Regular Language Membership Constraint

Niko Paltzer

Universität des Saarlandes, Deutschland,
nikopp@ps.uni-sb.de,
Advisor: Lutz Straßburger

Abstract. This paper mainly deals with the work of Gilles Pesant on the stretch constraint and its reformulation as a regular language membership constraint. Some definitions and examples should introduce and explain the notion of a stretch, the stretch constraint and the regular constraint. The consistency algorithm for the regular constraint is explained and illustrated with an additional example. Some comparative remarks on the stretch constraint and the regular constraint are included as well.

1 Introduction

The first question I asked myself when I went in for the regular language membership constraint was the question for the application area. It seemed to be something artificial that does not really matter in usual constraint problems as for example scheduling or logical problems. Therefore it was necessary to go a little step back in history to discover how Gilles Pesant hit on the introduction of the regular constraint in [1].

It turns out that the regular constraint is just a generalization of the stretch constraint that Pesant introduced in [3]. And the main application field of this stretch constraint are in fact scheduling especially rostering problems.

Therefore I first want to give an example for such a rostering problem namely the construction of a rotating schedule. Then this problem ist formulated with the stretch constraint. Finally, from the resulting example, a deterministic finite automaton (DFA) is deduced.

Then we consider the formal definition of the regular constraint. Afterwards, I shortly recall the definition of hyper arc consistency since that is what the subsequent described consistency algorithm for the regular constraint achieves. A small example should make clear, how this algorithm works. Some specialized benchmarks concerning the application to stretch instances are given.

Finally, another example shows a different possible (but unfortunately inefficient) application of the regular constraint, namely the alldifferent constraint.

2 Rotating Schedules

In todays business, it is normal for a lot of companies to be available 24 hours around the clock. Therefore it is necessary to have a time table for the workers that on the one hand ensures that a worker (a team of workers) is present the whole day and on the other hand guarantees that each worker has enough days off per week to regenerate.

For this problem, so called rotating schedules were introduced. These schedules are organised as follows:

- a shift table for a single worker or a team of workers for several weeks is created
- the different workers (resp. teams) have all the same shift table but each starts with a different offset of days

Therefore the shift table is done *in parallel*. While the first worker (or team) starts for example with the shifts of the first week in the table, the second one performs the shifts of the second week and so on.

Table 1 shows an example where we have three diffent shift types, namely *day shifts* (D), *night shifts* (N) and *days off* (-).

Table 1. a sample rotating schedule featuring day shifts, night shifts and days off

mo	tu	we	th	fr	sa	su
D	D	D	-	-	N	N
N	-	-	D	D	D	D
-	N	N	N	N	-	-

week 1	week 2	week 3
team 1	team 3	team 2
team 2	team 1	team 3
team 3	team 2	team 1

3 Stretch Constraint

Given a sequence of values, a *stretch* is a consecutive subsequence of identical values a. Additionally, a stretch is always as long as possible i.e. the preceding and succeeding values are different from a. A stretch S consisting of n values a is called an a-stretch of length n ($length(S) = n$).

The types of two consecutive stretches (considered as an ordered pair) are called a *pattern*. E.g. the sequence *aaacc* consists of an a-stretch of length 3 and a c-stretch of length 2 and these two stretches form the pattern (a, c).

The formal definition of the stretch constraint requires

- a set of shift types $T = \{t_1, \ \dots \ , t_m\}$,
- two vectors min and max of length m,
- a sequence of FD variables $s = \ < s_0, \ \dots \ , s_{n-1} >$,
- finite domains $D_{s_i} \subset T$,
- a set of patterns $\Pi \subset T \times T$,
- a boolean value *cyclic*.

The vectors min and max restrict the length of the occuring stretches. For each shift type $t \in T$, they contain an integer value. In oder to state a consistent constraint, $min_t \leq max_t$ must hold for all t.

The set Π contains all valid patterns i.e. only the patterns in Π are allowed to occur in a sequence of values assigned to s.

The boolean flag *cyclic* denotes if the sequence s should be considered as a cycle. For example if *cyclic* = `true` then the sequence *aaacc* also forms the pattern (c, a). And the sequence *caaac* only consists of two stretches (one c-stretch of length 2 and one a-stretch of length 3) instead of three stretches in the non-cyclic case (two c-stretches of length 1 and one a-stretch of length 3).

The stretch constraint is then stated with

$$\texttt{stretch}\ (s, min, max, \Pi, cyclic)$$

and it ensures that

- $\forall$ stretches S of type $t : min_t \leq length(S) \leq max_t$,
- $\forall$ consecutive stretches S and S' of type t and $t' : (t, t') \in \Pi$.

3.1 Example: Rotating Schedule

Suppose now that we want to model a stretch constraint to create a rotating schedule with the follwing properties:

- day shifts, night shifts and days off
- alternating between day and night work
- work stretches of length 3 or 4
- after (night) work, (1+) 1 or 2 days off
- exactly one day shift and one night shift per day

The last requirement can not be formulated within the stretch constraint, therefore we assume that it is ensured by some other global constraints.

Since we have two shifts a day (day shift and night shift), our rotating schedule has to have at least 14 days. But we want our workers to have some time for regeneration, we choose a length 21 days to guarantee enough space for days off. Therfore, we have 21 variables $s_0, \ \dots \ , s_{20}$.

Because we want the workers to alternate between day and night work, we have to introduce two different shift types for days off, namely O_D for days off after day work and O_N after night work. The respective patterns are (D, O_D) and (N, O_N). Now we only have to introduce two more patterns (O_D, N) and (O_N, D) that guarantee a night shift after day work and vice versa.

The full model of the stretch constraint looks as follows:

- $T = \{D, N, O_D, O_N\}$
- $s =< s_0, \ \ldots \ , s_{20} >$
- $D_{s_i} = T$
- $\Pi = \{(D, O_D), (O_D, N), (N, O_N), (O_N, D)\}$
- $min_{D,N,O_D,O_N} = (3, 3, 1, 2)$
- $max_{D,N,O_D,O_N} = (4, 4, 2, 3)$
- $cyclic =$ `true`

$$\texttt{stretch}\ (s, min, max, \Pi, cyclic)$$

Table 2 shows an example shift table fulfilling this constraint. It is equivalent to the one shown in Table 1 except for the two different types (O_D and O_N) for days off.

Table 2. Sample shift table fulfilling the stretch constraint

D	D	D	O_D	O_D	N	N
N	O_N	O_N	D	D	D	D
O_D	N	N	N	N	O_N	O_N

4 Regular Constraint

To illustrate the transformation of the stretch constraint into a regular language membership constraint we draw the DFA in Fig. 1.

The DFA restricts the length of the stretches in the same way the stretch constraint does, since for example only the nodes where three or four work shifts have been completed are final states.

Only the acyclic case is modelled here since the cyclic one causes some overhead that will be discussed later.

Since the class of languages accepted by a DFA is in fact equivalent to the class of languages described by a regular expression, it is indeed a regular language membership problem.

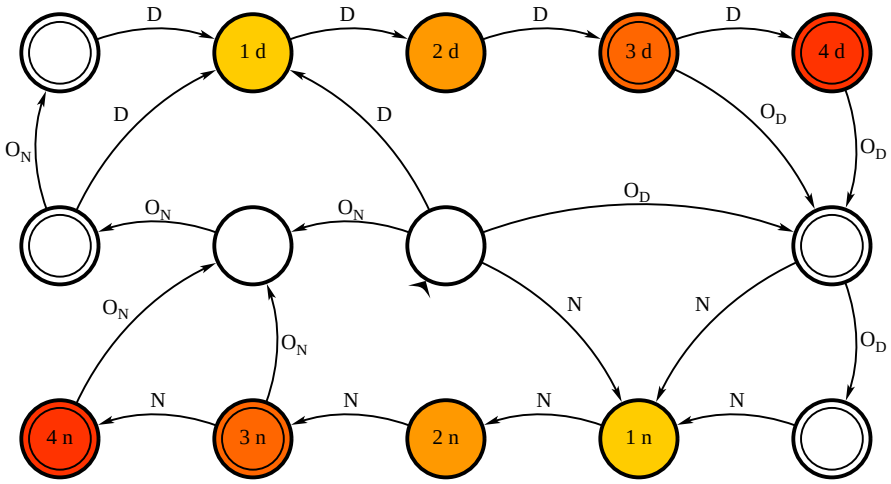

Fig. 1. the DFA "from stretch" of the example given in Section 3.1

The formal definition of the regular constraint is quite smaller than the one for the stretch constraint, since the information of min, max and Π is included in the DFA:

- sequence of FD variables $s = \ < s_1, s_2, \ \ldots \ , s_n >$
- DFA $M = (Q, \Sigma, \delta, q_0, F)$
- finite domains $D_{s_i} \subset \Sigma$

The DFA as usual consists of the finite set of states Q, the finite alphabet Σ, the transition function δ, the initial state q_0 and the set of final states F.

Stating the regular constraint

$$\texttt{regular}\ (s, M)$$

ensures that $value(s_1) \cdot value(s_2) \cdot \ \ldots \ \cdot value(s_n) \ \in \mathcal{L}(M)$ i.e. every sequence of values taken by the variables of s have to be a member of the regular language recognised by M.

5 Hyper Arc Consistency

The consistency algorithm for the regular constraint discussed in Section 6 achieves hyper arc consistency so let me shortly recall the definition from [2]. It is often also called generalized arc consistency or domain consistency. Remember that arc consistency itself is only defined for binary constraints i.e. constraints over exactly two variables. Since hyper arc consistency is the generalisation of arc consistency, it is defined for constraints over any (finite) number of variables.

The formal definition looks as follows:
A constraint $C \subset D_1 \times \ \ldots \ \times D_k$ is called hyper arc consistent iff
$\forall\ D_i\ \forall\ a \in D_i$:
$\exists (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k) \in D_1 \times \ldots \times D_{i-1} \times D_{i+1} \times \ldots \times D_k$:

$$(a_1, \ \ldots\ , a_{i-1}, a, a_{i+1}, \ \ldots\ , a_k) \in C$$

Intuitively expressed, a constraint C is hyper arc consistent iff a single arbitrary variable constrained by C can be determined to an arbitrary value from its domain and C is still feasible. So each value from any domain taken by the corresponding variable is part of at least one solution to C.

The advantage of a CSP, consisting of only one hyper arc consistent constraint is the fact that no backtracking is needed when distributing over the domains. Since the distribution methods only split domains with more than one variable, it follows from the definition of hyper arc consistency that such a CSP has at least two solutions. After the distribution step, each created subproblem has at least one of these solutions.

Apt defines in [2] hyper arc consistency even for CSPs: A CSP is hyper arc consistent iff all its constraints are hyper arc consistent.

In my opinion, this is a somehow missleading terminology because the above described feature is not guaranteed in such a CSP! In fact, a - per definition - hyper arc consistent CSP does not need to have a solution even if all domains are nonempty. Therefore I would prefer to call a CSP hyper arc consistent iff it consists of only one hyper arc consistent constraint.

6 Consistency Algorithm

Now I want to give an informal description of the consistency algorithm introduced by Gilles Pesant. The interested reader can find the formal definition and the pseudo code in [1].

The algorithm consists of the initial phase where the necessary data structures are initialized and a maintenance phase where these data structures are modified according to changes of the affected domains.

The main data structure is a directed acyclic graph (DAG). The nodes of the graph are grouped into levels. Each node (q, i) is labelled with the corresponding state q of M and the level i. Therefore any state of M occurs at most once on each level. Level 0 only contains the root of the DAG i.e. the initial state of M. Then level i corresponds to variable s_i. So the graph consists of $n + 1$ levels. The arcs in the graph are labelled with values. Arcs pointing to a node (q, i) can only be labelled with a value $v \in D_{s_i}$. Such an arc is called v-supporting arc. Vice versa, after the initial phase, a value v is removed from D_{s_i} if there is no v-supporting arc on level i. All nodes (q, n) on level n have to be final states of M i.e. $q \in F$.

The correlation between this graph and the constraint is the following: For each solution of the constraint there exists a path from the root node on level 0 to a node on level n in the graph.

6.1 Initial Phase

The initial phase again consists of three subphases, a forward phase, a backward phase and a clean-up phase. Each phase is performed with n steps.

forward phase At first, the initial node of M is inserted into level 0. Then we iterate from $i = 1$ to n.

In iteration i, for all nodes $k = (q, i-1)$ and all values $v \in D_{s_i}$, the node $k' = (q', i)$ with $q' = \delta(q, v)$ and the arc (k, k') are inserted into the DAG. [1]

backward phase This phase starts with removing all nodes (q, n) and their corresponding incoming arcs with $q \notin F$. Then we iterate from $i = n-1$ to 0.

In iteration i, we remove all nodes (q, i) and their corresponding incoming arcs that have no outgoing arcs.

clean-up phase Now we have the desired DAG, where each path from level 0 to level n corresponds to a solution to the constraint.

For $i = 1$ to n we remove all values v from D_{s_i} that have no supporting arc pointing to a node on level i.

After that, we have reached hyper arc consistency since all values that do not occur in at least one path (resp. one solution) have been removed.

6.2 Maintenance Phase

If other global constraints remove a value from a domain of the regular constraint, an update on the DAG needs to be performed.

For each value that is removed from D_{s_i}, the corresponding supporting arcs pointing to level i have to be removed from the DAG.

For each arc pointing to level i we remove, the following steps are executed recursively:

- remove unreachable nodes on level i (nodes that have no incoming arcs anymore) and their outgoing arcs pointing to level $i+1$ (recursion!)
- remove invalid nodes on level $i-1$ (nodes that have no outgoing arcs anymore) and their incoming arcs (recursion!)
- remove those values from $D_{s_{i-1}}$ and $D_{s_{i+1}}$ that have no supporting arcs anymore

[1] remember that δ is the transition function of M

6.3 Example: Regular Constraint

Now we want to consider a small example to visualise how the algorithm works. Therfore we introduce a sequence of four variables $s =< s_1, s_2, s_3, s_4 >$ with domains $D_{s_i} = \{a, b\}$. The DFA M is given by Fig. 2. The alphabet Σ only consists of the two letters a and b. The words of the language accepted by M are restricted to alternating a-stretches of length 1 and b-stretches of length 2 or 3.

Fig. 3 to Fig. 6 show the initialisation and the maintenance of the graph.

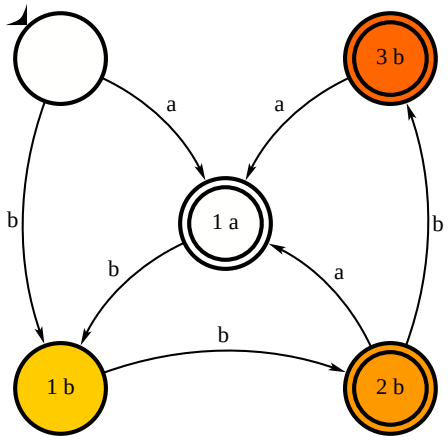

Fig. 2. sample DFA M that accepts alternating a-stretches of length 1 and b-stretches of length 2 or 3

6.4 Properties of the Algorithm

As already mentioned, this algorithm achieves hyper arc consistency because a value v only remains in a domain iff there exists a path in the graph that includes a v-supporting arc. And since a path in the graph is equivalent to a solution to the constraint, a value only remains in a domain iff it is part of at least one solution to the constraint.

The running time and space is in $\mathcal{O}(n \cdot |\Sigma| \cdot |Q|)$, details can be found in [1] as well as a short description of a slightly modified algorithm. This algorithm does not maintain the whole graph but only one v-supporting edge per value and level. Clearly, if one of these edges is removed, it has to be checked, if we can find another v-supporting edge that we omitted so far. But the given benchmarks do not show an significantly improved running time.

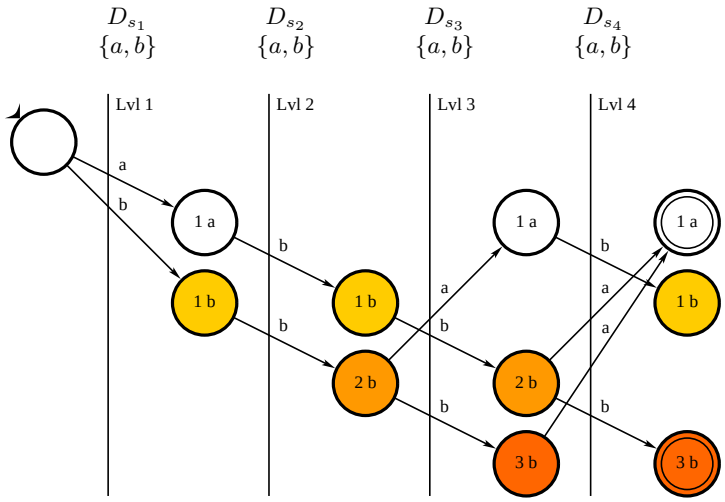

Fig. 3. sample DAG after the forward phase: final states of M are also marked as final nodes on level 4

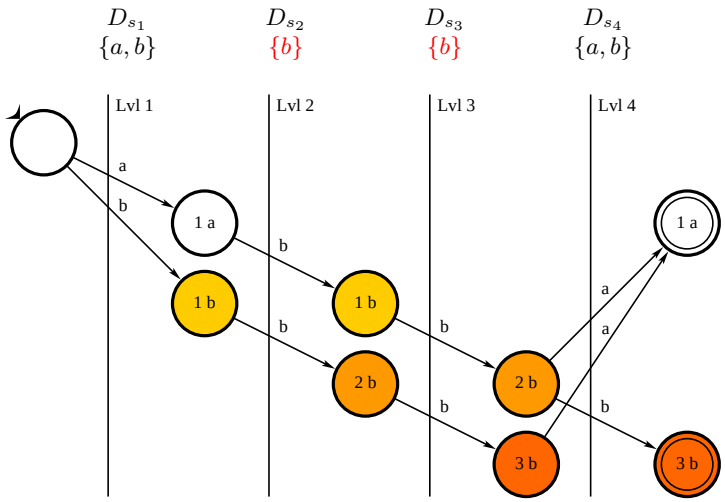

Fig. 4. sample DAG after the backward phase and the clean-up phase: the nonfinal node on level 4 and the invalid node on level 3 have been removed; value a has been removed from D_{s_2} and D_{s_3}

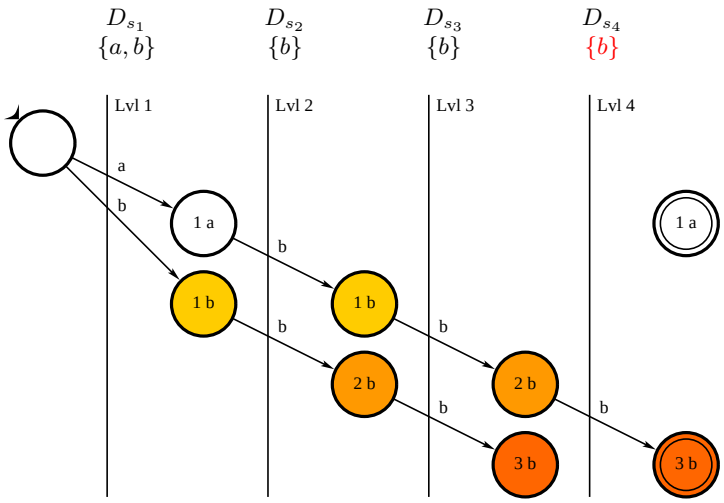

Fig. 5. sample DAG at the beginning of the maintenance phase: value a has been removed from D_{s_4} by another global constraint, therefore the corresponding supporting arcs have been removed

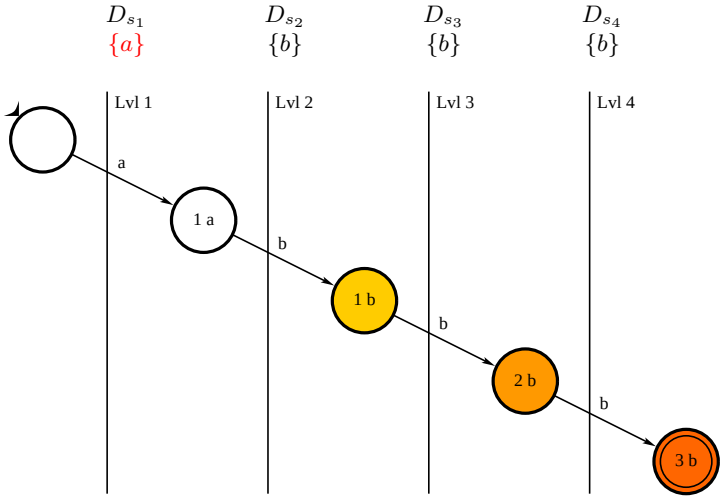

Fig. 6. sample DAG after the maintenance phase: all unreachable and invalid nodes and their corresponding arcs have been removed; value b has been removed from D_{s_1} since the only b-supporting arc has been removed

Pesant also gives the hint that, depending on the way the DFA is created, it might be preferable to construct a minimum state DFA in $\mathcal{O}(|Q| \cdot log|Q|)$ before passing it to the regular constraint.

6.5 Remarks on Stretch

In the benchmark from [1], the consistency algorithm for the regular constraint is faster than stretch filtering implementation from [3] for instances with $n \geq 50$ and $|\Sigma| \geq 5$. Since the filering algorithm does not achieve hyper arc consistency, it has to perform a lot of backtracking on harder instances.

Together with Lars Hellsten and Peter van Beek, Gilles Pesant presented in [4] a new algorithm for the stretch constraint that achieves hyper arc consistency and runs in $\mathcal{O}(n \cdot m^2)$. Unfortunately no benchmark is given that compares this new version with one for the regular constraint.

As already mentioned, it is not straight forward to model circular stretch problems as a regular constraint. For this purpose we have to duplicate l variables where l is the maximum of the allowed stretch-lengths. Furthermore we have to introduce another $2 \cdot l - 2$ dummy variables (for more details see [1]). Depending on the problem this can cause a significant amount of overhead.

6.6 Example: Alldifferent

In the end I want to give a last example not featuring stretches but the well known alldifferent constraint.

The regular constraint can be intuitively used to intuitively model this important global constraint. The states of the DFA simply represent the set of the already taken values. All states are marked final, since from the construction, no invalid allocation of the variables is permitted.

Unfortunately this is very inefficient, since the number of states is exponential in the number of letters in the alphabet ($|Q| = 2^{|\Sigma|}$).

If the number of variables n is strictly smaller than the number of different values, the size of the DFA can be reduced by removing all nodes that are more than n steps away from the initial node. The resulting DFA can again be minimized with the resp. algorithm.

Fig. 7 shows an example DFA for the alphabet $\Sigma = \{a, b, c\}$.

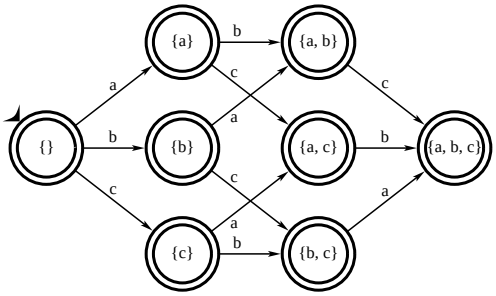

Fig. 7. sample DFA to formulate the alldifferent constraint unsing the regular constraint

References

1. Gilles Pesant: A Regular Language Membership Constraint for Finite Sequences of Variables. Springer-Verlag Berlin Heidelberg (2004)
2. Krzysztof R. Apt: Principles of Constraint Programming: Cambridge University Press (2003)
3. Gilles Pesant: A Filtering Algorithm for the Stretch Constraint. Springer-Verlag Berlin Heidelberg (2001)
4. Lars Hellsten, Gilles Pesant and Peter van Beek: A Domain Consistency Algorithm for the Stretch Constraint. Springer-Verlag Berlin Heidelberg (2004)