

ACADÉMIE DE MONTPELLIER
UNIVERSITÉ MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

MÉMOIRE DE STAGE DE MASTER

SPÉCIALITÉ : **Recherche en Informatique**
Mention : **Informatique, Mathématiques, Statistiques**

effectué au laboratoire LIRMM/INFO

—
sous la direction de DR. CHRISTIAN BESSIÈRE

The SLIDE constraint
Propagation algorithms and applications

par

Alexandre PAPADOPOULOS

Soutenu le 20 juin 2006

Table of Contents

1	Introduction	1
1.1	Global constraints	1
1.1.1	Description	1
1.1.2	Difficulties	2
1.2	A language for global constraints	2
1.2.1	The challenge	2
1.2.2	The first steps	3
1.3	The SLIDE constraint	3
2	General definitions	4
2.1	Type of variables	4
2.2	Constraint satisfaction problem	4
2.3	Consistency property	5
2.4	Consistency comparison	5
2.5	Notations	5
3	The fundamentals of SLIDE	7
3.1	A first definition	7
3.2	Examples	7
3.3	General complexity issues	8
4	Propagation algorithms	10
4.1	The basic scheme	10
4.2	Use of automata	12
4.2.1	The algorithm	12
4.2.2	Complexity	15
4.3	Enhancing space complexity	16
4.3.1	The general algorithm	16

4.3.2	Filtering	18
4.3.3	Propagating modifications	19
4.3.4	Use of automata	19
4.3.5	The automaton algorithm	21
5	Extensions of SLIDE	24
5.1	Multiple sequences	24
5.1.1	Example	24
5.1.2	Definition	25
5.1.3	Complexity	25
5.2	Merging sequences	25
5.2.1	Two sequences of same length	26
5.2.2	Two sequences of different length	26
5.2.3	Filtering algorithms for the multisequence SLIDE	27
5.3	Non-sliding variables	27
5.3.1	Definition and Complexity	27
5.3.2	Propagating	28
6	Examples and comparing with other approaches	29
6.1	Examples	29
6.2	Conjunction of constraints	30
6.3	Beldiceanu's extended automata	31
6.3.1	From automaton to Slide	32
6.3.2	Some examples	33
6.4	The REGULAR constraint	35
6.5	Graph characteristics	35
7	Experiments	36
8	Conclusion	37

1 Introduction

Constraint Programming (CP) has gained wide acceptance as an expressive way of modelling great combinatorial problems, on the one hand, while offering on the other hand a powerful means to resolve such problems. The power of this method lies to a great extent in its declarative aspect : a user posts some constraints on variables, modelling this way his problem. Then he expects this problem to be solved, regardless of how this is performed.

One of the main methods used to reason with constraints, if we focus on resolution aspects, is to try to compute and maintain a certain level of consistency, with regard to the constraints, within the domains. Filtering algorithms are responsible for pruning away of the domains the values that cannot belong to any solution given a certain constraint, and therefore to any solution at all. These algorithms are also called propagation algorithms, as the fact that a value is not allowed for a variable is propagated to the domain of other variables. This step is very important for resolution because when an inconsistent value is removed, all of the instantiations that contained this value will never be considered.

1.1 Global constraints

1.1.1 Description

Global constraints have a key importance in CP, and have participated to a great extent in its success. In a first approach, these are constraints simply holding on any unbounded number of variables. But we can refine this definition.

From an expressiveness point of view, one global constraint, instead of many binary (or at least of bounded arity) constraints, allows a user to express a general and precise pattern over a group of variables, somehow introducing an extra information on them. Additionally, when propagating a global constraint, an associated algorithm potentially allows that more values are pruned than if considering a logically equivalent decomposition with bounded arity constraints. Thus, the resolution performances are enhanced. We could even state, for some global constraints, that no decomposition exists such that the same amount of values can be pruned.

The typical example is the `ALLDIFFERENT` constraint (first introduced in [DVSA88]), which states that n variables should have a pairwise different value. For

example, with the variables X_1, X_2 and X_3 , taking their values in $\{1, 2\}$, $\{1, 2\}$ and $\{1, 2, 3\}$, if one considers an ALLDIFFERENT constraint over them, we can infer that 1 and 2 are forbidden values for X_3 . Whereas if one considers three binary inequalities, nothing can be inferred, precisely because there is a lack of a global vision. The ALLDIFFERENT constraint is not known to be decomposable or not : a decomposition in bounded arity constraints of polynomial size (number of variables, of constraints, domain sizes) that can prune the same amount of values has not been found, even though it has not been shown that such a decomposition cannot exist.

1.1.2 Difficulties

Although of a precious need, global constraints present some problems. As a result of their expressive specificity, a very big number of global constraints exist. In the Global Constraint Catalog [Bel05], Beldiceanu counts 235 of them. This induces first a modelling problem, regarding the user. How to choose, between such an amount of constraints, the more relevant for the considered problem ? In what semantically very close constraints differ ? Do they ? These are questions that come to a user new in Constraint Programming.

Furthermore, developing a propagation algorithm for a given global constraint is definitely not a easy task. Additionally, the consistency strength achieved by a filtering algorithm may vary from an constraint to another, such that the choice of a global constraint may also depend on efficiency issues. The ALLDIFFERENT constraint was one of the first global constraints for which a specific propagation algorithm was developed ([KR92] and then [Rég94]). But in the [Bel05] catalog, not all the global constraints have a propagation algorithm, and for those having one, the level of consistency achieved is not always clearly stated. To sum up, the somehow lack of unification within the global constraint framework is an obstacle to making CP easier to use.

1.2 A language for global constraints

1.2.1 The challenge

As a solution to the previous observations, a unifying approach has been proposed : establishing an as reduced as possible language of global constraints. The aim is to be able to identify a kernel of global constraints constituting a common denominator of all already existing global constraints.

This language of constraints must provide a way of expressing the biggest possible number of global constraints. As a result, a user who wants to take advantage of global constraints to express his problem does not have to go through a tedious process of learning a huge number of global constraints.

On the other hand, once having specific propagation algorithms for these kernel constraints, this leads automatically to having a propagation algorithm for any global constraint expressed within this language. The goal is that this algorithm gets as closer

as possible to an algorithm specifically designed for the considered global constraint, in computational efficiency as well as in the level of consistency achieved. In a perfect world, it would even be equivalent.

1.2.2 The first steps

In [BHH⁺05], Bessière *et al.* introduced two global constraints : the RANGE and the ROOTS constraints. They serve to specify counting and occurrences problems. They allow to express *and* to propagate some seventy out of the two hundreds or so global constraints. The next step has then been to study those constraint of the catalog that could not be expressed by using these two constraints. As a result, the SLIDE constraint has been proposed as the next element of the language.

1.3 The SLIDE constraint

Among the global constraints that could not be expressed with RANGE and ROOTS, a very frequent pattern occurred : that of a sequence property. This has been formalized by defining the SLIDE constraint, which expresses such a property. This report will concentrate on precisely determining the conditions where generalized arc-consistency can be polynomially achieved for any constraint expressed as SLIDE, how it can be, and finally how can in practice global constraints be expressed using the SLIDE constraint.

2 General definitions

Before continuing any further, let us first introduce the basic definitions and notations that will be used throughout the rest of this report.

2.1 Type of variables

There are different *type of variables*:

- Integer variable: variable which value ranges over a finite set of integer values, its domain. The domain can be either represented extensively, or by giving a minimum and maximum value, without any additional information about intermediate values. If X is an integer variable, $D(X)$ refers to its domain.
- Set variable: variable which value is a set of integer values. For efficiency's sake, the domain of such a variable is given by its lower and upper bound. If S is a set variable, $lb(S)$ contains all the definite values (that is the intersection of all the possible sets that can be assigned to S), and $up(S)$ the definite and potential values (that is union of all possible sets).
- Characteristic function: a set variable can be represented by its characteristic function, a boolean function stating for each value if it belongs to the set currently assigned to S . So a set variable can be represented by a sequence of boolean variables, one for each element of $up(S)$, a variable in this sequence being 1 iff the corresponding element is in the set assigned to S .

2.2 Constraint satisfaction problem

A *constraint satisfaction problem* instance is composed of a set of variables, each one having its domain, and a set of constraints. A *constraint* is a relation between some variables, should they be integer variables, set variables or both, restricting the values they are allowed to take. A *global constraint*, as it has been detailed in the introduction, is a constraint ranging over an unbounded number of variables. The variables of a constraint are also called the *scope* of the constraint. An *instanciation* of variables is given by a tuple of the corresponding values, with the order of the variables being

implicit in the context (for example the scope of a constraint, the index of the variables, etc.).

2.3 Consistency property

Given a constraint C , a *bound support* is a tuple that assigns to each integer variable a value between the minimum and the maximum and to each set variable a set between its lower and upper bound (with regard to the inclusion relation), and that satisfies C . A *hybrid support* is a bound support that assigns to each integer variable a value in its domain (nothing changes for set variables). A hybrid support that only contains integer variables is simply a *support*.

A constraint C is *bound consistent* (BC) iff for each integer variable, its minimum and maximum value belong to a bound support, and for each set variable, its lower bound is included in all bound supports and its upper bound in at least one. The constraint is *hybrid consistent* if for each integer variable X , every value in $D(X)$ belongs to a hybrid support. If C is hybrid consistent and only involves integer variables, then it is *generalized arc consistent* (GAC). Let us remark that enforcing HC on a constraint that involves set variables is equivalent to enforcing GAC if the set variables are represented by their characteristic function. The given definition of GAC is a generalization to global constraints of the definition of arc-consistency for binary constraints. But later in this report, it may happen that GAC be also called, informally, arc-consistency.

2.4 Consistency comparison

Consistency properties can be compared. Let P and Q be two consistency properties over a constraint C . P is as strong as Q iff for any domain if P holds then Q also holds. P is stronger than Q (or Q is weaker than P) iff for any domain P is as strong as Q , but not vice versa. P and Q are equivalent iff P is stronger than Q and vice versa. They are incomparable if neither P is as strong as Q , neither vice versa.

A propagation or filtering algorithm for a constraint is thus a procedure that enforces a certain consistency property to hold on this constraint, by removing in this order some values in the domains. If GAC or HC is \mathcal{NP} -Hard to enforce on a certain global constraint, then a weaker consistency property, such as BC, may be polynomially enforced on this constraint. We can say informally that a weaker level of consistency or a weaker propagation if performed, or even that the domains are less filtered.

2.5 Notations

In the notations used for global constraints, I will distinguish *variables and parameters*. Variables are those whose domain will be modified when enforcing consistency, and that are initially uninstantiated. On the other hand, parameters are variables that determine the exact semantics of the constraint but do not model anything whatsoever.

They could be integer variables, set variables, or a constraint, but must by all means be instantiated and fixed from the beginning and remain unchanged. No filtering is performed on them. To clearly distinguish them, the following notation will be used for global constraints:

$$\text{GC}[P_1, \dots, P_m](X_1, \dots, X_n)$$

where P_j are the parameters and X_i the variables.

Lastly, from a purely syntactical point of view, the variables of a global constraint may be grouped in *sequences*. This may suggest that the variables of each sequence have particular common semantics. For example, the following global constraint have two sequences of n and m variables :

$$\text{GC}([X_1, \dots, X_n], [Y_1, \dots, Y_m])$$

3 The fundamentals of SLIDE

The SLIDE constraint expresses a global property that is in fact composed of the repetition of a same smaller property. This pattern is very frequent, and occurs whenever, in order to check a property on a sequence of variables, there is no need to take into account this sequence as a whole. A good counter-example of this is the ALLDIFFERENT constraint where each variable is linked to any other.

3.1 A first definition

Let us introduce a first way to express such a pattern, by considering the following global constraint.

$$\text{SLIDE}[C](X_1, \dots, X_n)$$

This constraint, which we can call the one-sequence SLIDE constraint, is satisfied iff the constraint C of arity $k < n$, given as a parameter, is always satisfied when we slide it along the sequence X_1, \dots, X_n . The constraint C will very naturally be referred to as the slid or sliding constraint.

More formally, let us denote C_i the C constraint when applied on X_i, \dots, X_{i+k-1} . Then we simply have the logical equivalence:

$$\text{SLIDE}[C] \equiv C_1 \wedge C_2 \wedge \dots \wedge C_{n-k+1}$$

The constraint C_i is called the i th occurrence of C . This notation and vocabulary will always be used thereafter.

3.2 Examples

Let us immediately illustrate this with some very simple examples. We consider variables with domain $\{a, b\}$.

- Let C be a ternary constraint which authorized tuple are aba, bab . Then the solutions of the $\text{SLIDE}[C](X_1, \dots, X_n)$ constraint are $ababab\dots$ and $bababa\dots$
- Let C be the SYMMETRICAL constraint of arity 4, that is, which authorized tuple are $aaaa, abba, baab, bbbb$. Then the solutions of $\text{SLIDE}[C](X_1, \dots, X_n)$ are $aaaaaa\dots$ and $bbbbbb\dots$

A more realistic example which can give a more accurate idea of the semantics of SLIDE may be given by the following problem. On the assembly line of a car production plant, for technical reasons, a particular option (for example a sunroof) cannot be installed on every car. This limitation is represented by a ratio p/q : for every q consecutive cars, at most p can have a sunroof. The problem is to find in which order to build cars under this limitation.

The n cars of the assembly line will be represented by n variables X_1, \dots, X_n . Each variable takes a boolean value stating if the corresponding car has a sunroof or not. Then, the problem is simply modelled by the constraint $\text{SLIDE}[C](X_1, \dots, X_n)$, with $C(X_i, \dots, X_{i+q-1}) \equiv \sum_{j=0}^{q-1} X_{i+j} \leq p$.

3.3 General complexity issues

The complexity results follow almost immediately from those of the CARDPATH constraint, first introduced in [BC01], which are found in [BHHW04]. The CARDPATH constraint is very similar to the SLIDE constraint, except that there is an additional integer variable N specifying the number of occurrences of C that must be satisfied (instead of all occurrences, that is $N = n - k + 1$). Let us review the different results that are therefore shown.

Theorem 3.3.1 *Enforcing GAC on $\text{SLIDE}[C](X_1, \dots, X_n)$, where the variables can be repeated, is \mathcal{NP} -Hard.*

The proof given uses a reduction from 3COL. It uses for C the binary not-equal constraint, and N has $n - 1$ for value, that is the maximum possible (as $k = 2$). In other words, the proof uses a particular case of CARDPATH, which is exactly the SLIDE constraint, so the same result holds.

Theorem 3.3.2 *Enforcing GAC on $\text{SLIDE}[C](X_1, \dots, X_n)$ is \mathcal{NP} -Hard even if enforcing GAC on C is polynomial and no variable is repeated in the sequence.*

Here, the proof given of the theorem for the CARDPATH constraint is not directly adaptable. Indeed, it uses the reduction from MAX2SAT, which is a relaxed version of 2SAT: does an assignment of n boolean variables exist that violates at most k clauses in a 2SAT formula? In the reduction, if all clauses are satisfied, all occurrences of C are satisfied. But for the SLIDE, we always want all occurrences to be satisfied, in which case the problem from which we are making the reduction is just 2SAT, which is polynomial.

So we will make the reduction from CARDPATH. Let $\text{CARDPATH}[C](N, [X_1, \dots, X_n])$ be a CARDPATH instance, with C of arity k . Let us build the following sequence of variables: there is an alternation of an X_i variable and a counter variable, namely $X_1, c, \dots, X_{k-1}, c_0, X_k, c_1, \dots, X_n, c_{n-k+1}$. c is a dummy counter variable following every variable X_i with $i < k - 1$. Let us now define the constraint C' . C' is a constraint of arity $2k$. It is satisfied iff: either the first variable is

a counter variable, or the first variable is X_i and X_i, \dots, X_{i+k-1} form a valid instantiation of C and $c_i = c_{i-1} + 1$, or X_i, \dots, X_{i+k-1} does not form a valid instantiation of C and $c_i = c_{i-1}$. Enforcing GAC on C' is clearly polynomial. The domain of c_0 is set to $\{0\}$ and the domain of c_{n-k+1} is set to $D(N)$. It is clear that there is a one to one correspondance between the solutions of $\text{CARDPATH}[C](N, [X_1, \dots, X_n])$ and $\text{SLIDE}[C'](X_1, c, \dots, X_{k-1}, c_0, X_k, c_1, \dots, X_n, c_{n-k+1})$. Thus, as enforcing GAC on the first is \mathcal{NP} -Hard, then enforcing GAC on the second is also \mathcal{NP} -Hard.

The only remaining case is when C has a bounded arity and no variable is repeated. This is polynomial. In other words, enforcing GAC is fixed-parameter tractable with respect to the arity of C . Algorithms to obtain GAC are given in the following sections.

4 Propagation algorithms

4.1 The basic scheme

Let us first introduce the very basic algorithm, that uses dynamic programming (see [Wag95] for discussion about the dynamic programming algorithmic scheme).

This algorithm builds a table T that, eventually, associates $\forall i = 1, \dots, n - k + 2$ and for each partial instantiation of size $k - 1$ on (X_i, \dots, X_{i+k-2}) the number $T[i, (v_i, \dots, v_{i+k-2})]$ of solutions of $\text{SLIDE}[C](X_1, \dots, X_n)$ containing this partial instantiation, if there is any. This is performed in two steps.

The first step goes from line 1 to 9. At the end of the i th iteration, $T[i, (v_i, \dots, v_{i+k-2})]$ contains the number of tuples on $D(X_1) \times \dots \times D(X_{i-1}) \times \{v_i\} \times \dots \times \{v_{i+k-2}\}$ satisfying the constraint $C_1 \wedge \dots \wedge C_i$.

During the second step, from line 10 to 18, TB is similarly computed. At the end of the i th iteration, $TB[i, (v_i, \dots, v_{i+k-2})]$ contains the number of tuples on $\{v_i\} \times \dots \times \{v_{i+k-2}\} \times D(X_{i+k-1}) \times \dots \times D(X_n)$ satisfying $C_i \wedge \dots \wedge C_{n-k+1}$.

Now that these two tables have been computed, T can be updated by multiplying each cell with its corresponding in TB . Thus, each $T[i, (v_i, \dots, v_{i+k-2})]$ contains the number of tuples on $D(X_1) \times \dots \times D(X_{i-1}) \times \{v_i\} \times \dots \times \{v_{i+k-2}\} \times D(X_{i+k-1}) \times \dots \times D(X_n)$ satisfying $\text{SLIDE}[C]$. Arc-consistency can then be easily computed.

This algorithm runs in $\mathcal{O}(nd^k f_C(k, d))$ time, where $f_C(k, d)$ is the complexity of checking an instantiation of C . When k is bounded (independently of n), this algorithm is therefore polynomial, whereas it is unsurprisingly exponential otherwise.

The structure T , that is initially computed so as to perform a first propagation, is easily maintainable during further domain modifications. When a value is removed from the domain of a variable, we just have to pick up all the partial supports containing this value, decrement their counter, and propagate backward and forward as long as their counter fall to zero.

The practical (time and space) complexity of this method clearly depends on the tightness of the C constraint, and of the resulting $\text{SLIDE}[C]$. We could possibly dramatically improve the practical complexity by assuming the existence of an arc-consistency procedure for C . For example, in the initialization of the first line, we could in this manner at least consider only the tuples that belong to a solution of C . Similarly, during the extension step, the arc-consistency procedure allows to consider only the relevant

values of $D(X_{i+k-2})$.

Algorithm 1: Slide propagation

Data: The constraint C of arity $k < n$, the variables $[X_1, \dots, X_n]$
Result: Their domains are GAC with regard to $\text{SLIDE}[C]([X_1, \dots, X_n])$

```

// Forward step
1 foreach  $(v_1, \dots, v_{k-1}) \in D(X_1) \times \dots \times D(X_{k-1})$  do
2    $T[1, (v_1, \dots, v_{k-1})] \leftarrow 1$ 
3 for  $i \leftarrow 2$  to  $n - k + 2$  do
4   foreach  $v_{i+k-2} \in D(X_{i+k-2})$  do
5     foreach  $T[i-1, (v_{i-1}, \dots, v_{i+k-3})]$  defined do
6       if  $C(v_{i-1}, \dots, v_{i+k-3}, v_{i+k-2})$  then
7         if  $T[i, (v_i, \dots, v_{i+k-2})]$  undefined then
8            $T[i, (v_i, \dots, v_{i+k-2})] \leftarrow 0$ 
9            $T[i, (v_i, \dots, v_{i+k-2})] \leftarrow T[i-1, (v_{i-1}, \dots, v_{i+k-3})]$ 
// Backward step
10 foreach  $(v_{n-k+2}, \dots, v_n) \in D(X_{n-k+2}) \times \dots \times D(X_n)$  do
11    $TB[n-k+2, (v_{n-k+2}, \dots, v_n)] \leftarrow 1$ 
12 for  $i \leftarrow n - k + 1$  to  $1$  do
13   foreach  $v_i \in D(X_i)$  do
14     foreach  $TB[i+1, (v_{i+1}, \dots, v_{i+k-1})]$  defined do
15       if  $C(v_i, v_{i+1}, \dots, v_{i+k-1})$  then
16         if  $TB[i, (v_i, \dots, v_{i+k-2})]$  undefined then
17            $TB[i, (v_i, \dots, v_{i+k-2})] \leftarrow 0$ 
18            $TB[i, (v_i, \dots, v_{i+k-2})] \leftarrow TB[i+1, (v_{i+1}, \dots, v_{i+k-1})]$ 
// T update
19 for  $i \leftarrow 1$  à  $n - k + 2$  do
20   foreach  $T[i, (v_i, \dots, v_{i+k-2})]$  defined do
21      $T[i, (v_i, \dots, v_{i+k-2})] \leftarrow TB[i, (v_i, \dots, v_{i+k-2})]$ 
22     if  $T[i, (v_i, \dots, v_{i+k-2})] = 0$  then
23        $\text{Remove } T[i, (v_i, \dots, v_{i+k-2})]$ 
// Domain update
24 for  $i \leftarrow 1$  to  $n - k + 2$  do
25    $T_i[X_i, \dots, X_{i+k-2}] \leftarrow \{(v_i, \dots, v_{i+k-2}) / T[i, (v_i, \dots, v_{i+k-2})] \text{ défini}\}$ 
26   Compute AC on } T_i ;          /* Updates only  $D(X_{i+k-2})$  if  $i > 1$  */

```

4.2 Use of automata

We saw the basic scheme is to compute and store an initial structure, without much regard to the initial time it could take, as long as future propagations are fast enough. We now present an alternative structure, automata, that could in practice as well reduce the stored size as speed up propagation.

The idea is to build the minimal deterministic finite state automaton that recognizes the set of solutions of $\text{SLIDE}[C]$. Let \mathcal{A} be this automaton, and $\mathcal{L}(\mathcal{A})$ be the language defined by \mathcal{A} . If t is a tuple on $D(X_1) \times \dots \times D(X_n)$, we denote by $w(t)$ the word of length n obtained by the concatenation of the values of t . \mathcal{A} is therefore the minimal deterministic automaton such that $\mathcal{L}(\mathcal{A}) = \{w(t)/t \text{ is a solution of } \text{SLIDE}[C]\}$. It thus recognizes only a finite number of words, of fixed length. Every transition corresponding to a partial word of length i will be said to be of *level* i , and corresponds to a potential instantiation of X_i . If this transition allows to go from state q to state q' with the value v , it will be denoted by (q, v, q') . The transition function is denoted by δ , that is $\delta(q, v) = q'$. The states q and q' are said to be respectively of level $i - 1$ and i .

4.2.1 The algorithm

This algorithm first computes this automaton, then, GAC can simply be deduced by projecting the values of the transitions of the automaton. For more details about computing and maintaining GAC basing on an automaton, we can check [Pes04] and [AFM02].

Algorithm 2: Slide propagation using automata

Data: The constraint C of arity $k < n$, the variables $[X_1, \dots, X_n]$
Result: Their domains are GAC with regard to $\text{SLIDE}[C]([X_1, \dots, X_n])$

- 1 $\mathcal{A}_C \leftarrow \text{const-automaton}(C)$; /* Automaton for C */
- 2 Make domains GAC w.r. to each C_i
- 3 $\mathcal{A} \leftarrow$ copy of the $k - 1$ first levels of \mathcal{A}_C
- 4 $\text{process}(\mathcal{A}, q_0, 0)$
- 5 $\text{minimize}(\mathcal{A})$
- 6 **for** $i \leftarrow 1$ **to** n **do**
- 7 $D(X_i) \leftarrow \{v/\exists(q, v, q') \in \mathcal{A} \text{ of level } i\}$

The recursive function `process` is the one that strictly speaking builds \mathcal{A} . At the end of the initial call of line 4, \mathcal{A} is the desired deterministic automaton, but it is not minimal. There are as much full paths as solutions in $\text{SLIDE}[C]$. As this automaton is computed as a preprocessing step and is kept for use in future propagations, it is useful

to minimize its size, by a classic automaton minimization procedure.

Function `const-automaton(C)` : automaton

Data: The constraint C
Result: A deterministic automaton recognizing the supports of C

```

1  $D_C \leftarrow$  empty domain on  $Y_1, \dots, Y_k$ 
2 for  $i \leftarrow 1$  to  $k$  do
3    $D_C(Y_i) \leftarrow \cup_{j=1}^n D(X_j)$  // Unified domains
4  $\mathcal{A}_C \leftarrow$  New automaton
5  $q_0 \leftarrow$  initial state of  $\mathcal{A}_C$ 
6 add-transition( $Y_1, q_0$ )
7 return  $\mathcal{A}_C$ 
8 def add-transition( $Y_i, q$ ) begin
9   foreach  $v_i \in D_C(Y_i)$  do
10      $q'_i \leftarrow$  new state
11     add the transition ( $q, v_i, q'_i$ )
12     if  $i < k$  then
13        $D_C(Y_i) \leftarrow \{v_i\}$ 
14       Save domains
15       Compute GAC
16       add-transition( $Y_{i+1}, q_i$ )
17       Restore domains
18 end
```

The `const-automaton` function builds a deterministic automaton \mathcal{A}_C for C . It performs this by simply making a backtrack-free resolution, using the GAC procedure associated to C . This auxiliary structure allows to optimize in the main algorithm the

repeating consistency checks.

Function $\text{process}(\mathcal{A}, q, i) : \text{boolean}$

Data: \mathcal{A} the automaton being built, q a state of level i of \mathcal{A} with $0 \leq i \leq n - k + 1$

Result: The sub-automaton induced by q recognizes all the tuples of $C_{i+1} \wedge \dots \wedge C_{n-k+1}$ on the domains defined by \mathcal{A} .

```

1 if  $i = n - k + 1$  then
2   | return true
3 else
4   |  $\text{extend}(\mathcal{A}, q, \varepsilon)$ 
5   |  $\text{added} \leftarrow \text{false}$ 
6   | foreach  $q' \in \text{succ}(q)$  do
7     | if  $\text{process}(q', i + 1)$  then
8       |  $\text{added} \leftarrow \text{true}$ 
9     | else
10    |  $\text{Remove the transition } (q, q')$ 
11  | return  $\text{added}$ 

```

Function $\text{extend}(\mathcal{A}, q, m) : \text{boolean}$

Data: \mathcal{A} the automaton being built, q a state of \mathcal{A} of level i , m a partial word of \mathcal{A}_C .

Result: The sub-automaton induced by q recognizes all the tuples of C_{i+1} on the domains defined by \mathcal{A} , returns *false* if there is not any, *true* if at least one transition has been added in level $i + k$

```

1 if  $\text{succ}(q) = \emptyset$  then
2   |  $q_C \leftarrow \delta_C(q_0, m)$ 
3   | foreach  $(v, q'_C) \in \text{succ}_C(q_C)$  do
4     |  $q' \leftarrow \text{new state}$ 
5     |  $\text{add transition } (q, v, q')$ 
6   | return true
7 else
8   |  $\text{added} \leftarrow \text{false}$ 
9   | foreach  $(v, q') \in \text{succ}(q)$  do
10  | if  $m \cdot v \in \mathcal{L}(\mathcal{A}_C) \wedge \text{extend}(q', m \cdot v)$  then
11  |   |  $\text{added} \leftarrow \text{true}$ 
12  | else
13  |   |  $\text{Remove transition } (q, v, q')$ 
14  | return  $\text{added}$ 

```

The process function has the following invariants:

- when called, the sub-automaton of \mathcal{A} induced by q (that is, which q is the initial state) is defined for only $k - 1$ levels, corresponding to the variables

$X_{i+1}, \dots, X_{i+k-1}$, thus inducing a certain domain on these variables. It contains only words of length $k - 1$.

- at the end of its execution, this sub-automaton has been extended on the remaining levels, so that it recognizes all the tuples of $C_{i+1} \wedge \dots \wedge C_{n-k+1}$ on the induced domain. If there is not any, this sub-automaton is empty, reduced to q , and returns *false*.

For this to be achieved, `extend` is called, which extends the sub-automaton to all of the variables of C_{i+1} , then `process` is called again recursively for each successor of q , thus maintaining the invariant. If `extend` returned *false*, q does not have anymore any successor, and therefore `process` returns *false*. Otherwise, if the recursive call returned *false*, q' has not any successor anymore, by induction hypothesis. The transition (q, q') is then removed. If all the recursive calls returned *false*, again q has not any successor anymore, in which case the function also returns *false*. Consequently, the invariant that all paths starting from q are full paths (ie they reach level n) is maintained. As \mathcal{A} is correctly initialized, the result after the initial call is actually what was expected.

4.2.2 Complexity

As for the dynamic-programming algorithm, this algorithm computes its structure in $d^{k-1}(n - k + 1)$ steps : it is obvious that this algorithm is linear in the size of the computed automaton. Let us now show that this automaton has a size (in the number of states) that is bounded by $d^{k-1}(n - k + 1)$.

Theorem 4.2.1 *The number of states and transitions of the minimal deterministic automaton that represents the solutions of $\text{SLIDE}[C]([X_1, \dots, X_n])$ is bounded by $d^{k-1}(n - k + 1)$.*

Proof This can be shown by induction. Let us first consider the C_1 constraint. We have one initial state of level 0. As there are at most d possible values for X_1 , there are at most d states of level 1. Repeating that for all the variables until X_{k-1} , it follows that there are at most d^{k-1} states of level $k - 1$. Now let us consider the states of level k . At first, we can say that there are at most d^k states, one for each of the at most d^k possible words of length k over d symbols. However, the only information needed from now on is the $k - 1$ suffixes of these words. As there are at most only d^{k-1} such suffixes, each one of them is repeated at most d times. Therefore, for each $k - 1$ suffix, the d states that recognize it can be merged into a single one, thus still having at most d^{k-1} states in the level k . Now, when considering the C_i constraint, assuming that there are at most d^{k-1} states of level $i + k - 2$, the same reasoning can be applied for the states of level $i + k - 1$.

This upper bound of course is never reached. Indeed, the only constraint C for which the bounds used in the reasoning are reached is the *true* constraint (that allows any instantiation to be valid). The corresponding $\text{SLIDE}[C]$ constraint is also the true

constraint, and the minimal automaton needed to represent it has of course only one state per level (and d transitions between two consecutive levels). More generally, for a given C , the automaton for the SLIDE[C] can be seen as a sub-automaton of the maximal automaton (that is where the transitions whose value do not hold are removed), on which minimization is a heuristic for compacting its size.

4.3 Enhancing space complexity

The basic algorithm that computes arc-consistency for the SLIDE constraint given a constraint C of arity k computes supported instantiations (*ie* consistent with the SLIDE) for every subsequence of size $k - 1$ of the SLIDE constraint. Then, generalized arc-consistency is obtained by applying AC for every such partial instantiation. This implies to store $n - k$ constraints of size at most d^{k-1} . Eventhough we can't hope much improvement concerning time complexity (as obtaining GAC for the SLIDE is \mathcal{NP} -Hard), the drawback of this method is that it doesn't take into account the fact that it is a unique C that is slid (it could perfectly work for a conjunction of the same number of constraints of same arity, but with each one having its own semantics). The following method uses a single constraint in which each tuple is associated to a counter from which GAC can be computed, thus improving space complexity.

4.3.1 The general algorithm

As we must deal with only one constraint for all variables, we define this constraint on k auxiliary variables, let Y_1, \dots, Y_k be these variables, whose domains are the union of the domain of all X_i variables. We then compute the list of tuples of this unified constraint, simply by a BT-free resolution on Y_1, \dots, Y_k . This will be the only information stored, along with an extra information consisting of a left and right marker for each tuple of C . These markers, referred to as m_l and m_r , are computed by the following algorithm, in $\mathcal{O}(n.d^k)$ time.

Note that the right markers are computed in this algorithm from left to right, which might seem confusing, but, as we will see later on, they serve to propagate from right

to left, hence their name.

Algorithm 6: Computing right markers

Data: The unified constraint C
Result: Right markers associated to the tuples of C

```

/* Initialization */
1  $S \leftarrow \{ \text{suffix}(t, k-1), t \in C \}$ 
2 foreach  $s \in S$  do
3    $\lfloor \text{support}(s) \leftarrow 0$  // The number of tuples having this suffix
4 foreach  $t \in C$  do
5    $\lfloor m_r(t) \leftarrow n - k + 1$ 
6    $\lfloor s \leftarrow \text{suffix}(t, k-1)$ 
7    $\lfloor \text{support}(s) \leftarrow \text{support}(s) + 1$ 
/* Computation */
8 for  $i \leftarrow 1$  to  $n - k$  do
9   foreach  $t \in C$  s.t.  $\text{prefix}(t, k-1) \notin S$  do
10   $\lfloor m_r(t) \leftarrow i$ 
11   $\lfloor s \leftarrow \text{suffix}(t, k-1)$ 
12   $\lfloor \text{support}(s) \leftarrow \text{support}(s) - 1$ 
13   $\lfloor \text{if } \text{support}(s) = 0$  then  $S \leftarrow S \setminus \{s\}$ 

```

The m_l markers are computed conversely : every prefix is changed to suffix and vice versa, and iterations are made from $n - k + 1$ to 1 instead.

The meaning of the right (resp. left) marker is : if $m_r(t) = i$ (resp. $m_l(t) = i$), then t is a valid partial instantiation of X_i, \dots, X_{i+k-1} with regard to $C_1 \wedge \dots \wedge C_i$ (resp. $C_i \wedge \dots \wedge C_{n-k+1}$) and is not a valid partial instantiation of X_j, \dots, X_{j+k-1} with regard to $C_1 \wedge \dots \wedge C_j$ with $i < j$ (resp. for $C_j \wedge \dots \wedge C_{n-k+1}$, with $j < i$).

The algorithm fulfils this specification. To show this, consider the following invariant : after line 8, at iteration i S contains all the suffixes that can be valid for (at least) i occurrences of C (that is for every element s of S a solution t of $C_1 \wedge \dots \wedge C_i$ exists with $s = t[i+1, \dots, i+k-1]$), and only those. Initially S contains all the suffixes of the tuples of C , so it is true for $i = 1$. Now let us assume it is true $i - 1$, $i > 1$. Let t be a tuple satisfying the condition of line 9, and $p = \text{prefix}(t, k-1)$. By induction hypothesis, p cannot be valid for $i - 1$ occurrences of C . Therefore if t was the last support of its suffix s , s cannot either be valid for i occurrences of C , and is consequently removed from S in line 13. Thus, the invariant is maintained. As it is easy to see that at any iteration the suffix of t is in S , we can conclude that the two conditions on the markers are satisfied.

Example Consider the following constraint C , on the unified domain $\{a, b\}$, with $k = 4$ and $n = 6$, and with the left and right markers as computed by the algorithm.

C	$m_l(t)$	t	$m_r(t)$
	1	$abab$	1
	3	$abba$	3
	2	$abbb$	3
	1	$babb$	2
	3	$bbba$	3

4.3.2 Filtering

Based on the previous definitions of the markers, new constraints are now defined : $C_i^r = \{t \in C \text{ s.t. } m_r(t) \geq i\}$ and $C_i^l = \{t \in C \text{ s.t. } m_r(t) \leq i\}$. From these definitions, filtering is obtained according to the following theorem.

Theorem 4.3.1 *Arc-consistency applied to $C_{n+k-1}^r, \dots, C_1^r$, followed by $C_1^l, \dots, C_{n-k+1}^l$, in this order, is equivalent to GAC on $\text{SLIDE}[C]$, whatever the domain of the variables is.*

Before proving this theorem, let us first consider some properties. We assume for the moment that *all domains are equal* (that is, that the domain of each variable is equal to the unified domain).

Let us remark that, within this assumption, it holds from the properties on the markers that $C_i^r \equiv C_1 \wedge \dots \wedge C_i|_{X_i, \dots, X_{i+k-1}}$ and $C_i^l \equiv C_i \wedge \dots \wedge C_{n-k+1}|_{X_i, \dots, X_{i+k-1}}$.

Proposition 4.3.2 *Arc-consistency applied to $C_{n+k-1}^r, \dots, C_i^r$, in this order, will make the domains of X_i, \dots, X_n arc-consistent with regard to $\text{SLIDE}[C]$, if all domains are equal.*

Proof The result holds for C_{n-k+1}^r by definition of this constraint. Now suppose that, for a given $i < n - k + 1$, the domains of X_{i+1}, \dots, X_n are AC with regard to $\text{SLIDE}[C]$. Now we apply AC to C_i^r . As the prefix of every tuple of C_{i+1}^r is the suffix of a tuple of C_i^r , only the domain of X_i is possibly changed, that is, restricted (because all domains were assumed to be initially equal). Now let v_i be a value in the modified domain $D(X_i)$. By definition of C_i^r , there is an instantiation (v_i, \dots, v_{i+k-1}) of X_i, \dots, X_{i+k-1} consistent with $C_1 \wedge \dots \wedge C_i$. As the values $v_{i+1}, \dots, v_{i+k-1}$ are also consistent by hypothesis with $\text{SLIDE}[C]$, it holds that a complete instantiation containing v_i consistent with $\text{SLIDE}[C]$ exists.

Corollary 4.3.3 *Arc-consistency applied to $C_{n-k+1}^r, \dots, C_1^r$ leads to GAC domains with regard to $\text{SLIDE}[C]$.*

Corollary 4.3.4 *Clearly, all the converse results hold for the C^l constraints.*

Proof of theorem 4.3.1 Let us now consider back the general case where *not all domains are equal*. The problem arises when the following occurs: suppose that, if starting with all domains equal, when filtering the domain of X_i by C_i^r , it gets a certain value D , and that the effective starting domain (that is before filtering) of X_i has a value D' which is strictly included in D . A value $v \in D \setminus D'$ is in fact not allowed and could have

pruned more values in the domains of $X_j, i < j$. However, continuing on down to C_1^r , the absence of the value v will be correctly propagated to the domains of $X_j, j < i$. Now, we start from left to right using corollary 4.3.4. We will not encounter the same situation, or more precisely, it will be without consequence. Indeed, when filtering the domain of X_{i+k-1} by C_i^l , with the same notations, the absence of a value v will have already been propagated to the domain of the previous variables during the first phase. It will also be propagated now to the domain of the next variables. So all the domains that were not completely pruned during the backward (or right to left) phase will get during this forward (or left to right) phase.

Example Let us consider again the example. The following table gives the solutions of the resulting $\text{SLIDE}[C]([X_1, \dots, X_6])$, where all initial domains are $\{a, b\}$, and the resulting GAC domains.

Now let the initial domain of X_1 be just $\{a\}$. The values of the variables used in the above demonstrations here are $D = \{a, b\}$, $D' = \{a\}$ and therefore $v = b$. During the first, backward, phase, the value b is not removed from the domain of X_3 . Indeed, that is because the C_3^r constraint detects that there is for the moment a partial path coming from right supporting $X_3 = b$, which is $bbba$, but the only complete path to which it can be completed is $babbba$, thus supporting $X_1 = b$. The fact that this instantiation is not allowed is only propagated during the forward phase.

	X_1	X_2	X_3	X_4	X_5	X_6
Solutions	a	b	a	b	b	a
	a	b	a	b	b	b
	b	a	b	b	b	a
Domains	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{b\}$	$\{b\}$	$\{a, b\}$
After backward phase	$\{a\}$	$\{b\}$	$\{a, b\}$	$\{b\}$	$\{b\}$	$\{a, b\}$
After forward phase	$\{a\}$	$\{b\}$	$\{a\}$	$\{b\}$	$\{b\}$	$\{a, b\}$

4.3.3 Propagating modifications

When the domain of some variable X_i is reduced, we propagate this modification in order to have GAC domains simply using the C^r constraints backward, and the C^l constraints forward, as long as some modification is done. It is enough as, by hypothesis (that domains were GAC before the modification of the domain), we will never encounter a situation where a value v as defined before exists.

4.3.4 Use of automata

We can use automata with this method to take advantage of their ability to represent in a compact way the solutions of a constraint. As several tuples can share common transitions, the markers in this structure must be situated on the level of transitions.

Let C be the unified constraint (that is, expressed as the list of allowed tuples on the unified domains). We associate to it the minimal deterministic automaton \mathcal{A} which

recognizes the set of solutions of C . Now once the markers of the tuples of C have been computed, a right and left marker is associated to each transition t of \mathcal{A} .

Definition For each transition a of \mathcal{A} , a left and a right marker, denoted by $m_l(a)$ and $m_r(a)$, are defined, that satisfy the following properties :

- let $t \in C$ and p be the path of \mathcal{A} that supports t , $m_r(t) = i$ iff $\forall a \in p, m_r(a) \geq i$;
- let $t \in C$ and p be the path of \mathcal{A} that supports t , $m_l(t) = i$ iff $\forall a \in p, m_l(a) \leq i$;
- the values of the markers are optimal (that is minimal for right markers and maximal for left ones) : for a tuple t of C of right marker i , there is a transition a in its corresponding path of right marker $j > i$ iff there is a tuple $t' \in C$ of right marker j whose corresponding path contains a , and conversely for left markers.

Now we can define, $\forall i = 1, \dots, n - k + 1$, the automaton \mathcal{A}_i^r as the restriction of \mathcal{A} to the transitions a s.t. $m_r(a) \geq i$, and the automaton \mathcal{A}_i^l as the restriction of \mathcal{A} to the transitions a s.t. $m_l(a) \leq i$. Note that according to the optimality condition on the markers, every such automaton is consistent (*ie* there are only complete paths). It thereby holds that

$$t \in C_i^r \Leftrightarrow w(t) \in \mathcal{L}(\mathcal{A}_i^r) \text{ and } t \in C_i^l \Leftrightarrow w(t) \in \mathcal{L}(\mathcal{A}_i^l)$$

(where $w(t)$ is the word obtained from the concatenation of the symbols in the tuple t).

We thus have two equivalent structures, so all the previous methods still work, with the second, the automaton, having a possibly much better practical space complexity. However, it would not be worth using such a structure if in order to calculate it we had first to pass through the list-of-tuples method. Let us see how we can directly compute the automaton and its corresponding markers.

Example Figure 4.1 presents an example for the previously introduced constraint. The automaton \mathcal{A} represents the set of solutions of C , and on each transition are also present the left and the right marker.

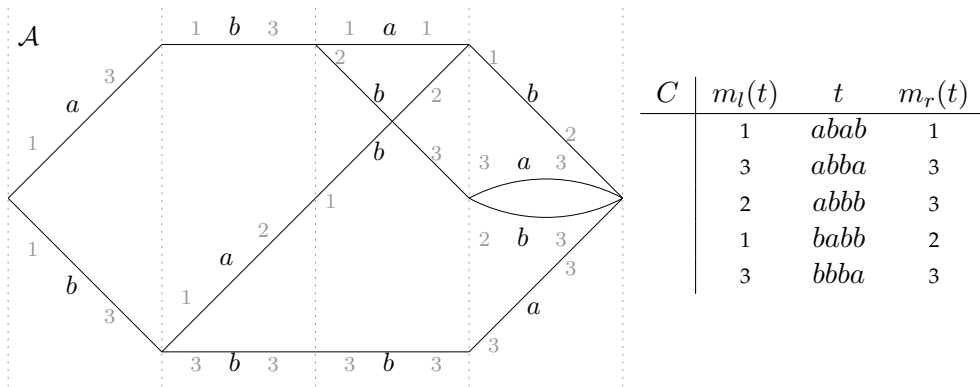


Figure 4.1: An example for a constraint C with $k = 4, n = 6$

4.3.5 The automaton algorithm

We first have to build the unified automaton \mathcal{A} , and then compute the left and right markers associated to each transition. For this to be performed, two auxiliary automata \mathcal{A}_P and \mathcal{A}_S are built from \mathcal{A} , representing respectively the $k - 1$ prefixes and the $k - 1$ suffixes of the words of \mathcal{A} . The data structures must be able to provide the transition of \mathcal{A}_P corresponding to each transition in \mathcal{A} of level $< k$, and the transition of \mathcal{A}_S corresponding to each transition in \mathcal{A} of level > 1 . Conversely, they must provide for each transition of \mathcal{A}_P and \mathcal{A}_S the corresponding transitions of \mathcal{A} (that is the inverse association). This association will be represented thanks to the sets $t_{\mathcal{A}_P}(a)$, $t_{\mathcal{A}_S}(a)$ and $t_{\mathcal{A}}(a)$, where a is a transition, the first two having a size 0 or 1, according the level of a , the third having a size at least 1.

These automata are simply computed by copying \mathcal{A} (and storing the transition association in both ways) and removing respectively the last and first level. Then for \mathcal{A}_P minimization is performed from last to first level and for \mathcal{A}_S determinization is performed from first to last level. Whenever two transitions a and a' are merged into a new transition a'' , we set $t_{\mathcal{A}}(a'')$ to $t_{\mathcal{A}}(a) \cup t_{\mathcal{A}}(a')$, and, if for example we are dealing with \mathcal{A}_P , the set $t_{\mathcal{A}_P}$ of each transition of $t_{\mathcal{A}}(a'')$ is set to $\{a''\}$.

Example For the previous automaton \mathcal{A} , the \mathcal{A}_P and \mathcal{A}_S automata would look like this.

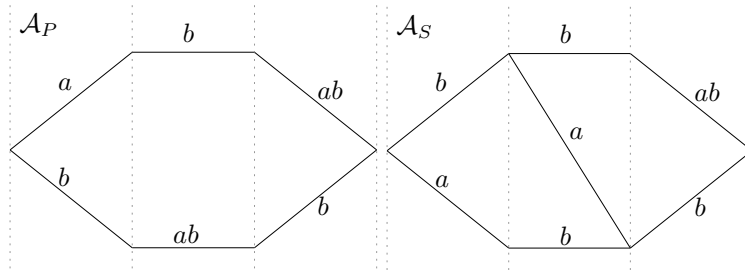


Figure 4.2: The prefix and suffix automata

Now the left and right markers can be computed, both similarly. Only the algorithm for the right markers is given, similarly with the first one. However, the algorithm for left markers is obtained exactly conversely : every P is changed to S and vice-versa, every r is changed to l , every $n - k + 1$ is changed to 1 and vice-versa, and eventually iteration order is inverted (that is $i \leftarrow i + 1$ is changed to $i \leftarrow i - 1$ and $i < n - k + 1$ to

$i > 1$).

Algorithm 7: Computing right markers for \mathcal{A}

Data: The unified automaton \mathcal{A} , the auxiliary automata \mathcal{A}_P and \mathcal{A}_S

Result: Right markers associated to the transitions of \mathcal{A}

// Initialization

```

1 foreach  $a \in \text{trans}(\mathcal{A})$  do
2    $m_r(a) \leftarrow n - k + 1$ 
// Computation
3  $i \leftarrow 1$ 
4  $\text{changed} \leftarrow \text{true}$ 
5 while  $i < n - k + 1 \wedge \text{changed}$  do
6    $q_P \leftarrow$  initial state of  $\mathcal{A}_P$ 
7    $q_S \leftarrow$  initial state of  $\mathcal{A}_S$ 
8    $\text{changed} \leftarrow \text{explore}(q_P, q_S, i)$ 
9    $i \leftarrow i + 1$ 

```

Function $\text{explore}(q_P, q_S, i)$

```

1  $\text{changed} \leftarrow \text{false}$ 
2 foreach  $(v, q'_P) \in \text{succ}(q_P)$  do
3    $a' \leftarrow (v, q'_P)$ 
4   if  $q'_S \leftarrow \delta_S(v, q_S)$  then // Is there any such transition?
5      $\text{changed} \leftarrow \text{explore}(q'_P, q'_S)$ 
6   else
7     foreach  $a \in t_{\mathcal{A}}(a')$  do
8        $q \leftarrow \text{in}(a)$ 
9        $q' \leftarrow \text{out}(a)$ 
10       $\text{mark}(a, i)$ 
11       $\text{mark-backward}(q, i)$ 
12       $\text{mark-forward}(q', i)$ 
13     $\text{changed} \leftarrow \text{true}$ 
14 return  $\text{changed}$ 

```

This algorithm starts by initializing all transitions to $n - k + 1$. Then it iterates from 1 to $n - k$ to possibly restrict the validity of an arc, that is, to mark it with a value less than $n - k + 1$. For that purpose the `explore` function explores in parallel \mathcal{A}_P and \mathcal{A}_S and marks those arcs in \mathcal{A}_P that are not in \mathcal{A}_S . These arcs support a prefix that is not in the current set of suffixes (recognized by \mathcal{A}_S), thus corresponding to the condition in line 9 of algorithm 6. After a transition is marked by i , thanks to the `mark` function, the information is propagated to every arc that for sure cannot belong to any \mathcal{A}_j^r with $i < j$, in order to remain consistent with the optimality condition of the definition of the markers. In addition, each time a transition is marked, the information is update in \mathcal{A}_P and \mathcal{A}_S so as to remove any transition supporting for example a suffix that is

not supported by anyone of the tuples recognized by the current \mathcal{A}_i^r , similarly to what is operated in the first algorithm. The only difference between the two algorithms is that the second operates in the level of a transition rather than of a whole tuple, in compliance with the definitions of the automaton markers.

Function mark(a, i)

```

1  $m_r(a) \leftarrow i$ 
2 foreach  $a' \in t_{\mathcal{A}_P}(a)$  do
3   Remove  $a$  from  $t_{\mathcal{A}}(a')$ 
4   if  $t_{\mathcal{A}}(a') = \emptyset$  then Remove transition  $a'$  from  $\mathcal{A}_P$ 
5 foreach  $a' \in t_{\mathcal{A}_S}(a)$  do
6   Remove  $a$  from  $t_{\mathcal{A}}(a')$ 
7   if  $t_{\mathcal{A}}(a') = \emptyset$  then Remove transition  $a'$  from  $\mathcal{A}_S$ 

```

Function mark-backward(q', i)

```

1 if  $\gamma^+(q') = 1$  in  $\mathcal{A}_i^r$  then
2   foreach  $a \in \text{pred}(q')$  do
3      $q \leftarrow \text{in}(a)$ 
4     mark( $a, i$ )
5     mark-backward( $q, i$ )

```

Function mark-forward(q, i)

```

1 if  $\gamma^-(q) = 1$  in  $\mathcal{A}_i^r$  then
2   foreach  $a \in \text{succ}(q)$  do
3      $q' \leftarrow \text{out}(a)$ 
4     mark( $a, i$ )
5     mark-forward( $q', i$ )

```

This algorithm processes at most once each transition of \mathcal{A} and its corresponding transition in \mathcal{A}_S and \mathcal{A}_P . On the other hand, the \mathcal{A}_P and \mathcal{A}_S structures and \mathcal{A} are isomorphic : the size of \mathcal{A}_P , including the $t_{\mathcal{A}}$ sets, is equal to the size of the first $k - 1$ levels of \mathcal{A} , and similarly for \mathcal{A}_S . Each iteration has a complexity linear in the size of \mathcal{A}_P and \mathcal{A}_S , and consequently of \mathcal{A} . The size of \mathcal{A} being bounded by d^k , this algorithm has a complexity of $\mathcal{O}(n \cdot d^k)$, which was also the complexity of algorithm 6.

5 Extensions of SLIDE

Until now, we introduced a very basic version of the SLIDE constraint. A certain number of algorithms have been developed to propagate this constraint. Eventhough these algorithms are not that obvious to obtain, this form of the constraint has so far little expressivity. Let us now introduce some extensions that add expressivity, while still allowing to easily readapt or even better reuse the existing algorithms.

5.1 Multiple sequences

This is the most useful case, where the constraint C is slid simultaneously on more than one sequence of variables. Let us illustrate it with an example.

5.1.1 Example

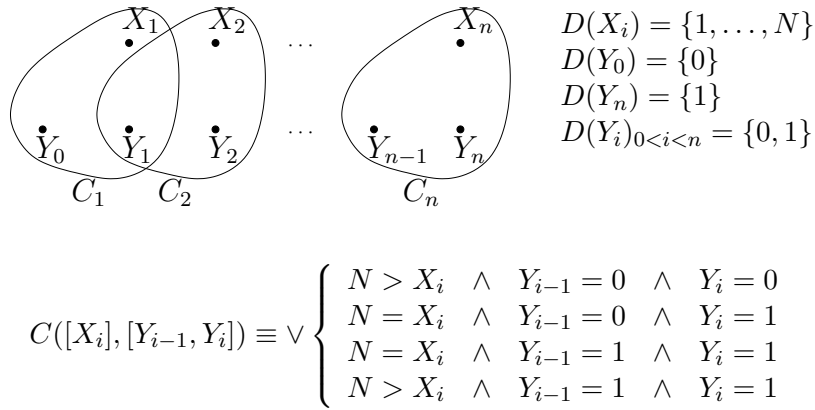


Figure 5.1: The constraint SLIDE[C] coding the constraint MAXIMUM

Figure 5.1 illustrates an example of using a multisequence SLIDE for the constraint MAXIMUM[N]($[X_1, \dots, X_n]$). This constraint is satisfied if N , a fixed given parameter, is the biggest value taken in the sequence of variables X_1, \dots, X_n . In the given coding in SLIDE, for any instantiation of the variables $X_1, \dots, X_n, Y_0, \dots, Y_n$ such that the constraint C is satisfied for each of its occurrences as specified, with $Y_0 = 0$ and $Y_n = 1$, the

sub-instanciation of the X_i variables only is a solution of $\text{MAXIMUM}[N]$. Conversely, to any solution of $\text{MAXIMUM}[N]$ corresponds an instanciation of the Y_i variables such that the whole instanciation is a solution of $\text{SLIDE}[C, N]$. Indeed, the values of the variables Y_i just indicate if the value N has been encountered in the X_i variables or not. Let us remark that the MAXIMUM constraint is in fact decomposable, as we can observe that the intersection size of the occurrences of C is only one, so that GAC on each C_i is equivalent to GAC on MAXIMUM .

5.1.2 Definition

Generally speaking, an instanciation of a multisequence SLIDE holds on s sequences, and each sequence j with $1 \leq j \leq s$ contains n_j variables. The variables of each sequence are indexed such that the last variable of each sequence is indexed with the same value, or said the other way around, the sum of the value of the first index of the sequence and n_j must always be the same. If fixing $n = n_1$, the sequence j with $1 \leq j \leq s$ is $[X_{1+n-n_j}^j, \dots, X_n^j]$. The constraint C holds on a subsequence of k_j variables of each j sequence, with the relation that $n_j - k_j$ be the same for each sequence. By fixing $k = k_1$, the general form of C is :

$$C_i([X_i^1, \dots, X_{i+k-1}^1], \dots, [X_{i+k-k_j}^s, \dots, X_{i+k-1}^s])$$

with $1 \leq i \leq n - k + 1$ (the number of occurrences of C).

In our example, we thus have $s = 2$ (two sequences), $n = n_1 = 3$ (the three variables X_1, X_2, X_3), $n_2 = 4$ (the four variables Y_0, \dots, Y_3), $k_1 = 1$ and $k_2 = 2$, with $X^1 = X$ et $X^2 = Y$.

5.1.3 Complexity

The complexity results here are exactly the same as for the one-sequence version of the SLIDE constraint. The important parameter is the intersection size of the set of variables of two consecutive occurrences of the slid constraint C . Let K be this number, it holds that $K = \sum_{j=1}^s (k_j - 1)$. If K is unbounded, obtaining GAC for the multi-sequence SLIDE is of course \mathcal{NP} -Hard, as it has been shown for the one-sequence SLIDE , which is only a particular case. If K is bounded, obtaining GAC is polynomial, with, similarly with the one-sequence case, fixed parameter complexity algorithms in d^K , as shown in next section. Let us remark that the number s of sequences do not have a direct incidence : we can add as many sequences as we want, as long as the parameter k_j of each one of them is worth 1 it does not change the value of K . The fact that the previously introduced MAXIMUM constraint is decomposable is a consequence of this.

5.2 Merging sequences

Let us show how to systematically convert a multisequence SLIDE so as to reduce the number of sequences used. This process can be referred to as merging two sequences

into one.

5.2.1 Two sequences of same length

When two sequences have the same length, it is easy to merge them into a single one. Let us consider the instance $\text{SLIDE}[C]([x_0, \dots, x_n], [y_0, \dots, y_n])$, with $C([x_{i-1}, x_i], [y_{i-1}, y_i])$. This constraint can perfectly be redefined as a SLIDE on only one sequence of pair variables (x_i, y_i) , each one having $D(x_i) \times D(y_i)$ for domain, with C redefined as a binary constraint $C'((x_{i-1}, y_{i-1}), (x_i, y_i))$, its expression remaining the same, up to the syntax. There is clearly a one-to-one correspondence between the solutions of the thus defined $\text{SLIDE}[C]([x_0, \dots, x_n], [y_0, \dots, y_n])$ and $\text{SLIDE}[C']([(x_0, y_0), \dots, (x_n, y_n)])$.

5.2.2 Two sequences of different length

When two sequences do not have the same length, merging them is not that intuitive. We want a method that on the one hand allows to transform two sequences into one, but without damaging on the other hand the complexity results. Concretely, we want that the parameter K remains unchanged.

To operate this, the sequences will be reformulated so as to simulate the fact that C will not slide on every consecutive variable, in order not to take into its scope variables it should not. To do this, dummy variables are introduced between the variables of each sequence. A dummy variable can only take a dummy value, denoted $*$. Let us illustrate it with an example.

Example Take the previously introduced example. We redefine the constraint MAXIMUM in a SLIDE of two same length sequences. Every X and Y variable is the same as

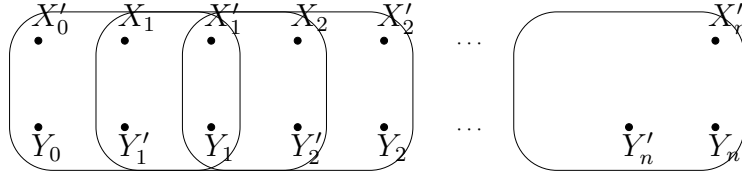


Figure 5.2: The MAXIMUM constraint coded in a two-sequence SLIDE

in the previous coding, and the X' and Y' variables are the dummy variables, which domain contains only the dummy value. The slid constraint becomes:

$$C'([u_1, u_2, u_3], [v_1, v_2, v_3]) \equiv \begin{cases} ((u_1 = u_3 = v_2 = *) \wedge C([u_2], [v_1, v_3])) \\ \vee \\ (v_1 = v_3 = u_2 = *) \end{cases}$$

Now that the two sequences have the same length, they can be merged as described before.

5.2.3 Filtering algorithms for the multisequence SLIDE

Now that the process of merging two sequences has been described, we can apply it to obtain a filtering algorithm for the multi-sequence SLIDE. For this, we just have to *flatten* the sequences into one, by recursively merging two consecutive sequences. Then anyone of the methods developed in the previous sections can be used to propagate the multisequence SLIDE. The properties of the merging process ensure that when flattening such a constraint, the complexity parameter d^K remains unchanged.

5.3 Non-sliding variables

This extension consists in introducing into the variables of the constraint what were first only fixed integer parameters. That is, we relax the condition that they must have a unitary domain, and pruning on their domains must be performed. These variables must occur in each occurrence of C , that is why they will be referred to as non-sliding variables.

For example, we had expressed the $\text{MAXIMUM}[N]([X_1, \dots, X_n])$ constraint with $\text{SLIDE}[C, N]([X_1, \dots, X_n], [e_0, \dots, e_n])$, where N was a fixed given parameter, and C defined on (X_i, e_{i-1}, e_i) and involving N in its definition. Now, if N is just an integer variable like any other, the constraint on it becomes $\text{MAXIMUM}(N, [X_1, \dots, X_n])$, and can simply be expressed in the extended version by $\text{SLIDE}[C](N; [X_1, \dots, X_n], [e_0, \dots, e_n])$, where C is now defined on the four variables $(N; X_i, e_{i-1}, e_i)$. Again, N is the same variable in every C_i , whatever value i has.

5.3.1 Definition and Complexity

The SLIDE in this extension has this general form:

$$\text{SLIDE}[C](N_1, \dots, N_m; [X_1^1, \dots, X_{n_1}^1], \dots, [X_1^s, \dots, X_{n_s}^s])$$

From a syntactic point of view, the non-sliding variables N_1, \dots, N_m are separated with a semicolon to distinguish them from the classical variable sequences.

Enforcing GAC on this constraint in \mathcal{NP} -hard. Take for example the USES constraint. The constraint $\text{USES}([X_1, \dots, X_m], [Y_1, \dots, Y_n])$ is satisfied iff the set of values of Y_j variables is included in the set of values of X_i variables. This constraint can simply be implemented in this extended SLIDE by $\text{SLIDE}[C](X_1, \dots, X_m; [Y_1, \dots, Y_n])$, with $C(X_1, \dots, X_m; Y_i) \equiv (Y_i = X_1) \vee \dots \vee (Y_i = X_m)$. But enforcing GAC on USES has been shown to be \mathcal{NP} -hard [BHH⁺05], hence the result for the extended SLIDE.

However, when m is bounded, enforcing GAC becomes polynomial, as stated in the following section.

5.3.2 Propagating

One first way to implement this extension is to express it with the basic multi-sequence SLIDE, by converting each non-sliding variable to a new sequence of variables, as follows:

$$\text{SLIDE}[C']([X_1^1, \dots, X_{n_1}^1], \dots, [X_1^s, \dots, X_{n_s}^s], [N_0^1, \dots, N_l^1], \dots, [N_0^m, \dots, N_l^m])$$

where l is the number of occurrences of C (and C'), and where C' is defined as follows:

$$\begin{aligned} & C'(X_i^1, \dots, X_{i+k_1-1}^1, \dots, X_i^s, \dots, X_{i+k_s-1}^s, N_{i-1}^1, N_i^1, \dots, N_{i-1}^m, N_i^m) \\ \equiv & C(N_{i-1}^1, \dots, N_{i-1}^m; X_i^1, \dots, X_{i+k_1-1}^1, \dots, X_i^s, \dots, X_{i+k_s-1}^s) \wedge \\ & (N_{i-1}^1 = N_i^1) \wedge \dots \wedge (N_{i-1}^m = N_i^m) \end{aligned}$$

Given an arc-consistency procedure for the constraint C , arc-consistency can be obtained for C' , therefore the methods seen for propagating the multi-sequence SLIDE can be applied as is to obtain a filtering algorithm for this extension. As for complexity, we saw that the prevailing parameter is d^p , where p is the intersection size of the set of variables of two consecutive occurrences of C . Here, each non-sliding variable increments by one the value of p (due to the equality constraint between the variables of its corresponding sequence). So, if m is not bounded but is rather an instance variable, this algorithm is exponential (as it could be expected).

As a matter of fact, one must keep in mind that this extension is more relevant for those cases where m is small (typically one parameter) rather than those where m is a variable (typically for the USES constraint, for which the SLIDE is not a good choice).

6 Examples and comparing with other approaches

Now that a wide range of SLIDE constraints has been introduced, we will show they have a great expressivity. In this purpose, first some typical constraints that can be expressed in SLIDE are described. Then the SLIDE approach will be compared with other, showing that we can at least achieve the same results.

6.1 Examples

The previously introduced example :

- $\text{MAXIMUM}[N](X_1, \dots, X_n)$
 $\equiv \text{SLIDE}[C, N]([X_1, \dots, X_n], [Y_0, \dots, Y_n])$, where
semantics “ N is the bigger value taken in the sequence.”
domains $D(Y_0) = \{0\}, D(Y_n) = \{1\}, D(Y_i)_{0 < i < n} = \{0, 1\}$
constraint

$$C([X_i], [Y_{i-1}, Y_i]) \equiv \vee \begin{cases} N > X_i \wedge Y_{i-1} = 0 \wedge Y_i = 0 \\ N = X_i \wedge Y_{i-1} = 0 \wedge Y_i = 1 \\ N \geq X_i \wedge Y_{i-1} = 1 \wedge Y_i = 1 \end{cases}$$

Some other constraints :

- $\text{MAXINDEX}[J]([X_1, \dots, X_n])$
 $\equiv \text{“SLIDE}[C]([X_1, \dots, X_n], [Y_0, \dots, Y_n], [J_0, \dots, J_n])\text{”}$
semantics J is the index of the maximum value taken by X_1, \dots, X_n
domains $D(Y_0) = \min(D(X_1)), D(J_0) = \{1\}, D(J_n) = \{J\}$
constraint

$$C([X_i], [Y_{i-1}, Y_i], [J_{i-1}, J_i]) \equiv \wedge \begin{cases} Y_i = \max(Y_{i-1}, X_i) \\ Y_i = X_i \wedge J_i = i \\ Y_i > X_i \wedge J_i = J_{i-1} \end{cases}$$

- $\text{CARDPATH}[C'](N, [X_1, \dots, X_n])$
 $\equiv \text{SLIDE}[C]([X_1, \dots, X_n], [Y_0, \dots, Y_{n-k+1}])$
semantics “ C' , a constraint of arity $k < n$ is satisfied N times on the sequence X_1, \dots, X_n .”
domains $D(Y_0) = \{0\}, D(Y_{n-k+1}) = D(N)$
constraint

$$C([X_i, \dots, X_{i+k-1}], [Y_{i-1}, Y_i]) \equiv \vee \begin{cases} C'(X_i, \dots, X_{i+k-1}) \wedge Y_i = Y_{i-1} + 1 \\ Y_i = Y_{i-1} \end{cases}$$
- $\text{LEXGEQ}[a]([X_1, \dots, X_n])$
 $\equiv \text{SLIDE}[C, a]([X_1, \dots, X_n], [u_0, \dots, u_n])$
semantics “The vector formed by the values of X_1, \dots, X_n is lexicographically greater than of equal to a , a fixed vector of size n .”
domains $D(u_0) = \{0\}, D(u_i)_{i>0} = \{0, 1\}$
constraint

$$C([X_i], [u_{i-1}, u_i]) \equiv \vee \begin{cases} X_i = a_i \wedge u_{i-1} = 0 \wedge u_i = 0 \\ a_i < X_i \wedge u_{i-1} = 0 \wedge u_i = 1 \\ u_{i-1} = 1 \wedge u_i = 1 \end{cases}$$
- $\text{LEXLEQ}[b]([X_1, \dots, X_n])$
 $\equiv \text{SLIDE}[C, b]([X_1, \dots, X_n], [v_0, \dots, v_n])$
semantics “The vector formed by the values of X_1, \dots, X_n is lexicographically less than of equal to b , a fixed vector of size n .”
domains $D(v_0) = \{0\}, D(v_i)_{i>0} = \{0, 1\}$
constraint

$$C([X_i], [v_{i-1}, v_i]) \equiv \vee \begin{cases} X_i = b_i \wedge v_{i-1} = 0 \wedge v_i = 0 \\ X_i < b_i \wedge v_{i-1} = 0 \wedge v_i = 1 \\ v_{i-1} = 1 \wedge v_i = 1 \end{cases}$$

6.2 Conjunction of constraints

The multisequence SLIDE allows to express a lot of global constraints. However, given a global constraint, there could be different equivalent expressions using a different number of sequences. Some of them might look clearer, more natural than others. Nevertheless, the merging process allows to change from one expression to another. A useful application of this technique is to systematically obtain a SLIDE expressing the conjunction of two global constraints already defined in SLIDE. Let us illustrate it with an example.

Consider the constraint $\text{BETWEEN}[a, b]([X_1, \dots, X_n])$, where a and b are two vectors of size n , that holds iff the vector formed by the values of X_1, \dots, X_n is lexicographically between a and b . Naturally this constraint is the conjunction of the $\text{LEXGEQ}[a]([X_1, \dots, X_n])$ and $\text{LEXLEQ}[b]([X_1, \dots, X_n])$ constraints. This conjunction can be expressed as follows, where C_1 and C_2 are the slid constraints of LEXGEQ and LEXLEQ :

- $\text{BETWEEN}[a, b]([X_1, \dots, X_n])$
 $\equiv \text{SLIDE}[C', a, b]([X_1, \dots, X_n], [u_0, \dots, u_n], [v_0, \dots, v_n])$
 $C'([X_i], [u_{i-1}, u_i], [v_{i-1}, v_i]) \equiv C_1([X_i], [u_{i-1}, u_i]) \wedge C_2([X_i], [v_{i-1}, v_i])$

But this formulation is not satisfying. Let us now merge the last two sequences. We now obtain a single sequence of (u_i, v_i) variables with values in the $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ set, renamed respectively as w_i and $\{0, 1, 2, 3\}$ for convenience. By expanding the $C_1 \wedge C_2$ conjunction as a DNF and using the new variable and value names, we obtain a constraint C , thus leading to the following coding :

- $\text{BETWEEN}[a, b]([X_1, \dots, X_n])$
 $\equiv \text{SLIDE}[C, a, b]([X_1, \dots, X_n], [w_0, \dots, w_n])$
domains $D(w_0) = \{0\}, D(w_i)_{i>0} = \{0, 1, 2, 3\}$

$$C([X_i], [w_{i-1}, w_i]) \equiv \vee \left\{ \begin{array}{l} X_i = a_i \wedge X_i = b_i \wedge w_{i-1} = 0 \wedge w_i = 0 \\ X_i = a_i \wedge X_i < b_i \wedge w_{i-1} = 0 \wedge w_i = 1 \\ X_i = a_i \quad \quad \quad \wedge w_{i-1} = 1 \wedge w_i = 1 \\ a_i < X_i \wedge X_i = b_i \wedge w_{i-1} = 0 \wedge w_i = 2 \\ a_i < X_i \wedge X_i < b_i \wedge w_{i-1} = 0 \wedge w_i = 3 \\ a_i < X_i \quad \quad \quad \wedge w_{i-1} = 1 \wedge w_i = 3 \\ \quad \quad \quad \quad \quad \quad \quad X_i = b_i \wedge w_{i-1} = 2 \wedge w_i = 2 \\ \quad \quad \quad \quad \quad \quad \quad X_i < b_i \wedge w_{i-1} = 2 \wedge w_i = 3 \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad w_{i-1} = 3 \wedge w_i = 3 \end{array} \right.$$

This coding is more in correspondance to the coding we would have thought of if trying to directly code the BETWEEN constraint with a SLIDE , though it would surely not be an easy task.

6.3 Beldiceanu's extended automata

In [BCP04], Beldiceanu *et al.* presented a method aiming at getting filtering algorithms for global constraints, as much automatically as possible. In some cases, generalized arc-consistency on those constraints is achieved, in other (not rare) cases a weaker pruning is done, staying in both cases with an algorithm that runs in polynomial time. The fundamental idea of this approach is to write a checker for the constraint for which we want a filtering algorithm, from which a particular automaton is built, more or less

systematically. This automaton must describe in some way the execution of the checker on a instantiation, thus accepting a valid instantiation and rejecting a non-valid one. Classical finite states automata are not rich enough to express a checker whichever it is, so an extension of the concept of automaton is introduced. This extension mainly consists in associating counters to states and transitions : they are initialized in the initial state, on each transition a function updates these counters, and the final states are associated to a condition on these counters, meaning that an instantiation is accepted if a final state has been reached, with the corresponding condition on the counters being satisfied. Filtering is obtained simply by computing arc-consistency on a conjunction of constraints that describes this automaton (and which is therefore logically equivalent to the global constraint). In some cases, arc-consistency for every constraint of the conjunction is sufficient for obtaining the generalized arc-consistency for the global constraint. The approach presented here consists of reformulating the automaton by a typical use of the SLIDE language (using multiple sequences). This shows that the SLIDE language is at least as expressive as these extended automata. Furthermore, generalized arc-consistency is always achieved, wether it is polynomial or not, using the techniques developed to propagate a SLIDE constraint.

6.3.1 From automaton to Slide

In Beldiceanu's extended automata, each transition is labelled with a condition on a subset of variables of the constraint, according to an index i . Such a subset is called a signature variable, and their number is set to m . Each time a transition is fired, this index i is implicitly incremented. This way, when we move in the automaton, for each transition a condition must be verified on a subset of variables, thus incrementing i , so that the next condition will have to be verified on the next subset of variables. A transition labelled by \$ can be fired iff $i = m$, that is to say when the sequence of variables has totally been checked (which implies we reached a final state). We can clearly see that there is a SLIDE constraint behind this.

Here is its description.

- We have multiple sequences :
 - one or more sequences of *constraint variables*. Let m be the number of signature variables of the automaton, we have one or more sequences of the form $X_1(\dots, X_l), \dots, X_m(\dots, X_{m+l-1=n})$, where l and the number of sequences depend on the signature variables. Each variable has its corresponding domain.
 - one sequence of *state variables*, of size $m + 1$, of the form q_0, \dots, q_m . To each one of the Q states of the automaton is associated an integer, thus initializing the domains as follows :

$$\begin{aligned}
 D(q_0) &= \{0\} && \text{(the initial state)} \\
 D(q_m) &= \{\text{final states}\} \\
 D(q_i)_{0 < i < m} &= \{0, \dots, Q - 1\}
 \end{aligned}$$

- one sequence (or more) of *counter variables*, if necessary, of size $m + 1$, of the form c_0, \dots, c_m . The domains are initialized as follows :

$$\begin{aligned} D(c_0) &= \{\text{initialization value}\} \\ D(c_m) &= \{\text{acceptation values}\} \\ D(c_i)_{0 < i < m} &= \{\text{all possible values}\} \end{aligned}$$

Let us remark that if a final state is reachable with a \$ transition, then the origin of the transition is also added to the domain of q_m . If a final state is only accessible through \$ transitions, it is not taken into account (in the value of E and therefore only its predecessors are put in the of q_m). As for the values of the counter variables, the acceptation values are those that are consistent with regard to the acceptation condition. The possible values depend on the update function and the initial value, we only impose that it contains all the values that the counter can take during the check of any valid instantiation, and only those.

- The C constraint :

The C constraint is defined on the variables $X_i, \dots, X_{i+l-1}, q_{i-1}, q_i, c_{i-1}, c_i, \forall i = 1, \dots, m$, adapted to the needs (more sequences of constraint variables, more or not counter variables, etc.). The semantics of this constraint is "when some condition is satisfied on X_i, \dots, X_{i+l-1} , we go from state q_{i-1} to state q_i , and update if necessary the counter from c_{i-1} to c_i ". This constraint looks close to the transition constraint defined by Beldiceanu in [BCP04].

We find here the same result as Beldiceanu's. When the constraint variables have an empty intersection (*ie* $l = 1$) and that no counter is needed, GAC for every C is enough to have GAC on the SLIDE. Indeed, the only intersection between C 's is of size 1 and concern the state variables. But when these conditions are not met, using the techniques for propagating the SLIDE constraint still ensures us to achieve GAC.

Moreover, this suggests that with at most three sequences (or more of constraint variables), the SLIDE has an important expressiveness. In fact, it's difficult to think of a situation where two counters should be needed, where the constraint is not the conjunction of two more primitive ones.

Let us make another remark. Beldiceanu also describes a process to obtain an automaton for expressing the conjunction of two other automata, so as to express the conjunction of the constraints expressed by the automata. Then the SLIDE obtained by the conjunctive automaton is strictly the same as the SLIDE that would directly be obtained by using the technique of the previous section.

6.3.2 Some examples

The four examples given in [BCP04] are coded in SLIDE as follows.

- GLOBALCONTIGUITY(X_1, \dots, X_n)
 \equiv SLIDE[C]([X_1, \dots, X_n], [q_0, \dots, q_n]), where :

semantics "There is no more than one group of contiguous 1's in the sequence."

parameters $m = n, (l = 1), E = 3$ (final state not counted), no counter needed

domains $D(q_0) = \{0\}, D(q_i)_{i>0} = \{0, 1, 2\}$

constraint $C([X_i], [q_{i-1}, q_i]) = \{000, 101, 111, 012, 022\}$

- LEXLEQ($[X_1, \dots, X_n], [Y_1, \dots, Y_n]$)
 \equiv SLIDE $[C]([X_1, \dots, X_n], [Y_1, \dots, Y_n], [q_0, \dots, q_n])$, where :

semantics "First sequence is lexicographically less than or equal to the second."

parameters $l = 1, E = 2$, no counter needed

domains $D(q_0) = \{0\}, D(q_i)_{i>0} = \{0, 1\}$

constraint

$$C([X_i], [Y_i], [q_{i-1}, q_i]) \equiv \vee \begin{cases} X_i = Y_i \wedge q_{i-1} = 0 \wedge q_i = 0 \\ X_i < Y_i \wedge q_{i-1} = 0 \wedge q_i = 1 \\ q_{i-1} = 1 \wedge q_i = 1 \end{cases}$$

- AMONG $[N, V]([X_1, \dots, X_n])$
 \equiv SLIDE $[C, N, V]([X_1, \dots, X_n], [c_0, \dots, c_n])$, where :

semantics " $|\{X_i\}_{i=1}^n \cap V| = N$, N and V fixed given parameters."

parameters $l = 1, E = 1$ (final state not counted, only one initial and final state so state sequence unnecessary), use of counters

domains $D(c_0) = \{0\}, D(c_n) = \{N\}, D(c_i)_{0<i<n} = \{0, \dots, N\}$

constraint

$$C([X_i], [c_{i-1}, c_i]) \equiv \vee \begin{cases} X_i \in V \wedge c_i = c_{i-1} + 1 \\ X_i \notin V \wedge c_i = c_{i-1} \end{cases}$$

- INFLEXION $[N]([X_1, \dots, X_n])$
 \equiv SLIDE $[C, N]([X_1, \dots, X_n], [q_0, \dots, q_{n-1}], [c_0, \dots, c_{n-1}])$, where :

semantics "The number of inflexions in the sequence is N "

parameters $m = n - 1 (l = 2), E = 3$ (final state not counted), use of counters

domains $D(q_0) = \{0\}, D(q_i)_{i>0} = \{0, 1, 2\}$

$D(c_0) = \{0\}, D(c_{n-1}) = \{N\}, D(c_i)_{0<i<n-1} = \{0, \dots, N\}$

constraint

$$C([X_i, X_{i+1}], [q_{i-1}, q_i], [c_{i-1}, c_i]) \equiv \vee \begin{cases} X_i = X_{i+1} \wedge q_{i-1} = q_i \wedge c_i = c_{i-1} \\ X_i < X_{i+1} \wedge q_{i-1} = 0 \wedge q_i = 1 \wedge c_i = c_{i-1} \\ X_i > X_{i+1} \wedge q_{i-1} = 0 \wedge q_i = 2 \wedge c_i = c_{i-1} \\ X_i < X_{i+1} \wedge q_i = q_{i-1} = 1 \wedge c_i = c_{i-1} \\ X_i > X_{i+1} \wedge q_i = q_{i-1} = 2 \wedge c_i = c_{i-1} \\ X_i < X_{i+1} \wedge q_{i-1} = 2 \wedge q_i = 1 \wedge c_i = c_{i-1} + 1 \\ X_i > X_{i+1} \wedge q_{i-1} = 1 \wedge q_i = 2 \wedge c_i = c_{i-1} + 1 \end{cases}$$

6.4 The REGULAR constraint

In [Pes04], Gille Pesant introduced the REGULAR constraint. This global constraint takes as a parameter a regular language L , and enforces that an instantiation t is such that the word $w(t)$ formed by the concatenation of the symbols of t belongs to L . A propagation algorithm is developed that achieves GAC in polynomial time. In fact, this constraint can be seen as a particular case of Beldiceanu's automata. We just have to express the language with a classic deterministic finite state automaton. This automaton could be a Beldiceanu's automaton where : no counters are used, each signature variable contains a single variable, and has a condition of the form $X_i = a$ where a is a symbol of the alphabet on which L is defined. Using exactly the same process, the REGULAR constraint can be coded in SLIDE as follows, where δ is the transition function of the automaton:

- $\text{REGULAR}[\mathcal{A}]([X_1, \dots, X_n])$
 $\equiv \text{SLIDE}[C, \mathcal{A}]([X_1, \dots, X_n], [q_0, \dots, q_n])$
domains $D(q_0) = \{\text{initial state}\}, D(q_n) = \{\text{final states}\}$
constraint $C([X_i], [q_{i-1}, q_i]) \equiv \delta(q_{i-1}, X_i) = q_i$

6.5 Graph characteristics

In [BPR05], Beldiceanu *et al.* present a method also allowing to obtain systematically and in a generic manner filtering algorithms for global constraints. Everything is based on expressing a global constraint as a graph property. By associating a graph to a current state of the variables (that is, whether they are instantiated or not, and if not, the values in their current domain), and by expressing the constraint on the variables as a constraint on a characteristic of this graph, some filtering can be performed.

This is another approach for obtaining systematically filtering algorithms, as long as we are able to express a global constraint as a graph characteristic. The main drawback of it is that we don't clearly know what is the level of consistency achieved. Not all of the constraints introduced in this article can be expressed in SLIDE, but for those that can, we know that GAC is achieved. Additionally, the propagation algorithm is more generic in that it is the same no matter how a global constraint is expressed in SLIDE, whereas a special attention must be done for each possible graph characteristic we might consider. For those constraints that cannot be expressed in SLIDE, it would be interesting to express them in RANGE and ROOTS and compare.

7 Experiments

To lead the experiments, we were based on the well studied Car Sequencing problem, firstly studied in a constraint programming perspective in [DSV88]. We observed that this problem can nearly be expressed as a simple SLIDE constraint. The idea is that thanks to the propagation algorithms, GAC can always be achieved, thus making a backtrack-free resolution of this problem. If the algorithms are efficient enough, this would surely give a clear advantage to the SLIDE approach.

The implemented algorithm is the first dynamic programming version, with some technical enhancements in order to hopefully have an acceptable working version. The idea is to show that a simple implementation can already give acceptable and promising results.

Unfortunately, the experiment results are not yet available to be included in this report.

8 Conclusion

In this report we saw that the SLIDE constraint allows to express a wide range of global constraints. Some examples have been shown, so as to give a feeling of the sort of global constraints that are good candidates to be coded in SLIDE. The expressivity power has also been established by comparing this approach with other similar in their aim, and thus observing that the SLIDE approach truly allows to express an important part of global constraints.

On the other hand, various algorithms have been exposed, which are totally generic, thus allowing to systematically obtain GAC for any global constraint as long as one could express it using the SLIDE constraint. The price to pay for using a generic method is the loss of particular semantics, and therefore of efficiency compared to a specific algorithm for a particular constraint. As a response to this, automata have been introduced betting it is an optimized structure for representing in a generic way a constraint. These results remain to be established by experiments.

If the results are validated, that is if we observe an acceptable efficiency difference between specific and generic algorithms, then the SLIDE constraint can be proposed to users as one of the few constraints for expressing global constraints. Giving a methodology for this to be done as more systematically as possible would then be the next step. For example, Beldiceanu gives in [BCP04] a rather general methodology for expressing a global constraint with his extended automata. A similar kind of approach could be a good starting point.

A last and tedious work that now must be done is to find all of the global constraints of the catalog [Bel05] that can be expressed using SLIDE. We saw that the RANGE and ROOTS constraints allow to express some 70 out of the 235 constraints. How many more constraints will be expressed with SLIDE ? And once again, what happens with the remaining constraints ? Can we express them by simply using a combination of the, up to now, three constraints of the language (sliding a RANGE/ROOTS constraint ?) ? Or what constraint must be added to the language so as to express them ? And finally, how many constraints will this language contain ?

Bibliography

- [AFM02] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs-Application to configuration . In *Artificial Intelligence* , volume 135, pages 199–234. Elsevier, février 2002. bb modif 28/02/02.
- [BC01] Nicolas Beldiceanu and Mats Carlsson. Revisiting the cardinality operator and introducing the cardinality-path constraint family. In *Proceedings of the 17th International Conference on Logic Programming*, pages 59–73, London, UK, 2001. Springer-Verlag.
- [BCP04] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. Technical report, École des Mines de Nantes, 2004.
- [Bel05] Nicolas Beldiceanu. Global constraint catalog. Technical report, École des Mines de Nantes, May 2005.
- [BHH⁺05] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. The Range and Roots Constraints: Specifying Counting and Occurrence Constraints. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 60–65, Edinburgh, Scotland, August 2005. Professional Book Center.
- [BHHW04] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The tractability of global constraints, 2004.
- [BPR05] Nicolas Beldiceanu, Thierry Petit, and Guillaume Rochart. Bounds of graph characteristics. In *CP*, pages 742–746, 2005.
- [DSV88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proc. of the 8th ECAI*, pages 290–295, Munich, Germany, 1988.
- [DVSA88] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, 1988.

-
- [KR92] D.E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal of Discrete Mathematics*, 5(3):422–427, 1992.
- [Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP*, pages 482–495, 2004.
- [Rég94] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI'94*, pages 362–367, Seattle WA, 1994.
- [Wag95] David B. Wagner. *Dynamic programming*, 1995.

Résumé Les contraintes globales ont joué un rôle considérable dans l'essor de la programmation par contraintes. Elles contribuent à exprimer de façon plus précise un problème. Dernièrement, on en recensait 235. Or ce grand nombre pose un obstacle contre la facilité d'utilisation de la programmation par contraintes. Des travaux entrepris par Bessière *et al.* ont pour but de définir un langage restreint de contraintes globales permettant d'exprimer *et* propager toute autre contrainte globale. Deux contraintes déjà définies ont ainsi permis d'en exprimer 70 parmi les contraintes recensées. Les contraintes restantes expriment souvent une propriété sur une *séquence* de variables. C'est pour exprimer ceci que la contrainte SLIDE a été introduite. Ce rapport montre dans un premier temps la complexité de la propagation de cette contrainte. Par la suite, son intérêt est démontré à travers une série d'algorithmes polynomiaux puis de mises en applications. De plus, des comparaisons avec d'autres approches donnent un avantage à cette contrainte, pour le but recherché. Sa grande expressivité semble ainsi s'établir. Enfin, des travaux expérimentaux sont prévus pouvant mesurer l'efficacité des algorithmes de propagation sur des problèmes réels.

Abstract Global constraints played a significant role in the development of constraint programming. Thanks to them, a problem can be described more precisely. At present, the number of such constraints stands at 235. This poses an obstacle in making constraint programming easy to use. Recent works initiated by Bessière *et al.* intend to define a small language of constraints allowing to express *and* to propagate any other global constraint. Two constraints have already been proposed that permit to express 70 among them. The remaining constraints often express a property on a *sequence* of variables. The SLIDE constraint has then been introduced to express such a property. This report first focuses on complexity issues with regard to propagating this constraint. Then its interest is shown through a number of polynomial algorithms and practical applications. In addition, comparisons drawn with other approaches give an advantage to this constraint, for the sought objective. Its large expressivity seems therefore to get established. At last, experiments are planned so as to measure the efficiency of the propagation algorithms on real-world problems.