An Optimal Arc Consistency Algorithm for a Particular Case of Sequence Constraint

Mohamed Siala, Emmanuel Hébrard, Marie-José Huguet

An Optimal Arc Consistency Algorithm for a Particular Case of Sequence Constraint

Mohamed Siala[1,2], Emmanuel Hebrard[1,3], and Marie-José Huguet[1,2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
[3] Univ de Toulouse, LAAS, F-31400 Toulouse, France
{siala, hebrard, huguet}@laas.fr

Abstract. The AtMostSeqCard constraint is the conjunction of a cardinality constraint on a sequence of n variables and of $n - q + 1$ constraints AtMost u on each subsequence of size q.
This constraint is useful in car-sequencing and crew-rostering problems. In [21], two algorithms designed for the AmongSeq constraint were adapted to this constraint with an $O(2^q n)$ and $O(n^3)$ worst case time complexity, respectively. In [10], another algorithm similarly adaptable to filter the AtMostSeqCard constraint with a time complexity of $O(n^2)$ was proposed.
In this paper, we introduce an algorithm for achieving arc consistency on the AtMostSeqCard constraint with an $O(n)$ (hence optimal) worst case time complexity. Next, we show that this algorithm can be easily modified to achieve arc consistency on some extensions of this constraint. In particular, the conjunction of a set of m AtMostSeqCard constraints sharing the same scope can be filtered in $O(nm)$. We then empirically study the efficiency of our propagator on instances of the car-sequencing and crew-rostering problems.

1 Introduction

In many applications, there are restrictions on the successions of decisions that can be taken. Some sequences are allowed or preferred while other are forbidden. For instance, in crew-rostering applications, it is often not recommended to have an employee work on an evening or a night shift, and then again on the morning shift of the next day.

Several constraints have been proposed to deal with this type of problems. The Regular [14] and cost-Regular constraints [6] make it possible to restrict sequences in an arbitrary way. However, there might often exist a more efficient algorithm for the particular case at hand. For instance, filtering algorithms have been proposed for the AmongSeq constraint in [5, 10, 21, 22]. This constraint ensures that all subsequences of size q have at least l but no more than u values in a set v. This constraint is often applied to car-sequencing and crew-rostering problems. However, the constraints in these two benchmarks do not correspond exactly to this definition. Indeed, there are often no lower bound restriction on the number of values ($l = 0$). Instead, the number of values in the set v is often constrained by an overall demand.

In this paper, we consider the constraint AtMostSeqCard. This constraint, posted on n variables $x_1, \ldots, x_n$, ensures that, in every subsequence of length q, no more than

u are set to a value in a set v, and that over all the sequence, exactly d are set to values in v. In car-sequencing, this constraint allows to state that, given an option, no subsequence of length q can involve more than u classes of cars requiring this option, and that exactly d cars require it overall. In crew-rostering problems, one can state, for instance, that a worker must have at least a 16h break between two 8h shifts, and at least two days off for every period of seven days, while enforcing a total number of worked hours over the scheduling period.

The rest of the paper is organized as follows: In Section 2 we give a brief background on Constraint Programming and sequence constraints. Then in Section 3, we give a linear time (hence optimal) algorithm for filtering the AtMostSeqCard constraint. Next, in Section 4, we show how to adapt the same pruning on more general constraints. Last, in Section 5, we evaluate our new propagators on car-sequencing and crew-rostering benchmarks, before concluding in Section 6.

2 CSP and Sequence Constraints

A constraint satisfaction problem (CSP) is a triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints that specify allowed combinations of values for subsets of variables. We denote by $\min(x)$ and $\max(x)$ the minimum and maximum values in $\mathcal{D}(x)$, respectively. An assignment of a set of variables $\mathcal{X}$ is a tuple w, where $w[i]$ denotes the value assigned to the variable x_i. A constraint $C \in \mathcal{C}$ on a set of variables $\mathcal{X}$ defines a relation on the domains of variables in $\mathcal{X}$. An assignment w is consistent for a constraint C iff it belongs to the corresponding relation. A constraint C is *arc consistent* (AC) iff, for every value j of every variable x_i in its scope, there exists a consistent assignment w such that $w[i] = j$, i.e., a *support*. There are several variants of the Sequence constraints, we first review them and then motivate the need for the variant proposed in this paper: the AtMostSeqCard constraint. In the following definitions, v is a set of integers and l, u, q are integers. Sequence constraints are conjunctions of Among constraints, constraining the number of occurrences of a set of values in a set of variables.

Definition 1. $\text{Among}(l, u, [x_1, \ldots, x_q], v) \Leftrightarrow l \leq |\{i \mid x_i \in v\}| \leq u$

Chains of Among Constraints: The AmongSeq constraint, first introduced in [2], is a chain of Among constraints of width q slid along a vector of n variables.

Definition 2. $\text{AmongSeq}(l, u, q, [x_1, \ldots, x_n], v) \Leftrightarrow \bigwedge_{i=0}^{n-q} \text{Among}(l, u, [x_{i+1}, \ldots, x_{i+q}], v)$

The first (incomplete) algorithm for filtering this constraint was proposed in 2001 [1]. Then, in [21, 22], two complete algorithms for filtering the AmongSeq constraint were introduced. First, a reformulation using the Regular constraint using 2^{q-1} states and hence achieving AC in $O(2^q n)$ time. Second, an algorithm achieving AC with a worst case time complexity of $O(n^3)$. Moreover, this last algorithm is able to handle arbitrary sets of Among constraints on consecutive variables (denoted Gen-Sequence), however in $O(n^4)$. Last, two flow-based algorithms were introduced in [10]. The first achieves AC on AmongSeq in $O(n^{3/2} \log n \log u)$, while the second achieves AC on Gen-Sequence in $O(n^3)$ in the worst case. These two algorithms have an amortized complexity down a branch of the search tree of $O(n^2)$ and $O(n^3)$, respectively.

Chain of ATMOST **Constraints:** Although useful in both applications, the AMONGSEQ constraint does not model exactly the type of sequences useful in car-sequencing and crew-rostering applications.

First, there is often no lower bound for the cardinality of the subsequences, i.e., $l = 0$. Therefore AMONGSEQ is unnecessarily general in that respect. Moreover, the capacity constraint on subsequences is often paired with a cardinality requirement.

For instance, in car-sequencing, classes of car requiring a given option cannot be clustered together, because a working station can only handle a fraction of the cars passing on the line (*at most* u times in any sequence of length q). The total number of occurrences of these classes is a requirement, and translates as an overall cardinality constraint rather than lower bounds on each subsequence.

In crew-rostering, allowed shift patterns can be complex, hence the REGULAR constraint is often used to model them. However, working in at most u shifts out of q is a useful particular case. If days are divided into three 8h shifts, ATMOSTSEQ with $u = 1$ and $q = 3$ makes sure that no employee work more than one shift per day *and* that there must be a 24h break when changing shifts. Moreover, similarly to car-sequencing, the lower bound on the number of worked shifts is global (monthly, for instance).

In other words, we often have a chain of ATMOST constraints, defined as follows:

Definition 3. $\text{ATMOST}(u, [x_1, \ldots, x_q], v) \Leftrightarrow \text{AMONG}(0, u, [x_1, \ldots, x_q], v)$

Definition 4. $\text{ATMOSTSEQ}(u, q, [x_1, \ldots, x_n], v) \Leftrightarrow \bigwedge_{i=0}^{n-q} \text{ATMOST}(u, [x_{i+1}, \ldots, x_{i+q}], v)$

However, it is easy to maintain AC on this constraint. Indeed, the ATMOST constraint is monotone, i.e., the set of supports for value 0 is a super-set of the set of supports for value 1. Hence an ATMOSTSEQ constraint is AC iff each ATMOST is AC [4].

A good tradeoff between filtering power and complexity can be achieved by reasoning about the total number of occurrences of values from the set v together with the chain of ATMOST constraints.[1] We therefore introduce the ATMOSTSEQCARD constraint, defined as the conjunction of an ATMOSTSEQ with a cardinality constraint on the total number of occurrences of values in v:

Definition 5. $\text{ATMOSTSEQCARD}(u, q, d, [x_1, \ldots, x_n], v) \Leftrightarrow$

$$\text{ATMOSTSEQ}(u, q, [x_1, \ldots, x_n], v) \wedge |\{i \mid x_i \in v\}| = d$$

The two AC algorithms introduced in [22] were adapted in [21] to achieve AC on the ATMOSTSEQCARD constraint. First, in the same way that AMONGSEQ can be encoded with a REGULAR constraint, ATMOSTSEQCARD can be encoded with a COST-REGULAR constraint, where the cost stands for the overall demand, and it is increased on transitions labeled with the value 1. This procedure has the same worst case time complexity, i.e., $O(2^q n)$. Second, the more general version of the polynomial algorithm (GEN-SEQUENCE) is used, to filter the following decomposition of the ATMOSTSEQCARD constraint into a conjunction of AMONG:

$$\text{ATMOSTSEQCARD}(u, q, d, [x_1, \ldots, x_n], v) \Leftrightarrow$$
$$\bigwedge_{i=0}^{n-q} \text{AMONG}(0, u, [x_{i+1}, \ldots, x_{i+q}], v) \ \wedge \ \text{AMONG}(d, d, [x_1, \ldots, x_n], v)$$

[1] This modeling choice is used in [21] on car-sequencing.

Since the number of AMONG constraints is linear, the algorithm of van Hoeve et al [21]. runs in $O(n^3)$ on this decomposition. Similarly, the algorithm of Maher et al. [10] runs in $O(n^2)$ on ATMOSTSEQCARD, which is the best known complexity for this problem.

Global Sequencing Constraint: Finally, in the particular case of car-sequencing, not only do we have an overall cardinality for the values in v, but each value corresponds to a class of car and has a required number of occurrences. Therefore, Puget and Régin [17] proposed to consider the conjunction of an AMONGSEQ and a GCC. Let c_l and c_u be two mapping on integers such that $c_l(j) \leq c_u(j)\ \forall j$, and let $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}(x_i)$. The Global Cardinality Constraint (GCC) is defined as follows:

Definition 6. $\text{GCC}(c_l, c_u, [x_1, \ldots, x_n]) \Leftrightarrow \bigwedge_{j \in \mathcal{D}} c_l(j) \leq |\{i \mid x_i = j\}| \leq c_u(j)$

Then, the Global Sequencing Constraint is defined as follows:

Definition 7. $\text{GSC}(l, u, q, c_l, c_u, [x_1, \ldots, x_n], v) \Leftrightarrow$

$$\text{AMONGSEQ}(l, u, q, [x_1, .., x_n], v) \wedge \text{GCC}(c_l, c_u, [x_1, .., x_n])$$

The mappings c_l and c_u are defined so that for a value v, both $c_l(v)$ and $c_u(v)$ map to the number of occurrences of the corresponding class of car. A reformulation of this constraint into a set of GCC was introduced in [17]. However, achieving AC on this constraint is NP-hard [3].

3 The ATMOSTSEQCARD Constraint

In this section, we introduce a linear filtering algorithm for the ATMOSTSEQCARD constraint. We first give a simple greedy algorithm for finding a support with an $O(nq)$ time complexity. Then, we show that one can use two calls to this procedure to enforce AC. Last, we show that its worst case time complexity can be reduced to $O(n)$.

It was observed in [21] and [10] that we can consider Boolean variables and $v = \{1\}$, since the following decomposition of AMONG (or ATMOST) does not hinder propagation as it is Berge-acyclic [5]:

$$\text{AMONG}(l, u, [x_1, \ldots, x_q], v) \Leftrightarrow \bigwedge_{i=1}^{q} (x_i \in v \leftrightarrow x'_i = 1) \wedge l \leq \sum_{i=1}^{q} x'_i \leq u$$

Therefore, throughout the paper, we shall consider the following restriction of the ATMOSTSEQCARD constraint, defined on Boolean variables, and with $v = \{1\}$:

Definition 8.

$$\text{ATMOSTSEQCARD}(u, q, d, [x_1, \ldots, x_n]) \Leftrightarrow \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u) \wedge (\sum_{i=1}^{n} x_i = d)$$

3.1 Finding a Support

Let w be an n-tuple in $\{0,1\}^n$, $w[i]$ denotes the i^{th} element of w, $|w| = \sum_{i=1}^{n} w[i]$ its cardinality, and $w[i:j]$ the $(|j-i|+1)$-tuple equal to w on the subsequence $[x_i, \ldots, x_j]$.

We first show that one can find a support by greedily assigning variables in a lexicographical order to the value 1 whenever possible, that is, while taking care of not violating the AtMostSeq constraint. More precisely, doing so leads to an assignment of maximal cardinality, which may easily be transformed into an assignment of cardinality d.

The greedy procedure `leftmost` (Algorithm 1) computes an assignment w that maximizes the cardinality of the sequence $(x_1, \ldots, x_n)$ subject to an AtMostSeq constraint (with parameters u and q),

Algorithm 1: `leftmost`

```
1  foreach i ∈ [1,...,n] do w[i] ← min(x_i);
   ;
   foreach i ∈ [1,...,q] do w[n+i] ← 0;
   ;
   c(1) ← w[1];
   foreach j ∈ [2,...,q] do c(j) ← c(j−1) + w[j];
   ;
   foreach i ∈ [1,...,n] do
2      if |D(x_i)| > 1 & max_{j∈[1,q]}(c(j)) < u then
           w[i] ← 1;
3          foreach j ∈ [1,...,q] do c(j) ← c(j) + 1;
           ;
4      foreach j ∈ [2,...,q] do c(j−1) ← c(j);
       ;
5      c(q) ← c(q−1) + w[i+q] − w[i];
   return w;
```

Algorithm `leftmost` works as follows.

First, the tuple w is initialized to the minimum value in the domain of each variable in Line 1. Then, at each step $i \in [1, \ldots, n]$ of the main loop, the cardinality of the j^{th} subsequence involving the variable x_i with respect to the current value of w is stored in $c(j)$. In other words, at step i, we have $c(j) = \sum_{l=\max(1,i-q+j)}^{\min(n,i+j-1)} w[l]$.

When exploring variable x_i, such that $\mathcal{D}(x_i) = \{0,1\}$ we set $w[i]$ to 1 iff this would not violate the capacity constraints, that is, if $c(j) < u$ for all $j \in [1, \ldots, q]$ (Condition Line 2). In that case, the cardinality of every subsequence involving x_i is incremented (Line 3). Finally, when moving to the next variable, the values of $c(j)$ are shifted (Line 4), and the value of $c(q)$ is obtained by adding the value of $w[i+q]$ and subtracting $w[i]$ to its previous value (Line 5).

$\mathcal{D}(x_i)$	. 0 . 1 . . . 0 . 0 1 . . 1 1
$\overrightarrow{w}[i]$	**1** 0 0 1 **1** 0 0 0 **1** 0 1 0 0 1 **1** 0 0 **1** 1 0 0 1
$c(1)$	**0** 1 1 2 **1** 2 2 1 **0** 0 2 2 1 2 **1** 2 2 **1** 1 2 2 2
$c(2)$	**0** 1 2 1 **1** 2 1 0 **0** 2 2 1 2 1 **1** 2 1 **0** **1** 2 2 1
$c(3)$	**0** 2 1 1 **1** 1 0 0 **1** 2 1 2 1 1 **1** 1 0 **0** **1** 2 1 1
$c(4)$	**1** 1 1 1 **0** 0 0 1 **1** 1 2 1 1 1 **0** 0 0 **1** 1 1 1 1
$\max(c)$	**1** 2 2 2 **1** 2 2 1 **1** 2 2 2 2 2 **1** 2 2 **1** 1 2 2 2

Fig. 1: Example of the execution of `leftmost` for ATMOSTSEQ$(2, 4, [x_1, \ldots, x_n])$ with maximal cardinality.

From now on, we shall use the following notations:

- $\overrightarrow{w}$ shall denote the assignment found by `leftmost` on the sequence $x_1, \ldots, x_n$.
- $\overleftarrow{w}$ shall denote the assignment found by the same algorithm, however on the sequence $x_n, \ldots, x_1$, that is, from right to left. Notice that, in order to simplify the notations, $\overleftarrow{w}[i]$ shall denote the value assigned by `leftmost` to the variable x_i, and not x_{n-i+1} as it would actually be if we gave the reversed sequence as input.

Example 1. We illustrate the behavior of `leftmost`on a simple example (see Figure 1). Let $[x_1, \ldots, x_{22}]$ be a sequence of variables with a capacity constraint of $2/4$, that is, constrained by: ATMOSTSEQ$(2, 4, [x_1, \ldots, x_{22}])$. Dots in the first row stand for unassigned variables. The second row shows the computed assignment $\overrightarrow{w}$, and the next rows show the state of the variables $c(1), c(2), c(3)$ and $c(4)$ at the start of each iteration of the main loop. The last row stands for the maximum value of $c(j)$. The bold values indicate that `leftmost` assigns the value 1.

Lemma 1. *`leftmost` maximizes* $\sum_{i=1}^{n} x_i$ *subject to* ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$.

Proof. Let $\overrightarrow{w}$ be the assignment found by `leftmost`, and suppose that there exists another assignment w (consistent for ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$) such that $|w| > |\overrightarrow{w}|$. Let i be the smallest index such that $\overrightarrow{w}[i] \neq w[i]$. By definition of $\overrightarrow{w}$, we know that $\overrightarrow{w}[i] = 1$ hence $w[i] = 0$. Now, let j be the smallest index such that $\overrightarrow{w}[j] < w[j]$ (it must exists since $|w| > |\overrightarrow{w}|$).

We first argue that the assignment w' equal to w except that $w'[i] = 1$ and $w'[j] = 0$ (as in $\overrightarrow{w}$) is consistent for ATMOSTSEQ. Clearly, its cardinality is not affected by this swap, hence $|w'| = |w|$. Now, we consider all the sum constraints whose scopes are changed by this swap, that is, the sums $\sum_{l=a}^{a+q-1} w'[l]$ on intervals $[a, a+q-1]$ such that $a \leq i < a+q$ or $a \leq j < a+q$. There are three cases:

1. Suppose first that $a \leq i < j < a+q$: in this case, the value of the sum is the same in w and w', therefore it is lower than or equal to u.
2. Suppose now that $i < a \leq j < a+q$: in this case, the value of the sum is decreased by 1 from w to w', therefore it is lower than or equal to u.
3. Last, suppose that $a \leq i < a+q \leq j$: in this case, for any $l \in [a, a+q-1]$, we have $w'[l] \leq \overrightarrow{w}[l]$ since j is the smallest integer such that $\overrightarrow{w}[j] < w[j]$, hence the sum is lower than or equal to u.

Therefore, given a sequence w of maximum cardinality that differs from $\overrightarrow{w}$ at rank i, we can build a sequence of equal cardinality that does not differ from $\overrightarrow{w}$ until rank $i+1$. By iteratively applying this argument, we can obtain a sequence identical to $\overrightarrow{w}$, albeit with cardinality $|w|$, therefore contradicting our hypothesis that $|w| > |\overrightarrow{w}|$. □

Corollary 1. *Let $\overrightarrow{w}$ be the assignment returned by* `leftmost`*. There exists a solution of* AtMostSeqCard$(u, q, d, [x_1, \ldots, x_n])$ *iff the three following propositions hold:*
(1) AtMostSeq$(u, q, [x_1, \ldots, x_n])$ is satisfiable *(2)* $\sum_{i=1}^{n} \min(x_i) \leq d$
(3) $|\overrightarrow{w}| \geq d$*.*

Proof. It is easy to see that these conditions are all necessary: (1) and (2) come from the definition, and (3) is a direct application of Lemma 1. Now, we prove that they are sufficient by showing that if these properties hold, then a solution exists. Since AtMostSeq$(u, q, [x_1, \ldots, x_n])$ is satisfiable, $\overrightarrow{w}$ does not violate the chain of AtMost constraints as the value 1 is assigned to x_i only if all subsequences involving x_i have cardinality $u-1$ or less. Moreover, since $\sum_{i=1}^{n} \min(x_i) \leq d \leq |\overrightarrow{w}|$, then there are at least $|\overrightarrow{w}| - d$ variables such that $\min(x_i) = 0$ and $\overrightarrow{w}[i] = 1$. Assigning them to 0 in $\overrightarrow{w}$ does not violate the AtMostSeq constraint. Hence we can build a support for AtMostSeqCard. □

Lemma 1 and Corollary 1 give us a polynomial support-seeking procedure for AtMostSeqCard. Indeed, the worst case time complexity of Algorithm 1 is in $O(nq)$. There are n steps and on each step, Lines 2, 3 and 4 involve $O(q)$ operations. Therefore, for each variable x_i, a support for $x_i = 0$ or $x_i = 1$ can be found in $O(nq)$.

Consequently, we have a naive AC procedure running in $O(n^2 q)$ time.

3.2 Filtering the Domains

In this section, we show that we can filter out all the values inconsistent with respect to the AtMostSeqCard constraint within the same time complexity as Algorithm 1.

First, we show that there can be inconsistent values only in the case where the cardinality $|\overrightarrow{w}|$ of the assignment returned by `leftmost` is exactly d: in any other case, the constraint is either violated (when $|\overrightarrow{w}| < d$) or AC, (when $|\overrightarrow{w}| > d$). The following lemma formalizes this:

Lemma 2. *The constraint* AtMostSeqCard$(u, q, d, [x_1, \ldots, x_n])$ *is* AC *if the three following propositions hold:*
(1) AtMostSeq$(u, q, [x_1, \ldots, x_n])$ *is* AC
(2) $\sum_{i=1}^{n} \min(x_i) \leq d$
(3) $|\overrightarrow{w}| > d$

Proof. By Corollary 1 we know that AtMostSeqCard$(u, q, d+1, [x_1, \ldots, x_n])$ is satisfiable. Let w be a satisfying assignment, and consider without loss of generality a variable x_i such that $|\mathcal{D}(x_i)| > 1$. Assume first that $w[i] = 1$. The solution w' equal to w except that $w'[i] = 0$ satisfies AtMostSeqCard$(u, q, d, [x_1, \ldots, x_n])$. Indeed, $|w'| = |w| - 1 = d$ and since AtMostSeq$(u, q, [x_1, \ldots, x_n])$ was satisfied by w it

must be satisfied by w'. Hence, for every variable x_i such that $|\mathcal{D}(x_i)| > 1$, there exists a support for $x_i = 0$.

Suppose that $w[i] = 0$, and let $a < i$ (resp. $b > i$) be the largest (resp. smallest) index such that $w[a] = 1$ and $\mathcal{D}(x_a) = \{0, 1\}$ (resp. $w[b] = 1$ and $\mathcal{D}(x_b) = \{0, 1\}$). Let w' be the assignment such that $w'[i] = 1$, $w'[a] = 0$, $w'[b] = 0$, and $w = w'$ otherwise. We have $|w'| = d$, and we show that it satisfies ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$. Consider a subsequence $x_j, \ldots, x_{j+q-1}$. If $j + q \leq i$ or $j > i$ then $\sum_{l=j}^{j+q-1} w'[l] \leq \sum_{l=j}^{j+q-1} w[l] \leq u$, so only indices j s.t. $j \leq i < j + q$ matter. There are two cases:

1. Either a or b or both are in the subsequence ($j \leq a < j + q$ or $j \leq b < j + q$). In that case $\sum_{l=j}^{j+q-1} w'[l] \leq \sum_{l=j}^{j+q-1} w[l]$.
2. Neither a nor b are in the subsequence ($a < j$ and $j + q \leq b$). In that case, since $\mathcal{D}(x_i) = \{0, 1\}$ and since ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$ is AC, we know that $\sum_{l=j}^{j+q-1} \min(x_l) < u$. Moreover, since $a < j$ and $j + q \leq b$, there is no variable x_l in that subsequence such that $w[l] = 1$ and $0 \in \mathcal{D}(x_l)$. Therefore, we have $\sum_{l=j}^{j+q-1} w[l] < u$, hence $\sum_{l=j}^{j+q-1} w'[l] \leq u$.

In both cases w' satisfies all capacity constraints. Hence it is support for the value 1. □

Remember that achieving AC on ATMOSTSEQ is trivial since AMONG is monotone. Therefore we focus of the case where ATMOSTSEQ is AC, and $|\overrightarrow{w}| = d$. In particular, Lemmas 3, 4, 6 and 7 only apply in that case. The equality $|\overrightarrow{w}| = d$ is therefore implicitly assumed in all of them.

Lemma 3. *If* $|\overrightarrow{w}[1 : i - 1]| + |\overleftarrow{w}[i + 1 : n]| < d$ *then* $x_i = 0$ *is not* AC.

Proof. Suppose that we have $|\overrightarrow{w}[1 : i - 1]| + |\overleftarrow{w}[i + 1 : n]| < d$ and suppose that there exists a consistent assignment w such that $w[i] = 0$ and $|w| = d$.

By Lemma 1 on the sequence $x_1, \ldots, x_{i-1}$ we know that $\sum_{l=1}^{i-1} w[l] \leq |\overrightarrow{w}[1 : i - 1]|$.

By Lemma 1 on the sequence $x_n, \ldots, x_{i+1}$ we know that $\sum_{l=i+1}^{n} w[l] \leq |\overleftarrow{w}[i + 1 : n]|$.

Therefore, since $w[i] = 0$, we have $|w| = \sum_{l=1}^{n} w[l] < d$, thus contradicting the hypothesis that $|w| = d$. Hence, there is no support for $x_i = 0$. □

Lemma 4. *If* $|\overrightarrow{w}[1 : i]| + |\overleftarrow{w}[i : n]| \leq d$ *then* $x_i = 1$ *is not* AC.

Proof. Suppose that we have $|\overrightarrow{w}[1 : i]| + |\overleftarrow{w}[i : n]| \leq d$ and suppose that there exists a consistent assignment w' such that $w'[i] = 1$ and $|w'| = d$.

By Lemma 1 on the sequence $x_1, \ldots, x_i$ we know that $\sum_{l=1}^{i} w'[l] \leq |\overrightarrow{w}[1 : i]|$.

By Lemma 1 on the sequence $x_n, \ldots, x_i$ we know that $\sum_{l=i}^{n} w'[l] \leq |\overleftarrow{w}[i : n]|$.

Therefore, since $w'[i] = 1$, we have $|w'| = \sum_{l=1}^{i} w'[l] + \sum_{l=i}^{n} w'[l] - 1 < d$, thus contradicting the hypothesis that $|w'| = d$. Hence there is no support for $x_i = 1$. □

Lemmas 3 and 4 entail a pruning rule. In a first pass, from left to right, one can use an algorithm similar to `leftmost` to compute and store the values of $|\overrightarrow{w}[1 : i]|$ for all $i \in [1, \ldots, n]$. In a second pass, the values of $|\overleftarrow{w}[i : n]|$ for all $i \in [1, \ldots, n]$ are similarly computed by simply running the same procedure on the same sequence of variables, however *reversed*, i.e., from right to left. Using these values, one can then

apply Lemma 3 and Lemma 4 to filter out the value 0 and 1, respectively. We detail this procedure in the next section.

We first show that these two rules are complete, that is, if AtMostSeq is AC, and the overall cardinality constraint is AC then an assignment $x_i = 0$ (resp. $x_i = 1$) is inconsistent iff Lemma 3 (resp. Lemma 4) applies. The following Lemma shows that the greedy rule maximizes the density of 1s on any subsequence starting on x_1, and therefore minimizes it on any subsequence finishing on x_n. Let $\texttt{leftmost}(k)$ denote the algorithm corresponding to applying `leftmost`, however stopping whenever the cardinality of the assignment reaches a given value k.

Lemma 5. *Let w be a satisfying assignment of* AtMostSeq$(u, q, [x_1, \ldots, x_n])$. *If $k \leq |w|$ then the assignment $\overrightarrow{w}_k$ computed by $\texttt{leftmost}(k)$ is such that, for any $1 \leq i \leq n$: $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$.*

Proof. Let m be the index at which $\texttt{leftmost}(k)$ stops. We distinguish two cases. If $i > m$, for any value l in $[m+1, \ldots, n]$, $\overrightarrow{w}_k[l] \leq w[l]$ (since $\overrightarrow{w}_k[l] = \min(x_l)$), hence $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. When $i \leq m$, clearly for $i = 1$, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$ since $|\overrightarrow{w}_k| \leq |w|$. Now consider the case of $i \neq 1$. Since $|\overrightarrow{w}_k| \leq |w|$, then $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq |w| - \sum_{l=1}^{i-1} \overrightarrow{w}_k[l]$. Thus, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l] + (\sum_{l=1}^{i-1} w[l] - \sum_{l=1}^{i-1} \overrightarrow{w}_k[l])$. Moreover,by applying Lemma 1, we show that $\sum_{l=1}^{i-1} \overrightarrow{w}_k[l]$ is maximum, hence larger than or equal to $\sum_{l=1}^{i-1} w[l]$. Therefore, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. □

Lemma 6. *If* AtMostSeq$(u, q, [x_1, \ldots, x_n])$ *is* AC, *and $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| \geq d$ then $x_i = 0$ has a support.*

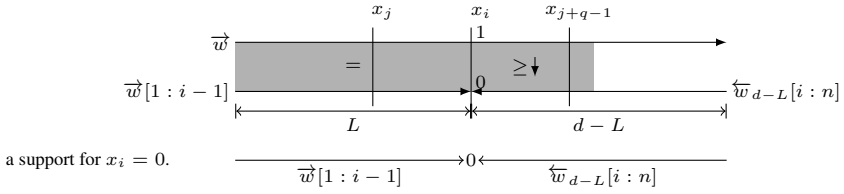

Fig. 2: Illustration of Lemma 6's proof. Horizontal arrows represent assignments.

Proof. Let $\overrightarrow{w}$ be the assignment found by `leftmost`. We consider, without loss of generality, a variable x_i such that $\mathcal{D}(x_i) = \{0, 1\}$ and $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| \geq d$, and show that one can build a support for $x_i = 0$. If $\overrightarrow{w}[i] = 0$ or $\overleftarrow{w}[i] = 0$ then there exists a support for $x_i = 0$, hence we only need to consider the case where $\overrightarrow{w}[i] = \overleftarrow{w}[i] = 1$.

Let $L = |\overrightarrow{w}[1:i-1]|$ and $\overleftarrow{w}_{d-L}$ be the result of $\texttt{leftmost}(d-L)$ on the subsequence $x_n, \ldots, x_i$. We will show that w, defined as the concatenation of $\overrightarrow{w}[1:i-1]$ and $\overleftarrow{w}_{d-L}[i:n]$ is a support for $x_i = 0$.

First, we show that $w[i] = 0$, that is $\overleftarrow{w}_{d-L}[i] = 0$. By hypothesis, since $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| \geq d$, we have $|\overleftarrow{w}[i+1:n]| \geq d - L$. Now, consider the sequence $x_i, \ldots, x_n$, and let w' be the assignment such that $w'[i] = 0$, and $w' = \overleftarrow{w}[i+1:n]$ otherwise. Since $|w'| = |\overleftarrow{w}[i+1:n]| \geq d - L$, by Lemma 5, we know that w' has a higher cardinality than $\overleftarrow{w}_{d-L}$ on any subsequence starting in i, hence $w[i] = \overleftarrow{w}_{d-L}[i] = w'[i] = 0$.

Now, we show that w does not violate the ATMOSTSEQ constraint. Obviously, since it is the concatenation of two consistent assignments, it can violate the constraint only on the junction, i.e., on a subsequence $x_j, \ldots, x_{j+q-1}$ such that $j \leq i$ and $i < j + q$.

We show that the sum of any such subsequence is less or equal to u by comparing with $\overrightarrow{w}$, as illustrated in Figure 2. We have $\sum_{l=j}^{j+q-1} \overrightarrow{w}[l] \leq u$, and $\sum_{l=j}^{i-1} \overrightarrow{w}[l] = \sum_{l=j}^{i-1} w[l]$. Moreover, by Lemma 5, since $|\overrightarrow{w}[i:n]| = |\overleftarrow{w}_{d-L}| = d - L$ we have $\sum_{l=i}^{j+q-1} \overleftarrow{w}_{d-L}[l] \leq \sum_{l=i}^{j+q-1} \overrightarrow{w}[l]$ hence $\sum_{l=i}^{j+q-1} w[l] \leq \sum_{l=i}^{j+q-1} \overrightarrow{w}[l]$. Therefore, we can conclude that $\sum_{l=j}^{j+q-1} w[l] \leq u$. □

Lemma 7. *If* ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$ *is* AC, $|\overrightarrow{w}[1:i]| + |\overleftarrow{w}[i:n]| > d$ *then* $x_i = 1$ *has a support.*

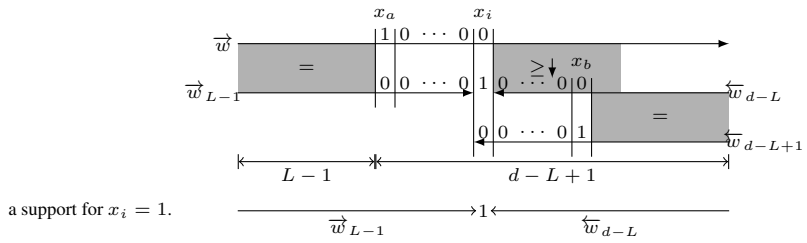

Fig. 3: Illustration of Lemma 7's proof. Horizontal arrows represent assignments.

Proof. Let $\overrightarrow{w}$ and $\overleftarrow{w}$ be the assignments found by `leftmost`, on respectively $x_1, \ldots, x_n$ and $x_n, \ldots, x_1$. We consider, without loss of generality, a variable x_i such that $\mathcal{D}(x_i) = \{0, 1\}$ and $|\overrightarrow{w}[1:i]| + |\overleftarrow{w}[i:n]| > d$, and show that one can build a support for $x_i = 1$. If $\overrightarrow{w}[i] = 1$ or $\overleftarrow{w}[i] = 1$ then there exists a support for $x_i = 1$, hence we only need to consider the case where $\overrightarrow{w}[i] = \overleftarrow{w}[i] = 0$.

Let $L = |\overrightarrow{w}[1:i]| = |\overrightarrow{w}[1:i-1]|$ (this equality holds since $\overrightarrow{w}[i] = 0$). Let $\overrightarrow{w}_{L-1}$ be the assignment obtained by using `leftmost`$(L-1)$ on the subsequence $x_1, \ldots, x_{i-1}$, and let $\overleftarrow{w}_{d-L}$ be the assignment returned by `leftmost`$(d-L)$ on the subsequence $x_n, \ldots, x_{i+1}$.

We show that w such that $w[i] = 1$, equal to $\overrightarrow{w}_{L-1}$ on $x_1, \ldots, x_{i-1}$ and to $\overleftarrow{w}_{d-L}$ on $x_{i+1}, \ldots, x_n$, is a support.

Clearly $|w| = d$, therefore we only have to make sure that all capacity constraints are satisfied. Moreover, since it is the concatenation of two consistent assignments, it

can violate the constraint only on the junction, i.e., on a subsequence $x_j, \ldots, x_{j+q-1}$ such that $j \leq i$ and $i < j + q$.

We show that the sum of any such subsequence is less or equal to u by comparing with $\overrightarrow{w}$ and $\overleftarrow{w}_{d-L}$ (see Figure 3). First, note that on the subsequence $x_1, \ldots, x_{i-1}$, $\overrightarrow{w}_{L-1} = \overrightarrow{w}$, except for the largest index a such that $\overrightarrow{w}[a] = 1$ and $w[a] = 0$. Similarly on $x_n, \ldots, x_{i+1}$, we have $\overleftarrow{w}_{d-L} = \overleftarrow{w}_{d-L+1}$, except for the smallest b such that $\overleftarrow{w}_{d-L+1}[b] = 1$. There are two cases:

Suppose first that $j > a$. In that case, $\sum_{l=j}^{j+q-1} w[l] = \sum_{l=i}^{j+q-1} \overleftarrow{w}_{d-L+1}[l]$ if $j + q - 1 \geq b$, and otherwise it is equal to 1. It is therefore always less than or equal to u since $i \geq j$ (and we assume $u \geq 1$).

Now suppose that $j \leq a$. In that case, consider first the subsequence $x_j, \ldots, x_i$. On this interval, the cardinality of w is the same as that of $\overrightarrow{w}$, i.e., $\sum_{l=j}^{i} w[l] = \sum_{l=j}^{i-1} \overrightarrow{w}_{L-1}[l] + 1 = \sum_{l=j}^{i} \overrightarrow{w}[l]$. On the subsequence $x_{i+1}, \ldots, x_{j+q-1}$, note that $|w[i+1:n]| = |\overrightarrow{w}[i+1:n]| = d - L$, hence by Lemma 5, we have $\sum_{l=i+1}^{j+q-1} w[l] = \sum_{l=i+1}^{j+q-1} \overleftarrow{w}_{d-L}[l] \leq \sum_{l=i+1}^{j+q-1} \overrightarrow{w}[l]$. Therefore $\sum_{l=j}^{j+q-1} w[l] \leq \sum_{l=j}^{j+q-1} \overrightarrow{w}[l] \leq u$. □

3.3 Algorithmic Complexity

Using Lemmas 3, 4, 6 and 7, one can design a filtering algorithm with the same worst case time complexity as `leftmost`. In this section, we introduce a linear time implementation of `leftmost`. We denote this algorithm `leftmost_count`, since we use it to compute an array "*count*" containing the values of $|\overrightarrow{w}[1:i]|$ for all values of i. We give the pseudo code for this procedure in Algorithm 2. The key idea that allows to reduce the complexity is that, at each step, a single new subsequence is to be considered. However, we also need to compute the new maximum across current subsequences, and increment all subsequences when assigning the value 1 to $w[i]$, both in constant time.

It is easy to see that `leftmost_count` has an $O(n)$ worst case time complexity.

In order to prove its correctness, we will show that the assignment computed by `leftmost_count` is the same as that computed by `leftmost`.

Lemma 8. *Algorithms 1 and 2 return the same assignment w.*

Proof (in Appendix). We give the full proof in Appendix.

The idea is to prove the following three invariants, true at the beginning of each step of the main loop:

- The cardinality of the j^{th} subsequence is equal to $c[(i+j-2) \bmod q] + count[i-1]$.
- The number of subsequences of cardinality k is equal to $occ[n - count[i-1] + k]$.
- The cardinality maximum of any subsequence is equal to $\max_c$.

Then, it is easy to check that `leftmost_count` computes the exact same assignment as `leftmost`. Furthermore, at the at the end of the algorithm, we will have $count[i] = |\overrightarrow{w}[1:i]|$ for all $i \in [1, n]$. □

3.4 Achieving Arc-Consistency on ATMOSTSEQCARD

Now, we can prove our main result, that AC on a constraint ATMOSTSEQCARD$(u, q, d, [x_1, \ldots, x_n])$ can be achieved in $O(n)$ time by Algorithm 3.

First, in Line 1, we achieve AC on ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$, so that the first condition for Lemma 2 holds. Achieving AC on ATMOSTSEQ can be done in linear time using a procedure essentially similar to `leftmost_count`. Indeed, since the constraint ATMOST is monotone, we simply need to achieve AC on every ATMOST. Moreover, a constraint ATMOST$(u, [x_{i_1}, \ldots, x_{i_q}])$ may prune the domain of a variable only if u other variables in $[x_{i_1}, \ldots, x_{i_q}])$ are assigned to 1. To do that, we run a truncated version of `leftmost_count`: the values of $w[i]$ are never updated, i.e., they are set to the minimum value in the domain and we never enter the if-then-else block starting at condition 1 in Algorithm 2. Now, if at step i we have $\max_c = u$, then there are u variables assigned to 1 in at least one subsequence involving $vari$, hence it should be set to 0 if possible.

Second, in Line 2, we achieve AC on the cardinality constraint, in order to satisfy the second condition of Lemma 2.

Third, in Line 4 we compute the vector L that maps each index i to the value of $|\overrightarrow{w}[1:i]|$. This is given by the array *count* returned by `leftmost_count` on the sequence $[x_1, \ldots, x_i]$. Notice that, we work with the residual demand, computed in Line 3, rather than the original demand. At this point, the third condition of Lemma 2 can be checked, and we know whether the constraint is AC, inconsistent, or if some pruning may be possible.

In the latter case, we compute the vector R, that maps each index i to the value of $|\overleftarrow{w}[i:n]|$, in Line 5.

Finally, we can activate the pruning rules that are shown to be correct and sufficient by Lemmas 3 and 6 for Line 6, and Lemmas 4 and 7 for Line 7.

Algorithm 3: AC(ATMOSTSEQCARD)

Data: $[x_1, \ldots, x_n], u, q, d$
Result: AC on ATMOSTSEQCARD$(u, q, d, [x_1, .., x_n])$
1 AC(ATMOSTSEQ)$(u, q, [x_1, \ldots, x_n])$;
2 AC$(\sum_{i=1}^{n} x_i = d)$;
3 $d_{res} \leftarrow d - \sum_{i=1}^{n} \min(x_i)$;
4 $L \leftarrow$ `leftmost_count`$([x_1, \ldots, x_n], u, q)$;
if $L[n] = d_{res}$ **then**
5 $R \leftarrow$ `leftmost_count`$([x_n, \ldots, x_1], u, q)$;
foreach $i \in [1, \ldots, n]$ *such that* $\mathcal{D}(x_i) = \{0, 1\}$ **do**
6 **if** $L[i] + R[n - i + 1] \leq d_{res}$ **then** $\mathcal{D}(x_i) \leftarrow \{0\}$;
;
7 **if** $L[i - 1] + R[n - i] < d_{res}$ **then** $\mathcal{D}(x_i) \leftarrow \{1\}$;
;
else if $L[n] < d_{res}$ **then**
return `failure`;
return `consistent`;

Theorem 1. *Algorithm 3 achieves* AC *on* ATMOSTSEQCARD *with an optimal worst case time complexity.*

Proof. The soundness of Algorithm 3 is a straight application of Lemmas 3 and 4. Its completeness is a consequence of Lemmas 2, 6 and 7.

Achieving AC on ATMOSTSEQ (Line 1) can be done with one call to `leftmost_count`. Achieving AC on a simple cardinality constraint (Line 2) can be done trivially in $O(n)$ time. Finally, pruning the domains requires at most two calls to `leftmost_count`, plus going through the sequence of variable to actually change the domains, that is, $O(n)$ time.

The worst case time complexity of Algorithm 3 is then $O(n)$, hence optimal. □

Example 2. We give an example of the execution of Algorithm 3 in Figure 4 for computing the AC of constraint ATMOSTSEQCARD with $u = 4,\ q = 8$ and $d = 12$.

The first line stands for current domains, dots are unassigned variables (hence $d_{res} = 10$). The two next lines give the assignments $\overrightarrow{w}$ and $\overleftarrow{w}$ obtained by running `leftmost_count` from left to right and from right to left, respectively. The third and fourth lines stand for the values of $L[i] = |\overrightarrow{w}[1:i]|$ and $R[n-i+1] = |\overleftarrow{w}[i:n]|$. The fifth and sixth lines correspond to the application of, respectively, Lemma 3 and 4. Last, the seventh line gives the arc consistent domains. Bold values indicate pruning.

4 Extensions

In this section, we show that the filtering algorithm described in the previous section can be extended in a number of ways to enforce AC on more general constraints. Some generalizations are straightforward. For instance, the parameter u does not need to be the same for all subsequences. Indeed neither Algorithm 1 nor Algorithm 2 relies on the fact that u is constant across all subsequences. We can easily give a list of upper bounds, one for each subsequence. Another relatively straightforward generalization is to have a domain variable, rather than a single value, for the demand d.

4.1 The ATMOSTSEQΔCARD constraint

Let δ be a domain variable, we define the ATMOSTSEQΔCARD as follows:

Definition 9.

$$\text{ATMOSTSEQ}\Delta\text{CARD}(u, q, \delta, [x_1, \ldots, x_n]) \Leftrightarrow \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u) \wedge (\sum_{i=1}^{n} x_i = \delta)$$

We show how one can achieve AC on the above generalization. The changes to Algorithm 3 required to handle this generalization are minimal. Indeed, tight lower and upper bounds on δ are easy to compute. They are given, respectively by $\sum_{i=1}^{n} \min(x_i)$, and $|\overrightarrow{w}|$. Moreover, by Lemma 2, we know there can be inconsistent values for a variable x_i only if $|\overrightarrow{w}| \leq d$. It follows that we only need to care about the lower bound of δ. We show these changes in Algorithm 4. The domain of δ is updated in Line 2 for the lower bound, and Line 5 for the upper bound. Also, the lower bound of δ ($\min(\delta)$) is used to compute the residual demand to reach in Line 3 instead of d.

Algorithm 4: AC(ATMOSTSEQΔCARD)

Data: $[x_1, \ldots, x_n], u, q, \delta$
Result: AC on ATMOSTSEQΔCARD$(u, q, \delta, [x_1, .., x_n])$
1 AC(ATMOSTSEQ)$(u, q, [x_1, \ldots, x_n])$;
2 AC$(\sum_{i=1}^{n} x_i = \delta)$;
3 $d_{res} \leftarrow \min(\delta) - \sum_{i=1}^{n} \min(x_i)$;
4 $L \leftarrow$ `leftmost_count`$([x_1, \ldots, x_n], u, q)$;
5 $\mathcal{D}(\delta) \leftarrow \mathcal{D}(\delta) \cap [0, L[n] + \sum_{i=1}^{n} \min(x_i)]$;
6 **if** $L[n] = d_{res}$ **then**
  $R \leftarrow$ `leftmost_count`$([x_n, \ldots, x_1], u, q)$;
  foreach $i \in [1, \ldots, n]$ *such that* $\mathcal{D}(x_i) = \{0, 1\}$ **do**
    if $L[i] + R[n - i + 1] \leq d_{res}$ **then** $\mathcal{D}(x_i) \leftarrow \{0\}$;
    ;
    if $L[i - 1] + R[n - i] < d_{res}$ **then** $\mathcal{D}(x_i) \leftarrow \{1\}$;
    ;
7 **else if** $L[n] < d_{res}$ **then**
  return `failure`;
return `consistent`;

Theorem 2. *Algorithm 4 achieves* AC *on* ATMOSTSEQΔCARD *with an optimal worst case time complexity.*

Proof. First, we need to prune inconsistent values from the domain of δ. By Lemma 1, the cardinality $|\overrightarrow{w}|$ of the assignment returned by `leftmost` is a valid upper bound for δ. Moreover, because of the cardinality constraint, $\sum_{i=1}^{n} \min(x_i)$ is a valid lower bound. It is easy to see that any value d within these bounds satisfies the conditions of Lemma 1. In other words, we can assign δ to any value in the interval $[\sum_{i=1}^{n} \min(x_i), |\overrightarrow{w}|]$ and extend it to an AC support of ATMOSTSEQΔCARD$(u, q, \delta, [x_1, \ldots, x_n])$. These bounds are therefore tight.

Second, we need to prune values in $\mathcal{D}(x_i)$ for all i in $1, \ldots, n$ that are not supported by any value in $\mathcal{D}(\delta)$. A naive algorithm for checking that would be to run `leftmost` for each value in $\mathcal{D}(\delta)$ and compute the union of possible values for the variables x_i. However, one can avoid this by distinguishing two cases after line 5. Suppose that $|\mathcal{D}(\delta)| > 1$, in this case, lines 1, 2 and 5 imply that Lemma 2 holds for $d = \min(\mathcal{D}(\delta))$. Hence all values for the variables x_i are consistent and in this case we will never enter lines 6 and 7. Suppose now that $|\mathcal{D}(\delta)| = 1$, in this case, we can simply apply the same filtering (Line 6) that we proposed previously for a fixed cardinality.

The whole procedure requires at most two calls to `leftmost_count`, which takes $O(n)$ time. □

4.2 The MULTIATMOSTSEQCARD Constraint

Here, we show that we can easily modify Algorithm 3 (or Algorithm 4) to achieve AC on the conjunction of several ATMOSTSEQCARD constraints.

For instance, in crew-rostering problems, the work pattern of an employee might require a conjunction of ATMOSTSEQCARD: one to limit the number of shifts per day,

and another to limit the number of shifts per week. In the crew-rostering benchmarks that we consider in Section 5, we have a variable x_i for each working shift i. Moreover, we want each employee to work at most one shift per day, at most five shifts per week, and between 17 and 18 shifts on the whole period. We model this with two AtMostSeqΔCard constraints: AtMostSeqΔCard$(1, 3, \delta, [x_1, \ldots, x_n])$ and AtMostSeqΔCard$(5, 21, \delta, [x_1, \ldots, x_n])$ s.t. $\mathcal{D}(\delta) = \{17, 18\}$. However, AC on these two constraints is not equivalent to AC on their conjunction. We illustrate this in Example 3 (using smaller instances of the constraints).

Example 3. Consider the conjunction of the two following AtMostSeqCard constraints:

$$\text{AtMostSeqCard}(1, 2, 9, [x_1, \ldots, x_{22}]) \;\&\; \tag{4.1}$$
$$\text{AtMostSeqCard}(2, 5, 9, [x_1, \ldots, x_{22}]) \tag{4.2}$$

Now, suppose that $\mathcal{D}(x_8) = \mathcal{D}(x_{14}) = \mathcal{D}(x_{20}) = \{0\}$, whilst all other domains are equal to $\{0, 1\}$. The first line of Table 1 shows the domains of $[x_1, \ldots, x_{22}]$, with a dot (.) standing for a full domain ($\{0, 1\}$) and the value 0 standing for the domain reduced to the singleton $\{0\}$. The second and third lines show the results of `leftmost` on $[x_1, \ldots, x_{22}]$ for $u/q = 1/2$ and $u/q = 2/5$, respectively. Since the demand d is equal to 9, both constraints 4.1 and 4.2 are AC. Last, the third line shows an assignment of maximum cardinality respecting simultaneously AtMostSeq$(1, 2, [x_1, \ldots, x_{22}])$ and AtMostSeq$(2, 5, [x_1, \ldots, x_{22}])$. It is obtained using the same principle as `leftmost`, however by checking two sets of subsequences, one for each AtMostSeqCard constraint. It is easy to see that the arguments of Lemma 1 are still valid when considering any number of subsequences. Therefore, the total cardinality of 8 is a valid upper bound, and since d is equal to 9, the conjunction of the two constraints has no solution.

Table 1: Maximal cardinality assignments.

x_i:	. . . . . 0 . . . . . 0 . . . . . 0 . .	
$\overrightarrow{w}$ on 4.1:	1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0	$\lvert\overrightarrow{w}\rvert = 11$
$\overrightarrow{w}$ on 4.2:	1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1	$\lvert\overrightarrow{w}\rvert = 10$
$\overrightarrow{w}$ on 4.1 & 4.2:	1 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0	$\lvert\overrightarrow{w}\rvert = 8$

We define the constraint MultiAtMostSeqCard, and show that the algorithm introduced in this paper can be adapted to enforce AC on this constraint in $O(nm)$ time, where m is the number of chains of AtMost constraints.

Definition 10. MultiAtMostSeqCard$(u_1, .., u_m, q_1, .., q_m, d, [x_1, \ldots, x_n]) \Leftrightarrow$

$$\bigwedge_{k=1}^{m} \bigwedge_{i=0}^{n-q_k} \left(\sum_{l=1}^{q_k} x_{i+l} \leq u_k\right) \wedge \left(\sum_{i=1}^{n} x_i = d\right)$$

Theorem 3. *One can achieve* AC *on* MultiAtMostSeqCard *in* $O(nm)$ *time.*

Proof (sketch). The main argument to show that this theorem holds is that all previous proofs and algorithms can be easily lifted to this case. We therefore only sketch its proof.

First, note that one can modify the procedure `leftmost` (or `leftmost_count`) to handle a conjunction of ATMOSTSEQ constraints instead of a single one. All we need to do is to duplicate m times the structures maintaining the cardinalities of the subsequences. We obtain a procedure that checks m chains in $O(nm)$ if we use Algorithm 2.

Second we show that Lemma 1 still holds with this new procedure, and with respect to several chains of ATMOST constraints. In other words, greedily assigning the value "1" while respecting m chains of ATMOST will produce a sequence of maximal cardinality. The argument used in the proof of Lemma 1 generalizes without modification to several chains. We show that if we make the hypothesis that an assignment w of cardinality higher than of $|\overrightarrow{w}|$ found by the greedy procedure leads to a contradiction. For each value of q, the same three cases arise, and can be analyzed in exactly the same way. Hence we can show that w can be made equal to $\overrightarrow{w}$ without changing its cardinality, hence a contradiction.

Indeed, in all subsequent proofs, we check subsequences of length q and show that they do not violate capacity constraints. Obviously, these proofs hold for any value of q (within $[1, n]$). Actually, the only difference is that when considering multiple chains, we might have to check subsequences of different lengths. □

5 Experimental Results

We tested our filtering algorithm on two benchmarks: car-sequencing and crew-rostering. All experiments ran on Intel Xeon CPUs 2.67GHz under Linux. All models are implemented using Ilog-Solver.

Since we compare propagators, we averaged the results across several branching heuristics to reduce the bias that these can have on the outcome. Moreover, these heuristics were randomized and for each instance and each heuristic we launched 5 randomized runs with a 20 minutes time cutoff.[2] For each considered data set, we primarily compare the total number of successful runs, denoted "*#solved*". We say that a run was successful if either a solution could be found or unsatisfiability could be proven. Then, we also consider the CPU time in seconds and number of backtracks, denoted *#backtracks*, both restricted to successful runs. When appropriate, we emphasize the statistics of the best method using bold face fonts.

5.1 Car-sequencing

Problem description. In the car-sequencing problem [7, 20], n vehicles have to be produced on an assembly line, subject to capacity and demand constraints. There are k classes of vehicles and p types of options. Each class c is associated with a demand D_c, that is, the number of occurrences of this class on the assembly line, and a set of options. Each option is handled by a working station able to process only a fraction of

[2] For a total of approximately one year of CPU time.

the vehicles passing on the line. The capacity related to an option j is defined by two integers u_j and q_j, and enforces that no subsequence of size q_j contains more than u_j vehicles requiring option j. We define also for each option j, the corresponding set of classes of vehicles requiring this option $\mathcal{C}_j = \{c \mid j \in \mathcal{O}_c\}$, and the option's demand $d_j = \sum_{c \in \mathcal{C}_j} D_c$.

Models and heuristics. We use a standard CSP model with two sets of variables. The first set corresponds to n integer variables $\{x_1, \ldots, x_n\}$ taking values in $\{1, \ldots, k\}$ and standing for the class of vehicles in each slot of the assembly line. The second set of variables corresponds to np Boolean variables $\{y_1^1, \ldots, y_n^p\}$, where y_i^j stands for whether the vehicle in the i^{th} slot requires option j. Regarding the constraints, first, the *demand* for each class is enforced with a GCC [16]. Second, to ensure the *capacity* constraints, we consider four models:

1. *sum*: we use the default decomposition into a chain of ATMOST constraints.
2. *gsc*: Let $card$ be a mapping on integers such that $card(c) = D_c, \forall c \in \{1, \ldots, k\}$. For each option j, we post the following GSC constraint: GSC$(0, u_j, q_j, card, card, [x_1, \ldots, x_n], \mathcal{C}_j)$
3. *amsc*: For each option j, we post the following ATMOSTSEQCARD constraint: ATMOSTSEQCARD$(u_j, q_j, d_j, [y_1^j, \ldots, y_n^j])$
4. *gsc+amsc*: we combine GSC with ATMOSTSEQCARD for each option.

Last, we *channel* integer and Boolean variables: $\forall j \in \{1, ..., p\}, \forall i \in \{1, ..., n\}$, $y_i^j = 1 \Leftrightarrow x_i \in \mathcal{C}_j$.

We use 42 search heuristics obtained by combining different ways of *exploring* the assembly line either in lexicographic order or from the middle to the sides; of *branching* on affectation of a class to a slot or of an option to a slot; of *selecting* the best class or option among a number of natural criteria (such as maximum demand, minimum u/q ratio, as well as other criteria described in or derived from [17], [18], [19]).

Benchmarks. We use benchmarks available in the CSPLib [8]. The first group of the CSPLib contains 70 satisfiable instances involving 200 cars, it is denoted by *set1*. The second group of the CSPLib corresponds to 4 satisfiable instances with 100 cars, denoted by *set2* and 5 unsatisfiable instances with 100 cars denoted by *set3*. The third group of the CSPLib contains 30 larger instances (ranging from 200 to 400 vehicles). *set4* concerns the 7 instances from this group that are known to be satisfiable. The status of the 23 remaining instances (*set5*) are still unknown. However, these instances are often treated as optimization problems. We therefore consider them separately later.

For each method, we report the average number of solved instances in Table 2, the average CPU time on solved instances in Table 3 and the average number of backtracks in Table 4. In each table, we also report the minimum and maximum value (for any heuristic, though averaged over randomized runs) as well as the standard deviation over the different heuristics. Table 2 shows that in all cases, the best method is either *gsc+amsc* or *amsc*. In some cases a stronger filtering seems to be key and *gsc+amsc* solves more instances than other methods: 95.46% of *set1* and 3.04% of *set3*. In other cases, exploration speed is more important and *amsc* is better: 55.95% and 14.55% of

Table 2: Evaluation of the filtering methods (solved instances count)

propagation	#solved in *set1* (70×5)				#solved in *set2* (4×5)			
	avg	min	max	dev	avg	min	max	dev
sum	268.33	70.00	**350.00**	88.95	2.95	0.00	15.00	3.66
gsc	333.52	154.00	**350.00**	42.16	10.11	0.00	**20.00**	5.25
amsc	321.35	80.00	**350.00**	64.05	**11.19**	0.00	**20.00**	5.22
gsc+amsc	**334.11**	154.00	**350.00**	41.88	10.45	0.00	**20.00**	5.06
propagation	#solved in *set3* (5×5)				#solved in *set4* (7×5)			
	avg	min	max	dev	avg	min	max	dev
sum	0.00	0.00	0.00	0.00	2.35	0.00	9.00	2.65
gsc	0.73	0.00	**10.00**	2.35	4.64	0.00	**10.00**	3.69
amsc	0.38	0.00	5.00	1.21	**5.09**	0.00	**10.00**	3.75
gsc+amsc	**0.76**	0.00	**10.00**	2.41	4.80	0.00	**10.00**	3.65

Table 3: Evaluation of the filtering methods (CPU time on solved instances)

propagation	CPU time (in sec.) on *set1* (70×5)				CPU time (in sec.) on *set2* (4×5)			
	avg	min	max	dev	avg	min	max	dev
sum	10.49	0.02	1145.20	80.39	58.74	0.01	766.25	178.88
gsc	3.16	0.52	1100.54	33.17	109.45	0.11	1096.37	237.46
amsc	3.79	0.03	1197.88	51.49	70.56	0.01	1014.57	186.87
gsc+amsc	3.03	0.53	1017.74	33.60	99.71	0.11	1155.40	222.85
propagation	CPU time (in sec.) on *set3* (5×5)				CPU time (in sec.) on *set4* (7×5)			
	avg	min	max	dev	avg	min	max	dev
sum	-	-	-	-	30.85	0.03	985.75	136.43
gsc	276.06	29.22	988.79	308.64	53.61	1.63	975.03	147.35
amsc	8.62	1.06	18.07	6.72	38.45	0.03	1171.78	124.29
gsc+amsc	285.43	6.01	1131.19	337.24	61.61	1.62	1180.53	175.23

Table 4: Evaluation of the filtering methods (search tree size on solved instances)

propagation	#backtracks on *set1* (70×5)				#backtracks on *set2* (4×5)			
	avg	min	max	dev	avg	min	max	dev
sum	174017	148	25062202	1341281	1101723	78	15324348	3439897
gsc	1408	99	2320312	34519	131062	58	1595137	306448
amsc	33600	92	13888040	468527	665205	61	10254401	1827516
gsc+amsc	1007	92	1180605	23649	104823	56	1055307	244135
propagation	#backtracks on *set3* (5×5)				#backtracks on *set4* (7×5)			
	avg	min	max	dev	avg	min	max	dev
sum	-	-	-	-	378475	170	13767766	1754180
gsc	55365	5852	218590	63211	23897	151	467396	75097
amsc	40326	5991	83454	29690	215349	146	5624744	653498
gsc+amsc	57725	1120	244787	69705	22974	146	428523	71552

solved instances for *set2* and *set4*, respectively. Overall, as witnessed by Table 4, *gsc* and *gsc+amsc* usually require exploring a much smaller tree than *amsc*. However, the propagator for GSC slows down the search by a substantial amount. Considering Table 3 as well as data from unsolved instances, we observed a factor 12.5 on the number of nodes explored per second between these two models. Moreover, the level of filtering obtained by these two methods are incomparable. Therefore combining them is always better than using GSC alone.

In [21] the authors applied their method to *set1*, *set2* and *set3* only. For their experiments, they considered the best result provided by 2 heuristics. When using COST-REGULAR or GEN-SEQUENCE filtering alone, 50.7% of problems are solved and when combining either COST-REGULAR or GEN-SEQUENCE with GSC, 65.2% of problems are solved (with a time out of 1 hour). In our experiments, in average over the 42 heuristics and the 5 randomized runs, ATMOSTSEQCARD and GSC solve respectively 84.29% and 87.19% of instances and combining ATMOSTSEQCARD with GSC solves 87.42% instances in a time out of 20 minutes. Moreover, using the model *gsc+amsc*, the best heuristic was able to solve 96.20% of these instances.

Next, we considered an optimization version of the same problem, introduced in [9] and used for instance in [12, 13]. Here, the objective is to minimize the number of empty slots in the assembly line. In other words, we want to find the minimum value of n such that the standard model is satisfiable. With this setting, we can tackle the 23 instances of *set5*, for which no solution has been found.

We consider three groups of instances with the same (original) number of variables, respectively 200, 300 and 400. For each model, we report the minimum and average objective values, as well as CPU time, all averaged across the instances in each group.

Table 5: Optimization on *set5*.

Instance	*amsc*			*gsc*			*gsc+amsc*			*sum*		
	Empty slots		time (s)	Empty slots		time (s)	Empty slots		time (s)	Empty slots		time (s)
	min	avg	avg	min	avg	avg	min	avg	avg	min	avg	avg
pb_200	7.75	8.32	13.06	7.87	8.35	44.03	**7.62**	**8.27**	53.09	7.75	8.32	21.52
pb_300	11.62	**12.37**	53.04	11.87	12.77	99.19	**11.50**	12.47	129.04	11.87	12.57	42.49
pb_400	**10.57**	11.45	10.28	11.14	11.74	185.44	11.00	11.71	175.28	**10.57**	**11.34**	6.58

In Table 5, we observe an outcome similar to the satisfaction case for the two smaller groups, where, *gsc+amsc* and *amsc* give better solutions for the first and second group, respectively. However, for larger instances, the extra pruning does not seem to help to get better solutions, and *sum* is as good as *amsc*, and in fact a little bit better on average.

5.2 Crew-rostering

Problem description. In this problem, working shifts have to be attributed to employees over a period, so that the required service is met at any time and working regulations are respected. The latter condition can entail a wide variety of constraints. Previous work

[11] [15] used allowed (or forbidden) patterns to express successive shift constraints. For example, with 3 shifts of 8 hours per day: D (day), E (evening) and N (night), ND can be forbidden since employees need some rest after night shifts. In this paper, we consider a simple case involving 20 employees with 3 shifts of 8 hours per days where no employee can work more than one 8h shift per day, no more than 5 days per period of 7 days, and the break between two worked shifts must be at least 16h. The planning horizon is of 28 days, and each employee must work 17 shifts over the 4 weeks period (i.e. 34 hours per week in average).

Models and heuristics. We use a model with one Boolean variable e_{ij} for each of the m employees and each of the n shifts stating if employee i works on shift j. The demand d_j^s on each shift j is enforced through a sum constraint $\sum_{i=1}^{m} e_{ij} = d_j^s$.The other constraints are stated using two AtMostSeqCard constraints per employee, one with ratio $u/q = 1/3$, another with ratio $5/21$, and both with the same demand $d = 17$. We compare four models. In the first (*sum*), we use a decomposition in a chain of AtMost constraints. In the second (*amsc*) we use two AtMostSeqCard constraints per employee j, of the form:

$$\text{AtMostSeqCard}(u, q, d, [e_{i1}, \ldots, e_{in}])$$

In the first constraint we have $u = 1, q = 3, d = 17$ and in the second constraint we have $u = 5, q = 21, d = 17$. Both are propagated using Algorithm 3. In the third model (*gsc*), we use the following Gsc constraint to encode the constraint $\text{AtMostSeqCard}(u, q, d, [e_{i1}, \ldots, e_{in}])$:

$$\text{Gsc}(0, u, q, \{0 : n - d, 1 : d\}, \{0 : n - d, 1 : d\}, [e_{i1}, \ldots, e_{in}], \{1\})$$

Note that in this case, since the domains are Boolean, the Gsc is in this case equivalent to AtMostSeqCard. Therefore, it cannot prune more since the latter enforces ac. However, it is stronger than the decomposition. Last, in the fourth model (*mamsc*) the conjunction of the the two AtMostSeqCard constraints is propagated using Algorithm 4.

We used the following four variable ordering heuristics.

1. *Lexicographic*: Explores shifts chronologically and pick an employee at random;
2. *Middle*: Similar as above, however we start exploring shifts from the middle;
3. *Employee*: Picks an employee with min slack, then a possible shift of max demand;
4. *Shift*: Similar as above, however, the shift is selected first, then the employee.

In all cases, we branch by assigning the value 1 to the chosen pair (employee, shift).

Benchmarks. We generated 341 instances, with worker availability ranging from 82% to 48% by increment of 0.1. This value denotes the probability that a given employee is willing to work during a given shift. It allows to vary the constrainedness of the problem. 228 of these instances were found feasible, 77 infeasible and 36 remain open. We report results for the *satisfiable* and *unsatisfiable* sets with 5 random runs per instance.

We report the results for the static heuristics in Table 6 and for the dynamic heuristics in Table 7. The first column indicates the total number of successful runs (#sol),

Table 6: Evaluation of the filtering methods: static branching (highest success counts are in bold fonts)

Lexicographic

Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
sum	0	-	-	-	-	170	0.05	0.02	86	452
gsc	25	308.93	344.29	74074	84301	175	2.56	9.71	262	1794
amsc	125	164.36	239.56	1828347	2759080	213	1.76	21.95	22621	292152
mamsc	**534**	87.29	188.81	685720	1491867	**271**	2.80	45.02	27150	444913

From the middle to the sides

Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
sum	1	166.76	0.00	5716015	0	160	0.04	0.00	0	0
gsc	7	253.20	301.63	53763	63110	165	1.07	0.08	0	0
amsc	57	161.38	267.23	2207676	3621762	201	0.20	1.46	1622	15809
mamsc	**336**	134.95	239.11	1410458	2525422	**265**	0.05	0.00	0	0

then we report CPU time and number of backtracks, averaged over all instances and runs, as well as the standard deviation on this sample. Clearly, achieving AC on the (MULTI)ATMOSTSEQCARD constraint have a significant impact on the efficiency of the model. The decomposition into sum constraints cannot solve any satisfiable instance with lexicographic branching, and only one when starting from the middle of the sequence. The model using GSC offers a much more potent filtering, however, it is not as strong as AC on the ATMOSTSEQCARD constraint and moreover, it is much slower. On the other hand, the model using Algorithm 3 for the ATMOSTSEQCARD constraint achieves AC whilst being as fast as the decomposed model in terms of exploration. Moreover, combining the two ATMOSTSEQCARD constraints and using Algorithm 4 allows to solve about four times more satisfiable instances with *Lexicographic* branching and six times more with *Middle* branching.

The COST-REGULAR constraint could be used to enforce the same level of consistency as the combination of two ATMOSTSEQCARD constraints. The possible patterns can be encoded through a finite automaton whilst the overall cardinality is encoded by the counter. Notice that using a REGULAR constraint (i.e., without cost) and modeling the overall work load with a cardinality constraint would not enforce a higher level of consistency than the decomposition into cardinality constraints (i.e., model *sum*) since ATMOST constraints are monotone. A worst case analysis would indicate that the number of states in the automaton is too large. However, using the *O.D.E.N* alphabet, standing for *Off*, *Day*, *Evening* and *Night* shift, respectively, the two ATMOSTSEQ constraints can be encoded using an automaton involving no more than 41 states.[3] Transitions labeled D,E or N have a cost 1, whilst transitions labeled O have a null cost. The

[3] We thank the anonymous reviewer who provided the automaton.

target cost is set to the required working load. We do not compare our approach against such a COST-REGULAR encoding, although it would be interesting to empirically assess the computational overhead of this latter method. It is possible, however, to get an idea of this overhead by considering the similar approach used in [21] on car-sequencing benchmarks. In this case, the COST-REGULAR encoding seems very close in CPU time and overall behavior to the cubic algorithm used for propagating the GEN-SEQUENCE constraint. Since the level of consistency would be strictly the same as in the model using the MULTIATMOSTSEQCARD constraint, we might expect similar results, albeit with slightly larger CPU times.

Table 7: Evaluation of the filtering methods (dynamic branching)

Most constrained employee

Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
sum	772	21.93	104.91	205087	1000794	165	0.06	0.00	0	2
gsc	746	65.75	180.29	14133	42235	175	0.98	0.09	0	3
amsc	818	20.51	103.76	147479	761261	215	0.13	0.55	330	2582
mamsc	**842**	20.78	111.00	125886	676061	**270**	0.05	0.01	0	2

Most constrained shift

Model	satisfiable (1140)					unsatisfiable (385)				
	#sol	CPU time		#backtracks		#sol	CPU time		#backtracks	
		avg	dev	avg	dev		avg	dev	avg	dev
sum	987	20.76	102.53	169964	853020	352	19.74	99.61	180161	967933
gsc	1006	33.30	107.08	8875	31586	335	15.97	95.36	5145	35824
amsc	1061	10.07	65.02	90247	593928	362	12.19	77.37	108797	736775
mamsc	**1074**	10.94	77.37	91222	667176	**377**	14.63	107.58	110244	834887

When using dynamic heuristics (see Table 7), the difference between the different models becomes much less spectacular. However, the trend is the same, with the model combining the pairs of ATMOSTSEQCARD constraint dominating the other models.

6 Conclusion

We have introduced a linear time algorithm for achieving arc consistency on the constraint ATMOSTSEQCARD, a particular case of sequence constraint useful for instance in car-sequencing and crew-rostering applications.

Previously, the best AC algorithm for that constraint had an $O(n^2)$ time complexity [10]. However, it ran in $O(n^2 \log n)$ time down a branch since subsequent calls cost $O(n \log n)$, whilst our algorithm is not incremental hence requires up to $O(n^2)$ steps down a branch.

We also proposed some extensions of ATMOSTSEQCARD and showed that one can easily adapt the initial algorithm to achieve arc consistency on these constraints.

In particular, we have shown that it is possible to achieve AC on a conjunction of ATMOSTSEQCARD constraints on the same scope with the same complexity as achieving AC on each conjunct.

Our empirical evaluation shows that current models for car-sequencing and crew-rostering problems can take advantage of this constraint. On car-sequencing problems, the filtering of our propagator is incomparable with that of GSC, albeit much less computationally costly. For some instances, it is thus worth combining both constraints. In other cases, a model using only the ATMOSTSEQCARD constraint is better. On crew-rostering problem, the REGULAR constraint is often used since it can model arbitrary working patterns. However, ATMOSTSEQCARD can model useful patterns and provide efficient and cheap filtering for them. Moreover, several such patterns can be combined to obtain a stronger filtering without degrading the time complexity.

Acknowledgments

We would like to thank Nina Narodytska for her precious help and comments, and the Cork Constraint Computation Center (4C) for kindly granting us access to its computing resources.

References

1. N. Beldiceanu and M Carlsson. Revisiting the Cardinality Operator and Introducing the Cardinality-Path Constraint Family. In *ICLP*, pages 59–73, 2001.
2. N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical Computation Modelling*, 20(12):97–123, 1994.
3. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The Slide Meta-Constraint. In *CPAI Workshop, held alongside CP*, 2006.
4. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Slide: A Useful Special Case of the Cardpath Constraint. In *ECAI*, pages 475–479, 2008.
5. S. Brand, N. Narodytska, C.-G. Quimper, P. J. Stuckey, and T. Walsh. Encodings of the Sequence Constraint. In *CP*, pages 210–224, 2007.
6. S. Demassey, G. Pesant, and L.-M. Rousseau. A Cost-Regular Based Hybrid Column Generation Approach. *Constraints*, 11(4):315–333, 2006.
7. M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI*, pages 290–295, 1988.
8. I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints, 1999.
9. K-S. Hindi and G. Ploszajski. "formulation and solution of a selection and sequencing problem in car manufacture". *Computers & Industrial Engineering*, 26(1):203 – 211, 1994.
10. M. J. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-Based Propagators for the SEQUENCE and Related Global Constraints. In *CP*, pages 159–174, 2008.
11. J. Menana and S. Demassey. Sequencing and Counting with the multicost-regular Constraint. In *CPAIOR*, pages 178–192, 2009.
12. L. Perron and P. Shaw. Combining Forces to Solve the Car Sequencing Problem. In *CPAIOR*, pages 225–239. 2004.
13. L. Perron, P. Shaw, and V. Furnon. Propagation Guided Large Neighborhood Search. In *CP*, pages 468–481, 2004.
14. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP*, pages 482–495, 2004.

15. G. Pesant. Constraint-Based Rostering. In *PATAT*, 2008.
16. J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *AAAI*, pages 209–215, 1996.
17. J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *CP*, pages 32–46, 1997.
18. M. Siala, E. Hebrard, and M-J. Huguet. A study of Branching Heuristics for the Car-sequencing Problem. In *SSNOW Workshop, held alongside CPAIOR*, 2012.
19. B.M. Smith. Succeed-first or Fail-first: A Case Study in Variable and Value Ordering, 1997.
20. C. Solnon, V. Cung, A. Nguyen, and C. Artigues. The car sequencing problem : Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *EJOR*, 191:912–927, 2008.
21. W. J. van Hoeve, G. Pesant, L.-M.Rousseau, and A. Sabharwal. New Filtering Algorithms for Combinations of Among Constraints. *Constraints*, 14(2):273–292, 2009.
22. W. J. van Hoeve, G. Pesant, L.-M. Rousseau, and Ashish Sabharwal. Revisiting the Sequence Constraint. In *CP*, pages 620–634, 2006.

Appendix

Complete proof of Lemma 8

Proof. We prove the following three invariants, true at the beginning of each step of the main loop:

- The cardinality of the j^{th} subsequence is equal to $c[(i+j-2) \bmod q]+count[i-1]$.
- The number of subsequences of cardinality k is equal to $occ[n - count[i-1] + k]$.
- The cardinality maximum of any subsequence is equal to $\max_c$.

Cardinality of the subsequences.

Let w_i denote the assignment w after $i-1$ steps of the loop. Notice that at the beginning and the end of the sequence of variables, subsequences are truncated. However, to simplify the notations, we will consider that $w[-q], w[-q+1], \ldots, w[-1]$ exist and are equal to 0. Thus we can write that the cardinality of the j^{th} is equal to $\sum_{l=i-q+j}^{i+j-1} w_i[l]$.

We prove the first invariant by induction, i.e., let $P(i)$ denote the fact that the following equalities hold at the beginning of a step i:

$$(\sum_{l=i-q+j}^{i+j-1} w_i[l]) = (c[i+j-2 \bmod q] + count[i-1])\ \forall j \in [1, \ldots, q]$$

The base case $P(1)$ is easily checkable from the initialization of c.

Now suppose that $P(i)$ holds, and consider the state of c at the beginning of step $i+1$. First, note that at step i of the loop, only the value of $c[i-1 \bmod q]$ changes. Consider $j \in [1, \ldots, q-1]$. In this case, $((i+1)+j-2 \bmod q) = (i+j-1 \bmod q) \neq (i-1 \bmod q)$. Therefore, $c[(i+1)+j-2 \bmod q]$ has not changed between step i and step $i+1$, and since $P(i)$ holds, we have:

$$(\sum_{l=i-q+(j+1)}^{i+(j+1)-1} w_i[l]) = (c[i+(j+1)-2 \bmod q] + count[i-1])$$

which can be rewritten as follows:

$$(\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_i[l]) = (c[(i+1)+j-2 \bmod q] + count[i-1])$$

Now there are two possibilities. Either $count$ is incremented, i.e., $count[i] = count[i-1]+1$, and in that case $w_{i+1}[i] = w_i[i]+1$. Or $count$ is not incremented, and in that case $w_{i+1}[i] = w_i[i]$.

In both cases we have:

$$\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_{i+1}[l] = \sum_{l=(i+1)-q+j; l\neq i}^{(i+1)+j-1} w_i[l] + w_{i+1}[i]$$

since $w_{i+1}[l] = w_i[l]$ for all $l \neq i$. Hence we obtain:

$$(\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_{i+1}[l]) = (c[i+(j+1)-2 \bmod q] + count[i-1]) - w_i[i] + w_{i+1}[i]$$

which can be rewritten as:

$$(\sum_{l=(i+1)-q+j}^{(i+1)+j-1} w_{i+1}[l]) = (c[(i+1)+j-2 \bmod q] + count[i])$$

Thus $P(i+1)$ holds.

Now we look at the last case: $j = q$. Here, at step i the value of $c[i-1 \bmod q]$ is set to $c[i+q-2 \bmod q] + w_{i+1}[i+q] - w_{i+1}[i]$. Since $P(i)$ holds, we can replace $c[i+q-2 \bmod q]$ by $\sum_{l=i}^{i+q-1} w_i[l]) - count[i-1]$, so at the beginnning of step $i+1$ we have:

$$c[(i+1)+q-2 \bmod q] = (\sum_{l=i}^{i+q-1} w_i[l]) - count[i-1] + w_{i+1}[i+q] - w_{i+1}[i]$$

however, since $\sum_{l=i}^{i+q-1} w_i[l]) = w_i[i] + \sum_{l=i+1}^{i+q-1} w_{i+1}[l])$ we have:

$$c[(i+1)+q-2 \bmod q] = \sum_{l=i+1}^{i+q} w_{i+1}[l] - count[i-1] + w_i[i] - w_{i+1}[i]$$

Therefore, since $count[i] = count[i-1] + w_{i+1}[i] - w_i[i]$, the following holds:

$$c[(i+1)+q-2 \bmod q] = \sum_{l=i+1}^{i+q} w_{i+1}[l] - count[i]$$

We have shown that $P(i)$ implies $P(i+1)$, and we can therefore conclude that at the beginnig of each step i of the loop $P(i)$ (that is, the first invariant) holds.

Occurrences of each cardinality.

We proceed as for the first invariant, and prove it by induction. The base case is easy to check. Since $count[0] = 0$, and since the array c is properly initialized.

Now we assume that there are exactly $occ[n - count[i-1] + k]$ subsequences involving x_i which cardinility is equal to k in w_i, and we show that at the beginning of step $i+1$ there will be $occ[n-count[i]+k]$ subsequences involving x_{i+1} of cardinality k in w_{i+1}.

There are two reasons for cardininalities to change.

First, when moving up to the next step in the loop, we move from subsequences involving x_i to subsequences involving x_{i+1}. There are $q-1$ subsequences involving both x_i and x_{i+1}. So we simply need to make sure that the occurrences are updated to reflect the fact that the subsequence $x_{i-q+1}, \ldots, x_i$ should not be counted anymore, whilst the subsequence $x_{i+1}, \ldots, x_{i+q}$ should now be. Let k_1 (resp. k_2) be the cardinality of the former (resp. latter) subsequence. As established by the first invariant, $k_1 = c[(i-1) \bmod q] + count[i-1]$, that is the value $prev$ in Line 2 is set to $k_1 - count[i-1]$. Moreover, $next$ is given the value $c[(i+q-2) \bmod q] + w[i+q] - w[i]$. However, from invariant 1, we have $c[(i+q-2) \bmod q] + count[i-1] = \sum_{l=i}^{i+q-1} w[l]$. It follows that

$$next = \sum_{l=i}^{i+q-1} w[l] + w[i+q] - w[i] - count[i-1] = \sum_{l=i+1}^{i+q} w[l] - count[i-1]$$

therefore $next = k_2 - count[i-1]$. To maintain invariant (2), we therefore need to increment the value of $occ[n - count[i-1] + k_2]$ and decrement the value of $occ[n - count[i-1] + k_1]$. However, this is precisely what is done in Line 4 and 5.

Second, when the conditions in Line 1 are met, the value of $w[i]$ is set to 1. Since its value was previously 0, the cardinality of every subsequence involving $w[i]$ should be incremented before starting the next step $(i+1)$. This happens automatically because in this case the value of $count[i]$ will be set to $count[i-1]+1$. Indeed, for any integer k, the number of occurrences of subsequences of cardinality $k-1$ at the beginning of step i is $occ[n - count[i-1] + k - 1]$. Therefore, since $count[i] = count[i-1] + 1$, at the beginning of step $i+1$, we have $occ[n - (count[i]-1) + k - 1]$, that is, $occ[n - count[i] + k]$.

Cardinality maximum.

Here we show that the maximum value of the cardinalities of the current subsequences is properly maintained. When the number of occurrences of a cardinality k becomes non-null and if $k > \max_c$, then $\max_c$ is set to k. Similarly, When the number of occurrences of a cardinality k becomes null and if $k = \max_c$, then $\max_c$ is decreased. Last, when the cardinality of all subsequences is incremented, $\max_c$ is incremented too.

These operations are correct because from one step i to $i+1$, the value of $\max_c$ cannot change by more than 1. Indeed, only the first subsequence is removed, the other $q-1$ subsequences remain unchanged. Moreover, the first subsequence is replaced by the last subsequence to which a value $a \in [0,1]$ is added, and another value $b \in [0,1]$ is substracted. Therefore its value cannot change by more than 1, hence $\max_c$.

Now having these three invariants, one can check that at each step i the values of $w[i]$ will be the same as in Algorithm 1. □

Algorithm 2: `leftmost_count`

Data: $u, q, [x_1, \ldots, x_n]$
Result: $count : [0, \ldots, n] \mapsto [0, \ldots, n]$
foreach $i \in [1, \ldots, n]$ **do**
  $w[i] \leftarrow \min(x_i)$;
  $occ[i] = 0$;
foreach $i \in [0, \ldots, n]$ **do** $count[i] \leftarrow 0$;
;
$c[0] \leftarrow w[1]$;
foreach $i \in [1, \ldots, u]$ **do** $occ[n + i] = 0$;
;
foreach $i \in [1, \ldots, q]$ **do**
  $w[n + i] \leftarrow 0$;
  if $i < q$ **then** $c[i] \leftarrow c[i - 1] + w[i + 1]$;
  ;
  $occ[n + c[i - 1]] \leftarrow occ[n + c[i - 1]] + 1$;
$\max_c \leftarrow \max(\{c[i] \mid i \in [0, \ldots, q - 1]\})$;
foreach $i \in [1, \ldots, n]$ **do**
1 **if** $\max_c < u$ & $|\mathcal{D}(x_i)| > 1$ **then**
    $\max_c \leftarrow \max_c + 1$;
    $count[i] \leftarrow count[i - 1] + 1$;
    $w[i] \leftarrow 1$;
  else $count[i] \leftarrow count[i - 1]$;
  ;
2 $prev \leftarrow c[(i - 1) \bmod q]$;
3 $next \leftarrow c[(i + q - 2) \bmod q] + w[i + q] - w[i]$;
  $c[(i - 1) \bmod q] \leftarrow next$;
  if $prev \neq next$ **then**
4 $occ[n + prev] \leftarrow occ[n + prev] - 1$;
5 $occ[n + next] \leftarrow occ[n + next] + 1$;
    if $next + count[i] > \max_c$ **then** $\max_c \leftarrow \max_c + 1$;
    ;
    if $occ[n + prev] = 0$ & $prev + count[i] = \max_c$ **then**
      $\max_c \leftarrow \max_c - 1$;
return $count$;

$\mathcal{D}(x_i)$		.	0	.	.	.	.	.	.	0	1	0	.	.	.	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		**1**	0	**1**	**1**	**1**	0	0	0	0	1	0	**1**	**1**	**1**	0	0	0	**1**	0	**1**	**1**	1	
$\overleftarrow{w}[i]$		**1**	0	0	**1**	**1**	**1**	0	0	0	1	0	**1**	**1**	**1**	0	0	0	0	**1**	**1**	**1**	1	
$L[i]$	0	1	1	2	3	4	4	4	4	4	4	4	5	6	7	7	7	7	8	8	9	10	10	
$R[n-i+1]$		10	9	9	9	8	7	6	6	6	6	6	6	5	4	3	3	3	3	3	2	1	0	0
$L[i]+R[n-i+1]$		11	10	11	12	12	11	**10**	**10**	10	10	10	11	11	11	**10**	**10**	**10**	11	11	11	11	10	
$L[i-1]+R[n-i]$		**9**	10	10	10	10	10	10	10	10	10	10	**9**	**9**	**9**	10	10	10	10	10	**9**	**9**	10	
AC($\mathcal{D}(x_i)$)		**1**	0	.	.	.	.	**0**	**0**	0	1	0	**1**	**1**	**1**	**0**	**0**	**0**	.	.	**1**		1	

Fig. 4: Example of the execution of Algorithm 3 for the constraint AtMostSeqCard(u=4, q=8, d=12, $[x_1, \ldots, x_n]$)