Global Constraints

Jean-Charles Régin

ILOG
1681, route des Dolines,
Sophia Antipolis, 06560 Valbonne, France
e-mail: jcregin@ilog.fr

Abstract. Constraint programming (CP) is mainly based on filtering algorithms; their association with global constraints is one of the main strengths of CP. This chapter is an overview of these two techniques. Some of the most frequently used global constraints are presented. In addition, the filtering algorithms establishing arc consistency for two useful constraints, the alldiff and the global cardinality constraints, are fully detailed. Filtering algorithms are also considered from a theoretical point of view: three different ways to design filtering algorithms are described and the quality of the filtering algorithms studied so far is discussed. A categorization is then proposed. Over-constrained problems are also mentioned and global soft constraints are introduced.

1 Introduction

A constraint network (CN) consists of a set of variables; domains of possible values associated with each of these variables; and a set of constraints that link up the variables and define the set of combinations of values that are allowed. The search for an instantiation of all variables that satisfies all the constraints is called a Constraint Satisfaction Problem (CSP), and such an instantiation is called a solution of a CSP.

A lot of problems can be easily coded in terms of CSP. For instance, CSP has already been used to solve problems of scene analysis, placement, resource allocation, crew scheduling, time tabling, scheduling, frequency allocation, car sequencing, and so on. An interesting paper of Simonis [Simonis, 1996] presents a survey on industrial studies and applications developed over the last ten years.

Unfortunately, a CSP is an NP-Complete problem. Thus, much work has been carried out in order to try to reduce the time needed to solve a CSP. Constraint Programming (CP) is one of these techniques.

Constraint Programming proposes to solve CSPs by associating with each constraint a filtering algorithm that removes some values of variables that cannot belong to any solution of the CSP. These filtering algorithms are repeatedly called until no new deduction can be made. This process is called the propagation mechanism. Then, CP uses a search procedure (like a backtracking algorithm) where filtering algorithms are systematically applied when the domain of a variable is modified. Therefore, with respect to the current domains of the variables and thanks to filtering algorithms, CP removes once and for all certain inconsistencies that would have been discovered several times otherwise. Thus, if the cost of the calls of the filtering algorithms at each node is less than the time required by the search procedure to discover many times the same inconsistency, then the resolution will be speeded up.

One of the most interesting properties of a filtering algorithm is arc consistency. We say that a filtering algorithm associated with a constraint establishes arc consistency if it removes all the values of the variables involved in the constraint that are not consistent with the constraint. For instance, consider the constraint $x + 3 = y$ with the domain of x equals to $D(x) = \{1, 3, 4, 5\}$ and the domain of y equal to $D(y) = \{4, 5, 8\}$. Then the establishing of arc consistency will lead to $D(x) = \{1, 5\}$ and $D(y) = \{4, 8\}$.

Since constraint programming is based on filtering algorithms, it is quite important to design efficient and powerful algorithms. Therefore, this topic caught the attention of many researchers, who then discovered a large number of algorithms. Nevertheless, a lot of studies on arc consistency have been limited to binary constraints that are defined in extension, in other words by the list of allowed combinations of values. This limitation was justified by the fact that any constraint can always be defined in extension and by the fact that any non-binary constraint network can be translated into an equivalent binary one with additional variables [Rossi et al., 1990]. However, in practice, this approach has several drawbacks:

- it is often inconceivable to translate a non-binary constraint into an equivalent set of binary ones because of the underlying computational and memory costs (particularly for non-representable ones [Montanari, 1974]).
- the structure of the constraint is not used at all. This prevents us from developing more efficient filtering algorithm dedicated to this constraint. Moreover, some non-binary constraints lose much of their structure when encoded into a set of binary constraints. This leads, for example, to a much less efficient pruning behavior for arc consistency algorithms handling them.

The advantage of using the structure of a constraint can be emphasized on a simple example: the constraint $x \leq y$. Let $min(D)$ and $max(D)$ be respectively the minimum and the maximum value of a domain. It is straightforward to establish that all the values of y in the range $[min(D(x)), max(D(y)]$ are consistent with the constraint and all the values of x in the range $[min(D(x)), max(D(y)]$ are consistent with the constraint. This means that arc consistency can be efficiently and easily established by removing the values that are not in the above ranges. Moreover, the use of the structure is often the only way to avoid memory

consumption problems when dealing with non-binary constraints. In fact, this approach prevents you from explicitly representing all the combinations of values allowed by the constraint.

Thus, researchers interested in the resolution of real life applications with constraint programming, and notably those developing languages that encapsulate CP (like PROLOG), designed specific filtering algorithms for the most common simple constraints (like $=, \neq, <, \leq, ...$) and also general frameworks to exploit efficiently some knowledge about binary constraints (like AC-5 [Van Hentenryck et al., 1992]). However, they have been confronted with two new problems: the lack of expressiveness of these simple constraints and the weakness of domain reduction of the filtering algorithms associated with these simple constraints. It is, indeed, quite convenient when modeling a problem in CP to have at one's disposal some constraints corresponding to a set of constraints. Moreover, these new constraints can be associated with more powerful filtering algorithms because they can take into account the simultaneous presence of simple constraints to further reduce the domains of the variables. These constraints encapsulating a set of other constraints are called **global constraints**.

One of the most well known examples is the alldiff constraint, especially because the filtering algorithm associated with this constraint is able to establish arc consistency in a very efficient way.

An alldiff constraint defined on X, a set of variables, states that the values taken by variables must be all different. This constraint can be represented by a set of binary constraints. In this case, a binary constraint of difference is built for each pair of variables belonging to the same constraint of difference. But the pruning effect of arc consistency for these constraints is limited. In fact, for a binary alldiff constraint between two variables i and j, arc-consistency removes a value from domain of i only when the domain of j is reduced to a single value. Let us suppose we have a CSP with 3 variables x_1, x_2, x_3 and an alldiff constraint involving these variables with $D(x_1) = \{a, b\}$, $D(x_2) = \{a, b\}$ and $D(x_3) = \{a, b, c\}$. Establishing arc consistency for this alldiff constraint removes the values a and b from the domain of x_3, while arc-consistency for the alldiff represented by binary constraints of difference does not delete any value.

We can further emphasize the advantage of global constraints on a more realistic example that involves global cardinality constraints (GCC).

	Mo	Tu	We	Th	...
peter	D	N	O	M	
paul	D	B	M	N	
mary	N	O	D	D	
...					

A = {M,D,N,B,O}, P = {peter, paul, mary, ...}
W = {Mo, Tu, We, Th, ...}
M: morning, D: day, N: night B: backup, O: day-off

Fig. 1. An Assignment Timetable.

A GCC is specified in terms of a set of variables $X = \{x_1, ..., x_p\}$ which take their values in a subset of $V = \{v_1, ..., v_d\}$. It constrains the number of times a value $v_i \in V$ is assigned to a variable in X to be in an interval $[l_i, u_i]$. GCCs arise in many real life problems. For instance, consider the example derived from a real problem and given in [Caseau et al., 1993] (cf. Figure 1). The task is to schedule managers for a directory-assistance center, with 5 activities (set A), 7 persons (set P) over 7 days (set W). Each day, a person can perform an activity from the set A. The goal is to produce an assignment matrix that satisfies the following global and local constraints:

- **general constraints** restrict the assignments. First, for each day we may have a minimum and maximum number for each activity. Second, for each week, a person may have a minimum and maximum number for each activity. Thus, for each row and each column of the assignment matrix, there is a global cardinality constraint.
- **local constraints** mainly indicate incompatibilities between two consecutive days. For instance, a morning schedule cannot be assigned after a night schedule.

Each general constraint can be represented by as many min/max constraints as the number of involved activities. Now, these min/max constraints can be easily handled with, for instance, the **atmost/atleast** operators proposed in [Van Hentenryck and Deville, 1991]. Such operators are implemented using local propagation. But as is noted in [Caseau et al., 1993]: "The problem is that efficient resolution of a timetable problem requires a global computation on the set of min/max constraints, and not the efficient implementation of each of them separately." Hence, this way is not satisfactory. Therefore global cardinality constraints associated with efficient filtering algorithms (like ones establishing arc consistency) are needed.

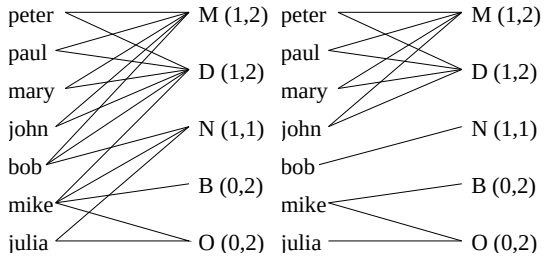

Fig. 2. An example of a global cardinality constraint.

In order to show the difference in global and local filtering, consider a GCC associated with a day (cf figure 2). The constraint can be represented by a bipartite graph called a value graph (left graph in Figure 2). The left set corresponds

to the person set, the right set to the activity set. There exists an edge between a person and an activity when the person can perform the activity. For each activity, the numbers between parentheses express the minimum and the maximum number of times the activity has to be assigned. For instance, John can work the morning or the day but not the night; one manager is required to work the morning, and at most two managers work the morning. We recall that each person has to be associated with exactly one activity.

Encoding the problem with a set of atmost/atleast constraints leads to no deletion. Now, we can carefully study this constraint. Peter, Paul, Mary, and John can work only in the morning and during the day. Moreover, morning and day can be assigned together to at most 4 persons. Thus, no other persons (i.e. Bob, Mike, nor Julia) can perform activities M and D. So we can delete the edges between Bob, Mike, Julia and D, M. Now only one possibility remains for Bob: N, which can be assigned at most once. Therefore, we can delete the edges {mike,N} and {julia,N}. This reasoning leads to the right graph in Figure 2. It corresponds to the establishing of arc consistency for the global constraint.

This chapter is organized as follows. First, some preliminaries are reviewed and the definition and the significance of global constraints are discussed. Some of the most frequently used global constraints are then presented. Section 3 deals with the possible types of filtering algorithms (FA). Three types of filtering algorithm are presented. In section 4, the filtering algorithms establishing arc consistency for the alldiff and the global cardinality constraint are detailed. Section 5 deals with over-constrained problems and shows the advantages of modeling the Maximal Constraint Satisfaction problem by a global constraint Max-Sat. This section also introduces the global soft constraints and two general definitions of violation costs associated with global constraints. The soft alldiff constraint is taken as example. In Section 6 the quality of filtering algorithms is discussed and a classification is proposed. Some miscellaneous considerations about filtering algorithms are mentioned in Section 7. Finally, we conclude.

2 Global Constraints

2.1 Preliminaries

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **domains** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible **values** for variable x_i, and a set $\mathcal{C}$ of **constraints** between variables. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$. Indeed, we consider that any constraint network $\mathcal{N}$ can be associated with an initial domain $\mathcal{D}_0$ (containing $\mathcal{D}$), on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_1, \ldots, x_r$. An element of $D_0(x_1) \times \cdots \times D_0(x_r)$ is called a **tuple on** $X(C)$. $|X(C)|$ is the **arity** of C.

A value a for a variable x is often denoted by (x, a). $var(C, i)$ represents the i^{th} variable of $X(C)$, while $index(C, x)$ is the position of variable x in $X(C)$. $\tau[k]$ denotes the k^{th} value of the tuple τ. $\tau[index(C, x)]$ will be denoted by $\tau[x]$ when no confusion is possible. $D(X)$ denotes the union of domains of variables of X (i.e. $D(X) = \cup_{x_i \in X} D(x_i)$). $\#(a, \tau)$ is the number of occurrences of the value a in the tuple τ.

Let C be a constraint. A tuple τ on $X(C)$ is **valid** if $\forall (x, a) \in \tau, a \in D(x)$. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $a = \tau[index(C, x)]$. (τ is the called a **support** for (x, a) on C.) A constraint is **arc consistent** iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C.

A **filtering algorithm** associated with a constraint C is an algorithm which may remove some values that are inconsistent with C; and that does not remove any consistent values. If the filtering algorithm removes all the values inconsistent with C we say that it establishes the arc consistency of C.

The **propagation** is the mechanism that consists of calling the filtering algorithm associated with the constraints involving a variable x each time the domain of this variable is modified. Note that if the domains of the variables are finite, then this process terminates because a domain can be modified only a finite number of times.

2.2 Definition and Advantages

Global constraints are constraints that are equal to a set of other constraints:

Definition 1 *Let $\mathcal{C} = \{C_1, C_2, .., C_n\}$ be a set of constraints. The constraint C_G equals to the conjunction of all the constraints of $\mathcal{C}$: $C_G = \wedge\{C_1, C_2, ..C_n\}$ is a* **global constraint**.
The set of tuples of $\mathcal{C}$ is equal to the set of solutions of $(\cup_{C \in \mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, .., C_n\})$.

Global constraints are often defined from a set of variables and some prototypes of non-decomposable constraints. For instance, an alldiff constraint is just defined by: alldiff(X) which means that it corresponds to all the constraints $\neq$ stated for each pair of variables of X.

Global constraints have three main advantages:

- Expressiveness: it is more convenient to define one constraint corresponding to a set of constraints than to define independently each constraint of this set.
- Since a global constraint corresponds to a set of constraints it is possible to deduce some information from the simultaneous presence of constraints.
- Powerful filtering algorithms can be designed because the set of constraints can be taken into account as a whole. Specific filtering algorithms make it possible to use Operations Research techniques or graph theory.

The last point is emphasized by the following property:

Property 1 *The establishing of arc consistency on* $\mathcal{C} = \wedge\{C_1, C_2, .., C_n\}$ *is stronger (that is, cannot remove fewer values) than the establishing of arc consistency of the network* $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, .., C_n\})$.

proof: By Definition 1 the set of tuples of $\mathcal{C} = \wedge\{C_1, C_2, .., C_n\}$ corresponds to the set of solution of $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, .., C_n\})$. Therefore, the establishing of arc consistency of $\wedge\{C_1, C_2, .., C_n\}$ removes all the values that do not belong to a solution of $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, .., C_n\})$ which is stronger than the arc consistency of the previous network.

Therefore, arc consistency on global constraints is a strong property. The following proposition is an example of the gap between arc consistency for a global constraint and arc consistency for the network corresponding to this global constraint

Property 2 *Arc Consistency for* $C =$*alldiff*(X) *corresponds to the arc consistency of a Constraint Network with an exponential number of constraints defined by:*
$\forall A \subseteq X$*:* $|D(A)| = |A| \Rightarrow D(X - A)$ *is reduced to* $D(X) - D(A)$

proof: see [Régin, 1995].

2.3 Examples

The purpose of this section is not to be exhaustive, but to present some of the global constraints that are useful in pratice. We will give a short review of:

- cumulative
- diff-n
- cycle
- sort
- alldiff and permutation
- symmetric alldiff
- global cardinality
- global cardinality with costs
- sum and scalar product of alldiff variables
- sum and binary inequalities
- sequence
- stretch
- minimum global distance
- k-diff
- number of distinct values

Cumulative Constraint Here is the definition of the constraint cumulative(S, D, H, u) from [Beldiceanu and Contejean, 1994]: "The cumulative constraint matches directly the single resource scheduling problem, where the S variables correspond to the start of the tasks, the D variables to the duration of the tasks, and the H

variable to the heights of the resources that is the amounts of resource used by each task. The natural number u is the total amount of available resource that must be shared at any instant by the different tasks. The cumulative constraint states that, at any instant i of the schedule, the summation of the amount of resource of the tasks that overlap i does not exceed the upper limit u."

Definition 2 *Consider A a set of activities where each activity i is associated with 3 variables: s_i the start variable representing the start time of the activity, d_i the duration variable representing the duration of the activity, and h_i the consumption activity representing the amount of resource which is needed by the activity.*
A **cumulative constraint** *is a constraint C associated with a positive integer u and A a set of activities, such that:*
$T(C) = \{\ \tau$ *s.t.* τ *is a tuple of* $X(C)$
and $\forall i = [1, |A|] : \tau[s_i] = a \Leftrightarrow \sum_{j/\tau[s_j] \leq a \leq \tau[s_j]+\tau[d_j]-1} \tau[h_j] \leq u\}$

A filtering algorithm is detailed in [Beldiceanu and Carlsson, 2002]. Its complexity is $O(mn \log n + mnp)$ where m is the number of resources, n the number of tasks, and p the number of tasks that are not totally fixed. Other algorithms have been proposed for disjunctive scheduling problems. In this case, each resource can execute at most one activity at a time. For instance, the reader can consult [Baptiste et al., 1998],or [Carlier and Pinson, 1994] for a presentation of the edge-finder algorithm with the lowest worst case complexity so far.

Diff-n Constraint We present here only the diff-n/1 constraint. We quote [Beldiceanu and Contejean, 1994]: "The diff-n constraint was introduced in CHIP in order to handle multi-dimensional placement problems that occur in scheduling, cutting or geometrical placement problems. The intuitive idea is to extend the alldiff constraint which works on a set of domain variables to a nonoverlapping constraint between a set of objects defined in a n-dimensional space."

Definition 3 *Consider R a set of multidirectional rectangles. Each multidirectional rectangle i is associated with 2 set of variables $O_i = \{o_{i1}, .., o_{in}\}$ and $L_i = \{l_{i1}, .., l_{in}\}$. The variables of O_i represent the origin of the rectangle for every dimension, for instance the variable o_{ij} corresponds to the origin of the rectangle for the j^{th} dimension. The variables of L_i represent the length of the rectangle for every dimension, for instance the variable l_{ij} represents the length of the rectangle for the j^{th} dimension.*
A **diff-n constraint** *is a constraint C associated with a set R of multidirectional rectangles, such that:*
$T(C) = \{\ \tau$ *s.t.* τ *is a tuple of* $X(C)$
and $\forall i \in [1, m], \forall j \in [1, m], j \neq i, \exists k \in [1, n]$
s.t. $\tau[o_{ik}] \geq \tau[o_{jk}] + \tau[l_{jk}]$ *or* $\tau[o_{jk}] \geq \tau[o_{ik}] + \tau[l_{ik}]\}$

This constraint is mainly used for packing problems. In [Beldiceanu et al., 2001], an $O(d)$ filtering algorithm for the non-overlapping constraint between two d-dimensional boxes and so a filtering algorithm for the non-overlapping constraint between two convex polygons are presented.

Cycle Constraint We present here only the cycle/2 constraint. Here is the idea of this constraint [Beldiceanu and Contejean, 1994]: "The cycle constraint was introduced in CHIP to tackle complex vehicle routing problems. The cycle/2 constraint can be seen as the problem of finding N distinct circuits in a directed graph in such a way that each node is visited exactly once. Initially, each domain variable x_i corresponds to the possible successors of the i^{th} node of the graph."

Definition 4 *A* **cycle constraint** *is a constraint C associated with a positive integer n and defined on a set X of variables, such that:*
$T(C) = \{$ *τ s.t. τ is a tuple of $X(C)$*
and the graph defined from the arcs $(k, \tau[k])$
has n connected components
and every connected component is a cycle$\}$

This constraint is mentioned in the literature but no filtering algorithm is explicitly given. It is mainly used for vehicle routing problems or crew scheduling problems.

Sort Constraint This constraint has been proposed by [Bleuzen-Guernalec and Colmerauer, 1997]: "A sortedness constraint expresses that an $n-$tuple $(y_1, ..., y_n)$ is equal to the $n-$tuple obtained by sorting in increasing order the terms of another $n-$tuple $(x_1, ..., x_n)$".

Definition 5 *A* **sort constraint** *is a constraint C defined on two sets of variables $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_n\}$ such that*
$T(C) = \{\tau$ *s.t. τ is a tuple on $X(C)$ and $\exists f$ a permutation of $[1..n]$ s.t.*
$\forall i \in [1..n]\ \tau[x_{f(i)}] = \tau[y_i]\}$

The best filtering algorithm establishing bound consistency has been proposed by [Melhorn and Thiel, 2000]. Its running time is $O(n)$ plus the time required to sort the interval endpoints of the variables of X. If the interval endpoints are from an integer range of size $O(n^k)$ for some constant k the algorithm runs in linear time, because this sort becomes linear.

A sort constraint involving 3 sets of variables has also been proposed by [Zhou, 1996,Zhou, 1997]. The n added variables are used for making explicit a permutation linking the variables of X and those of Y. Well known difficult job shop scheduling problems have been solved thanks to this constraint.

Alldiff and Permutation Constraints The alldiff constraint constrains the values taken by a set of variables to be pairwise different. The permutation constraint is an alldiff constraint in which $|D(X(C))| = |X(C)|$.

Definition 6 *An* **alldiff constraint** *is a constraint C such that*
$T(C) = \{\tau$ *s.t. τ is a tuple on $X(C)$ and $\forall a_i \in D(X(C)) : \#(a_i, \tau) \leq 1\}$*

This constraint is used in a lot of real world problems like rostering or resource allocation. It is quite useful to express that two things cannot be at the

same place at the same moment.
A filtering algorithm establishing arc consistency for the alldiff is given in this chapter and also in [Régin, 1994]. Its complexity is in $O(m)$ with $m = \sum_{x \in X} |D(x)|$, after the computation of the consistency of the constraint which requires $O(\sqrt{n}m)$. When the domain of the variables are intervals, [Melhorn and Thiel, 2000] proposed a filtering algorithm establishing bound consistency with a complexity which is asymptotically the same as for sorting the internal endpoints. If the interval endpoints are from an integer range of size $O(n^k)$ for some constant k the algorithm runs in linear time. Therefore, Melhorn's algorithm is linear for a permutation constraint.

On the other hand, [Leconte, 1996] has proposed an algorithm which considers that the domains are intervals, but which can create "holes" in the domain. His filtering algorithm is in $O(n^2 d)$.

Symmetric Alldiff Constraint The symmetric alldiff constraint constrains some entities to be grouped by pairs. It is a particular case of the alldiff constraint, a case in which variables and values are defined from the same set S. That is, every variable represents an element e of S and its values represent the elements of S that are compatible with e. This constraint requires that all the values taken by the variables are different (similar to the classical alldiff constraint) and that if the variable representing the element i is assigned to the value representing the element j, then the variable representing the element j is assigned to the value representing the element i.

Definition 7 *Let X be a set of variables and σ be a one-to-one mapping from $X \cup D(X)$ to $X \cup D(X)$ such that*
$\forall x \in X$: $\sigma(x) \in D(X)$; $\forall a \in D(X)$: $\sigma(a) \in X$ and $\sigma(x) = a \Leftrightarrow x = \sigma(a)$.
A **symmetric alldiff constraint** *defined on X is a constraint C associated with σ such that:*
$T(C) = \{ \tau$ *s.t.* τ *is a tuple on* X
and $\forall a \in D(X) : \#(a, \tau) = 1$
and $a = \tau[index(C, x)] \Leftrightarrow \sigma(x) = \tau[index(C, \sigma(a))]\}$

This constraint has been proposed by [Régin, 1999b]. It is useful to be able to express certain items that should be grouped as pairs, for example in the problems of sports scheduling or rostering. Arc consistency can be established in $O(nm)$ after computing the consistency of the constraint which is equivalent to the search for a maximum matching in a non-bipartite graph, which can be performed in $O(\sqrt{n}m)$ by using the complex algorithm of [Micali and Vazirani, 1980].

In [Régin, 1999b], another filtering algorithm is proposed. It is difficult to characterize it but its complexity is $O(m)$ per deletion. In this paper, it is also shown how the classical alldiff constraint plus some additional constraints can be useful to solve the original problem. The comparison between this approach, the symmetric alldiff constraint, and the alldiff constraint has been carried out by [Henz et al., 2003].

Global Cardinality Constraint A global cardinality constraint (GCC) constrains the number of times every value can be taken by a set of variables. This is certainly one of the most useful constraints in practice. Note that the alldiff constraint corresponds to a GCC in which every value can be taken at most once.

Definition 8 *A* **global cardinality constraint** *is a constraint C in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i with $l_i \leq u_i$ and*
$T(C) = \{ \tau$ *s.t.* τ *is a tuple on* $X(C)$
and $\forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i\}$
It is denoted by $gcc(X, l, u)$.

This constraint is present in almost all rostering or car-sequencing problems. A filtering algorithm establishing arc consistency for this constraint is described in [Régin, 1996] and is detailed in this chapter. The consistency of the constraint can be checked in $O(nm)$ and the arc consistency can be computed in $O(m)$ providing that a maximum flow has been defined.

Global Cardinality Constraint with Costs A global cardinality constraint with costs (costGCC) is the conjunction of a GCC constraint and a sum constraint:

Definition 9 *A* **cost function on a variable set** X *is a function which associates with each value (x, a), $x \in X$ and $a \in D(x)$ an integer denoted by $cost(x, a)$.*

Definition 10 *A* **global cardinality constraint with costs** *is a constraint C associated with* `cost` *a cost function on $X(C)$, an integer H and in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i*
$T(C) = \{ \tau$ *s.t.* τ *is a tuple on* $X(C)$
and $\forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i$
and $\Sigma_{i=1}^{|X(C)|} cost(var(C, i), \tau[i]) \leq H$ $\}$
It is denoted by $costgcc(X, l, u, cost, H)$.

This constraint is used to model some preferences between assignments in resource allocation problems.
Note that there is no assumption made on the sign of costs.

The integration of costs within a constraint is quite important, especially to solve optimization problems, because it improves back-propagation, which is due to the modification of the objective variable. In other words, the domain of the variables can be reduced when the objective variable is modified. [Caseau and Laburthe, 1997] have used an alldiff constraint with costs, but only the consistency of the constraint has been checked, and no specific filtering has been used. The first proposed filtering algorithm comes from [Focacci et al., 1999a] and [Focacci et al., 1999b], and is based on reduced cost. A filtering algorithm establishing arc consistency has been proposed by [Régin, 1999a] and [Régin, 2002]. The consistency of this constraint can be checked by searching for a minimum

cost flow and arc consistency can be established in $O(|\Delta|S(m, n+d, \gamma))$ where $|\Delta|$ is the number of values that are taken by a variable in a tuple, and where $S(m, n+d, \gamma))$ is the complexity of the search for shortest paths from a node to every node in a graph with m arcs and n nodes with a maximal cost γ.

Sum and Scalar product of Alldiff Variables An interesting example of costGCC is the constraint on the sum of all different variables. More precisely, for a given set of variable X, this constraint is the conjunction of the constraint $\sum_{x_i \in X} x_i \leq H$ and alldiff(X). Similarly, we can define the constraint which is the conjunction of the constraint $\sum_{x_i \in X} \alpha_i x_i \leq H$ and alldiff(X).

Definition 11 *A* **scalar product of alldiff variables constraint** *is a constraint C associated with α a set of coefficients, one for each variable, an integer H, such that:*
$T(C) = \{\ \tau$ *s.t.* τ *is a tuple on* $X(C)$
and $\forall a_i \in D(X(C)) : \#(a_i, \tau) \leq 1$
and $\Sigma_{i=1}^{|X(C)|} \alpha_i \tau[i] \leq H\ \}$

The following model is used to compute arc consistency for this constraint [Régin, 1999a].
Let us define the boundaries and cost function as follows:

- For each value $a_i \in D(X)$ we define $l_i = 0$ and $u_i = 1$
- For each variable $x \in X$ and for each value $a \in D(x)$, $cost(x, a) = \alpha_i a$

Then, it is easy to prove that the costGCC constraint $costgcc(X, l, u, cost, H)$ represents the conjunction of the constraint $\sum_{x_i \in X} \alpha_i x_i \leq H$ and alldiff(X).

Therefore, establishing arc consistency for this constraint is equivalent to establish arc consistency to the costGCC constraint defined as above.

Note that we could generalize this constraint to deal with a global cardinality constraint defined on the variables instead of an alldiff constraint.

This constraint is used, for instance, to solve the golomb ruler problem.

Sum and Binary Inequalities Constraint This constraint is the conjunction of a sum constraint and a set of distance constraints, that is constraints of the form $x_j - x_i \leq c$.

Definition 12 *Let $SUM(X, y)$ be a sum constraint, and $\mathcal{I}_{neq}$ be a set of binary inequalities defined on X. The* **sum and binary inequalities constraint** *is a constraint C associated with $SUM(X, y)$ and $\mathcal{I}_{neq}$ such that:*
$T(C) = \{\ \tau$ *s.t.* τ *is a tuple of* $X \cup y$
and $(\sum_{i=1}^{|X|} \tau[i]) = \tau[y]$
and the values of τ *satisfy* $\mathcal{I}_{neq}\ \}$

This constraint has been proposed by [Régin and Rueher, 2000]. It is used to minimize the delays in scheduling applications.
Bound consistency can be computed in $O(n(m+n \log n))$, where m is the number of inequalities and n the number of variables. It is also instructive to remark that the bound consistency filtering algorithm still works when $y = \Sigma_{i=1}^{i=n} \alpha_i x_i$ where α is non-negative real number.

Sequence Constraint A global sequencing constraint C is specified in terms of a ordered set of variables $X(C) = \{x_1, ..., x_p\}$ which take their values in $D(C) = \{v_1, ..., v_d\}$, some integers q, min and max and a given subset V of $D(C)$. On one hand, a gsc constrains the number of variables in $X(C)$ instantiated to a value $v_i \in D(C)$ to be in an interval $[l_i, u_i]$. On the other hand, a gsc constrains for each sequence S_i of q consecutive variables of $X(C)$, that at least min and at most max variables of S_i are instantiated to a value of V.

Definition 13 *A* **global sequencing constraint** *is a constraint C associated with three positive integers min, max, q and a subset of values $V \subseteq D(C)$ in which each value $v_i \in D(C)$ is associated with two positive integers l_i and u_i and*
$T(C) = \{$ *t such that t is a tuple of $X(C)$*
and $\forall v_i \in D(C) : l_i \leq \#(v_i, t) \leq u_i$
and for each sequence S of q consecutive
variables: $min \leq \sum_{v_i \in V} \#(v_i, t, S) \leq max\}$

This constraint arises in car sequencing or in rostering problems. A filtering algorithm is described in [Régin and Puget, 1997]. Thanks to it, some problems of the CSP-Lib have been closed.

Stretch Constraint This constraint has been proposed by [Pesant, 2001]. This constraint can be seen as the opposite of the sequence constraint. The stretch constraint aims to group the values by sequence of consecutive values, whereas the sequence is often used to obtain a homogenous repartition of values.

A stretch constraint C is specified in terms of an ordered set of variables $X(C) = \{x_1, ..., x_p\}$ which take their values in $D(C) = \{v_1, ..., v_d\}$, and two set of integers $l = \{l_1, ..., l_d\}$ and $u = \{u_1, ..., u_d\}$, where every value v_i of $D(C)$ is associated with l_i the i^{th} integer of L and u_i the i^{th} integer of U. A stretch constraint states that if $x_j = v_i$ then x_j must belong to a sequence of consecutive variables that also take value v_i and the length of this sequence (the span of the stretch) must belong to the interval $[l_i, u_i]$.

Definition 14 *A* **stretch constraint** *is a constraint C associated with a subset of values $V \subseteq D(C)$ in which each value $v_i \in D(C)$ is associated with two positive integers l_i and u_i and*
$T(C) = \{$ *t s.t. t is a tuple of $X(C)$*
and $\forall x_j \in [1..|X(C)|], (x_j = v_i$ and $v_i \in D(C)) \Leftrightarrow \exists p, q$ with $q \geq p$,
$q - p + 1 \in [l_i, u_i]$ s.t. $j \in [p, q]$ and $\forall k \in [p, q]$ $x_k = v_i\}$

This constraint is used in rostering or in car sequencing problems (especially in the paint shop part).
A filtering algorithm has been proposed by [Pesant, 2001]. The case of cyclic sequence (that is, the successor of the last variable is the first one) is also taken into account by this algorithm. Its complexity is in $O(m^2 max(u) max(l))$. G. Pesant also described some filtering algorithms for some variations of this constraint, notably one that deals with patterns and constrains the successions of patterns (that is some patterns cannot immediately follow some other patterns).

Global Minimum Distance Constraint This constraint has been proposed by [Régin, 1997] and is mentioned in [ILOG, 1999]. A global minimum distance constraint defined on X, a set of variables, states that for any pair of variable x and y of X the constraint $|x - y| \geq k$ must be satisfied.

Definition 15 *A* **global minimum distance constraint** *is a constraint C associated with an integer k such that*
$T(C) = \{ \tau$ *s.t.* τ *is a tuple of* $X(C)$
and $\forall a_i, aj \in \tau : |a_i - a_j| \geq k\}$

This constraint is present in frequency allocation problems.
A filtering algorithm has been proposed for this constraint [Régin, 1997]. Note that there is a strong relation between this constraint and the sequence constraint. A $1/q$ sequence constraint constrained two variables assigned to the same value to be separated by at least $q - 1$ variables, in regard to the variable ordering. Here we want to select the values taken by a set of variables such that are all pairs of values are at least k units apart.

k-diff Constraint The k-diff constraint constrains the number of variables that are different to be greater than or equal to k.

Definition 16 *A* **k-diff constraint** *is a constraint C associated with an integer k such that*
$T(C) = \{\tau$ *s.t.* τ *is a tuple on* $X(C)$ *and*
$|\{a_i \in D(X(C))$ *s.t.* $\#(a_i, \tau) \leq 1\}| \geq k\}$

This constraint has been proposed by [Régin, 1995]. It is useful to model some parts of over-constrained problems where it corresponds to a relaxation of the alldiff constraint.
A filtering algorithm establishing arc-consistency is detailed in [Régin, 1995]. Its complexity is the same as for the alldiff constraint, because the filtering algorithm of the alldiff constraint is used when the cardinality of the maximum matching is equal to k. When this cardinality is strictly greater than k, we can prove that the constraint is arc consistent (see [Régin, 1995].)

Number of Distinct Values Constraint The number of distinct values constraint constrains the number of distinct values taken by a set of variables to be equal to another variable.

Definition 17 *An* **number of distinct values constraint** *is a constraint C defined on a variable y and a set of variables X such that*
$T(C) = \{\tau$ *s.t.* τ *is a tuple on* $X(C)$ *and*
$|\{a_i \in D(X(C))$ *s.t.* $\#(a_i, \tau) \leq 1\}| = \tau[y]\}$

This constraint is quite useful for modeling some complex parts of problems. A filtering algorithm based on the search of a lower bound of the dominating set problem [Damaschke et al., 1990] has been proposed by [Beldiceanu, 2001]. When all the domains of the variables are intervals this lead to an $O(n)$ algorithm, if the intervals are already sorted.

3 Filtering Algorithms

There are several ways to design a filtering algorithm associated with a constraint. However, for global constraints we can see at least three different and important types of filtering algorithms:

- the filtering algorithms based on constraints addition. That is, from the simultaneous presence of constraints the filtering algorithm consists of adding some new constraints.
- the filtering algorithms using the general filtering algorithm (GAC-Schema) establishing arc consistency. In this case, there is no new algorithm to write provided that an algorithm checking the consistency of the constraint is given.
- the dedicated filtering algorithms. That is, a custom-written filtering algorithm is designed in order to take into account and to exploit the structure of the constraint.

For convenience, we introduce the notion of pertinent filtering algorithm for a global constraint:

Definition 18 *A filtering algorithm associated with* $\mathcal{C} = \wedge\{C_1, C_2, .., C_n\}$ *is* **pertinent** *if it can remove more values than the propagation mechanism called on the network* $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, .., C_n\})$.

3.1 Algorithms Based on Constraints Addition

A simple way to obtain a pertinent filtering algorithm is to deduce from the simultaneous presence of constraints, some new constraints. In this case, the global constraint is replaced by a set of constraints that is a superset of the one defining the global constraint. That is, no new filtering algorithm is designed.

For instance, consider a set of 5 variables: $X = \{x_1, x_2, x_3, x_4, x_5\}$ with domains containing the integer values from 0 to 4; and four constraints $atleast(X, 1, 1)$, $atleast(X, 1, 2)$, $atleast(X, 1, 3)$, and
$atleast(X, 1, 4)$ which mean that each value of $\{1, 2, 3, 4\}$ has to be taken at least one time by a variable of X in every solution.

An $atleast(X, \#time, val)$ constraint is a local constraint. If such a constraint is considered individually then the value val cannot be removed while it belongs to more than one domain of a variable of X. A filtering algorithm establishing arc consistency for this constraint consists of assigning a variable x to val if and only if x is the only one variable whose domain contains val.

Thus, after the assignments $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$, no failure is detected. The domains of x_4 and x_5, indeed, remain the same because every value of $\{1, 2, 3, 4\}$ belongs to these two domains. Yet, there is obviously no solution including the previous assignments, because 4 values must be taken at least 1 time and only 2 variables can take them.

For this example we can deduce another constraint by applying the following property: If 4 values must be taken at least 1 time by 5 variables, then the other values can be taken at most $5 - 4 = 1$, that is we have $atmost(x, 1, 0)$.

This idea can be generalized for a $gcc(X, l, u)$. Let $card(a_i)$ be a variable associated with each value a_i of $D(X)$ which counts the number of domains of X that contain a_i. We have $l_i \leq card(a_i) \leq u_i$. Then, we can simply deduce the constraint $\sum_{a_i \in D(X)} card(a_i) = |X|$; and each time the minimum or the maximum value of $card(a_i)$ is modified, the values of l_i and u_i are accordingly modified and the GCC is modified.

This method is usually worthwhile because it is easy to implement. However, the difficulty is to find the constraints that can be deduced from the simultenous presence of other constraints.

3.2 General Arc Consistency Filtering Algorithm

The second way to easily define a powerful filtering algorithm, but which may be time consuming, is to use the general arc consistency algorithm [Bessière and Régin, 1997].

In constraint programming, to solve a problem, we begin by designing a model using predefined constraints, such as sum, alldiff, and so on. Next, we define other constraints specific to the problem. Then we call a procedure to search for a solution.

Often when we are solving a real problem, say $\mathcal{P}$, the various simple models that we come up with cannot be solved within a reasonable period of time. In such a case, we may consider a sub-problem of the original problem, say $\mathcal{R}$. We then try to improve the resolution of $\mathcal{R}$ with the hope of thus eventually solving $\mathcal{P}$. That is, we try to identify sub-problems of $\mathcal{P}$ where the resolution can be improved by defining a particular constraint for each of these sub-problems along with a filtering algorithm associated with these constraints.

More precisely, for each possible relevant sub-problem of $\mathcal{P}$, we construct a *global* constraint that is the conjunction of the constraints involved in the sub-problem. Suppose that we then apply arc consistency to these new constraints and that this improves the resolution of $\mathcal{P}$ (i.e., the number of backtracks markedly decreases). In this case, we know that it is worthwhile to write another algorithm dedicated to solving the sub-problem $\mathcal{R}$ under consideration. In contrast, if the number of backtracks decreases only slightly, then we know that the resolution of $\mathcal{R}$ has only a modest effect on the resolution of $\mathcal{P}$. By proceeding in this way, we can improve the resolution of $\mathcal{P}$ much faster. Therefore, a general algorithm can be really useful in practice.

Preliminaries Suppose that you are provided with a function, denoted by existSolution(P), which is able to know whether a particular problem $P = (X, \mathcal{C}, \mathcal{D})$ has a solution or not. In this section, we present two general filtering algorithms establishing arc consistency for the constraint corresponding to the problem, that is the global constraint $C(P) = \wedge\mathcal{C}$

These filtering algorithms correspond to particular instantiations of a more general algorithm: GAC-Schema [Bessière and Régin, 1997].

For convenience, we will denote by $P_{x=a}$ the problem P in which it is imposed that $x = a$, in other words $P_{x=a} = (X, \mathcal{C} \cup \{x = a\}, \mathcal{D})$.

Establishing arc consistency on $C(P)$ is done by looking for supports for the values of the variables in X. A support for a value (y, b) on $C(P)$ can be searched by using any search procedure since a support for (y, b) is a solution of problem $P_{y=b}$.

A First Algorithm A simple algorithm consists of calling the function existSolution with $P_{x=a}$ as a parameter for every value a of every variable x involved in P, and then to remove the value a of x when existSolution($P_{x=a}$) has no solution. Algorithm 1 is a possible implementation.

Algorithm 1: Simple general filtering algorithm establishing arc consistency

```
SimpleGeneralFilteringAlgorithm(C(P): constraint; deletionSet: list): Bool
for each a ∈ X do
    for each a ∈ D(x) do
        if ¬ existSolution(P_{x=a}) then
            remove a from D(x)
            if D(x) = ∅ then return False
            add (y, b) to deletionSet
return True
```

This algorithm is quite simple but it is not efficient because each time a value will be removed, the existence of a solution for all the possible assignments needs to be recomputed.

If $O(P)$ is the complexity of function existSolution(P) then we can recapitulate the complexity of this algorithms as follows:

	Consistency checking		Establishing Arc consistency	
	best	worst	best	worst
From scratch	$\Omega(P)$	$O(P)$	$nd \times \Omega(P)$	$nd \times O(P)$
After k modifications	$k \times \Omega(P)$	$k \times O(P)$	$knd \times \Omega(P)$	$knd \times O(P)$

A better general algorithm This section shows how a better general algorithm establishing arc consistency can be designed provided that function existSolution(P) returns a solution when there is one instead of being Boolean.

First, consider that a value (x, a) has been removed from $D(x)$. We must study the consequences of this deletion. So, for all the values (y, b) that were supported by a tuple containing (x, a) another support must be found. The list

Algorithm 2: function GeneralFilteringAlgorithm

```
GeneralFilteringAlgorithm(C(P): constraint; x: variable; a: value,
deletionSet: list): Bool
1  for each τ ∈ S_C(x, a) do
       for each (z, c) ∈ τ do remove τ from S_C(z, c)
2      for each (y, b) ∈ S(τ) do
           remove (y, b) from S(τ)
           if b ∈ D(y) then
3              σ ← seekInferableSupport(y, b)
               if σ ≠ nil then add (y, b) to S(σ)
               else
4                  σ ← existSolution(P_{y=b})
                   if σ ≠ nil then
                       add (y, b) to S(σ)
                       for k = 1 to |X(C)| do add σ to S_C(var(C(P), k), σ[k])
                   else
                       remove b from D(y)
                       if D(y) = ∅ then return False
                       add (y, b) to deletionSet
return True
```

of the tuples containing (x, a) and supporting a value is the list $S_C(x, a)$; and the values supported by a tuple τ is given by $S(\tau)$.

Therefore, Line 1 of Algorithm 2 enumerates all the tuples in the S_C list and Line 2 enumerates all the values supported by a tuple. Then, the algorithm tries to find a new support for these values either by "inferring" new ones (Line 3) or by explicitly calling function existSolution (Line 4).

Here is an example of this algorithm:
Consider $X = \{x_1, x_2, x_3\}$ and $\forall x \in X, D(x) = \{a, b\}$;
and $T(C(P)) = \{(a, a, a), (a, b, b), (b, b, a), (b, b, b)\}$ (i.e. these are the possible solutions of P.)
First, a support for (x_1, a) is sought: (a, a, a) is computed and (a, a, a) is added to $S_C(x_2, a)$ and $S_C(x_3, a)$, (x_1, a) in (a, a, a) is added to $S((a, a, a))$.
Second, a support for (x_2, a) is sought: (a, a, a) is in $S_C(x_2, a)$ and it is valid, therefore it is a support. There is no need to compute another solution.
Then a support is searched for all the other values.
Now, suppose that value a is removed from x_2, then all the tuples in $S_C(x_2, a)$ are no longer valid: (a, a, a) for instance. The validity of the values supported by this tuple must be reconsidered, that is the ones belonging to $S((a, a, a))$, so a new support for (x_1, a) must be searched for and so on...

The program which aims to establish arc consistency for $C(P)$ must create and initialize the data structures (S_C, S), and call function GeneralFilteringAlgorithm$(C(P), x, a, deletionSet)$ (see Algorithm 2) each

time a value a is removed from a variable x involved in $C(P)$, in order to propagate the consequences of this deletion. *deletionSet* is updated to contain the deleted values not yet propagated. S_C and S must be initialized in a way such that:

- $S_C(x, a)$ contains all the allowed tuples τ that are the current support for some value, and such that $\tau[index(C(P), x)] = a$.
- $S(\tau)$ contains all values for which τ is the current support.

Function seekInferableSupport of GeneralFilteringAlgorithm "infers" an already checked allowed tuple as support for (y, b) if possible, in order to ensure that it never looks for a support for a value when a tuple supporting this value has already been checked. The idea is to exploit the property: "If (y, b) belongs to a tuple supporting another value, then this tuple also supports (y, b)". Therefore, elements in $S_C(y, b)$ are good candidates to be a new support for (y, b). Algorithm 3 is a possible implementation of this function.

Algorithm 3: function seekInferableSupport

```
seekInferableSupport(y: variable, b: value): tuple
while S_C(y, b) ≠ ∅ do
    σ ← first(S_C(y, b))
    if σ is valid then return σ     /* σ is a support */
    else remove σ from S_C(y, b)
return nil
```

The complexity of the GeneralFilteringAlgorithm is given in the following table:

	Consistency checking		Establishing Arc consistency	
	best	worst	best	worst
From scratch	$\Omega(P)$	$O(P)$	$nd \times \Omega(P)$	$nd \times O(P)$
After k modifications	$\Omega(1)$	$k \times O(P)$	$nd \times \Omega(P)$	$knd \times O(P)$

Moreover, the space complexity of this algorithm is $O(n^2 d)$, where d is the size of the largest domain and n is the number of variables involved in the constraint. This space complexity depends on the number of tuples needed to support all the values. Since there are nd values and only one tuple is required per value, we obtain the above complexity.

Discussion and Example Algorithm 2 can be efficiently improved, if the search for a solution of P can be made according to a predefined ordering of the tuple. In this case, a more complex algorithm can be designed. Moreover, it is also possible to use the solver in itself to search for a solution in P. All these algorithms are fully detailed in [Bessière and Régin, 1997] and [Bessière and Régin, 1999]. These papers also detail how Algorithm 2 can be adapted to constraints that are given by the list of tuples that satisfy the constraint (in this case the resolution of P corresponds to the search for a valid tuple

in that list) or by the list of forbidden combinations of value for the constraint (i.e. the complement of the previous list).

[Bessière and Régin, 1999] have proposed to study a configuration problem as an example of the application of the general filtering algorithm establishing arc consistency. The general formulation is: given a supply of components and bins of given types, determine all assignments of components to bins satisfying specified assignment constraints subject to an optimization criterion.

In the example we will consider that there are 5 types of components: {glass, plastic, steel, wood, copper}. There are three types of bins: {red, blue, green} whose capacity constraints are: red has capacity 5, blue has capacity 5, green has capacity 6.
The containment constraints are:

- red can contain glass, copper, wood
- blue can contain glass, steel, copper
- green can contain plastic, copper, wood

The requirement constraints are (for all bin types): wood requires plastic.
Certain component types cannot coexist: glass excludes copper

Certain bin types have capacity constraints for certain components:

- red contains at most 1 of wood
- green contains at most 2 of wood
- for all the bins there is either no plastic or at least 2 plastic.

Given an initial supply of 12 of glass, 10 of plastic, 8 of steel, 12 of wood, and 8 of copper, what is the minimum total number of bins required to contain the components?

A description of a possible implementation of a similar problem is given in [ILOG, 1999]. We will call it the "standard model".

Almost all the constraints between types of bins and components are local. The filtering algorithm associated with them leads to few domain reductions. Therefore, they can be grouped inside a single global constraint. That is, problem P is formed by all these constraints and P is solved by using another CP solver.

Here are the results we obtained:

	# Backtracks	time (s)
standard model	1,361,709	430
new algorithm	12,659	11

These results clearly show the advantages of global constraints and prove that a general filtering algorithm establishing arc consistency may be useful in practice to solve some real life problems. However, in practice, when the problems become big the complexity of the GAC-Schema often prevents its use, and specific filtering algorithm establishing arc consistency have to be used. In [Bessière and Régin, 1999] some other examples show by using GAC-Schema that sometimes arc consistency is useless. Even in this case the search for a good model is improved because wrong models can be identified more quickly.

3.3 Dedicated Filtering Algorithms

The third method to design a pertinent filtering algorithm is to use the structure of the constraint in order to define some properties identifying that some values are not consistent with the global constraint.

The use of the structure of a constraint has four main advantages:

- The search for a support can be speeded up.
- Some inconsistent values can be identified without explicitly checking for every value whether it has a support or not.
- The call of the filtering algorithm, that is the needed to check the consistency of some values, can be limited to some events that can be clearly identified.
- Advantages of possible incrementality.

For instance, consider the constraint $(x < y)$. Then:

- The search for a support for a value a of $D(x)$ is immediate because any value b of $D(y)$ such that $b > a$ is a support, so a is consistent with the constraint if $a < max(D(y))$.
- We can immediately state that $max(D(x)) < max(D(y))$ and $min(D(y)) > min(D(x))$ which mean that all values of $D(x)$ greater than or equal to $max(D(y))$ and all values of $D(y)$ less than or equal to $min(D(x))$ can be removed.
- Since the deletions of values of $D(y)$ depends only on $max(D(y))$ and the deletions of values of $D(x)$ depends only on $min(D(x))$, the filtering algorithm must be called only when $max(D(y))$ or $min(D(x))$ are modified. It is useless to call it for the other modifications.

We propose an original contribution for a well-known problem: the n-queens problem.

The n-queens problem involves placing n queens on a chess board in such a way that none of them can capture any other using the conventional moves allowed by a queen. In other words, the problem is to select n squares on a chessboard so that any pair of selected squares is never aligned vertically, horizontally, nor diagonally.

This problem is usually modeled by using one variable per queen; the value of this variable represents the column in which the queen is set. If x_i represents the variable corresponding to queen i (that is the queen in row i) the constraints can be stated in the following way. For every pair (i, j), with $i \neq j$, $x_i \neq x_j$ guarantees that the columns are distinct; and $x_i + i \neq x_j + j$ and $x_i - i \neq x_j - j$ together guarantee that the diagonals are distinct.
These relations are equivalent to defining an alldiff constraint on the variables x_i, an alldiff constraint on the variables $x_i + i$, and an alldiff constraint on the variables $x_i - i$.

We propose to use a specific constraint that is defined on x_i and try to take into account the simultaneous presence of three alldiff constraints. Consider a queen q: if there are more than three values in its domain, this queen cannot

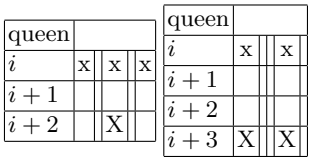

Fig. 3. Rules of the ad-hoc filtering algorithm for the n-queens problem.

lead to the deletion of one value of another queen, because three directions are constrained (the column and the two diagonals) and so at least one value of queen q does not belong to one of these directions. Therefore, a first rule can be stated:

• While a queen has more than three values in its domain, it is useless to study the consequence of the deletion of one of its values.

From a careful study of the problem we can deduce some rules (see Figure 3):

• If a queen i has 3 values $\{a, b, c\}$, with $a < b < c$ in its domain then the value b of queens $i - k$ and the value b of queen $i + k$ can be deleted if $b = a + k$ and $c = b + k$;

• If $D(x_i) = \{a, b\}$ with $a < b$, then the values a and b of queens $i - (b - a)$ and of queens $i + (b - a)$ can be deleted.

• If $D(x_i) = \{a\}$, then the value $a + j$ for all queens $i + j$, and the value $a - j$ for all queens $i - j$ can be deleted.

• While a queen has more than 3 values in its domain, then this constraint cannot deduce anything.

Therefore, a careful study of a constraint can lead to efficient filtering algorithms. This method is certainly the most promising way. However, it implies a lot of work. In [Bessière and Régin, 1999], it is proposed to try to use first the general arc consistency algorithm in order to study if the development of a powerful filtering algorithm could be worthwhile for the considered problem. Using the solver itself then solves the consistency of the constraint.

4 Two Successful Filtering Algorithms

In this section, the filtering algorithms associated with two of the most frequently used constraints in practice - the alldiff and the global cardinality constraint - are presented. The advantages of these filtering algorithms is that they clearly show how Operational Research algorithms can be integrated into Constraint Programming.

4.1 Preliminaries

The definitions about graph theory are from [Tarjan, 1983]. The definitions, theorems and algorithms about flow are based on books of [Berge, 1970,Lawler, 1976,Tarjan, 1983,Ahuja et al.,

A **directed graph** or **digraph** $G = (X, U)$ consists of a **node set** X and an **arc set** U, where every arc (u, v) is an ordered pair of distinct nodes. We will denote by $X(G)$ the node set of G and by $U(G)$ the arc set of G.

A **path** from node v_1 to node v_k in G is a list of nodes $[v_1, ..., v_k]$ such that (v_i, v_{i+1}) is an arc for $i \in [1..k-1]$. The path **contains** node v_i for $i \in [1..k]$ and arc (v_i, v_{i+1}) for $i \in [1..k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $v_1 = v_k$.

If $\{u, v\}$ is an edge of a graph, then we say that u and v are the **ends** or the **extremities** of the edge. A **matching** M on a graph is a set of edges no two of which have a common node. The **size** $|M|$ of M is the number of edges it contains. The **maximum matching problem** is that of finding a matching of maximum size. M **covers** X when every node of X is an endpoint of some edge in M.

Let M be a matching. An edge in M is a **matching edge**; every edge not in M is **free**. A node is **matched** if it is incident to a matching edge and **free** otherwise.

An **alternating path or cycle** is a simple path or cycle whose edges are alternately matching and free. The **length** of an alternating path or cycle is the number of edges it contains.

Let G be a graph for which each arc (i, j) is associated with two integers l_{ij} and u_{ij}, respectively called the **lower bound capacity** and the **upper bound capacity** of the arc.

A **flow** in G is a function f satisfying the following two conditions:

- For any arc (i, j), f_{ij} represents the amount of some commodity that can "flow" through the arc. Such a flow is permitted only in the indicated direction of the arc, i.e., from i to j. For convenience, we assume $f_{ij} = 0$ if $(i, j) \notin U(G)$.
- A **conservation law** is observed at each node: $\forall j \in X(G) : \sum_i f_{ij} = \sum_k f_{jk}$.

We will consider two problems of flow theory:

- **the feasible flow problem**: Does there exist a flow in G that satisfies the **capacity constraint**? That is, find f such that $\forall (i, j) \in U(G)\ l_{ij} \leq f_{ij} \leq u_{ij}$.

- **the problem of the maximum flow for an arc** (i, j): Find a feasible flow in G for which the value of f_{ij} is maximum.

Without loss of generality (see p.45 and p.297 in [Ahuja et al., 1993]), and to overcome notation difficulties, we will consider that:

- if (i, j) is an arc of G then (j, i) is not an arc of G.
- all boundaries of capacities are nonnegative integers.

In fact, if all the upper bounds and all the lower bounds are integers and if there exists a feasible flow, then for any arc (i, j) there exists a maximum flow from j to i which is integral on every arc in G (See [Lawler, 1976] p113.)

The **value graph** [Laurière, 1978] of an non-binary constraint C is the bipartite graph $GV(C) = (X(C), D(X(C)), E)$ where $(x, a) \in E$ iff $a \in D(x)$.

4.2 The Alldiff Constraint

Consistency and Arc Consistency We have the relation [Régin, 1994]:

Proposition 1 *Let C be an alldiff constraint.*
A matching which covers X in the value graph of C is a tuple of $T(C)$.

Therefore we have:

Proposition 2 *A constraint C=alldiff(X) is consistent iff there exists a matching that covers $X(C)$ in $GV(C)$.*

From proposition 2 and by the definition of arc consistency, we have:

Proposition 3 *A value a of a variable x is consistent with C if and only if the edge $\{x, a\}$ belongs to a matching that covers $X(C)$ in $GV(C)$.*

Proposition 4 ([Berge, 1970]) *An edge belongs to some but not all maximum matchings, iff, for an arbitrary maximum matching, it belongs to either an even alternating path which begins at a free node, or an even alternating cycle.*

Proposition 5 *Given a bipartite graph $G = (X, Y, E)$ with a matching M which covers X and the graph $O(G, M) = (X \cup \{s\}, Y, Succ)$, obtained from G by orienting the edge in M from their y-endpoint to their x-endpoint, the edge not in M from their x-endpoint to their y-endpoint, and by adding an arc from every free node of Y to every matched node of Y. Then, we have the two properties*
1) Every directed cycle of $O(G, M)$ which does not contain an arc from a free node of Y to a matched node of Y corresponds to an even alternating cycle of G, and conversely.
2) Every directed cycle of $O(G, M)$ which contains an arc from a free node of Y to a matched node of Y corresponds to an even alternating path of G which begins at a free node, and conversely.

proof
1) G and $O(G, M)$ are bipartite and by definition of $O(G, M)$ the first property holds.
2) $O(G, M)$ is bipartite therefore all directed cycles of $O(G, M)$ are even. An even alternating path which begins at a free node y_f in Y, necessarily ends at a matched node y_m in Y, because all nodes of X are matched and in $O(G, M)$ there is only one arc from a node x in X to a node in Y: the matching edge involving x. Hence, by definition of $O(G, M)$ there is an arc from s to y_f and an arc from y_m to s, so every even alternating path of G is a directed cycle in $O(G, M)$. Conversely, a directed cycle involving s can be decomposed into a path from a free node y_f in Y to a node y_m in Y and the path $[y_m, s, y_f]$. Since the cycle is even the path is also even and it corresponds to an alternating path of G by definition of $O(G, M)$. Therefore the property holds.

From this proposition we immediately have:

Proposition 6 *Arc consistency of an alldiff constraint C is established by computing M a matching which covers $X(C)$ in $GV(C)$ and by removing all the values (x, a) such that $(x, a) \notin M$ and a and x belong to two different strongly connected components of $O(GV(C), M)$.*

proof: By definition of the strongly connected components, there exists a cycle between two nodes belonging to the same strongly connected components. Therefore, from Proposition 5 the proposition holds.

Complexity Let m be the number of edges of $GV(C)$, and $n = |X(C)|$ and $d = |D(X(C))|$. A matching covering $X(C)$ can be computed, or we can prove there is none, in $O(\sqrt{n}\,m)$ [Hopcroft and Karp, 1973]. The search for strongly connected component can be performed in $O(m+n+d)$. Hence arc consistency for an alldiff constraint can be established in $O(m+n+d)$.

Moreover, consider M a matching which covers X and suppose that k edges of the value graph are deleted (this means that k values have been removed from the domain of their variable). Then a new matching which covers X can be recomputed from M in $O(\sqrt{k}\,m)$ and arc consistency can be established in $O(m+n+d)$.

It is important to note that arc consistency may remove $O(n^2)$ values [Puget, 1998]. For instance, consider an alldiff constraint defined on $X = \{x_1, ..., x_n\}$ with the domains: $\forall i \in [1, \frac{n}{2}]$, if i is odd then $D(x_i) = [2i-1, 2i]$ else $D(x_i) = D(x_{i-1})$; and $\forall i \in [\frac{n}{2}+1, n]$ $D(x_i) = [1, n]$. For instance, for $n = 12$ we will have: $D(x_1) = D(x_2) = [1, 2]$, $D(x_3) = D(x_4) = [5, 6]$, $D(x_5) = D(x_6) = [9, 10]$, $D(x_7) = D(x_8) = D(x_9) = D(x_10) = D(x_11) = D(x_12) = [1, 12]$. Then, if arc consistency is established, the intervals corresponding to the domains of the variables from x_1 ro $x_{\frac{n}{2}}$ will be removed from the domains of the variables from $x_{\frac{n}{2}+1}$ to x_n. That is, $2 \times \frac{n}{2}$ values will be effectively removed from the domains of $(n-(\frac{n}{2}+1))$ variables. Therefore $O(n^2)$ values are deleted. Since m is bounded by n^2, the filtering algorithm establishing arc consistency for the alldiff constraint can be considered as an optimal algorithm.

The complexities are reported here:

	Consistency checking	Establishing Arc consistency
From scratch	$O(\sqrt{n}\,m)$	$O(m+n+d)$
After k modifications	$O(\sqrt{k}\,m)$	$O(m+n+d)$

Two important works carried out for the alldiff constraint must be mentioned. [Melhorn and Thiel, 2000] have proposed a very efficient filtering algorithm establishing bound consistency for the sort and alldiff constraint. A linear complexity is reached in a lot of practical cases (for a permutation, for instance). [Stergiou and Walsh, 1999] made a comparison between different filtering algorithm associated with the alldiff constraints and showed the advantages of this constraint in practice.

Some Results A graph-coloring problem consists of choosing colors for the nodes of a graph so that adjacent nodes are not the same color. Since we want to highlight the advantages of the filtering algorithm establishing arc consistency for the alldiff constraint we will consider only a very special kind of graph for this example.

The kind of graph that we will color is one with $n * (n-1)/2$ nodes, where n is odd and where every node belongs to exactly two maximal cliques of size n.

For example, for $n = 5$, there is a graph consisting of the following maximal cliques:
$c0 = \{0, 1, 2, 3, 4\}$, $c1 = \{0, 5, 6, 7, 8\}$, $c2 = \{1, 5, 9, 10, 11\}$
$c3 = \{2, 6, 9, 12, 13\}$, $c4 = \{3, 7, 10, 12, 14\}$, $c5 = \{4, 8, 11, 13, 14\}$

The minimum number of colors needed for this graph is n since there is a clique of size n. Consequently, our problem is to find out whether there is a way to color such a graph in n colors.

We compare the results obtained with the alldiff constraint and without it (that is only binary constraints of difference are used). Times are expressed in seconds:

	clique size							
	27		31		51		61	
	#fails	time	#fails	time	#fails	time	#fails	time
binary $\neq$	1	0.17	65	0.37	24512	66.5	?	$> 6h$
alldiff	0	1.2	4	2.2	501	25.9	5	58.2

These results show that using global constraints establishing arc consistency is not systematically worthwhile when the size of the problem is small, even if the number of backtracks is reduced. However, when the size of problem is increased, efficient filtering algorithm are needed.

4.3 The Global Cardinality Constraint

Consistency and Arc Consistency A GCC C is consistent iff there is a flow in an directed graph $N(C)$ called the value network of C [Régin, 1996]:

Definition 19 *Given $C = gcc(X, l, u)$ a GCC; the* **value network** *of C is the directed graph $N(C)$ with lower bound capacity and upper bound capacity on each arc. $N(C)$ is obtained from the value graph $GV(C)$, by:*

- *orienting each edge of $GV(C)$ from values to variables. For such an arc (u, v): $l_{uv} = 0$ and $u_{uv} = 1$.*
- *adding a node s and an arc from s to each value. For such an arc (s, a_i): $l_{sa_i} = l_i$, $u_{sa_i} = u_i$.*
- *adding a node t and an arc from each variable to t. For such an arc (x, t): $l_{xt} = 1$, $u_{xt} = 1$.*
- *adding an arc (t, s) with $l_{ts} = u_{ts} = |X(C)|$.*

Proposition 7 *Let C be a GCC and $N(C)$ be the value network of C; the following two properties are equivalent:*

- *C is consistent;*
- *there is a feasible flow in $N(C)$.*

sketch of proof: We can easily check that each tuple of $T(C)$ corresponds to a flow in $N(C)$ and conversely. $\odot$

Definition 20 *The* **residual graph** *for a given flow f, denoted by $R(f)$, is the digraph with the same node set as in G. The arc set of $R(f)$ is defined as follows: $\forall (i,j) \in U(G)$:*

• *$f_{ij} < u_{ij} \Leftrightarrow (i,j) \in U(R(f))$ and has cost $rc_{ij} = c_{ij}$ and upper bound capacity $r_{ij} = u_{ij} - f_{ij}$.*

• *$f_{ij} > l_{ij} \Leftrightarrow (j,i) \in U(R(f))$ and has cost $rc_{ji} = -c_{ij}$ and upper bound capacity $r_{ji} = f_{ij} - l_{ij}$.*

All the lower bound capacities are equal to 0.

Proposition 8 *Let C be a consistent GCC and f be a feasible flow in $N(C)$. A value a of a variable x is not consistent with C if and only if $f_{ax} = 0$ and a and x do not belong to the same strongly connected component in $R(f)$.*

proof: It is well known in flow theory that the flow value for an arc (a,x) is constant if there is no path from a to x in $R(f) - \{(a,x)\}$ and no path from x to a in $R(f) - \{(x,a)\}$. Moreover, $u_{ax} = 1$ thus (a,x) and (x,a) cannot belong simultaneously to $R(f)$, hence f_{ax} is constant iff there is no cycle containing (x,a) or (a,x) in $R(f)$. That is, if x and a belong to different strongly connected components. $\odot$

The advantage of this proposition is that all the values not consistent with the GCC can be determined by only one identification of the strongly connected components in $R(f)$.

Complexity For our problem, a feasible flow can be computed in $O(nm)$ therefore we have the same complexity for the check of the constraint consistency. Moreover flow algorithms are incremental.

The search for strongly connected components can be done in $O(m+n+d)$ [Tarjan, 1983], thus a good complexity for computing arc consistency for a GCC is obtained.

Corollary 1 *Let C be a consistent GCC and f be a feasible flow in $N(C)$. Arc consistency for C can be established in $O(m+n+d)$.*

Here is a recapitulation of the complexities:

	Consistency	Arc consistency
From scratch	$O(n\,m)$	$O(m+n+d)$
After k modifications	$O(k\,m)$	$O(m+n+d)$

Some results This section considers the sport-scheduling problem described in [McAloon et al., 1997] and in [Van Hentenryck et al., 1999]. The problem consists of scheduling games between n teams over $n-1$ weeks. In addition, each week is divided into $n/2$ periods. The goal is to schedule a game for each period of every week so that the following constraints are satisfied:

1. Every team plays against every other team;
2. A team plays exactly once a week;
3. A team plays at most twice in the same period over the course of the season.

The meeting between two teams is be called a *matchup* and takes place in a *slot* i.e. in a particular period in a particular week.

The following table gives a solution to this problem for 8 teams:

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	0 vs 1	0 vs 2	4 vs 7	3 vs 6	3 vs 7	1 vs 5	2 vs 4
Period 2	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
Period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
Period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

In fact, the problem can be made more uniform by adding a "dummy" final week and requesting that all teams play exactly twice in each period. The rest of this section considers this equivalent problem for simplicity.

The sport-scheduling problem is an interesting application for constraint programming. On the one hand, it is a standard benchmark (submitted by Bob Daniel) to the well known MIP library and it is claimed in [McAloon et al., 1997] that state of the art MIP solvers cannot find a solution for 14 teams. The model presented in this section is computationally much more efficient. On the other hand, the sports scheduling application demonstrates fundamental features of constraint programming including global and symbolic constraints. In particular, the model makes heavy use of arc consistency for the GCCs.

The main modeling idea is to use two classes of variables: team variables that specify the team playing on a given week, period, and slot and the matchup variables specifying which game is played on a given week and period. The use of matchup variables makes it simple to state the constraint that every team must play against each other team. Games are uniquely identified by their two teams. More precisely, a game consisting of home team h and away team a is uniquely identified by the integer $(h1) * n + a$.

These two sets of variables must be linked together to make sure that the matchup and team variables for a given period and a given week are consistent. This link is ensured by a constraint whose set of tuples is explicitly given. For 8 teams, this set consists of tuples of the form $(1, 2, 1)$ (which means that the game 1 vs 2 is the game number 1), $(1, 3, 2)$, ..., $(7, 8, 55)$.

The games that are played in a given week can be determined by using a round robin schedule. As a consequence, once the round robin schedule is selected, it is only necessary to determine the period of each game, not its schedule week. In addition, it turns out that a simple round robin schedule makes it possible to find solutions for large numbers of teams.

The basic idea is to fix the set of games of each week, but without fixing the period of each game. The goal is then to assign a period to each game such

that the constraints on periods are satisfied. If there is no solution then another round robin is selected.

The constraints on periods are taken into account with the global cardinality constraints. For every period a GCC is defined on the team variables involved in the period. Every value is associated with the two integers: 0 and 2 if the dummy week is not considered, otherwise if the team variables of the dummy week are involved the two integers are equal to 2.

The search procedure which is used consists of generating values for the matchups in the first period and in the first week, then in the second period and the second week, and so on. The results obtained by this method implemented with ILOG Solver are given in the following table. Times are expressed in seconds and the experiments have been run on a Pentium III 400Mhz machine. As far as we know, this method gives the best results for this problem.

#teams	8	10	12	14	16	18	20	24	30	40
#fails	10	24	58	21	182	263	226	2,702	11,895	2,834,754
time	0	0	0.2	0.2	0.6	0.9	1.2	10.5	138	6h

5 Global Constraints and Over-constrained Problems

Global constraints have been proved to be very useful in modelling and in improving the resolution of CSPs. This section aims to show that they can also be useful to model and to improve the resolution of over-constrained problems.

A problem is over-constrained when no assignment of values to variables satisfies all constraints. In this situation, the goal is to find a compromise. Violations are allowed in solutions, providing that such solutions retain a practical advantage. Therefore, it is mandatory to respect some rules and acceptance criteria defined by the user. Usually the set of initial constraints is divided into two sets: the hard constraints, that is the ones that must necessarily be satisfied, and the soft constraints, that is constraints whose violation is possible. A *violation cost* is generally associated with every soft constraint. Then, a global objective related to the whole set of violation costs is usually defined. For instance, the goal can be to minimize the total sum of costs. In some applications it is necessary to express more complex rules on violations, which involve several costs independent from the objective function. Such rules can be defined through meta-constraints [Petit et al., 2000]. In order to model easily the part of the problem containing the soft constraints a global constraint involving the soft ones can be defined.

Moreover, in practice, among some other possibilities, two important types of violation costs can be identified:

- The violation cost depends only on the fact that the constraint is violated or not. In other words, either the constraint is satisfied and the violation cost is equal to 0, or the constraint is violated and the cost is equal to a given value. That is all the possible violations of a constraint have the same cost.
- The violation cost depends on the way the constraint is violated. The violation is quantified, thus we will call it quantified violation cost. For instance,

consider a cost associated with the violation of a temporal constraint imposing that a person should stop working before a given date: the violation cost can be proportional to the additional amount of working time she performs.

In this section we show two different advantages of the global constraints for solving over-constraint problems. First, we consider the Maximal Constraint Satisfaction Problem (Max-CSP), where the goal is to minimize the number of constraint violations, and we show that Max-CSP can be simply and efficiently modeled by a new global constraint. Then, we show how a quantified violation cost can be efficiently taken into account for a constraint and how new global constraints can be designed. These new constraints are called global soft constraints.

For more information about over-constrained problems and global constraint the reader can consult [Petit, 2002].

5.1 Satisfiability Sum Constraint

Let $\mathcal{N} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network containing some soft constraints. Max-CSP can be represented by a single constraint, called the Satisfiability Sum Constraint (SSC):

Definition 21 *Let $\mathcal{C} = \{C_i, i \in \{1, \ldots, m\}\}$ be a set of constraints, and $S[\mathcal{C}] = \{s_i, i \in \{1, \ldots, m\}\}$ be a set of variables and unsat be a variable, such that a one-to-one mapping is defined between $\mathcal{C}$ and $S[\mathcal{C}]$. A* **Satisfiability Sum Constraint** *is the constraint $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ defined by:*

$$[unsat = \sum_{s_i=1}^{m} s_i] \wedge \bigwedge_{i=1}^{m} [(C_i \wedge (s_i = 0)) \vee (\neg C_i \wedge (s_i = 1))]$$

The variables $S[\mathcal{C}]$ are used in order to express which constraints of $\mathcal{C}$ must be violated or satisfied: value 0 assigned to $s \in S[\mathcal{C}]$ expresses that its corresponding constraint C is satisfied, whereas 1 expresses that C is violated. Variable *unsat* represents the objective, that is the number of violations in $\mathcal{C}$, equal to the number of variables of $S[\mathcal{C}]$ whose value is 1.

Throughout this formulation, a solution of a Max-CSP is an assignment that satisfies the SSC with the minimal possible value of *unsat*. A lower bound of the objective of a Max-CSP corresponds to a necessary consistency condition of the SSC. The different domain reduction algorithms established for Max-CSP correspond to specific filtering algorithms associated with the SSC.

This point of view has some advantages in regard to the previous studies:
1. Any search algorithm can be used. This constraint can be associated with other ones, in order to separate soft constraints from hard ones.
2. No hypothesis is made on the arity of constraints $\mathcal{C}$.
3. If a value is assigned to $s_i \in S[\mathcal{C}]$, then a filtering algorithm associated with $C_i \in \mathcal{C}$ (resp. $\neg C_i$) can be used in a way similar to classical CSPs.

Moreover, the best algorithms to solve over-constrained problems like PFC-MRDAC [Larrosa et al., 1998] and the ones based on conflict-sets detection

[Régin et al., 2000,Régin et al., 2001] can be implemented as specific filtering algorithms associated with this constraint. A filtering algorithm based on a PFC-MRDAC version dealing only with the boundaries of the domain of the variable has also been described [Petit et al., 2002].

Furthermore, an extension of the model can be performed [Petit et al., 2000], in order to deal with Valued CSPs. Basically it consists of defining larger domains for variables in $S[\mathcal{C}]$.

5.2 Global Soft Constraints

In this section we consider that the constraints are associated with quantified violation costs. This section is based on [Petit et al., 2001].

Most of the algorithms dedicated to over-constrained problems are generic. However, the use of constraint-specific filtering algorithms is generally required to solve real-world applications, as their efficiency can be much higher.

Regarding over-constrained problems, existing constraint-specific filtering algorithms can be used only in the particular case where the constraint must be satisfied. Indeed, they remove values that are not consistent with the constraint: the deletion condition is linked to the fact that it is mandatory to satisfy the constraint. This condition is not applicable when the violation is allowed.

However, domains can be reduced from the objective and from the costs associated with violations of constraints. The main idea of this section is to perform this kind of filtering specifically, that is, to take advantage of the structure of a constraint and from the structure of its violation to efficiently reduce the domains of the variables it constrains.

The deletion condition will be linked to the necessity of having an acceptable cost, instead of being related to the satisfaction requirement.

For instance, let C be the constraint $x \leq y$. In order to quantify its violation, a *cost* is associated with C. It is defined as follows:

– if C is satisfied then $cost = 0$.

– if C is violated then $cost > 0$ and its value is proportional to the gap between x and y, that is, $cost = x - y$.

Assume that $D(x) = [90001, 100000]$ and $D(y) = [0, 200000]$, and that the cost is constrained to be less than 5. Then, either C is satisfied: $x - y \leq 0$, or C is violated: $x - y = cost$ and $cost \leq 5$, which implies $x - y \leq 5$. Hence, we deduce that $x - y \leq 5$, and, by propagation, $D(y) = [89996, 200000]$.

Such a deduction is made directly by propagating *bounds* of the variables x, y and *cost*. Inequality constraints admit such propagation on bounds without consideration of the domain values that lie between. Such propagation, which depends on the structure of the inequality constraint, is fundamentally more efficient than the consideration for deletion of each domain value in turn. If we ignore the structure of the constraint in the example, the only way to filter a value is to study the cost of each tuple in which this value is involved. Performing the reduction of $D(y)$ in the example above is costly: at least $|D(x)| * 89996 = 899960000$ checks are necessary. This demonstrates the gain that can be made

by directly integrating constraints on costs into the problem and employing constraint-specific filtering algorithms.

Following this idea, our goal is to allow the *same* modeling flexibility with respect to violation costs as with any other constrained variable. The most natural way to establish this is to include these violation costs as variables in a new constraint network.

For sake of clarity, we consider that the values of the cost associated with a constraint C are positive integers. 0 expresses the fact that C is satisfied, and strict positive values are proportional to the importance of a violation. This assumption is not a strong restriction; it just implies that values of cost belong to a totally ordered set.

A new optimization problem derived from the initial problem can be solved. It involves the same set of hard constraints $\mathcal{C}_h$, but a set of disjunctive constraints replaces $\mathcal{C}_s$. This set of disjunctive constraints is denoted by $\mathcal{C}_{disj}$ and a one-to-one correspondence is defined between $\mathcal{C}_s$ and $\mathcal{C}_{disj}$. Each disjunction involves a new variable $cost \in X_{costs}$, which is used to express the cost of $C \in \mathcal{C}_s$. A one-to-one correspondence is also defined between $\mathcal{C}_s$ and X_{costs}. Given $C \in \mathcal{C}_s$, the disjunction is the following:

$$[C \wedge [cost = 0]] \vee [\bar{C} \wedge [cost > 0]]$$

$\bar{C}$ is the constraint including the variable *cost* that expresses the violation. A specific filtering algorithm can be associated with it. Regarding the preliminary example, the constraints C and $\bar{C}$ are respectively $x \leq y$ and $cost = x - y$:

$$[[x \leq y] \wedge [cost = 0]] \vee [[cost = x - y] \wedge [cost > 0]]$$

The new defined problem is not over-constrained: it consists of satisfying the constraints $\mathcal{C}_h \cup \mathcal{C}_{disj}$, while optimizing an objective defined over all the variables X_{costs} (we deal with a classical optimization problem); constraints on a variable *cost* can be propagated.

Such a model can be used for encoding directly over-constrained problems with existing solvers [Régin et al., 2000]. Moreover, additional constraints on cost variables can be defined in order to select solutions that are acceptable for the user [Petit et al., 2000].

General Definitions of Cost When the violation of a constraint can be naturally defined, we use it (for instance, the constraint of the preliminary example $C : x \leq y$). However, this is not necessarily the case. When there is no natural definition associated with the violation of a constraint, different definitions of the cost can be considered, depending on the problem.

For instance, let C be an alldiff constraint defined on variables $var(C) = \{x_1, x_2, x_3, x_4\}$, such that $\forall i \in [1, 4], D(x_i) = \{a, b, c, d\}$. If we ignore the symmetric cases by considering that no value has more importance than another, we have the following possible assignments: (a, b, c, d), (a, a, c, d), (a, a, c, c), (a, a, a, c), (a, a, a, a).

Intuitively, it is straightforward that the violation of case (a, a, a, a) is more serious than the one of case (a, a, c, d). This fact has to be expressed through the cost.

Two general definitions of the cost associated with the violation of a non-binary constraint exist:

Definition 22 : Variable Based Violation Cost *Let C be a constraint. The cost of its violation can be defined as the number of assigned values that should change in order to make C satisfied.*

The advantage of this definition is that it can be applied to any (non-binary) constraint. However, depending on the application, it can be inconvenient. In the Alldiff example above we will have $cost((a, b, c, d)) = 0$, $cost((a, a, c, d)) = 1$, $cost((a, a, c, c)) = 2$, $cost((a, a, a, c)) = 2$, and $cost((a, a, a, a)) = 3$. A possible problem is that assignments (a, a, c, c) and (a, a, a, c) have the same cost according to definition 22. For an Alldiff involving more than four variables, a lot of different assignments have the same cost.

Therefore, there is another definition of the cost, which is well suited to constraints that are representable through a primal graph [Dechter, 1992]:

Definition 23 *The* **primal graph** $Primal(C) = (var(C), E_p)$ *of a constraint C is a graph such that each edge represents a binary constraint, and the set of solutions of the CSP defined by*
$\mathcal{N} = (var(C), D(var(C)), E_p)$ *is the set of allowed tuples of C.*

For an Alldiff C, $Primal(C)$ is a complete graph where each edge represents a binary inequality.

Definition 24 : Primal Graph Based Violation Cost *Let C be a constraint representable by a primal graph. The cost of its violation can be defined as the number of binary constraints violated in the CSP defined by $Primal(C)$.*

In the Alldiff case, the user may aim at controlling the number of binary inequalities implicitly violated. The advantage of this definition is that the granularity of the quantification is more accurate. In the example, the costs are $cost((a, b, c, d)) = 0$, $cost((a, a, c, d)) = 1$, $cost((a, a, c, c)) = 2$, $cost((a, a, a, c)) = 3$, and $cost((a, a, a, a)) = 6$.

Soft Alldiff Constraint The constraint obtained by combining a variable based violation cost and alldiff constraint, is, in fact, a k-diff constraint where k is the minimum value of the cost variable. Therefore, if the modification of k is dynamically maintained, which is relatively easy because it can only be increased, then we obtain a filtering algorithm establishing algorithm for this global soft constraint.

The constraint formed by the combination of a primal graph based cost and an alldiff constraint is much more complex. A specific filtering algorithm for this constraint has been designed by [Petit et al., 2002]. Its complexity is in $O(|var(C)|^2\sqrt{|var(C)|}Kd)$, where $K = \sum |D(x)|, x \in var(C)$ and $d = max(|D(x)|), x \in var(C)$.

6 Quality of Filtering Algorithms

In this section, we try to characterize some properties of a good filtering algorithm.

Section 3.2 presents a general filtering algorithm establishing arc consistency. From a problem P for which a method giving a solution is known, this algorithm is able to establish and maintain arc consistency of $C(P)$ in $nd \times O(P)$. Therefore, there is no need to develop a specific algorithm with the same complexity. Every dedicated algorithm must improve that complexity otherwise it is not worthwhile.

From this remark we propose the following classification:

Definition 25 *Let C be a constraint for which the consistency can be computed in $O(C)$. A filtering algorithm establishing arc consistency associated with C is:*

- **poor** *if its complexity is $O(nd) \times O(C)$;*
- **medium** *if its complexity is $O(n) \times O(C)$;*
- **good** *if its complexity is $O(C)$;*

Some good filtering algorithms are known for some constraints. We can cite the alldiff constraint or the global cardinality constraint.

Some medium filtering algorithms have also been developed for some constraints like global cardinality constraint with costs, and symmetric alldiff. Thus, these algorithms can be improved.

Good filtering algorithms are not perfect and the definition of the quality we propose is based on worst-case complexity. This definition can be refined to be more accurate with the use of filtering algorithms in CP, because the incrementality is quite important:

Definition 26 *A filtering algorithm establishing arc consistency is* **perfect** *if it always has the same cost as the consistency checking.*

This definition means that the complexity must be the same in all the cases and not only for the worst one. For instance, such an algorithm is not known for the alldiff constraint, because the consistency of this constraint can sometimes be checked in $O(1)$ and the arc consistency needs at least $O(nd)$.

The only one constraint for which a perfect filtering algorithm is known is the constraint $(x < y)$.

Two other points play an important part in the quality of a filtering algorithm: the incrementality and the amortized complexity. These points are linked together.

The incremental behavior of a filtering algorithm is quite important in CP, because the algorithms are systematically called when a modification of a variable involved in the constraint occurs. However, the algorithm should not be focus only on this aspect. Sometimes, the computation from scratch can be much more quicker. This point has been emphasized for general filtering algorithms based on the list of supported values of a value [Bessière and Régin, 2001]. An adaptive

algorithm has been proposed which outperforms both the non-incremental version and the purely incremental version. This is one in which the consequences of the deletion of a value are systematically studied from the information associated with the deleted value and never from scratch. There are two possible ways to improve the incremental behavior of the algorithm:

• The previous computations are taken into account when a new computation is made in order to avoid doing the same treatment twice. For instance, this is the idea behind the last support in some general filtering algorithm algorithms.

• The filtering algorithm is not systematically called after each modification. Some properties that cannot lead to any deletions are identified, and the filtering algorithm is called only when these properties are not satisfied. For instance, this is the case for the model we present to solve the n-queens problem.

When a filtering algorithm is incremental we can expect to compute its amortized complexity. This is the complexity in regard to the number of deletions, or for one branch of the tree-search. This is why the complexity can be analyze after a certain number of modifications. The amortized complexity is often more accurate for filtering algorithm. Moreover, it can lead to new interesting algorithms that are not too systematic. For instance, there is a filtering algorithm for the symmetric alldiff constraint that is based on this idea. The filtering algorithm establishing arc consistency calls another algorithm A n times, therefore its complexity is $n \times O(A)$. Another algorithm has been proposed in [Régin, 1999b], which can be described as follows: pick a variable then run A, and let k be the number of deletions made by A. Then you can run A for k other variables. By proceeding like that the complexity is $O(A)$ per deletions. Of course, the algorithm does not necessarily establish arc consistency but this is possibly a good compromise.

7 Discussion

7.1 Incomplete Algorithms and Fixed-Point Property

Some global constraints correspond to NP-Complete problems. Hence, it is not possible to check polynomially the consistency of the constraint to establish arc consistency. Nevertheless, some filtering algorithms can be still proposed. This is the case for a lot of constraints: the cumulative constraint, the diff-n constraint, the sequence constraint, the stretch constraint, the global minimum distance constraint, the number of distinct values constraints, and so on. When the problem is NP-Complete the filtering algorithm considers a relaxation, which is no longer difficult. Currently, the filtering algorithms associated with such constraints are independent of the definition of the problem. In other words, a propagation mechanism using them will reach a fixed-point. That is, the set of values that are deleted is independent from the ordering according to the constraints defined and from the ordering according to the filtering algorithms called. In order to guarantee such a property, the filtering algorithm is based

either on a set of properties that can be exactly computed (not approximated), or on a relaxation of the domains of the variables (that is, the domains are considered as ranges instead of as a set of enumerated values). The loss of the fixed-point property leads to several consequences: the set of values deleted by propagation will depend on the ordering along with the stated constraints and on the ordering along with the variables involved in a constraint. This means that the debugging will be a much more difficult task because fewer constraints can lead to more deleted values, and more constraints can lead to fewer deleted values.

In the future, we will certainly need filtering algorithms with which the fixed-point property of the propagation mechanism will be lost, because more domain-reduction could be done with such algorithms. For instance, suppose that a filtering algorithm is based on the removal of nodes in a graph that belong to a clique of size greater than k. Removing all the values that do not satisfy this property is an NP-Complete problem; therefore the filtering algorithms will not be able to do it. However, some of these values can be removed, for instance by searching for one clique for every node. The drawback of this approach is that it will be difficult to guarantee that for a given node the graph will be traversed according to the same ordering of nodes. This problem is closed to the canonical representation of a graph; and currently this problem is unclassified: we do not know whether it is NP-Complete or not.

7.2 Closure

In general, a filtering algorithm removes some values that do not satisfy a property. The question is "Should a filtering algorithm be closed with regard to this property?"

Consider the values deleted by the filtering algorithm. Then, the consequences of these new deletions can be:

- taken into account by the same pass of the filtering algorithm;
- or ignored by the same pass of the filtering algorithm.

In the first case, there is no need to call the filtering algorithm again and in the second case the filtering algorithm should be called again. When the filtering algorithm is good, usually the first solution is the good one, but when the filtering algorithm consists of calling another algorithm for every variable or every value, it is possible that any deletion calls the previous computations into question. Then, the risk is to have to check again and again the consistency of some values. It is also possible that the filtering algorithm internally manages a mechanism that is closed to the propagation mechanism of the solver, which is redundant.
In this case, it can be better to stop the filtering algorithm when some modifications occur in order to use the other filtering algorithms to further reduce the domains of the variable and to limit the number of useless calls.

7.3 Power of a Filtering Algorithm

Arc consistency is a strong property, but establishing it costs sometimes in practice. Thus, some researchers have proposed to use weaker properties in practice. That is, to let the user to choose which type of filtering algorithm should be associated with a constraint. In some commercial CP Solvers, like ILOG-Solver, the user is provided with such a possibility. Therefore it is certainly interesting to develop some filtering algorithms establishing properties weaker than arc consistency. However, arc consistency has some advantages that must not be ignored:

- The establishing of arc consistency is much more robust. Sometimes, it is time consuming, but it is often the only way to design a good model. During the modeling phase, it is very useful to use strong filtering algorithms, even if, sometimes, some weaker filtering algorithms can be used to improve the time performance of the final model. It is rare to be able to solve some problems in a reasonable amount of time with filtering algorithms establishing properties weaker than arc consistency and not be able to solve these problems with a filtering algorithm establishing arc consistency.
- There is a room for the improvement of filtering algorithms. Most of the CP solvers were designed before the introduction of global constraints in CP. We could imagine that a solver especially designed to efficiently handle global constraints could lead to better performance. On the other hand, the behavior of filtering algorithms could also be improved in practice, notably by identifying more quickly the cases where no deletion is possible.
- For binary CSPs,for a long time it was considered that the Forward Checking algorithm (the filtering algorithms are triggered only when some variables are instantiated) was the most efficient one, but several studies showed that the systematic call of filtering algorithms after every modification is worthwhile (for instance see [Bessière and Régin, 1996]). All industrial solver vendors aim to solve real world applications and claim that the use of strong filtering algorithms is often essential.

Thus, we think that the studies about filtering algorithms establishing properties weaker than arc consistency should take into account the previous points and mainly the second point. On the other hand, we think that it is really worthwhile to work on techniques stronger than arc consistency, like singleton arc consistency which consists of studying the consequences of the assignments of every value to every variable.

8 Conclusion

Filtering algorithms are one of the main strengths of CP. In this chapter, we have presented several useful global constraints with references to the filtering algorithms associated with them. We have also detailed the filtering algorithms establishing arc consistency for the alldiff constraint and the global cardinality

constraint. We have also tried to give a characterization of filtering algorithms. We have also showed how the global constraint can be useful for over-constrained problems and notably, we have presented the global soft constraints. At last, the the filtering algorithms we presented are mainly based on arc consistency, we think that some interesting work based on bound-consistency could be carried out.

References

[Ahuja et al., 1993] Ahuja, R., Magnanti, T., and Orlin, J. (1993). *Network Flows*. Prentice Hall.

[Baptiste et al., 1998] Baptiste, P., Le Pape, C., and Peridy, L. (1998). Global constraints for partial csps: A case-study of resource and due date constraints. In *Proceedings CP'98*, pages 87–101, Pisa, Italy.

[Beldiceanu, 2001] Beldiceanu, N. (2001). Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proceedings CP'01*, pages 211–224, Pathos, Cyprus.

[Beldiceanu and Carlsson, 2002] Beldiceanu, N. and Carlsson, M. (2002). A new multi-resource cumulatives constraint with negative heights. In *Proceedings CP'02*, pages 63–79, Ithaca, NY, USA.

[Beldiceanu and Contejean, 1994] Beldiceanu, N. and Contejean, E. (1994). Introducing global constraints in chip. *Journal of Mathematical and Computer Modelling*, 20(12):97–123.

[Beldiceanu et al., 2001] Beldiceanu, N., Guo, Q., and Thiel, S. (2001). Non-overlapping constraints between convex polytopes. In *Proceedings CP'01*, pages 392–407, Pathos, Cyprus.

[Berge, 1970] Berge, C. (1970). *Graphe et Hypergraphes*. Dunod, Paris.

[Bessière and Régin, 1996] Bessière, C. and Régin, J.-C. (1996). Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *CP96, Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, Cambridge, MA, USA.

[Bessière and Régin, 1997] Bessière, C. and Régin, J.-C. (1997). Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya.

[Bessière and Régin, 1999] Bessière, C. and Régin, J.-C. (1999). Enforcing arc consistency on global constraints by solving subproblems on the fly. In *Proceedings of CP'99*, pages 103–117, Alexandria, VA, USA.

[Bessière and Régin, 2001] Bessière, C. and Régin, J.-C. (2001). Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle, WA, USA.

[Bleuzen-Guernalec and Colmerauer, 1997] Bleuzen-Guernalec, N. and Colmerauer, A. (1997). Narrowing a 2n-block of sortings in o(nlog(n)). In *Proceedings of CP'97*, pages 2–16, Linz, Austria.

[Carlier and Pinson, 1994] Carlier, J. and Pinson, E. (1994). Adjustments of heads and tails for the jobshop problem. *European Journal of Operational Research*, 78:146–161.

[Caseau et al., 1993] Caseau, Y., Guillo, P.-Y., and Levenez, E. (1993). A deductive and object-oriented approach to a complex scheduling problem. In *Proceedings of DOOD'93*.

[Caseau and Laburthe, 1997] Caseau, Y. and Laburthe, F. (1997). Solving various weighted matching problems with constraints. In *Proceedings CP97*, pages 17–31, Austria.

[Damaschke et al., 1990] Damaschke, P., Müller, H., and Kratsch, D. (1990). Domination in convex and chrodal bipartite graphs. *Information Processing Letters*, 36:231–236.

[Dechter, 1992] Dechter, R. (1992). From local to global consistency. *Artificial Intelligence*, 55:87–107.

[Focacci et al., 1999a] Focacci, F., Lodi, A., and Milano, M. (1999a). Cost-based domain filtering. In *Proceedings CP'99*, pages 189–203, Alexandria, VA, USA.

[Focacci et al., 1999b] Focacci, F., Lodi, A., and Milano, M. (1999b). Integration of cp and or methods for matching problems. In *Proceedings CP-AI-OR 99*, Ferrara, Italy.

[Henz et al., 2003] Henz, M., Müller, T., and Thiel, S. (2003). Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, page To appear.

[Hopcroft and Karp, 1973] Hopcroft, J. and Karp, R. (1973). $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2:225–231.

[ILOG, 1999] ILOG (1999). *ILOG Solver 4.4 User's manual*. ILOG S.A.

[Larrosa et al., 1998] Larrosa, J., Meseguer, P., Schiex, T., and Verfaillie, G. (1998). Reversible dac and other improvements for solving max-csp. *Proceedings AAAI*, pages 347–352.

[Laurière, 1978] Laurière, J.-L. (1978). A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127.

[Lawler, 1976] Lawler, E. (1976). *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston.

[Leconte, 1996] Leconte, M. (1996). A bounds-based reduction scheme for constraints of difference. In *Constraint-96, Second International Workshop on Constraint-based Reasoning*, Key West, FL, USA.

[McAloon et al., 1997] McAloon, K., Tretkoff, C., and Wetzel, G. (1997). Sports league scheduling. In *Proceedings of ILOG user's conference*, Paris.

[Melhorn and Thiel, 2000] Melhorn, K. and Thiel, S. (2000). Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Proceedings of CP'00*, pages 306–319, Singapore.

[Micali and Vazirani, 1980] Micali, S. and Vazirani, V. (1980). An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings 21st FOCS*, pages 17–27.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132.

[Pesant, 2001] Pesant, G. (2001). A filtering algorithm for the stretch constraint. In *Proceedings CP'01*, pages 183–195, Pathos, Cyprus.

[Petit, 2002] Petit, T. (2002). *Modelization and Algorithms for solving over-constrained Problems*. PhD thesis, Université de Montpellier II.

[Petit et al., 2000] Petit, T., Régin, J.-C., and Bessière, C. (2000). Meta constraints on violations for over-constrained problems. In *Proceedings ICTAI-2000*, pages 358–365.

[Petit et al., 2001] Petit, T., Régin, J.-C., and Bessière, C. (2001). Specific filtering algorithms for over-constrained problems. In *Proceedings CP'01*, pages 451–465, Pathos, Cyprus.

[Petit et al., 2002] Petit, T., Régin, J.-C., and Bessière, C. (2002). Range-based algorithms for max-csp. In *Proceedings CP'02*, pages 280–294, Ithaca, NY, USA.

[Puget, 1998] Puget, J.-F. (1998). A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of AAAI-98*, pages 359–366, Menlo Park, USA.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI-94*, pages 362–367, Seattle, Washington.

[Régin, 1995] Régin, J.-C. (1995). *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* PhD thesis, Université de Montpellier II.

[Régin, 1996] Régin, J.-C. (1996). Generalized arc consistency for global cardinality constraint. In *Proceedings AAAI-96*, pages 209–215, Portland, Oregon.

[Régin, 1997] Régin, J.-C. (1997). The global minimum distance constraint. Technical report, ILOG.

[Régin, 1999a] Régin, J.-C. (1999a). Arc consistency for global cardinality with costs. In *Proceedings of CP'99*, pages 390–404, Alexandria, VA, USA.

[Régin, 1999b] Régin, J.-C. (1999b). The symmetric alldiff constraint. In *Proceedings of IJCAI'99*, pages 425–429, Stockholm, Sweden.

[Régin, 2002] Régin, J.-C. (2002). Cost based arc consistency for global cardinality constraints. *Constraints, an International Journal*, 7(3-4):387–405.

[Régin et al., 2000] Régin, J.-C., Petit, T., Bessière, C., and Puget, J.-F. (2000). An original constraint based approach for solving over constrained problems. In *Proceedings of CP'00*, pages 543–548, Singapore.

[Régin et al., 2001] Régin, J.-C., Petit, T., Bessière, C., and Puget, J.-F. (2001). New lower bounds of constraint violations for over-constrained problems. In *Proceedings CP'01*, pages 332–345, Pathos, Cyprus.

[Régin and Puget, 1997] Régin, J.-C. and Puget, J.-F. (1997). A filtering algorithm for global sequencing constraints. In *CP97, proceedings Third International Conference on Principles and Practice of Constraint Programming*, pages 32–46.

[Régin and Rueher, 2000] Régin, J.-C. and Rueher, M. (2000). A global constraint combining a sum constraint and difference constraints. In *Proceedings of CP'00*, pages 384–395, Singapore.

[Rossi et al., 1990] Rossi, F., Petrie, C., and Dhar, V. (1990). On the equivalence of constraint satisfaction problems. In *Proceedings ECAI'90*, pages 550–556, Stockholm, Sweden.

[Simonis, 1996] Simonis, H. (1996). Problem classification scheme for finite domain constraint solving. In *CP96, Workshop on Constraint Programming Applications: An Inventory and Taxonomy*, pages 1–26, Cambridge, MA, USA.

[Stergiou and Walsh, 1999] Stergiou, K. and Walsh, T. (1999). The difference all-difference makes. In *Proceedings IJCAI'99*, pages 414–419, Stockholm, Sweden.

[Tarjan, 1983] Tarjan, R. (1983). *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Mathematics.

[Van Hentenryck and Deville, 1991] Van Hentenryck, P. and Deville, Y. (1991). The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of ICLP-91*, pages 745–759, Paris, France.

[Van Hentenryck et al., 1992] Van Hentenryck, P., Deville, Y., and Teng, C. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321.

[Van Hentenryck et al., 1999] Van Hentenryck, P., Michel, L., L.Perron, and Régin, J.-C. (1999). Constraint programming in opl. In *PPDP 99, International Conference on the Principles and Practice of Declarative Programming*, pages 98–116, Paris, France.

[Zhou, 1996] Zhou, J. (1996). A constraint program for solving the job-shop problem. In *Proceedings of CP'96*, pages 510–524, Cambridge.

[Zhou, 1997] Zhou, J. (1997). *Computing Smallest Cartesian Products of Intervals: Application to the Jobshop Scheduling Problem.* PhD thesis, Université de la Méditerranée, Marseille.