From Constraints to Finite Automata to Filtering Algorithms

Mats Carlsson[1] and Nicolas Beldiceanu[2,3]

[1] SICS, P.O. Box 1263, SE-752 37 KISTA, Sweden
matsc@sics.se

[2] LINA FRE CNRS 2729
École des Mines de Nantes
La Chantrerie
4, rue Alfred Kastler, B.P. 20722
FR-44307 NANTES Cedex 3, France
Nicolas.Beldiceanu@emn.fr

[3] This research was carried out while N. Beldiceanu was at SICS.

Abstract. We introduce an approach to designing filtering algorithms by derivation from finite automata operating on constraint signatures. We illustrate this approach in two case studies of constraints on vectors of variables. This has enabled us to derive an incremental filtering algorithm that runs in $O(n)$ plus amortized $O(1)$ time per propagation event for the lexicographic ordering constraint over two vectors of size n, and an $O(nmd)$ time filtering algorithm for a chain of $m-1$ such constraints, where d is the cost of certain domain operations. Both algorithms maintain hyperarc consistency. Our approach can be seen as a first step towards a methodology for semi-automatic development of filtering algorithms.

1 Introduction

The design of filtering algorithms for global constraints is one of the most creative endeavors in the construction of a finite domain constraint programming system. It is very much a craft and requires a good command of e.g. matching theory [1], flow theory [2] scheduling theory [3], or combinatorics [4], in order to successfully bring to bear results from these areas on specific constraints. As a first step towards a methodology for semi-automatic development of filtering algorithms, we introduce an approach to designing filtering algorithms by derivation from finite automata operating on constraint signatures, an approach that to our knowledge has not been used before. We illustrate this approach in two case studies of constraints on vectors of variables, for which we have developed one filtering algorithm for $\vec{x} \leq_{\text{lex}} \vec{y}$, the lexicographic ordering constraint over two vectors $\vec{x}$ and $\vec{y}$, and one filtering algorithm for lex_chain, a chain of $\leq_{\text{lex}}$ constraints.

The rest of the article is organized as follows: We first define some necessary notions and notation. We proceed with the two case studies: Sect. 3 treats $\leq_{\text{lex}}$, and Sect. 4 applies the approach to lex_chain, or more specifically to the constraint $\vec{a} \leq_{\text{lex}} \vec{x} \leq_{\text{lex}}$

$\vec{b}$, where $\vec{a}$ and $\vec{b}$ are vectors of integers. This latter constraint is the central building-block of $\mathrm{lex\_chain}$. Filtering algorithms for these constraints are derived. After quoting related work, we conclude with a discussion.

For reasons of space, lemmas and propositions are given with proofs omitted. Full proofs and pseudocode algorithms can be found in [5] and [6]. The algorithms have been implemented and are part of the CLP(FD) library of SICStus Prolog [7].

2 Preliminaries

We shall use the following notation: $[i, j]$ stands for the interval $\{v \mid i \leq v \leq j\}$; $[i, j)$ is a shorthand for $[i, j-1]$; (i, j) is a shorthand for $[i+1, j-1]$; the *subvector* of $\vec{x}$ with start index i and last index j is denoted by $\vec{x}_{[i,j]}$.

A *constraint store* (X, D) is a set of variables, and for each variable $x \in X$ a domain $D(x)$, which is a finite set of integers. In the context of a current constraint store: $\underline{x}$ denotes $\min(D(x))$; $\overline{x}$ denotes $\max(D(x))$; $\mathrm{next\_value}(x, a)$ denotes $\min\{i \in D(x) \mid i > a\}$, if it exists, and $+\infty$ otherwise; and $\mathrm{prev\_value}(x, a)$ denotes $\max\{i \in D(x) \mid i < a\}$, if it exists, and $-\infty$ otherwise. The former two operations run in constant time whereas the latter two have cost d[1]. If for $\Gamma = (X, D)$ and $\Gamma' = (X, D')$, $\forall x \in X : D'(x) \subseteq D(x)$, we say that $\Gamma' \sqsubseteq \Gamma$, Γ' is *tighter* than Γ.

The constraint store is *pruned* by applying the following operations to a variable x: $\mathrm{fix\_interval}(x, a, b)$ removes from $D(x)$ any value that is not in $[a, b]$, and $\mathrm{prune\_interval}(x, a, b)$ removes from $D(x)$ any value that is in $[a, b]$. Each operation has cost d and succeeds iff $D(x)$ remains non-empty afterwards.

For a constraint C, a variable x mentioned by C, and a value v, the assignment $x = v$ *has support* iff $v \in D(x)$ and C has a solution such that $x = v$. A constraint C is *hyperarc consistent* iff, for each such variable x and value $v \in D(x)$, $x = v$ has support. A filtering algorithm maintains hyperarc consistency of C iff it removes any value $v \in D(x)$ such that $x = v$ does not have support. By convention, a filtering algorithm returns one of: $fail$, if it discovers that there are no solutions; $succeed$, if it discovers that C will hold no matter what values are taken by any variables that are still nonground; and $delay$ otherwise.

A constraint satisfaction problem (CSP) consists of a set of variables and a set of constraints connecting these variables. The solution to a CSP is an assignment of values to the variables that satisfies all constraints. In solving a CSP, the constraint solver repeatedly calls the filtering algorithms associated with the constraints. The removal by a filtering algorithm of a value from a domain is called a *propagation event*, and usually leads to the resumption of some other filtering algorithms. The constraint kernel ensures that all propagation events are eventually served by the relevant filtering algorithms.

A *string* S over some alphabet A is a finite sequence $\langle S_0, S_1, \ldots \rangle$ of letters chosen from A. A *regular expression* E denotes a *regular language* $L(E)$, i.e. a subset of all the possible strings over A, recursively defined as usual: a single letter a denotes the language with the single string $\langle a \rangle$; EE' denotes $L(E)L(E')$ (concatenation); $E \mid E'$ denotes $L(E) \cup L(E')$ (union); and $E^\star$ denotes $L(E)^\star$ (closure). Parentheses are used for grouping.

[1] E.g. if a domain is represented by a bit array, d is linear in the size of the domain.

Let $\mathcal{A}$ be an alphabet, C a constraint over vectors of length n, and Γ a constraint store. We will associate to C a string $\sigma(C, \Gamma, \mathcal{A})$ over $\mathcal{A}$ of length $n+1$ called the *signature of* C.

3 Case Study: $\leq_{\text{lex}}$

Given two vectors, $\vec{x}$ and $\vec{y}$ of n variables, $\langle x_0, \ldots, x_{n-1}\rangle$ and $\langle y_0, \ldots, y_{n-1}\rangle$, let $\vec{x} \leq_{\text{lex}} \vec{y}$ denote the lexicographic ordering constraint on $\vec{x}$ and $\vec{y}$. The constraint holds iff $n = 0$ or $x_0 < y_0$ or $x_0 = y_0$ and $\langle x_1, \ldots, x_{n-1}\rangle \leq_{\text{lex}} \langle y_1, \ldots, y_{n-1}\rangle$. Similarly, the constraint $\vec{x} <_{\text{lex}} \vec{y}$ holds iff $x_0 < y_0$ or $x_0 = y_0$ and $\langle x_1, \ldots, x_{n-1}\rangle <_{\text{lex}} \langle y_1, \ldots, y_{n-1}\rangle$. We now present an alphabet and a finite automaton for this constraint, and an incremental filtering algorithm.

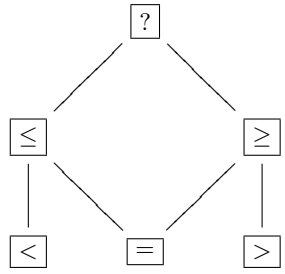
3.1 Signatures

Let $\mathcal{A}$ be the alphabet $\{\boxed{<}, \boxed{=}, \boxed{>}, \boxed{\leq}, \boxed{\geq}, \boxed{?}, \boxed{\$}\}$. It is worth noting that each symbol except $\boxed{\$}$ corresponds to a subset of the fundamental arithmetic relations. The signature $S = \sigma(C, \Gamma, \mathcal{A})$ of a constraint $C \equiv \vec{x} \leq_{\text{lex}} \vec{y}$ wrt. a constraint store Γ is defined by $S_n = \boxed{\$}$, to mark the end of the string, and for $0 \leq i < n$:

$$S_i = \begin{cases} \boxed{<}, \text{ if } \Gamma \models x_i < y_i \\ \boxed{=}, \text{ if } \Gamma \models x_i = y_i \\ \boxed{>}, \text{ if } \Gamma \models x_i > y_i \\ \boxed{\leq}, \text{ if } \Gamma \models x_i \leq y_i \wedge \Gamma \not\models x_i < y_i \wedge \Gamma \not\models x_i = y_i \\ \boxed{\geq}, \text{ if } \Gamma \models x_i \geq y_i \wedge \Gamma \not\models x_i > y_i \wedge \Gamma \not\models x_i = y_i \\ \boxed{?}, \text{ if } \Gamma \text{ does not entail any relation on } x_i, y_i \end{cases}$$

From a complexity point of view, it is important to note that the tests $\Gamma \models x_i \circ y_i$ where $\circ \in \{<, \leq, =, \geq, >\}$ can be implemented by domain bound inspection, and are all $O(1)$ in any reasonable domain representation; see left part of Fig. 1.

S_i	**Condition**
$\boxed{<}$	$\overline{x_i} < \underline{y_i}$
$\boxed{=}$	$\underline{x_i} = \overline{x_i} = \underline{y_i} = \overline{y_i}$
$\boxed{>}$	$\underline{x_i} > \overline{y_i}$
$\boxed{\leq}$	$\overline{x_i} = y_i \wedge \underline{x_i} < \overline{y_i}$
$\boxed{\geq}$	$\overline{y_i} = \underline{x_i} \wedge \underline{y_i} < \overline{x_i}$
$\boxed{?}$	otherwise



$D(x)$	$D(y)$	**Signature letter**
$\{0,1\}$	$\{0,1\}$	$\boxed{?}$
$\{0\}$	$\{0,1\}$	$\boxed{\leq}$
$\{0\}$	$\{1\}$	$\boxed{<}$

Fig. 1. Computing the signature letter at position i

The letters of $\mathcal{A}$ (except $\boxed{\$}$) form the partially ordered set $(\mathcal{A}, \preceq)$ of Fig. 1. For all $\leq_{\text{lex}}$ constraints C and all i $(0 \leq i < n)$, we have that:

$$\Gamma' \sqsubseteq \Gamma \Rightarrow \sigma(C, \Gamma', \mathcal{A})_i \preceq \sigma(C, \Gamma, \mathcal{A})_i$$

The right part of Fig. 1 also illustrates how a signature letter becomes more ground (smaller wrt. $\preceq$) as the constraint store gets tighter.

3.2 Finite Automaton

Fig. 2 shows a deterministic finite automaton LFA for signature strings, from which we shall derive the filtering algorithm, and the automaton at work on a small example. State 1 is the initial state. There are seven terminal states, F, T1–T3 and D1–D3, each corresponding to a separate case. Case F is the failure case; cases T1–T3 are cases where the algorithm detects that either C must hold or C can be replaced by a $<$ or a $\leq$ constraint; cases D1–D3 are cases where ground instances of C can be either true or false, and so the algorithm must suspend.

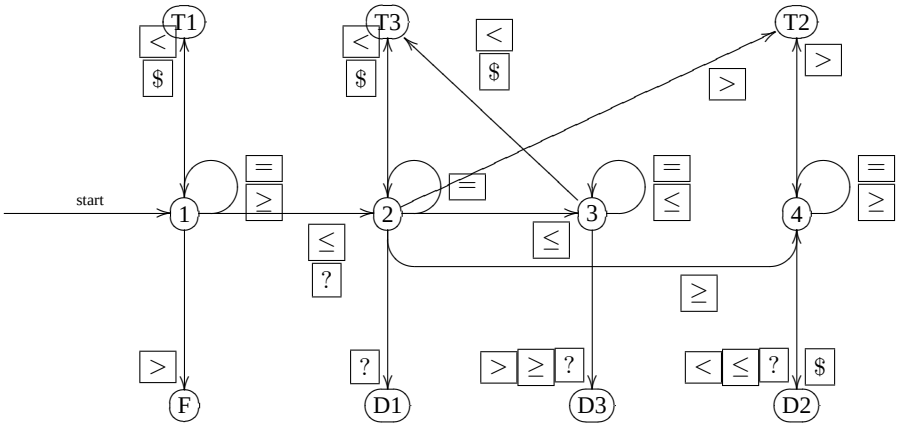


Fig. 2. Case analysis of $\leq_{\text{lex}}$ as finite automaton LFA and an example, where the automaton stops in state T3, detecting entailment.

3.3 Case Analysis

We now discuss seven regular expressions covering all possible cases of signatures of C. Where relevant, we also derive pruning rules for maintaining hyperarc consistency. Each regular expression corresponds to one of the terminal states of LFA. Note that, without loss of generality, each regular expression has a common prefix $P = (\boxed{=} \mid \boxed{\geq})^\star$. For C to hold, clearly for each position $i \in P$ where $S_i = \boxed{\geq}$, we must enforce $x_i = y_i$. We assume that the filtering algorithm does so in each case. In the regular expressions, q denotes the position of the transition out of state 1, r denotes the position of the transition out of state 2, and s denotes the position of the transition out of state 3 or 4. We now discuss the cases one by one.

Case F.

$$(\boxed{=} \mid \boxed{\geq})^\star \boxed{>} \mathcal{A}^\star \tag{F}$$

Clearly, if the signature of C is accepted by F, the signature of any ground instance will contain a $\boxed{>}$ before the first $\boxed{<}$, if any, so C has no solution.

Case T1.

$$\underbrace{(\boxed{=} \mid \boxed{\geq})^\star}_{P} \underbrace{(\boxed{<} \mid \boxed{\$})}_{q} \mathcal{A}^\star \tag{T1}$$

C will hold; we are done.

Case T2.

$$\underbrace{(\boxed{=} \mid \boxed{\geq})^\star}_{P} \underbrace{(\boxed{\leq} \mid \boxed{?})}_{q} (\boxed{=} \mid \boxed{\geq})^\star \boxed{>} \mathcal{A}^\star \tag{T2}$$

For C to hold, we must enforce $x_q < y_q$, in order for there to be at least one $\boxed{<}$ preceding the first $\boxed{>}$ in any ground instance.

Case T3.

$$\underbrace{(\boxed{=} \mid \boxed{\geq})^\star}_{P} \underbrace{(\boxed{\leq} \mid \boxed{?})}_{q} (\boxed{=} \mid \boxed{\leq})^\star (\boxed{<} \mid \boxed{\$}) \mathcal{A}^\star \tag{T3}$$

For C to hold, all we have to do is to enforce $x_q \leq y_q$.

Case D1.

$$\underbrace{(\boxed{=} \mid \boxed{\geq})^\star}_{P} \underbrace{(\boxed{\leq} \mid \boxed{?})}_{q} \boxed{=}^\star \underbrace{\boxed{?}}_{r} \mathcal{A}^\star \tag{D1}$$

Consider the possible ground instances. Suppose that $x_q > y_q$. Then C is false. Suppose instead that $x_q < y_q$. Then C holds no matter what values are taken at r. Suppose instead that $x_q = y_q$. Then C is false iff $x_r > y_r$. Thus, the only relation at q and r that doesn't have support is $x_q > y_q$, so we enforce $x_q \leq y_q$.

Case D2.

$$\underbrace{(\boxed{=} \mid \boxed{\geq})^{\star}}_{P}\ \underbrace{(\boxed{\leq} \mid \boxed{?})}_{q}\ \boxed{=}^{\star}\ \underbrace{\boxed{\geq}}_{r}\ (\boxed{=} \mid \boxed{\geq})^{\star}\ \underbrace{(\boxed{<} \mid \boxed{\leq} \mid \boxed{?} \mid \boxed{\$})}_{s}\ \mathcal{A}^{\star} \quad \text{(D2)}$$

Consider the possible ground instances. Suppose that $x_q > y_q$. Then C is false. Suppose instead that $x_q < y_q$. Then C holds no matter what values are taken in $[r, s]$. Suppose instead that $x_q = y_q$. Then C is false iff $x_r > y_r \vee \cdots \vee x_{s-1} > y_{s-1} \vee (s < n \wedge x_s > y_s)$. Thus, the only relation in $[q, s]$ that doesn't have support is $x_q > y_q$, so we enforce $x_q \leq y_q$.

Case D3.

$$\underbrace{(\boxed{=} \mid \boxed{\geq})^{\star}}_{P}\ \underbrace{(\boxed{\leq} \mid \boxed{?})}_{q}\ \boxed{=}^{\star}\ \underbrace{\boxed{\leq}}_{r}\ (\boxed{=} \mid \boxed{\leq})^{\star}\ \underbrace{(\boxed{>} \mid \boxed{\geq} \mid \boxed{?})}_{s}\ \mathcal{A}^{\star} \quad \text{(D3)}$$

Consider the possible ground instances. Suppose that $x_q > y_q$. Then C is false. Suppose instead that $x_q < y_q$. Then C holds no matter what values are taken in $[r, s]$. Suppose instead that $x_q = y_q$. Then C is false iff $x_r = y_r \wedge \cdots \wedge x_{s-1} = y_{s-1} \wedge x_s > y_s$. Thus, the only relation in $[q, s]$ that doesn't have support is $x_q > y_q$, so we enforce $x_q \leq y_q$.

3.4 Non-Incremental Filtering Algorithm

By augmenting LFA with the pruning actions mentioned in Sect. 3.3, we arrive at a filtering algorithm for $\leq_{\text{lex}}$, `FiltLex`. When a constraint is posted, the algorithm will succeed, fail or delay, depending on where LFA stops. In the delay case, the algorithm will restart from scratch whenever a propagation event (a bounds adjustment) arrives, until it eventually succeeds or fails. We summarize the properties of `FiltLex` in the following proposition.

Proposition 1.

1. `FiltLex` *covers all cases of* $\leq_{\text{lex}}$.
2. `FiltLex` *doesn't remove any solutions.*
3. `FiltLex` *doesn't admit any non-solutions.*
4. `FiltLex` *never suspends when it could in fact decide, from inspecting domain bounds, that the constraint is necessarily true or false.*
5. `FiltLex` *maintains hyperarc consistency.*
6. `FiltLex` *runs in* $O(n)$ *time.*

3.5 Incremental Filtering Algorithm

In a tree search setting, it is reasonable to assume that each variable is fixed one by one after posting the constraint. In this scenario, the total running time of `FiltLex` for reaching a leaf of the search tree would be $O(n^2)$. We can do better than that. In this section, we shall develop incremental handling of propagation events so that the

total running time is $O(n + m)$ for handling m propagation events after posting the constraint.

Assume that a $C \equiv \vec{x} \leq_{\text{lex}} \vec{y}$ constraint has been posted, `FiltLex` has run initially, has reached one of its suspension cases, possibly after some pruning, and has suspended, recording: the state $u \in \{2, 3, 4\}$ that preceded the suspension, and the positions q, r, s. Later on, a propagation event arrives on a variable x_i or y_i, i.e. one or more of $\underline{x_i}, \overline{x_i}, \underline{y_i}$ and $\overline{y_i}$ have changed.

We assume that updates of the constraint store and of the variables u, q, r, s are trailed [8], so that their old values can be restored on backtracking. Thus whenever the algorithm resumes, the constraint store will be tighter than last time (modulo backtracking). We shall now discuss the various cases for handling the event.

Naive Event Handling Our first idea is to simply restart the automaton at position i, in state u. The reasoning is that either everything up to position i is unchanged, or there is a pending propagation event at position $j < i$, which will be dealt with later:

– $i \in P$ is impossible, for after enforcing $x_i = y_i$ for all $i \in P$, all those variables are ground. This follows from the fact that:

$$\begin{array}{l} \underline{x_i} = \overline{x_i} = \underline{y_i} = \overline{y_i}, \text{ if } \Gamma \models x_i = y_i \\ \underline{x_i} = \overline{y_i}, \text{ if } \Gamma \models x_i \geq y_i \end{array} \tag{1}$$

for any constraint store Γ.
– If $i = q$, we resume in state 1 at position i.
– If $i = r$, we resume in state 2 at position i.
– If $u > 2 \wedge i = s$, we resume in state u at position i.
– If $u > 2 \wedge r < i < s$:
  • If the signature letter at position i is unchanged or is changed to $\boxed{=}$, we do nothing.
  • Otherwise, we resume in state u at position i, immediately reaching a terminal state.
– Otherwise, we just suspend, as LFA would perform the same transitions as last time.

Better Event Handling The problem with the above event handling scheme is that if $i = q$, we may have to re-examine any number of signature letters in states 2, 3 and 4 before reaching a terminal state. Similarly, if $i = r$, we may have to re-examine any number of positions in states 3 and 4. Thus, the worst-case total running time remains $O(n^2)$. We can remedy this problem with a simple device: when the finite automaton resumes, it simply ignores the following positions:

– In state 2, any letter before position r is ignored. This is safe, for the ignored letters will all be $\boxed{=}$.
– In states 3 and 4, any letter before position s is ignored. Suppose that there is a pending propagation event with position j, $r < j < s$ and that S_j has changed to $\boxed{<}$ (in state 3) or $\boxed{>}$ (in state 4), which should take the automaton to a terminal state. The pending event will lead to just that, when it is processed.

Incremental Filtering Algorithm Let `FiltLexI` be the `FiltLex` algorithm augmented with the event handling described above. As before, we assume that each time the algorithm resumes, the constraint store will be tighter than last time. We summarize the properties of `FiltLexI` in Proposition 2.

Proposition 2.

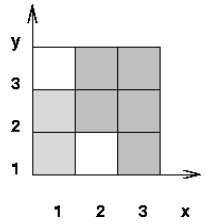
1. `FiltLex` *and* `FiltLexI` *are equivalent.*
2. *The total running time of* `FiltLexI` *for posting a* $\leq_{\text{lex}}$ *constraint followed by* m *propagation events is* $O(n+m)$.

4 Case Study: lex_chain

In this section, we consider a chain of $\leq_{\text{lex}}$ constraints, $\text{lex\_chain}(\vec{x}_0, \ldots, \vec{x}_{m-1}) \equiv \vec{x}_0 \leq_{\text{lex}} \cdots \leq_{\text{lex}} \vec{x}_{m-1}$. As mentioned in [9], chains of lexicographic ordering constraints are commonly used for breaking symmetries arising in problems modelled with matrices of decision variables. The authors conclude that finding an hyperarc consistency algorithm for lex_chain "may be quite challenging". This section addresses this open question. Our contribution is a filtering algorithm for lex_chain, which maintains hyperarc consistency and runs in $O(nmd)$ time per invocation, where d is the cost of certain domain operations (see Sect. 2).

The key idea of the filtering algorithm is to compute feasible lower and upper bounds for each vector $\vec{x}_i$, and to prune the domains of the individual variables wrt. these bounds. Thus at the heart of the algorithm is the ancillary constraint $\text{between}(\vec{a}, \vec{x}, \vec{b})$, which is a special case of a conjunction of two $\leq_{\text{lex}}$ constraints. The point is that we have to consider globally both the lower and upper bound, lest we miss some pruning, as illustrated by Fig. 3.

We devote most of this section to the between constraint, applying the finite automaton approach to it. We then give some additional building blocks required for a filtering algorithm for lex_chain, and show how to combine it all.



$x \in 1..3$
$y \in 1..3$
$\text{between}(\langle 1,3\rangle, \langle x,y\rangle, \langle 2,1\rangle)$

Fig. 3. The between constraint. $\langle 1,3\rangle \leq_{\text{lex}} \langle x,y\rangle \leq_{\text{lex}} \langle 2,1\rangle$ has no solution for $y=2$, but the conjunction of the two $\leq_{\text{lex}}$ constraints doesn't discover that.

4.1 Definition and Declarative Semantics of $\mathrm{between}$

Given two vectors, $\vec{a}$ and $\vec{b}$ of n integers, and a vector $\vec{x}$ of n variables, let $C \equiv \mathrm{between}(\vec{a}, \vec{x}, \vec{b})$ denote the constraint $\vec{a} \leq_{\mathrm{lex}} \vec{x} \leq_{\mathrm{lex}} \vec{b}$.

For technical reasons, we will need to work with tight, i.e. lexicographically largest and smallest, as well as feasible wrt. $\vec{x}$[2], versions $\vec{a'}$ and $\vec{b'}$ of $\vec{a}$ and $\vec{b}$, i.e.:

$$\forall i \in [0, n) : a'_i \in D(x_i) \wedge b'_i \in D(x_i) \tag{2}$$

This is not a problem, for under these conditions, the $\mathrm{between}(\vec{a}, \vec{x}, \vec{b})$ and $\mathrm{between}(\vec{a'}, \vec{x}, \vec{b'})$ constraints have the same set of solutions. Algorithms for computing $\vec{a'}$ and $\vec{b'}$ from $\vec{a}$, $\vec{b}$ and $\vec{x}$ are developed in Sect. 4.6.

It is straightforward to see that the declarative semantics is:

$$C \equiv \bigvee \begin{cases} n = 0 & (3.1) \\ a'_0 = x_0 = b'_0 \wedge \vec{a'}_{[1,n)} \leq_{\mathrm{lex}} \vec{x}_{[1,n)} \leq_{\mathrm{lex}} \vec{b'}_{[1,n)} & (3.2) \\ a'_0 = x_0 < b'_0 \wedge \vec{a'}_{[1,n)} \leq_{\mathrm{lex}} \vec{x}_{[1,n)} & (3.3) \\ a'_0 < x_0 = b'_0 \wedge \vec{x}_{[1,n)} \leq_{\mathrm{lex}} \vec{b'}_{[1,n)} & (3.4) \\ a'_0 < x_0 < b'_0 & (3.5) \end{cases} \tag{3}$$

and hence, for all $i \in [0, n)$:

$$C \wedge (a'_0 = b'_0) \wedge \cdots \wedge (a'_{i-1} = b'_{i-1}) \Rightarrow a'_i \leq x_i \leq b'_i \tag{4}$$

4.2 Signatures of $\mathrm{between}$

Let $\mathcal{B}$ be the alphabet $\{\boxed{<}, \boxed{\hat{<}}, \boxed{=}, \boxed{\hat{=}}, \boxed{>}, \boxed{\hat{>}}, \boxed{\$}\}$. The signature $S = \sigma(C, \Gamma, \mathcal{B})$ of C wrt. a constraint store Γ is defined by $S_n = \boxed{\$}$, to mark the end of the string, and for $0 \leq i < n$:

$$S_i = \begin{cases} \boxed{<}, \text{ if } a'_i < b'_i \wedge \Gamma \models (x_i \leq a'_i \vee x_i \geq b'_i) \\ \boxed{\hat{<}}, \text{ if } a'_i < b'_i \wedge \Gamma \not\models (x_i \leq a'_i \vee x_i \geq b'_i) \\ \boxed{=}, \text{ if } a'_i = b'_i \wedge \Gamma \models a'_i = x_i = b'_i \\ \boxed{\hat{=}}, \text{ if } a'_i = b'_i \wedge \Gamma \not\models a'_i = x_i = b'_i \\ \boxed{>}, \text{ if } a'_i > b'_i \wedge \Gamma \models b'_i \leq x_i \leq a'_i \\ \boxed{\hat{>}}, \text{ if } a'_i > b'_i \wedge \Gamma \not\models b'_i \leq x_i \leq a'_i \end{cases}$$

From a complexity point of view, we note that the tests $\Gamma \models a'_i = x_i = b'_i$ and $\Gamma \models b'_i \leq x_i \leq a'_i$ can be implemented with domain bound inspection and run in constant time, whereas the test $\Gamma \models (x_i \leq a'_i \vee x_i \geq b'_i)$ requires the use of $\mathrm{next\_value}$ or $\mathrm{prev\_value}$, and has cost d; see Table 1.

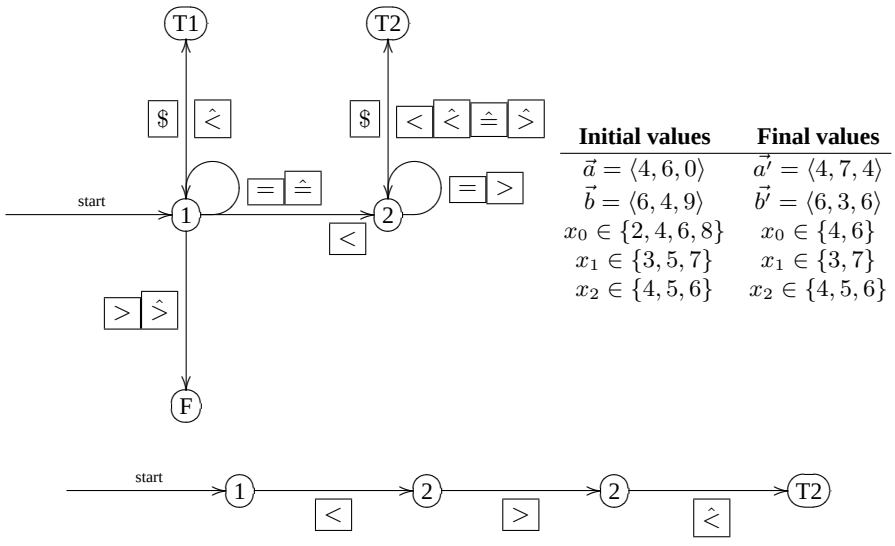
[2] The adjective *feasible* refers to the requirement that $\vec{a'}$ and $\vec{b'}$ be instances of $\vec{x}$.

Table 1. Computing the signature letter at position i. Note that if $a < b$ then $\mathrm{next\_value}(x, a) \geq b$ holds iff $D(x)$ has no value in (a, b).

S_i	**Condition**
$<$	$a'_i < b'_i \wedge \mathrm{next\_value}(x_i, a'_i) \geq b'_i$
$\hat{<}$	$a'_i < b'_i \wedge \mathrm{next\_value}(x_i, a'_i) < b'_i$
$=$	$\underline{x_i} = \overline{x_i} = a'_i = b'_i$
$\hat{=}$	$\underline{x_i} \neq a'_i = b'_i \vee \overline{x_i} \neq a'_i = b'_i$
$>$	$a'_i > b'_i \wedge b'_i \leq \underline{x_i} \leq \overline{x_i} \leq a'_i$
$\hat{>}$	$a'_i > b'_i \wedge (\underline{x_i} < b'_i \vee a'_i < \overline{x_i})$

4.3 Finite Automaton for between

Fig. 4 shows a deterministic finite automaton BFA for signature strings, from which we shall derive the filtering algorithm. State 1 is the initial state. There are three terminal states, F, T1 and T2, each corresponding to a separate case. State F is the failure case, whereas states T1–T2 are success cases.



Initial values	**Final values**
$\vec{a} = \langle 4, 6, 0 \rangle$	$\vec{a'} = \langle 4, 7, 4 \rangle$
$\vec{b} = \langle 6, 4, 9 \rangle$	$\vec{b'} = \langle 6, 3, 6 \rangle$
$x_0 \in \{2, 4, 6, 8\}$	$x_0 \in \{4, 6\}$
$x_1 \in \{3, 5, 7\}$	$x_1 \in \{3, 7\}$
$x_2 \in \{4, 5, 6\}$	$x_2 \in \{4, 5, 6\}$

Fig. 4. Case analysis of $\mathrm{between}(\vec{a}, \vec{x}, \vec{b})$ as finite automaton BFA, and an example, where BFA stops in state T2.

4.4 Case Analysis of between

We now discuss three regular expressions covering all possible cases of signatures of C. Where relevant, we also derive pruning rules for maintaining hyperarc consistency. Each regular expression corresponds to one of the terminal states of BFA. Note that, without loss of generality, each regular expression has a common prefix $P = (\boxed{=} \mid \boxed{\hat{=}})^\star$. For C to hold, clearly for each position i in the corresponding prefix of $\vec{x}$, by (3.2) the filtering algorithm must enforce $a'_i = x_i = b'_i$. In the regular expressions, q and r denote the position of the transition out of state 1 and 2 respectively. We now discuss the cases one by one.

Case F.

$$\underbrace{(\boxed{=} \mid \boxed{\hat{=}})^\star}_{P} \underbrace{(\boxed{>} \mid \boxed{\hat{>}})}_{q} \mathcal{B}^\star \tag{F}$$

We have that $a'_0 = b'_0 \wedge \cdots \wedge a'_{q-1} = b'_{q-1} \wedge a'_q > b'_q$, and so by (4), C must be false.

Case T1.

$$\underbrace{(\boxed{=} \mid \boxed{\hat{=}})^\star}_{P} \underbrace{(\boxed{\hat{<}} \mid \boxed{\$})}_{q} \mathcal{B}^\star \tag{T1}$$

We have that $a'_0 = b'_0 \wedge \cdots \wedge a'_{q-1} = b'_{q-1} \wedge (q = n \vee a'_q < b'_q)$. If $q = n$, we are done by (3.1) and (3.2). If $q < n$, we also have that $(a'_q, b'_q) \cap D(x_q) \neq \emptyset$. Thus by (3.5), all we have to do after P for C to hold is to enforce $a'_q \leq x_q \leq b'_q$.

Case T2.

$$\underbrace{(\boxed{=} \mid \boxed{\hat{=}})^\star}_{P} \underbrace{\boxed{<}}_{q} (\boxed{>} \mid \boxed{=})^\star \underbrace{(\boxed{<} \mid \boxed{\hat{<}} \mid \boxed{\hat{=}} \mid \boxed{\hat{>}} \mid \boxed{\$})}_{r} \mathcal{B}^\star \tag{T2}$$

We have that:

$$\bigwedge \begin{cases} a'_0 = b'_0 \wedge \cdots \wedge a'_{q-1} = b'_{q-1} \\ a'_q < b'_q \\ (a'_q, b'_q) \cap D(x_q) = \emptyset \\ a'_{q+1} \geq b'_{q+1} \wedge \cdots \wedge a'_{r-1} \geq b'_{r-1} \\ \forall i \in (q, r) : b'_i \leq \underline{x_i} \leq \overline{x_i} \leq a'_i \end{cases}$$

Consider position q, where $a'_q < b'_q$ and $(a'_q, b'_q) \cap D(x_q) = \emptyset$ hold. Since by (4) $a'_q \leq x_q \leq b'_q$ should also hold, x_q must be either a'_q or b'_q, and we know from (2) that both $x_q = a'_q$ and $x_q = b'_q$ have support.

It can be shown by induction that there are exactly two possible values for the subvector $\vec{x}_{[0,r)}$: $\vec{a'}_{[0,r)}$ and $\vec{b'}_{[0,r)}$.

Thus for C to hold, after P we have to enforce $x_i \in \{a'_i, b'_i\}$ for $q \leq i < r$. From (3.3) and (3.4), we now have that C holds iff

$$\bigvee \begin{cases} \vec{x}_{[0,r)} = \vec{a'}_{[0,r)} \wedge \vec{a'}_{[r,n)} \leq_{\text{lex}} \vec{x}_{[r,n)} \\ \vec{x}_{[0,r)} = \vec{b'}_{[0,r)} \wedge \vec{x}_{[r,n)} \leq_{\text{lex}} \vec{b'}_{[r,n)} \end{cases}$$

i.e.

$$\bigvee \begin{cases} r = n \wedge \vec{x}_{[0,r)} = \vec{a'}_{[0,r)} & (5.1) \\ r = n \wedge \vec{x}_{[0,r)} = \vec{b'}_{[0,r)} & (5.2) \\ r < n \wedge \vec{x}_{[0,r)} = \vec{a'}_{[0,r)} \wedge x_r > a'_r & (5.3) \\ r < n \wedge \vec{x}_{[0,r)} = \vec{a'}_{[0,r)} \wedge x_r = a'_r \wedge \vec{a'}_{(r,n)} \leq_{\text{lex}} \vec{x}_{(r,n)} & (5.4) \\ r < n \wedge \vec{x}_{[0,r)} = \vec{b'}_{[0,r)} \wedge x_r < b'_r & (5.5) \\ r < n \wedge \vec{x}_{[0,r)} = \vec{b'}_{[0,r)} \wedge x_r = b'_r \wedge \vec{x}_{(r,n)} \leq_{\text{lex}} \vec{b'}_{(r,n)} & (5.6) \end{cases} \quad (5)$$

Finally, consider the possible cases for position r, which are:

- $r = n$, signature letter $\boxed{\$}$. We are done by (5.1) and (5.2).
- $a'_r < b'_r$, signature letters $\boxed{<}$ and $\boxed{\hat{<}}$. Then from (2) we know that we have solutions corresponding to both (5.3) and (5.5). Thus, all values for $\vec{x}_{[r,n)}$ have support, and we are done.
- $a'_r \geq b'_r$, signature letters $\boxed{\hat{>}}$ and $\boxed{\hat{=}}$. Then from (2) and from the signature letter, we know that we have solutions corresponding to both (5.4), (5.6), and one or both of (5.3) and (5.5). Thus, all values v for x_r such that $v \leq b'_r \vee v \geq a'_r$, and all values for $\vec{x}_{(r,n)}$, have support. Hence, we must enforce $x_r \notin (b'_r, a'_r)$.

4.5 Filtering Algorithm for between

By augmenting BFA with the pruning actions mentioned in Sect. 4.4, we arrive at a filtering algorithm `FiltBetween` ([5, Alg. 1]) for $\text{between}(\vec{a}, \vec{x}, \vec{b})$. When a constraint is posted, the algorithm will delay or fail, depending on where BFA stops. The filtering algorithm needs to recompute feasible upper and lower bounds each time it is resumed. We summarize the properties of `FiltBetween` in the following proposition.

Proposition 3.

1. `FiltBetween` *doesn't remove any solutions.*
2. `FiltBetween` *removes all domain values that cannot be part of any solution.*
3. `FiltBetween` *runs in* $O(nd)$ *time.*

4.6 Feasible Upper and Lower Bounds

We now show how to compute the tight, i.e. lexicographically largest and smallest, and feasible vectors $\vec{a'}$ and $\vec{b'}$ that were introduced in Sect. 4.1, given a constraint $\text{between}(\vec{a}, \vec{x}, \vec{b})$.

Upper Bounds The algorithm, $\texttt{ComputeUB}(\vec{x}, \vec{b}, \vec{b'})$, has two steps. The key idea is to find the smallest i, if it exists, such that b'_i must be less than b_i.

1. Compute α as the smallest $i \geq -1$ such that one of the following holds:
   (a) $i \geq 0 \wedge b_i \notin D(x_i) \wedge b_i > \underline{x_i}$
   (b) $\vec{b}_{(i,n)} <_{\text{lex}} \underline{\vec{x}}_{(i,n)}$

   In both cases, a value $b'_i < b_i$ must be chosen from $D(x_i)$. If no such i exists, let $\alpha = n$. If $\alpha = -1$, the algorithm fails, meaning that $\vec{x} \leq_{\text{lex}} \vec{b}$ can't hold. For example, $\alpha = 1$ in the example shown in Fig. 4. See [5, Alg. 2].
2. b'_i is computed as follows for $0 \leq i < n$:

$$b'_i = \begin{cases} b_i, \text{ if } i < \alpha \\ \text{prev\_value}(x_i, b_i), \text{ if } i = \alpha \\ \overline{x_i}, \text{ if } i > \alpha \end{cases}$$

We summarize the properties of `ComputeUB` in the following lemma.

Lemma 1. `ComputeUB` *is correct and runs in* $O(n + d)$ *time.*

Lower Bounds The feasible lower bound algorithm, `ComputeLB`, is totally analogous to `ComputeUB`, and not discussed further.

4.7 Filtering Algorithm

We now have the necessary building blocks for constructing a filtering algorithm for lex_chain; see [5, Alg. 3]. The idea is as follows. For each vector in the chain, we first compute a tight and feasible upper bound by starting from $\vec{x}_{m-1}$. We then compute a tight and feasible lower bound for each vector by starting from $\vec{x}_0$. Finally for each vector, we restrict the domains of its variables according to the bounds that were computed in the previous steps. Any value removal is a relevant propagation event. We summarize the properties of `FiltLexChain` in the following proposition.

Proposition 4.

1. `FiltLexChain` *maintains hyperarc consistency.*
2. *If there is no variable aliasing,* `FiltLexChain` *reaches a fixpoint after one run.*
3. *If there is no variable aliasing,* `FiltLexChain` *runs in* $O(nmd)$ *time.*

5 Related Work

Within the area of logic, automata have been used by associating with each formula defining a constraint an automaton recognizing the solutions of the constraint [10].

An $O(n)$ filtering algorithm maintaining hyperarc consistency of the $\leq_{\text{lex}}$ constraint was described in [9]. That algorithm is based on the idea of using two pointers α and β. The α pointer gives the position of the most significant pair of variables that are not

ground and equal, and corresponds to our q position. The β pointer, if defined, gives the most significant pair of variables from which $\leq_{\text{lex}}$ cannot hold. It has no counterpart in our algorithm. As the constraint store gets tighter, α and β get closer and closer, and the algorithm detects entailment when $\alpha + 1 = \beta \vee \overline{x_\alpha} < \underline{y_\alpha}$. The algorithm is only triggered on propagation events on variables in $[\alpha, \beta)$. It does not detect entailment as eagerly as ours, as demonstrated by the example in Fig. 2. `FiltLex` detects entailment on this example, whereas Frisch's algorithm does not. Frisch's algorithm is shown to run in $O(n)$ on posting a constraint as well as for handling a propagation event.

6 Discussion

The main result of this work is an approach to designing filtering algorithms by derivation from finite automata operating on constraint signatures. We illustrated this approach in two case studies, arriving at:

- A filtering algorithm for $\leq_{\text{lex}}$, which maintains hyperarc consistency, detects entailment or rewrites itself to a simpler constraint whenever possible, and runs in $O(n)$ time for posting the constraint plus amortized $O(1)$ time for handling each propagation event.
- A filtering algorithm for lex_chain, which maintains hyperarc consistency and runs in $O(nmd)$ time per invocation, where d is the cost of certain domain operations.

In both case studies, the development of the algorithms was mainly manual and required several inspired steps. In retrospect, the main benefit of the approach was to provide a rigorous case analysis for the logic of the algorithms being designed. Some work remains to turn the finite atomaton approach into a methodology for semi-automatic development of filtering algorithms. Relevant, unsolved research issues include:

1. **What class of constraints is amenable to the approach?** It is worth noting that $\leq_{\text{lex}}$ and between can both be defined inductively, so it is tempting to conclude that any inductively defined constraint is amenable. Constraints over sequences [11, 12] would be an interesting candidate for future work.
2. **Where does the alphabet come from?** In retrospect, this was the most difficult choice in the two case studies. In the $\leq_{\text{lex}}$ case, the basic relations used in the definition of the constraint are $\{<, =, >\}$, each symbols of $\mathcal{A}$ denoting a set of such relations. In the between case, the choice of alphabet was far from obvious and was influenced by an emerging understanding of the necessary pruning rules. As a general rule, the cost of computing each signature letter has a strong impact on the overall complexity, and should be kept as low as possible.
3. **Where does the finite automaton come from?** Coming up with a regular language and corresponding finite automaton for *ground* instances is straightforward, but there is a giant leap from there to the nonground case. In our case studies, it was mainly done as a rational reconstruction of an emerging understanding of the necessary case analysis.

4. **Where do the pruning rules come from?** This was the most straightforward part in our case studies. At each non-failure terminal state, we analyzed the corresponding regular language, and added pruning rules that prevented there from being failed ground instances, i.e. rules that removed domain values with no support.
5. **How do we make the algorithms incremental?** The key to incrementality for $\leq_{\text{lex}}$ was the observation that the finite automaton could be safely restarted at an internal state. This is likely to be a general rule for achieving some, if not all, incrementality. We could have done this for $\text{between}(\vec{a}, \vec{x}, \vec{b})$, except in the context of lex_chain, between is not guaranteed to be resumed with $\vec{a}$ and $\vec{b}$ unchanged, and the cost of checking this would probably outweigh the savings of an incremental algorithm.

Acknowledgements

We thank Justin Pearson and Zeynep Kızıltan for helpful discussions on this work, and the anonymous referees for their helpful comments.

References

1. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of the National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.
2. J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. of the National Conference on Artificial Intelligence (AAAI-94)*, pages 209–215, 1996.
3. P. Baptiste, C. LePape, and W. Nuijten. *Constraint-Based Scheduling*. Kluwer Academic Publishers, 2001.
4. Alan Tucker. *Applied Combinatorics*. John Wiley & Sons, 4th edition, 2002.
5. Mats Carlsson and Nicolas Beldiceanu. Arc-consistency for a Chain of Lexicographic Ordering Constraints. Technical Report T2002-18, Swedish Institute of Computer Science, 2002.
6. Mats Carlsson and Nicolas Beldiceanu. Revisiting the Lexicographic Ordering Constraint. Technical Report T2002-17, Swedish Institute of Computer Science, 2002.
7. Mats Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 3.10 edition, January 2003. `http://www.sics.se/sicstus/`.
8. N. Beldiceanu and A. Aggoun. Time stamps techniques for the trailed data in CLP systems. In *Actes du Séminaire 1990 - Programmation en Logique*, Tregastel, France, 1990. CNET.
9. A. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Global Constraints for Lexicographic Orderings. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming – CP'2002*, volume 2470 of *LNCS*, pages 93–108. Springer-Verlag, 2002.
10. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata/`.
11. COSYTEC S.A. *CHIP Reference Manual*, version 5 edition, 1996. The *sequence* constraint.
12. J.-C. Régin and J. F. Puget. A filtering algorithm for global sequencing constraints. In G. Smolka, editor, *Principles and Practice of Constraint Programming – CP'97*, volume 1330 of *LNCS*, pages 32–46. Springer-Verlag, 1997.