A Regular Language Membership Constraint for Finite Sequences of Variables

Gilles Pesant[1,2]

[1] École Polytechnique de Montréal, Montreal, Canada
[2] Centre for Research on Transportation (CRT),
Université de Montréal, C.P. 6128, succ. Centre-ville, Montreal, H3C 3J7, Canada
pesant@crt.umontreal.ca

Abstract. This paper describes a global constraint on a fixed-length sequence of finite-domain variables requiring that the corresponding sequence of values taken by these variables belong to a given regular language, thereby generalizing some other known global constraints. We describe and analyze a filtering algorithm achieving generalized arc consistency for this constraint. Some comparative empirical results are also given.

1 Introduction

For constraint programming (CP) to become widely used as a problem modeling and solving methodology, it must be expressive but also powerful. Global constraints are an important step in that direction: they represent substructures commonly found in certain problems; they encapsulate efficient algorithms to reason about these substructures and about the rest of the problem through shared variables. This paper describes a global constraint on a fixed-length sequence of finite-domain variables requiring that the corresponding sequence of values taken by these variables belong to a given regular language[1]. Regular languages are a good compromise between expressiveness and computational efficiency for deciding membership. One finds such a substructure, for example, in rostering and car sequencing problems.

We briefly recall regular expressions, regular languages, and their connection with automata theory (the interested reader may consult, for example, [7]). An alphabet Σ is a finite set of symbols. A string over an alphabet is a finite sequence of symbols from that alphabet. The (infinite) set of all strings over Σ is denoted by $\Sigma^\star$. Any subset of $\Sigma^\star$ is called a language. Deciding whether a particular string belongs to a given language is easier for some languages than for others.

A regular expression over Σ is built from Σ and the symbols "(", ")", "ϵ", "+", and "$\star$", according to the following recursive definition:

- ϵ (the empty string) and each member of Σ is a regular expression;
- if α and β are regular expressions then so is $(\alpha\beta)$;

[1] A preliminary version of this paper appeared as [10].

- if α and β are regular expressions then so is $(\alpha + \beta)$;
- if α is a regular expression then so is $\alpha^\star$.

Every regular expression represents a regular language in $\Sigma^\star$, according to the interpretation of "+" and "$\star$" as set union and Kleene star, respectively.

A deterministic finite automaton (DFA) may be described by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a partial transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. Given an input string, the automaton starts in the initial state q_0 and processes the string one symbol at a time, applying the transition function δ at each step to update the current state. The string is accepted if and only if the last state reached belongs to the set of final states F. The languages recognized by DFA's are precisely regular languages.

We define our global constraint using a DFA — equivalently, we could have used a regular expression without any significant impact on computation time [3].

Definition 1 (regular language membership constraint).
Let $M = (Q, \Sigma, \delta, q_0, F)$ denote a deterministic finite automaton and $\mathbf{x}$ *a sequence of finite-domain variables $\langle x_1, x_2, \ldots, x_n \rangle$ with respective domains D_1, D_2, ..., $D_n \subseteq \Sigma$. Under a* regular language membership constraint `regular(`$\mathbf{x}, M$`)`*, any sequence of values taken by the variables of* $\mathbf{x}$ *must belong to the regular language recognized by M.*

Example 1. Consider a sequence $\mathbf{x}$ of five variables with $D_i = \Sigma = \{a, b, c\}$, $1 \leq i \leq 5$ and regular expression $aa^\star bb^\star aa^\star + c^\star$, equivalently described by the automaton M of Figure 1. Under constraint `regular(`$\mathbf{x}, M$`)`, assigned sequences $\langle a, a, b, b, a \rangle$ and $\langle c, c, c, c, c \rangle$ are valid but $\langle a, b, b, b, c \rangle$ and $\langle a, a, b, b, b \rangle$ are not.

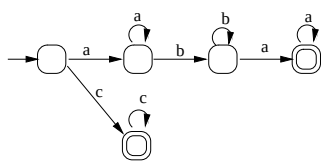

Fig. 1. A DFA corresponding to $aa^\star bb^\star aa^\star + c^\star$. Final states are shown with a double circle.

Such a definition may appear restrictive since it only allows the strings of the regular language that have a given length n. But because that constraint, like most other global constraints, is destined to be integrated into a larger model where other constraints are present, it is a much better choice to use a fixed-length sequence of variables each ranging over a finite alphabet than a single variable ranging over an infinite set of strings of arbitrary length.

In CP, combinatorial problems are typically modeled with several finite-domain variables. Constraints are then expressed on different subsets of these variables and cooperate through shared variables. Consider for example a rostering problem modeled with a two-dimensional array of variables, each taking their value from the set of possible activities (different work shifts, day off, training, and so forth). Each row, made up of seven variables, represents a week and each column, a day of the week. There could be a `regular` constraint on each individual row to indicate valid patterns of activities during a week and another `regular` constraint on the column for Monday to limit the number of Monday evening work shifts in a row. There could also be a cardinality constraint on weekend variables to ensure enough weekends off, and so on. Breaking up the roster into single-day chunks allows us to express the previous constraints directly from the original variables.

The next section presents related work on the integration of regular languages and patterns in CP. Section 3 describes the data structures and the filtering algorithm encapsulated in the regular language membership constraint, analyses the time and space complexity of the algorithm, and shows that it achieves generalized arc consistency. Section 4 discusses an improvement on the algorithm of the previous section. Section 5 shows how some important global constraints are special cases of this one. Finally, Section 6 presents some empirical results.

2 Related Work

In the literature, the study of regular language membership constraints has usually involved variables ranging over an infinite set of strings. We do not insist on such works and only mention them if they are embedded in CP systems. We rather concentrate on membership constraints for sequences of finite-domain variables.

The constraint logic programming language CLP($\Sigma^\star$) [14] introduces membership constraints in regular languages described by regular expressions, of the form "`X in` ρ", where `X` is a single variable. One important difference with the work described in this paper is that the domain of a variable is an infinite set, the strings in $\Sigma^\star$, as opposed to a finite set of symbols from Σ in our case. Its constraint solver is based on a flexible constraint scheduling strategy to ensure termination (which is cause for concern with infinite domains) and on a collection of deduction rules to determine the satisfaction of individual constraints. With an application to software project planning in mind, [6] discusses constraint reasoning over infinite sets of strings restricted to belong to regular languages. This allows the sets to be represented by finite automata on which operations such as intersection, concatenation, and negation are efficiently computed. Here again variables have an infinite domain.

Other constraint logic programming languages have offered some support to reason about sequences of symbols. PROLOG III [5] features equations on lists. Lists are built from constants, list variables and a concatenation operator. A length operator is also defined. CLP($\mathcal{S}$) [11] manipulates equations on strings,

built very much like PROLOG III's lists except that each string variable has a length parameter associated to it. The constraint solver exploits equalities and inequalities on these lengths to speed up the rule-based unification algorithm used for string equations.

Some constraints enforcing patterns of values have also been defined for finite-domain variables. ILOG Solver's `IlcTableConstraint` [8] takes as main arguments a sequence of n finite-domain variables and a set of n-tuples representing the valid assignments of values to these variables. For sets described by a regular expression, as in our case, the number of n-tuples would usually need to be very large and this possibly reflects on the computational complexity of the constraint. CHIP's `sequence` constraint [4] is a very expressive global constraint on the number of times a certain pattern of length ℓ appears in a sequence of variables. Patterns are expressed as sums and cardinalities of values taken by some of the ℓ variables in the subsequence. The constraint is not designed for (regular) patterns of arbitrary length over the whole sequence.

Finally, there is some work related to the approach used in this paper. In [2], a specific DFA is used to build the filtering algorithm of a lexicographic ordering constraint. Automata provide a more concise representation for a search tree defined over a constraint satisfaction problem in [13, 1]. In a dynamic programming approach to knapsack constraints, [12] builds and updates a graph whose structure is very similar to the one introduced in the next section.

3 The Consistency Algorithm

The idea behind the consistency algorithm for the `regular` constraint is to process the sequence $\mathbf{x}$ with the automaton M in a two-stage forward-backward manner, collecting for each variable x_i in $\mathbf{x}$ and each value $v_j \in D_i$ the set of states from Q that support variable-value pair (x_i, v_j). The sets are updated as the domains change and whenever one of these sets becomes empty, a value becomes unsupported and can be removed from a domain.

Let $\Sigma = \{v_1, v_2, \ldots, v_m\}$. The two-stage process is best seen as constructing a layered directed multigraph $(N^1, N^2, \ldots, N^{n+1}, A)$ where each layer $N^i = \{q_0^i, q_1^i, \ldots, q_{|Q|-1}^i\}$ contains a different node for each state of M and arcs only appear between consecutive layers. The graph is acyclic by construction. Arcs are related to variable-value pairs: an arc from q_k^i to q_ℓ^{i+1} is admissible for inclusion in A only if there exists some $v_j \in D_i$ such that $\delta(q_k, v_j) = q_\ell$. The arc is labeled with the value v_j allowing the transition between the two states. In the first layer the only node with outgoing arcs is q_0^1 since q_0 is the only initial state. Observe that it is enough, given a variable-value pair, to store an arc as the origin state: the destination state can be obtained by one application of the transition function. Accordingly, we maintain sets of states Q_{ij} to stand for the arcs related to variable-value pair (x_i, v_j).

Set Q_{ij} also acts as the support for variable x_i taking value v_j. As we will see later, multiset A is built to contain exactly the admissible arcs that belong to a path from q_0 in the first layer to a member of F in the last layer. When this

condition is met for some arc related to variable-value pair (x_i, v_j), we can be sure that there is at least one sequence of values taken from the respective current domains of the variables of $\mathbf{x}$, and with value v_j for variable x_i in particular, that belongs to the regular language. As long as Q_{ij} is not empty, we know that one such arc exists.

Given automaton M and sequence $\mathbf{x}$ of variables (with their respective domains) in the statement of the constraint, Algorithm 1 builds the graph by initializing our main data structure, the Q_{ij} sets. The incoming and outgoing arcs at each node of the graph are stored in $\mathit{inarcs}[i+1][k]$ and $\mathit{outarcs}[i][k]$, $1 \leq i \leq n,\ 0 \leq k \leq |Q| - 1$ as doubly-linked lists. Each arc in one structure is also linked to its twin in the other structure and to the Q_{ij} sets, to allow constant time updates. In addition, the in- and out-degree of each node is maintained in $\mathit{indeg}[i+1][k]$ and $\mathit{outdeg}[i][k]$, $1 \leq i \leq n,\ 0 \leq k \leq |Q| - 1$. All of these data structures are restorable upon backtracking. In the first phase, candidate arcs are collected by reaching forward from the initial state q_0 in the first layer using the domains of the x_i's. In the second phase, the arcs collected during the first phase are only kept if they can be reached backward from a state of F in the last layer. Finally, domains are possibly filtered if some Q_{ij}'s are empty.

Algorithm 2 is executed whenever some value v_j is removed from the domain of variable x_i in the course of the computation. Removing a value corresponds to removing one or several arcs in the graph (as many as there are supporting states in Q_{ij}). The in- and out-degrees of the corresponding nodes must then be updated accordingly (Algorithms 3 and 4). If some degree reaches zero, the node no longer belongs to a path from q_0 in the first layer to a member of F in the last layer: that information is propagated along such former paths going through that node and other domains are possibly filtered as a result.

Example 2. In rostering problems, the assignment of consecutive shifts must often follow certain patterns. Consider a sequence $\mathbf{x}$ of five variables with $D_1 = \{a, b, c, o\}$, $D_2 = \{b, o\}$, $D_3 = \{a, c, o\}$, $D_4 = \{a, b, o\}$, and $D_5 = \{a\}$. Consider also the following pattern: between a's and b's, a's and c's, or b's and c's, there should be at least one o; furthermore, a's followed by o's followed by c's is not allowed, and neither are b's followed by o's followed by a's nor c's followed by o's followed by b's. Figure 2 gives the corresponding DFA. (In the figures, we identify the states as integers, for readability.) Algorithm 1 builds the graphs of Fig. 3.

Non-empty support sets are

$Q_{1a} : \{1\}$, $Q_{1b} : \{1\}$, $Q_{1c} : \{1\}$, $Q_{1o} : \{1\}$,
$Q_{2o} : \{1, 2, 3, 4\}$,
$Q_{3a} : \{1, 5, 7\}$, $Q_{3c} : \{1, 6, 7\}$, $Q_{3o} : \{1, 5, 7\}$,
$Q_{4a} : \{1, 2, 5, 7\}$, $Q_{4o} : \{1, 2, 4, 5, 7\}$,
$Q_{5a} : \{1, 2, 5, 7\}$.

As a result, value b is removed from D_2 and D_4. Suppose now that value o is removed from D_4 in the course of the computation. Algorithm 2 updates the graph as shown in Figure 4, causing values b and c to be removed from D_1 and D_3 respectively.

```
procedure initialize():
{clear the data structures}
for all i ∈ {1, 2, ..., n} do
   for all j ∈ {1, 2, ..., m} do
      Q_ij := ∅;
   for all k ∈ {0, 1, ..., |Q| − 1} do
      outarcs[i][k] := ∅;
      outdeg[i][k] := 0;
      inarcs[i + 1][k] := ∅;
      indeg[i + 1][k] := 0;
   N_{i+1} := ∅;
{forward phase: accumulate}
N_1 := {q_0};
for i := 1 to n  do
   for all v_j ∈ D_i, q_k ∈ N_i do
      if δ(q_k, v_j) is defined then
         add q_k to Q_ij; {state q_k is a candidate for support}
         add δ(q_k, v_j) to N_{i+1};
{backward phase: validate}
N_{n+1} := N_{n+1} ∩ F;
for i := n downto 1 do
   for all q_k ∈ N_i do
      mark[k] := false;
   for all v_j ∈ D_i, q_k ∈ Q_ij do
      if δ(q_k, v_j) ∈ N_{i+1}  {state q_k confirmed as support} then
         add (q_k, δ(q_k, v_j)) to outarcs[i][k];
         increment outdeg[i][k];
         add (q_k, δ(q_k, v_j)) to inarcs[i + 1][δ(q_k, v_j)];
         increment indeg[i + 1][δ(q_k, v_j)];
         mark[k] := true;
      else
         remove q_k from Q_ij;
   for all q_k ∈ N_i do
      if mark[k] = false then
         remove q_k from N_i;
{clean up the domains}
for all i ∈ {1, 2, ..., n}, j ∈ {1, 2, ..., m} do
   if Q_ij = ∅ then
      remove value v_j from domain D_i;
```

Algorithm 1: Upon the constraint being posted, build the graph to initialize the data structures and filter out inconsistent values.

```
procedure propagate(i, j):
for all q_k ∈ Q_ij do
   remove (q_k, δ(q_k, v_j)) from outarcs[i][k];
   remove (q_k, δ(q_k, v_j)) from inarcs[i + 1][δ(q_k, v_j)];
   decrement_outdeg(i, k);
   decrement_indeg(i + 1, δ(q_k, v_j));
Q_ij := ∅;
```

Algorithm 2: Upon v_j being removed from D_i in the course of the computation, update the data structures and filter out inconsistent values.

```
procedure decrement_outdeg(i, k):
decrement outdeg[i][k];
if outdeg[i][k] = 0 and i > 1 then
   for all (q_ℓ, q_k) ∈ inarcs[i][k] do
      let v_j be the label of arc (q_ℓ, q_k);
      remove (q_ℓ, q_k) from outarcs[i − 1][ℓ];
      remove q_ℓ from Q_{i−1,j};
      if Q_{i−1,j} = ∅ then
         remove value v_j from domain D_{i−1};
      decrement_outdeg(i − 1, ℓ);
   inarcs[i][k] := ∅;
```

Algorithm 3: Decrementing the out-degree of node q_k^i.

```
procedure decrement_indeg(i, k):
decrement indeg[i][k];
if indeg[i][k] = 0 and i < n then
   for all (q_k, q_ℓ) ∈ outarcs[i][k] do
      let v_j be the label of arc (q_k, q_ℓ);
      remove (q_k, q_ℓ) from inarcs[i + 1][δ(q_k, v_j)];
      remove q_k from Q_ij;
      if Q_ij = ∅ then
         remove value v_j from domain D_i;
      decrement_indeg(i + 1, δ(q_k, v_j));
   outarcs[i][k] := ∅;
```

Algorithm 4: Decrementing the in-degree of node q_k^i.

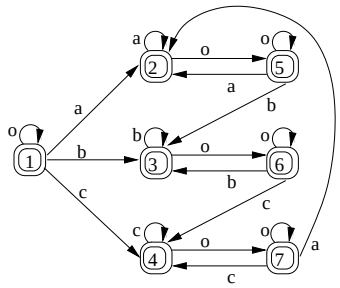

Fig. 2. A deterministic finite automaton for a common pattern in rostering.

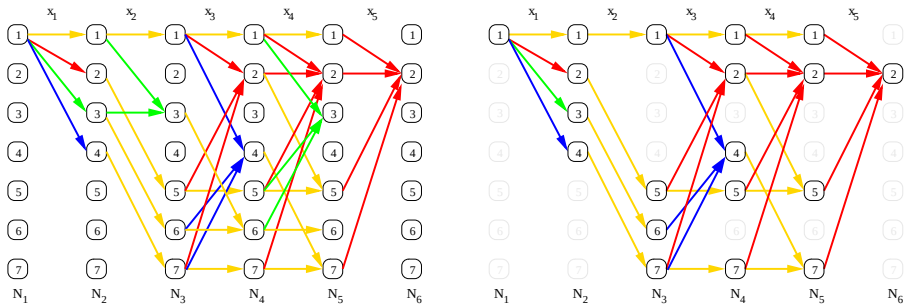

Fig. 3. The layered directed graph at the end of the forward (left) and backward (right) phases.

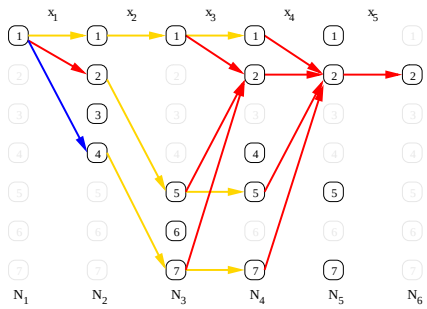

Fig. 4. The layered directed graph after removing value o from the domain of x_4.

We now show that the algorithms achieve domain consistency (i.e. generalized arc consistency). Let $L_{\mathbf{x}} = \{\langle a_1, a_2, \ldots, a_n\rangle \mid a_k \in D_k, 1 \leq k \leq n\}$ be the set of all sequences of symbols (strings) formed by replacing each variable of $\mathbf{x}$ by a value in its domain, and $L_{\mathbf{x}}(M) \subseteq L_{\mathbf{x}}$ the subset of these strings that are accepted by M. For convenience, we extend the definition of the transition function so that it applies to strings as well: $\delta(q_k, \langle a_1, a_2, \ldots, a_\ell\rangle) = \delta(\delta(\ldots\delta(\delta(q_k, a_1), a_2), \ldots), a_\ell)$. At the end of Algorithm 1, the following holds:

Lemma 1. $Q_{ij} = \{q \in Q \mid \exists\ \langle a_1, \ldots, a_{i-1}, v_j, a_{i+1}, \ldots, a_n\rangle \in L_{\mathbf{x}}(M)$ *with* $\delta(q_0, \langle a_1, \ldots, a_{i-1}\rangle) = q\}$.

Proof. Considering first the forward phase, it is easy to prove by induction on i that N_i contains all the nodes that may be reached by processing the first i characters of a word in $L_{\mathbf{x}}$. It follows that $Q_{ij} = \{q \in Q \mid \exists\ \langle a_1, \ldots, a_{i-1}, \ldots\rangle \in L_{\mathbf{x}}$ with $\delta(q_0, \langle a_1, \ldots, a_{i-1}\rangle) = q$ and $\delta(q, v_j)$ defined $\}$ since every possible state (corresponding to a node of N_i) is considered.

Similarly for the backward phase, we can prove by induction on i that N_i contains all the nodes that may reach F by processing the last $n-i+1$ characters of a word in $L_{\mathbf{x}}$. Hence a state q is kept in Q_{ij} during the backward phase if and only if $\exists\langle \ldots, v_j, a_{i+1}, \ldots, a_n\rangle \in L_{\mathbf{x}}$ such that $\delta(q, \langle \ldots, v_j, a_{i+1}, \ldots, a_n\rangle) \in F$.

Putting the two together, we get the result.

Theorem 1. *Constraint* `regular(`$\mathbf{x}, M$`)` *is domain consistent if and only if* $Q_{ij} \neq \emptyset,\ \forall\ 1 \leq i \leq n,\ j \in D_i$.

Proof. By the previous lemma, if a Q_{ij} set is non-empty then there is at least one sequence of values for variables $\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\rangle$ supporting the assignment of value v_j to variable x_i. If the sets are non-empty for every variable and every value in its domain then every variable-value pair has a support and the constraint is domain consistent.

If a Q_{ij} set is empty then by the same lemma there is no supporting path for that variable-value pair and the constraint cannot be domain consistent.

Since Algorithm 2 updates the graph throughout the computation in order to mirror changes in the domains, domain consistency is maintained.

We now turn to complexity analysis. The worst-case running time of Algorithm 1, called once, is dominated by the construction of the graph and is in $\mathcal{O}(nm|Q|)$. Its space complexity is in $\mathcal{O}(nm|Q|)$ as well. Each time a domain is modified, Algorithm 2 is called: for each state removed from Q_{ij}, the corresponding arc is removed from the *inarcs* and *outarcs* lists in constant time and Algorithms 3 and 4 are called. If the decremented degrees remains greater than zero, no further arcs are removed and these calls take constant time. Otherwise, for each new arc removed some constant time work is performed plus one recursive call. In all, the initial call of Algorithm 2 therefore takes constant time per arc removed from the graph, which is a good way to evaluate the amount of work performed at each call. It is actually a bounded incremental algorithm whose complexity is linearly related to the number of changes in the data structures.

Because the number of states of the automaton influences both the time and space complexity, it is desirable to use the smallest one possible. Computing a minimum state DFA can be achieved in $\mathcal{O}(|Q| \log |Q|)$ time [15].

4 Cutting Down on Supports

At first, it seems difficult to improve the time complexity of the incremental filtering algorithm since it is optimally efficient for each arc removed. But the numbers of arcs may be reduced. Because we maintain all supports, there may be many arcs stored in our data structures. We only really need one support per variable-value pair. Fewer arcs means less work in maintaining the data structures. Following this idea, we implemented another version of the consistency algorithm which maintains at least one support per variable-value pair and we sketch it below.

Starting from the graph outputted by Algorithm 1, we build a subgraph made up of supporting paths covering every consistent variable-value pair. Pairs obviously share paths and a pair may be covered by several paths. Heuristics help select arcs so that the collection of paths uses as few arcs as possible. Whenever a value is removed from a domain, those of the corresponding arcs from the original graph which belong to supporting paths disappear. Every such path is left broken in two pieces that must each be repaired with other arcs, if possible. Repairing them may involve undoing some of the path.

5 Instances of the `regular` constraint

Regular languages allow us to express many non trivial relationships between the variables of a sequence. In particular, a regular language membership constraint generalizes some known global constraints.

Definition 2 (stretch). *A* stretch *is a maximal subsequence of identical values in an assigned sequence* $\mathbf{x}$. *The* type *of a stretch is the value* v_j *taken by its variables and we denote it as* v_j*-stretch.*

The `stretch` constraint [9] puts restrictions on the length of stretches in a sequence. This can also be expressed as a regular language and encoded in an automaton using as many states for each value as the maximum length of the corresponding stretch. Minimum and maximum lengths for each type of stretch are expressed by choosing the final states appropriately. For example, the DFA of Fig. 5 enforces a-stretches of length 2 and b-stretches of length 2 or 3.

Sometimes the sequence of variables for a `stretch` constraint is considered circular, i.e. x_1 follows x_n. This could be problematic for our approach. Fortunately, because what must be verified is bounded above (in terms of the number of variables) by the largest maximum length of a stretch, we can model the circular case by duplicating and adding blocks of variables to the original sequence. We sketch this below.

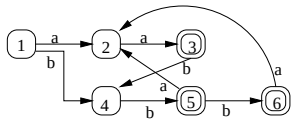

Fig. 5. A deterministic finite automaton for a `stretch` constraint.

Let ℓ be the largest maximum length among all types of stretches. We duplicate variables $x_1, x_2, \ldots, x_\ell$ (call this block **a**) and concatenate them at the end of the original sequence. Then we create two distinct blocks of $\ell - 1$ new variables each (call them **b** and **b**$'$) whose domains contain every possible value from the original sequence plus an extra dummy value whose stretches are unconstrained. Add one such block at each end of the modified sequence. The resulting sequence looks like **bxab**$'$. Block **a** allows any stretch begun near the end of **x** to continue from the beginning of **x** and blocks **b** and **b**$'$ ensure that we do not unnecessarily constrain the extremities of **xa**, allowing any first or last stretch to complete and padding with the dummy value.

Definition 3 (pattern). *We call* pattern *two or more consecutive stretches. A pattern made up of p stretches of type $v_1, v_2, \ldots, v_p$, in that order, we call a* p-pattern *and denote it by $[\![v_1 v_2 \cdots v_p]\!]$. It follows from the definition of a stretch that $v_j \neq v_{j+1}$, $1 \leq j \leq p-1$.*

Patterns are a proper subclass of regular expressions: an a-stretch can be described as $aa^\star$; the 3-pattern $[\![aob]\!]$, as $aa^\star oo^\star bb^\star$ (recall Ex. 2).

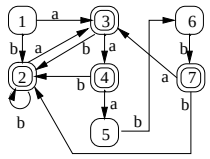

Fig. 6. A deterministic finite automaton for a patterned stretch constraint.

A mix of the previous two is sometimes requested in rostering problems: for example, "after three consecutive night shifts, a doctor should be off for at least the next two days". We will call such a constraint a *patterned stretch*. The corresponding regular language is represented by the automaton in Figure 6: any three consecutive a's must be followed by at least two b's.

Even the well-known `alldifferent` constraint could be expressed using `regular`. Since the alphabet Σ is finite, the language of strings made up of distinct symbols is necessarily finite as well (each string is of length at most $|\Sigma|$) and may be represented by a regular expression which, at the very least, takes

the union of the individual strings. For n variables and $m \geq n$ values, we require at least 2^n states, one for each subset of Σ of size at most n. A transition is allowed on symbol v_j from state-subset S (with $v_j \notin S$) to state-subset $S \cup \{v_j\}$, the initial state corresponds to the empty subset, and final states correspond to subsets of size n. However, such an approach clearly suffers from an exponential growth of the automaton and one would quickly run into memory consumption issues.

6 Empirical Results

In this section we wish to compare the computation time of the two implementations proposed for the regular language membership constraint. We also examine the possible trade-off between overall computation time and number of backtracks for these implementations and specialized filtering algorithms for special cases of regular languages.

			regular	
nb of vars	nb of values	nb of states	all supports time(s)	a support time(s)
25	5	10	0.002	0.002
		20	0.005	0.004
		40	0.009	0.008
		80	0.021	0.019
25	10	10	0.005	0.004
		20	0.011	0.007
		40	0.023	0.015
		80	0.045	0.032
25	20	10	0.011	0.008
		20	0.023	0.014
		40	0.045	0.028
		80	0.091	0.060
50	5	10	0.007	0.005
		20	0.012	0.010
		40	0.023	0.021
		80	0.051	0.050
50	10	10	0.013	0.010
		20	0.025	0.018
		40	0.051	0.037
		80	0.100	0.076
50	20	10	0.026	0.018
		20	0.051	0.033
		40	0.098	0.066
		80	0.196	0.133

Table 1. Comparing our two versions of `regular` on randomly generated automata.

In our experiments, we vary the number of variables in our sequence and the number of values in the initial domains. The variable and value selection heuristics we use simply choose a variable or value at random. Each line in a table represents an average over 30 runs.

We first compare our two implementations using randomly generated automata with a varying number of states (Table 1). The proportion of undefined transitions in δ was set to 30% and the proportion of final states to 50% (the value of the former has some influence on relative performance but the latter seems to have very little). Our second implementation steadily dominates the other but not spectacularly. There is probably still plenty of room for improvement in the way arcs are selected for supporting paths and such choices should have an impact on performance.

		stretch		regular	
nb of vars	nb of values	backtracks	time(s)	all supports time(s)	a support time(s)
25	3	2	0.002	0.003	0.006
	5	5	0.003	0.007	0.008
	7	40	0.006	0.015	0.013
	9	6	0.003	0.026	0.022
	11	183	0.019	0.039	0.035
50	3	4	0.007	0.007	0.008
	5	93	0.012	0.013	0.015
	7	15773	0.913	0.027	0.023
	9	34695	1.734	0.045	0.041
	11	512945	45.210	0.070	0.057
100	3	418729	62.178	0.014	0.017
	5	-	-	0.029	0.025
	7	-	-	0.051	0.051
	9	-	-	0.082	0.072
	11	-	-	0.129	0.108

Table 2. A comparison on stretch constraint instances.

Table 2 reports on instances of circular stretch constraints with randomly generated minimum and maximum stretch lengths. It compares the `stretch` constraint [9] (Columns 2 and 4) with our two implementations of `regular` (Columns 5 and 6). For the former, we report the average number of backtracks and the average computation time in seconds. For the latter, we only give the average computation time — there are never any backtracks since our filtering algorithms guarantee that a partial solution can always be completed. Entries marked '−' indicate that several of the runs could not be solved within one hour of computing time. We observe that in cases where few backtracks are performed with `stretch`, that implementation is faster. But as soon as the instances become harder, that specialized implementation significantly falls behind in computation

time. Comparing the two implementations of `regular` for stretch constraints, there is no clear dominance on these instances.

nb of vars	shiftchange backtracks	shiftchange time(s)	regular all supports time(s)	regular a support time(s)
25	0	0.002	0.003	0.003
50	0	0.004	0.004	0.005
100	0.3	0.005	0.013	0.009

Table 3. A comparison on pattern constraint instances.

Table 3 reports on a few experiments with the pattern described in Example 2. It compares a sophisticated specialized filtering algorithm previously designed by the author to our versions of `regular` with the automaton of Figure 2. The specialized algorithm performs better on these instances but, as observed in the "backtracks" column, it does not achieve domain consistency. More difficult instances mixed with other constraints would probably provide a more accurate picture.

7 Conclusion

This paper introduced an expressive global constraint that can help model complex sequencing rules present in many problems. A filtering algorithm that achieves generalized arc consistency was described for the constraint. Its theoretical complexity was analyzed and its practical efficiency was demonstrated against other special purpose filtering algorithms. From a user interface point of view, it would probably benefit from front-ends designed for particular contexts (such as stretch or pattern constraints) that would feel more natural than a regular expression to the ordinary user and that would automatically translate it into the appropriate DFA.

Even though regular languages already offer a very rich formalism to express constraints on sequences of variables, as exemplified by the previous global constraints equivalently modeled by `regular`, we can actually go beyond regular languages with that same constraint. This is because the finiteness of the sequence of variables yields a finite language of finite strings. In the extreme, any finite language, regular or not, is easily given in extension as a regular expression. Hence membership can be enforced with a `regular` constraint. However the number of states needed in the automaton can become very large, making it inefficient to use that constraint for such a purpose. A natural question is whether membership constraints for richer formal languages, such as context-free grammars, could be efficiently implemented on sequences of finite-domain variable but the usefulness of their additional expressiveness should first be established.

Acknowledgements

The author wishes to thank Geneviève Hernu for implementing part of the constraint and conducting the experiments, and anonymous referees for their comments on an earlier version of this paper. This work was partially supported by the Canadian Natural Sciences and Engineering Research Council under grant OGP0218028.

References

1. J. Amilhastre, H. Fargier, and P. Marquis. Consistency Restoration and Explanations in Dynamic CSPs – Application to Configuration. *Artificial Intelligence*, 135:199–234, 2002.
2. M. Carlsson and N. Beldiceanu. Revisiting the Lexicographic Ordering Constraint. Technical Report T2002:17, SICS, 2002. 13 p.
3. C. Chang and R. Paige. From Regular Expressions to DFA's Using Compressed NFA's. *Theoretical Computer Science*, 178:1–36, 1997.
4. The Sequence Global Constraint of CHIP. Technical Report COSY/SEQ/032, COSYTEC, Orsay, France, 1999.
5. A. Colmerauer. An Introduction to PROLOG III. *Communications of the ACM*, 33(7):69–90, 1990.
6. K. Golden and W. Pang. Constraint Reasoning over Strings. In *Principles and Practice of Constraint Programming – CP 2003: Proceedings of the Ninth International Conference*, pages 377–391. Springer-Verlag LNCS 2833, 2003.
7. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
8. ILOG S.A., Gentilly, France. *ILOG Solver Reference Manual, version 4.4*, 1999.
9. G. Pesant. A Filtering Algorithm for the Stretch Constraint. In *Principles and Practice of Constraint Programming – CP 2001: Proceedings of the Seventh International Conference*, pages 183–195. Springer-Verlag LNCS 2239, 2001.
10. G. Pesant. A Regular Language Membership Constraint for Sequences of Variables. In *Proc. Second International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, Principles and Practice of Constraint Programming (CP'03)*, pages 110–119, Kinsale, Ireland, 2003.
11. A. Rajasekar. Applications in Constraint Logic Programming with Strings. In *Principles and Practice of Constraint Programming: Proc. Second International Workshop*, pages 109–122, Rosario, Orcas Island, Washington, USA, 1994. Springer-Verlag LNCS 874.
12. M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.
13. N.R. Vempaty. Solving Constraint Satisfaction Problems Using Finite State Automata. In *Proc. National Conference on Artificial Intelligence (AAAI-92)*, pages 453–458. AAAI Press, 1992.
14. C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 181–196, Lisbon, Portugal, 1989. MIT Press.
15. B.W. Watson. A taxonomy of finite automata minimization algorithms. Technical Report Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.