The Range and Roots Constraints: Algorithms and Implementation

Christian Bessiere[1], Emmanuel Hebrard[2], Brahim Hnich[3], Zeynep Kiziltan[4], and Toby Walsh[2]

[1] LIRMM, CNRS/University of Montpellier, France, bessiere@lirmm.fr
[2] NICTA and UNSW, Sydney, Australia, {ehebrard, tw}@cse.unsw.edu.au
[3] 4C, University College Cork, Ireland, brahim@4c.ucc.ie
[4] University of Bologna, Italy, zkiziltan@deis.unibo.it

Abstract. We recently proposed a simple declarative language for specifying a wide range of counting and occurrence constraints. The language uses just two global primitives: the RANGE constraint, which computes the range of values used by a set of variables, and the ROOTS constraint, which computes the variables mapping onto particular values. In this paper, we demonstrate that this specification language is executable by proposing efficient algorithms for propagating these two global constraints. We show that decomposing global counting and occurrence constraints using RANGE and ROOTS is effective and efficient in practice.

1 Introduction

Constraints that put restrictions on the occurrence of particular values (*occurrence* constraints) or constraints that put restrictions on the number of values or variables meeting some conditions (*counting* constraints) are very useful in many real world problems, especially those involving resources. For instance, we may want to limit the number of distinct values assigned to a set of variables. Many of the global constraints proposed in the past are counting and occurrence constraints (see, for example, [1–5]). In [6], we show that many occurrence and counting constraints can be expressed by means of two new global constraints, RANGE and ROOTS, together with some classical elementary constraints. This language also provides us with a method to propagate counting and occurrence constraints. We just need to provide efficient propagation algorithms for the RANGE and ROOTS constraints. This paper does exactly this. We first give an efficient algorithm for propagating the RANGE constraint based on a flow algorithm. We then prove that it is intractable to propagate the ROOTS constraint completely. We therefore propose a linear time algorithm for propagating it partially that is complete under conditions met frequently in practice. In addition, a simple relaxation of the algorithm achieves bound consistency in general.

2 Formal background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for subsets of variables. We use capitals for variables (e.g. X, Y and S), and lower case for values (e.g. v and w). We write $D(X)$ for the domain of a variable X. For totally ordered domains, we write $min(D(X))$ and $max(D(X))$ for the minimum and maximum values. A solution is an assignment of values to the variables satisfying the constraints. A variable is *ground* when it is assigned a value. We consider both *integer* and *set* variables. A set variable S is represented by its lower bound $lb(S)$ which contains the definite elements and an upper bound $ub(S)$ which also contains the potential elements.

Constraint solvers typically explore partial assignments enforcing a local consistency property using either specialized or general purpose propagation algorithms. Given a constraint C, a *bound support* on C is a tuple that assigns to each integer variable a value between its minimum and maximum, and to each set variable a set between its lower and upper bounds which satisfies C. A bound support in which each integer variable is assigned a value in its domain is called an *hybrid support*. If C involves only integer variables, an hybrid support is a *support*. A constraint C is *bound consistent* (BC) iff for each integer variable X_i, its minimum and maximum values belong to a bound support, and for each set variable S_j, the values in $ub(S_j)$ belong to S_j in at least one bound support and the values in $lb(S_j)$ belong to S_j in all bound supports. A constraint C is *hybrid consistent* (HC) iff for each integer variable X_i, every value in $D(X_i)$ belongs to an hybrid support, and for each set variable S_j, the values in $ub(S_j)$ belong to S_j in at least one hybrid support, and the values in $lb(S_j)$ belong to S_j in all hybrid supports. A constraint C involving only integer variables is *generalized arc consistent* (GAC) iff for each variable X_i, every value in $D(X_i)$ belongs to a support. If all variables in C are integer variables, hybrid consistency reduces to generalized arc-consistency, and if all variables in C are set variables, hybrid consistency reduces to bound consistency. To illustrate these different concepts, consider the constraint $C(X_1, X_2, S)$ that holds iff the set variable S is assigned exactly the values used by the integer variables X_1 and X_2. Let $D(X_1) = \{1, 3\}$, $D(X_2) = \{2, 4\}$, $lb(S) = \{2\}$ and $ub(S) = \{1, 2, 3, 4\}$. BC does not remove any value since all domains are already bound consistent. On the other hand, HC removes 4 from $D(X_2)$ and from $ub(S)$ as there does not exist any tuple satisfying C in which X_2 does not take value 2. Note that since BC deals with bounds, value 2 was considered as possible for X_1.

A total function $\mathcal{F}$ from a set $\mathcal{S}$ into a set $\mathcal{T}$ is denoted by $\mathcal{F} : \mathcal{S} \longrightarrow \mathcal{T}$ where $\mathcal{S}$ is the domain of $\mathcal{F}$ and $\mathcal{T}$ is the range of $\mathcal{F}$. The set of all elements in the domain of $\mathcal{F}$ that have the same image $j \in \mathcal{T}$ is $\mathcal{F}^{-1}(j) = \{i : i \in \mathcal{S}, \mathcal{F}(i) = j\}$. Throughout, we will view a set of variables, X_1 to X_n as a function $\mathcal{X} : \{1, .., n\} \longrightarrow \bigcup_{i=1}^{i=n} D(X_i)$. That is, $\mathcal{X}(i)$ is the value of X_i.

3 An Executable Language

One of the simplest ways to propagate a new constraint is to decompose it into existing primitive constraints. We can then use the propagation algorithms associated with these primitives. Of course, such decomposition may reduce the number of domain values pruned. In [6], we show that many global counting and occurrence constraints can be decomposed into two new global constraints, Range and Roots, together with simple non-global constraints over integer variables (like $X \leq m$) and simple non-global constraints over set variables (like $S_1 \subseteq S_2$ or $|S| = k$). Adding Range and Roots and their propagation algorithms to a constraint toolkit thus provides a simple executable language for specifying a wide range of counting and occurrence constraints.

Given a function $\mathcal{X}$ representing a set of variables X_1 to X_n, the Range constraint holds iff a set variable T is the range of this function, restricted to the indices belonging to a second set variable, S.

$$\textsc{Range}([X_1, .., X_n], S, T) \text{ iff } T = \bigcup_{i \in S} \mathcal{X}(i)$$

The Roots constraint holds iff S is the set of indices which map to an element belonging to the set variable T.

$$\textsc{Roots}([X_1, .., X_n], S, T) \text{ iff } S = \bigcup_{j \in T} \mathcal{X}^{-1}(j)$$

Range and Roots are not exact inverses. A Range constraint can hold, but the corresponding Roots constraint may not, and vice versa. For instance, $\textsc{Range}([1,1], \{1\}, \{1\})$ holds but not $\textsc{Roots}([1,1], \{1\}, \{1\})$ since $\mathcal{X}^{-1}(1) = \{1,2\}$, and $\textsc{Roots}([1,1,1], \{1,2,3\}, \{1,2\})$ holds but not $\textsc{Range}([1,1,1], \{1,2,3\}, \{1,2\})$ as no X_i is assigned to 2.

In an associated report, we present a catalog containing over 70 global constraints from [7] specified with this simple language. We present here just two examples. The NValue constraint is useful in a wide range of problems involving resources since it counts the number of distinct values used by a sequence of variables [8]. $\textsc{NValue}([X_1, .., X_n], N)$ holds iff $N = |\{X_i \mid 1 \leq i \leq n\}|$. This can be decomposed into a Range constraint:

$$\textsc{NValue}([X_1, .., X_n], N) \text{ iff } \textsc{Range}([X_1, .., X_n], \{1, .., n\}, T) \ \wedge \ |T| = N$$

Enforcing HC on the decomposition is weaker than GAC on the original NValue constraint. However, it is NP-hard to enforce GAC on a NValue constraint [9].

The Among constraint was introduced in CHIP to model resource allocation problems like car sequencing [2]. It counts the number of variables using values from a given set. $\textsc{Among}([X_1, .., X_n], [d_1, .., d_m], N)$ holds iff $N = |\{i \mid X_i \in \{d_1, .., d_m\}\}|$. It can be decomposed using a Roots constraint:

$$\textsc{Among}([X_1, .., X_n], [d_1, .., d_m], N) \text{ iff } \textsc{Roots}([X_1, .., X_n], S, \{d_1, .., d_m\}) \wedge |S| = N$$

GAC on Among is equivalent to HC on this decomposition [6].

4 Range Constraint

Enforcing hybrid consistency on the Range constraint is polynomial. In this section, we propose an efficient way to enforce HC on Range using a maximum network flow problem. To simplify the presentation, the use of the flow is limited to a constraint that performs only part of the work needed for enforcing HC on RANGE. This constraint, that we name $\textsc{Occurs}(Y_1, \ldots, Y_n, T)$, ensures that all the values in the set variable T are used by the integer variables Y_1 to Y_n:

$$\textsc{Occurs}([Y_1, \ldots, Y_n], T) \text{ iff } T \subseteq \bigcup_{i \in 1..n} \mathcal{Y}(i)$$

We first present an algorithm for achieving HC on OCCURS (Section 4.1), and then use this to propagate the RANGE constraint (Section 4.2).

4.1 Occurs Constraint

We achieve hybrid consistency on $\textsc{Occurs}([X_1, \ldots, X_n], T)$ using a network flow. We use an unit capacity network [10] in which capacities between two nodes can only be 0 or 1. This is represented by a directed graph where an arc from node x to node y means that a maximum flow of 1 is allowed between x and y while the absence of arc means that the maximum flow allowed is 0. The unit capacity network $G_C = (N, E)$ of the constraint $C = \textsc{Occurs}(X_1, \ldots, X_n, T)$ is built in the following way. $N = \{s\} \cup N_1 \cup N_2 \cup \{t\}$, where s is a source node, t is a sink node, $N_1 = \{y_v \mid v \in \bigcup D(X_i)\}$ and $N_2 = \{z_v \mid v \in \bigcup D(X_i)\} \cup \{x_i \mid i \in [1..n]\}$. The set of arcs E is as follows:

$$E = (\{s\} \times N_1) \cup \{(y_v, z_v), \forall v \notin lb(T)\} \cup \{(y_v, x_i) \mid v \in D(X_i)\} \cup (N_2 \times \{t\})$$

G_C is quadripartite, i.e., $E \subseteq (\{s\} \times N_1) \cup (N_1 \times N_2) \cup (N_2 \times \{t\})$. The intuition behind this graph is that when a flow uses an arc from a node y_v to a node x_i this means that $X_i = v$, and when a flow uses the arc (y_v, z_v) this means that v is not necessarily used by the X_i's.

In the particular case of unit capacity networks, a flow is any set $E' \subseteq E$: any arc in E' is assigned 1 and the arcs in $E \setminus E'$ are assigned 0. A *feasible* flow from s to t in G_C is a subset E_f of E such that $\forall n \in N \setminus \{s, t\}$, $|\{(n', n) \in E_f\}| = |\{(n, n'') \in E_f\}|$. The value of the flow E_f from s to t, denoted $val(E_f, s, t)$, is $val(E_f, s, t) = |\{(s, n) \in E_f\}|$. A *maximum* flow from s to t in G_C is a feasible flow E_M such that there does not exist a feasible flow E_f, with $val(E_f, s, t) > val(E_M, s, t)$. By construction a feasible flow cannot have a value greater than $|N_1|$. In addition, a feasible flow cannot contain two arcs entering a node x_i from N_2. Hence, we can define a function φ linking feasible flows and partial instantiations on the X_i's. Given any feasible flow E_f from s to t in G_C, $\varphi(E_f) = \{(X_i, v) \mid (y_v, x_i) \in E_f\}$. The way G_C is built induces the following:

Theorem 1. *Let $G_C = (N, E)$ be the capacity network of a constraint $C = \textsc{Occurs}(X_1, \ldots, X_n, T)$.*

1. *A value v in the domain $D(X_i)$ for some $i \in [1..n]$ is GAC iff there exists a flow E_f from s to t in G_C with $val(E_f, s, t) = |N_1|$ and $(y_v, x_i) \in E_M$*
2. *If the X_i's are GAC, T is BC iff $ub(T) \subseteq \bigcup_i D(X_i)$*

Proof. (1.$\Rightarrow$) Let I be a solution for C with $(X_i, v) \in I$. Build the following flow H: Put (y_v, x_i) in H; $\forall w \in I[T], w \neq v$, take a variable X_j such that $(X_j, w) \in I$ (we know there is at least one since I is solution), and put (y_w, x_j) in H; $\forall w' \notin I[T], w' \neq v$, add $(y_{w'}, z_{w'})$ to H. Add to H the edges from s to N_1 and from N_2 to t so that we obtain a feasible flow. By construction, all $y_w \in N_1$ belong to an edge of H. So, $val(H, s, t) = |N_1|$ and H is a maximum flow with $(y_v, x_i) \in H$.

(1.$\Leftarrow$) Let E_M be a flow from s to t in G_C with $(y_v, x_i) \in E_M$ and $val(E_M, s, t) = |N_1|$. By construction of G_C, we are guaranteed that all nodes in N_1 belong to an arc in $E_M \cap (N_1 \times N_2)$, and that for every value $w \in lb(T)$, $\{n \mid (y_w, n) \in E\} \subseteq \{x_i \mid i \in [1..n]\}$. Thus, for each $w \in lb(T), \exists X_j \mid (X_j, w) \in \varphi(E_M)$. Hence, any extension of $\varphi(E_M)$ where each unassigned X_j takes any value in $D(X_j)$ and $T = lb(T)$ is a solution of C with $X_i = v$.

(2.$\Rightarrow$) If T is BC, all values in $ub(T)$ appear in at least one solution tuple. Since C ensures that $T \subseteq \bigcup_i \{X_i\}$, $ub(T)$ cannot contain a value appearing in none of the $D(X_i)$.

(2.$\Leftarrow$) Let $ub(T) \subseteq \bigcup_i D(X_i)$. Since all X_i's are GAC, we know that each value v in $\bigcup_i D(X_i)$ is taken by some X_i in at least one solution tuple I. Build the tuple I' so that $I'[X_i] = I[X_i]$ for each $i \in [1..n]$ and $I'[T] = I[T] \cup \{v\}$. I' is still solution of C. So, $ub(T)$ is as tight as it can be wrt BC. In addition, since all X_i's are GAC, this means that in every solution tuple I, for each $v \in lb(T)$ there exists i such that $I[X_i] = v$. So, $lb(T)$ is BC. □

We also need the notion of *residual graphs*. Given a unit capacity network G_C and a maximal flow E_M from s to t in G_C, the residual graph obtained from G_C and E_M is the directed graph $R_{G_C}(E_M) = (N, E_R)$ where $E_R = \{(x, y) \in E \setminus E_R\} \cup \{(y, x) \mid (x, y) \in E \cap E_R\}$. Given a maximum flow E_M from s to t in G_C, given $(x, y) \in N_1 \times N_2 \cap E \setminus E_M$, there exists a maximum flow containing (x, y) iff (x, y) belongs to a cycle in $R_{G_C}(E_M)$ [11]. Furthermore, finding all the arcs (x, y) that do not belong to a cycle in a graph can be performed by building the strongly connected components (SCCs) of the graph.

We thus have all the tools for achieving HC on any Occurs constraint. We first build G_C. We then compute a maximum flow E_M from s to t in G_C; if $val(E_M, s, t) < |N_1|$, we fail. Otherwise we compute $R_{G_C}(E_M)$, build the SCCs in $R_{G_C}(E_M)$, and remove from $D(X_i)$ any value v such that (y_v, x_i) belongs to neither E_M nor to a SCC in $R_{G_C}(E_M)$. Finally, we set $ub(T)$ to $ub(T) \cap \bigcup_i D(X_i)$. Following Theorem 1 and properties of residual graphs, this algorithm enforces HC on Occurs$([X_1, .., X_n], T)$. Building G_C (step 1.) is in $O(nd)$. We need then to find a maximum flow E_M in G_C (step 2.). This can be done in two sub-steps. First, we use the arc (y_v, z_v) for each $v \notin lb(T)$ (in $O(|\bigcup_i D(X_i)|)$). Afterwards, we compute a maximum flow on the subgraph composed of all paths traversing nodes y_w with $w \in lb(T)$ (because there is no arc (y_w, z_w) in G_C for such w). The complexity of finding a maximum flow in a unit capacity network is in $O(\sqrt{k} \cdot e)$ if k is the number of nodes and e the number of edges. This gives a complexity in $O(\sqrt{|lb(T)|} \cdot n \cdot |lb(T)|)$ for this second sub-step. Building the residual graph (step 3.) and computing the SCCs (step 4.) is in $O(nd)$. Extracting the GAC domains

Algorithm 1: `Hybrid consistency on` RANGE

procedure *Propag-Range*($[X_1, \ldots, X_n], S, T$);
1 Introduce the set of integer variables $Y = \{Y_i \mid i \in ub(S)\}$, with $D(Y_i) = D(X_i) \cup \{dummy\}$;
2 Achieve hybrid consistency on the constraint $\textsc{Occurs}(Y, T)$;
3 Achieve hybrid consistency on the constraints $i \in S \leftrightarrow Y_i \in T$, for all $Y_i \in Y$;
4 Achieve GAC on the constraints $(Y_i = dummy) \vee (Y_i = X_i)$, for all $Y_i \in Y$;

for the X_i's is direct (step 5.). There remains to compute BC on T (step 6.), which takes $O(nd)$. Therefore, the total complexity is in $O(nd + n \cdot |lb(T)|^{3/2})$.

4.2 Hybrid Consistency on Range

The algorithm *Propag-Range*, enforcing HC on the RANGE constraint, is presented in Algorithm 1. In line 1, a special encoding is built, where an Y_i is introduced for each X_i with index in $ub(S)$. The domain of an Y_i is the same as that of X_i plus a dummy value. In line 2, HC on OCCURS removes a value from an Y_i each time it contains other values that are necessary to cover $lb(T)$ in every solution tuple. HC also removes values from $ub(T)$ that cannot be covered by any Y_i in a solution. Line 3 updates the bounds of S and the domain of Y_i's. Finally, in line 4, the channelling constraints between Y_i and X_i propagate removals on X_i for each i used in all solutions.

Theorem 2. *The algorithm Propag-Range is a correct algorithm for enforcing HC on* RANGE*, that runs in $O(nd + n \cdot |lb(T)|^{3/2})$ time, where d is the maximal size of X_i domains.*

Proof. Soundness. A value v is removed from $D(X_i)$ in line 4 if it is removed from Y_i together with *dummy* in lines 2 or 3. If a value is removed from Y_i in line 2, this means that any tuple on variables in Y covering $lb(T)$ requires that Y_i takes another value from $D(Y_i)$. So, we cannot find a solution of RANGE in which $X_i = v$ since $lb(T)$ must be covered as well. A value v is removed from $D(Y_i)$ in line 3 if $i \in lb(S)$ and $v \notin ub(T)$. In this case, RANGE cannot be satisfied by a tuple where $X_i = v$. If a value v is removed from $ub(T)$ in line 2, none of the tuples of values for variables in Y covering $lb(T)$ can cover v as well. Since variables in Y duplicate variables X_i with index in $ub(S)$, there is no hope to satisfy RANGE if v is in T. Note that $ub(T)$ cannot be modified in line 3 since Y contains only variables Y_i for which i was in $ub(S)$. If a value v is added to $lb(T)$ in line 3, this is because there exists i in $lb(S)$ such that $D(Y_i) \cap ub(T) = \{v\}$. Hence, v is necessarily in T in all solutions of RANGE. An index i can be removed from $ub(S)$ only in line 3. This happens when the domain of Y_i does not intersect $ub(T)$. In such a case, this is evident that a tuple where $i \in S$ could not satisfy RANGE since X_i could not take a value in T. Finally, if an index i is added to $lb(S)$ in line 3, this is because $D(Y_i)$ is included in $lb(T)$, which means that the dummy value has been removed from $D(Y_i)$ in line 2. This means that Y_i takes a value from $lb(T')$ in all solutions of OCCURS. X_i also has to take a value from $lb(T)$ in all solutions of RANGE.

Completeness (Sketch). Suppose that a value v is not pruned from $D(X_i)$ after line 4 of *Propag-Range*. If $Y_i \in Y$, we know that after line 2 there was an instantiation I on Y and T, solution of OCCURS with $I[Y_i] = v$ or with $Y_i = dummy$ (thanks to the channelling constraints in line 4). We can build the tuple I' on $X_1, ..X_n, S, T$ where X_i takes value v, every X_j with $j \in ub(S)$ and $I[Y_j] \in I[T]$ takes $I[Y_j]$, and the remaining X_j's take any value in their domain. T is set to $I[T]$ plus the values taken by X_j's with $j \in lb(S)$. These values are in $ub(T)$ thanks to line 3. Finally, S is set to $lb(S)$ plus the indices of the Y_j's with $I[Y_j] \in I[T]$. These indices are in $ub(S)$ since the only j's removed from $ub(S)$ in line 3 are such that $D(Y_j) \cap ub(T) = \emptyset$, which prevents $I[Y_j]$ from taking a value in $I[T]$. Thus I' is a solution of RANGE with $I'[X_i] = v$. We have proved that the X_i's are hybrid consistent after *Propag-Range*.

Suppose a value $i \in ub(S)$ after line 4. Thanks to constraint in line 3 we know there exists v in $D(Y_i) \cap ub(T)$, and so, $v \in D(X_i) \cap ub(T)$. Now, X_i is hybrid consistent after line 4. Thus $X_i = v$ belongs to a solution of RANGE. If we modify this solution by putting i in S and v in T (if not already there), we keep a solution.

Completeness on $lb(S)$, $lb(T)$ and $ub(T)$ is proved in a similar way.

Complexity. The important thing to notice in *Propag-Range* is that constraints in lines 2–4 are propagated in sequence. Thus, OCCURS is propagated only once, for a complexity in $O(nd + n \cdot |lb(T)|^{3/2})$. Lines 1, 3, and 4 are in $O(nd)$. Thus, the complexity of *Propag-Range* is in $O(nd + n \cdot |lb(T)|^{3/2})$. This reduces to linear time complexity when $lb(T)$ is empty. □

4.3 Range and Cardinality

Constraint toolkits like [13] additionally represent an interval on the cardinality of each set variable. This extra information can improve propagation. Unfortunately, enforcing HC on an extended version of the RANGE constraint which takes into account such cardinality information would be NP-hard (since we then subsume the NVALUE constraint). However, we can partially take into account such cardinality information. If $\text{RANGE}([X_1, \ldots, X_n], S, T)$ & $|T| = N$ & $|S| = M$, then we use the following decomposition:

$$
\begin{array}{ll}
 & \text{GCC}([Y_1, \ldots, Y_n], [1, \ldots, m+1], [B_1, \ldots, B_{m+1}]) \\
\forall i \in [1..n] & i \in S \leftrightarrow Y_i \in T \\
\forall i \in [1..n] & (X_i = Y_i) \vee (Y_i = m+1) \\
\forall v \in [1..m+1] & v \in T \leftrightarrow B_v \neq 0 \\
\forall v \in [1..m+1] & B_v \leq 1 + M - N \\
 & \sum_{v \in [1..m]} B_v = M
\end{array}
$$

where $m = |\bigcup_{i \in [1..n]}(D(X_i))|$ and GCC is the global cardinality constraint [3].

We have $\forall i \in [1..n],\ D(Y_i) = D(X_i) \cup \{m+1\}$ and $\forall v \in [1..m+1],\ D(B_v) = [0..n]$. We enforce GAC on the X's and Y's and BC on S, T and the B's. This algorithm has $O(n^2 d)$ complexity, which is typically worse than *Propag-Range* which ignores such cardinality information. It remains an open problem if we can extend *Propag-Range* to include some cardinality information, and if we can do so without changing its complexity.

5 Roots Constraint

The ROOTS constraint is more complex to handle.

Theorem 3. *Enforcing HC on the* ROOTS *constraint is NP-hard.*

Proof. We transform 3SAT into the problem of the existence of a solution for ROOTS. Let $\varphi = \{c_1, \ldots, c_m\}$ be a 3CNF on the Boolean variables $x_1, \ldots, x_n$. We build the constraint ROOTS$([X_1, \ldots, X_{n+m}], S, T)$ as follows. Each Boolean variable x_i is represented by the variable X_i with domain $D(X_i) = \{i, -i\}$. Each clause $c_p = x_i \vee \neg x_j \vee x_k$ is represented by the variable X_{n+p} with domain $D(X_{n+p}) = \{i, -j, k\}$. We build S and T in such a way that it is impossible that the index i of a Boolean variable x_i and its negative counterpart $-i$ both belong to T. So, we set $lb(T) = \emptyset$ and $ub(T) = \bigcup_{i=1}^{n}\{i, -i\}$, and $lb(S) = ub(S) = \{n+1, \ldots, n+m\}$. An interpretation M on the Boolean variables $x_1, \ldots, x_n$ is a model of φ iff the tuple τ in which $\tau[X_i] = i$ iff $M[x_i] = 0$ can be extended to a solution of ROOTS. (This extension puts in T value i iff $M[x_i] = 1$ and assigns X_{n+p} with the value corresponding to the literal satisfying c_p in M.) □

We thus have to look for a lesser level of consistency for ROOTS or for particular cases on which HC is polynomial.

5.1 Propagation Algorithm for Roots

The polynomial algorithm we propose to propagate the ROOTS constraint partially is based on the following six properties, directly entailed by HC:

P1 if $D(X_i)$ does not intersect $ub(T)$ then i is outside $ub(S)$
P2 if $D(X_i)$ is included in $lb(T)$ then i is in $lb(S)$
P3 if i is in $lb(S)$ then $D(X_i)$ is included in $ub(T)$
P4 if i is outside $ub(S)$ then $D(X_i)$ does not intersect $lb(T)$
P5 if v is the only possible value for X_i and $i \in lb(S)$ then v is in $lb(T)$
P6 if v is the only possible value for X_i and $i \notin ub(S)$ then v is outside $ub(T)$

In order to enforce these properties, we simply activate the head of a condition each time the body is true. A brute-force algorithm would loop on these six conditions until no more changes occur. However, this is not optimal in time. Algorithm 2 provides a linear way to enforce the six properties. It uses several data structures that avoid useless work. `Last_ubT`$[i]$ and `Last_lbT`$[i]$ are pointers as in AC2001 that prevent us from traversing the same domain several times. `Last_ubT`$[i]$ is the smallest value in $D(X_i) \cap ub(T)$ and `Last_lbT`$[i]$ is the smallest value in $D(X_i) \setminus lb(T)$. We store in `Cin_lbT` and `Cout_ubT` ("C" for *current*) the set of values that are added to $lb(T)$ and removed from $ub(T)$ during one pass on the six conditions. `Pin_lbT` and `Pout_ubT` ("P" for *previous*) contain those

Algorithm 2: `Algorithm for propagating Roots`

procedure *Propag-Roots*$([X_1, \ldots, X_n], S, T)$;
1 `Cout_ubT` $\leftarrow \bigcup_{i \in ub(S)} D(X_i) \setminus ub(T)$; `Cin_lbT` $\leftarrow lb(T)$;
2 **foreach** $i \in 1..n$ **do**
3   `Last_ubT`$[i] \leftarrow min(D(X_i))$; `Last_lbT`$[i] \leftarrow min(D(X_i))$;
4 **repeat**
5   `Pout_ubT` $\leftarrow$ `Cout_ubT`; `Cout_ubT` $\leftarrow \emptyset$; `Pin_lbT` $\leftarrow$ `Cin_lbT`; `Cin_lbT` $\leftarrow \emptyset$;
6   **foreach** $i \in 1..n$ **do**
7   **if** $i \in ub(S)$ **then**
8   **if** *Last_ubT*$[i] \in$ *Pout_ubT* ***and*** $i \notin lb(S)$ **then**
9   `Last_ubT`$[i] \leftarrow$ smallest $v >$ `Last_ubT`$[i]$ in $D(X_i) \cap ub(T)$;
10   **if** *Last_ubT*$[i] = nil$ **then** $ub(S) \leftarrow ub(S) \setminus \{i\}$;
11   **else**
12             $D(X_i) \leftarrow D(X_i) \setminus$ `Pin_lbT`;
13   **if** $|D(X_i)| = 1$ ***and*** $D(X_i) \subseteq ub(T)$ **then**
14                 $ub(T) \leftarrow ub(T) \setminus D(X_i)$; `Cout_ubT` $\leftarrow$ `Cout_ubT` $\cup D(X_i)$;
15   **if** $i \in lb(S)$ **then**
16             $D(X_i) \leftarrow D(X_i) \setminus$ `Pout_ubT`;
17   **if** $|D(X_i)| = 1$ ***and*** $D(X_i) \not\subseteq lb(T)$ **then**
18                 $lb(T) \leftarrow lb(T) \cup D(X_i)$; `Cin_lbT` $\leftarrow$ `Cin_lbT` $\cup D(Xi)$;
19   **else**
20   **if** *Last_lbT*$[i] \in$ *Pin_lbT* ***and*** $i \in ub(S)$ **then**
21   `Last_lbT`$[i] \leftarrow$ smallest $v >$ `Last_lbT`$[i]$ in $D(X_i) \setminus lb(T)$;
22   **if** *Last_lbT*$[i] = nil$ **then** $lb(S) \leftarrow lb(S) \cup \{i\}$;
23 **until** (*Cout_ubT* $\cup$ *Cin_lbT*) $= \emptyset$

changes of the previous pass, for which the consequences have to be propagated. This avoids multiple traversals of $lb(T)$ and $ub(T)$.

Line 1 initially sets `Cin_lbT` and `Cout_ubT` with the values already in $lb(T)$ and $ub(T)$ before propagation. Lines 2–3 initialize the pointers `Last_lbT`$[i]$ and `Last_ubT`$[i]$ to the smallest value in each domain $D(X_i)$. Then, the main loop is executed to enforce P1 to P6. Line 5 assigns `Pin_lbT` and `Pout_ubT` with the values of `Cin_lbT` and `Cout_ubT`, which are themselves reset empty before starting a new pass. For indices i not in $lb(S)$, lines 7–9 maintain `Last_ubT`$[i]$ as a witness of non emptiness of $D(X_i) \cap ub(T)$. If empty, line 10 enforces property P1. (For indices in $lb(S)$, P1 is subsumed by P3.) Line 12 enforces P4. Lines 13–14 enforce P6. Symmetrically, lines 15–22 enforce properties P2, P3, and P5.

For brevity, we omitted the tests for failure. They appear after each modification of a domain and return 'failure' if the domain becomes empty.

Theorem 4. *Propag-Roots enforces properties P1–P6 in $O(nd)$ time, where $d = max(max_i(|D(X_i)|), |ub(T)|)$.*

Proof. It is sufficient to show that when the test in line 23 is satisfied, all six properties hold (either the body is false or the head is true). By construction, all 6 properties are achieved at least once on S, T, and X_i's. They can become false because of some changes due to enforcing a property on other variables. First, we see that the changes on S (lines 10 and 22) cannot make false a property already satisfied. The changes on some X_i (lines 12 and 16) can invalidate properties P5 and P6 but they are re-enforced immediately after (lines 13–14 and 17–18). The only changes that require a new loop are those on T (lines 14 and 18). Now, these changes are stored in `Cout_ubT` and `Cin_lbT`. So, the test of line 23 cannot be satisfied before the changes on T are propagated on S and on all X_i.

The algorithm *Propag-Roots* is composed of a main loop (line 4) that is performed as many times as $lb(T)$ or $ub(T)$ are modified during a pass. Thus, the total number of times the loop is processed is bounded above by $|ub(T)|$. Since this main loop is itself composed of another loop (line 6) which is performed n times per call, we know that each line from lines 7 to 22 is performed at most $n \cdot |ub(T)|$ times. The lines which are not constant time are lines 9, 12, 16, and 21. But the data structures permit an amortized complexity on these lines which must thus be analysed globally for the total amount of work. In line 9, `Last_ubT`$[i]$ is a pointer that memorizes the last value found in $D(X_i) \cap ub(T)$. This ensures that $D(X_i)$ will be traversed at most once, for a cost in $O(|D(X_i)|)$ for a given index i. In lines 12 and 16, the total amount of work performed on $D(X_i)$ cannot exceed the number of values added to $lb(T)$ or removed from $ub(T)$, namely $O(|ub(T)|)$. Finally, in line 21, thanks to `Last_lbT`$[i]$ —a pointer that stores the last value found in $D(X_i) \setminus lb(T)$, $D(X_i)$ will be traversed at most once, for a cost in $O(|D(X_i)|)$ for a given index i. Therefore, *Propag-Roots* is in $O(nd)$. □

This algorithm partially propagates the ROOTS constraint. In the next section, we identify exactly when it achieves HC.

5.2 Polynomial Cases for HC on Roots

Many decompositions of counting and occurrence constraints don't use the ROOTS constraint in its more general form, but have some restrictions on the variables S, T or X_i's. We select four important special cases and show that enforcing HC on ROOTS is then tractable.

C1 $\forall i \in lb(S), D(X_i) \subseteq lb(T)$,
C2 $\forall i \notin ub(S), D(X_i) \cap ub(T) = \emptyset$,
C3 $X_1 \dots X_n$ are ground,
C4 T is ground.

In fact, we will show that *Propag-Roots* achieves HC in these cases. We first characterize two additional properties ensured by *Propag-Roots*.

Lemma 1. *Let* ROOTS$([X_1, \dots, X_n], S, T)$ *and* $i \in [1..n]$*:*

P7 *if Propag-Roots removes i from $ub(S)$ then $D(X_i)$ does not intersect $ub(T)$*
P8 *if Propag-Roots puts i in $lb(S)$ then $D(X_i)$ is included in $lb(T)$*

Proof. (P7) A value i can be removed from $ub(S)$ only in line 10 of *Propag-Roots*. This means that `Last_ubT[i]` is *nil*. So, according to line 9 this means that $D(X_i) \cap ub(T)$ is empty.

(P8) A value i can be added to $lb(S)$ only in line 22 of *Propag-Roots*. This means that `Last_lbT[i]` is *nil*. So, according to line 21 this means that $D(X_i) \setminus lb(T)$ is empty. □

We introduce a lemma that will simplify the proof of the main theorem.

Lemma 2. *If one of the conditions C1—C4 holds, then Propag-Roots fails iff the* Roots *constraint has no solutions.*

Proof. Suppose *Propag-Roots* does not fail. Build the following tuple τ of values for the X_i, S, and T. Initialize $\tau[S]$ and $\tau[T]$ with $lb(S)$ and $lb(T)$ respectively. Now, let us consider the four cases separately.

(C1) For each $i \in \tau[S]$ choose any value in $D(X_i)$ for $\tau[X_i]$. (By construction it is in $\tau[T]$ ($=lb(T)$) —from the assumption and from property P8. For each other i, assign X_i with any value in $D(X_i) \setminus lb(T)$. (This set is not empty thanks to property P2). τ obviously satisfies Roots.

(C2) For each $i \in \tau[S]$ ($=lb(S)$) choose any value in $D(X_i)$ for $\tau[X_i]$. (By construction such a value is in $ub(T)$ thanks to property P3). If necessary, add $\tau[X_i]$ to $\tau[T]$. For each other $i \in ub(S)$, assign X_i with any value in $D(X_i) \setminus \tau[T]$ if possible. Otherwise assign X_i with any value in $D(X_i)$ and add i to $\tau[S]$. For each $i \notin ub(S)$, assign X_i any value from its domain. By assumption and by property P7 we know that $D(X_i) \cap ub(T) = \emptyset$. Thus, τ satisfies Roots.

(C3) $\tau[X_i]$ is already assigned for all X_i. For each $i \in \tau[S]$, $\tau[X_i]$ is in $\tau[T]$ —property P5, and for each $i \notin lb(S)$, $\tau[X_i]$ is outside $lb(T)$ —property P2. τ satisfies Roots.

(C4) For each $i \in \tau[S]$ choose any value in $D(X_i)$ for $\tau[X_i]$. (By construction it is in $\tau[T]$ ($= ub(T)$) —by assumption and property P3). For each i outside $ub(S)$, assign X_i with any value in $D(X_i)$. (This value is outside $\tau[T]$ ($= lb(T)$) —property P4). For each other i, assign X_i with any value in $D(X_i)$ and update $\tau[S]$ if necessary. τ satisfies Roots.

Suppose now that *Propag-Roots* failed. This was necessarily in line 10, 12, 14, 16, 18 or 22. It is direct to see that Roots had no solution. □

Theorem 5. *If one of the conditions C1—C4 holds, then Propag-Roots is a correct algorithm for enforcing HC on* Roots.

Proof. Soundness. Soundness follows immediately from the algorithm.
Completeness. We have to prove that all the values in a $D(X_i)$ belong to a solution of Roots, and that the bounds on S and T are as tight as possible. Suppose the tuple τ is a solution of the Roots constraint. (We know such a tuple exists thanks to Lemma 2.)

Let $v \notin lb(T)$ and $v \in \tau[T]$. We show that there exists a solution with $v \notin \tau[T]$. (Remark that this case is irrelevant to condition C4.) We remove v from $\tau[T]$. For each $i \notin lb(S)$ such that $\tau[X_i] = v$ we remove i from $\tau[S]$. With C1 we are sure that none of the i in $lb(S)$ have $\tau[X_i] = v$ (thanks to property P8 and the fact that $v \notin lb(T)$). With C3 we are sure that none of the i in $lb(S)$ have $\tau[X_i] = v$ (thanks to property P5 and the fact that $v \notin lb(T)$). There remains to check C2. For each $i \in lb(S)$, we know that $\exists v' \neq v, v' \in D(X_i) \cap ub(T)$ (thanks to properties P3 and P5). We set X_i to v' in τ, we add v' to $\tau[T]$ and add all k with $\tau[X_k] = v'$ to $\tau[S]$ (we are sure that $k \in ub(S)$ since we are in condition C2 and $v' \in ub(T)$ —property P7).

Completeness on X_i's, $lb(S)$, $lb(T)$ and $ub(T)$ are shown with similar proofs. □

Algorithm 3: `Bound consistency on Roots`

procedure *Bound-Propag-Roots*$([X_1, \dots, X_n], S, T)$;

...

`Last_ubT`$[i] \leftarrow$ smallest $v >$ `Last_ubT`$[i]$ in $[min(D(X_i), max(D(X_i))] \cap ub(T)$;

...

$min(D(X_i)) \leftarrow min(D(X_i) \setminus lb(T))$; $max(D(X_i)) \leftarrow max(D(X_i) \setminus lb(T))$;

...

$min(D(X_i)) \leftarrow min(D(Xi) \cap ub(T))$; $max(D(X_i)) \leftarrow max(D(Xi) \cap ub(T))$;

...

`Last_lbT`$[i] \leftarrow$ smallest $v >$ `Last_lbT`$[i]$ in $[min(D(X_i), max(D(X_i))] \setminus lb(T)$;

...

5.3 Bound Consistency on Roots

In addition to being able to enforce HC on Roots in some special cases, *Propag-Roots* always enforces a level of consistency at least as strong as BC. In fact, we can relax some of the operations performed by *Propag-Roots* and still obtain a BC algorithm. In Algorithm 3 we present the lines that are modified wrt *Propag-Roots* in Algorithm 2.

Theorem 6. *The algorithm Bound-Propag-Roots is a correct algorithm for BC on the* Roots *constraint that runs in* $O(nd)$ *time.*

Proof. Soundness. Soundness follows immediately from the algorithm.
Completeness. The proof follows the same structure as that in Theorem 5. We relax the properties P1–P4 into properties P1'–P4'.

P1' if $[min(D(X_i)), max(D(X_i))]$ does not intersect $ub(T)$ then $i \notin ub(S)$
P2' if $[min(D(X_i)), max(D(X_i))]$ is included in $lb(T)$ then i is in $lb(S)$
P3' if i is in $lb(S)$ then the bounds of X_i are included in $ub(T)$
P4' if i is outside $ub(S)$ then the bounds of X_i are outside $lb(T)$

The properties P1'–P4', P5,P6 hold after *Bound-Propag-Roots* for the same reason as P1–P6 after *Propag-Roots*.

Let o be the total ordering on $D = \bigcup_i D(X_i) \cup ub(T)$. Build the tuples σ and τ as follows: For each $v \in lb(T)$: put v in $\sigma[T]$ and $\tau[T]$. For each $v \in ub(T) \setminus lb(T)$, following o, do: put v in $\sigma[T]$ or $\tau[T]$ alternately. For each $i \in lb(S)$, P3' guarantees that both $min(D(X_i))$ and $max(D(X_i))$ are in $ub(T)$. By construction of $\sigma[T]$ (and $\tau[T]$) with alternation of values, if $min(D(X_i)) \neq max(D(X_i))$, we are sure that there exists a value in $\sigma[T]$ (in $\tau[T]$) between $min(D(X_i))$ and $max(D(X_i))$. In the case $|D(X_i)| = 1$, P5 guarantees that the only value is in $\sigma[T]$ (in $\tau[T]$). Thus, we assign X_i in σ (in τ) with such a value in $\sigma[T]$ (in $\tau[T]$). For each $i \notin ub(S)$, we assign X_i in σ with a value in $[min(D(X_i)), max(D(X_i))] \setminus \sigma[T]$ (the same for τ). We know that such a value exists with the same reasoning as for $i \in lb(S)$ on alternation of values, and thanks to P4' and P6. We complete σ and τ by building $\sigma[S]$ and $\tau[S]$ consistently with the assignments of X_i and T. The resulting tuples satisfy Roots. From this we deduce that $lb(T)$ and $ub(T)$ are BC since all values in $ub(T) \setminus lb(T)$ are either in σ or in τ, but not both.

We show that the X_i are BC. Take any X_i and its lower bound $min(D(X_i))$. If $i \in lb(S)$ we know that $min(D(X_i))$ is in T either in σ or in τ thanks to P3' and by construction of σ and τ. We assign $min(D(X_i))$ to X_i in the relevant tuple. This remains a solution of ROOTS. If $i \notin ub(S)$, we know that $min(D(X_i))$ is outside T either in σ or in τ thanks to P4' and by construction of σ and τ. We assign $min(D(X_i))$ to X_i in the relevant tuple. This remains a solution of ROOTS. If $i \in ub(S) \setminus lb(S)$, assign X_i to $min(D(X_i))$ in σ. If $min(D(X_i)) \notin \sigma[T]$, remove i from $\sigma[S]$ else add i to $\sigma[S]$. The tuple obtained is a solution of ROOTS using the lower bound of X_i. By the same reasoning, we show that the upper bound of X_i is BC also, and therefore, all X_i's are BC.

We prove that $lb(S)$ and $ub(S)$ are BC with similar proofs.

Complexity. Using a very similar argument to before. □

6 Experimental Results

The purpose of this section is twofold. First, we demonstrate that decomposing global counting and occurrence constraints using RANGE and ROOTS is effective and efficient in practice. Second, we show that propagating RANGE and ROOTS using the algorithms introduced in this paper is more effective than propagating them using their straightforward decompositions:

$$\text{RANGE}([X_1, \ldots, X_n], S, T) \leftrightarrow \\ i \in S \rightarrow X_i \in T \;\wedge\; j \in T \rightarrow \exists i \in S. X_i = j$$

$$\text{ROOTS}([X_1, \ldots, X_n], S, T) \leftrightarrow \\ i \in S \leftrightarrow \; X_i \in T$$

We used a model for the Mystery Shopper problem [12] due to Helmut Simonis that appears in CSPLib. We used the same problem instances as in [6]. We partition the constraints of this problem into three groups:

1. All visits for any week are made by different shoppers. Similarly, a particular area cannot be visited more than once by the same shopper.
2. Each shopper makes exactly the number of visits he is assigned to.
3. A saleslady must be visited by some shoppers from at least 2 different groups (the shoppers are partitioned into groups).

Whilst the first group can be modelled by using ALLDIFF constraints [1], the second can be modelled by GCC [3] and the third by AMONG constraints [2]. We experimented with several models using Ilog Solver where these constraints are alternatively implemented as their Ilog Solver primitives (respectively, `IloAllDiff`, `IloDistribute`, and a decomposition using `IloSum` on Boolean variables) or RANGE and ROOTS. Due to space limitation, we report results for just half of the 2^3 possible models: `Alldiff-Gcc-Sum`, `Alldiff-Gcc`-ROOTS, `Alldiff`-ROOTS-ROOTS and finally RANGE-ROOTS-ROOTS.

When branching on the integer variables, the `Alldiff-Gcc-Sum` model is superior to every other model (see the #solved column in Table 1). However, we

	`Alldiff-Gcc-Sum`			`Alldiff-Gcc`-Roots			`Alldiff`-Roots-Roots			Range-Roots-Roots		
Size	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved
10	6	0.01	9/10	6	0.01	9/10	7	0.03	9/10	7	0.06	9/10
15	6566	0.76	29/52	6468	1.38	29/52	10749	19.47	28/52	10749	23.35	28/52
20	98497	14.52	21/35	2425	0.83	20/35	2429	7.30	20/35	2429	8.56	20/35
25	317	0.13	13/20	317	0.20	13/20	285	1.37	11/20	285	2.15	11/20
30	93461	26.09	7/10	93461	43.89	7/10	7239	42.00	5/10	7239	51.68	5/10
35	52435	16.33	22/56	23094	14.25	21/56	13	1.10	18/56	13	3.17	18/56

Table 1. Mystery Shopper, branching on the integer variable with minimum domain, Range and Roots are implemented using our algorithms. In this and the following tables, each instance has a 5 minutes limit, #fails and cpu time are averaged on the instances solved (#solved) by each method.

	`Alldiff-Gcc-Sum`			`Alldiff-Gcc`-Roots			`Alldiff`-Roots-Roots			Range-Roots-Roots		
Size	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved
10	6	0.01	9/10	4247	0.83	3/10	318	0.38	10/10	318	0.51	10/10
15	6566	0.76	29/52	17210	4.31	16/52	102	0.25	52/52	102	0.49	52/52
20	98497	14.52	21/35	150473	49.95	7/35	930	2.95	32/35	930	4.86	32/35
25	317	0.13	13/20	265219	124.49	2/20	2334	11.17	19/20	2334	18.43	19/20
30	93461	26.09	7/10	37	0.08	1/10	6766	39.63	9/10	4111	25.31	8/10
35	52435	16.33	22/56	1216	0.53	4/56	4798	35.60	49/56	4798	55.15	49/56

Table 2. Mystery Shopper, branching on set variables when possible, Range and Roots are implemented using our algorithms.

obtain the best results by branching on the set variables introduced for modelling with Range and Roots (see Table 2). By encoding the second and the third groups of constraints using Roots (the `Alldiff`-Roots-Roots model) and branching on the set variables, we are able to solve more instances. Replacing `Alldiff` with Range (the Range-Roots-Roots model) does not increase the number of fails except on one set of instances, but does increase run-times. This is not surprising as Range is a very general purpose global constraint. Nevertheless, the model using just Range and Roots constraints performs much better than the `Alldiff-Gcc-Sum` model, thanks to new variables we can branch on during search. The results indicate that encoding global counting and occurrence constraints using Range and Roots is effective and efficient in practice.

Finally we report in Table 3 the results obtained by the decompositions of Range and Roots. For each model we used the strategy that proved to be the most efficient between Table 1 and Table 2. As the number of Roots constraints increase and Range constraints are introduced, we observe a substantial gain using the algorithms introduced in this paper in preference to their decompositions.

7 Conclusion

We have presented a comprehensive study of Range and Roots, two global constraints that can express many other global constraints, such as occurrence and counting constraints [6]. We proposed an algorithm for propagating the Range constraint. We proved that the Roots constraint is intractable in general. Nevertheless, we proposed a linear algorithm to propagate it partially. This algorithm

	`Alldiff-Gcc-Sum`			`Alldiff-Gcc`-ROOTS			`Alldiff`-ROOTS-ROOTS			RANGE-ROOTS-ROOTS		
Size	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved
10	6	0.01	9/10	6	0.01	9/10	137	0.08	10/10	-	-	0/10
15	6566	0.76	29/52	6487	0.78	29/52	2263	0.41	52/52	-	-	0/52
20	98497	14.52	21/35	2425	0.54	20/35	30455	5.45	35/35	-	-	0/35
25	317	0.13	13/20	317	0.14	13/20	769931	129.44	16/20	-	-	0/20
30	93461	26.09	7/10	93461	27.11	7/10	-	-	0/10	-	-	0/10
35	52435	16.33	22/56	52386	16.55	22/56	-	-	0/56	-	-	0/56

Table 3. Mystery Shopper, branching on the integer variable with minimum domain for `Alldiff-Gcc-Sum` and `Alldiff-Gcc`-ROOTS, on set variables for `Alldiff`-ROOTS-ROOTS and RANGE-ROOTS-ROOTS. The constraints RANGE and ROOTS are implemented using their decomposition.

achieves hybrid consistency under some simple conditions and can be relaxed to achieve bound consistency in general. Our experiments show the benefit we can obtain by incorporating these two constraints in a constraint toolkit.

References

1. Régin, J.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings AAAI'94, Seattle WA (1994) 362–367
2. Beldiceanu, N., Contejean, E.: Introducing global constraints in chip. Mathl. Comput. Modelling **20** (1994) 97–123
3. Régin, J.: Generalized arc consistency for global cardinality constraint. In: Proceedings AAAI'96, Portland OR (1996) 209–215
4. Beldiceanu, N.: Pruning for the minimum constraint family and for the number of distinct values constraint family. In: Proceedings CP'01, Paphos, Cyprus (2001) 211–224
5. Beldiceanu, N., Katriel, I., Thiel, S.: Filtering algorithms for the *same* and *usedby* constraints. In: MPI Technical Report MPI-I-2004-1-001. (2004)
6. Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: The Range and Roots constraints: Specifying counting and occurrence problems. In: Proceedings IJCAI'05, Edinburgh, Scotland (2005)
7. Beldiceanu, N.: Global constraints as graph properties on a structured network of elementary constraints of the same type. Technical report, Swedish Institute of Computer Science (2000)
8. Pachet, F., Roy, P.: Automatic generation of music programs. In: Proceedings CP'99, Alexandria VA (1999) 331–345
9. Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of global constraints. In: Proceedings AAAI'04, San Jose CA (2004) 112–117
10. Ahuja, R., Magnanti, T., Orlin, J.: Network flows. Prentice Hall, Upper Saddle River NJ (1993)
11. Schrijver, A.: Combinatorial Optimization - Polyhedra and Efficiency. Springer-Verlag, Berlin (2003)
12. Cheng, B.M.W., Choi, K.M.F., Lee, J.H.M., Wu J.C.K. Increasing Constraint Propagation by Redundant Modeling: an Experience Report. *Constraints*, 1998.
13. *ILOG Solver 5.3 Reference and User Manual*, 2002.