}(x_i, v_1)]$ or $k \in [\underline{s}(x_i, v_2), \overline{s}(x_i, v_2)]$ (there is no hole in the range formed by the union of these two intervals). □

4.3 Binary Neighborhood Substitutable Constraints

Property 4 is central to providing a GAC filtering algorithm which is based on the count, for each B-coherent value in a domain, of the minimum and maximum number of *C-stretches* among all the possible complete instantiations. This section determines which class of binary constraints B guarantees that.

Definition 11 (neighborhood substitutable values [10]). *Given a binary constraint $C(X)$, two values v_1 and v_2 in the domain of $x \in X$, v_1 is* neighborhood substitutable *for v_2 in C iff $\{w \mid v_2\, C\, w\} \subseteq \{w \mid v_1\, C\, w\}$.*

We extend this definition to define the class of binary constraints that are neighborhood substitutable whatever the values in domains are.

Definition 12 (neighborhood substitutable binary constraint). *A binary constraint C is* neighborhood substitutable *iff $\forall x \in vars(C)$, $\forall D(x) \in \mathcal{D}$ a domain for x, $\forall u \in D(x)$, $\forall v \in D(x)$, either u is neighborhood substitutable for v in C, or v is neighborhood substitutable for u in C.*

The binary constraints $<$, $>$, $\leq$ and $\geq$ are neighborhood substitutable, as well as any binary constraint that corresponds to one of these two constraints thanks to a renaming of values in domains. Obviously, the universal constraint `true` is also neighborhood substitutable. Conversely, the binary constraint $=$ is not neighborhood substitutable.

Property 5. Given an instance of SEQ_BIN(N, X, C, B), B is neighborhood substitutable iff for any variable $x_i \in X$, $0 \leq i < n-1$, for any values $v_1, v_2 \in D(x_i)$, there exists $w \in D(x_{i+1})$ s.t. $v_1 B w$ and $v_2 B w$.

Proof. ($\Rightarrow$) From Definition 12 and since we consider only B-coherent values, each value has at least one support on B. Moreover, from Definition 12, $\{w \mid v_2 C w\} \subseteq \{w \mid v_1 C w\}$ or $\{w \mid v_1 C w\} \subseteq \{w \mid v_2 C w\}$. The property holds. ($\Leftarrow$) Suppose that the second proposition is true and B is not neighborhood substitutable. From Definition 12, if B is not neighborhood substitutable then $\exists v_1$ and v_2 in the domain of a variable $x_i \in X$ s.t., by considering the constraint B on the pair of variables (x_i, x_{i+1}), neither $\{w \mid v_2 C w\} \subseteq \{w \mid v_1 C w\}$ nor $\{w \mid v_1 C w\} \subseteq \{w \mid v_2 C w\}$. Thus, there exists a support $v_1 B w$ s.t. (v_2, w) is not a support on B, and a support $v_2 B w'$ s.t. (v_1, w') is not a support on B. We can have $D(x_{i+1}) = \{w, w'\}$, which leads to a contradiction with the second proposition. The property holds. □

4.4 Checking Feasibility

From Property 5, we here establish an equivalence relation between the existence of a solution for SEQ_BIN and the current variable domains of X and N. According to Definition 5, Proposition 1 first proposes a necessary condition for SEQ_BIN.

Notation 2 *Given a sequence of variables $X = [x_0, \ldots, x_{n-1}]$, $\underline{s}(X)$ (resp. $\overline{s}(X)$) denotes the minimum (resp. maximum) value of $\underline{s}(x_0, v)$ (resp. $\overline{s}(x_0, v)$).*

Proposition 1. *Given an instance of* SEQ_BIN(N, X, C, B)*, if* $\underline{s}(X) > \max(D(N))$ *or* $\overline{s}(X) < \min(D(N))$ *then the constraint has no solution.*

W.l.o.g., $D(N)$ can be restricted to $[\underline{s}(X), \overline{s}(X)]$. However, note that $D(N)$ may have holes or may be strictly included in $[\underline{s}(X), \overline{s}(X)]$. We prove that, provided B is neighborhood substitutable, for any value k in $[\underline{s}(X), \overline{s}(X)]$ there exists a value $v \in D(x_0)$ s.t. $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$. Thus, any value in $D(N) \cap [\underline{s}(X), \overline{s}(X)]$ is generalized arc-consistent.

Proposition 2. *Consider an instance of* SEQ_BIN(N, X, C, B) *such that B is neighborhood substitutable, with $X = [x_0, x_1, \ldots, x_{n-1}]$. For any integer k in $[\underline{s}(X), \overline{s}(X)]$ there exists v in $D(x_0)$ s.t. $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$.*

Proof. Let $v_1 \in D(x_0)$ a value s.t. $\underline{s}(x_0, v_1) = \underline{s}(X)$. Let $v_2 \in D(x_0)$ a value s.t. $\overline{s}(x_0, v_2) = \overline{s}(X)$. By Property 5, either $n = 1$ or $\exists w \in D(x_1)$ s.t. $v_1 B w$ and $v_2 B w$. Thus, from Property 4, $\forall k \in [\underline{s}(X), \overline{s}(X)]$, either $k \in [\underline{s}(x_0, v_1), \overline{s}(x_0, v_1)]$ or $k \in [\underline{s}(x_0, v_2), \overline{s}(x_0, v_2)]$. The proposition holds. □

Proposition 3. *Given an instance of* SEQ_BIN(N, X, C, B) *such that the constraint B is neighborhood substitutable,* SEQ_BIN(N, X, C, B) *has a solution if and only if $[\underline{s}(X), \overline{s}(X)] \cap D(N) \neq \emptyset$.*

Proof. ($\Rightarrow$) Assume SEQ_BIN(N, X, C, B) has a solution. Let $I[\{N\} \cup X]$ be such a solution. By construction (Lemmas 3 and 4), the number of C-stretches $I[N]$ belongs to $[\underline{s}(X), \overline{s}(X)]$. ($\Leftarrow$) Let $k \in [\underline{s}(X), \overline{s}(X)] \cap D(N)$ (not empty). From Proposition 2, for any value k in $[\underline{s}(X), \overline{s}(X)]$ there exists $v \in D(x_0)$ s.t. $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$. By Definition 5 and since Lemmas 3 and 4 consider only B-coherent values, there is a solution of SEQ_BIN(N, X, C, B) with k C-stretches. □

4.5 Necessary and Sufficient Filtering Condition

Given an instance of SEQ_BIN(N, X, C, B), Proposition 3 can be used to filter the variable N according to the sequence of variables X. In this way, Propositions 1 and 2 ensure that every remaining value in $[\underline{s}(X), \overline{s}(X)] \cap D(N)$ is involved in at least one solution satisfying SEQ_BIN. Now, this section focuses, with the following proposition, on how to filter variables X according to N.

Proposition 4. *Given an instance of* SEQ_BIN(N, X, C, B) *such that B is neighborhood substitutable, let $i \in [0, n-1]$ be an integer and let v be a value in $D(x_i)$. Let Δ be an integer value s.t. $\Delta = 1$ iff $C(v, v)$ is satisfied, $\Delta = 0$ otherwise. The two following propositions are equivalent:*

1. *v is B-coherent and v is generalized arc-consistent w.r.t.* SEQ_BIN.
2. $[\underline{p}(x_i, v) + \underline{s}(x_i, v) - \Delta, \overline{p}(x_i, v) + \overline{s}(x_i, v) - \Delta] \cap D(N) \neq \emptyset$.

Proof. If v is not B-coherent then, by Definition 5, v is not GAC. Otherwise, $\underline{p}(x_i, v)$ (resp. $\underline{s}(x_i, v)$) is the exact minimum number of *C-stretches* among B-coherent instantiations $I[x_0, \ldots, x_i]$ (resp. $I[x_i, \ldots, x_{n-1}]$) s.t. $I[x_i] = v$.

Thus, by Lemma 3 for $\underline{s}$ and symmetrically for $\underline{p}$, the exact minimum number of C-stretches among B-coherent instantiations $I[x_0, \ldots, x_{n-1}]$ s.t. $I[x_i] = v$ is $\underline{p}(x_i, v) + \underline{s}(x_i, v) - \Delta$. Let $\mathcal{D}_v \subseteq \mathcal{D}$ be the set of domains s.t. 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 2, $\underline{p}(x_i, v) + \underline{s}(x_i, v) - \Delta = \underline{s}(X_v)$. By a symmetrical reasoning, $\overline{p}(x_i, v) + \overline{s}(x_i, v) - \Delta = \overline{s}(X_v)$. By Proposition 3, the proposition holds. □

Note that, in Proposition 4, the quantity Δ depends on the satisfaction of $C(v, v)$: it prevents us from counting twice a C-stretch corresponding to an extremity x_i of each sequence of variables, $[x_0, \ldots, x_i]$ or $[x_i, \ldots, x_{n-1}]$.

5 A Linear Time Generic GAC Filtering Algorithm

Based on the necessary and sufficient filtering condition introduced in Section 4 (see Proposition 4), this section provides an implementation of the corresponding GAC filtering algorithm for SEQ_BIN(N, X, C, B). We first sketch its principle, second describe the services that are used in order to make the algorithm generic, third give the detailed algorithm, fourth discuss its worst-case complexity.

5.1 Principle of the Filtering Algorithm

Given the sequence $X = [x_0, x_1, \ldots, x_{n-1}]$, the algorithm is decomposed into the following four successive phases:

① It removes all non B-coherent values from the variables of X.
② For all prefixes and suffixes of X, it computes the minimum and maximum number of *C-stretches*.
③ It respectively adjusts the minimum and maximum value of N w.r.t. the minimum and maximum number of *C-stretches* of X.
④ By using the information computed in Phase ② as well as Proposition 4, it tries to prune each remaining B-coherent value.

5.2 Services Used by the Filtering Algorithm

A first class of services deals with the efficient computation of the minimum and maximum number of *C-stretches* on suffixes (and prefixes) of the sequence of variables X that must be done in Phase ②. A second class of services, used in Phase ①, provides some checking and filtering facilities for binary constraints (e.g., constraints C and B in our case). Finally, a third class used all over the algorithm, describes the access to the variables (i.e., getting an information related to the domain of a variable or restricting its domain).

We are first interested in the minimum and maximum number of C-stretches related services. Given the two binary constraints C and B of SEQ_BIN(N, X, C, B), Phase ② needs both to *compute* and to *record* the minimum and maximum number of C-stretches on the prefixes and suffixes of X. For this purpose we respectively use the following services:

– **Computing:** As we will see later on in Section 6, different data structures are required for different pairs of binary constraints C and B for efficiently evaluating the formulae of Lemmas 3 and 4. The service AllocDataStructure(C, B, *data_structure*) allocates the data structure that is required to efficiently evaluate $\underline{s}[i-1, v]$ and $\overline{s}[i-1, v]$ for the different values v in x_i w.r.t. the binary constraints C and B. The service InitDataStructure(C, B, i, *data_structure*) initializes the data structure part related to column i required to efficiently evaluate $\underline{s}[i-1, v]$ and $\overline{s}[i-1, v]$ for the different values v in x_i. The service ComputeSuffix(C, B, i, *data_structure*), computes $\underline{s}[i, v]$ and $\overline{s}[i, v]$ for the different values v in x_i with the help of the data structure computed on the previous column $i+1$.
– **Recording:** To record the minimum and maximum number of *C-stretches* on the prefixes and suffixes of X, we reuse from [4] the *sparse matrices data structure*. Write and read accesses are always done by iterating in increasing or decreasing order through the rows in a given column (i.e., the domain of a given variable x_i). ScanInit(*matrices*, i, *dir*) indicates that we will iterate through the i^{th} column of the sparse matrices in *matrices* in increasing order (*dir* $=\uparrow$) or decreasing order (*dir* $=\downarrow$). Set(*matrix*, i, j, *info*) performs the assignment *matrix*$[i, j]$:= *info*. Get(*matrix*, i, j):int returns the content of entry *matrix*$[i, j]$ or the default value if such entry does not belong to the sparse matrix.

Now, we can detail services related to the binary constraints. Given a binary constraint C and two values u and v, the service Check(C, u, v) returns 1 if uCv and 0 otherwise. Given a binary constraint B and two variables x and y, the service Filter(B, x_i, x_{i+1}) removes from $D(x_i)$ all the values that have no support on B, given the current domain of x_{i+1}. It returns false if the domain of x becomes empty, and true otherwise.[2]

Finally, we present services related to the variables. get_prev(x, v):int (resp. get_next(x, v):int) returns the largest (resp. smallest) value w in $D(x)$ such that $w < v$ (resp. $w > v$) if it exists, and v otherwise. adjust_min(x, v):boolean (resp. adjust_max(x, v):boolean) adjusts the minimum (resp. maximum) of variable x to value v. Finally, remove_val(x, v):boolean removes value v from domain $D(x)$.

5.3 The Generic Filtering Algorithm

Algorithm 2 implements Proposition 4. Since it can be interpreted as a generalization of the filtering algorithm associated with the Increasing_Nvalue constraint [4] it follows its structure, except that it uses the set of services we just introduced. First, it restricts the domains of variables $[x_0, x_1, \ldots, x_{n-1}]$ in order to only keep B-coherent values. Second, it computes the information related to the minimum and maximum number of stretches on the prefix and suffix matrices $\underline{p}$, $\overline{p}$, $\underline{s}$, $\overline{s}$. Next, based on this information, it adjusts the bounds of N

[2] When B is the universal constraint `true`, the service does not remove any value.

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

```
1  AllocDataStructure(C, B, data_structure); ScanInit({s̲, s̄}, n − 1, ↓); v := max(x_{n−1});
2  repeat
3  |   u :=Check(C, v, v); Set(s̲, n − 1, v, u); Set(s̄, n − 1, v, u);
4  |   w := v; v :=getPrev(x_{n−1}, v);
5  until w = v ;
6  for i := n − 2 downto 0 do
7  |   InitDataStructure(C, B, i + 1, data_structure); ScanInit({s̲, s̄}, i, ↓); v := max(x_i);
8  |   repeat
9  |   |   ComputeSuffix(C, B, i, data_structure); w := v; v :=getPrev(x_i, v);
10 |   until w = v ;
```

and does the necessary pruning on variables $x_0, x_1, \ldots, x_{n-1}$. Using Lemmas 3 and 4, Algorithm 1 builds the suffix matrices $\underline{s}$ and $\overline{s}$ used in Algorithm 2.

Algorithm 2: SEQ_BIN$(N, [x_0, \ldots, x_{n-1}], C, B)$: boolean.

```
1  for i = 1 to n − 1 do
2  |   if (i < n − 1 ∧ ¬Filter(B, x_{i−1}, x_i)) ∨ (i = n − 1 ∧ ¬Filter(¬B, x_{n−1}, x_{n−2})) then
3  |   |   return false;
4  AllocDataStructure p̲, p̄, s̲, s̄;
5  build_prefix(C, B, [x_0, ..., x_{n−1}], p̲, p̄); build_suffix(C, B, [x_0, ..., x_{n−1}], s̲, s̄);
6  ScanInit({s̲, s̄}, 0, ↑);
7  if ¬adjust_min(N, min_{v∈D(x_0)}(Get(s̲, 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({p̲, p̄, s̲, s̄}, i, ↑); v := min(x_i);
11 |   repeat
12 |   |   u :=Check(C, v, v);
13 |   |   N̲_v :=Get(p̲, i, v)+Get(s̲, i, v) − u; N̄_v :=Get(p̄, i, v)+Get(s̄, i, v) − u;
14 |   |   if [N̲_v, N̄_v] ∩ D(N) = ∅ ∧ ¬remove_val(x_i, v) then return false;
15 |   |   w := v; v := getNext(x_i, v);
16 |   until w = v ;
17 return true;
```

5.4 Complexity

To provide the overall worst-case complexity of the generic filtering algorithm we first give the complexity of the services that were previously introduced:

- ***C-stretches* related services:** For the computing part, AllocDataStructure is assumed to be done in constant time, while InitDataStructure and all the calls to ComputeSuffix, w.r.t. a variable x_i, are assumed to be done in $O(|D(x_i)|)$. The next section will show how to implement such services for different concrete constraints. For the recording part, Check is

done in constant time, while FILTER(B, x_i, x_{i+1}) is done in time $O(|D(x_i)|)$ (details at `http://choco.emn.fr`).
- **Binary constraints related services:** SCANINIT and SET are assumed to be done in constant time, while a set of p consecutive calls to GET on the same column i and in increasing or decreasing row indexes is in $O(p)$.
- **Variables related services** are assumed to take constant time.

From the previous assumptions, Algorithms 1 and 2 both mainly scan the different variables x_i and the values in their domains, each is done in $O(\Sigma_{\text{Di}})$ time.

6 Implementing Specific Constraints

Some specific data structures are needed to achieve GAC in $O(\Sigma_{\text{Di}})$ for all the constraints that were introduced in Section 3. They are used for evaluating the minimum number of stretches on a suffix of the sequence of variables $[x_0, x_1, \ldots, x_{n-1}]$ w.r.t. a variable x_i $(0 \leq i < n)$ and one B-coherent value $v \in D(x_i)$ (see Lemma 3).[3] For this purpose, we provide the sketch of an efficient implementation of the services ALLOCDATASTRUCTURE, INITDATASTRUCTURE and COMPUTESUFFIX for the constraints CHANGE$(\neq)$, CHANGE$(\leq)$, SMOOTH, INCREASING_NVALUE, AMONG. In all cases, lines 7-10 of Algorithm 1 are performed in $O(|D(x_i)|)$ time, which leads to overall time complexity for achieving GAC in $O(\Sigma_{\text{Di}})$ time.

$\diamond$ CHANGE$(N, X, \neq)$: SEQ_BIN where C is '=' and B is '`true`'. For any value $v \in D(x_i)$, we need to evaluate the quantity $\underline{s}(x_i, v)$ without scanning each value in the domain of next variable x_{i+1}. By Lemma 3, $\underline{s}(x_i, v) = \min_{w \in D(x_{i+1})} \big(\underline{s}(x_{i+1}, v), \min_{[w \neq v]}(\underline{s}(x_{i+1}, w)) + 1\big)$. This formula can be simplified to $\underline{s}(x_i, v) = \min\big(\underline{s}(x_{i+1}, v), \min_{w \in D(x_{i+1})}(\underline{s}(x_{i+1}, w)) + 1\big)$. The data structure is a single integer value min_1, the minimum value of the i^{th} column of matrix $\underline{s}[][]$. We sketch the implementation of the services:

1. [ALLOCDATASTRUCTURE]: Allocates, without initializing it, the item min_1.
2. [INITDATASTRUCTURE]: min_1 is set to the minimum of the i^{th} column of matrix $\underline{s}[][]$.
3. [COMPUTESUFFIX]: For each value $v \in D(x_i)$, $0 \leq i < n-1$, the quantity $\underline{s}(x_i, v)$ is equal to $\min(\underline{s}(x_{i+1}, v), min_1 + 1)$.

Note that, as required, both INITDATASTRUCTURE and all the calls to COMPUTESUFFIX can be done in $O(|D(x_i)|)$ time, by iterating through the values of $D(x_i)$ with the GETNEXT primitive. The state of the art shows an algorithm achieving GAC for such a CHANGE constraint with a time complexity of $O(n^3 m)$, where m is the total number of values in the domain of the variables of X,

[3] Since data structures for the maximum number of stretches on a suffix and for the minimum and maximum number of stretches on a prefix can be done in the similar way, they will be omitted.

sketched in [11, page 57].

◇ CHANGE$(N, X, \leq)$: SEQ_BIN where C is '$>$' and B is '`true`'. For any value $v \in D(x_i)$, we need to evaluate the quantity $\underline{s}(x_i, v) = \min_{w \in D(x_{i+1})} \big(\min_{[v>w]}(\underline{s}(x_{i+1}, w)), \min_{[v \leq w]}(\underline{s}(x_{i+1}, w)) + 1\big)$. The question is how to efficiently evaluate the terms $\min_{[v>w]}(\underline{s}(x_{i+1}, w))$ and $\min_{[v \leq w]}(\underline{s}(x_{i+1}, w))$. For this purpose, the data structures we need for a column i consist of two sparse arrays (i.e., a sparse matrix with one single column) $gt[]$ and $leq[]$: (1) $gt[v]$ is the minimum value over entries $\underline{s}[i][w]$, $w \in]v, \max(x_i)]$, (2) $leq[v]$ is the minimum value over entries $\underline{s}[i][w]$, $w \in [\min(x_i), v]$. We sketch the implementation of the services:

1. [ALLOCDATASTRUCTURE] Allocates, without initializing them, the sparse arrays $gt[]$ and $leq[]$.
2. [INITDATASTRUCTURE] First, $gt[\max(x_i)]$ is set to $\max(N) + 1$. By iterating through the values of variable x_i from GETPREV$(\max(x_i))$ down to $\min(x_i)$, we set $gt[v]$ to $\min(gt[\text{GETNEXT}(x_i, v)], \underline{s}[i][\text{GETNEXT}(x_i, v)])$. Second, $leq[\min(x_i)]$ is set to $\underline{s}[i][\min(x_i)]$. By iterating through the values of variable x_i from GETNEXT$(\min(x_i))$ up to $\max(x_i)$, we set $leq[v]$ to $\min(leq[\text{GETPREV}(x_i, v)], \underline{s}[i][v])$.
3. [COMPUTESUFFIX] For each value $v \in D(x_i)$, the quantity $\underline{s}(x_i, v)$ is set to $\min(gt[v], leq[v] + 1)$.

INITDATASTRUCTURE and all the calls to COMPUTESUFFIX can be done in $O(|D(x_i)|)$ time. Regarding the state of the art, no explicit algorithm achieving GAC is available for CHANGE$(N, X, \leq)$, but using the SLIDE constraint [8] leads to a time complexity of $O(nd^4)$ since the sliding constraint involves four variables (i.e., similarly to the encoding of CARDPATH with SLIDE [12, page 30], two extra variables are needed for counting the number of times $x_i \neq x_{i+1}$ is satisfied).

◇ SMOOTH(N, X, cst): SEQ_BIN where C is '$|x_i - x_{i+1}| > cst$' and B is '`true`'. For any value $v \in D(x_i)$, we need to evaluate the quantity $\min_{w \in D(x_{i+1})}(\min_{[|v-w| \leq cst]}(\underline{s}(x_{i+1}, w)), \min_{[|v-w|>cst]}(\underline{s}(x_{i+1}, w)) + 1)$. The question is how to efficiently evaluate the terms $\min_{[|v-w| \leq cst]}(\underline{s}(x_{i+1}, w))$ and $\min_{[|v-w|>cst]}(\underline{s}(x_{i+1}, w))$. For this purpose, the data structures we need for a column i consist of two sparse arrays $leq[]$ and $gt[]$: (1) $leq[v]$ is the minimum value over entries $\underline{s}[i][w]$, $w \in [\max(v - cst, \min(x_i)), \min(v + cst, \max(x_i))]$, (2) $gt[v]$ is the minimum value over entries $\underline{s}[i][w]$, $w \in [\min(x_i), v]$, $w \notin [\max(v - cst, \min(x_i)), \min(v + cst, \max(x_i))]$. The question which remains to solve is: given a column i, how to initialize the sparse arrays $leq[]$ and $gt[]$ in $O(|D(x_i)|)$ time complexity? Let us first focus on the $leq[]$ array. The set of intervals $[\max(v - cst, \min(x_i)), \min(v + cst, \max(x_i))]$, $v \in D(x_i)$, defines a set of sliding windows for which both the starts and the ends are increasing sequences (not necessarily strictly). The *ascending minima algorithm* (see `http://home.tiac.net/~cri/2001/slidingmin.html`) can be used for this purpose. Its time complexity is linear in the number of elements

in $leq[]$. Initializing the $gt[]$ array can be done by taking advantage of the following observations. All the intervals located before (resp. after) the intervals $[\max(v - cst, \min(x_i)),\ \min(v + cst, \max(x_i))]$, $v \in D(x_i)$, correspond to nested intervals starting a position $\min(x_i)$ (resp. ending at position $\max(x_i)$). Consequently, their minimum can be evaluated in one single scan over the values in $D(x_i)$. Finally, each entry $gt[v]$ is initialized by taking the minimum value among the windows corresponding to intervals $[\min(x_i), v - cst - 1]$ and $[v + cst + 1, \max(x_i)]$. Regarding the state of the art, no algorithm achieving GAC is available for Smooth, but using the Slide constraint [8] leads to a time complexity of $O(nd^4)$ since, as for the Change($\leq$) constraint, the sliding constraint involves four variables.

$\diamond$ Increasing_Nvalue(N, X): seq_bin where C is '=' and B is '$\leq$'. For any value $v \in D(x_i)$, we need to evaluate the quantity $\min_{w \in D(x_{i+1})}(\underline{s}(x_{i+1}, v), \min_{[v<w]}(\underline{s}(x_{i+1}, w)) + 1)$. The question is how to efficiently evaluate the term $\min_{[v<w]}(\underline{s}(x_{i+1}, w))$. Similarly to what was done for Change($N, X, \leq$), we create a sparse array $lt[]$, where $lt[v]$ is the minimum value over entries $\underline{s}[i][w]$, $w \in]v, \max(x_i)]$. The services InitDataStructure and ComputeSuffix can be encoded similarly to what was done for Change($N, X, \leq$), with the same worst-case complexity $O(|D(x_i)|)$. Our generic approach has the same time complexity as the dedicated algorithm described in [4].

$\diamond$ Among: seq_bin where C is notIn_bin($\mathcal{V}$) and B is '`true`'. For any value $v \in D(x_i)$, we need to evaluate the quantity $\min_{w \in D(x_{i+1})}(\text{Check}(\text{notIn\_bin}(\mathcal{V}), x_i, x_{i+1}) + \min(\underline{s}(x_{i+1}, w)))$. For this purpose AllocDataStructure allocates, without initializing it, an item min_1 that will be set by InitDataStructure to the minimum of the i^{th} column of matrix $\underline{s}[][]$. Finally, for each value $v \in D(x_i)$, ComputeSuffix sets the quantity $\underline{s}(x_i, v)$ to Check(notIn_bin($\mathcal{V}$), x_i, x_{i+1}) + min_1. This can be done in $O(|D(x_i)|)$ time, by iterating through the values of $D(x_i)$ with the getNext primitive. The generic algorithm described in this paper for Among has the same time complexity as the dedicated algorithm described in [7]. With respect to Increasing_Among there is no algorithm achieving GAC, but again using the Slide constraint [8] leads to a time complexity of $O(nd^4)$ for the same reason as the one mentioned for Change($N, X, \leq$). Finally, notice that the case of the Increasing_Among constraint (i.e. a seq_bin where C is notIn_bin($\mathcal{V}$) and B is '$\leq$') is directly derived from Among and has the same time complexity.

7 Conclusion

Our main contribution is a structural characterization of a class of counting constraints for which we come up with a GAC filtering algorithm which is linear in the sum of domain sizes. A still open question is whether it would be possible or not to extend this class (i.e., go beyond neighborhood-substitutability) without degrading complexity or giving up on GAC, in order to capture more constraints.

References

1. N. Beldiceanu. Pruning for the *minimum* Constraint Family and for the *number of distinct values* Constraint Family. In *proceedings of CP'01*, pages 211–224, 2001.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog. Technical Report T2005-08, Swedish Institute of Computer Science, 2005.
3. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Math. Comput. Modelling*, 20(12):97–123, 1994.
4. N. Beldiceanu, F. Hermenier, X. Lorca, and T. Petit. The *increasing nvalue* Constraint. In *Proc. CPAIOR*, 2010.
5. C. Berge. *Graphs and hypergraphs*. Dunod, Paris. 1970.
6. C. Bessière. Constraint Propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.
7. C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. Among, common and disjoint constraints. In *Proceedings CSCLP*, pages 29–43, 2005.
8. C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. SLIDE: A useful special case of the CARDPATH constraint. In *Proc. ECAI*, 2008.
9. COSYTEC. *CHIP Reference Manual*, release 5.1 edition, 1997.
10. E. C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *AAAI*, pages 227–233, 1991.
11. L. Hellsten. Consistency propagation for *stretch* constraints. Master's thesis, Waterloo University, 2004.
12. A. Papadopoulos. The *slide* Constraint, Propagation Algorithms and Applications. Master's thesis, Montpellier II University, 2006.
13. G. Pesant. A Filtering Algorithm for the *Stretch* Constraint. In *CP'01*, volume 2239 of *LNCS*, pages 183–195, 2001.