Arc-Consistency for a Chain of Lexicographic Ordering Constraints

Mats Carlsson and Nicolas Beldiceanu
SICS, Lägerhyddsv. 18, SE-752 37 UPPSALA, Sweden
{matsc,nicolas}@sics.se

October 17, 2002
SICS Technical Report T2002:18
ISRN: SICS-T–2002/18-SE
ISSN: 1100-3154

Abstract. We present an arc-consistency algorithm for a chain of lexicographic ordering constraints on $m$ vectors of $n$ variables each. The algorithm maintains arc-consistency and runs in $O(mnd)$ time per invocation, where $d$ is the cost of certain domain operations.

Keywords: Constraint Programming, Global Constraints, Lexicographic Ordering, Symmetry.

1 Introduction

Given two vectors, $\vec{x}$ and $\vec{y}$ of $n$ variables, $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$, 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 $(x_1, \ldots, x_{n-1}) \leq_{\text{lex}} (y_1, \ldots, y_{n-1})$. This constraint is available e.g. in ECLiPSe 5.4 [1], where it is named lexic_order/2. An $O(n)$ filtering algorithm maintaining arc-consistency of the constraint was described in [3]. Similarly, the constraint $\vec{x} \preceq_{\text{lex}} \vec{y}$ holds iff $x_0 < y_0$ or $x_0 = y_0$ and $(x_1, \ldots, x_{n-1}) <_{\text{lex}} (y_1, \ldots, y_{n-1})$.

In this report, 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 [3], 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 arc-consistency algorithm for $\text{lex\_chain}$ “may be quite challenging”. This report addresses this challenge. Our contribution is a filtering algorithm for $\text{lex\_chain}$, which maintains arc-consistency and runs in $O(mnd)$ time per invocation, where $d$ is the cost of certain domain operations.
At the heart of the algorithm is a procedure for pruning a vector of variables wrt. fixed, feasible lower and upper bounds. This procedure was derived from a finite automaton operating on a signature of the relation $\bar{a} \leq_{\text{lex}} \bar{x} \leq_{\text{lex}} \bar{b}$, a methodology which to our knowledge has not been used before in filtering algorithm construction. The point is that we have to consider globally both the lower and upper bound, lest we may miss some pruning, as illustrated by the following example. Consider the following system of domain and lexicographic ordering constraints:

- $c_1: x_0 \in [1, 3]$
- $c_2: x_1 \in [0, 5]$
- $c_3: x_2 \in [0, 5]$
- $c_4: \langle 1, 5, 3 \rangle \leq_{\text{lex}} \langle x_0, x_1, x_2 \rangle$
- $c_5: x_0 \neq 2$
- $c_6: \langle x_0, x_1, x_2 \rangle \leq_{\text{lex}} \langle 3, 0, 1 \rangle$

Fig. 1 shows the forbidden triples of values according to constraints $c_4$, $c_5$, and $c_6$. From the figure, we see that $x_0$ should be restricted to interval $[1, 3]$, and that $x_1$ and $x_2$ should be respectively restricted to $\{0, 5\}$ and $\{0, 1, 3, 4, 5\}$. Removing value 2 from $x_2$ is impossible if we consider the three constraints $c_4$, $c_5$ and...
Independently. One can also observe that, unlike the standard lexicographic ordering constraint between two vectors of variables, the pruning according to a fixed lower and upper bound can create holes in the domains. Furthermore it can also take advantage of holes in the domains since the previous pruning would not occur any more if \( x_0 \) could take value 2.

The rest of the report is organized as follows: We first define some necessary notions and notation. We then give a filtering algorithm for the constraint \( \bar{a} \leq_{\text{lex}} \bar{b} \) where \( \bar{a} \) and \( \bar{b} \) are feasible vectors of integers, i.e. \( \bar{x} = \bar{a} \) and \( \bar{x} = \bar{b} \) are both solutions to the constraint. We then show how to compute lexicographically largest and smallest feasible vectors of values, given a vector of variables and upper and lower bound vectors. We then have the necessary building blocks for a filtering algorithm for lex\_chain. We conclude with some comments on possible extensions and improvements.

2 Preliminaries

We need 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 \( \bar{x} \) with start index \( i \) and last index \( j \) is denoted by \( \bar{x}[i,j] \).

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 \( \max(D(x)) \); \( \text{next\_value}(x, a) \) denotes \( \min\{i \in D(x) \mid i > a\} \), if it exists, and \(+\infty\) otherwise; and \( \text{prev\_value}(x, a) \) 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 \( T \).

The domain store is pruned by applying the following operations to a variable \( x \): \( \text{fix\_interval}(x, a, b) \) removes from \( D(x) \) any value that is not in \([a,b]\) and runs in constant time; and \( \text{prune\_interval}(x, a, b) \) removes from \( D(x) \) any value that is in \([a,b]\) and has cost \( d \). Each operation 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 arc-consistent iff, for each such variable \( x \) and value \( v \in D(x) \), \( x = v \) has support. A filtering algorithm maintains arc-consistency of \( C \) iff it removes any value \( v \in D(x) \) such that \( x = v \) does not have support.

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 \( \langle a \rangle \); \( \ldots \) denotes any string over
denotes concatenation; $E \cup E'$ denotes $L(E) \cup L(E')$ (union); and $E^*$ denotes $L(E)^*$ (closure). Parentheses are used for grouping.

Given two vectors, $\vec{a}$ and $\vec{b}$ of $n$ integers, and a vector $\vec{x}$ of $n$ variables, let $\text{between}(\vec{a}, \vec{x}, \vec{b})$ denote the constraint $\vec{a} \leq_{\text{lex}} \vec{x} \leq_{\text{lex}} \vec{b}$. It has the following precondition, which always holds in the algorithms presented herein:

$$\forall i \in [0, n) : a_i \in D(x_i) \land b_i \in D(x_i)$$

Let $\mathcal{A}$ be the alphabet $\{\lessgtr, \lessless, \lesslessless, \lesseqqgtr, \lesseqqless, \equiv, \gtrless, \gtrlessless, \gtrlesslessless\}$. The signature of a constraint $C \equiv \text{between}(\vec{a}, \vec{x}, \vec{b})$ wrt. the current constraint store $\Gamma$ is a string $S$ over $\mathcal{A}$ of length $n + 1$ where $S_n = \equiv$ to mark the end of the string, and for $0 \leq i < n$:

$$S_i = \left\{ \begin{array}{l}
\lessgtr, \text{ if } a_i < b_i \land \Gamma \models (x_i \leq a_i \lor x_i \geq b_i) \\
\lessless, \text{ if } a_i < b_i \land \Gamma \not\models (x_i \leq a_i \lor x_i \geq b_i) \\
\lesslessless, \text{ if } a_i = b_i \land \Gamma \models a_i = x_i = b_i \\
\equiv, \text{ if } a_i = b_i \land \Gamma \not\models a_i = x_i = b_i \\
\lesseqqgtr, \text{ if } a_i > b_i \land \Gamma \models b_i \leq x_i \leq a_i \\
\lesseqqless, \text{ if } a_i > b_i \land \Gamma \not\models b_i \leq x_i \leq a_i 
\end{array} \right.$$ 

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 $\Gamma \models (x_i \leq a_i \lor x_i \geq b_i)$ requires the use of $\text{next\_value}$ or $\text{prev\_value}$, and has cost $d$; see Table 1. Each letter $S_i$ is called the signature letter at pos. $i$ of $C$ wrt. $\Gamma$.

<table>
<thead>
<tr>
<th>$S_i$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lessgtr$</td>
<td>$a_i &lt; b_i \land \text{next_value}(x_i, a_i) \geq b_i$</td>
</tr>
<tr>
<td>$\lessless$</td>
<td>$a_i &lt; b_i \land \text{next_value}(x_i, a_i) &lt; b_i$</td>
</tr>
<tr>
<td>$\lesslessless$</td>
<td>$x_i = \overline{x_i} = a_i = b_i$</td>
</tr>
<tr>
<td>$\equiv$</td>
<td>$x_i \neq a_i = b_i \lor \overline{x_i} \neq a_i = b_i$</td>
</tr>
<tr>
<td>$\lesseqqgtr$</td>
<td>$a_i &gt; b_i \land b_i \leq x_i \leq a_i$</td>
</tr>
<tr>
<td>$\lesseqqless$</td>
<td>$a_i &gt; b_i \land (x_i &lt; b_i \lor a_i &lt; \overline{x_i})$</td>
</tr>
</tbody>
</table>

Table 1. Computing the signature letter at pos. $i$
3 Filtering for \textit{between}(\vec{a}, \vec{x}, \vec{b})

3.1 Declarative Semantics

Consider a constraint $C \equiv \text{between}(\vec{a}, \vec{x}, \vec{b})$ where $\vec{a}$ and $\vec{b}$ are feasible vectors of integers of size $n$ and $\vec{x}$ is a vector of $n$ variables. It is straightforward to see that the declarative semantics is:

$$C \equiv \begin{cases} n = 0 \\ a_0 = x_0 = b_0 \land \vec{a}_{[1,n]} \leq_{\text{lex}} \vec{x}_{[1,n]} \leq_{\text{lex}} \vec{b}_{[1,n]} \\ a_0 = x_0 < b_0 \land \vec{a}_{[1,n]} \leq_{\text{lex}} \vec{x}_{[1,n]} \leq_{\text{lex}} \vec{b}_{[1,n]} \\ a_0 < x_0 = b_0 \land \vec{a}_{[1,n]} \leq_{\text{lex}} \vec{x}_{[1,n]} \leq_{\text{lex}} \vec{b}_{[1,n]} \\ a_0 < x_0 < b_0 \end{cases}$$

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

$$C \land (a_0 = b_0) \land \cdots \land (a_{i-1} = b_{i-1}) \Rightarrow a_i \leq x_i \leq b_i$$

3.2 A Finite Automaton

Fig. 2 shows a deterministic finite automaton BFA for signature strings, from which we will derive the filtering algorithm. State 1 is the initial state. There are three terminal states, F1, T1 and T2, each corresponding to a separate case. State F1 is the failure case, whereas states T1–T2 are success cases.

![Figure 2](image_url)

Fig. 2. Case analysis of \textit{between}(\vec{a}, \vec{x}, \vec{b}) as finite automaton BFA
3.3 Case Analysis

We now discuss three regular expressions covering all possible cases of signatures of $C$. Where relevant, we also derive pruning rules for maintaining arc-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 $P = \begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix}$. For $C$ to hold, clearly for each pos. $i$ in the corresponding prefix of $\bar{x}$, by (2.2) 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 F1.**

$$\begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix} \ast \begin{pmatrix} > \\ > \end{pmatrix}_{q} \ldots$$

We have that $a_0 = b_0 \land \cdots \land a_{q-1} = b_{q-1} \land a_q > b_q$, and so by (3), $C$ must be false.

**Case T1.**

$$\begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix} \ast \begin{pmatrix} > \\ > \end{pmatrix}_{q} \begin{pmatrix} < \\ $ \end{pmatrix}_{r} \ldots$$

We have that $a_0 = b_0 \land \cdots \land a_{q-1} = b_{q-1} \land (q = n \lor a_q < b_q)$. If $q = n$, we are done by (2.1) and (2.2). If $q < n$, we also have that $(a_q, b_q) \cap D(x_q) \neq \emptyset$. Thus by (2.5), all we have to do after $P$ for $C$ to hold is to enforce $a_q \leq x_q \leq b_q$.

**Case T2.**

$$\begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix} \ast \begin{pmatrix} > \\ > \end{pmatrix}_{q} \begin{pmatrix} < \\ < \end{pmatrix}_{r} \begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix} \begin{pmatrix} > \\ > \end{pmatrix}_{r} \begin{pmatrix} < \\ $ \end{pmatrix}_{r} \ldots$$

We have that:

\[
\begin{align*}
& a_0 = b_0 \land \cdots \land a_{q-1} = b_{q-1} \\
& a_q < b_q \\
& (a_q, b_q) \cap D(x_q) = \emptyset \\
& a_{q+1} \geq b_{q+1} \land \cdots \land a_{r-1} \geq b_{r-1} \\
& \forall i \in (q, r) : b_i \leq x_i \leq r_i \leq a_i
\end{align*}
\]

Consider pos. $q$, where $a_q < b_q$ and $(a_q, b_q) \cap D(x_q) = \emptyset$ hold. Since by (3) $a_q \leq x_q \leq b_q$ should also hold, $x_q$ must be either $a_q$ or $b_q$, and we know from (1) that both $x_q = a_q$ and $x_q = b_q$ have support.

We will now show by induction that there are exactly two possible values for the subvector $\bar{x}_{[0,r)}$, $\bar{a}_{[0,r)}$ and $\bar{b}_{[0,r)}$. 
**Base step.** We have already established that \( x_q = a_q \lor x_q = b_q \) holds, and for the prefix \( P \) we have that \( \vec{a}_{[0, q)} = \vec{x}_{[0, q)} = \vec{b}_{[0, q)} \).

**Induction step.** Now consider pos. \( i \) where \( q < i < r \). Recall that \( b_i \leq x_i \leq a_i \) must hold. If \( S_i = \boxed{\text{[}} \) then \( a_i = x_i = b_i \), and we are done. If \( S_i = \boxed{\rangle} \) and \( x_i = a_i - 1 \), then \( x_i < a_i \) would contradict (2.3). If \( S_i = \boxed{\rangle} \) and \( x_i = b_i - 1 \), then \( b_i > x_i \) would contradict (2.4). Thus, \( x_i = a_i \) iff \( x_{i-1} = a_{i-1} \), and \( x_i = b_i \) iff \( x_{i-1} = b_{i-1} \).

Thus for \( C \) to hold, after \( P \) we have to enforce \( x_i \in \{a_i, b_i\} \) for \( q \leq i < r \). From (2.3) and (2.4), we now have that \( C \) holds iff

\[
\begin{align*}
\forall \left\{ \begin{array}{l}
\vec{x}_{[0, r)} &= \vec{a}_{[0, r)} \land \vec{a}_{[r, n)} \leq_{\text{lex}} \vec{x}_{[r, n)} \\
\vec{x}_{[0, r)} &= \vec{b}_{[0, r)} \land \vec{x}_{[r, n)} \leq_{\text{lex}} \vec{b}_{[r, n)}
\end{array} \right. \\
\end{align*}
\]

i.e.

\[
\begin{align*}
\forall \left\{ \begin{array}{l}
r = n \land \vec{x}_{[0, r)} &= \vec{a}_{[0, r)} \\
r = n \land \vec{x}_{[0, r)} &= \vec{b}_{[0, r)} \\
r > n \land \vec{x}_{[0, r)} &= \vec{a}_{[0, r)} \land x_r > a_r \\
r > n \land \vec{x}_{[0, r)} &= \vec{a}_{[0, r)} \land x_r < a_r \\
r > n \land \vec{x}_{[0, r)} &= \vec{b}_{[0, r)} \land x_r > b_r \\
r > n \land \vec{x}_{[0, r)} &= \vec{b}_{[0, r)} \land x_r < b_r \\
\end{array} \right. (4) 
\end{align*}
\]

Finally, consider the possible cases for pos. \( r \), which are:

- \( r = n \), signature letter \( \boxed{\text{[}} \). We are done by (4.1) and (4.2).
- \( a_r < b_r \), signature letters \( \boxed{\text{[}} \) and \( \boxed{\text{<}} \). Then from (1) we know that we have solutions corresponding to both (4.3) and (4.5). Thus, all values for \( \vec{x}_{[r, n)} \) have support, and we are done.
- \( a_r \geq b_r \), signature letters \( \boxed{\text{[}} \) and \( \boxed{\text{<}} \). Then from (1) and from the signature letter, we know that we have solutions corresponding to both (4.4), (4.6), and one or both of (4.3) and (4.5). Thus, all values \( v \) for \( x_r \) such that \( v \leq b_r \lor v \geq a_r \), and all values for \( \vec{x}_{[r, n)} \), have support. Hence, we must enforce \( x_r \notin (b_r, a_r) \).

### 3.4 A Filtering Algorithm

By augmenting BFA with the pruning actions mentioned in Sect. 3.3, we arrive at a filtering algorithm \( \text{Bounds\_Lex} \) (Alg. 1) for between \( (\vec{a}, \vec{x}, \vec{b}) \). When a constraint is posted, the algorithm will delay or fail, depending on where BFA stops. We summarize the properties of \( \text{Bounds\_Lex} \) in the following proposition.
Proposition 1.

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

Proof:

1. BFA has one failure case, F1. In Sect. 3.3, we showed that the corresponding instances have no solutions. Furthermore, no pruning action removes any value that has support.
2. BFA has two success cases, T1–T2. We have showed that all corresponding ground instances are solutions, provided that:
   - We enforce $a_i \leq x_i \leq b_i$ for $0 \leq i \leq q$.
   - In state T2, we enforce (i) $x_i \in \{a_i, b_i\}$ for $q < i < r$, and (ii) $x_r \notin \{b_r, a_r\}$.
3. At most $n + 1$ signature letters are examined, and each decision and pruning action costs at most $d$.

4 Computing Feasible Upper and Lower Bounds

In this section, we show how to compute tight, i.e. lexicographically largest and smallest, and feasible vectors of values, given a vector of variables, an upper bound vector, and a lower bound vector, where the bound vectors are not necessarily feasible. The computed vectors are called feasible upper and lower bounds. The feasibility is equivalent to precondition (1) of Bounds_Lex, where these vectors will be used.

4.1 Upper Bounds

We address the following problem: given a vector $\vec{x}$ of variables and a vector $\vec{b}$ of integers, we want to compute the lexicographically largest vector $\vec{u}$ such that:

$$\vec{u} \leq_{lex} \vec{b} \land \forall i \in [0,n) : u_i \in D(x_i)$$

The algorithm, ComputeUB($\vec{x}, \vec{b}, \vec{a}$), has two steps. The key point is to compute $\alpha$ as the smallest $i$ such that $\vec{u}_{[0,i]} = \vec{b}_{[0,i]}$ is not a prefix of the computed upper bound.

1. Compute $\alpha$ as the smallest $i \geq -1$ such that one of the following holds:
PROCEDURE  Bounds_Lex($\bar{a}, \bar{x}, \bar{b}$) : (fail, delay)

Require: $\forall i \in [0, n) : a_i \in D(x_i) \land b_i \in D(x_i)$

Ensure: If fail, then between($\bar{a}, \bar{x}, \bar{b}$) has no solution.

Ensure: If delay, then for each $i \in [0, n)$ and value $v_i \in D(x_i)$ there is a solution $\bar{x} = \bar{v}$ such that $\forall j \in [0, n) : v_j \in D(x_j)$.

1: $i \leftarrow 0$ // enter state 1
2: while $i < n \land a_i = b_i$ do
3:  if fix_interval($x_i, a_i, b_i$) = fail then // prune inside $P$
4:     return fail
5:  end if
6:  $i \leftarrow i + 1$
7: end while
8: if $i < n \land \text{fix interval}(x_i, a_i, b_i) = \text{fail}$ then // prune pos. $q$
9:  return fail
10: end if
11: if $i = n \lor \text{next value}(x_i, a_i) < b_i$ then
12:  return delay
13: end if
14: $i \leftarrow i + 1$ // enter state 2
15: while $i < n \land x_i = b_i \land A_{\bar{x}} = a_i$ do
16:  if prune_interval($x_i, b_i + 1, a_i - 1$) = fail then // prune inside $(q, r)$
17:     return fail
18:  end if
19:  $i \leftarrow i + 1$
20: end while
21: if $i < n \land \text{prune interval}(x_i, b_i + 1, a_i - 1) = \text{fail}$ then // prune pos. $r$
22:  return fail
23: else
24:  return delay
25: end if

Algorithm 1: Filtering algorithm for between($\bar{a}, \bar{x}, \bar{b}$)
The intuition behind the two cases is:

(a) \( x_i < b_i \not\in D(x_i) \land b_i > x_i \)

(b) \( \tilde{b}_{(i,n)} \not\leq_{\text{lex}} \tilde{x}_{(i,n)} \)

We have that \( b_i > x_i \land b_{i+1} = x_{i+1} \land \cdots \land b_{k-1} = x_{k-1} \land b_k < x_k \).

In both cases, a smaller value for \( u_i \) must be chosen from \( D(x_i) \). If no such \( i \) exists, let \( \alpha = n \). If \( \alpha = -1 \), the algorithm fails, meaning that \( \tilde{x} \not\leq_{\text{lex}} \tilde{b} \) can’t hold.

2. \( u_i \) is computed as follows for \( 0 \leq i < n \):

\[
u_i = \begin{cases} 
    b_i, & \text{if } i < \alpha \\
    \text{prev\_value}(x_i, b_i), & \text{if } i = \alpha \\
    \overline{x_i}, & \text{if } i > \alpha 
\end{cases}
\]

We summarize the properties of ComputeUB in the following lemma.

**Lemma 1.** ComputeUB implements the specification (5) and runs in \( O(n + d) \) time.

**Proof.** It should be clear from the above that \( \tilde{u} \not\leq_{\text{lex}} \tilde{b} \) and that \( \forall i \in [0, n) : u_i \in D(x_i) \).

It remains to be shown that there does not exist any \( \tilde{\sigma} \) such that \( \tilde{u} <_{\text{lex}} \tilde{\sigma} \leq_{\text{lex}} \tilde{b} \) and \( \forall i \in [0, n) : v_i \in D(x_i) \).

If \( \alpha = n, \tilde{u} = \tilde{b} \), so we are done. Otherwise, construct the lexicographically smallest \( \tilde{\sigma} \) such that \( \tilde{u} <_{\text{lex}} \tilde{\sigma} \) as follows. Let \( \beta \) be the least significant position \( i \) such that \( u_i < \overline{x_i} \). If no such \( \beta \) exists, we are done. Otherwise, compute \( v_i \) as follows for \( 0 \leq i < n \):

\[
v_i = \begin{cases} 
    u_i, & \text{if } i < \beta \\
    \text{next\_value}(x_i, u_i), & \text{if } i = \beta \\
    \overline{x_i}, & \text{if } i > \beta 
\end{cases}
\]

By construction, \( \beta \leq \alpha \). Consider the two cases:

**Case \( \beta < \alpha \):** We have that \( v_{\{0,\beta\}} = b_{\{0,\beta\}} \) and \( b_\beta < v_\beta = \text{next\_value}(x_\beta, b_\beta) \).

Hence, \( \tilde{b} <_{\text{lex}} \tilde{\sigma} \), a contradiction, so we are done.

**Case \( \beta = \alpha \):** We have that \( v_{\{0,\beta\}} = b_{\{0,\beta\}} \) and \( v_\beta = \text{next\_value}(x_\beta, \text{prev\_value}(x_\beta, b_\beta)) \).

Assume that \( b_\beta \not\in D(x_\beta) \). Then \( b_\beta < v_\beta \). Assume instead that \( b_\beta \in D(x_\beta) \).

Then \( b_\beta = v_\beta \), but then we also have that \( \tilde{b}_{(\beta,n)} <_{\text{lex}} \tilde{\sigma}_{(\beta,n)} = \tilde{x}_{(\beta,n)} \). In both cases, \( \tilde{b} <_{\text{lex}} \tilde{\sigma} \), a contradiction, so we are done.

Given \( \alpha, \tilde{u} \) is computed in \( O(n + d) \) time. Alg. 2 computes \( \alpha \) in \( O(n) \) time. \( \square \)
4.2 Lower Bounds

We address the following problem: given a vector $\vec{x}$ of variables and a vector $\vec{a}$ of integers, we want to compute the lexicographically smallest vector $\vec{t}$ such that:

$$\vec{a} \leq_{\text{lex}} \vec{t} \land \forall i \in [0, n) : t_i \in D(x_i)$$  \hspace{1cm} (6)

The algorithm, ComputeLB($\vec{x}$, $\vec{a}$, $\vec{b}$), has two steps. The key point is to compute $\alpha$ as the smallest $i$ such that $\vec{t}_{[0,i]} = \vec{a}_{[0,i]}$ is not a prefix of the computed lower bound.

1. Compute $\alpha$ as the smallest $i \geq -1$ such that one of the following holds:
   (a) $i \geq 0 \land a_i \notin D(x_i) \land a_i < x_i$
   (b) $\vec{a}_{(t,n)} \leq_{\text{lex}} \vec{a}_{(t,n)}$

   The intuition behind the two cases is:
   (a) $\vec{x}[i] > a_i \in \vec{a}_i$.
   (b) We have that $a_i < x_i \land a_{i+1} = x_{i+1} \land \ldots \land a_{k-1} = x_{k-1} \land a_k > x_k$.

   In both cases, a larger value for $l_i$ must be chosen from $D(x_i)$. If no such $i$ exists, let $\alpha = n$. If $\alpha = -1$, the algorithm fails, meaning that $\vec{a} \leq_{\text{lex}} \vec{x}$ can’t hold.

2. $l_i$ is computed as follows for $0 \leq i < n$:

$$l_i = \begin{cases} a_i, & \text{if } i < \alpha \\ \text{next\_value}(x_i, a_i), & \text{if } i = \alpha \\ x_i, & \text{if } i > \alpha \end{cases}$$

We summarize the properties of ComputeLB in the following lemma.

**Lemma 2.** ComputeLB implements the specification (6) and runs in $O(n + d)$ time.

**Proof.** Analogous to the proof of Lemma 1.  \hfill $\Box$
4.3 Strict Versions of Compute UB and ComputeLB

Suppose that we want to replace the $\leq_{\text{lex}}$ constraint in (5.6) by $<_{\text{lex}}$. To achieve that, all we have to do is to modify the definition of $\alpha$, replacing $\leq_{\text{lex}}$ by $<_{\text{lex}}$. In Alg. 2, the condition on line 8 is changed to $i < n \land b_i > x_i$.

5 A Filtering Algorithm for $\text{lex}_\text{chain}$

We now have the necessary building blocks for constructing a filtering algorithm for $\text{lex}_\text{chain}$; see Alg. 3. The idea is as follows. For each vector $\hat{X}_t$ in the chain, we first compute a tight and feasible upper bound by starting from $\hat{X}_{m-1}$. We then compute a tight and feasible lower bound for each vector by starting from $\hat{X}_0$. Finally, for each vector, we restrict the domains of its variables according to the bounds that were computed in the previous steps.

```plaintext
PROCEDURE Lex_Chain($\hat{X}_0, \ldots, \hat{X}_{m-1}$) : (fail, delay)
1: \hspace{1cm} \text{// $\hat{X}_t$ is variable vector $t$}
2: \hspace{1cm} \text{// $\hat{UB}_t$ is the upper bound vector $t$}
3: \hspace{1cm} \text{// $\hat{LB}_t$ is the lower bound vector $t$}
4: $\hat{UB}_{m-1} \leftarrow \hat{X}_{m-1}$
5: \hspace{1cm} \text{for } t \leftarrow m - 2 \text{ downto } 0 \text{ do}
6: \hspace{3cm} \text{if ComputeUB($\hat{X}_t$, $\hat{UB}_{t+1}$, $\hat{UB}_t$) = fail then}
7: \hspace{5cm} \text{return fail}
8: \hspace{3cm} \text{end if}
9: \hspace{1cm} \text{end for}
10: $\hat{LB}_0 \leftarrow \hat{X}_0$
11: \hspace{1cm} \text{for } t \leftarrow 1 \text{ to } m - 1 \text{ do}
12: \hspace{3cm} \text{if ComputeLB($\hat{X}_t$, $\hat{LB}_{t-1}$, $\hat{LB}_t$) = fail then}
13: \hspace{5cm} \text{return fail}
14: \hspace{3cm} \text{end if}
15: \hspace{1cm} \text{end for}
16: \hspace{1cm} \text{for } t \leftarrow 0 \text{ to } m - 1 \text{ do}
17: \hspace{3cm} \text{if Bounds_Lex($\hat{LB}_t$, $\hat{X}_t$, $\hat{UB}_t$) = fail then}
18: \hspace{5cm} \text{return fail}
19: \hspace{3cm} \text{end if}
20: \hspace{1cm} \text{end for}
21: \hspace{1cm} \text{return delay}
```

**Algorithm 3**: Filtering algorithm for a chain of $\leq_{\text{lex}}$ constraints

We summarize the properties of Lex_Chain in the following proposition.

**Proposition 2.**

1. Lex_Chain maintains arc-consistency.
2. If there is no variable aliasing, \texttt{Lex\_Chain} reaches a fixpoint after one run.
3. If there is no variable aliasing, \texttt{Lex\_Chain} runs in $O(nmd)$ time.

Proof.

1. We have already shown that \texttt{ComputeUB} and \texttt{ComputeLB} are correct and that \texttt{Bounds\_Lex} removes exactly those values that do not have support. Furthermore, the vectors computed by \texttt{ComputeUB} and \texttt{ComputeLB} are vectors of values that $\vec{X}_t$ can take, which is required by \texttt{Bounds\_Lex}.

   It remains to be shown that \texttt{Lex\_Chain} computes tight lower and upper bounds for each vector $t$. We will reason over the feasible vectors of values $\vec{v} = (v_0, \ldots, v_{n-1})$ i.e. $\forall i \in [0, n) : v_i \in D(X_{t,i})$.

   Consider the lower bounds for $t > 0$. For $t > 0$, \texttt{Lex\_Chain} computes $L\vec{B}_t$ as the smallest such feasible $\vec{v}$ such that $L\vec{B}_{t-1} \leq_{\text{lex}} \vec{v}$. Any $\vec{u} <_{\text{lex}} L\vec{B}_t$ would admit a non-solution (namely $\vec{u}$), and any $L\vec{B}_t <_{\text{lex}} \vec{u}$ would remove a solution (namely $L\vec{B}_t$), so $L\vec{B}_t$ is correct. An analogous argument holds for the upper bounds for $t < m - 1$.

2. From Proposition 1 (item 1), we have that $UB_{t,k}$ and $LB_{t,k}$ remain in $D(X_{t,k})$ for all variables on exit, and by assumption there is no communication caused by aliasing among the variables. Assume that we run \texttt{Lex\_Chain} a second time. Then by inspection, we see that \texttt{ComputeUB}(\vec{X}_t, U\vec{B}_{t+1}, U\vec{B}'_t) will compute $U\vec{B}'_t = U\vec{B}_t$ for $0 \leq t < m-1$, and similarly for \texttt{ComputeLB}. Hence, \texttt{Lex\_Chain} did reach a fixpoint after the first run.

3. By the previous item, \texttt{Lex\_Chain} calls \texttt{ComputeUB} and \texttt{ComputeLB} $m-1$ times each and \texttt{Bounds\_Lex} $m$ times. From this fact, Proposition 1, and Lemmas 1 and 2, we conclude that \texttt{Lex\_Chain} runs in $O(nmd)$ time.

$\square$

6 Extensions

Strict version. A filtering algorithm for a chain of $\leq_{\text{lex}}$ constraints can be derived easily by making the changes to \texttt{ComputeUB} and \texttt{ComputeLB} that were mentioned in Section 4.3. The other algorithms remain unchanged.

Entailment. $\text{lex\_chain}((\vec{X}_0, \ldots, \vec{X}_{m-1}) \equiv \vec{X}_0 \leq_{\text{lex}} \cdots \leq_{\text{lex}} \vec{X}_{m-1}$ is entailed iff $\vec{X}_i \leq_{\text{lex}} \vec{X}_{i+1}$ is entailed for all $0 \leq i < m - 1$. Our filtering algorithm does not detect entailment. In [2], we report an $O(n)$ filtering algorithm for $\leq_{\text{lex}}$, which does detect entailment. So one could use the entailment detection part of the algorithm in [2] for detecting entailment of $\text{lex\_chain}$ in $O(nm)$ time.
Decomposition. If $\tilde{X}_{i-1} \preceq_{\text{lex}} \tilde{X}_i \preceq_{\text{lex}} \tilde{X}_{i+1}$ is entailed for some $i$, we can ignore $\tilde{X}_i$ on future resumptions of the filtering algorithm.

If $\tilde{X}_{i-1} \preceq_{\text{lex}} \tilde{X}_i$ and $\tilde{X}_{i+1} \preceq_{\text{lex}} \tilde{X}_{i+2}$ are entailed for some $i$, we can use the algorithm in [2] instead of Lex_Chain for solving $\tilde{X}_1 \preceq_{\text{lex}} \tilde{X}_{i+1}$.

Handling a DAG of lexicographic ordering constraints. It should be straightforward to extend lex_chain to handling a directed acyclic graph of lexicographic ordering constraints over vectors of size $n$. The only change consists in computing a tight lower bound for each vector by considering the vectors in topological sort order, and by maintaining the greatest lower bound obtained from each predecessor of a vector. A similar change needs to be done for the upper bounds.

Acknowledgements

We are thankful to Justin Pearson for helpful discussions on this work.

References