# Algorithme optimal d'arc-consistance pour une séquence de contraintes AtMost avec cardinalité

Emmanuel Hébrard, Marie-José Huguet, Mohamed Siala

# Algorithme optimal d'arc-consistance pour une séquence de contraintes AtMost avec cardinalité

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

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

**Résumé**

La contrainte AtMostSeqCard est la conjonction entre une contrainte de cardinalité sur une séquence de $n$ variables et de $n - q + 1$ contraintes AtMost $u$ sur toutes sous-séquences de variables de taille $q$. Elle se retrouve en particuliers dans des problèmes de type car-sequencing et crew-rostering. Une adaptation de deux algorithmes conçus pour la contrainte AmongSeq afin de traiter la contrainte AtMostSeqCard a été proposée dans [18]. Ces algorithmes ont une complexité temporelle respective au pire en $O(2^{q-1}n)$ et $O(n^3)$. Dans [10], un autre algorithme a été adapté de manière similaire pour la contrainte AtMostSeqCard avec une complexité temporelle au pire en $O(n^2 \log n)$. Cet article présente un algorithme réalisant l'arc-consistance de la contrainte AtMostSeqCard avec une complexité temporelle au pire en $O(n)$ (et donc optimal). Enfin, des expérimentations sont présentées pour évaluer l'efficacité de cet algorithme sur des problèmes de car-sequencing et de crew-rostering.

## 1 Introduction et état de l'art

Cet article s'intéresse à l'existence de restrictions sur des séquences de décision : certaines séquences peuvent être autorisées ou préférées et d'autres interdites. Par exemple, dans les problèmes de confection d'horaires pour des équipes (crew-rostering), il est, souvent interdit de positionner la même équipe sur un travail de soirée ou de nuit puis de la placer le lendemain sur un travail en matinée.

Les contraintes Regular [12] et cost-Regular [7] permettent d'exprimer des restrictions quelconques sur des séquences de décision. Toutefois, dans des cas particuliers, il existe des algorithmes plus efficaces. Par exemple, des algorithmes de filtrage ont été proposés pour la contrainte AmongSeq [6, 10, 19, 18]. Cette contrainte vérifie que toutes les sous-séquences de taille $q$ contiennent au minimum $l$ et au maximum $u$ valeurs prises dans un ensemble $v$ et est très employée pour modéliser des problèmes de car-sequencing et de crew-rostering. Mais, les contraintes de ces deux familles de problèmes ne se limitent pas uniquement à cette définition. En effet, souvent, il n'y a pas de borne inférieure sur le nombre de valeurs ($l = 0$) ; de plus, le nombre de valeurs prises dans l'ensemble $v$ est généralement contraint par une demande globale.

Cet article étudie la contrainte AtMostSeqCard qui répond à ce besoin. Cette contrainte, impliquant $n$ variables $x_1, \ldots, x_n$, assure que, dans chaque sous-séquence de longueur $q$, pas plus de $u$ choix sont fixés à une valeur prise dans l'ensemble $v$, et que sur toute la longueur de la séquence, il y a exactement $d$ choix fixés à une valeur prise dans $v$. Dans le cas de problèmes de car-sequencing, cette contrainte permet d'exprimer que, pour une option donnée, dans toute sous-séquence de taille $q$, il ne peut y avoir plus de $u$ classes de véhicules avec cette option et que le nombre total de véhicules ayant cette option vaut exactement $d$. Dans le cas de problèmes de crew-rostering, cette contrainte permet de traduire, par exemple, qu'un employé doit disposer d'au minimum 16h de pause entre deux périodes de travail de 8h ou qu'une semaine de travail ne peut dépasser 40h, tout en fixant un nombre total d'heures devant être travaillées sur la période de planification.

Un problème de Satisfaction de Contraintes (CSP) est défini par un triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ où $\mathcal{X}$ représente l'ensemble des variables, $\mathcal{D}$ l'ensemble des domaines finis pour chacune des variables et $\mathcal{C}$ l'ensemble des contraintes qui spécifient les combinaisons de valeurs autorisés sur des sous-ensembles de variables. On suppose que $\mathcal{D}(x) \subset \mathbb{Z}$ pour tout $x \in \mathcal{X}$, et on note $\min(x)$ et $\max(x)$ les valeurs minimale et maximale, de $\mathcal{D}(x)$. Une instanciation

d'un ensemble de variables $\mathcal{X}$ est un tuple $w$ tel que $w[i]$ représente la valeur affectée à la variable $x_i$. Une contrainte $C \in \mathcal{C}$ portant sur un ensemble de variables $\mathcal{X}$ caractérise une relation, i.e., un sous-ensemble de tuples du produit cartésien du domaine des variables de $\mathcal{X}$. Une instanciation $w$ est dite consistante pour une contrainte $C$ ssi elle appartient à la relation correspondante. Une contrainte $C$ est dite *arc-consistante* (AC) ssi, pour toute valeur $j$ de chaque variable $x_i$ qu'elle met en jeu, il existe une instanciation consistante $w$ telle que $w[i] = j$. Pour $x_i = j$, on appelle *support*, une telle instanciation $w$ .

Cet article est organisé de la manière suivante : la section 2, présente un bref état de l'art sur les contraintes de séquence. La section 3 expose un algorithme en temps linéaire (et donc optimal) pour propager la contrainte ATMOSTSEQCARD. Des expérimentations sur des instances de car-sequencing et de crew-rostering sont données en section 4 avant une conclusion en section 5.

## 2 Les contraintes de SEQUENCE

Plusieurs variantes des contraintes de séquence sont passées en revue avant de motiver le besoin de la contrainte ATMOSTSEQCARD.

Soient $v$ un ensemble d'entiers et $l, u, q$ des entiers. Les contraintes de séquence s'expriment par des conjonctions de contraintes AMONG limitant, pour un ensemble de variables, le nombre d'occurrences d'un ensemble de valeurs.

**Définition 2.1** $\text{AMONG}(l, u, [x_1, \ldots, x_q], v) \Leftrightarrow$

$$l \leq |\{i \mid x_i \in v\}| \leq u$$

### 2.1 Séquence de contraintes AMONG

La contrainte AMONGSEQ est une séquence de contraintes AMONG de largeur glissante $q$ sur un vecteur de $n$ variables.

**Définition 2.2** $\text{AMONGSEQ}(l, u, q, [x_1, \ldots, x_n], v) \Leftrightarrow$

$$\bigwedge_{i=0}^{n-q} \text{AMONG}(l, u, [x_{i+1}, \ldots, x_{i+q}], v)$$

Elle a été proposée dans [2], et le premier algorithme (incomplet) pour propager cette contrainte a été proposé dans [1]. Par la suite, deux algorithmes complets ont été développés pour propager cette contrainte AMONGSEQ [19, 18]. Le premier algorithme utilise une reformulation basée sur la contrainte REGULAR et réalise l'AC avec une complexité temporelle en $O(2^{q-1}n)$. Le second un algorithme réalise l'AC avec une complexité temporelle en $O(n^3)$. Cet algorithme peut être étendu pour traiter tout ensemble de contraintes AMONG portant sur des variables consécutives (contrainte GEN-SEQUENCE) et a alors une complexité temporelle en $O(n^4)$ dans le pire des cas. Dans le cas qui nous intéresse, la complexité de cet algorithme est ramenée à $O(n^3)$ car ATMOSTSEQCARD, se décompose en $O(n)$ AMONG. Puis, un algorithme de flot, réalisant l'AC en $O(n^2)$ a été proposé dans [10]. Cet algorithme a pu également être adapté, en utilisant des outils plus généraux de programmation linéaire, pour traiter la contrainte GEN-SEQUENCE avec une complexité temporelle au pire en $O(n^2 \log n)$.

### 2.2 Séquence de contraintes ATMOST

Malgré son intérêt, la contrainte AMONGSEQ ne modélise pas exactement le type de contraintes nécessaires pour le car-sequencing et le crew-rostering. En effet, souvent il n'y a pas de borne inférieure sur la cardinalité des sous-séquences, i.e., $l = 0$. AMONGSEQ est ainsi inutilement générale sur ce plan là. De plus, la contrainte de capacité est souvent associée à une exigence de cardinalité.

Par exemple, pour le car-sequencing, les classes de véhicules demandant une option donnée ne peuvent être groupées ensemble car la station de travail nécessaire ne peut réaliser consécutivement qu'une partie des véhicules (*at most* $u$ véhicules dans toute sous-séquence de longueur $q$). Le nombre total d'occurrences des différentes classes est également connu, et se traduit en une contrainte de cardinalité globale plutôt qu'en termes de bornes inférieures sur chaque sous-séquence.

Pour le crew-rostering, les motifs sur les périodes de travail autorisées peuvent être complexes, et la contrainte REGULAR est souvent utilisée pour les modéliser. Cependant, travailler au plus $u$ périodes sur $q$ est un cas important. Si les journées sont divisées en 3 périodes de 8h, ATMOSTSEQ avec $u = 1$ et $q = 3$ traduit le fait qu'aucun employé ne peut travailler plus d'une période par jour *et* qu'il doit y avoir une pause de 24h avant de changer de période de travail. De plus, comme pour le car-sequencing, la borne inférieure sur le nombre de périodes travaillées est globale (et dépend ici du contrat de travail).

Ainsi, il faut souvent traiter une séquence de contraintes ATMOST définie par :

**Définition 2.3** $\text{ATMOST}(u, [x_1, \ldots, x_q], v) \Leftrightarrow$

$$\text{AMONG}(0, u, [x_1, \ldots, x_q], v)$$

**Définition 2.4** $\text{ATMOSTSEQ}(u, q, [x_1, \ldots, x_n], v) \Leftrightarrow$

$$\bigwedge_{i=0}^{n-q} \text{ATMOST}(u, [x_{i+1}, \ldots, x_{i+q}], v)$$

On peut noter qu'il est facile de maintenir l'AC de cette contrainte. En effet, la contrainte ATMOST est monotone, c'est-à-dire, que l'ensemble des supports pour la valeur 0 est un sur-ensemble strict de l'ensemble des supports pour la valeur 1. On en déduit que la contrainte ATMOSTSEQ est AC ssi chaque contrainte ATMOST est AC [4].

Un bon compromis entre la puissance de filtrage et la complexité peut être effectué en couplant les raisonnements sur le nombre total d'occurrences de valeurs dans l'ensemble $v$ et sur la séquence de contraintes AtMost. [1] La contrainte AtMostSeqCard est donc définie par la conjonction entre la contrainte AtMostSeq et des contraintes de cardinalité sur le nombre total d'occurrences de chaque valeur de $v$ :

**Définition 2.5** $\text{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n], v) \Leftrightarrow$

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

Les deux algorithmes AC proposés dans [19] ont été adaptés [18] pour établir l'AC sur la contrainte AtMostSeqCard. Premièrement, de la même manière que AmongSeq peut être écrite avec la contrainte Regular, AtMostSeqCard peut s'exprimer à l'aide de la contrainte cost-Regular dans laquelle le cout représente la demande globale et est augmenté sur les transitions ayant la valuation 1. Cette procédure a la même complexité au pire, i.e., $O(2^{q-1}n)$. Deuxièment, une variante plus générale de l'algorithme polynomial (Gen-Sequence) est utilisée pour propager la contrainte AtMostSeqCard décomposée en une conjonction de contraintes Among de la manière suivante :

$$\begin{aligned}\text{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n], v) \quad &\Leftrightarrow \\ \bigwedge_{i=0}^{n-q} \text{Among}(0, u, q, [x_{i+1}, \ldots, x_{i+q}], v) \quad &\wedge \\ \text{Among}(d, d, [x_1, \ldots, x_n], v)&\end{aligned}$$

Comme dans cette décomposition, la dernière contrainte Among représentant la cardinalité n'a pas les mêmes paramètres $l$ et $u$ que les autres, on obtient, en utilisant respectivement les algorithmes de van Hoeve et al. ou de Maher et al., une procédure en $O(n^3)$ ou en $O(n^2 \log n)$ pour établir l'AC de la contrainte AtMostSeqCard.

### 2.3 La contrainte « Global Sequencing »

Dans le cas particulier du car-sequencing, il n'y a pas seulement une cardinalité globale pour les valeurs de $v$, mais chaque valeur correspond à une classe de véhicules et est contraint par un nombre d'occurrences. Ainsi, Puget et Régin ont considéré la conjonction entre la contrainte AmongSeq et la contrainte Gcc. Soient $c_l$ et $c_u$ deux fonctions sur les entiers tels que $c_l(j) \leq c_u(j)\ \forall j$, et soit $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}(x_i)$. La contrainte de cardinalité globale (Gcc) est définie de la manière suivante :

**Définition 2.6** $\text{Gcc}(c_l, c_u, [x_1, \ldots, x_n]) \Leftrightarrow$

$$\bigwedge_{j \in \mathcal{D}} c_l(j) \leq |\{i \mid x_i = j\}| \leq c_u(j)$$

La contrainte « Global Sequencing » (Gsc) est définie par :

**Définition 2.7** $\text{Gsc}(l, u, q, c_l, c_u, [x_1, \ldots, x_n], v) \Leftrightarrow$

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

Les fonctions $c_l$ et $c_u$ sont définis de telle sorte que pour une valeur $v$, $c_l(v)$ et $c_u(v)$ correspondent au nombre d'occurrences de la classe correspondante d'un véhicule. Une reformulation de cette contrainte en un ensemble de contraintes Gcc a été introduite dans [15]. Cependant, effectuer l'AC de cette contrainte est NP-difficile [3]. En fait, réaliser la consistance de bornes l'est aussi. [2]

## 3 La contrainte AtMostSeqCard

Cette section, présente un algorithme de filtrage linéaire pour la contrainte AtMostSeqCard. Tout d'abord un algorithme glouton pour trouver un support avec une complexité temporelle en $O(nq)$ est exposé. Puis, on montre qu'avec deux appels à cette procédure il est possible d'assurer l'AC de la contrainte. Enfin, on montre que la complexité dans le pire cas peut se réduire à $O(n)$.

[18] et [10] ont fait remarquer qu'on pouvait considérer des variables booléennes et $v = \{1\}$, puisque la décomposition suivante des contraintes Among conserve le même niveau de consistance car elle est Berge-acyclique [6] :

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

Par la suite, on considére la restriction ci-dessous de AtMostSeqCard à des variables booléennes et avec $v = \{1\}$ :

**Définition 3.1** $\text{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \Leftrightarrow$

$$\bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u) \wedge (\sum_{i=1}^{n} x_i = d)$$

Soit $w$ un $n$-tuple dans $\{0,1\}^n$, $w[i]$ représente le $i^{ime}$ élément de $w$, $|w| = \sum_{i=1}^{n} w[i]$ sa cardinalité et $w[i:j]$ le $(|j-i|+1)$-tuple égal à $w$ sur la sous-séquence $[x_i, \ldots, x_j]$.

On montre tout d'abord qu'on peut trouver un support, de manière gloutonne, en fixant à 1 les variables dans l'ordre lexicographique à chaque fois que cela est possible, c'est à dire en respectant la contrainte AtMostSeq. En faisant cela, on obtient une instanciation de cardinalité maximale qui peut être facilement modifiée en une instanciation de cardinalité $d$. L'Algorithme 1 présente cette

1. Choix de modélisation utilisé dans [18] pour le car-sequencing.

2. Note interne "Comics" (Nina Narodytska).

règle gloutonne de calcul d'une instanciation $w$ maximisant la cardinalité de la séquence $(x_1, \ldots, x_n)$ en respectant la contrainte ATMOSTSEQ.

En ligne 1, le vecteur $w$ est initialisé à la valeur minimale du domaine de chaque variable. Puis, à chaque étape $i \in [1, \ldots, n]$ de la boucle principale, la variable $c(j)$ représente la cardinalité de la $j^{eme}$ sous-séquence impliquant la variable $x_i$. C'est à dire, à chaque étape $i$ :

$$c(j) = \sum_{l=\max(1,i-q+j)}^{\min(n,i+j-1)} w[l]$$

En suivant la règle gloutonne présentée plus haut, $w[i]$ est instancié à 1 ssi $x_i$ n'est pas encore instancié ($\mathcal{D}(x_i) = \{0, 1\}$) et si les contraintes de capacité ne sont pas violées , c'est à dire, qu'il n'y a pas de sous-séquence de cardinalité $u$ ou plus impliquant $x_i$. Pour cela, on teste la valeur maximale de $c(j)$ pour $j \in [1, \ldots, q]$ (condition ligne 2). Dans ce cas, la cardinalité de chaque sous-séquence impliquant $x_i$ est incrémentée (ligne 3). Enfin, lorsqu'on passe à la variable suivante, les valeurs de $c(j)$ sont décalées (ligne 4) et la valeur de $c(q)$ est obtenue à partir de sa valeur précédente en ajoutant $w[i+q]$ et en soustrayant $w[i]$ (ligne 5). Dans la suite, on note $\overrightarrow{w}$ (resp. $\overleftarrow{w}$), l'instanciation déterminée par l'algorithme `leftmost` pour la séquence $x_1.., x_n$ (resp. $x_n, .., x_1$). Pour simplifier, on note $\overleftarrow{w}[i]$ la valeur retournée par l'algorithme `leftmost` pour la variable $x_i$ (au lieu de $x_{n-i+1}$).

**Algorithm 1:** `leftmost`

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

**Lemme 3.1** *`leftmost` maximise $\sum_{i=1}^{n} x_i$ sous les contraintes* ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$.

*Preuve.* Soit $\overrightarrow{w}$ l'instanciation déterminée par `leftmost` et supposons qu'il existe une autre instanciation $w$ tel que $|w| > |\overrightarrow{w}|$. Soit $i$, le plus petit indice tel que $\overrightarrow{w}[i] \neq w[i]$. Par définition de $\overrightarrow{w}$, on sait que $\overrightarrow{w}[i] = 1$ et donc que $w[i] = 0$. Soit également $j$ le plus petit indice tel que $\overrightarrow{w}[j] < w[j]$ (un tel indice existe car $|w| > |\overrightarrow{w}|$).

On affirme que l'instanciation $w'$, égale à $w$, sauf pour $w'[i] = 1$ et $w'[j] = 0$ (comme pour $\overrightarrow{w}$) est consistante. En effet, sa cardinalité n'est pas modifiée par ce changement, donc $|w'| = |w|$. Si on considère toutes les contraintes de somme qui sont touchées par cette permutation de valeurs, c'est à dire les sommes $\sum_{l=a}^{a+q-1} w'[l]$ définies sur les intervalles $[a, a+q-1]$ tels que $a \leq i < a+q$ ou $a \leq j < a+q$. trois cas se présentent :

1. $a \leq i < j < a+q$. Dans ce cas, la valeur de la somme est identique pour $w$ et $w'$, donc elle est inférieure ou égale à $u$.
2. $i < a \leq j < a+q$. Dans ce cas, la valeur de la somme dans $w'$ est diminuée de 1 par rapport à $w$, elle est donc inférieure ou égale à $u$.
3. $a \leq i < a+q \leq j$. Dans ce cas, pour tout $l \in [a, a+q-1]$, on a $w'[l] \leq \overrightarrow{w}[l]$ car $j$ est le plus petit indice tel que $\overrightarrow{w}[j] < w[j]$. Donc, la somme est inférieure ou égale à $u$.

Donc, à partir d'une instanciation $w$ de cardinalité maximale qui diffère de $\overrightarrow{w}$ au rang $i$, on peut construire une autre instanciation de même cardinalité qui ne diffère de $\overrightarrow{w}$ jusqu'à $i+1$. En itérant, on peut obtenir une instanciation identique à $\overrightarrow{w}$, mais de cardinalité $|w|$, ce qui contredit donc notre hypothèse $|w| > |\overrightarrow{w}|$. □

**Corollaire 3.1** *Soit $\overrightarrow{w}$ un vecteur obtenu par `leftmost`,* ATMOSTSEQCARD$(u, q, d, [x_1, \ldots, x_n])$ *est satisfiable ssi les trois conditions suivantes sont respectées :*

$$\text{ATMOSTSEQ}(u, q, [x_1, \ldots, x_n]) \text{ est satisfiable} \quad (3.1)$$

$$\sum_{i=1}^{n} min(x_i) \leq d \quad (3.2)$$

$$|\overrightarrow{w}| \geq d \quad (3.3)$$

*Preuve.* Ces trois conditions sont nécessaires : (1) et (2) proviennent de la définition et (3) est une application directe du Lemme 3.1. Prouvons que ces conditions sont suffisantes en montrant que si elles sont vérifiées alors il existe une solution. Comme ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$ est satisfiable, $\overrightarrow{w}$ ne viole pas de séquence de contraintes ATMOST car la valeur 1 est affectée à $x_i$ ssi toutes les sous-séquences impliquant $x_i$ ont une cardinalité de $u-1$ ou moins. De plus, comme $\sum_{i=1}^{n} min(x_i) \leq d$, il y a au moins $|\overrightarrow{w}| - d$ variables telles que $min(x_i) = 0$ et $\overrightarrow{w}[i] = 1$. Instancier ces variables à 1 ne viole pas la contrainte ATMOSTSEQ. Par conséquent, il existe un support. □

Le lemme 3.1 et le corollaire 3.1 fournissent une procédure polynomiale de recherche de support pour la contrainte ATMOSTSEQCARD. En effet, la complexité dans le pire cas de l'algorithme 1 est en $O(nq)$ : il comporte $n$ étapes et pour chacune, les lignes 2, 3 et 4 nécessitent $O(q)$ opérations. Ainsi, pour chaque variable $x_i$, un suport pour $x_i = 0$ ou $x_i = 1$ peut être obtenu en $O(nq)$.

Donc, on a une procédure d'AC avec une complexité au pire en $O(n^2 q)$.

### 3.1 Filtrage des domaines

Cette partie montre que l'on peut filtrer toutes les valeurs inconsistantes pour la contrainte AtMostSeqCard avec seulement deux appels à l'algorithme 1, c'est à dire avec la même complexité dans le pire cas.

Soit $w[a:b]$ la projection de l'instanciation $w$ sur la sous-séquence $x_a, \ldots, x_b$.

En exécutant l'algorithme `leftmost` de gauche à droite et de droite à gauche sur la séquence $x_1, \ldots, x_n$, on obtient les valeurs de $|\overrightarrow{w}[1:i]|$ et de $|\overleftarrow{w}[i:n]|$ pour tout $i \in [1, \ldots, n]$. Ceci peut être exploité pour assurer l'arc-consistance de AtMostSeqCard. Tout d'abord, on montre qu'il n'y a besoin de ne considérer que le cas où la cardinalité $|\overrightarrow{w}|$ du vecteur calculé par `leftmost` est exactement égale à $d$ ; dans les autres cas la contrainte est soit valide soit insatisfiable.

**Proposition 3.1** *Soit $\overrightarrow{w}$ le vecteur calculé par `leftmost`, si les trois propositions suivantes sont vérifiées :*

$$\text{AtMostSeq}(u, q, [x_1, \ldots, x_n]) \text{ is AC} \tag{3.4}$$

$$\sum_{i=1}^{n} min(x_i) \leq d \tag{3.5}$$

$$|\overrightarrow{w}| > d \tag{3.6}$$

*alors* $\text{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n])$ *est* AC.

*Preuve.* Avec le corollaire 3.1 on sait que $\text{AtMostSeqCard}(u, q, d + 1, [x_1, \ldots, x_n])$ est satisfiable. Soit $w$ une instanciation satisfiable, et considérons sans perte de généralité, une variable $x_i$ telle que $|\mathcal{D}(x_i)| > 1$. Supposons tout d'abord que $w[i] = 1$. La solution $w'$ égale à $w$ sauf pour $w'[i] = 0$ satisfait $\text{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n])$ En effet, $|w'| = |w| - 1 = d$ et comme $\text{AtMostSeq}(u, q, [x_1, \ldots, x_n])$ est satisfaite par $w$, elle est satisfaite par $w'$. Ainsi, pour toute variable $x_i$, il existe un support pour $x_i = 0$.

Supposons que $w[i] = 0$, et soit $a < i$ (resp. $b > i$) le plus grand indice (resp. le plus petit) tel que $w[a] = 1$ et $\mathcal{D}(x_a) = \{0, 1\}$ (resp. $w[b] = 1$ et $\mathcal{D}(x_b) = \{0, 1\}$). Soit $w'$ une instanciation telle que $w'[i] = 1$, $w'[a] = 0$, $w'[b] = 0$, et $w = w'$ d'autre part. On a $|w'| = d$, et on peut montrer qu'elle satisfait $\text{AtMostSeq}(u, q, [x_1, \ldots, x_n])$. Considérons une sous-séquence $x_j, \ldots, x_{j+q-1}$. Si $j+q \leq i$ ou $j > i$ alors $\sum_{l=j}^{j+q-1} w'[l] \leq \sum_{l=j}^{j+q-1} w[l] \leq u$, ainsi seuls les indices tels que $j \leq i < j + q$ sont à prendre en compte. Il y a deux cas :

1. Soit $a$ ou $b$ ou les deux sont dans la sous-séquence ($j \leq a < j + q$ ou $j \leq b < j + q$). Dans ce cas, $\sum_{l=j}^{i+q-1} w'[l] \leq \sum_{l=j}^{i+q-1} w[l]$.
2. Ni $a$ ni $b$ ne sont dans la sous-séquence ($a < j$ et $j + q \leq b$). Dans ce cas, comme $\mathcal{D}(x_i) = \{0, 1\}$ et comme la condition 3.4 est vérifiée, on a $\sum_{l=j}^{i+q-1} \min(x_l) < u$. De plus, comme $a < j$ et $j + q \leq b$, il n'y a pas de variable $x_l$ dans cette sous-séquence telle que $w[l] = 1$ et $0 \in \mathcal{D}(x_l)$. Ainsi, on a $\sum_{l=j}^{i+q-1} w[l] < u$, c'est à dire $\sum_{l=j}^{i+q-1} w'[l] \leq u$.

Dans les deux cas, $w$ satisfait les contraintes de capacité. □

Rappelons que réaliser l'AC sur AtMostSeq est triviale car Among est monotone. On se concentre donc sur le cas où AtMostSeq est AC, et $|\overrightarrow{w}| = d$. Ainsi, les propositions 3.2, 3.3, 3.4 et 3.5 suivantes ne s'appliquent que dans ce cas. L'égalité $|\overrightarrow{w}| = d$ est donc implicite dans chacune d'entre elles.

**Proposition 3.2** *Si $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| < d$ alors $x_i = 0$ n'est pas* AC.

*Preuve.* Supposons que l'on a $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| < d$ et qu'il existe une instanciation $w'$ tel que $w'[i] = 0$ et $|w'| = d$.

En appliquant le Lemme 3.1 sur la séquence $x_1, \ldots, x_{i-1}$, on a $\sum_{l=1}^{i-1} w'[l] \leq |\overrightarrow{w}[1:i-1]|$.

En appliquant le Lemme 3.1 sur la séquence $x_n, \ldots, x_{i+1}$, on a $\sum_{l=i+1}^{n} w'[l] \leq |\overrightarrow{w}[i+1:n]|$.

Comme $w'[i] = 0$, on a $|w'| = \sum_{l=1}^{n} w'[l] < d$, ce qui est en condradiction avec l'hypothèse $|w'| = d$. Par conséquence, il n'y a pas de support pour $x_i = 0$. □

**Proposition 3.3** *Si $|\overrightarrow{w}[1:i]| + |\overleftarrow{w}[i:n]| \leq d$ alors $x_i = 1$ n'est pas* AC.

*Preuve.* Supposons que l'on a $|\overrightarrow{w}[1:i]| + |\overleftarrow{w}[i:n]| \leq d$ et qu'il existe une instanciation $w'$ tel que $w'[i] = 1$ et $|w'| = d$.

Par application du Lemme 3.1 sur la séquence $x_1, \ldots, x_i$, on a $\sum_{l=1}^{i} w'[l] \leq |\overrightarrow{w}[1:i]|$.

Par application du Lemme 3.1 sur la séquence $x_n, \ldots, x_i$, on a $\sum_{l=i}^{n} w'[l] \leq |\overleftarrow{w}[i:n]|$.

Donc, comme $w'[i] = 1$, on a $|w'| = \sum_{l=1}^{i} w'[l] + \sum_{l=i}^{n} w'[l] - 1 < d$, ce qui contredit l'hypothèse $|w'| = d$. Par conséquent, il n'y a pas de support pour $x_i = 1$. □

Les propositions 3.2 et 3.3 définissent une règle de filtrage. Dans une première passe, de la gauche vers la droite, on peut utiliser un algorithme similaire à `leftmost` pour calculer l'instanciation $|\overrightarrow{w}[1:i]|$ pour tout $i \in [1, \ldots, n]$. Dans une seconde passe, l'instanciation $|\overleftarrow{w}[i:n]|$ pour tout $i \in [1, \ldots, n]$ est déterminée en déroulant la même procédure sur la même séquence de variables mais *en sens inverse*, ie. de la droite vers la gauche. En utilisant ces instanciations, on peut utiliser alors les propositions 3.2 et 3.3 pour filtrer respectivement les valeurs 0 et 1. Cette procédure est détaillée ci-après.

On montre tout d'abord que ces deux règles sont complètes, c'est à dire si AtMostSeq est AC et si la contrainte

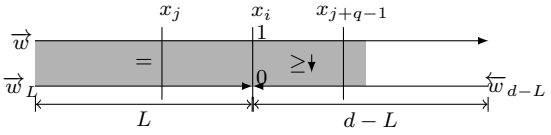


FIGURE 1: Illustration de la preuve de la Proposition 3.4. Les flêches horizontales représentent les instanciations.

de cardinalité globale est AC alors une instanciation $x_i = 0$ (resp. $x_i = 1$) est inconsistante ssi la proposition 3.2 (resp. 3.3) s'applique. Soit $\texttt{leftmost}(k)$ l'algorithme correspondant à l'application de `leftmost` s'arrêtant lorsque la cardinalité d'une instanciation atteint une valeur $k$ donnée.

**Lemme 3.2** *Soit $w$ une instanciation satisfiable pour* ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$. *Si $k \leq |w|$ alors l'instanciation $\overrightarrow{w}_k$ déterminée par $\texttt{leftmost}(k)$ est telle que pour tout $1 \leq i \leq n$ :*

$$\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$$

*Preuve.* Il est clair que pour $i = 1$, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. Pour $i \neq 1$, soit $m$ l'indice pour lequel $\texttt{leftmost}(k)$ s'arrête. On distingue deux cas. Lorsque $i > m$, pour toute valeur $l$ dans $[m+1, \ldots, n]$, on a $\overrightarrow{w}_k[l] \leq w[l]$ (car $\overrightarrow{w}_k[l] = \min(x_l)$), donc $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. Lorsque $i \leq m$, puisque $|\overrightarrow{w}_k| \leq |w|$ alors $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq |w| - \sum_{l=1}^{i-1} \overrightarrow{w}_k[l]$. Ainsi, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l] + (\sum_{l=1}^{i-1} w[l] - \sum_{l=1}^{i-1} \overrightarrow{w}_k[l])$. De plus, en appliquant le lemme 3.1, la valeur $\sum_{l=1}^{i-1} \overrightarrow{w}_k[l]$ est maximale, donc supérieure ou égale à $\sum_{l=1}^{i-1} w[l]$. Par conséquent, $\sum_{l=i}^{n} \overrightarrow{w}_k[l] \leq \sum_{l=i}^{n} w[l]$. □

**Proposition 3.4** *Si* ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$ *est* AC *et si $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| \geq d$ alors $x_i = 0$ possède un support.*

*Preuve.* Soit $\overrightarrow{w}$ une instanciation déterminée par `leftmost`. On considère, sans perte de généralité, une variable $x_i$ telle que $\mathcal{D}(x_i) = \{0, 1\}$ et $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| \geq d$ et on montre que l'on peut construire un support pour $x_i = 0$. Si $\overrightarrow{w}[i] = 0$ ou $\overleftarrow{w}[i] = 0$ alors il existe un support pour $x_i = 0$, donc il n'y a besoin de ne considérer que le cas $\overrightarrow{w}[i] = \overleftarrow{w}[i] = 1$.

Soient $L = |\overrightarrow{w}[1:i-1]|$ et $\overleftarrow{w}_{d-L}$ le résultat de $\texttt{leftmost}(d-L)$ sur la sous-séquence $x_n, \ldots, x_i$. On veut montrer que $w$ correspondant à la concaténation de $\overrightarrow{w}[1:i-1]$ et de $\overleftarrow{w}_{d-L}[i:n]$ est un suport pour $x_i = 0$.

Tout d'abord, on montre que $w[i] = 0$, c'est à dire $\overleftarrow{w}_{d-L}[i] = 0$. Par hypothèse, comme $|\overrightarrow{w}[1:i-1]| + |\overleftarrow{w}[i+1:n]| \geq d$, on a $|\overleftarrow{w}[i+1:n]| \geq d - L$. Considérons maintenant la séquence $x_i, \ldots, x_n$, et soit $w'$ le vecteur tel que $w'[i] = 0$ et $w' = \overleftarrow{w}[i+1:n]$ dans les autres cas. Comme $|w'| = |\overleftarrow{w}[i+1:n]| \geq d - L$, avec le lemme 3.2 on sait que $w'$ a une cardinalité plus grande que $\overleftarrow{w}_{d-L}$ sur toute sous-séquence débutant à $i$, donc $w[i] = \overleftarrow{w}_{d-L}[i] = w'[i] = 0$.

On montre maintenant que $w$ ne viole pas la contrainte ATMOSTSEQCARD. De manière évidente, comme cette instanciation est la concaténation de deux instanciations consistantes, elle ne peut violer la contrainte qu'à la jonction, i.e. sur une sous-séquence $x_j, \ldots, x_{j+q-1}$ telle que $j \leq i$ et $i < j + q$.

On montre que la somme de toute sous-séquence est inférieure ou égale à $u$ en les comparant avec $\overrightarrow{w}$, comme illustré dans la Figure 1. On a $\sum_{l=j}^{j+q-1} \overrightarrow{w}[l] \leq u$ et $\sum_{l=j}^{i-1} \overrightarrow{w}[l] = \sum_{l=j}^{i-1} w[l]$. De plus, avec le lemme 3.2 comme $|\overrightarrow{w}[i:n]| = |\overleftarrow{w}_{d-L}| = d - L$, on a $\sum_{l=i}^{j+q-1} \overleftarrow{w}_{d-L}[l] \leq \sum_{l=i}^{j+q-1} \overrightarrow{w}[l]$, donc $\sum_{l=i}^{j+q-1} w[l] \leq \sum_{l=i}^{j+q-1} \overrightarrow{w}[l]$. Par conséquent, on conclue que $\sum_{l=j}^{j+q-1} w[l] \leq u$. □

**Proposition 3.5** *Si* ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$ *est* AC *et si $|w[1:i]| + |w[n:i]| > d$ alors il y a un support pour $x_i = 1$.*

*Preuve.* Soient $\overrightarrow{w}$ et $\overleftarrow{w}$ deux instanciations déterminées par `leftmost` sur respectivement $x_1, \ldots, x_n$ et $x_n, \ldots, x_1$. On considère sans perte de généralité une variable $x_i$ telle que $\mathcal{D}(x_i) = \{0, 1\}$ et $|\overrightarrow{w}[1:i]| + |\overleftarrow{w}[i:n]| > d$ et on montre que l'on peut construire un support pour $x_i = 1$. Si $\overrightarrow{w}[i] = 1$ ou $\overleftarrow{w}[i] = 1$ alors il existe un support pour $x_i = 1$, donc il n'y a besoin de ne considérer que le cas $\overrightarrow{w}[i] = \overleftarrow{w}[i] = 0$.

Soit $L = |\overrightarrow{w}[1:i]| = |\overrightarrow{w}[1:i-1]|$ (cette égalité est vérifiée car $\overrightarrow{w}[i] = 0$). Soit $\overrightarrow{w}_{L-1}$ une instanciation obtenue en utilisant $\texttt{leftmost}(L-1)$ sur la sous-séquence $x_1, \ldots, x_{i-1}$ et soit $\overleftarrow{w}_{d-L}$ une instanciation obtenue par $\texttt{leftmost}(d-L)$ sur la sous-séquence $x_n, \ldots, x_{i+1}$.

On montre que $w$ vallant $w[i] = 1$, $\overrightarrow{w}_{L-1}$ sur $x_1, \ldots, x_{i-1}$ et $\overleftarrow{w}_{d-L}$ sur $x_{i+1}, \ldots, x_n$, est un support.

De manière évidente, $|w| = d$, ainsi il reste à vérifier que seules les contraintes de capacité sont satisfaites. De plus, comme cette instanciation est la concaténation de deux instanciations consistantes, elle peut violer la contrainte qu'à la jonction, i.e., pour une sous-séquence $x_j, \ldots, x_{j+q-1}$ telle que $j \leq i$ et $i < j + q$.

On montre que la somme de toute sous-séquence est inférieure ou égale à $u$ en la comparant avec $\overrightarrow{w}$ et

$\overleftarrow{w}_{d-L}$. Tout d'abord, notons que sur la sous-séquence $x_1, \ldots, x_{i-1}$, $\overrightarrow{w}_{L-1} = \overrightarrow{w}$, sauf pour le plus grand indice $a$ tel que $\overrightarrow{w}[a] = 1$ et $w[a] = 0$. De la même façon, sur $x_n, \ldots, x_{i+1}$, on a $\overleftarrow{w}_{d-L} = \overleftarrow{w}_{d-L+1}$, sauf pour le plus petit indice $b$ tel que $\overleftarrow{w}_{d-L+1}[b] = 1$. On distingue deux cas. Si $j > a$, alors $\sum_{l=j}^{j+q-1} w[l] = \sum_{l=i}^{j+q-1} \overleftarrow{w}_{d-L+1}[l]$ si $j + q - 1 \geq b$, et vaut 1 autrement. Et donc inférieur ou égal à $u$ car $i \geq j$ et $u \geq 1$. Si $j \leq a$. Dans ce cas, considérons la sous-séquence $x_j, \ldots, x_i$. Sur cet intervalle, la cardinalité de $w$ est la même que celle de $\overrightarrow{w}$, i.e., $\sum_{l=j}^{i} w[l] = \sum_{l=j}^{i-1} \overrightarrow{w}_{L-1}[l] + 1 = \sum_{l=j}^{i} \overrightarrow{w}[l]$. Sur la sous-séquence $x_{i+1}, \ldots, x_{j+q-1}$, notons que $|w[i+1:n]| = |\overrightarrow{w}[i+1:n]| = d - L$, donc par le lemme 3.2, on a $\sum_{l=i+1}^{j+q-1} w[l] = \sum_{l=i+1}^{j+q-1} \overleftarrow{w}_{d-L}[l] \leq \sum_{l=i+1}^{j+q-1} \overrightarrow{w}[l]$. Par conséquent, $\sum_{l=j}^{j+q-1} w[l] \leq \sum_{l=j}^{j+q-1} \overrightarrow{w}[l] \leq u$. □

### 3.2 Complexité

Avec les propositions 3.2, 3.3, 3.4 et 3.5 on peut concevoir un algorithme de filtrage avec la même complexité au pire que `leftmost`. Dans cette section, une impleméntation en temps linéaire de `leftmost` est proposée, appelée `leftmost_count`, qui retourne les valeurs de $\overrightarrow{w}[1:i]$ pour tout $i$ (Algorithme 2).

**Algorithm 2:** `leftmost_count`

```
Data: [x_1, ..., x_n], u, q, d,
Result: count : [0, ..., n] ↦ [0, ..., n − 1]
foreach i ∈ [1, ..., n] do
    w[i] ← min(x_i);
    occ[i] = 0;
foreach i ∈ [0, ..., n] do count[i] ← 0;
c[0] ← w[1];
foreach i ∈ [1, ..., u + 1] do occ[n + i] = 0;
foreach i ∈ [1, ..., q] do
    w[n + i] ← 0;
    c[i] ← c[i − 1] + w[i + 1];
    occ[n + c[i − 1]] ← occ[n + c[i − 1]] + 1;
max_c ← max({c[i] | i ∈ [1, ..., q]});
foreach i ∈ [1, ..., n] do
1       if max_c < u & |D(x_i)| > 1 then
            max_c ← max_c + 1;
            count[i] ← count[i − 1] + 1;
            w[i] ← 1;
        else
            count[i] ← count[i − 1];
        prev ← c[i − 1 mod q];
        next ← c[i + q − 2 mod q] + w[i + q] − w[i];
        c[i − 1 mod q] ← next;
        if prev ≠ next then
            occ[n + next] ← occ[n + next] + 1;
            if next + count > max_c then
                max_c ← max_c + 1;
            occ[n + prev] ← occ[n + prev] − 1;
            if occ[n + prev] = 0 & prev + count = max_c then
                max_c ← max_c − 1;
return count;
```

Il est simple de voir que `leftmost_count` a une complexité au pire en $O(n)$. Pour prouver sa validité, on montre qu'une instanciation déterminée par `leftmost_count` est identique à celle déterminée par `leftmost`.

*Preuve.* On donne ici l'idée générale de la preuve. Les trois invariants ci-dessous sont vrais à chaque étape $i$ de la boucle principale :

- la cardinalité de la $j^{ime}$ sous-séquence est donnée par $c[i + j - 2 \bmod q] + count[i - 1]$.
- le nombre de sous-séquences de cardinalité $k$ est donné par $occ[n - count[i-1] + k]$.
- la cardinalité maximale de toute sous-séquence est donnée par $max\_c$.

*$1^{ier}$ invariant :* La valeur de $c[j]$ est actualisée de deux façons dans `leftmost`. Tout d'abord, à chaque étape de la boucle, les valeurs de $c[1]$ à $c[q]$ sont décalées vers la gauche. Donc, il n'y a qu'une seule valeur nouvelle. En utilisant l'opération modulo, on peut mettre à jour seulement ces valeurs. Puis, lorsque $w[i]$ est fixé à 1, on incrémente $c[1]$ à $c[q]$. Comme cela se produit exactement $count[i-1]$ au début de l'étape $i$, on peut simplement ajouter $count[i-1]$ pour obtenir la même valeur que dans `leftmost`.

*$2^{ime}$ invariant :* La structure de données $occ$ est un tableau mémorisant, pour l'instanciation courante $w$, à l'indice $n + k$, le nombre de sous-séquence incluant $x_i$ avec une cardinalité $k$ Donc, en décrémentant le pointeur vers le premier élément du tableau, on décale en pratique le tableau complet. Ici aussi, la valeur de $count[i-1]$, ou plutôt l'expression $(n - count[i-1])$, pointe exactement sur le début recherché dans le tableau.

*$3^{ime}$ invariant :* La cardinalité maximale (ou minimale) des sous-séquences incluant $x_i$, peut changer au plus d'une unité au cours de chaque étape. Donc, quand la variable $max\_c$ doit changer, elle peut être juste incrémentée (si $occ[n - count[i-1] + max\_c + 1]$ passe de 0 à 1) ou décrémentée (si $occ[n - count[i-1] + max\_c]$ passe de 1 à 0). □

**Algorithm 3:** AC(ATMOSTSEQCARD)

```
Data: [x_1, ..., x_n], u, q, d
Result: AC on AtMostSeqCard(u, q, d, [x_1, .., x_n])
AC(AtMostSeq)(u, q, [x_1, ..., x_n]);
ub ← d − Σ_{i=1}^{n} min(x_i);
L ← leftmost_count([x_1, ..., x_n], u, q, d);
if L[n] = ub then
    R ← leftmost_count([x_n, ..., x_1], u, q, d);
    foreach i ∈ [1, ..., n] such that D(x_i) = {0, 1} do
        if L[i] + R[n − i + 1] ≤ ub then D(x_i) ← {0};
        if L[i − 1] + R[n − i] ≤ ub then D(x_i) ← {1};
```

L'algorithme 3 détermine l'AC de la contrainte ATMOSTSEQCARD$(u, q, d, [x_1, \ldots, x_n])$. A la première ligne, l'AC de la contrainte ATMOSTSEQ$(u, q, [x_1, \ldots, x_n])$ est établie. Cela permet d'assurer que la règle de filtrage exposée dans ce papier est vérifiée. Pour des raisons de place, on ne donne pas le pseudo-code pour l'AC de ATMOSTSEQ mais elle peut être réalisée en temps linéaire en utilisant une procédure

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{D}(x_i)$ | | . | 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 | |

FIGURE 2: Exemple de déroulement de l'algorithme 3 pour $u = 4$, $q = 8$, $d = 12$. La première ligne représente les domaines courrants, les points représentent des variables non instanciées (donc $ub = 10$). Les deux lignes suivantes donnent les instanciations $\overrightarrow{w}$ et $\overleftarrow{w}$ obtenues par `leftmost_count` de gauche à droite et de droite à gauche. Les troisième et quatrième lignes donnent les valeurs de $L[i] = |\overrightarrow{w}[1:i]|$ ou $R[n-i+1] = |\overleftarrow{w}[i:n]|$. Les cinquième et sixième ligne correspondent à l'application des propositions 3.2 et 3.3 respectvement. La septième ligne donne l'arc-consistance des domaines

similaire à `leftmost_count`. On cherche à calculer, pour une instanciation correspondant à la borne inférieure de chaque domaine, si une variable non instanciée est couverte par une sous-séquence de cardinalité $u$ pour les bornes inférieures des domaines. Ceci est effectué avec une version tronquée de `leftmost_count` : les valeur de $w[i]$ ne sont jamais mises à jour, i.e., elles sont fixées à leur valeur minimale et on n'entre pas dans le bloc « if-then-else » (condition 1 de l'Algorithme 2). En plus, on mémorise dans un tableau la valeur de $max\_c$ pour tout $i$. Ce tableau est parcouru pour effecter la valeur 0 à chaque variable non instanciée couverte par au moins une sous-séquence de cardinalité $u$ afin de réaliser l'AC de ATMOSTSEQ.

La suite est une application directe des propositions 3.2, 3.3, 3.4 et 3.5. Un exemple d'exécution est donné dans la Figure 2. La complexité au pire de l'Algorithme 3 est ainsi en $O(n)$, donc optimale.

# 4 Résultats expérimentaux

L'algorithme de propagation a été testé sur des instances de car-sequencing et de crew-rostering. Les expérimentations ont été effectuées sur un processeur Intel Xeon à 2.67GHz sous Linux. Les développements ont été faits avec Ilog-Solver. 5 exécutions aléatoires de chaque instance ont été lancées avec un temps limite de 20 minutes. [3]

Pour comparer l'efficacité relatives des propagations, les résultats sont moyennés sur plusieurs heuristiques pour réduire le biais que celles-ci pourraient introduire. Pour chaque ensemble de données, *#sol* représente le nombre d'instances résolues. Puis, on donne le temps CPU (*time*) en secondes et le nombre de retour-arrière (*avg bts*) (tous les deux moyennés sur le nombre de solutions obtenues, le nombre d'instances et d'heuristiques) ainsi que le nombre maximum de retour-arrière (*max bts*). Les résultats (en termes de *#sol*) de la meilleure méthode sont notés en gras.

3. pour un temps total de CPU d'environ 200 jours.

## 4.1 Car-sequencing

Pour le car-sequencing [8, 17], $n$ véhicules doivent être fabriqués sur une ligne d'assemblage tout en respectant des contraintes de capacités et de demande.

Nous nous sommes basés sur le modèle standard implémenté avec Ilog-Solver. Il y a $n$ variables entières représentant les classes de véhicules de chaque position dans la ligne d'assemblage et $nm$ variables booléennes $y_i^j$ représentant le fait que le véhicule placé en $i^{ime}$ position nécessite l'option $j$. La demande de chaque classe est exprimée avec une contrainte de cardinalité globale (GCC) et l'arc-consistance de cette contrainte est réalisée avec Ilog [14]. Quatre modélisations pour les contraintes de capacité sont comparées (disant que pour chaque option $j$, chaque sous-séquence de taille $q_j$ contient au plus $u_j$ variables d'option fixées à 1) associées aux contraintes de demande de chaque option (déduites des contraintes de demande sur chaque classe). La première modélisation (*sum*) comprend une simple décomposition en une séquence de contraintes somme pour les contraintes de capacité ainsi qu'une contrainte somme supplémentaire exprimant la demande. La seconde modélisation, (*gsc*) utilise une contrainte GSC par option. La troisième, (*amsc*) est une application de la procédure d'AC introduite dans cet article pour la contrainte ATMOSTSEQCARD. Enfin, la quatrième modélisation, (*amsc+gsc*) combine les contraintes ATMOSTSEQCARDet GSC.

34 heuristiques obtenues par combinaisons de différents paramètres ont été utilisées : *exploration* de la ligne d'assemblage soit de manière lexicographique soit depuis le milieu vers les bords ; *branchement* sur l'affectation d'une classe ou d'une option à une position dans la ligne ; *sélection* de la meilleure classe ou option à l'aide de différents critères évidents (demande maximum, ratio $u/q$ minimum) ou de critères plus complexes dérivés de ceux proposés dans [5, 16].

3 groupes d'instances de la CSPLib [9] ont été considérés. Le premier groupe, appelé set1 par la suite, comporte 70 instances à 200 véhicules, toutes satisfiables. Dans le

second groupe, il y a 9 instances de 100 véhicules réparties en 4 satisfiables, notées set2 et 5 instances insatisfiables, noté set3. Le troisième groupe contient 30 instances de plus grande taille (jusqu'à 400 véhicules), parmi elles, on a considéré les 7 instances connues pour être satisfiables, notées set4.

Les résultats sont présentés dans la Table 1. Dans tous les cas, la meilleure valeur en terme de nombre de problèmes résolus est obtenue soit avec *amsc+gsc* (pour les instances de petite taille ou insatisfiables) ou avec *amsc* seul (pour les instances de plus grande taille set2 et set4). Globalement, on note que GSC permet d'élaguer beaucoup plus de valeurs incohérentes que ATMOSTSEQCARD. Toutefois, la propagation de GSC ralentit très significativement la recherche (on a observé un facteur 12.5 sur le nombre de nœuds explorés par seconde). Par ailleurs, les niveaux de filtrage obtenus par ces deux méthodes sont incomparables. En conséquent combiner ces deux contraintes est toujours bénéfique plutôt que d'utiliser GSC seule.

Dans [18], les auteurs ont évalué leur proposition sur les instances set1, set2 et set3. Dans leurs expérimentations, ils ont considéré les meilleurs résultats obtenus pour 2 heuristiques ($\sigma$ et min domain). En propageant uniquement COST-REGULAR ou GEN-SEQUENCE $50.7\%$ des instances sont résolues alors qu'en les combinant avec GSC $65.2\%$ sont résolues (avec un temps maximum de 1h). Dans nos expérimentations, en moyenne sur les 34 heuristiques et sur les 5 exécutions aléatoires, ATMOSTSEQCARD et GSC résolvent respectivement $82.5\%$ et $86.11\%$ des instances et la combinaison des deux permet de résoudre $86.36\%$ des instances avec un temps CPU limite de 20 minutes.

### 4.2 Crew-rostering

Pour le crew-rostering, des périodes de travail doivent être allouées à des employés sur un horizon donné de telle sorte qu'il y ait suffisament de personnes pour assurer un service à chaque période et que les contraintes sur l'organisation du travail soit respectées. Cette dernière condition peut entraîner une grande variété de contraintes. Des travaux antérieurs [11, 13] s'appuient sur des motifs autorisés (ou interdits) pour exprimer des contraintes sur des périodes successives. Par exemple, s'il y a 3 périodes de 8 heures par jour : J (jour), S (soir) et N (nuit), NJ peut être interdit pour exprimer le fait que les employés ont besoin d'un repos après une période de travail de nuit. Cet article considère un cas simple avec 20 employés, 3 périodes de travail de 8 heures par jour sur lesquelles on ne peut travailler plus de 8 heures par jour, pas plus de 5 jours par semaines et que les pauses entre deux périodes de travail doivent être d'au minimum 16 heures. L'horizon est de 28 jours, chaque employé doit travailler en moyenne 34 heures par semaine (17 périodes sur 4 semaines).

La modélisation se base sur une variable booléenne $e_{ij}$ pour chacun des $m$ employés et des $n$ périodes traduisant le fait qu'un employé $i$ travaille en période $j$. La demande $d^s_j$ de chaque période $j$ est propagée avec une contrainte de somme : $\sum_{i=1}^{m} e_{ij} = d^s_j$.[4] Les autres contraintes sont posées en utilisant deux contraintes ATMOSTSEQCARD par employé : l'une avec le ratio $u/q = 1/3$, l'autre avec le ratio $5/21$, les deux contraintes ont la même demande $d = 17$ correspondant à 34 heures de travail par semaine. Trois modélisations sont comparées. Dans la première, (*sum*), ces deux contraintes sont décomposées avec une séquence de contraintes de somme. La seconde, (*gsc*) utilise une contrainte GSC. Notons que dans ce cas, comme les variables sont booléenne, la contrainte GCC de GSC n'est rien d'autre qu'une contrainte de somme. Donc, elle ne peut pas élaguer plus que la contrainte ATMOSTSEQCARD (mais elle est plus puissante que la décomposition en somme). La troisième modélisation, (*amsc*) emploie la contrainte ATMOSTSEQCARD présentée dans cet article.

Les demandes sur l'horizon sont régulières sur les 4 semaines : pour les périodes de jour et de soir, 6 employés sont nécessaires en semaine et 2 pour le week-end ; sur les périodes de nuit 3 employés sont nécessaires en semaine et seulement 1 en week end. De plus, un ensemble d'indisponibilités a été générées aléatoirement pour chaque employé sur les différentes période. 281 instances ont été générées, avec des indisponibilités de 18% à 46% par pas de 0.1. Ces instances peuvent être partitionnées en trois groupes : le premier groupe comporte 126 instances avec peu d'indisponibilités (toutes les instances sont satisfiables), le troisième groupe est composé de 44 instances (principalement insatisfiables) avec un fort ratio d'indisponibilité, le reste des instances constituent le second groupe.

L'heuristique de branchement utilisée est la suivante : pour chaque employé $i$ on sélectionne celui ayant un slack minimal $\sigma_i = n_i - \frac{21 d_i}{5}$ et on l'affecte à une période possible ayant le slack minimum $\sigma'_j = m_j - d^s_j$, où $n_i$ (resp. $m_j$) représente le nombre de variables $e_{ij}$ (resp. $e_{kj}, k \in \{1..20\}$) non instanciées et $d_i$ (resp. $d^s_j$) le nombre résiduel de périodes devant être travaillées par l'employé $i$ (resp. d'employés devant être affectés à une période $j$). De plus, on utilise une seconde heuristique s'appuyant sur les mêmes principes mais sélectionnant d'abord la période puis l'employé.

Les résultats obtenus sont données dans la Table 2. L'algorithme d'AC proposé réalise plus de filtrage que la décomposition en contraintes somme et que la contrainte GSC. Cependant, selon les heuristiques et les exécutions aléatoires, la taille de l'arbre de recherche n'est pas toujours la plus petite. On observe que notre filtrage ATMOSTSEQCARD n'est que marginalement plus lent en

4. Les instances générées ont exactement autant d'employés que nécessaire pour assurer le service total, on peut ainsi utiliser une contrainte d'égalité.

TABLE 1: Evaluation des méthodes de filtrage (en moyenne sur toutes les heuristiques)

| Méthodes | set1 ($70 \times 34 \times 5$) | | | | set2 ($4 \times 34 \times 5$) | | | | set3 ($5 \times 34 \times 5$) | | | | set4 ($7 \times 34 \times 5$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #sol | avg bts | max bts | time | #sol | avg bts | max bts | time | #sol | avg bts | max bts | time | #sol | avg bts | max bts | time |
| sum | 8480 | 231.2K | 25.0M | 13.93 | 95 | 1.4M | 15.3M | 76.60 | 0 | - | - | $> 1200$ | 64 | 543.3K | 13.7M | 43.81 |
| gsc | 11218 | 1.7K | 2.3M | 3.60 | 325 | 131.7K | 1.5M | 110.99 | 31 | 55.3K | 218.5K | 276.06 | 140 | 25.2K | 321.9K | 56.61 |
| amsc | 10702 | 39.1K | 13.8M | 4.43 | **360** | **690.8K** | **10.2M** | **72.00** | 16 | 40.3K | 83.4K | 8.62 | **153** | **201.4K** | **3.2M** | **33.56** |
| amsc+gsc | **11243** | **1.2K** | **1.1M** | **3.43** | 339 | 118.4K | 1.0M | 106.53 | **32** | **57.7K** | **244.7K** | **285.43** | 147 | 23.8K | 371.0K | 66.45 |

TABLE 2: Evaluation des méthodes de filtrage (en moyenne sur toutes les heuristiques)

| Benchmarks | sous-contraints ($5 \times 2 \times 126$) | | | | difficiles ($5 \times 2 \times 111$) | | | | sur-contraints ($5 \times 2 \times 44$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #sol | avg bts | max bts | time | #sol | avg bts | max bts | time | #sol | avg bts | max bts | time |
| sum | 1229 | 110.5K | 10.1M | 12.72 | 574 | 370.7K | 13.4M | 38.45 | 272 | 52.1K | 5.7M | 5.56 |
| gsc | 1210 | 6.2K | 297.2K | 29.19 | 579 | 23.5K | 433.5K | 77.78 | 276 | 7.7K | 378.9 | 24.14 |
| amsc | **1237** | **34.2K** | **7.5M** | **5.82** | **670** | **213.4K** | **7.5M** | **31.01** | **284** | **51.3** | **7.5M** | **6.22** |

termes de nœuds explorés par seconde que la décomposition en somme et beaucoup plus rapide (avec un facteur 20.4) que GSC. Il peut résoudre 15.7% et 16.7% de problèmes en plus (dans le temps limite de 20 minutes) parmi les instance les plus difficiles que la contrainte GSC et la décomposition en somme, respectivement.

## 5 Conclusion

Cet article propose un algorithme linéaire pour réaliser l'arc-consistance de la contrainte ATMOSTSEQCARD. Les évaluations expérimentales, sur des problèmes de car-sequencing et de crew-rostering, ont montré que cette contrainte pouvait être utile pour ces applications.

**Acknowledgements :** The authors would like to thank Nina Narodytska for her useful comments.

## Références

[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 Card-path Constraint. In *ECAI*, pages 475–479, 2008.

[5] S. Boivin, M. Gravel, M. Krajecki, and C. Gagné. Résolution du Problème de Car-sequencing à l'Aide d'une Approche de Type FC. In *JFPC*, 2005.

[6] S. Brand, N. Narodytska, C.-G. Quimper, P. J. Stuckey, and T. Walsh. Encodings of the Sequence Constraint. In *CP*, pages 210–224, 2007.

[7] S. Demassey, G. Pesant, and L.-M. Rousseau. A Cost-Regular Based Hybrid Column Generation Approach. *Constraints*, 11(4) :315–333, 2006.

[8] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI*, pages 290–295, 1988.

[9] I. P. Gent and T. Walsh. Csplib : a benchmark library for constraints, 1999.

[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] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP*, pages 482–495, 2004.

[13] G. Pesant. Constraint-Based Rostering. In *PATAT*, 2008.

[14] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *AAAI*, pages 209–215(2), 1996.

[15] J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *CP*, pages 32–46, 1997.

[16] B.M. Smith. Succeed-first or Fail-first : A Case Study in Variable and Value Ordering, 1996.

[17] 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.

[18] 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, June 2009.

[19] W. J. van Hoeve, G. Pesant, L.-M. Rousseau, and Ashish Sabharwal. Revisiting the Sequence Constraint. In *CP*, pages 620–634, 2006.