Revisiting the Lexicographic Ordering Constraint

Mats Carlsson and Nicolas Beldiceanu

SICS, Lägerhyddsv. 18, SE-752 37 UPPSALA, Sweden
{matsc,nicolas}@sics.se

October 16, 2002
ISRN: SICS-T–2002/17-SE
ISSN: 1100-3154

Abstract. We present a global consistency algorithm for the lexicographic ordering constraint on two vectors of \( n \) variables. The algorithm maintains arc-consistency, runs in \( O(n) \) time on posting plus amortized \( O(1) \) time per propagation event, and detects entailment or rewrites itself to a simpler constraint whenever possible. The algorithm was derived from a finite automaton operating on a string which captures the relationship between each variable pair of the two vectors.

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

1 Introduction

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 if \( n = 0 \) or \( x_0 < y_0 \) or \( x_0 = y_0 \) and \( \langle x_1, \ldots, x_{n-1} \rangle \leq_{\text{lex}} \langle y_1, \ldots, y_{n-1} \rangle \). This constraint is available e.g. in ECLiPSe 5.4 [1], where it is named \texttt{lexico_le/2}. An \( O(n) \) filtering algorithm maintaining arc-consistency of the constraint was described in [2]. Similarly, the constraint \( \vec{x} \leq_{\text{lex}} \vec{y} \) holds if \( x_0 < y_0 \) or \( x_0 = y_0 \) and \( \langle x_1, \ldots, x_{n-1} \rangle <_{\text{lex}} \langle y_1, \ldots, y_{n-1} \rangle \).

In this report, we revisit this constraint and propose an alternative filtering algorithm based on automata theory. Our contribution is as follows: (i) in addition to maintaining arc-consistency, our algorithm detects entailment or rewrites itself to a simpler constraint whenever possible; (ii) it runs in \( O(n) \) time for posting the constraint plus amortized \( O(1) \) time for handling each propagation event; (iii) it was derived from a finite automaton operating on a signature of the constraint, a methodology which to our knowledge has not been used before in filtering algorithm construction.
The rest of the report is organized as follows: We first define some necessary notions and notation. After treating ground instances of $\leq_{\text{lex}}$, we generalize this idea to nonground instances and show how to derive a non-incremental filtering algorithm and its properties. We then modify it into an incremental algorithm and show its complexity. After comparing with related work, we conclude with some comments on possible extensions and improvements.

2 Preliminaries

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. $\underline{x}$ and $\overline{x}$ denote respectively $\min(D(x))$ and $\max(D(x))$ in the context of a current constraint store. If for $\Gamma = (X, D)$ and $\Gamma' = (X, D')$, $\forall x \in X : D'(x) \subseteq D(x)$, we say that $\Gamma' \subseteq \Gamma$, $\Gamma'$ is more constrained than $\Gamma$. The domain store is pruned by applying the following constant-time operations to a variable $x$: fix\_min$(x, a)$ removes from $D(x)$ any value $v < a$, and fix\_max$(x, b)$ removes from $D(x)$ any value $v > b$. The constant-time operation $\_a`*bced$ posts the constraint $d$.

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. 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, in which case $C$ is entailed; 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 solver ensures that all propagation events are eventually served by the relevant filtering algorithms. Bounds adjustments are the relevant propagation events for the $\leq_{\text{lex}}$ constraint.

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
Let \( \mathcal{A} \) be the alphabet \{<, \leq, \geq, ?, \} \}. The signature of a constraint \( C \equiv \vec{x} \leq_{\text{lex}} \vec{y} \) wrt. the current constraint store \( \Gamma \) is a string \( S \) over \( \mathcal{A} \) of length \( n+1 \) where \( S_n = ? \) to mark the end of the string, and for \( 0 \leq i < n \):

\[
S_i = \begin{cases} 
< & \text{if } \Gamma \models x_i < y_i \\
\equiv & \text{if } \Gamma \models x_i = y_i \\
> & \text{if } \Gamma \models x_i > y_i \\
\leq & \text{if } \Gamma \models x_i \leq y_i \land \neg x_i < y_i \land \neg x_i = y_i \\
\geq & \text{if } \Gamma \models x_i \geq y_i \land \neg x_i > y_i \land \neg x_i = y_i \\
? & \text{otherwise}
\end{cases}
\]

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 Table 1. Each letter \( S_i = \sigma(C, i, \Gamma) \) 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>&lt;</td>
<td>( x_i &lt; y_i )</td>
</tr>
<tr>
<td>\equiv</td>
<td>( x_i = y_i )</td>
</tr>
<tr>
<td>&gt;</td>
<td>( x_i &gt; y_i )</td>
</tr>
<tr>
<td>\leq</td>
<td>( x_i \leq y_i \land \neg x_i &lt; y_i \land \neg x_i = y_i )</td>
</tr>
<tr>
<td>\geq</td>
<td>( x_i \geq y_i \land \neg x_i &gt; y_i \land \neg x_i = y_i )</td>
</tr>
<tr>
<td>?</td>
<td>otherwise</td>
</tr>
</tbody>
</table>

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

\[
\Gamma' \sqsubseteq \Gamma \Rightarrow \sigma(C, i, \Gamma') \preceq \sigma(C, i, \Gamma)
\]

For signature letters \( \delta \) and \( \delta' \), \( \delta' \preceq \delta \) means that either \( \delta = \delta' \) or \( \delta \) can change to \( \delta' \) in a more constrained constraint store.

## 3 Ground Instances of \( \leq_{\text{lex}} \)

Given a ground instance \( C \equiv \vec{x} \leq_{\text{lex}} \vec{y} \). Clearly, the set of signatures of true instances is the regular language denoted by:
Fig. 1. Partially ordered set \((A, \preceq)\)

\[
\begin{array}{c}
\preceq \\
\leq \\
< \\
\geq \\
\geq \\
> \\
<
\end{array}
\]

whereas the set of signatures of false instances is the regular language denoted by:

\[
[<][\$]\ldots
\]

Thus, we have reduced the problem of deciding \(\vec{x} \preceq_{\text{lex}} \vec{y}\) to a recognition problem for a simple regular language. We will now extend this idea to non-ground instances and use regular expressions to characterize the various cases where the filtering algorithm can detect failure or entailment, and when it must suspend.

4 Filtering for \(\preceq_{\text{lex}}\)

4.1 A Finite Automaton

Fig. 2 shows a deterministic finite automaton FALEX for signature strings, from which we will derive the filtering algorithm. State 1 is the initial state. There are seven terminal states, F1, T1–T3 and D1–D3, each corresponding to a separate case. Case F1 is the failure case; cases T1–T3 are cases where the algorithm detects that either \(C\) is entailed 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.

4.2 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 arc-consistency. Each regular expression corresponds to one of the terminal states.
of FALEX. Note that, without loss of generality, each regular expression has a common prefix \( P = (=|\geq)^* \). For \( C \) to hold, clearly for each pos. \( i \in P \) where \( S_i = \geq \), 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 F1.**

\[
(=|\geq)^* > \ldots
\]  \hspace{1cm} (F1)

Clearly, if the signature of \( C \) is accepted by F1, the signature of any ground instance of \( C \) will be accepted by GF, so \( C \) has no solution.

**Case T1.**

\[
(=|\geq)^* < q | \ldots
\]  \hspace{1cm} (T1)

\( C \) will hold; we are done.

**Case T2.**

\[
(=|\geq)^* < q | \ldots
\]  \hspace{1cm} (T2)
For $C$ to hold, we must enforce $x_q < y_q$, in order for there to be at least one $\leq$ preceding the first $\geq$ in any ground instance.

Case T3.

$\begin{array}{cccc}
\geq & \geq & \leq & ? \\ p & q & r & \ldots \end{array}$

(T3)

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

Case D1.

$\begin{array}{cccc}
\geq & \geq & \leq & ? \\ p & q & r & \ldots \end{array}$

(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.

$\begin{array}{cccc}
\geq & \geq & \leq & ? \\ p & q & r & \ldots \end{array}$

(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 $x_r > y_r \lor \cdots \lor x_{s-1} > y_{s-1} \lor (s < n \land x_s > y_s)$. 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.

$\begin{array}{cccc}
\geq & \geq & \leq & ? \\ p & q & r & \ldots \end{array}$

(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 $x_r = y_r \land \cdots \land x_{s-1} = y_{s-1} \land x_s > y_s$. Thus, the only relation in $[q, s]$ that doesn’t have support is $x_q > y_q$, so we enforce $x_q \leq y_q$. 6
4.3 A Filtering Algorithm

By augmenting FALEX with the pruning actions mentioned in Sect. 4.2, we arrive at a filtering algorithm for $\leq_{\text{lex}}$, $\text{FiltLex}$. When a constraint is posted, the algorithm will succeed, fail or delay, depending on where FALEX stops. In the delay case, the algorithm will restart from scratch whenever a propagation event arrives, until it eventually succeeds or fails.

We summarize the properties of $\text{FiltLex}$ in the following proposition.

**Proposition 1.**

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

**Proof.**

1. FALEX is a deterministic finite automaton. Each of its four non-terminal state has a defined transition for each letter of $\mathcal{A}$.
2. FALEX has one failure case, F1. In Sect. 4.2, we showed that the corresponding instances have no solutions. Furthermore, no pruning action removes any relation that has support.
3. FALEX has three cases, T1–T3, where it detects entailment, possibly with the aid of posting a primitive constraint. In Sect. 4.2, we showed that all corresponding ground instances are solutions, provided that:
   - $x_i = y_i$ is enforced for all $i \in P$.
   - $x_q < y_q$ is enforced in case T2, if necessary by posting $a < \text{constraint}$.
   - $x_q \leq y_q$ is enforced in case T3, if necessary by posting $a \leq \text{constraint}$.
4. FALEX has three suspension cases, D1–D3. In Sect. 4.2, we showed that in each case, there could be both true and false ground instances, yet no pruning action in $[q, s]$ can eliminate the false cases without also removing some solutions.
5. From items 2 and 3 of this proof, it follows that arc-consistency is maintained in the failure and entailment cases. Consider again cases D1–D3. By the previous item, no pruning action is valid in $[q, s]$, except $x_q \leq y_q$, which must hold in all solutions. Thus, provided that we enforce $x_i = y_i$ for all $i \in P$, and $x_q \leq y_q$, arc-consistency is maintained also in cases D1–D3.
6. Each signature letter is examined at most once, and all decisions and pruning actions run in constant time.

$\square$
5 Incremental Filtering for $\leq_{\text{lex}}$

If $\text{FiltLex}$ suspends, it would be perfectly valid to restart from scratch each time it is resumed by some propagation event. 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 $\text{FiltLex}$ for reaching a leaf of the search tree would be $O(n^2)$. We can do better than that. In this section, we will 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 $C \equiv \overline{x} \leq_{\text{lex}} \overline{y}$ constraint has been posted, $\text{FiltLex}$ 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 $x_i, x_i, y_i, y_i$ have changed.

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

5.1 Naive Event Handling

Our first idea is to simply restart the automaton at pos. $i$, in state $u$. The reasoning is that either everything up to pos. $i$ is unchanged, or there is a pending propagation event at pos. $j < i$, 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:

  \[
  \begin{align*}
  &\overline{x_i} = \overline{x_i} = y_i, \text{ if } \Gamma \models x_i = y_i \\
  &\overline{x_i} = \overline{y_i}, \text{ if } \Gamma \models x_i \geq y_i
  \end{align*}
  \]

  for any constraint store $\Gamma$.

- If $i = q$, we resume in state 1 at pos. $i$.
- If $i = r$, we resume in state 2 at pos. $i$.
- If $u > 2 \land i = s$, we resume in state $u$ at pos. $i$.
- If $u > 2 \land r < i < s$:
  - If the signature letter at pos. $i$ is unchanged or is changed to $=$, we do nothing.

\[\text{Assuming no variable occurs twice; see Sect. 7.}\]
• Otherwise, we resume in state \( u \) at pos. \( i \), immediately reaching a terminal state.
• Otherwise, we just suspend, as FALEX would perform the same transitions as last time.

5.2 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 pos. \( r \) is ignored. This is safe, for the ignored letters will all be \([\leq] \).
– In states 3 and 4, any letter before pos. \( s \) is ignored. Suppose that there is a pending propagation event with pos. \( j \), \( r < j < s \) and that \( S_j \) has changed to \([\leq] \) (in state 3) or \([\geq] \) (in state 4), which should take the automaton to a terminal state. The pending event will lead to just that, when it is processed.

5.3 An Incremental Filtering Algorithm for \( \leq_{\text{lex}} \)

Let \( \text{FiltLexI} \) be the \( \text{FiltLex} \) algorithm augmented with the event handling described in this section, as illustrated by Alg. 1. As before, we assume that each time the algorithm resumes, the constraint store will be more constrained than last time. We summarize the properties of \( \text{FiltLex} \) in Proposition 2 below, but first we need a simple lemma:

**Lemma 1.** From one resumption of \( \text{FiltLexI} \) to the next, \( q, r \) and \( s \) (if defined) are nondecreasing.

**Proof.** Assume that \( q \), the position of the transition out of state 1, has decreased. This implies that arc \( 1 \rightarrow 1 \) is taken fewer times, i.e. that letter \( \delta = [\leq, \geq] \) has changed to a letter \( \delta = [\leq, ?] \). But that is impossible, for \( \delta \not\in \delta \) for all such letters, so the assumption is false.

Assume that \( r \), the position of the transition out of state 2, has decreased. Since \( q \) is nondecreasing, this implies that letter \([\leq] \) has changed, which is impossible, so the assumption is false.

If \( s \) is defined, it is the position of the transition out of state 3 or 4, and \( r \) is the position of the transition to that state from state 2. Assume that \( s \) has
decreased and that \( u = 3 \). Since \( r \) is nondecreasing, this implies that arc \( 3 \rightarrow 3 \) is taken fewer times, i.e. that letter \( \delta \in [\leq, \geq] \) has changed to a letter \( \delta \in [\geq, \leq] \). But that is impossible, for \( \delta \neq \delta \) for all such letters. The letter \( \leq \) can change to \( \leq \), but that leads to state \( T1 \), so \( s \) does not decrease. The analysis is similar for \( u = 4 \), so the assumption must be false.

\[ \Box \]

**Proposition 2.**

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

**Proof.**

1. Consider first the case where a single propagation event has arrived since the algorithm last suspended. It should be clear from Sect. 5.1 and 5.2 that \( \text{FiltLexI} \) stops in the same terminal state as \( \text{FiltLex} \).
Consider now the case of more than one propagation event. \( \text{FiltLexI} \) handles this case by effectively serializing the propagation events, whereas \( \text{FiltLex} \) handles them all at once. Assume now that \( \text{FiltLexI} \) and \( \text{FiltLex} \) arrive at different results. But this means that \( \text{FiltLex} \) would arrive at one result when serializing the events, and at another result when handling them all at once. This would contradict the fact that \( \text{FiltLex} \) maintains arc-consistency (Proposition 1), so the assumption is false.
2. The case analysis in Sect. 5.1 takes constant time. Consider now the total number of times a given FALEX state transition can be made.

- **any state to T1–T3 or F1** At most once.
- **any state to D1–D3** At most \( m + 1 \) times.
- **state 1 to 2, 2 to 3, or 2 to 4** At most \( m + 1 \) times.
- **state 1 to 1** On resumption, pos. \( q \) is examined next here. Each time this transition is made, \( q \) is incremented by one. Since \( q \) is nondecreasing (Lemma 1), at most \( n \) transitions are possible.
- **state 2 to 2** On resumption, pos. \( r \) is examined next here. Each time this transition is made, \( r \) is incremented by one. Since \( r \) is nondecreasing (Lemma 1), at most \( n \) transitions are possible.
- **state 3 to 3 or 4 to 4** On resumption, pos. \( s \) is examined next here, unless a signature letter before \( s \) was changed to \( \leq \leq \) in state 3 (4), in which case state \( T3 \) (T2) is reached immediately. Each time this transition is made, \( s \) is incremented by one. Since \( s \) is nondecreasing (Lemma 1), at most \( n \) transitions are possible.

Thus, the total number of state transitions, and hence the total running time, is \( O(n + m) \).

\[ \Box \]
\textbf{PROCEDURE} FilterLex(\(\bar{x}, \bar{b}, i\) : \((\text{fail, delay, succeed})\) \\
\textbf{Require:} \(i < 0\) on posting the constraint \\
\textbf{Require:} \(i \geq 0\) when handling a propagation event, implies \(q,r,s,u\) have valid values \\
\textbf{Ensure:} delay implies \(q,r,s,u\) have valid values \\
1: if \(i < 0\) then // initial call \\
2: \(q \leftarrow r \leftarrow s \leftarrow 0; \text{goto} \) line 13 // start in state 1 \\
3: else if \(i = q\) then // propagation event at pos. \(i\) \\
4: \text{goto} \) line 13 // resume in state 1 \\
5: else if \(i = r\) then \\
6: \text{goto} \) line 22 // resume in state 2 \\
7: else if \(u = 3 \land (i = s \lor (i < s \land \bar{x}_i \neq y_i))\) then \\
8: \text{goto} \) line 33 // resume in state 3 \\
9: else if \(u = 4 \land (i = s \lor (i < s \land \bar{x}_i \neq y_i))\) then \\
10: \text{goto} \) line 38 // resume in state 4 \\
11: else \\
12: \text{return} \) delay \hspace{1cm} // state 1 \\
13: while \(i < n \land \bar{x}_i = \bar{y}_i\) do \hspace{1cm} // state T1 \\
14: if \(\text{fix_max}(x_i, \bar{y}_i) = \text{fail} \lor \text{fix_min}(y_i, x_i) = \text{fail}\) then // enforce \(x_i = y_i\) \\
15: \text{return} \) fail \\
16: \(q \leftarrow i \leftarrow i + 1\) \\
17: if \(i = n \lor \bar{x}_i < y_i\) then \\
18: \text{return} \) succeed \hspace{1cm} // state T2: rewrite to \(x_q \leq y_q\) \\
19: if \(\text{fix_min}(x_i, y_i) = \text{fail} \lor \text{fix_max}(y_i, x_i) = \text{fail}\) then // enforce \(x_q \leq y_q\) \\
20: \text{return} \) fail \\
21: \(r \leftarrow i \leftarrow \text{max}(i + 1, r)\) \\
22: while \(i < n \land \bar{x}_i = \bar{y}_i = y_i\) do // state 2 \\
23: \(r \leftarrow i \leftarrow i + 1\) \\
24: if \(i = n \lor \bar{x}_i < y_i\) then \\
25: \text{post} \(x_q \leq y_q\); \text{return} \) succeed \hspace{1cm} // state T3: rewrite to \(x_q \leq y_q\) \\
26: else if \(\bar{x}_i > y_i\) then \\
27: \text{post} \(x_q < y_q\); \text{return} \) succeed \hspace{1cm} // state T2: rewrite to \(x_q < y_q\) \\
28: else if \(\bar{x}_i = y_i \land \bar{x}_i < y_i\) then \\
29: \(s \leftarrow i \leftarrow \text{max}(i + 1, s); \text{goto} \) line 33 \\
30: else if \(\bar{x}_i = y_i \land \bar{x}_i > y_i\) then \\
31: \(s \leftarrow i \leftarrow \text{max}(i + 1, s); \text{goto} \) line 38 \\
32: \(u \leftarrow 2; \text{return} \) delay \hspace{1cm} // state D1 \\
33: while \(i < n \land \bar{x}_i = y_i\) do // state 3 \\
34: \(s \leftarrow i \leftarrow i + 1\) \\
35: if \(i = n \lor \bar{x}_i < y_i\) then \\
36: \text{post} \(x_q \leq y_q\); \text{return} \) succeed \hspace{1cm} // state T3: rewrite to \(x_q \leq y_q\) \\
37: \(u \leftarrow 3; \text{return} \) delay \hspace{1cm} // state D3 \\
38: while \(i < n \land \bar{x}_i = y_i\) do // state 4 \\
39: \(s \leftarrow i \leftarrow i + 1\) \\
40: if \(i < n \land \bar{x}_i > y_i\) then \\
41: \text{post} \(x_q < y_q\); \text{return} \) succeed \hspace{1cm} // state T2: rewrite to \(x_q < y_q\) \\
42: \(u \leftarrow 4; \text{return} \) delay \hspace{1cm} // state D2

\textbf{Algorithm 1:} Filtering algorithm for \(\bar{x} \leq_{\text{lex}} \bar{y}\)
6 Related Work

The algorithm by Frisch et al. [2] is based on the idea of using two pointers \( \alpha \) and \( \beta \). The \( \alpha \) pointer gives the position of the most significant pair of variables that are not ground and equal, and corresponds to our \( q \) position. The \( \beta \) 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 domain store gets more constrained, \( \alpha \) and \( \beta \) get closer and closer, and the algorithm detects entailment when \( \alpha + 1 = \beta \lor x_{\alpha} < y_{\alpha} \). 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:

\[
\begin{align*}
x_0 &\in \{0, 1\} & x_1 &= 0 \\
y_0 &= 1 & y_1 &= 1 \\
\langle x_0, x_1 \rangle &\leq_{\text{lex}} \langle y_0, y_1 \rangle
\end{align*}
\]

\textsc{FiltLex} detects entailment (state T3) on this example, whereas Frisch’s algorithm does not (\( \alpha = 0, \beta = \infty \)).

Frisch’s algorithm is shown to run in \( O(n) \) on posting a constraint as well as for handling a propagation event. No better bound than \( O(nn) \) is claimed for posting a constraint followed by \( m \) propagation events, although we think that this bound can be tightened.

On reflection, the fact that the \( \beta \) pointer moves toward more significant positions seems counter-intuitive: it would seem more natural to focus on most significant positions, which carry all the information we need, before examining less significant ones.

7 Extensions

\textit{Strict version}. An algorithm for the \( <_{\text{lex}} \) constraint can be derived easily by modifying the transitions of the finite automaton.

\textit{Variable aliasing}. It is straightforward to make the algorithm take variable aliasing partially into account. All it takes is to make the test \( \Gamma \models x_i = y_i \) sensitive to variable aliasing. If this is done, the variables \( x_i \) and \( y_k, i \in P \) are not necessarily ground, but any propagation events on them can safely be ignored.

Furthermore, if a given pair \( (x_i, y_k) \) of variables occurs more than once, we can simplify the constraint and ignore all occurrences but the first one.
More eager entailment detection. With our definition of \( (\mathcal{A}, \leq) \), for \( S = \{ x_i \} \), we have no information on whether \( x_i = y_i \) is feasible. To check this, we would have to test whether \( D(x_i) \) intersects \( D(y_i) \). This test takes more than constant time for any domain representation (e.g. bit array, list of intervals) that we are aware of. If we knew that \( x_i = y_i \) is infeasible, we could detect entailment in some cases where we now delay instead. Consider the example:

\[
\begin{align*}
    x_0 & \in \{1, 3\} & x_1 & \in \{1, 2\} \\
    y_0 & \in \{2, 4\} & y_1 & \in \{1, 2\} \\
    \langle x_0, x_1 \rangle & \leq_{\text{lex}} \langle y_0, y_1 \rangle
\end{align*}
\]

which yields the signature string \([2 \ 3 \ 8\] \). Thus FALEX stops in state D1, since for all the algorithm knows, \( x_0 = y_0 \) is feasible, and so it may have to enforce \( x_1 \leq y_1 \) in a more constrained constraint store. However, for the three pairs of values for \( (x_0, y_0) \) that have support, namely \((1, 2), (1, 4), (3, 4)\), we have that \( x_0 < y_0 \). With the knowledge that \( x_0 = y_0 \) is infeasible, the automaton could stop in state T1, enforcing \( x_0 < y_0 \).

To implement this idea, the following changes would have to be made:

1. \( (\mathcal{A}, \leq) \) would have to be extended with a new letter, \([\leq]\) say, to capture the case that \( x_i \) can be less than or greater than, but not equal to, \( y_i \).
2. The following finite automaton arcs would be annotated with \([\leq]\): \( 1 \mapsto T1, 2 \mapsto D1, 3 \mapsto D3, 4 \mapsto D2 \).
3. In terminal state T1, the algorithm must enforce \( x_q < y_q \) if \( q < n \).
4. Removing a value from the interior of a domain would be a relevant propagation event.

However, the worse complexity means that this idea is hardly worth considering, especially as no more pruning is gained.

Acknowledgements

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

References