Constraint programming,
First poly-time filtering algorithm
for Sequence constraint

Constraints Journal, 2009.

Best Paper Award at CP-06.

New Filtering Algorithms for Combinations of Among Constraints

Willem-Jan van Hoeve[1] Gilles Pesant[2,3]
Louis-Martin Rousseau[2,3,4] Ashish Sabharwal[1]

[1] Dept. of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.
{vanhoeve,sabhar}@cs.cornell.edu

[2] École Polytechnique de Montréal, Montreal, Canada
[3] Centre for Research on Transportation (CRT),
Université de Montréal, Montreal, H3C 3J7, Canada
[4] Oméga Optimisation Inc.
{pesant,louism}@crt.umontreal.ca

Abstract

Several combinatorial problems, such as car sequencing and rostering, feature `sequence` constraints, restricting the number of occurrences of certain values in every subsequence of a given length. We present three new filtering algorithms for the `sequence` constraint, including the first to establish domain consistency in polynomial time. The filtering algorithms have complementary strengths: One borrows ideas from dynamic programming; another reformulates it as a `regular` constraint; the last is customized. The last two algorithms establish domain consistency, and the customized one does so in polynomial time. We provide experimental results that show the practical usefulness of each. We also show that the customized algorithm equally applies to a generalized version of the `sequence` constraint for subsequences of varied lengths. The significant computational advantage of using one generalized `sequence` constraint over a semantically equivalent combination of `among` or `sequence` constraints is demonstrated experimentally.

1 Introduction

The `sequence` constraint appears in several combinatorial problems such as car manufacturing and rostering problems. It can be regarded as a collection of `among` constraints, that must hold simultaneously. An `among` constraint restricts the number of variables to be assigned to a value from a specific set. For example, consider a nurse-rostering problem in which each nurse can work at

most 2 night shifts during every 7 consecutive days. The `among` constraint specifies the 2-out-of-7 relation, while the `sequence` constraint imposes such `among` for every subsequence of 7 days.

The `sequence` constraint has been a topic of study in the constraint programming community since 1988, when the car sequencing problem was first introduced [7]. Initially, the `among` constraints were treated individually, while Beldiceanu and Contejean [4] first proposed to view them together as one global `sequence` constraint. The constraint is also referred to as `among_seq` in [3].

Beldiceanu and Carlsson [2] proposed a filtering algorithm for the `sequence` constraint, while Régin and Puget [13] presented a filtering algorithm for the `sequence` constraint in combination with a global cardinality constraint [11] for a car sequencing application. Neither approach, however, establishes domain consistency. As the constraint is inherent to many real-life problems, improved filtering can have a substantial industrial impact.

In this work we present three novel filtering algorithms for the `sequence` constraint. The first is based on dynamic programming concepts and runs in polynomial time, but it does not establish domain consistency. The second algorithm is based on the `regular` constraint [10] and establishes domain consistency. It needs exponential time in the worst case, but in many practical cases it is very efficient. Our third algorithm establishes domain consistency in polynomial time and is the first to do so. It can be applied to a generalized version of the `sequence` constraint, for which the subsequences are of varied length. Moreover the number of occurrences may also vary per subsequence. Each algorithm has advantages over the others, either in terms of (asymptotic) running time or in terms of filtering.

Our experimental results demonstrate that our newly proposed algorithms significantly improve the state of the art. On individual `sequence` constraints, these algorithms are much faster than the standard (partial) filtering implementations available in the Ilog Solver library; they often reduce the number of backtracks from over a hundred thousand to zero or near-zero. On the car sequencing problem benchmark, these algorithms are able to solve more instances or achieve substantial speed-up (either on their own or as a redundant constraint added to the Ilog sequence constraint). Finally, when certain more complex combinations of `among` constraints are present, such as in the rostering example above, our generalized `sequence` constraint implementation is able to treat them all as a single global constraint, and reduces the filtering time from around half an hour to just a few seconds.

The rest of the paper is structured as follows. Section 2 presents some background and notation on constraint programming. Section 3 recalls and discusses the `among` and `sequence` constraints. Sections 4 to 6 describe our three new filtering algorithms for `sequence`. Section 7 compares the algorithms experimentally. Finally, Section 8 summarizes the contributions of the paper and discusses possible extensions.

2 Constraint Programming Preliminaries

We first introduce basic constraint programming concepts. For more information on constraint programming we refer to [1] and [5].

Let x be a variable. The *domain* of x is a set of values that can be assigned to x and is denoted by $D(x)$. In this paper we only consider variables with *finite* domains. Let $X = x_1, x_2, \ldots, x_k$ be a sequence of variables. We denote $D(X) = \bigcup_{1 \leq i \leq k} D(x_i)$. A *constraint* C on X is defined as a subset of the Cartesian product of the domains of the variables in X, i.e. $C \subseteq D(x_1) \times D(x_2) \times \cdots \times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a *solution* to C. We also say that the tuple *satisfies* C. A value $d \in D(x_i)$ for some $i = 1, \ldots, k$ is *inconsistent* with respect to C if it does not belong to a tuple of C, otherwise it is *consistent*. C is *inconsistent* if it does not contain a solution. Otherwise, C is called *consistent*.

A *constraint satisfaction problem*, or a *CSP*, is defined by a finite sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_n$, together with a finite set of constraints $\mathcal{C}$, each on a subsequence of $\mathcal{X}$. The goal is to find an assignment $x_i = d_i$ with $d_i \in D(x_i)$ for $i = 1, \ldots, n$, such that all constraints are satisfied. This assignment is called a *solution to the CSP*.

The solution process of constraint programming interleaves *constraint propagation*, or *propagation* in short, and *search*. The search process essentially consists of enumerating all possible variable-value combinations, until we find a solution or prove that none exists. We say that this process constructs a *search tree*. To reduce the exponential number of combinations, *constraint propagation* is applied to each node of the search tree: Given the current domains and a constraint C, remove domain values that do not belong to a solution to C. This is repeated for all constraints until no more domain value can be removed. The removal of inconsistent domain values is called *filtering*.

In order to be effective, filtering algorithms should be efficient, because they are applied many times during the solution process. Further, they should remove as many inconsistent values as possible. If a filtering algorithm for a constraint C removes *all* inconsistent values from the domains with respect to C, we say that it makes C *domain consistent*. Formally:

Definition 1 (Domain consistency, [9])**.** A constraint C on the variables $x_1, \ldots, x_k$ is called *domain consistent* if for each variable x_i and each value $d_i \in D(x_i)$ $(i = 1, \ldots, k)$, there exist a value $d_j \in D(x_j)$ for all $j \neq i$ such that $(d_1, \ldots, d_k) \in C$.

In the literature, domain consistency is also referred to as *hyper-arc consistency* or *generalized-arc consistency*.

Establishing domain consistency for *binary constraints* (constraints defined on two variables) is inexpensive. For higher arity constraints this is not necessarily the case since the naive approach requires time that is exponential in the number of variables. Nevertheless the underlying structure of a constraint can sometimes be exploited to establish domain consistency much more efficiently.

3 The Among and Sequence Constraints

The `among` constraint restricts the number of variables to be assigned to a value from a specific set:

Definition 2 (Among constraint, [4])**.** Let $X = x_1, x_2, \ldots, x_q$ be a sequence of variables and let S be a set of domain values. Let $0 \leq \ell \leq u \leq q$ be constants. Then

$$\texttt{among}(X, S, \ell, u) = \{(d_1, \ldots, d_q) \mid \forall i \in \{1, \ldots, q\}\ d_i \in D(x_i), \\ \ell \leq |\{i \in \{1, \ldots, q\}\ :\ d_i \in S\}| \leq u\}.$$

Establishing domain consistency for the `among` constraint is not difficult. Subtracting from ℓ, u, and q the number of variables that must take their value in S, and subtracting further from q the number of variables that cannot take their value in S, we are in one of four cases:

1. $u < 0$ or $\ell > q$: the constraint is inconsistent;
2. $u = 0$: remove values in S from the domain of all remaining variables, making the constraint domain consistent;
3. $\ell = q$: remove values not in S from the domain of all remaining variables, making the constraint domain consistent;
4. $u > 0$ and $\ell < q$: the constraint is already domain consistent.

The `sequence` constraint applies the same `among` constraint on every q consecutive variables:

Definition 3 (Sequence constraint, [4])**.** Let $X = x_1, x_2, \ldots, x_n$ be an ordered sequence of variables (according to their respective indices) and let S be a set of domain values. Let $1 \leq q \leq n$ and $0 \leq \ell \leq u \leq q$ be constants. Then

$$\texttt{sequence}(X, S, q, \ell, u) \ = \ \bigwedge_{i=1}^{n-q+1} \texttt{among}(s_i, S, \ell, u),$$

where s_i represents the sequence $x_i, \ldots, x_{i+q-1}$.

In other words, the `sequence` constraint states that every sequence of q consecutive variables is assigned to at least ℓ and at most u values in S. Note that working on each `among` constraint separately, and hence locally, is not as powerful as reasoning globally. In particular, establishing domain consistency on each `among` of the conjunction does not ensure domain consistency for `sequence`.

Example 1. Let $X = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ be an ordered sequence of variables variables with domains $D(x_i) = \{0, 1\}$ for $i \in \{3, 4, 5, 7\}$, $D(x_1) = D(x_2) = \{1\}$, and $D(x_6) = \{0\}$. Consider the constraint $\texttt{sequence}(X, \{1\}, 5, 2, 3)$, i.e., every sequence of five consecutive variables must

account for two or three 1's. Each individual `among` is domain consistent but it is not the case for `sequence`: value 0 is unsupported for variable x_7. ($x_7 = 0$ forces at least two 1's among $\{x_3, x_4, x_5\}$, which brings the number of 1's for the leftmost `among` to at least four.)

Establishing domain consistency for the `sequence` constraint is not nearly as easy as for `among`. The algorithms proposed so far in the literature may miss such global reasoning. The filtering algorithm proposed in [13] and implemented in Ilog Solver does not filter out 0 from $D(x_7)$ in the previous example.

We note that domain consistency can be established in linear time when ℓ equals u. Namely, if there is a solution, then x_i must equal x_{i+q} because of the constraints $a_i + a_{i+1} + \cdots + a_{i+q-1} = \ell$ and $a_{i+1} + \cdots + a_{i+q} = \ell$. Hence, if one divides the sequence up into n/q consecutive subsequences of size q each, they must all look exactly the same. Thus, establishing domain consistency now amounts to propagating the "settled" variables (i.e. $D(x_i) \subseteq S$ or $D(x_i) \cap S = \varnothing$) to the first subsequence and then applying the previously described algorithm for `among`. Two of the filtering algorithms we describe below establish domain consistency in the general case, i.e., when ℓ does not necessarily equal u.

Without loss of generality, we shall consider instances of `sequence` in which $S = \{1\}$ and the domain of each variable is a subset of $\{0, 1\}$. Using an `element` constraint, we can map every value in S to 1 and every other value (i.e., $D(X) \backslash S$) to 0, yielding an equivalent instance on new variables.

4 A Graph-Based Filtering Algorithm

We propose a first filtering algorithm that considers the individual `among` constraints of which the `sequence` constraint is composed. First, it filters the `among` constraints for each sequence of q consecutive variables s_i, similar to the dynamic programming approach taken in [14] to filter knapsack constraints. Then it filters the conjunction of every pair of consecutive sequences s_i and s_{i+1}. This is presented as SuccessiveLocalGraph (SLG) in Algorithm 1, and discussed below.

4.1 Filtering the `among` Constraints

The individual `among` constraints are filtered with the algorithm FilterLocalGraph. For each sequence $s_i = x_i, \ldots, x_{i+q-1}$ of q consecutive variables in $X = x_1, \ldots, x_n$, we build a digraph $G_{s_i} = (V_i, A_i)$ as follows. The vertex set and the arc set are defined as

$$V_i = \{v_{j,k} \mid j \in \{i-1, \ldots, i+q-1\}, k \in \{0, \ldots, j\}\},$$

$$\begin{aligned} A_i = &\{(v_{j,k}, v_{j+1,k}) \mid j \in \{i-1, \ldots, i+q-2\}, k \in \{0, \ldots, j\}, D(x_{j+1}) \setminus S \neq \emptyset\} \cup \\ &\{(v_{j,k}, v_{j+1,k+1}) \mid j \in \{i-1, \ldots, i+q-2\}, k \in \{0, \ldots, j\}, D(x_{j+1}) \cap S \neq \emptyset\}. \end{aligned}$$

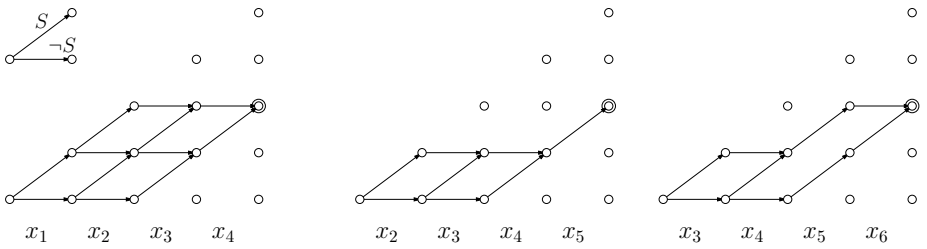

Figure 1: Filtered Local Graphs of Example 2.

In other words, the arc $(v_{j,k}, v_{j+1,k+1})$ represents variable x_{j+1} taking its value in S, while the arc $(v_{j,k}, v_{j+1,k})$ represents variable x_{j+1} not taking its value in S. The index k in $v_{j,k}$ represents the number of variables in $x_i, \ldots, x_j$ that take their value in S.

For each local graph G_{s_i}, we define a set of *goal vertices* as $\{v_{i+q-1,k} \mid \ell \leq k \leq u\}$. We then have the following immediate result:

Lemma 1. *A solution to* `among` *constraint on sequence s_i corresponds to a directed path from $v_{i-1,0}$ to a goal vertex in G_{s_i}, and vice versa.*

Next we apply Lemma 1 to make individual `among` constraints domain consistent. For the `among` constraint on sequence s_i, we remove all arcs that are not on any path from $v_{i-1,0}$ to a goal vertex in G_{s_i}. This can be done in linear time (in the size of the graph, $\Theta(q^2)$) by breadth-first search starting from the goal vertices. If the filtered graph contains no arc $(v_{j,k}, v_{j+1,k})$ for all k, we remove S from $D(x_{j+1})$. Similarly, we remove $D(X) \setminus S$ from $D(x_{j+1})$ if it contains no arc $(v_{j,k}, v_{j+1,k+1})$ for all k.

Example 2. Let $X = x_1, x_2, x_3, x_4, x_5, x_6$ be an ordered sequence of variables with domains $D(x_i) = \{0, 1\}$ for $i \in \{1, 2, 3, 4, 6\}$ and $D(x_5) = \{1\}$. Let $S = \{1\}$. Consider the constraint `sequence`$(X, S, 4, 2, 2)$. The filtered local graphs of this constraint are depicted in Figure 1.

4.2 Filtering for a Sequence of `among`

We next filter the conjunction of two "consecutive" `among` constraints. Our algorithm has a "forward" phase and a "backward" phase. In the forward phase, we compare the `among` on s_i with the `among` on s_{i+1} for increasing i, using the algorithm Compare. This is done by *projecting* $G_{s_{i+1}}$ onto G_{s_i} such that corresponding variables overlap. Doing so, the projection keeps only arcs that appear in both original local graphs. We can either project vertex $v_{i+1,0}$ of $G_{s_{i+1}}$ onto vertex $v_{i+1,0}$ of G_{s_i}, or onto vertex $v_{i+1,1}$ of G_{s_i}. We consider both projections separately, and label all arcs "valid" if they belong to a path from vertex $v_{i,0}$ to goal vertex in $G_{s_{i+1}}$ in one of the composite graphs. All other arcs are labeled "invalid", and are removed from the original graphs G_{s_i} and $G_{s_{i+1}}$.

```
SuccessiveLocalGraph(X, S, q, ℓ, u) begin
    build a local graph G_{s_i} for each sequence s_i (1 ≤ i ≤ n − q)
    for i = 1, ..., n − q do
        FilterLocalGraph(G_{s_i})
    for i = 1, ..., n − q − 1 do
        Compare(G_{s_i}, G_{s_i+1})
    for i = n − q − 1, ..., 1 do
        Compare(G_{s_i}, G_{s_i+1})
end

FilterLocalGraph(G_{s_i}) begin
    mark all arcs of G_{s_i} "invalid".
    by breadth-first search, mark every arc on a path from v_{i−1,0} to a goal
    vertex "valid"
    remove all invalid arcs
end

Compare(G_{s_i}, G_{s_{i+1}}) begin
    mark all arcs in G_{s_i} and G_{s_{i+1}} "invalid"
    for k = 0, 1 do
        project G_{s_{i+1}} onto vertex v_{i,k} of G_{s_i}
        by breadth-first search, mark all arcs on a path from v_{i−1,0} to a goal
        vertex in G_{s_{i+1}} "valid"
    remove all invalid arcs
end
```

Algorithm 1: Filtering algorithm for the `sequence` constraint.

In the backward phase, we compare the `among` on s_i with the `among` on s_{i+1} for decreasing i, similar to the forward phase.

4.3 Analysis

SuccessiveLocalGraph does not establish domain consistency for the sequence constraint. We illustrate this in the following example.

Example 3. Let $X = x_1, x_2, \ldots, x_{10}$ be an ordered sequence of variables with domains $D(x_i) = \{0, 1\}$ for $i \in \{3, 4, 5, 6, 7, 8\}$ and $D(x_i) = \{0\}$ for $i \in \{1, 2, 9, 10\}$. Let $S = \{1\}$. Consider the constraint $\texttt{sequence}(X, S, 5, 2, 3)$, i.e., every sequence of 5 consecutive variables must take between 2 and 3 values in S. The first `among` constraint imposes that at least two variables out of $\{x_3, x_4, x_5\}$ must be 1. Hence, at most one variable out of $\{x_6, x_7\}$ can be 1, by the third `among`. This implies that x_8 must be 1 (from the last `among`). Similarly, we can deduce that x_3 must be 1. This is however not deduced by our algorithm.

The problem occurs in the Compare method, when we merge the valid arcs coming from different projection. Up until that point there is a direct equivalence between a path in a local graph and a support for the constraint. However the union of the two projection breaks this equivalence and thus prevents this algorithm from being domain consistent.

The time complexity of the algorithm is polynomial since the local graphs are all of size $O(q \cdot u)$. Hence FilterLocalGraph runs in $O(q \cdot u)$ time, which is called $n - q$ times. The algorithm Compare similarly runs for $O(q \cdot u)$ steps and is called $2(n-q)$ times. Thus, the filtering algorithm runs in $O((n-q) \cdot q \cdot u)$ time. As $u \leq q$, it follows that the algorithm runs in $O(nq^2)$ time.

5 Reaching Domain Consistency through `regular`

The `regular` constraint [10], defining the set of allowed tuples for a sequence of variables as the language recognized by a given automaton, admits an incremental filtering algorithm establishing domain consistency. In this section, we give an automaton recognizing the tuples of the `sequence` constraint whose number of states is potentially exponential in q. Through that automaton, we can express `sequence` as a `regular` constraint, thereby obtaining domain consistency.

The idea is to record in a state the last q values encountered, keeping only the states representing valid numbers of 1's for a sequence of q consecutive variables and adding the appropriate transitions between those states. Let Q_k^q denote the set of strings of length q featuring exactly k 1's and $q - k$ 0's — there are $\binom{q}{k}$ such strings. Given the constraint $\texttt{sequence}(X, \{1\}, q, \ell, u)$, we create states for each of the strings in $\bigcup_{k=\ell}^{u} Q_k^q$. By a slight abuse of notation, we will refer to a state using the string it represents. Consider a state $d_1 d_2 \dots d_q$ in $Q_k^q, \ell \leq k \leq u$. We add a transition on 0 to state $d_2 d_3 \dots d_q 0$ if and only if $d_1 = 0 \vee (d_1 = 1 \wedge k > \ell)$. We add a transition on 1 to state $d_2 d_3 \dots d_q 1$ if and only if $d_1 = 1 \vee (d_1 = 0 \wedge k < u)$.

We must add some other states to encode the first $q-1$ values of the sequence: one for the initial state, two to account for the possible first value, four for the first two values, and so forth. There are at most $2^q - 1$ of those states, considering that some should be excluded because the number of 1's does not fall within $[\ell, u]$. More precisely, we will have states

$$\bigcup_{i=0}^{q-1} \bigcup_{k=\max(0,\ell-(q-i))}^{\min(i,u)} Q_k^i.$$

We define transitions from a state $d_1 \dots d_i$ in Q_k^i to state $d_1 \dots d_i 0$ in Q_k^{i+1} on value 0 and to state $d_1 \dots d_i 1$ in Q_{k+1}^{i+1} on value 1, provided such states are part of the automaton. Every state in the automaton is considered a final (accepting) state. Figure 2 illustrates the automaton that would be built for the constraint $\texttt{sequence}(X, \{1\}, 4, 1, 2)$.

The filtering algorithm for `regular` guarantees domain consistency provided that the automaton recognizes precisely the solutions of the constraint. By construction, the states $Q_\star^q$ of the automaton represent all the valid configurations of q consecutive values and the transitions between them imitate a shift to the right over the sequence of values. In addition, the states $Q_\star^i,\ 0 \leq i < q$ are linked

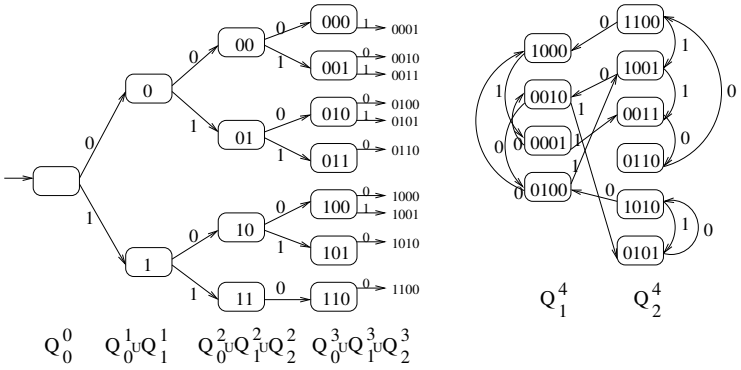

Figure 2: Automaton for $\texttt{sequence}(X, \{1\}, 4, 1, 2)$.

so that the first q values reach a state that encodes them. All states are accepting states so the sequence of n values is accepted if and only if the automaton completes the processing. Such a completion corresponds to a successful scan of every subsequence of length q, precisely our solutions.

The resulting algorithm runs in time linear in the size of the underlying graph, which has $O(n2^q)$ vertices and arcs in the worst case. Nevertheless, in many practical problems q is much smaller than n. Note also that subsequent calls of the algorithm run in time proportional to the number of updates in the graph and not to the size of the whole graph.

6 Reaching Domain Consistency in Polynomial Time

The filtering algorithms we considered thus far apply to `sequence` constraints with fixed `among` constraints for the same q, ℓ, and u. In this section we present a polynomial-time algorithm that achieves domain consistency in a more generalized setting, where we have m *arbitrary* among constraints over sequences of consecutive variables in X. These m constraints may have different ℓ and u values, be of different length, and overlap in an arbitrary fashion. However, they must be defined using the same set of values S. A conjunction of k `sequence` constraints over the same ordered set of variables, for instance, can be expressed as a *single* generalized sequence constraint. We define the generalized sequence constraint, `gen-sequence`, formally as follows:

Definition 4 (Generalized sequence constraint)**.** Let $X = x_1, \ldots, x_n$ be an ordered sequence of variables (according to their respective indices) and S be a set of domain values. For $1 \leq j \leq m$, let s_j be a sequence of consecutive variables in X, $|s_j|$ denote the length of s_j, and integers ℓ_j and u_j be such that $0 \leq \ell_j \leq u_j \leq |s_j|$. Let $\Sigma = \{s_1, \ldots, s_m\}$, $\mathrm{L} = \{\ell_1, \ldots, \ell_m\}$, and $\mathrm{U} =$

$\{u_1, \ldots, u_m\}$. Then

$$\texttt{gen-sequence}(X, S, \Sigma, \mathrm{L}, \mathrm{U}) \;=\; \bigwedge_{j=1}^{m} \texttt{among}(s_j, S, \ell_j, u_j).$$

For simplicity, we will identify each $s_j \in \Sigma$ with the corresponding `among` constraint on s_j. The basic structure of the filtering algorithm for the `gen-sequence` constraint is presented as Algorithm 2. The main loop, CompleteFilteringGS, simply considers all possible domain values of all variables. If a domain value is yet unsupported, we check its consistency via procedure CheckConsistency. If it has no support, we remove it from the domain of the corresponding variable.

Procedure CheckConsistency is the heart of the algorithm. It finds a single solution to the `gen-sequence` constraint, or proves that none exists. It uses a single array y of length $n+1$, such that $y[0] = 0$ and $y[i]$ represents the number of 1's among $x_1, \ldots, x_i$. The invariant for y maintained throughout is that $y[i+1] - y[i]$ is either 0 or 1. Initially, we start with the lowest possible array, in which y is filled according to the lower bounds of the variables in X.

For clarity, let L_j and R_j denote the left and right end-points, respectively, of the `among` constraint $s_j \in \Sigma$; $R_j = L_j + |s_j| - 1$. As an example, for the usual `sequence` constraint with `among` constraints of size q, L_j would be j and R_j would be $j + q - 1$. The *value* of s_j is computed using the array y: $\mathrm{value}(s_j) = y[R_j] - y[L_j - 1]$. In other words, $\mathrm{value}(s_j)$ counts exactly the number of 1's in the sequence s_j. Hence, the constraint s_j is satisfied if and only if $\ell_j \leq \mathrm{value}(s_j) \leq u_j$. In order to find a solution, we consider all `among` constraints $s_j \in \Sigma$. Whenever a constraint s_j is violated, we make it consistent by "pushing up" either $y[R_j]$ or $y[L_j - 1]$:

if $\mathrm{value}(s_j) < \ell_j$, then push up $y[R_j]$ with value $\ell_j - \;\mathrm{value}(s_j)$,

if $\mathrm{value}(s_j) > u_j$, then push up $y[L_j - 1]$ with value $\mathrm{value}(s_j) - u_j$.

Such a "push up" may result in the invariant for y being violated. We therefore *repair* y in a minimal fashion to restore its invariant as follows. Let $y[idx]$ be the entry that has been pushed up. We first push up its neighbors on the left side (from idx downward), starting with $idx - 1$. In case x_{idx-1} is fixed to 0, we push up $y[idx - 1]$ to the same level $y[idx]$. Otherwise, we push it up to $y[idx] - 1$. This continues until the difference between all neighbors to the left of idx is at most 1 and respects the values of the fixed variables. Whenever $y[i] > i$ for some i during this process, this indicates that we need more 1's than there are variables up to i, and we therefore report an immediate failure. Repairing the array on the right side of idx is done in a similar way. Here, in case x_{idx+1} is fixed to 1, we push up $y[idx + 1]$ to $y[idx] + 1$. Otherwise, we push it up to the same level $y[idx]$. The process continues to the right as far as necessary.

```
CompleteFilteringGS(X, S = {1}, Σ, L, U) begin
    for x_i ∈ X do
        for d ∈ D(x_i) do
            if CheckConsistency(x_i, d) = false then
                D(x_i) ← D(x_i) \ {d}
end
CheckConsistency(x_i, d) begin
    fix x_i = d, i.e., temporarily set D(x_i) = {d}
    y[0] ← 0
    for ℓ ← 1, ..., n do
        y[ℓ] ← number of forced 1's among x_1, ..., x_ℓ
    while a constraint s_j ∈ Σ is violated, i.e., value(s_j) < ℓ_j or value(s_j) > u_j
    do
        if value(s_j) < ℓ_j then
            idx ← right end-point of s_j
            PushUp(idx, ℓ_j − value(s_j))
        else
            idx ← left end-point of s_j
            PushUp(idx, value(s_j) − u_j)
        if s_j still violated then
            return false
    return true
end
PushUp(idx, v) begin
    y[idx] ← y[idx] + v
    if y[idx] > idx then return false
    // repair y on the left
    while
    (idx>0)∧((y[idx]−y[idx–1] > 1)∨((y[idx]−y[idx–1] = 1)∧(1 ∉ D(x_{idx−1}))))
    do
        if 1 ∉ D(x_{idx−1}) then
            y[idx − 1] ← y[idx]
        else
            y[idx − 1] ← y[idx] − 1
        if y[idx − 1] > idx − 1 then
            return false
        idx ← idx − 1
    // repair y on the right
    while
    (idx < n)∧((y[idx]−y[idx+1] > 0)∨((y[idx]−y[idx+1] = 0)∧(0 ∉ D(x_{idx}))))
    do
        if 0 ∉ D(x_{idx}) then
            y[idx + 1] ← y[idx] + 1
        else
            y[idx + 1] ← y[idx]
        idx ← idx + 1
end
```

Algorithm 2: Complete filtering algorithm for the `gen-sequence` constraint.

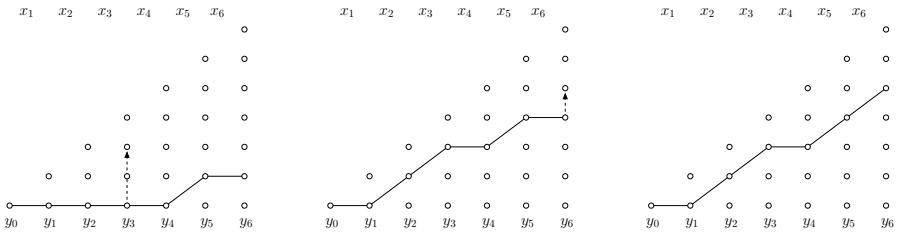

Figure 3: Finding a minimum solution to Example 4.

Example 4. Consider the constraint `sequence`$(X, S, 3, 2, 2)$ with $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $D(x_i) = \{0, 1\}$ for $i \in \{1, 2, 3, 4, 6\}$, $D(x_5) = \{1\}$, and $S = \{1\}$. The four `among` constraints are over $s_1 = \{x_1, x_2, x_3\}$, $s_2 = \{x_2, x_3, x_4\}$, $s_3 = \{x_3, x_4, x_5\}$, and $s_4 = \{x_4, x_5, x_6\}$. We apply CheckConsistency to find the "minimum" solution to this constraint. (We defer a formal discussion of the minimum solution to next subsection.) The various steps are depicted in Figure 3.

We start with $y = [0, 0, 0, 0, 0, 1, 1]$ because x_5 is forced to be 1, and then evaluate the different `among` constraints to check whether any of them is violated. We first consider s_1, which is violated: value$(s_1) = y[3] - y[0] = 0 - 0 = 0$, while it should be at least 2. Hence, we push up $y[3]$ by 2 units, and obtain $y = [0, 0, 1, 2, 2, 3, 3]$ after repairs. Note that we push up $y[5]$ to 3 because x_5 is fixed to 1. Next we consider s_2 with value $y[4] - y[1] = 2$, which is not violated. We continue with s_3 with value $y[5] - y[2] = 2$, which is not violated. Then we consider s_4 with value $y[6] - y[3] = 1$, which is violated as it should be at least 2. Hence, we push up $y[6]$ by 1, and obtain $y = [0, 0, 1, 2, 2, 3, 4]$. As there are no more violated `among` constraints, we conclude consistency, with minimum solution $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 1$.

The basic procedure in Algorithm 2 can be optimized in several ways; our implementation includes these optimizations. The main loop of CompleteFilteringGS is improved by maintaining a support for all domain values. Specifically, one call to CheckConsistency (with a positive response) yields a support for n domain values. This immediately reduces the number of calls to CheckConsistency by half, while in practice the cumulative reduction is even more. A second improvement is achieved by starting out CompleteFilteringGS with the computation of the "minimum" and the "maximum" solutions to `gen-sequence`, in a manner very similar to the computation in CheckConsistency but without restricting the value of any variable. This defines bounds $y_{\min}$ and $y_{\max}$ within which y must lie for all subsequent consistency checks (details are presented in the following section). As we will shortly see, this is further generalized to maintaining one minimum and one maximum solution for each variable-value pair, yielding an improvement in the cumulative time complexity of the algorithm from the root of the search tree to any leaf.

6.1 Analysis

A solution to a `gen-sequence` constraint can be thought of as the corresponding binary sequence given by the x array or, equivalently, as the cumulative y array. This y array representation has a useful property which we use for analyzing the correctness and the complexity of the algorithm. Let y and y' be two solutions. Define array $y \oplus y'$ to be the smaller of y and y' at each point, i.e., $(y \oplus y')[i] = \min(y[i], y'[i])$.

Lemma 2. *If y, y' are solutions to a* `gen-sequence` *constraint, then so is $y \oplus y'$.*

Proof. Suppose for the sake of contradiction that $y^* = y \oplus y'$ violates an `among` constraint s of the `gen-sequence` constraint. Let L and R denote the left and right end-points of s, respectively. Suppose y^* violates the ℓ constraint, i.e., $y^*[R] - y^*[L-1] < \ell(s)$. Since y and y' satisfy s, it must be that y^* agrees with y on one end-point of s and with y' on the other. W.l.o.g., assume $y^*[L-1] = y'[L-1]$ and $y^*[R] = y[R]$. By the definition of y^*, it must be that $y[L-1] \geq y'[L-1]$, so that $y[R] - y[L-1] \leq y[R] - y'[L-1] = y^*[R] - y^*[L-1] < \ell(s)$. In other words, y itself violates s, a contradiction. A similar reasoning works when y^* violates the u constraint of s. □

As a consequence of this property, we can unambiguously define an absolute *minimum* solution for `gen-sequence` as the one whose y value is the point-wise lowest over all solutions. Denote this solution by $y_{\min}$; we have that for all solutions y and for all i, $y_{\min}[i] \leq y[i]$. Similarly, define the absolute *maximum* solution, $y_{\max}$. For clarity, we will only focus on the minimum solution, which suffices for the proofs that follow; our implementation uses both the minimum and the maximum solutions to reduce the running time in practice.

Lemma 3. *The procedure* CheckConsistency *constructs the minimum solution to the* `sequence` *constraint, respectively* `gen-sequence` *constraint, or proves that none exists, in $O(n^2)$ time, respectively $O(n^3)$ time.*

Proof. CheckConsistency reports success only when no `among` constraint in `gen-sequence` is violated by the current y values maintained by it, i.e., y is a solution. Hence, if there is no solution, this fact is detected. We will argue that if there is a solution, CheckConsistency reports success and its current y array exactly equals $y_{\min}$.

We first show by induction that y never goes above $y_{\min}$ at any point, i.e., $y[i] \leq y_{\min}[i], 0 \leq i \leq n$ throughout the procedure. For the base case, $y[i]$ is clearly initialized to a value not exceeding $y_{\min}[i]$, and the claim holds trivially. Assume inductively that the claim holds after processing $t \geq 0$ `among` constraint violations. Let s be the $t+1^{st}$ violated constraint processed. We will show that the claim still holds after processing s.

Let L and R denote the left and right end-points of s, respectively. First consider the case that the ℓ constraint was violated, i.e., $y[R] - y[L-1] < \ell(s)$, and index R was pushed up so that the new value of $y[R]$, denoted $\hat{y}[R]$, became $y[L-1] + \ell(s)$. Since this was the first time a y value exceeded $y_{\min}$, we have

$y[L-1] \leq y_{\min}[L-1]$, so that $\hat{y}[R] = y[L-1] + \ell(s) \leq y_{\min}[L-1] + \ell(s) \leq y_{\min}[R]$. Therefore, $\hat{y}[R]$ itself does not exceed $y_{\min}[R]$. It may, in principle, still be the case that the resulting repair on the left or the right causes a $y_{\min}$ violation. However, the repair operations only lift up y values barely enough to be consistent with the possible domain values of the relevant variables. In particular, repair on the right "flattens out" y values to equal $\hat{y}[L-1]$ (forced 1's being exceptions) as far as necessary to "hit" the solution again. It follows that since $\hat{y}[R] \leq y_{\min}[R]$, all repaired y values must also not go above $y_{\min}$. A similar argument works when instead the u constraint is violated. This finishes the inductive step.

This shows that by performing repeated PushUp operations, one can never accidentally "go past" the solution $y_{\min}$ at any point. Further, since each PushUp increases y in at least one place, repeated calls to it will eventually "hit" $y_{\min}$ as a solution.

For the time complexity of CheckConsistency, note that $y[i] \leq i$. Since we monotonically increase the y values, we can do so at most $\sum_{i=1}^{n} i = O(n^2)$ times. The cost of each PushUp operation can be charged to the y values it changes because the while loops in it terminate as soon as they find a y value that need not be changed.

Finally, we discuss the detection of the violated `among` constraints during CheckConsistency. For this, we maintain a stack of indices, corresponding to (possibly) violated constraints. When processing a constraint, we remove it from the stack, and potentially push up some indices to make it consistent. Whenever we push up y_i, we insert all violated `among` constraints that have i as a left or right endpoint. In the case of the normal `sequence` constraint, we check in constant time whether s_i or s_{i+1-q} should be added to the stack. This yields the desired overall time complexity of $O(n^2)$. Notice that the stack is of size $O(n^2)$, because we only insert pushed-up indices, of which there are at most $O(n^2)$ during the process. In the case of the `gen-sequence` constraint, we maintain additionally two vectors indicating for each index i the `among` constraints that start, respectively end, at i. Notice that at most n constraints can start or end at an index i. Using this representation, the detection of violated `among` constraints for pushed up index i takes at most $O(n)$ time, which results in a net worst-case time complexity of $O(n^3)$ for the `gen-sequence` constraint. □

Theorem 1. *Algorithm* CompleteFilteringGS *establishes domain consistency on the* `sequence` *constraint, respectively the* `gen-sequence` *constraint, or proves that it is inconsistent. Further, it can be implemented to run in $O(n^3)$ time, respectively $O(n^4)$ time, along every path from the root to a leaf of the search tree.*

Proof. Lemma 3 and the simple loop structure of CompleteFilteringGS together imply that we obtain domain consistency, or prove inconsistency, each time the algorithm is called. To obtain the desired time complexity from the root to a leaf of the search tree, we maintain for every variable-value pair a minimum y solution. There are $O(n)$ such minimum solutions, each of size

n. When checking consistency of such a pair, we start from the corresponding minimum y solution and update it if we find a new one at this point in the search tree. As we proceed from the root to a leaf in the search tree, these minimum solutions can only (point-wise) increase. Moreover, the number of push-ups is at most $O(n^2)$ for each variable-value pair from the root to a leaf, following the same reasoning as in the proof of Lemma 3. It follows that for all n variables, we obtain a time complexity $O(n^3)$ for the `sequence` constraint, and $O(n^4)$ for the `gen-sequence` constraint, along every path from the root to a leaf in the search tree. □

7 Experimental Results

To evaluate the different filtering algorithms presented, we used three sets of benchmark problems. The first is a very simple model, constructed with only one sequence constraint, allowing us to isolate and evaluate the performance of each method separately. We then report on a series of experiments on the well-known car sequencing problem. Finally, we consider a combination of `sequence` constraints of varied lengths, to evaluate the `gen-sequence` constraint.

In the following, Successive Local Graph (SLG), Generalized Sequence (GS), and `regular`-based implementation (REG) are compared with the `sequence` constraint provided in the Ilog Solver library in both basic (IB) and extended (IE) propagation modes. Experiments were run with Ilog Solver 6.2 on a bi-processor Intel Xeon HT 2.8Ghz machine with 3GB RAM.

7.1 Single Sequence

To evaluate the filtering both in terms of domain reduction and efficiency, we build a very simple model consisting of only one sequence constraint. The instances are generated in the following manner. All instances contain n variables of domain size d and the S set is composed of the first $d/2$ elements. We generate a family of instances by varying the size of q and of the difference between ℓ and u, $\Delta = u - \ell$. For each family we try to generate 10 challenging instances by randomly filling the domain of each variable and by enumerating all possible values of ℓ. These instances are then solved using a random choice for both variable and value selection, keeping only the ones that are solved with more than 10 backtracks by method IB. All runs were stopped after one minute of computation. The runtimes in seconds (CPU) and the number of backtracks (BT) are reported as the average over various instances with the same parameters.

Table 1 reports on instances with a fixed number of variables (100) and varying q and Δ. Table 2 reports on instances with a fixed Δ (1) and growing number of variables.

These results demonstrate that the new filtering algorithms are very efficient. Both REG and SLG require no backtracks because they achieve domain consistency. The average number of backtracks for SLG is also typically very low. As predicted by its time complexity, GS is very stable for fixed n in the

q	Δ	IB		IE		SLG		REG		GS	
		BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
5	1	—	—	33,976.9	18.210	0.2	0.069	0	**0.009**	0	0.014
6	2	361,770	54.004	19,058.3	6.390	0	0.078	0	0.018	0	**0.013**
7	1	380,775	54.702	113,166.0	48.052	0	0.101	0	0.020	0	**0.012**
7	2	264,905	54.423	7,031.0	4.097	0	0.129	0	0.039	0	**0.016**
7	3	286,602	48.012	0	0.543	0	0.129	0	0.033	0	**0.015**
9	1	—	—	60,780.5	42.128	0.1	0.163	0	0.059	0	**0.010**
9	3	195,391	43.024	0	0.652	0	0.225	0	0.187	0	**0.016**

Table 1: *Comparison on instances with* $n = 100, d = 10$.

q	Δ	IB		IE		SLG		REG		GS	
		BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
5	50	459,154	18.002	22,812	18.019	0.4	0.007	0	**0.001**	0	**0.001**
5	100	192,437	12.008	11,823	12.189	1.0	0.041	0	**0.005**	0	**0.005**
5	500	48,480	12.249	793	41.578	0.7	1.105	0	**0.023**	0	0.466
5	1000	942	1.111	2.3	160.000	1.1	5.736	0	**0.062**	0	4.374
7	50	210,107	12.021	67,723	12.309	0.2	0.015	0	0.006	0	**0.001**
7	100	221,378	18.030	44,963	19.093	0.4	0.059	0	0.010	0	**0.005**
7	500	80,179	21.134	624	48.643	2.8	2.115	0	**0.082**	0	0.499
7	1000	30,428	28.270	46	138.662	588.5	14.336	0	**0.167**	0	3.323
9	50	18,113	1.145	18,113	8.214	0.9	0.032	0	0.035	0	**0.001**
9	100	3,167	0.306	2,040	10.952	1.6	0.174	0	0.087	0	**0.007**
9	500	48,943	18.447	863	65.769	2.2	4.311	0	0.500	0	**0.485**
9	1000	16,579	19.819	19	168.624	21.9	16.425	0	**0.843**	0	3.344

Table 2: *Comparison on instances with* $\Delta = 1, d = 10$.

first table but becomes more time consuming as n grows in the second table. The performance of SLG and REG decreases as q grows but REG remains competitive throughout these tests. We expect that the latter would suffer with still larger values of q and Δ but it proved difficult to generate challenging instances in that range — they tended to be loose enough to be easy for every algorithm.

7.2 Car Sequencing

In order to evaluate our algorithms in a more realistic setting, we turned to the car sequencing problem (see prob001 of CSPLib [8] for a detailed description). We ran experiments using the first set of 78 instances on CSPLib. Table 3 compares and contrasts the effectiveness of the following combinations of `among` constraints. IE, as before, is the extended IloSequence constraint available in the ILOG Solver library. This constraint also allows one to specify individual cardinalities for values in the set S, and is thus richer than our basic version of `sequence`. REG, again as before, is the `sequence` constraint encoded as a `regular` constraint. cREG extends REG by including a cost variable [6] which restricts the total number of cars with a particular option. This is more expressive than the normal `sequence` constraint, but not as rich as the IloSequence constraint IE. Finally, GS here is the `gen-sequence` constraint which includes the normal `sequence` constraint with an additional `among` constraint restricting

instance	IE	REG	cREG	GS	IE+REG	IE+cREG	IE+GS
carseq01	0.39	0.04	0.06	**0.03**	0.41	0.49	0.42
carseq02	83.83	—	19.92	19.63	95.99	19.92	**18.57**
carseq03	0.60	—	**0.04**	0.65	0.76	0.74	0.63
carseq07	112.97	—	**18.24**	27.70	138.19	114.98	93.38
carseq08	0.34	0.07	0.08	**0.03**	0.36	0.41	0.32
carseq09	18.74	2253.14	21.82	**16.92**	22.80	20.71	18.50
carseq10	—	—	1361.94	912.00	—	4.06	**3.64**
carseq11	1.31	—	0.46	**0.20**	1.32	1.66	1.67
carseq12	24.36	—	—	—	**23.93**	38.75	48.80
carseq13	1.37	—	0.35	**0.18**	1.40	1.76	1.53
carseq14	**2.93**	—	—	—	**2.92**	3.16	3.11
carseq15	3.13	**0.07**	0.41	0.23	3.47	3.45	3.70
carseq16	87.09	—	—	—	105.28	3.50	**2.84**
carseq17	2.83	**0.07**	0.43	0.32	3.13	3.44	3.46
carseq19	—	—	2.54	**1.35**	—	—	—
carseq20	—	—	0.43	**0.18**	—	9.58	8.51
carseq21	**1.54**	—	—	—	1.56	1.92	1.69
carseq22	451.83	—	0.47	**0.18**	516.35	899.38	914.96
carseq23	1.58	—	—	—	**1.53**	1.92	1.85
carseq24	**3.35**	—	—	—	3.37	3.63	3.27
carseq30	—	—	1774.50	**1484.62**	—	—	—
carseq31	1.70	71.91	0.48	**0.18**	1.71	2.09	1.95
carseq33	5.11	239.02	2.64	**1.46**	6.73	7.99	6.26
carseq34	**3.75**	—	—	—	3.79	4.20	3.94
carseq41	1.83	**0.07**	0.34	0.16	1.86	2.20	1.97
carseq43	**4.69**	137.48	28.75	12.78	4.83	5.46	5.24
carseq44	3.94	—	—	—	**3.89**	4.19	4.03
carseq48	4.04	**0.46**	3.88	3.07	4.14	4.12	4.29
carseq49	**4.00**	93.10	1928.90	659.18	4.01	4.60	4.28
carseq51	4.46	**0.07**	0.40	0.20	4.49	4.93	4.27
carseq53	4.35	—	0.58	**0.35**	4.90	4.71	4.64
carseq54	4.54	**0.08**	0.68	0.37	4.15	4.96	4.81
carseq55	4.46	**0.06**	0.40	0.23	4.13	4.41	4.70
carseq58	4.30	**0.08**	0.43	0.34	4.30	2.42	2.12
carseq59	4.15	**0.06**	0.35	0.23	4.23	4.53	4.41
carseq61	**5.05**	—	—		5.19	5.40	5.26
carseq62	3.24	**0.07**	0.33	0.25	3.34	3.61	3.46
carseq63	**4.98**	1321.83	190.42	68.42	5.45	5.80	5.65
carseq65	**5.07**	—	—		5.10	5.45	5.41
carseq67	3.08	**0.12**	1.25	0.61	3.72	3.52	2.93
carseq70	6.33	**0.08**	0.37	0.29	6.46	6.75	6.65
carseq72	202.28	—	—	—	213.92	180.55	**154.64**
carseq73	**5.96**	—	—	—	6.08	6.57	6.23
carseq74	5.92	**0.06**	0.50	0.33	6.02	6.26	6.24
carseq75	3.16	**0.06**	0.34	0.18	1.78	2.00	2.15
carseq76	7.03	**2.02**	4.38	2.38	7.00	7.50	7.30
carseq78	5.55	**0.08**	0.34	0.27	5.60	5.85	5.88
total solved	43	24	35	35	43	45	45

Table 3: *Runtime comparison on car sequencing problems.*

instance			gcc's + sequence's		gen-sequence	
characteristics	size	#solutions	BT	CPU	BT	CPU
max6/8-min22/30	40	2,284	185,287	216.49	0	**0.77**
	50	4,575	186,408	369.12	0	**2.09**
	60	6,567	188,242	621.99	0	**3.60**
	70	2,810	195,697	840.52	0	**1.88**
	80	730	198,091	1061.62	0	**0.61**
max6/9-min20/30	40	3	393,748	390.93	0	**0.01**
	50	3	393,748	660.74	0	**0.02**
	60	3	393,748	1074.26	0	**0.03**
	70	3	393,748	1432.20	0	**0.04**
	80	3	393,748	1786.62	0	**0.05**
max7/9-min22/30	40	137,593	328,376	417.63	0	**34.43**
	50	388,726	456,937	1061.24	0	**150.87**
	60	718,564	729,766	2822.09	0	**339.89**
	70	105,618	1,743,518	5048.84	0	**60.82**
	80	22,650	1,847,335	7457.36	0	**15.41**

Table 4: `gen-sequence` *constraint on sequences of varied lengths.*

the total number of cars with a particular option, similar to cREG.

In addition to these constraints, we also consider the combinations IE+REG, IE+cREG, and IE+GS, where REG, cREG, and GS, respectively, are added to the IloSequence constraint as redundant constraints. We note that all versions were run with two different search heuristics: the specialized ordering proposed by Régin and Puget [13] for the car sequencing problems, and the min-domain ordering. In the table, we report the best time (in seconds) for each version.[1] The time out used for these experiments was one hour.

Table 3 indicates that no single filtering method clearly dominates the others on this relatively complex problem domain. On the positive side, it also shows that our proposed algorithms can be very effective here, either applied solely or as a redundant constraint in conjunction with IloSequence. Specifically, many of the instances can be solved much faster with one of our algorithms than with IloSequence. For example, on all instances that are solved by both GS and IE, the GS algorithm achieves a median speed-up of 11.3 times. Furthermore, there are 4 instances (namely, carseq10, carseq19, carseq20, and carseq30) that cannot be solved using IloSequence alone within a one hour time limit, while applying our algorithms cREG and GS does allow solving them. The most striking examples are carseq19 and carseq20, which can now be solved in 0.18 and 1.35 seconds, respectively, by using the GS algorithm.

7.3 Sequence Constraints of Varied Lengths

Table 4 evaluates the performance of the `gen-sequence` constraint on three families of instances constructed so that they are challenging for a CP approach using previously known constraints. The instances specify restrictions for work/rest patterns in individual schedules of rostering problems, inspired

[1] Details of these experiments are available from the authors upon request.

by contexts in which several levels of time granularity (day, week, month, year) coexist [15]. Every instance requires between 4 and 5 days worked per calendar week. Its identification, of the form "maxA/B-minC/D", indicates that at most A days are worked in any B consecutive days but that at least C days are worked in any D consecutive days. Instance size, i.e., the number of days in the scheduling horizon, varies from 40 to 80 within each family. We wish to enumerate all the solutions. The "`gcc`'s + `sequence`'s" model uses one `gcc` constraint per week and two separate `sequence` constraints for the A/B and C/D restrictions. Note that the `sequence` constraints are implemented with our customized algorithm. The "`gen-sequence`" model uses a single `gen-sequence` constraint. Since we achieve domain consistency for this constraint and the whole problem is captured by this one constraint, the number of backtracks is always zero. These few experiments suffice to show that using the `gen-sequence` constraint in the general setting can lead to computational savings of several orders of magnitude.

8 Discussion

We proposed, analyzed, and experimentally evaluated three new filtering algorithms for the `sequence` constraint. They have different strengths that complement each other well. The local graph approach of Section 4 does not guarantee domain consistency but causes quite a bit of filtering, as witnessed in the experiments. Its asymptotic time complexity is $O(nq^2)$. The reformulation as a `regular` constraint, described in Section 5, establishes domain consistency but its asymptotic time and space complexity are exponential in q, namely $O(n2^q)$. Nevertheless it performs very well, partly due to its incremental nature, for small values of q, not uncommon in applications: in car sequencing, values between 2 and 5 are frequent; in rostering, the shift assignment problem typically features values between 5 and 9 whereas the shift construction problem may require values up to 12. The customized algorithm of Section 6 also establishes domain consistency on the `sequence` constraint. It has an asymptotic time complexity that is polynomial in n, namely $O(n^3)$ along each path from the root to a leaf in the search tree. Also in practice this algorithm performed very well, being often even faster than the local graph approach. It should be noted that previously known algorithms did not establish domain consistency.

In our experimental section, we demonstrated the advantages of our proposed algorithms in practice. On single `sequence` constraints with various parameters, our algorithms outperform the state of the art with several orders of magnitude. Also in more realistic settings such as the car sequencing problem, we showed that our algorithms allowed solving more instances, or again could improve the state of the art significantly. Finally, on more complex combinations of `among` constraints, our generalized `sequence` algorithm achieved computational savings of orders of magnitude.

Our contribution extends beyond the `sequence` constraint and into more general combinations of `among` (-like) constraints. The algorithm of Section

6 also establishes domain consistency in $O(n^4)$ time on freer combinations of `among` constraints, as long as each is defined on consecutive variables with respect to a fixed ordering. Not every combination of `among` constraints is tractable — Régin proved that finding a solution to an arbitrary combination of `among` constraints is NP-complete [12]. Another interesting extension for our first two algorithms is that they lend themselves to a generalization of `among` in which the number of occurrences is represented by a set (as opposed to an interval of values).

Acknowledgments

We thank Marc Brisson and Sylvain Mouret for their help with some of the implementations and experiments.

This work was partially supported by the Canadian Natural Sciences and Engineering Research Council (Discovery grant OGP0218028), the Intelligent Information Systems Institute (IISI), Cornell University (AFOSR grant FA9550-04-1-0151), and DARPA (REAL grant FA8750-04-2-0216, COORDINATORs grant FA8750-05-C-0033). A preliminary version of this work appeared in the 12^{th} International Conference on Principles and Practice of Constraint Programming (CP-06), Nantes, France, September 2006.

References

[1] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] N. Beldiceanu and M. Carlsson. Revisiting the Cardinality Operator and Introducing the Cardinality-Path Constraint Family. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming (ICLP 2001)*, volume 2237 of *LNCS*, pages 59–73. Springer, 2001.

[3] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog. Technical Report T2005-08, SICS, 2005.

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

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[6] S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11:315–333, 2006.

[7] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 290–295, 1988.

[8] I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, TR APES-09-1999, 1999. Available at http://www.csplib.org.

[9] R. Mohr and G. Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI)*, pages 651–656, 1988.

[10] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.

[11] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, volume 1, pages 209–215. AAAI Press / The MIT Press, 1996.

[12] J.-C. Régin. Combination of Among and Cardinality Constraints. In R. Barták and M. Milano, editors, *Proceedings of the Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2005)*, volume 3524 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2005.

[13] J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP97)*, volume 1330 of *LNCS*, pages 32–46. Springer, 1997.

[14] M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.

[15] T. Zemmouri, P. Chan, M. Hiroux, and G. Weil. Multiple-level Models: an application to employee timetabling. In E. K. Burke and M. Trick, editors, *Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling (PATAT'04)*, pages 397–412, 2004.