On Matrices, Automata, and Double Counting

Nicolas Beldiceanu[1], Mats Carlsson[2], Pierre Flener[3], and Justin Pearson[3]

[1] Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France
Nicolas.Beldiceanu@emn.fr

[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

[3] Uppsala University, Department of Information Technology, Box 337, SE-751 05 Sweden
Pierre.Flener@it.uu.se, Justin.Pearson@it.uu.se

Abstract. Matrix models are ubiquitous for constraint problems. Many such problems have a matrix of variables $\mathcal{M}$, with the same constraint defined by a finite-state automaton $\mathcal{A}$ on each row of $\mathcal{M}$ and a global cardinality constraint *gcc* on each column of $\mathcal{M}$. We give two methods for deriving, by double counting, necessary conditions on the cardinality variables of the *gcc* constraints from the automaton $\mathcal{A}$. The first method yields linear necessary conditions and simple arithmetic constraints. The second method introduces the *cardinality automaton*, which abstracts the overall behaviour of all the row automata and can be encoded by a set of linear constraints. We evaluate the impact of our methods on a large set of nurse rostering problem instances.

1 Introduction

Several authors have shown that matrix models are ubiquitous for constraint problems. Despite this fact, only a few constraints that consider a matrix and some of its constraints as a whole have been considered: the $allperm$ [8] and $lex2$ [7] constraints were introduced for breaking symmetries in a matrix, while the $colored\_matrix$ constraint [13] was introduced for handling a conjunction of *gcc* constraints on the rows and columns of a matrix. We focus on another recurring pattern, especially in the context of personnel rostering, which can be described in the following way.

Given three positive integers R, K, and V, we have an $R \times K$ matrix $\mathcal{M}$ of decision variables that take their values within the finite set of values $\{0, 1, \ldots, V-1\}$, as well as a $V \times K$ matrix $\mathcal{M}^{\#}$ of cardinality variables that take their values within the finite set of values $\{0, 1, \ldots, R\}$. Each row r (with $0 \leq r < R$) of $\mathcal{M}$ is subject to a constraint defined by a finite-state automaton $\mathcal{A}$ [2,12]. For simplicity, we assume that each row is subject to the same constraint. Each column k (with $0 \leq k < K$) of $\mathcal{M}$ is subject to a *gcc* constraint that restricts the number of occurrences of the values according to column k of $\mathcal{M}^{\#}$: let $\#_k^v$ denote the number of occurrences of value v (with $0 \leq v < V$) in column k of $\mathcal{M}$, that is, the cardinality variable in row v and column k of $\mathcal{M}^{\#}$. We call this pattern the *matrix-of-automata-and-gcc* pattern. In the context of personnel rostering, a possible interpretation of this pattern is:

– R, K, and V respectively correspond to the number of persons, days, and types of work (e.g., *morning shift*, *afternoon shift*, *night shift*, or *day off*) we consider.

- Each row r of $\mathcal{M}$ corresponds to the work of person r over K consecutive days.
- Each column k of $\mathcal{M}$ corresponds to the work by the R persons on day k.
- The automaton $\mathcal{A}$ on the rows of $\mathcal{M}$ encodes the rules of a valid schedule for a person; it can be the product of several automata defining different rules.
- The *gcc* constraint on column k represents the demand of services for day k. In this context, the cardinality associated with a given service can either be fixed or be specified to belong to a given range.

A typical problem with this kind of pattern is the lack of interaction between the row and column constraints. This is especially problematic when, on the one hand, the row constraint is a sliding constraint expressing a distribution rule on the work, and, on the other hand, the demand profile (expressed with the *gcc* constraints) varies drastically from one day to the next (e.g., during weekends and holidays in the context of personnel rostering). This issue is usually addressed by experienced constraint programmers by manually adding necessary conditions (implied constraints) that are most of the time based on some simple counting conditions depending on some specificity of the row constraints. Let us first introduce a toy example to illustrate this phenomenon.

Example 1. Take a 3×7 matrix $\mathcal{M}$ of 0/1 variables (i.e., $R = 3, K = 7, V = 2$), where on each row we have a *global_contiguity* constraint (all the occurrences of value 1 are contiguous) for which Figure 1 depicts a corresponding automaton (the reader can ignore the assignments to counters c and d at this moment). In addition, $\mathcal{M}^{\#}$ defines the following *gcc* constraints on the columns of $\mathcal{M}$:

- Columns 0, 2, 4, and 6 of $\mathcal{M}$ must each contain two 0s and a single 1.
- Columns 1, 3, and 5 of $\mathcal{M}$ must each contain two 1s and a single 0.

A simple double counting argument proves that there is no solution to this problem. Indeed, consider the sequence of numbers of occurrences of 1s on the seven columns of $\mathcal{M}$, that is 1, 2, 1, 2, 1, 2, 1. Each time there is an increase of the number of 1s between two adjacent columns, a new group of consecutive 1s starts on at least one row of the matrix. From this observation we can deduce that we have at least four groups of consecutive ones, namely one group starts at the first column (since implicitly before the first column we have zero occurrences of value 1) and three groups start at the columns

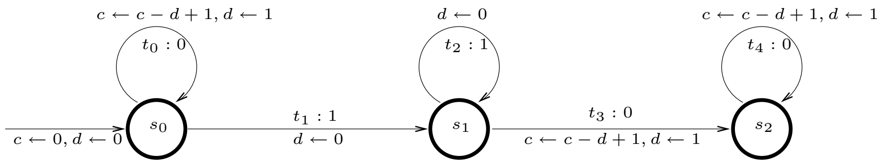
$c \leftarrow c - d + 1, d \leftarrow 1$
$t_0 : 0$
$d \leftarrow 0$
$t_2 : 1$
$c \leftarrow c - d + 1, d \leftarrow 1$
$t_4 : 0$
$c \leftarrow 0, d \leftarrow 0$
s_0
$t_1 : 1$
$d \leftarrow 0$
s_1
$t_3 : 0$
$c \leftarrow c - d + 1, d \leftarrow 1$
s_2

Fig. 1. Automaton associated with the *global_contiguity* constraint, with initial state s_0, final states s_0, s_1, s_2, and transitions t_0, t_1, t_2, t_3, t_4 labelled by values 0 or 1. The missing transition for value 1 from state s_2 is assumed to go to a dead state. The automaton has been annotated with counters [2]: the final value of counter c is the number of stretches of value 0, whereas d is an auxiliary counter.

containing two 1s. But since we have a $global\_contiguity$ constraint on each row of the matrix and since the matrix only has three rows, there is a contradiction.

The contributions of this paper include:

- Methods for deriving necessary conditions on the cardinality variables of the gcc constraints from string properties that hold for an automaton $\mathcal{A}$ (Sections 2.1 to 2.3).
- A method for annotating an automaton $\mathcal{A}$ with counter variables extracting string properties from $\mathcal{A}$ (Section 2.4).
- Another method for deriving necessary conditions on the cardinality variables, called the *cardinality automaton*, which simulates the overall behaviour of all the row automata (Section 3).
- An evaluation of the impact of our methods in terms of runtime and search effort on a large set of nurse rostering problem instances (Section 4).

Since our methods essentially generate linear constraints as necessary conditions, they may also be relevant in the context of linear programming.

2 Deriving Necessary Conditions from String Properties

We develop a first method for deriving necessary conditions for the *matrix-of-automata-and-gcc* pattern. The key idea is to approximate the set of solutions to the row constraint by string properties such as:

- Bounds on the number of letters, words, prefixes, or suffixes (see Section 2.1).
- Bounds on the number of stretches of a given value (see Section 2.2).
- Bounds on the lengths of stretches of a given value (see Section 2.3).

We first develop a set of formulae expressed in terms of simple arithmetic constraints for such string properties. Each formula gives a necessary condition for the *matrix-of-automata-and-gcc* pattern provided that the set of solutions of the row constraint satisfies a given string property. We then show how to extract automatically such string properties from an automaton (see Section 2.4) and outline a heuristic for selecting relevant string properties (see Section 2.5). String properties can also be seen as a communication channel for enhancing the propagation between row and column constraints.

In Sections 2.1 and 2.2, the derived constraints use the well-known combinatorial technique of *double counting* (see for example [9]). Here we use the two-dimensional structure of the matrix, counting along the rows and the columns. Some feature is considered, such as the number of appearances of a word or stretch, and the occurrences of that feature are counted for the rows and columns separately. When the counting is exact, these two values will coincide. In order to derive useful constraints that will propagate, we derive lower and upper bounds on the given feature occurring when counted columnwise. These are then combined into inequalities saying that the sum of these column-based lower bounds is *at most* the sum of given row-based upper bounds, or that the sum of these column-based upper bounds is *at least* the sum of given row-based lower bounds.

2.1 Constraining the Number of Occurrences of Words, Prefixes, and Suffixes

A *word* is a fixed sequence of values, seen as letters. Suppose we have the following bounds for each row on how many times a given word occurs (possibly in overlapping fashion) on that row, all numbering starting from zero:

- $LW_r(w)$ is the minimum number of times that the word w occurs on row r.
- $UW_r(w)$ is the maximum number of times that the word w occurs on row r.

Note that letters are just singleton words. It is not unusual that the $LW_r(w)$ (or $UW_r(w)$) are equal for all rows r for a given word w. From this information, we now infer by double counting two necessary conditions for each such word.

Necessary Conditions. Let $|w|$ denote the length of word w, and let w_j denote the jth letter of word w. The following bounds

$$lw_k(w) = \max\left(\left(\sum_{j=0}^{|w|-1} \#_{k+j}^{w_j}\right) - (|w|-1)\cdot R, 0\right) \tag{1}$$

$$uw_k(w) = \min_{j=0}^{|w|-1} \#_{k+j}^{w_j} \tag{2}$$

correspond respectively to the minimum and maximum number of occurrences of word w that start at column $k \in [0, K-|w|]$. These bounds can be obtained as follows:

- Since the cardinality variables only count the number of times a value occurs in each column and does not constrain *where* it occurs, the lower bound (1) is the worst-case intersection of all column value occurrences.
- A word cannot occur more often than its minimally occurring letter, hence bound (2).

Note that if some cardinality variable is not fixed, then the expressions above should be interpreted as arithmetic constraints. We get the following necessary conditions:

$$\sum_{k=0}^{K-|w|} lw_k(w) \leq \sum_{r=0}^{R-1} UW_r(w) \quad \text{(3a)} \qquad \sum_{k=0}^{K-|w|} uw_k(w) \geq \sum_{r=0}^{R-1} LW_r(w) \quad \text{(3b)}$$

Note that (3b) trivially holds when all $LW_r(w)$ are zero.

Generalisation: Replacing Each Letter by a Set of Letters. In the previous paragraph, all letters of the word w were fixed. We now consider that each letter of a word can be replaced by a finite non-empty set of possible letters. For this purpose, let w_j now denote the jth set of letters of word w. Hence the bounds $lw_k(w)$ and $uw_k(w)$ are now defined by aggregation as follows:

$$lw_k(w) = \max\left(\left(\sum_{j=0}^{|w|-1} \sum_{c\in w_j} \#_{k+j}^{c}\right) - (|w|-1)\cdot R, 0\right) \tag{4}$$

$$uw_k(w) = \min_{j=0}^{|w|-1} \left(\sum_{c\in w_j} \#_{k+j}^{c}\right) \tag{5}$$

We get the same necessary conditions as before. Note that (4) and (5) specialise respectively to (1) and (2) when all w_j are singleton sets.

Extension: Constraining Prefixes and Suffixes. We now consider constraints on a word occurring as a prefix (the first letter of the word is at the first position of the row) or suffix (the last letter of the word is at the last position of the row). Suppose we have the following bounds:

- $LWP_r(w)$ is the minimum number of times (0 or 1) word w is a prefix of row r.
- $UWP_r(w)$ is the maximum number of times (0 or 1) word w is a prefix of row r.
- $LWS_r(w)$ is the minimum number of times (0 or 1) word w is a suffix of row r.
- $UWS_r(w)$ is the maximum number of times (0 or 1) word w is a suffix of row r.

From these bounds, we get the following necessary conditions:

$$lw_0(w) \leq \sum_{r=0}^{R-1} UWP_r(w) \quad \text{(6a)} \qquad uw_0(w) \geq \sum_{r=0}^{R-1} LWP_r(w) \quad \text{(6b)}$$

$$lw_{K-|w|}(w) \leq \sum_{r=0}^{R-1} UWS_r(w) \quad \text{(7a)} \qquad uw_{K-|w|}(w) \geq \sum_{r=0}^{R-1} LWS_r(w) \quad \text{(7b)}$$

Note that (6b) trivially holds when all $LWP_r(w)$ are zero, and that (7b) trivially holds when all $LWS_r(w)$ are zero. Note that these necessary conditions also hold when each letter of a constrained prefix or suffix is replaced by a set of letters.

2.2 Constraining the Number of Occurrences of Stretches

Given a row r of fixed variables and a value v, a *stretch* of value v is a maximum sequence of values on row r that only consists of value v. Suppose now that we have bounds for each row on how many times a stretch of a given value v can occur on that row:

- $LS_r(v)$ is the minimum number of stretches of value v on row r.
- $US_r(v)$ is the maximum number of stretches of value v on row r.

It is not unusual that the $LS_r(v)$ (or $US_r(v)$) are equal for all rows r for a given value v.

Necessary Conditions. The following bounds (under the convention that $\#_{-1}^v = 0$ for each value v)

$$ls_k^+(v) = \max(0, \#_k^v - \#_{k-1}^v) \tag{8}$$
$$us_k^+(v) = \#_k^v - \max(0, \#_{k-1}^v + \#_k^v - R) \tag{9}$$

correspond respectively to the minimum and maximum number of stretches of value v that *start* at column k. Again, *if some cardinality variable is not fixed, then the expressions above should be interpreted as arithmetic constraints.* The intuitions behind these formulae are as follows:

– If the number of occurrences of value v on column k (i.e., $\#_k^v$) is strictly greater than the number of occurrences of value v on column $k-1$ (i.e., $\#_{k-1}^v$), then this means that at least $\#_k^v - \#_{k-1}^v$ new stretches of value v can start at column k.
– If the total of the number of occurrences of value v on column k (i.e., $\#_k^v$) and the number of occurrences of value v on column $k-1$ (i.e., $\#_{k-1}^v$) is strictly greater than the number of rows R, then the quantity $\#_{k-1}^v + \#_k^v - R$ represents the minimum number of stretches of value v that cover both column $k-1$ and column k. From this minimum intersection we get the maximum number of new stretches that can start at column k.

By aggregating these bounds for all the columns of the matrix, we get the following necessary conditions through double counting:

$$\sum_{k=0}^{K-1} ls_k^+(v) \leq \sum_{r=0}^{R-1} US_r(v) \qquad \text{(10a)} \qquad \sum_{k=0}^{K-1} us_k^+(v) \geq \sum_{r=0}^{R-1} LS_r(v) \qquad \text{(10b)}$$

Similarly, the following bounds (under the convention that $\#_K^v = 0$ for each value v)

$$ls_k^-(v) = \max(0, \#_k^v - \#_{k+1}^v) \tag{11}$$
$$us_k^-(v) = \#_k^v - \max(0, \#_{k+1}^v + \#_k^v - R) \tag{12}$$

correspond respectively to the minimum and maximum number of stretches of value v that *end* at column k. We get similar necessary conditions:

$$\sum_{k=0}^{K-1} ls_k^-(v) \leq \sum_{r=0}^{R-1} US_r(v) \qquad \text{(13a)} \qquad \sum_{k=0}^{K-1} us_k^-(v) \geq \sum_{r=0}^{R-1} LS_r(v) \qquad \text{(13b)}$$

Note that (10b) and (13b) trivially hold when all $LS_r(v)$ are zero.

Generalisation: Replacing the Value by a Set of Values. In the previous paragraph, the value v of a stretch was fixed. We now consider that a stretch may consist of a finite non-empty set, denoted by $\hat{v}$, of possible letters that are all considered equivalent. Let $\#_k^{\hat{v}}$ denote the quantity $\sum_{v\in\hat{v}}(\#_k^v)$, that is the total number of occurrences of the values of $\hat{v}$ in column k. The bounds (8), (9), (11), (12) are generalised as follows:

$$ls_k^+(\hat{v}) = \max(0, \#_k^{\hat{v}} - \#_{k-1}^{\hat{v}}) \tag{14}$$
$$us_k^+(\hat{v}) = \#_k^{\hat{v}} - \max(0, \#_{k-1}^{\hat{v}} + \#_k^{\hat{v}} - R) \tag{15}$$
$$ls_k^-(\hat{v}) = \max(0, \#_k^{\hat{v}} - \#_{k+1}^{\hat{v}}) \tag{16}$$
$$us_k^-(\hat{v}) = \#_k^{\hat{v}} - \max(0, \#_{k+1}^{\hat{v}} + \#_k^{\hat{v}} - R) \tag{17}$$

and we get the following necessary conditions:

$$\sum_{k=0}^{K-1} ls_k^+(\hat{v}) \leq \sum_{v\in\hat{v}} \sum_{r=0}^{R-1} US_r(v) \qquad \text{(18a)} \qquad \sum_{k=0}^{K-1} us_k^+(\hat{v}) \geq \sum_{v\in\hat{v}} \sum_{r=0}^{R-1} LS_r(v) \qquad \text{(18b)}$$

$$\sum_{k=0}^{K-1} ls_k^-(\hat{v}) \leq \sum_{v\in\hat{v}} \sum_{r=0}^{R-1} US_r(v) \quad \text{(19a)} \qquad \sum_{k=0}^{K-1} us_k^-(\hat{v}) \geq \sum_{v\in\hat{v}} \sum_{r=0}^{R-1} LS_r(v) \quad \text{(19b)}$$

Note that (18a), (18b), (19a), and (19b) specialise respectively to (10a), (10b), (13a), and (13b) when $\hat{v} = \{v\}$.

2.3 Constraining the Minimum and Maximum Length of a Stretch

Suppose now that we have lower and upper bounds on the length of a stretch of a given value v for each row:

- $LLS(v)$ is the minimum length of a stretch of value v in every row.
- $ULS(v)$ is the maximum length of a stretch of value v in every row.

Necessary Conditions

$$\forall k \in [0, K-1] : \#_k^v \geq \sum_{j=\max(0,k-LLS(v)+1)}^{k} ls_j^+(v) \tag{20}$$

$$\forall k \in [0, K-1] : \#_k^v \geq \sum_{j=k}^{\min(K-1,k+LLS(v)-1)} ls_j^-(v) \tag{21}$$

The intuition behind (20) resp. (21) is that the stretches starting resp. ending at the considered columns j must overlap column k.

$$\begin{aligned}&\forall k \in [0, K-1-ULS(v)] :\\ &ls_k^+(v) + \sum_{j=LLS(v)}^{ULS(v)} \#_{k+j}^v - (ULS(v) - LLS(v) + 1) \cdot R \leq 0\end{aligned} \tag{22}$$

$$\begin{aligned}&\forall k \in [ULS(v), K-1] :\\ &ls_k^-(v) + \sum_{j=LLS(v)}^{ULS(v)} \#_{k-j}^v - (ULS(v) - LLS(v) + 1) \cdot R \leq 0\end{aligned} \tag{23}$$

The intuition behind (22) is as follows. Consider a stretch beginning at column k. Then there must be an element distinct from v in column $j \in [k + LLS(v), k + ULS(v)]$ of the same row. So at least one of the terms in the summation of (22) will get a zero contribution from the given row. The reasoning in (23) is similar but considers stretches ending at column k.

2.4 Extracting Occurrence, Word, and Stretch Constraints from an Automaton, or How to Annotate an Automaton with String Properties

Toward automatically inferring the constant bounds $LW_r(w)$, $LWP_r(w)$, $LWS_r(w)$, $LS_r(w)$, etc, of the previous sub-sections, we now describe how a given automaton

Table 1. For each annotation in the first column, the second column gives the number of new counters, the third column gives their initial values, and the fourth column shows the string property variable among the final counter values. In the first three rows, ℓ is the word length.

Annotation	Number of counters	Initial values	Final values
$wordocc(\hat{v}^+, n)$	ℓ	$[0, ..., 0]$	$[\_, ..., n]$
$wordprefix(\hat{v}^+, b)$	$\ell + 1$	$[1, 0, ..., 0]$	$[\_, ..., b]$
$wordsuffix(\hat{v}^+, b)$	ℓ	$[0, ..., 0]$	$[\_, ..., b]$
$stretchocc(\hat{v}, n)$	2	$[0, 0]$	$[n, \_]$
$stretchminlen(\hat{v}, n)$	3	$[+\infty, +\infty, 0]$	$[n, \_, \_]$
$stretchmaxlen(\hat{v}, n)$	2	$[0, 0]$	$[n, \_]$

$\mathcal{A}$ can be automatically annotated with counter variables constrained to reflect properties of the strings that the automaton recognises. This is especially useful if $\mathcal{A}$ is a product automaton for several constraints. For this purpose, we use the *automaton* constraint introduced in [2], which (unlike the *regular* constraint [12]) allows us to associate counters to a transition. Each string property requires (i) a counter variable whose final value reflects the value of that string property, (ii) possibly some auxiliary counter variables, (iii) initial values of the counter variables, and (iv) update formulae in the automaton transitions for the counter variables. We now give the details for some string properties.

In this context, n denotes an integer or decision variable, b denotes a 0/1 integer or decision variable, $\hat{v}$ denotes a set of letters, $\hat{v}^+$ denotes a nonempty sequence of letters in $\hat{v}$, and s_i denotes the i^{th} letter of word s. We describe the annotation for the following string properties for any given string:

- $wordocc(\hat{v}^+, n)$: Word $\hat{v}^+$ occurs n times.
- $wordprefix(\hat{v}^+, b)$: $b = 1$ iff word $\hat{v}^+$ is a prefix of the string.
- $wordsuffix(\hat{v}^+, b)$: $b = 1$ iff word $\hat{v}^+$ is a suffix of the string.
- $stretchocc(\hat{v}, n)$: Stretches of letters in set $\hat{v}$ occur n times.
- $stretchminlen(\hat{v}, n)$: If letters in set $\hat{v}$ occur, then n is the length of the shortest such stretch, otherwise $n = +\infty$.
- $stretchmaxlen(\hat{v}, n)$: If letters in set $\hat{v}$ occur, then n is the length of the longest such stretch, otherwise $n = 0$.

For a given annotation, Table 1 shows which counters it introduces, as well as their initial and final values, while Table 2 shows the formulae for counter updates to be used in the transitions. Figure 1 shows an automaton annotated for $stretchocc(\{0\}, n)$.

An automaton can be annotated with multiple string properties—annotations do not interfere with one another—and can be simplified in order to remove multiple occurrences of identical counters that come from different string properties.

It is worth noting that propagation is possible from the decision variables to the counter variables, and vice-versa.

Table 2. Given an annotation and a transition of the automaton reading letter u, the table gives the counter update formulae to be used in this transition. For each annotation in the first column, the second column shows the counter names, and the third column shows the update formulae. The fourth column shows the condition under which each formula is used. In the first three multirows, ℓ is the word length.

Annotation	Counter values	New counter values	Condition
$wordocc(\hat{v}^+, n)$	$[c_1, ..., c_\ell]$	$[1, ...]$	$u \in \hat{v}_1^+$
		$[..., c_{i-1}, ...]$	$1 < i < \ell \wedge u \in \hat{v}_i^+$
		$[..., c_\ell + c_{\ell-1}]$	$u \in \hat{v}_\ell^+$
		$[..., 0, ...]$	$0 < i < \ell \wedge u \notin \hat{v}_i^+$
		$[..., c_\ell]$	$u \notin \hat{v}_\ell^+$
$wordprefix(\hat{v}^+, b)$	$[c_0, c_1, ..., c_\ell]$	$[0, ..., c_{i-1}, ...]$	$0 < i < \ell \wedge u \in \hat{v}_i^+$
		$[0, ..., \max(c_\ell, c_{\ell-1})]$	$u \in \hat{v}_\ell^+$
		$[0, ..., 0, ...]$	$0 < i < \ell \wedge u \notin \hat{v}_i^+$
		$[0, ..., c_\ell]$	$u \notin \hat{v}_\ell^+$
$wordsuffix(\hat{v}^+, b)$	$[c_1, ..., c_\ell]$	$[1, ...]$	$u \in \hat{v}_1^+$
		$[..., c_{i-1}, ...]$	$1 < i < \ell \wedge u \in \hat{v}_i^+$
		$[..., c_{\ell-1}]$	$u \in \hat{v}_\ell^+$
		$[..., 0, ...]$	$0 < i < \ell \wedge u \notin \hat{v}_i^+$
		$[..., c_\ell]$	$u \notin \hat{v}_\ell^+$
$stretchocc(\hat{v}, n)$	$[c, d]$	$[c - d + 1, 1]$	$u \in \hat{v}$
		$[c, 0]$	$u \notin \hat{v}$
$stretchminlen(\hat{v}, n)$	$[c, d, e]$	$[\min(d, e+1), d, e+1]$	$u \in \hat{v}$
		$[c, c, 0]$	$u \notin \hat{v}$
$stretchmaxlen(\hat{v}, n)$	$[c, d]$	$[\max(c, d+1), d+1]$	$u \in \hat{v}$
		$[c, 0]$	$u \notin \hat{v}$

2.5 Heuristics for Selecting Relevant String Properties for an Automaton

In our experiments (see Section 4), we chose to look for the following string properties:

- For each letter, lower and upper bounds on the number of its occurrences.
- For each letter, lower and upper bounds on the number or length of its stretches.
- Each word of length at most 3 that cannot occur at all.
- Each word of length at most 3 that cannot occur as a prefix or suffix.

These properties are derived, one at a time, as follows. We annotate the automaton as described in the previous section by the candidate string property. Then we compute by labelling the feasible values of the counter variable reflecting the given property, giving up if the computation does not finish within 5 CPU seconds. Among the collected word, prefix, suffix, and stretch properties, some properties are subsumed by others and are thus filtered away. Other properties could certainly have been derived, e.g., not only forbidden words, but also bounds on the number of occurrences of words. Our choice was based on (a) which properties we are able to derive necessary conditions for, and (b) empirical observations of what actually pays off in our benchmarks.

3 The Cardinality Automaton of an Automaton

The previous section introduced different complementary ways of generating necessary conditions (expressed in terms of arithmetic constraints) from a given automaton for the row constraints of the matrix $\mathcal{M}$ when its columns are subject to *gcc* constraints. This section presents an orthogonal systematic approach, again based on double counting, that can handle a larger class of column constraints completely mechanically.

Consider an $R \times K$ matrix $\mathcal{M}$, where on each row we have the same constraint, represented by an automaton $\mathcal{A}$ of p states $s_0, \ldots, s_{p-1}$, and on each column we have a *gcc* or linear (in)equality constraint where all the coefficients are the same. We will first construct an automaton that simulates the parallel running of the R copies of $\mathcal{A}$ and consumes entire columns of $\mathcal{M}$. Since this new automaton has p^R states, we then abstract it by just *counting* the automata that are in each state of $\mathcal{A}$. As even this abstracted automaton has a size exponential in p, we then use a linear-size encoding with linear constraints that allows us to consider also the column constraints on $\mathcal{M}$.

3.1 Necessary Row Constraints

The vector automaton $\overline{\mathcal{A}_R}$ consumes vectors of size R. Its states are sequences of R states of $\mathcal{A}$, where entry ℓ is the state of the automaton of row ℓ. There is a transition from state $\langle s_{i_0}, \ldots, s_{i_{R-1}} \rangle$ to state $\langle s_{j_0}, \ldots, s_{j_{R-1}} \rangle$ if and only if for each ℓ there is a transition in $\mathcal{A}$ from s_{i_ℓ} to s_{j_ℓ}. A state $\langle s_{i_0}, \ldots, s_{i_{R-1}} \rangle$ is initial (resp. final) if each of the s_{i_ℓ} is the initial (resp. a final) state of $\mathcal{A}$.

The cardinality (vector) automaton $\#\left(\overline{\mathcal{A}_R}\right)$ is an abstraction of the vector automaton $\overline{\mathcal{A}_R}$ that also consumes vectors of size R. Its states are sequences of p numbers, whose sum is R, where entry i is the number of automata $\mathcal{A}$ in state s_i. There is a transition from state $\langle c_{i_0}, \ldots, c_{i_{p-1}} \rangle$ to state $\langle c_{j_0}, \ldots, c_{j_{p-1}} \rangle$ if and only if there exists a multiset of R transitions in $\mathcal{A}$ such that for each ℓ there are c_{i_ℓ} of these R transitions going out from s_ℓ, and for each m there are c_{j_m} of these R transitions arriving into s_m. A state $\langle c_{i_0}, \ldots, c_{i_{p-1}} \rangle$ is initial (resp. final) if $c_{i_\ell} = 0$ whenever s_ℓ is not the initial (resp. a final) state of $\mathcal{A}$.

The number of states of $\#\left(\overline{\mathcal{A}_R}\right)$ is the number of ordered partitions of p, and thus exponential in p. However, it is possible to have a compact encoding via constraints. Toward this, we use $K+1$ sequences of p decision variables S_i^k in the domain $\{0, 1, \ldots, R\}$ to encode the states of an arbitrary path of length K (the number of columns) in $\#\left(\overline{\mathcal{A}_R}\right)$. For $k \in \{1, \ldots, K\}$, the sequence $\langle S_0^k, S_1^k, \ldots, S_{p-1}^k \rangle$ has as possible values the states of $\#\left(\overline{\mathcal{A}_R}\right)$ after it has consumed column $k-1$; the sequence $\langle S_0^0, S_1^0, \ldots, S_{p-1}^0 \rangle$ is fixed to $\langle R, 0, \ldots, 0 \rangle$ when, without loss of generality, s_0 is the initial state of $\mathcal{A}$. We get the following constraints:

$$\forall k \in \{0, \ldots, K\} : S_0^k + S_1^k + \cdots + S_{p-1}^k = R \tag{24}$$

$$\forall i \in \{0, \ldots, p-1\} : S_i^K = 0 \leftarrow s_i \text{ is not a final state of } \mathcal{A} \tag{25}$$

Assume that $\mathcal{A}$ has a set $\mathcal{T} = \{(a_0, \ell_0, b_0), (a_1, \ell_1, b_1), \ldots, (a_{q-1}, \ell_{q-1}, b_{q-1})\}$ of q transitions, where transition (a_i, ℓ_i, b_i) goes from state $a_i \in \{s_0, s_1, \ldots, s_{p-1}\}$ to state

$b_i \in \{s_0, s_1, \ldots, s_{p-1}\}$ upon reading letter $\ell_i \in \{0, 1, \ldots, V-1\}$. We use K sequences of q decision variables T_i^k in the domain $\{0, 1, \ldots, R\}$ to encode the transitions of an arbitrary path of length K in $\#\left(\overline{\mathcal{A}_R}\right)$. For $k \in \{0, \ldots, K-1\}$, the sequence $\langle T^k_{(a_0,\ell_0,b_0)}, T^k_{(a_1,\ell_1,b_1)}, \ldots, T^k_{(a_{q-1},\ell_{q-1},b_{q-1})}\rangle$ gives the numbers of automata $\mathcal{A}$ with transition $(a_0, \ell_0, b_0), (a_1, \ell_1, b_1), \ldots, (a_{q-1}, \ell_{q-1}, b_{q-1})$ upon reading the character of their row in column k. We get the following constraint for column k:

$$T^k_{(a_0,\ell_0,b_0)} + T^k_{(a_1,\ell_1,b_1)} + \cdots + T^k_{(a_{q-1},\ell_{q-1},b_{q-1})} = R \tag{26}$$

Consider two state encodings $\langle S_0^k, S_1^k, \ldots, S_{p-1}^k\rangle$ and $\langle S_0^{k+1}, S_1^{k+1}, \ldots, S_{p-1}^{k+1}\rangle$, and consider the transition encoding $\langle T^k_{(a_0,\ell_0,b_0)}, T^k_{(a_1,\ell_1,b_1)}, \ldots, T^k_{(a_{q-1},\ell_{q-1},b_{q-1})}\rangle$ between these two state encodings (with $0 \leq k < K$). To encode paths of length K in $\#\left(\overline{\mathcal{A}_R}\right)$, we introduce the following constraints. First, we constrain the number of automata $\mathcal{A}$ at any state s_j before reading column k to equal the number of firing transitions going out from s_j when reading column k:

$$\forall j \in \{0, \ldots, p-1\} : S_j^k = \sum_{(a_i,\ell_i,b_i)\in\mathcal{T}\ :\ a_i = s_j} T^k_{(a_i,\ell_i,b_i)} \tag{27}$$

Second, we constrain the number of automata $\mathcal{A}$ at state s_j after reading column k to equal the number of firing transitions coming into s_j when reading column k:

$$\forall j \in \{0, \ldots, p-1\} : S_j^{k+1} = \sum_{(a_i,\ell_i,b_i)\in\mathcal{T}\ :\ b_i = s_j} T^k_{(a_i,\ell_i,b_i)} \tag{28}$$

A reformulation with linear constraints when $R = 1$ and there are *no* column constraints is described in [6].

3.2 Necessary Column Constraints and Channelling Constraints

The necessary constraints above on the state and transition variables only handle the row constraints, but they can also be used to handle column constraints of the considered kinds. These necessary constraints can thus be seen as a communication channel for enhancing the propagation between row and column constraints.

If column k has a *gcc*, then we constrain the number of occurrences of value v in column k to equal the number of transitions on v when reading column k:

$$\forall v \in \{0, \ldots, V-1\} : \#_k^v = \sum_{(a_i,\ell_i,b_i)\in\mathcal{T}\ :\ \ell_i = v} T^k_{(a_i,\ell_i,b_i)} \tag{29}$$

If column k constrains the sum of the column, then we constrain that sum to equal the value-weighted number of transitions on v when reading column k:

$$\sum_{r=0}^{R-1} \mathcal{M}[r,k] = \sum_{v=0}^{V-1} v \cdot \left(\sum_{(a_i,\ell_i,b_i)\in\mathcal{T}\ :\ \ell_i = v} T^k_{(a_i,\ell_i,b_i)} \right) \tag{30}$$

Furthermore, for more propagation, we can link the variables S_i^k back to the state variables [2] of the R automata $\mathcal{A}$. For this purpose, let the variables $Q_i^0, Q_i^1, \ldots, Q_i^K$ (with $0 \leq i < R$) denote the $K+1$ states visited by automaton $\mathcal{A}$ on row i of length K. We get the following *gcc* necessary constraints:

$$\forall k \in \{0,\ldots,K\} : gcc(\langle Q_0^k, Q_1^k, \ldots, Q_{R-1}^k\rangle, \langle 0 : S_0^k, 1 : S_1^k, \ldots, p{-}1 : S_{p-1}^k\rangle) \quad (31)$$

Example 2. In the context of an $R = 4$ by $K = 6$ matrix with a *global_contiguity* constraint on each row and a *gcc* constraint on each column, we illustrate the set of linear constraints associated with column k (where $0 \leq k < 6$) of the matrix. An automaton $\mathcal{A}$ associated with the *global_contiguity* constraint was described by Figure 1 of Example 1. It has $p = 3$ states s_0, s_1, s_2 and $q = 5$ transitions $(s_0, 0, s_0)$, $(s_0, 1, s_1)$, $(s_1, 1, s_1)$, $(s_1, 0, s_2)$, $(s_2, 0, s_2)$ labelled by values 0 and 1. The encoding has $p \cdot (K+1) = 21$ variables S_i^k such that $S_0^k + S_1^k + S_2^k = 4$ for every k. Since s_0 is the initial state of $\mathcal{A}$, we require that $S_0^0 = 4$ since $S_1^0 = 0 = S_2^0$. Since $\mathcal{A}$ only has final states, no S_j^K is constrained to be zero. The encoding also has $q \cdot K = 30$ variables T_i^k such that $T_{(s_0,0,s_0)}^k + T_{(s_0,1,s_1)}^k + T_{(s_1,1,s_1)}^k + T_{(s_1,0,s_2)}^k + T_{(s_2,0,s_2)}^k = 4$ for every k. The following three sets of linear necessary constraints link the variables above for every k:

$$\begin{array}{ll}
S_0^k = T_{(s_0,0,s_0)}^k + T_{(s_0,1,s_1)}^k & (\textit{transitions that exit state } s_0)\\
S_1^k = T_{(s_1,1,s_1)}^k + T_{(s_1,0,s_2)}^k & (\textit{transitions that exit state } s_1)\\
S_2^k = T_{(s_2,0,s_2)}^k & (\textit{transitions that exit state } s_2)\\
\\
S_0^{k+1} = T_{(s_0,0,s_0)}^k & (\textit{transitions that enter state } s_0)\\
S_1^{k+1} = T_{(s_0,1,s_1)}^k + T_{(s_1,1,s_1)}^k & (\textit{transitions that enter state } s_1)\\
S_2^{k+1} = T_{(s_1,0,s_2)}^k + T_{(s_2,0,s_2)}^k & (\textit{transitions that enter state } s_2)\\
\\
\#_k^0 = T_{(s_0,0,s_0)}^k + T_{(s_1,0,s_2)}^k + T_{(s_2,0,s_2)}^k & (\textit{transitions labelled by value } 0)\\
\#_k^1 = T_{(s_0,1,s_1)}^k + T_{(s_1,1,s_1)}^k & (\textit{transitions labelled by value } 1)
\end{array}$$

4 Evaluation and Conclusion

NSPLib [14] is a very large repository of (artificially generated) instances of the *nurse scheduling problem* (NSP), which is about constructing a duty roster for nursing staff. Let N be the number of nurses, D the number of days of the scheduling horizon, and S the number of shifts. The objective is to construct an $N \times D$ matrix of values in the integer interval $[1, S]$, with value S representing the off-duty "shift".

In *instance files*, there are hard *coverage constraints* and soft preference constraints; we only use the former here: they give for each day d and shift s the lower bound on the number of nurses that must be assigned to shift s on day d, and can be modelled by a global cardinality constraint (*gcc*) on the columns. We stress that the *gcc* constraints on any two columns are in general *not* the same. There are instance files for $N \times 7$ rosters with $N \in \{25, 50, 75, 100\}$, and for $N \times 28$ rosters with $N \in \{30, 60\}$.

In *case files*, there are hard constraints on the rows. For each shift s, there are lower and upper bounds on the number of occurrences of s in any row (the daily assignment of some nurse): this can be modelled by gcc constraints on the rows. There are even lower and upper bounds on the cumulative number of occurrences of the working shifts $1, \dots, S-1$ in any row: this can be modelled by gcc constraints on the off-duty value S and always gives tighter occurrence bounds on S than in the previous gcc constraints. For each shift s, there are also lower and upper bounds on the length of any stretch of value s in any row: this can be modelled by $stretch\_path$ constraints on the rows. Finally, there are lower and upper bounds on the length of any stretch of the working shifts $1, \dots, S-1$ in any row: this can be modelled by generalised $stretch\_path\_partition$ constraints [3] on the rows. We stress that the constraints on any two rows are the *same*. There are 8 case files for the $N \times 7$ rosters, and another 8 case files for the $N \times 28$ rosters. We automatically generated (see [3] for details) deterministic finite automata (DFA) for all the row constraints of each case, but used their minimised product DFA instead (obtained through standard DFA algorithms), thereby getting domain consistency on the conjunction of all row constraints [2]. For each case, string properties were automatically selected off-line as described in Section 2.5, and cardinality automata were automatically constructed off-line as described in Section 3.

Under these choices, the NSPLib benchmark corresponds to the pattern studied in this paper. To reduce the risk of reporting improvements where another search procedure can achieve much of the same impact, we use a two-phase search that exploits the fact that there is a single domain-consistent constraint on each row and column:

- Phase 1 addresses the column (coverage) constraints only: it seeks to assign enough nurses to given shifts on given days to satisfy *all but one* coverage constraint. To break row symmetries, an equivalence relation is maintained: two rows (nurses) are in the same equivalence class while they are assigned to the same shifts and days.
- In Phase 2, one column constraint and all row constraints remain to be satisfied. But these constraints form a Berge-acyclic CSP [1], and so the remaining decision variables can be trivially labelled without search.

This search procedure is much more efficient than row-wise labelling under decreasing value ordering (value S always has the highest average number of occurrences per row) in the presence of a decreasing lexicographic ordering constraint on the rows.

The objective of our experiments is to measure the impact in runtime and backtracks when using either or both of our methods. The experiments were run under SICStus Prolog 4.1.1 and Mac OS X 10.6.2 on a 2.8 GHz Intel Core 2 Duo with a 4GB RAM. All runs were allocated 1 CPU minute. For each case and nurse count N, we used the *first* 10 instances for each configuration of the NSPLib coverage complexity indicators, that is instances 1–270 for the $N \times 7$ rosters and 1–120 for the $N \times 28$ rosters.

Table 3 summarises the running of these 3120 instances using neither, either, and both of our methods. Each row first indicates the number of known instances of some satisfiability status ('sat' for satisfiable, and 'unsat' for unsatisfiable) for a given case and nurse count N, and then the performance of each method to the first solution, namely the number of instances decided to be of that status without timing out, as well as the total runtime (in seconds) and the total number of backtracks on all instances where *none* of the four methods timed out (it is very important to note that this means that these totals are *comparable*, but also that they do not reveal any performance gains on instances where *at least one* of the methods timed out). Numbers in boldface indicate

Table 3. NSPlib benchmark results

				Neither			String Properties			Cardinality DFA			Both		
Case	N	Status	Known	#Inst	Time	#Bktk	#Inst	Time	#Bktk	#Inst	Time	#Bktk	#Inst	Time	#Bktk
7	25	sat	230	**230**	**16.7**	32099	**230**	42.6	13909	**230**	39.8	13813	**230**	74.8	**13781**
		unsat	38	37	51.9	113413	**38**	57.1	19491	**38**	**37.2**	21133	**38**	57.9	**12877**
7	50	sat	216	213	**9.5**	12165	**216**	24.0	**11055**	214	32.4	11077	**216**	49.8	11057
		unsat	43	40	**55.0**	79629	42	87.5	22082	**43**	107.5	61092	**43**	55.0	**10863**
7	75	sat	210	208	**13.0**	12709	209	22.1	628	**210**	48.8	12421	**210**	49.1	**340**
		unsat	48	**48**	78.5	155490	**48**	**36.3**	8860	**48**	45.3	12455	47	42.0	**8267**
7	100	sat	220	217	**9.0**	361	**219**	30.7	361	217	52.2	**355**	**219**	74.1	**355**
		unsat	26	22	26.3	8909	24	4.9	**452**	23	4.9	993	**25**	**2.8**	**452**
8	25	sat	263	**263**	**2.2**	282	**263**	10.3	282	**263**	14.4	**76**	**263**	22.6	**76**
		unsat	7	**7**	36.2	121367	**7**	**0.0**	**19**	**7**	0.2	**19**	**7**	0.2	**19**
8	50	sat	259	**259**	**4.5**	**136**	**259**	17.3	**136**	**259**	27.8	**136**	**259**	40.8	**136**
		unsat	11	10	28.0	49358	**11**	**3.2**	715	10	58.8	29784	**11**	4.0	**592**
8	75	sat	246	245	**7.2**	449	245	23.4	**230**	**246**	46.2	449	**246**	61.4	**230**
		unsat	22	21	54.4	112880	**22**	**0.1**	**21**	**22**	0.4	53	**22**	0.4	**21**
8	100	sat	262	261	**10.7**	**239**	**262**	32.5	**239**	261	65.5	**239**	**262**	87.9	**239**
		unsat	6	4	0.2	73	**6**	**0.0**	**4**	4	0.4	73	**6**	0.1	**4**
15	30	sat	87	84	**245.3**	**37**	86	257.3	**37**	86	1205.6	**37**	**87**	1219.5	**37**
		unsat	23	9	26.8	2513	**23**	**1.9**	**9**	18	17.9	83	**23**	6.0	**9**
15	60	sat	87	**87**	**361.8**	**131**	**87**	380.4	**131**	**87**	2108.2	**131**	**87**	2137.1	**131**
		unsat	13	8	32.8	1001	**13**	**2.9**	**8**	11	40.9	390	**13**	6.3	**8**
16	30	sat	100	**100**	**567.5**	**153**	**100**	578.6	**153**	**100**	2541.0	**153**	**100**	2557.8	**153**
		unsat	10	4	11.0	172	**10**	**1.4**	**4**	6	68.5	165	**10**	4.9	**4**
16	60	sat	105	**105**	**706.9**	**142**	**105**	722.0	**142**	88	3329.9	**142**	88	3350.2	**142**
		unsat	3	1	25.7	579	**3**	**0.0**	**1**	2	0.8	**1**	**3**	0.8	**1**

best performance in a row. It turned out that Cases 1–6, 9–10, 12–14 are very simple (in the absence of preference constraints), so that our methods only decrease backtracks on one of those 2220 instances, but increase runtime. It also turned out that Case 11 is very difficult (even in the absence of preference constraints), so that even our methods systematically time out, because the product automaton of all row constraints is very big; we could have overcome this obstacle by using the built-in *gcc* constraint and the product automaton of the remaining row constraints, but we wanted to compare all the cases under the same scenario. Hence we do not report any results on Cases 1–6, 9–14.

An analysis of Table 3 reveals that our methods decide more instances without timing out, and that they often drastically reduce the runtime and number of backtracks (by up to four orders of magnitude), especially on the shared unsatisfiable instances. However, runtimes are often increased (by up to one order of magnitude) on the shared satisfiable instances. String properties are only rarely defeated by the cardinality DFA on any of the three performance measures, but their combination is often the overall winner, though rarely by a large margin. A more fine-grained evaluation is necessary to understand when to use which string properties without increasing runtime on the satisfiable instances. The good performance of our methods on unsatisfiable instances is indicative of gains when exploring the whole search space, such as when solving an optimisation problem or using soft (preference) constraints.

With constraint programming, NSPLib instances (without the soft preference constraints) were also used in [4,5], but under row constraints that are different from those of the NSPLib case files that we used. NSP instances from a different repository were used in [11], though with soft global constraints: one of the insights reported there was the need for more interaction between the global constraints, and our paper shows steps that can be taken in that direction.

Since both our methods essentially generate linear constraints, they may also be relevant in the context of linear programming. Future work may also consider the integration of our techniques with the *multicost-regular* constraint [10], which allows the direct handling of a *gcc* constraint in the presence of automaton constraints (as on the rows of NSPLib instances) without explicitly computing the product automaton, which can be very big.

References

1. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. Journal of the ACM 30, 479–513 (1983)
2. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer, Heidelberg (2004)
3. Beldiceanu, N., Carlsson, M., Rampon, J.-X.: Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science (2005), The current working version is at: `www.emn.fr/x-info/sdemasse/gccat/doc/catalog.pdf`
4. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: SLIDE: A useful special case of the CARDPATH constraint. In: ECAI 2008, pp. 475–479. IOS Press, Amsterdam (2008)
5. Brand, S., Narodytska, N., Quimper, C.-G., Stuckey, P.J., Walsh, T.: Encodings of the *sequence* constraint. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 210–224. Springer, Heidelberg (2007)
6. Côté, M.-C., Gendron, B., Rousseau, L.-M.: Modeling the *regular* constraint with integer programming. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 29–43. Springer, Heidelberg (2007)
7. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–476. Springer, Heidelberg (2002)
8. Frisch, A.M., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 318–332. Springer, Heidelberg (2003)
9. Jukna, S.: Extremal Combinatorics. Springer, Heidelberg (2001)
10. Menana, J., Demassey, S.: Sequencing and counting with the *multicost-regular* constraint. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 178–192. Springer, Heidelberg (2009)
11. Métivier, J.-P., Boizumault, P., Loudni, S.: Solving nurse rostering problems using soft global constraints. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 73–87. Springer, Heidelberg (2009)
12. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
13. Régin, J.-C., Gomes, C.: The *cardinality matrix* constraint. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 572–587. Springer, Heidelberg (2004)
14. Vanhoucke, M., Maenhout, B.: On the characterization and generation of nurse scheduling problem instances. European Journal of Operational Research 196(2), 457–467 (2009); NSPLib is at: `www.projectmanagement.ugent.be/nsp.php`