Revisiting the Sequence Constraint

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

[1] Department of Computer Science, Cornell University,
4130 Upson Hall, Ithaca, NY 14853, USA
{vanhoeve, sabhar}@cs.cornell.edu
[2] École Polytechnique de Montréal, Montreal, Canada
[3] Centre for Research on Transportation (CRT),
Université de Montréal, C.P. 6128, succ. Centre-ville, Montreal, H3C 3J7, Canada
[4] Oméga Optimisation Inc.
{pesant, louism}@crt.umontreal.ca

Abstract. Many 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 width. To date, none of the filtering algorithms proposed guaranteed domain consistency. In this paper, we present three filtering algorithms for the `sequence` constraint, with 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. Our customized algorithm does so in polynomial time, and can even be applied to a generalized `sequence` constraint for subsequences of variable widths. Experimental results show the practical usefulness of each.

1 Introduction

The `sequence` constraint was introduced by Beldiceanu and Contejean [4] as a set of overlapping `among` constraints. The constraint is also referred to as `among_seq` in [3]. 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.

Beldiceanu and Carlsson [2] have proposed a filtering algorithm for the `sequence` constraint, while Régin and Puget [10] have presented a filtering algorithm for the `sequence` constraint in combination with a global cardinality constraint [8] for a car sequencing application. Neither approach establishes domain consistency, however. As the constraint is inherent to many real-life problems, improved filtering could 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 [7]. It establishes domain consistency,

F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 620–634, 2006.

but needs exponential time in the worst case. In most practical cases it is very efficient however. Our third algorithm establishes domain consistency in polynomial time. It can be applied to a generalized version of the `sequence` constraint, for which the subsequences are of variable 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.

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 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].

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, [6]). *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 $0 \leq \min \leq \max \leq q$ be constants. Then*

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

Establishing domain consistency for the `among` constraint is not difficult. Subtracting from min, max, 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. $\max < 0$ or $\min > q$: the constraint is inconsistent;
2. $\max = 0$: remove values in S from the domain of all remaining variables, making the constraint domain consistent;
3. $\min = q$: remove values not in S from the domain of all remaining variables, making the constraint domain consistent;
4. $\max > 0$ and $\min < q$: 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 $0 \leq \min \leq \max \leq q$ be constants. Then*

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

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 min and at most max 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 `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 [10] and implemented in Ilog Solver does not filter out 0 from $D(x_7)$ in the previous example. However in some special cases domain consistency can be efficiently computed: When min equals max, it can be established in linear time. Namely, if there is a solution, then x_i must equal x_{i+q} because of the constraints $a_i + a_{i+1} + \cdots + a_{i+q-1} = \min$ and $a_{i+1} + \cdots + a_{i+q} = \min$. 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.

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. 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

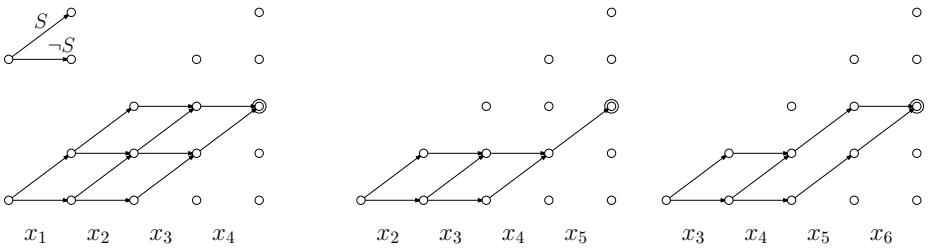

Fig. 1. Filtered Local Graphs of Example 2

$X = x_1, \ldots, x_n$, we build a digraph $G_{s_i} = (V_i, A_i)$ as follows. The vertex set and the arc set are defined as

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

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

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 $x_i, \ldots, x_{j-1}$ that take their value in S. This is similar to the dynamic programming approach taken in [11] to filter knapsack constraints.

Next, the individual `among` constraint on sequence s_i is filtered by removing all arcs that are not on a path from vertex $v_{i-1,0}$ to a *goal vertex* $v_{i+q-1,k}$ with $\min \leq k \leq \max$. 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. Naturally, 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 filter the conjunction of two "consecutive" `among` constraints. This 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

Algorithm 1. Filtering algorithm for the `sequence` constraint

```
SuccessiveLocalGraph(X, S, q, min, max) 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} as invalid.
    by breadth-first search, mark as valid every arc on a path from v_{i−1,0} to a goal vertex
    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
```

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 $G_{s_{i+1}}$. In the backward phase, we compare the `among` on s_i with the `among` on s_{i+1} for decreasing i, similarly 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 `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 complexity of the algorithm is polynomial since the local graphs are all of size $O(q \cdot \max)$. Hence FilterLocalGraph runs in $O(q \cdot \max)$ time, which is called $n - q$ times. The algorithm Compare similarly runs for $O(q \cdot \max)$ steps

and is called $2(n-q)$ times. Thus, the filtering algorithm runs in $O((n-q)\cdot q\cdot \max)$ time. As $\max \leq q$, it follows that the algorithm runs in $O(nq^2)$ time.

5 Reaching Domain Consistency Through `regular`

The `regular` constraint [7], 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

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

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 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.

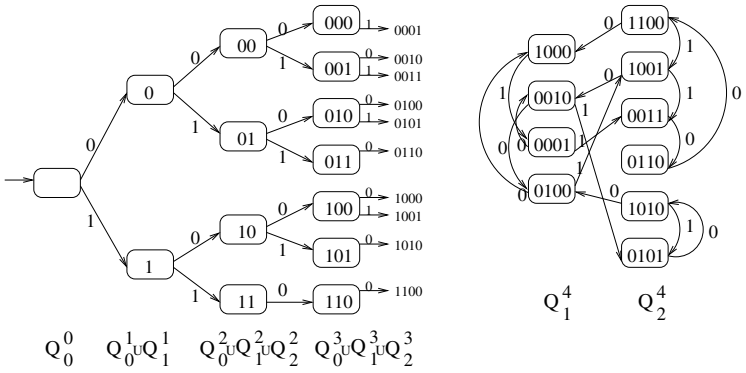

Fig. 2. Automaton for $\texttt{sequence}(X, \{1\}, 4, 1, 2)$

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 most 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, min, and max. 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 min and max values, be of different length, and overlap in an arbitrary fashion. 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 $\min_j$ and $\max_j$ be such that $0 \leq \min_j \leq \max_j \leq |s_j|$. Let $\Sigma = \{s_1, \ldots, s_m\}$, $\mathrm{Min} = \{\min_1, \ldots, \min_m\}$, and $\mathrm{Max} = \{\max_1, \ldots, \max_m\}$. Then*

$$\texttt{gen-sequence}(X, S, \Sigma, \mathrm{Min}, \mathrm{Max}) \;=\; \bigwedge_{j=1}^{m} \texttt{among}(s_j, S, \min_j, \max_j).$$

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`

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

```
CompleteFiltering(X, S = {1}, Σ, Min, Max) 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) < min_j or value(s_j) > max_j do
        if value(s_j) < min_j then
            idx ← right end-point of s_j
            PushUp(idx, min_j − value(s_j))
        else
            idx ← left end-point of s_j
            PushUp(idx, value(s_j) − max_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
```

constraint is presented as Algorithm 2. The main loop, CompleteFiltering, 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 i and R_j would be $i+q-1$. The *value* of s_j is computed using the array y: $\text{value}(s_j) = y[R_j] - y[L_j - 1]$. In other words, $\text{value}(s_j)$ counts exactly the number of 1's in the sequence s_j. Hence, a constraint s_j is satisfied if and only if $\min_j \leq \text{value}(s_j) \leq \max_j$. 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 $\text{value}(s_j) < \min_j$, then push up $y[R_j]$ with value $\min_j - \text{value}(s_j)$,
if $\text{value}(s_j) > \max_j$, then push up $y[L_j - 1]$ with value $\text{value}(s_j) - \max_j$.

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). 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 is at most 1. Whenever $y[i] > i$ for some i, we need more 1's than there are variables up to i, and we report an immediate failure. Repairing the array on the right side is done in a similar way.

Example 4. Consider again the `sequence` constraint from Example 2, i.e., the constraint $\texttt{sequence}(X, S, 4, 2, 2)$ 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 a minimum solution. The different steps are depicted in Figure 3. We start with $y = [0, 0, 0, 0, 0, 1, 1]$, and consider the different `among` constraints. First we consider s_1, which is violated. Namely, $\text{value}(s_1) = y[3] - y[0] = 0 - 0 = 0$, while it should be at least 2. Hence, we push up $y[3]$ with 2 units, and obtain $y = [0, 0, 1, 2, 2, 3, 3]$. 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 $y = [0, 0, 1, 2, 2, 3, 4]$. One more loop over the `among` constraint concludes consistency, with minimum solution $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 1$.

We have optimized the basic procedure in Algorithm 2 in several ways. The main loop of CompleteFiltering is improved by maintaining a support for all domain values. Namely, one call to CheckConsistency (with positive response) yields a support for n domain values. This immediately reduces the number of calls to CheckConsistency by half, while in practice the reduction is even more. A second improvement is achieved by starting out CompleteFiltering 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

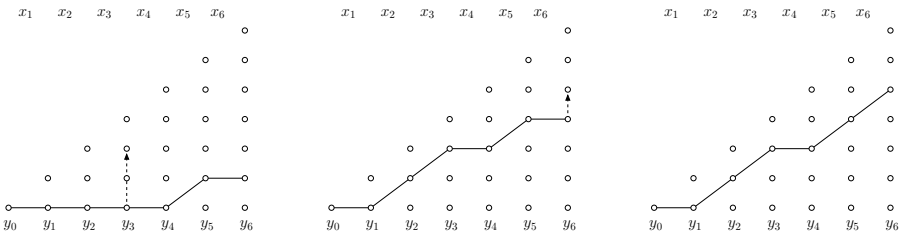

Fig. 3. Finding a minimum solution to Example 4

y_{min} and y_{max} within which y must lie for all subsequent consistency checks (details in the following section).

6.1 Analysis

A solution to a `gen-sequence` constraint can be thought of as the corresponding binary sequence or, equivalently, as the y array for it. This y array representation has a useful property. 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 1. *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 min constraint, i.e., $y^*[R] - y^*[L-1] < \min(s)$. 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 $y[R] - y[L-1] \leq y[R] - y'[L-1] = y^*[R] - y^*[L-1] < \min(s)$. In other words, y itself violates s, a contradiction. A similar reasoning works when y^* violates the max 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 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}.

Lemma 2. *The procedure* CheckConsistency *constructs the minimum solution to the* `gen-sequence` *constraint or proves that none exists, in $O(n^2)$ 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 when CheckConsistency does report success, its 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 min constraint was violated, i.e., $y[R] - y[L-1] < \min(s)$, and index $L-1$ was pushed up so that the new value of $y[L-1]$, denoted $\hat{y}[L-1]$, became $y[R] - \min(s)$. Since this was the first time a y value exceeded y_{min}, we have $y[R] \leq y_{min}[R]$, so that $\hat{y}[L-1] \leq y_{min}[R] - \min(s) \leq y_{min}[L-1]$. It follows that $\hat{y}[L-1]$ itself does not exceed $y_{min}[L-1]$. It may still be 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 $\hat{y}[L-1] \leq y_{min}[L-1]$, all repaired y values must also not go above y_{min}. A similar argument works when instead the max 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}. 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 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, simple book-keeping can be used to locate a violated constraint in constant time. This proves the desired bound of $O(n^2)$ overall. □

The simple loop structure of CompleteFiltering immediately implies:

Theorem 1. *Algorithm* CompleteFiltering *establishes domain consistency on the* `gen-sequence` *constraint or proves that it is inconsistent, in* $O(n^3)$ *time.*

Remark 1. Régin proved that finding a solution to an arbitrary combination of `among` constraints is NP-complete [9]. Our algorithm finds a solution in polynomial time to a more restricted problem, namely, when each `among` constraint is defined on a sequence of consecutive variables with respect to a fixed ordering.

7 Experimental Results

To evaluate the different filtering algorithms presented, we used two 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. Then we conduct a limited series of experiments on the well-known car sequencing problem. 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

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

		IB		IE		SLG		GS		REG	
q	Δ	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
5	1	–	–	33976.9	18.210	0.2	0.069	0	0.014	0	0.009
6	2	361770	54.004	19058.3	6.390	0	0.078	0	0.013	0	0.018
7	1	380775	54.702	113166	48.052	0	0.101	0	0.012	0	0.020
7	2	264905	54.423	7031	4.097	0	0.129	0	0.016	0	0.039
7	3	286602	48.012	0	0.543	0	0.129	0	0.015	0	0.033
9	1	–	–	60780.5	42.128	0.1	0.163	0	0.010	0	0.059
9	3	195391	43.024	0	0.652	0	0.225	0	0.016	0	0.187

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

		IB		IE		SLG		GS		REG	
q	n	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
5	50	459154	18.002	22812	18.019	0.4	0.007	0	0.001	0	0.001
5	100	192437	12.008	11823	12.189	1	0.041	0	0.005	0	0.005
5	500	48480	12.249	793	41.578	0.7	1.105	0	0.466	0	0.023
5	1000	942	1.111	2.3	160.000	1.1	5.736	0	4.374	0	0.062
7	50	210107	12.021	67723	12.309	0.2	0.015	0	0.001	0	0.006
7	100	221378	18.030	44963	19.093	0.4	0.059	0	0.005	0	0.010
7	500	80179	21.134	624	48.643	2.8	2.115	0	0.499	0	0.082
7	1000	30428	28.270	46	138.662	588.5	14.336	0	3.323	0	0.167
9	50	18113	1.145	18113	8.214	0.9	0.032	0	0.001	0	0.035
9	100	3167	0.306	2040	10.952	1.6	0.174	0	0.007	0	0.087
9	500	48943	18.447	863	65.769	2.2	4.311	0	0.485	0	0.500
9	1000	16579	19.819	19	168.624	21.9	16.425	0	3.344	0	0.843

extended (IE) propagation modes. Experiments were run with Ilog Solver 6.2 on a bi-processor Intel Xeon HT 2.8Ghz, 3G 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 first series of instances is 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 min and max, $\Delta = \max - \min$. 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 min. 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.

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. The results confirm that the new algorithms are very efficient. The average number of backtracks for SLG is generally very low. As predicted by its time complexity, GS is very stable for fixed n in the 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

Table 3. Comparison on small car sequencing instances

Version	Average		Median	
	BT	CPU	BT	CPU
A	1067	26.5	0	4.6
B	1067	10.3	0	3.8
C	802	8.4	0	4.1
D	798	34.3	0	7.0

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 this constraint in a more realistic setting, we turned to the car sequencing problem. We ran experiments using the first set of instances on the CSPLib web site and out of the 78 instances we kept the 31 that could be solved within 5 minutes using a program found in the Ilog distribution. Recall that the Ilog version of the `sequence` constraint also allows to specify individual cardinalities for values in S so it is richer than our version of `sequence` . Table 3 compares the following versions of the sequencing constraint: **(A)** original Ilog program; **(B)** A + REG (added as a redundant constraint); **(C)** A + REG with cost [5], using the cost variable to restrict the total number of cars with a particular option; **(D)** A + REG with cost, using the cost variable to restrict the total number of cars of a particular configuration for each option. For A,B and C we thus introduce one constraint per option and for D we add one constraint per configuration and option.

It is interesting to see that adding REG as a redundant constraint significantly improves performance as it probably often detects a dead end before IloSequence does, thus avoiding expensive work. The simple cost version (C) does quite well since it also incorporates a weak form of cardinality constraint within the `sequence` constraint. For a fairer comparison, we chose not to compare our two other algorithms as we do not currently have incremental implementations.

8 Discussion

We have proposed, analyzed, and evaluated experimentally three 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 for small q, not uncommon in applications, it performs very well partly due to its incremental algorithm. The generalized sequence approach of

Section 6 also establishes domain consistency on the `sequence` constraint, as well as on a more general variant defined on arbitrary `among` constraints. It has an asymptotic time complexity that is polynomial in both n and q, namely $O(n^3)$. 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.

Since q plays an important role in the efficiency of some of the approaches proposed, it is worth estimating it in some typical applications. For example, in car sequencing values between 2 and 5 are frequent, whereas the shift construction problem may feature widths of about 12.

As a possible extension of this work, our first two algorithms lend themselves to a generalization of `sequence` 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, as well as the referees for their comments.

References

1. K.R. Apt. *Principles of Constraint Programming*. Cambridge Univ. Press, 2003.
2. N. Beldiceanu and M. Carlsson. Revisiting the Cardinality Operator and Introducing the Cardinality-Path Constraint Family. In *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. S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*. under final review.
6. R. Mohr and G. Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI)*, pages 651–656, 1988.
7. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP 2004*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.
8. J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *AAAI/IAAI*, pages 209–215. AAAI Press/The MIT Press, 1996.
9. J.-C. Régin. Combination of Among and Cardinality Constraints. In *CPAIOR 2005*, volume 3524 of *LNCS*, pages 288–303. Springer, 2005.
10. J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *CP'97*, volume 1330 of *LNCS*, pages 32–46. Springer, 1997.
11. M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.