9-2007

Length-Lex Open Constraints

Gregoire Dooms
Brown University

Luc Mercier
Brown University

Pascal van Hentenryck
Brown University

Willem-Jan van Hoeve
Cornell University, vanhoeve@andrew.cmu.edu

Laurent Michel
University of Connecticut - Storrs

Length-Lex Open Constraints

Gregoire Dooms, Luc Mercier,Pascal Van
Hentenryck, Willem van Hoeve and Laurent Michel

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-07-09
September 2007

Length-Lex Open Constraints

Grégoire Dooms[1], Luc Mercier[1], Pascal van Hentenryck[1], Willem-Jan van Hoeve[2], and Laurent Michel[3]

[1] Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, U.S.A. {gdooms,mercier,pvh}@cs.brown.edu
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A. vanhoeve@cs.cornell.edu
[3] Department of Computer Science & Engineering, University of Connecticut, Storrs, CT 06269, U.S.A. ldm@engr.uconn.edu

Abstract. Open constraints were introduced to model the many industrial applications in which a task can be handled by several resources. Open constraints are unique because the set of variables over which the the constraint is defined is a set-variable. Régin and van Hoeve recently showed how to filter an open GCC constraint when the set variable use a subset-bound domain. This paper considers open constraints in which the set-variables use the richer length-lex domain of Gervet and Van Hentenryck which includes cardinality and lexicographic information, while enforcing bound-consistency for a variety of important constraints. The paper makes two orthogonal contributions. First, it shows how to derive a filtering algorithm for the length-lex open constraint from the cost-based version of the closed version. The key insight is that well chosen weights allow to map the total order of length-lex sets with the total order of set weights. Second, it shows how to derive a filtering algorithm for a length-lex open constraint from the filtering algorithm of the subset-bound open constraint. This technique is entirely generic and adds a factor n in complexity to the subset-bound open constraints. The key underlying insight is to recognize that a length-lex interval can be represented as the union of $O(n)$ subset-bound intervals and their cardinalities. This result also allows to systematically lift filtering algorithms from subset-bound intervals to length-lex intervals.

1 Introduction

An open constraint [2, 5] is a constraint whose scope (i.e., the set of variables on which it is defined) is itself a set variable. These constraints are useful in many applications. In particular, they allow to model the many industrial applications in which a task can be handled by several resources. It is thus important to study the design of filtering algorithms for open constraints.

This research direction was considered by van Hoeve and Régin who produced a beautiful result: they showed how to filter an open GCC constraint when the set variable use a subset-bound domain. Their key insight is that the information from the subset-bound interval can directly be encoded inside the network flow, allowing the filtering algorithm to reuse the GCC filtering algorithm. Unfortunately, the subset-bound domains is rather weak in general for representing the values of set variables and it is important to study filtering algorithms for richer set domains, especially the recent length-lex domain introduced of Gervet and Van Hentenryck [3]. Indeed, the length-lex domain has a number of appealing properties: it captures both cardinality and lexicographic constraints directly and allows the design of efficient algorithms for enforcing bound consistency on many constraints.

The goal of this paper is to develop efficient filtering algorithms for length-lex open constraints, i.e., open constraints whose scope is represented by a set variable with a length-lex domain. Since designing filtering algorithms for global constraint is often tedious and error-prone, the paper focuses on generic techniques and makes two orthogonal contributions. First, the paper shows how to derive a filtering algorithm for the length-lex open version of C from its cost-based version. The filtering algorithm adds $O(n)$ in time and space over the complexity of the cost-based filtering algorithm. Second, it shows how to derive a filtering algorithm for a length-lex open constraint from the filtering algorithm of the subset-bound open constraint. This technique is entirely generic and adds a factor $O(n)$ in time complexity to the filtering algorithm for the subset-bound open constraint. The key insight is to recognize that a length-lex interval can be represented as the union of $O(n)$ subset-bound intervals and their cardinalities. In particular, this result allows us to derive a complete filtering algorithm for the length-lex open GCC, adding only an $O(n)$ factor in time. Equally important, the length-lex decomposition enables us to lift any filtering algorithm for a constraint from the subset-bound to the length-lex domain, adding a factor $O(n^k)$ where k is the number of set variables. In particular, this technique allows us to achieve a complete pruning for the conjunction of two length-lex open GCCs by adding a factor $O(n^2)$ to its subset-bound counterpart.

The rest of this paper recalls some basic notions about set domains, open constraints, and the open GCC constraint. It then presents the two main contributions of this paper, before discussing how to lift any constraint from the subset-bound domain to the length-lex domains.

2 Set Variables and Length-Lex Domains

Let $N = \{1, \dots, n\}$ be the universe of values. A set variable S on N is a variable whose possible values are the subsets of N. Given an order $\lhd$ over 2^N, the domain of a set-variable can be represented as an $\lhd$-interval $[L, U]_\lhd$ denoting the set $\left\{ s \in 2^N \mid L \lhd s \lhd U \right\}$. The subset-bound domain [4, 7] is probably the most widely used representation for set domains. It uses inclusion as a partial order and $[L, U]_\subseteq = \{s | L \subseteq s \subseteq U\}$. The consistency notion typically applied to this domain can be defined as follows.

Definition 1 *Let $\lhd$ be an order on 2^N, and S a set variable on N whose domain is $D(S) = [L, U]_\lhd$. Let C be a constraint on S. C is said to be* $\lhd$-consistent *if L is the greatest lower bound and U the least upper bound of values s of S that are consistent. That is,*

$$L = \inf_\lhd \{ s \in [L, U]_\lhd \mid C(s) \}, \qquad U = \sup_\lhd \{ s \in [L, U]_\lhd \mid C(s) \}.$$

The definition captures the main drawback of the order $\subseteq$: these greatest lower bounds or least upper bounds can be far from any actual set consistent with the constraint.[4] The cardinal domain [1] is an attempt to remedy this limitation by including constraints on the size of the sets.

Definition 2 *The* subset+card *domain consists of tuples* $\langle L, U, c_l, c_u \rangle$ *denoting the sets of sets* $\{s | L \subseteq s \subseteq U \wedge c_l \leq |s| \leq c_u\}$*. These tuples are denoted* $[L, U]\#[c_l, c_u]$ *for syntactic convenience. A constraint* $C(S)$ *over subset-card variable* S *with domain* $[L, U]\#[c_l, c_u]$ *is* $\subseteq/\#$-consistent *if*

$$L = \inf_{\subseteq} \{ s \,| s \in [L, U]\#[c_l, c_u] \wedge C(s) \}, \qquad U = \sup_{\subseteq} \{ s \,| s \in [L, U]\#[c_l, c_u] \wedge C(s) \},$$

$$c_l = \min \{ |s| \,| s \in [L, U]\#[c_l, c_u] \wedge C(s) \}, \quad c_u = \max \{ |s| \,| s \in [L, U]\#[c_l, c_u] \wedge C(s) \}.$$

Such an hybrid domain still does not ensure consistency of the bounds. Using a total order over 2^N solves this issue. Indeed, when $\lhd$ is a total order (that is, for any s and t, either $s \lhd t$ or $t \lhd s$), the greatest lower bound and the least upper bounds are necessarily solutions. The length-lex order recently proposed in [3] is total and these authors provided even additional evidence about why it should be preferred to the subset-bound or subset+card. In particular, it admits efficient filtering for a variety of important constraints.

Definition 3 *For* $s \subseteq N$*, denote* $s = \{s_1, \ldots, s_{|s|}\}$*, with* $s_1 < s_2 < \ldots s_{|s|}$*. The length-lex order over* 2^N *is a total order denoted* $\preceq$ *and defined as*

$$s \preceq t \iff \begin{cases} & s = \emptyset \\ \textit{or} & |s| < |t| \\ \textit{or} & |s| = |t| \neq 0 \textit{ and } s_1 < t1 \\ \textit{or} & |s| = |t| \neq 0 \textit{ and } s_1 = t1 \textit{ and } s \setminus \{s_1\} \preceq t \setminus \{t_1\}. \end{cases}$$

This order induces a strict order $\prec$ *defined as* $s \prec t \iff (s \preceq t \wedge s \neq t)$*.*

As an example, if $N = \{1, 2, 3\}$, the order is:

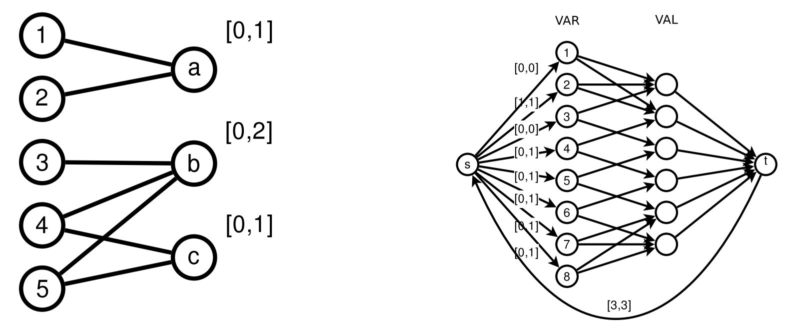
$$\emptyset \prec \{1\} \prec \{2\} \prec \{3\} \prec \{1, 2\} \prec \{1, 3\} \prec \{2, 3\} \prec \{1, 2, 3\}$$

3 Open Constraints

The *scope* of a constraint is the set of variables it involves and open constraints are constraints whose scope is itself a set variable.

Definition 4 *Let* $X = \{x_1, \ldots, x_n\}$ *be a set of* n *variables,* $N = \{1, \ldots, n\}$*, and* C *be a constraint whose semantics is well-defined when its scope is any subset of* X*. Given a set variable* S *with universe* N*, the open constraint* $C(S, X)$ *has the following semantics: the assignment* $\{S \leftarrow \{i_1, \ldots, i_k\}; x_1 \leftarrow v_1; \ldots; x_n \leftarrow v_n\}$ *is a solution to* $C(S, X)$ *if and only if the assignment* $\{x_{i_1} \leftarrow v_{i_1}; \ldots; x_{i_k} \leftarrow v_{i_k}\}$ *is a solution to* C*.*

[4] This is why calling this notion bound consistency is misleading: the bounds are typically not consistent.



(a) A Flow Graph with 5 Variables and 3 Values.

(b) Capacity encoding of a subset-card interval constraint

Fig. 1. Flow graphs for the open-GCC constraint

In the following, the set X is often implicit and we sometimes abuse notations and do not distinguish the set of indices S from the set of variables it denotes.

Figure 1(a) illustrates the concept of open constraints with the GCC constraint. The figure uses the set of variables $X = \{x_1, x_2, x_3, x_4, x_5\}$ and the values $\{a, b, c\}$, The closed constraint $GCC(X)$ is inconsistent, because both x_1 and x_2 have to be assigned to a, which cannot accommodate more than one variable. However, if S is a set variable with domain $S \in [\{x_1\}, X]_\subseteq$, the open constraint $GCC(S, X)$ is consistent and the filtering algorithm should update the domain of S to $S \in [\{x_1\}, \{x_1, x_3, x_4, x_5\}]_\subseteq$

An open constraint must prune both its scope, represented by the set variable S, and the remaining finite domain variables $x_1, \cdots, x_5$.

Definition 5 [5] *An open constraint $C(S, X)$ is said to be* arc consistent *if its finite domain variables are all arc consistent, meaning for each $x_i \in X$,*

1. *either there exists $s \in D(S)$ such that $i \notin s$*
2. *or for all $d \in D(x_i)$, there exists $s \in D(S)$ such that there is a solution of $C(X, S)$ in which S is assigned to s, x_i to d and other variables $x_j (j \neq i)$ to a value in their domain $D(x_j)$.*

An open constraint is said to be $\preceq$-consistent *if variable S is $\preceq$-consistent for the constraint $C(S, X) \wedge \exists x_1, \cdots, x_n : x_1 \in D(x_1) \wedge \cdots \wedge x_n \in D(x_n)$. An open constraint is* $\preceq$-arc-consistent *if it is arc consistent and $\preceq$-consistent. The concepts of $\subseteq/\#$-consistency and $\subseteq/\#$-arc-consistency are similarly defined.*

4 Filtering an Open GCC with Subset+Card Bounds

This section shows how to filter an open GCC when the set variable uses a subset-bound representation and a cardinality interval. It corrects the algorithm in [5], which

[5] The definition of set domain consistency [5, def.3] is incorrect and should not be used anymore. Willem van Hoeve was contacted and agreed that the present definition should be used instead.

was based on an incorrect definition, and is quite similar to the standard GCC algorithm [8]. The open algorithm uses a network flow whose structure is as follows.

1. a source s and a sink t, and an arc (t, s) of capacity $[0, \infty]$;
2. for each $x \in X$, a variable node x and an arc (s, x) with capacity c_x;
3. for each $d \in \bigcup_{x \in X} D(x)$, a value node d and an arc (d, t) with capacity $c_d = [l_d, u_d]$ if between l_d and u_d variables need to be assigned to d;
4. for each $x \in X$ and $d \in D(X)$, an arc (x, d) of capacity [0,1].

In a standard GCC, $c_x = [1, 1]$ for all variable x: a feasible flow has to span, or cover, every variable. In that case, there is a one-to-one correspondence between feasible flows and possible assignments of variables.

In an open GCC, the set variable S specifies the variables involved in the GCC. If $D(S) = [L, U]_{\subseteq}$, then variables in L must be in S, which can be expressed by $c_x = [1, 1]$ for $x \in L$. Variables not in U are prohibited, which is modelled by setting $c_x = [0, 0]$ for $x \notin U$. Finally, the remaining variables are optional, which is represented by $c_x = [0, 1]$ for $x \in U \setminus L$.

The GCC filtering algorithm shows how to determine efficiently if an arc belongs to all feasible flows, some feasible flows, or no feasible flow. To filter the scope S, it suffices to apply this technique to the set of arcs from the source to the variable nodes: If an arc is mandatory, the corresponding variable must belong to S; if it is forbidden, it cannot belong to S. Once the set is filtered given a new domain $[L', U']_{\subseteq}$, the open algorithm filters the variables in L', since the remaining variables satisfy condition (1) of Definition 5. Again, a variable x in L' can be pruned by considering the arcs (x, d) using the standard GCC technique.

The above algorithm can be easily extended to subset+card domains consisting a subset-bound domain and a cardinality interval $[l, u]$. Indeed, to represent the cardinality constraint $l \leq |S| \leq u$, simply change the capacity of the arc (t, s) to $[l, u]$ and any feasible flow will necessarily span between l and u variables. The resulting algorithm still has a complexity of $O(nm)$ where $n = |X|$ and $m = \sum_{x \in X} |D(x)|$. Unfortunately, it is less obvious to achieve $\preceq$-consistency when S has a length-lex domain. Indeed, the fundamental contribution of Régin and van Hoeve is to recognize that the information in the subset-bound domain is naturally encoded in the underlying network flow. But it does not suffice to require or exclude variables. The next two sections present two generic and orthogonal techniques to do so.

5 Length-Lex Open Constraints: A Cost-Based Approach

In this section, we show how to automatically obtain a filtering algorithm for a length-lex open $C(S, X)$ from the cost-based filtering algorithm for C. For instance, the filtering algorithm of a length-lex open GCC is obtained directly from the cost-based GCC.

A semi-generic opening technique The constraint C must be non-strict which intuitively means that it must provide a way to represent the non-assignment of variables. More formally, given some specific values $\perp_i$ (e.g., brand new values) for each variable x_i, a constraint is non-strict if, for each solution $\{x_1 \leftarrow v_1; \ldots; x_i \leftarrow \perp_i; \ldots; x_n \leftarrow v_n\}$ of C, the assignment $\{x_1 \leftarrow v_1; \ldots; x_{i-1} \leftarrow v_{i-1}; x_{i+1} \leftarrow v_{i+1}, \ldots, x_n \leftarrow v_n\}$ is also a solution of C. For instance, the GCC constraint is non-strict: we introduce a new value $\perp_i$ in the domain of each variable x_i, and specify that between 0 and 1 variable has to be assigned to $\perp_i$. Every solution assigning $\perp_i$ to x_i is also a solution to the GCC in which x_i has been removed. The introduction of such specific values in a constraint $C(Y)$ allow to build a subset-bounds open version of the constraint, $C(S, X)$. Let $Y = \{y_1, \ldots, y_n\}$ be fresh variables with initial domains $D(y_i) = D(x_i) \cup \{\perp_i\}$. To filter $C(S, X)$ with $S \in [L, U]$, first filter $C(Y)$. Then, if a variable y_i gets assigned to $\perp_i$, post the constraints $i \notin S$. If $\perp_i \notin D(y_i)$, post the constraints $i \in S$ and $x_i = y_i$. Propagating these new constraints will not require to iterate: a fixpoint will immediately be reached.

The Reduction Consider a cost vector $(W^i(d))_{d \in D(x_i)}$ for each variable x_i. Semantically, the cost-based version of a constraint $C(X)$ with respect to the cost vectors is the constraint $P_C^W(X, \underline{w}, \overline{w})$ over X specified as the conjunction $C(X) \ \wedge \ \underline{w} \leq \sum_{i=1}^{n} W^i(x_i) \leq \overline{w}$.

Let $w_1^n, \ldots w_n^n$ be a vector of weights and define $w : 2^N \rightarrow \mathbb{N}$ by $w(s) = \sum_{i \in s} w_i^n$. Assume that w is an *order isomorphism* with respect to the length-lex ordering, that is $\forall s, t \subseteq N, \ s \prec t \iff w(s) < w(t)$. As an example, consider the universe $\{1, 2, 3, 4\}$ and the weights $w_1^4 = 3$, $w_2^4 = 5$, $w_3^4 = 6$ and $w_4^4 = 7$. The following table lists some sets and their weights:

set	$\{4\}$	$\{1,2\}$	$\{3,4\}$	$\{1,2,3\}$	$\{2,3,4\}$	$\{1,2,3,4\}$
weight	7	8	13	14	18	21

With such weights at hand, the filtering of an open length-lex constraint $C(S, X)$ can be reduced to the filtering of the opened version of the cost-based version of C. Define weights for the Y variables as $W^i(d) = w_i^n \ \forall d \neq \perp_i$ and $W^i(\perp_i) = 0$. Since there is a direct mapping between the set weights and the length-lex order, by enforcing bounds on the weights with $\underline{w} = w(L)$ and $\overline{w} = w(U)$, we actually make sure the solution takes its value in the length-lex interval. $\forall d \neq \perp_i$ and $W^i(\perp_i) = 0$. Since there is a direct mapping between the set weights and the length-lex order, by enforcing bounds on the weights, we actually make sure the solution takes its value in the length-lex interval. If the filtering algorithm is able to compute new tight bounds, we get new length-lex bounds on our set S', otherwise, since the semi-generic opening technique allows to infer inclusion and exclusion constraints, we can post them on the length-lex set to filter it.

This generic reduction is a real strength of length-lex domains. Indeed, such a reduction is not possible using subset-domains, because there is no order isomorphism from a partially ordered set to a totally ordered one. Note however that establishing arc-consistency is often NP-hard for cost-based constraints with the cost bounded from above and below (reduction from SUBSET-SUM). However many such algorithm provide arc consistency in polynomial time when the cost is bounded for one side only, still leading to partial consistency with two bounds.

For instance, to propagate the open constraint $GCC(S,X)$, simply propagate the cost-based version $GCC(Y,\underline{w},\overline{w})$ with variables y_i of domains $D(y_i) = D(x_i) \cup \{\perp\}_i$ and with no cardinality restriction on $\perp_i$ (or [0,1]). Since the cost-based GCC [9] can only handle one bound at a time, the resulting filtering is not complete.

Computing the weights It remains to show how to compute the weights.

Theorem 1 *Let $w_i^n = 2^n - 2^{n-i}$. Then, with $w : 2^N \rightarrow \mathbb{N}$ defined by $w(s) = \sum_{i \in s} w_i^n$, we have $\forall s, t \subseteq N,\ s \prec t \iff w(s) < w(t)$.*

Proof. (a) First we prove the $\Rightarrow$. Suppose $s \prec t$. If $|s| < |t|$, $w(t) > (|t|-1)2^n$, because the subtracted terms cannot sum to more than $2^n - 1$. Moreover, $w(s) \leq |s|\, 2^n$, and so $w(s) < w(t)$.

Now suppose $|s| = |t|$. Let x be the smallest element of $s \setminus t$ and symmetrically let y be the smallest element of $t \setminus s$. Then $x < y$, and we want to bound $w(t) - w(s) = \sum_{i \in s \setminus t} 2^{n-i} - \sum_{i \in t \setminus s} 2^{n-i}$. The first term is lower-bounded by $\sum_{i \in s \setminus t} 2^{n-i} \geq 2^{n-x}$, and the second is upper-bounded by $\sum_{i \in t \setminus s} 2^{n-i} \leq \sum_{k=x+1}^{n} 2^{n-k} = 2^{n-x} - 1$. Thus $w(t) - w(s) \geq 1$.

(b) Now we prove $\Leftarrow$. Let s and t such that $w(s) < w(t)$. Since $\preceq$ is total, we have $s \prec t$, $s = t$ or $s \succ t$. In the last case, (a) implies $w(s) > w(t)$: contradiction. If $s = t$, then $w(s) = w(t)$: contradiction. So $s \prec t$. $\square$

Using these weights, the maximal weight of a set is $\sum_{i=1}^{n} w_i^n = (n-1)2^n + 1$. The set can be represented with $O(n)$ bits which slows down a cost-based filtering algorithm by the same factor by using arbitrarily large integers, as offered by the library gmp in C, or the class BigInteger in Java. We can do a little better by observing that [6] studies an equivalent question in voting theory and gives a recursion formulae for weights conjectured to be minimal. However, for $n = 10^4$, these better weights only save 3 bits for the binary representation of $w(N)$. Indeed, using $O(n)$ bits is optimal, because there are 2^n sets that must have all different weights, so the largest set must weight at least 2^n.

6 Length-Lex Open Constraints: A Partition Approach

The last section shows how to obtain a filtering algorithm for the length-lex open version of a constraint C from the cost-based version of C. This section presents an orthogonal result. It demonstrates how to obtain a filtering algorithm for the length-lex open version of a constraint C from the filtering algorithm designed for the subset+card open constraint. The result is entirely generic: The subset+card filtering algorithm is used as a black-box. Moreover, the generic scheme preserves consistency: it achieves arc-consistency on the scalar variables and $\lhd$-consistency on the set variable for the length-lex domain whenever the black-box produces similar results for the subset+card domain. The key insight is to recognize that a length-lex interval can be seen as the union of $O(n)$ subset+card intervals.

Algorithm 1: $Filter(X, S)$ is the generic filtering algorithm for open length-lex constraints.

```
[L_⪯, U_⪯] ← S;
S ← [ ComputeNewLB(L_⪯), ComputeNewUB(U_⪯) ];
A ← {(i, v) | i ∈ [1, n] & v ∈ D(x_i)};
R ← [1, n];
foreach [L, U]#[c_l, c_u] ∈ PartitionInterval(S) do
    ⟨F, [L', U']#[c'_l, c'_u]⟩ = SBCFilter(X, [L, U]#[c_l, c_u]);
    A ← A ∩ F;
    R ← R ∩ L';
foreach i ∈ R ∪ Req(S) do
    D(x_i) ← D(x_i) \ {v|(i, v) ∈ A};
```

Algorithm 2: **FindMinCard**(i,$[M, P]_{\subseteq}$,c), initially called as **FindMinCard**(1,$[\emptyset, N]_{\subseteq}$, c)

```
while |M| ≤ c and i ≤ n do
    if SBCFeasible([M ∪ {i} , P]#[c, c]) then
        M ← M ∪ {i};
    else
        P ← P \ {i};
    i ← i + 1;
if i > n then return ⊥;
else return M;
```

Algorithm 3: **FindMinBoundedFC**(L,i,$[M, P]_{\subseteq}$), initially called as **FindMinBoundedFC**(L,1,$[\emptyset, N]_{\subseteq}$).

```
if i > |L| then return L;
if SBCFeasible([M ∪ {L_i} , P \ {L_{i−1} . . . L_i − 1}]#[|L|, |L|]) then
    Sol = FindMinBoundedFC(L, i + 1, [M ∪ {L_i} , P \ {L_{i−1} . . . L_i − 1}]_⊆);
    if Sol ≠ ⊥ then return Sol;
return FindMinCard(L_i + 1, [M, P \ {L_i}]_⊆, |L|);
```

Algorithm 4: **Split**($[L, U]_{\preceq}$)

```
if |L| < |U| then
    intervals ← SplitLowerBounded(L);
    foreach k ∈ [|L| + 1, |U| − 1] do
        Append CardInterval(k) to intervals;
    Concat SplitUpperBounded(U) to intervals;
else
    intervals ← SplitSameCard(L, U);
intervals2 ← list();
foreach i ∈ intervals do
    Concat SplitFurther(i) to intervals2;
return intervals2;
```

Algorithm 5: **SplitLowerBounded**(L)

intervals $\leftarrow$ list();
for $i \leftarrow (|L| - 1)$ **downto** 0 **do**
  U $\leftarrow$ `maxWithPrefix`(prefix$(L, i), |L|$);
  if $L \leq U$ **then** `Append` $[L, U]$ to intervals ;
  $L \leftarrow$ `next`(U);
`Append` $[L,$ `maxWithCard`$(|L|)]$ to intervals;
return intervals;

Algorithm 6: **SplitFurther**($[L, U]_{\preceq}$)

intervals $\leftarrow$ list();
for $i \leftarrow (|L| - 1)$ **downto** 0 **do**
  if $L_i \neq U_i$ **then** **break** ;
if $i = |L| - 1$ **then**
  `Append` $[L, U]$ to intervals;
  return intervals;
forall $v \in [L_i..U_i]$ **do**
  $L2 \rightarrow$ `minWithPrefix`$(L_{[0..i]} + v, |L|)$;
  $U2 \rightarrow$ `maxWithPrefix`$(L_{[0..i]} + v, |L|)$;
  `Prepend` $[L2, U2]$ to intervals;
return intervals;

6.1 The Generic Filtering Algorithm

Algorithm 1 presents our generic algorithm to implement a length-lex filtering algorithm based on a subset+card filtering algorithm.

Bound consistency is achieved for the length-lex interval by explicitly computing minimal and maximal solutions according to the length-lex order (line 2). This step relies on the function `SBCFeasible` which determines if the constraint has a solution for a given subset+card interval. The complexity of this step is $O(F + n^2 I)$, where F is the complexity of the subset+card feasibility algorithm and I is the incremental feasibility test upon insertion or deletion of a value in the subset+card domain.

The algorithm then filters the finite-domain variables and relies on the availability of a filtering algorithm for the open constraint with a subset+card interval: `SBCFilter(L,U,l,u)` returns a new subset+card interval and a set F of filtered variable-value pairs (called arcs from now on). Our generic algorithm proceeds by partitioning the length-lex interval S into a linear ($O(n)$) number of subset+card intervals (line 5). We apply `SBCFilter` on each of those intervals and collect the new set bounds and the filtered arcs (line 6). The complexity of this step is $O(nP)$, where P is the complexity of the subset+card filtering algorithm. An arc can only be pruned if it is filtered for each sub-interval. Thus the generic algorithm computes the intersection of the sets of filtered arcs (line 7). By definition of open constraints, a variable filtering is possible only if the variable is known to be in the scope of the open-constraint. Thus, the algorithm computes the intersection of the subset+card lower bounds returned for each sub-interval (line 8) and, for each variable in this intersection or in the variables determined to be required based on S (line 9), it removes from its domain all values filtered in all sub-intervals (line 10). The complexity of this step is $O(m+n)$. This gives an overall time complexity of $O(F+n^2I+nP+m)$. For the length-lex open GCC, this gives an $O(n^2(n+m))$ time complexity.

6.2 Computing New Length-Lex Bounds

To update the bounds of the length-lex set, we must compute length-lex minimal and length-lex maximal solutions. We first show how to compute a length-lex minimal solution of fixed cardinality. We then extend this algorithm to find a length-lex minimal solution greater or equal to some lower bound. The upper bound computation is symmetric.

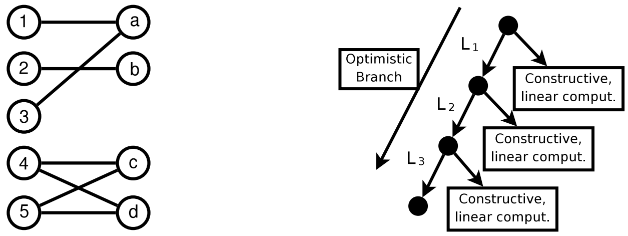
Computing a Length-Lex Minimum Solution of Fixed Cardinality To motivate the algorithm, consider the simple open GCC in figure 1(a), in which variables 1 and 2 cannot belong to a solution simultaneously. In a length-lex interval of minimum cardinality 3, this prunes the sets $\{1,2,3\}$, $\{1,2,4\}$, and $\{1,2,5\}$ from the domain of the set variable. This is achieved by computing the length-lex minimal set $\{1,3,4\}$ of 3 variables which satisfies the Open-GCC.

Algorithm 2 starts with an empty set and iteratively tries to include one additional variable in increasing order[6] Consider the GCC graph of Figure 1(a). The algorithm starts with an empty set of variables. The first iteration tries to insert variable 1 in the lower bound of the subset+card interval $[M,P]\#[c,c]$ and checks for feasibility. Clearly this is satisfiable, so this variable is inserted in line 3. The second iteration tries to force variable 2 into the set but this fails, since value a can be used at most once: this variable is removed from the upper bound on line 5. The algorithm proceeds with variable 3 then 4 to obtain the length-lex minimal set of 3 variables which satisfies the open-GCC.

Computing a Length-Lex Minimal Solution Bounded by Below We now generalize the algorithm to return a length-lex minimal solution greater than a lower bound. The algorithm first searches for a set s of cardinality $|L|$ with $L \preceq s$ using Algorithm 3. If it fails, it applies Algorithm 2 with increasing cardinalities.

Algorithm 3 aims at finding a set $s \succeq L$ with $|s| = |L|$. It has the same common structure as many algorithms of [3]: it is recursive and can backtrack. Moreover, it only backtracks to one of the values of L and then proceeds without branching again. As illustrated in Figure 2(b), this bounds the number of backtracks to $|L|$. The left branch aims at finding a solution covering exactly the length-lex lower bound L. We call this branch the optimistic mode of the algorithm. While the algorithm is optimistic, it includes all variables of L in increasing order, while forbidding the variables which are not part of L. Each recursive call is done for a variable of L and tries to compute a solution including this variable. As soon as one variable i of L cannot be included in a solution, the algorithm goes into a constructive mode in which it tries to extend the current partially computed set greedily. If it is not possible, it backtracks to the last

[6] The parameters $[M,P]$ can be ignored for now and considered to be equal to $[\emptyset, N]$; They will only be used in the next section.



(a) Matching example (b) Structure of the backtracking

Fig. 2. The length-lex minimum set greater or equal to some bound

variable $j \in L$ considered while in optimistic mode. Since the currently included set of variables forms a prefix which cannot lead to a feasible solution, the algorithm removes variables j from the set and tries all subsequent variables in constructive mode.

Consider the matching example in Figure 2(a), the length-lex minimum matching of cardinality 3 is $\{1, 2, 4\}$. Now consider how Algorithm 3 behaves for a lower bound $\{1, 3, 4\}$. It first includes variable 1 in the set and this variable can be assigned value a. Since a set $\{1, 2, x\}$ would be lower than our lower bound, the algorithm excludes variable 2 from the set. Variable 3 is then considered and produces the prefix $\{1, 3\}$. But introducing this variable in the set fails the open GCC, since the only value 3 can take is a and a is already used by variable 1. Finally, the algorithm considers variables 4 and 5 which can both be inserted into the set, giving the result: $\{1, 4, 5\}$. This example illustrates the optimistic and constructive modes but does not illustrate backtracking. If the lower bound is $\{1, 3, 4, 5\}$, backtracking would be necessary and would lead to the result $\{2, 3, 4, 5\}$.

Observe again that if Algorithm 3 does not find a solution of cardinality $|L|$, it is necessary to consider cardinalities $|L| + 1$, $|L| + 2$,... using algorithm 2. In the GCC case, if there is no solution of cardinality $|L| + 1$, there is no solution of cardinality $|L| + 2$ and only one iteration is necessary.[7]

6.3 Partitioning a Length-Lex Domain

The problem we are facing in the length-lex open-GCC is that we cannot directly force the flow to belong to a length-lex interval and compute the set of arcs which have a support. As we saw in section 4, it is much simpler in the subset bounds open-GCC, since this can be done by adjusting some capacity intervals. Hence by encoding our length-lex domain as an union of subset+card, we can compute filtered arcs for each sub-problem then take their intersection. The beauty of this solution is that it applies to any length-lex constraint.

The partition of a length-lex interval in several subset+card intervals is done by identifying sub-intervals of the length-lex order which define the same set of sets as some subset+card interval. Formally, these intervals $[L, U]$ are such that

[7] This is true if $|L| + 1$ is large enough to satisfy the lower capacity bounds in the GCC. There is no point in trying to find flows of value lower than this lower limit.

$[L, U]_{\preceq} = [L, U]_{\subseteq}$. A first type of length-lex interval which is equivalent to a subset+card interval is $[minWithCard_C, maxWithCard_D]$ where $minWithCard_C = \langle 1, 2, \ldots, C\rangle$ and $maxWithCard_C = \langle n - C + 1, n - C + 2, \ldots, n\rangle$. This interval is simply converted to a subset+card interval $[\emptyset, N]\#[C, D]$. Another type of set interval which can be expressed by intervals of the two types of set domains is $[minWithPrefix_C(p), maxWithPrefix_C(p)]$ where $minWithPrefix_C(p)$ is the smallest set of card C containing a prefix p: if $p =< p_1, p_2, \ldots, p_n >$, we get $minWithPrefix_C(p) =< p_1, p_2, \ldots p_n, p_n + 1, \ldots, p_n + C - n >$. All elements in $[1, p_n]$ are either included in all sets (if in p) or included in no sets, thus they define the lower bound and the impossible elements of the subset+card interval. This form of intervals can be extended to

$$[minWithPrefix_C(p \cup \{i\}), maxWithPrefix_C(p)]$$

where $i > \max(p)$. All elements between p_n and i are excluded from the subset+card interval. Note that p can be empty in which case $maxWithPrefix_C(p)$ is equivalent to $maxWithCard_C$. Note that other, more complex, intervals also exist (e.g., $[\{1, 3, 4, 6, 7\}, \{1, 4, 5, 6, 7\}]$ with universe $[1, 7]$). However, these are not necessary for our results.

The following three examples illustrate the three types of interval. The first is bounded only by cardinality, the second has a common prefix and the third has an augmented prefix in the lower bound:

$$\begin{aligned}
[\{1,2,3,4\}, \{5,6,7,8,9\}]_{\preceq} &= [\emptyset, \{1,2,3,4,5,6,7,8,9\}]_{\subseteq}\#[4,5] \\
[\{2,4,5,6\}, \{2,4,8,9\}]_{\preceq} &= [\{2,4\}, \{2,4,5,6,7,8,9\}]_{\subseteq}\#[4,4] \\
[\{2,4,6,7\}, \{2,4,8,9\}]_{\preceq} &= [\{2,4\}, \{2,4,6,7,8,9\}]_{\subseteq}\#[4,4]
\end{aligned}$$

If $|L| < |U|$, a length-lex interval is first split in three sub-intervals:

$$\begin{aligned}
[L, U] = {} & [L, maxWithCard(|L|)] \cup \\
& [minWithCard(|L| + 1), maxWithCard(|U - 1|)] \cup \\
& [minWithCard(|U|), U]
\end{aligned}$$

The middle interval is either empty or equivalent to a subset+card interval. The two other intervals must be split based on common prefixes. If $|L| = |U|$, the interval is bounded from both directions and is handled directly by a slightly more complex splitting algorithm. The complete algorithm is presented in algorithm 4 and we successively discuss the first case (bounded from below by L), the third case (bounded from above by $|U|$) and the $|L| = |U|$ case.

The Lower Bound Case The key idea behind the decomposition is to reason about prefixes. Consider $L = \{2, 3, 5\}$ and $N = [1, 8]$. The sets of cardinality 3 greater than L can share the prefixes $(2, 3)$, (2), and the empty prefix with L, giving us the decomposition

$$[\{2, 3, 5\}, \{2, 3, 8\}] \cup [\{2, 4, 5\}, \{2, 7, 8\}] \cup [\{3, 4, 5\}, \{6, 7, 8\}] .$$

Algorithm 5 shows how to compute those sub-intervals. Intuitively, the algorithm proceeds from right to left, starting with a prefix containing all variables but the last one and continuing with smaller and smaller prefixes. At each step, the added set starts with a prefix and goes to the maximal set with the prefix. After each iteration, the algorithm takes the next set (greater than the largest set produced so far) as the new lower bound.[8]

The Upper Bounded and Fixed Cardinality Cases We now consider the case when the interval is bounded from above by $|U|$. We can proceed as before, but with a small addition. Splitting like in lower-bound case, by proceeding with decreasing prefixes, leads to sub-intervals which are not equivalent to subset+card intervals. For instance, by splitting all sets of cardinality 3 smaller than $U = \{2, 5, 7\}$ with $N = [1, 8]$, we obtain $[\{2, 3, 4\}, \{2, 4, 8\}]$ as one of the sub-intervals. This interval features an "at least one out of" constraint: either value 3 or value 4 must be present in the set. To obtain subset+card algorithms, it is necessary to enumerate the values subject to that disjunctive constraint and obtain sub-intervals with bounds sharing a prefix: For instance $[\{2, 3, 4\}, \{2, 3, 8\}]$. Algorithm 6 performs this splitting[9]

When considering intervals with bounds of the same cardinality, it suffices to generate the sub-intervals for each bound independently by decreasing prefix according to algorithm 5 and its upper bound counterpart. Then the algorithm must select all sub-intervals included in the interval and generate an additional sub-interval to fill the gap in the middle. All those intervals need to be processed by algorithm 6.

For each of these cases, the total number of intervals is at most $2n$. There are $n - 1$ prefixes and the total additional number of splitting over all prefixes is bounded by n since it can only consider a value once.

7 General Length-Lex Constraints

The decomposition technique introduced in the previous section suggests a way to lift any constraint for which there exists filtering algorithms for the subset+card domains to the length-lex domain.

Indeed, consider a constraint $C(S_1, \ldots S_k)$ over k variables, whose domains are $D(S_i) = [L_i, U_i] \forall i \in [1..k]$ and the decompositions of all the intervals $[L_1, U_1], \ldots, [L_k, U_k]$ introduced in the previous section: $[L_i, U_i] = \bigcup_{j=1}^{n_i} [L_i^j, U_i^j] \# [l_i^j, u_i^j]$. Then $C(S_1, \ldots, S_k)$ can be made length-lex-consistent using $\prod_i n_i$ calls to an algorithm that establishes subset+card consistency for the same constraint.

This basic idea is as follows: first enumerate all the vectors $(z_1, \ldots, z_k)$ in $Z = [1, n_1] \times \cdots \times [1, n_k]$. For each such vector z, C_z denotes the problem consisting of the

[8] Note that by carry propagation, this set might not share a prefix of length $|L| - 2$ with L which justifies the *if* statement in the algorithm. See for instance $L = \{1, 2, 4, 5, 6\}$ with universe $[1, 6]$.

[9] Note that this post-processing can be avoided for the length-lex Open-GCC, by encoding the "at least one out of" constraint in the flow graph by using an intermediate source node which mediates the flow going to those variables and with a lower bound capacity of 1

constraint C with domains

$$S_1 \in [L_1^{z_1}, U_1^{z_1}]\#[l_1^{z_1}, u_1^{z_1}], \ldots, S_k \in [L_k^{z_k}, U_k^{z_k}]\#[l_k^{z_k}, u_k^{z_k}].$$

For all z, call a subset+card filtering algorithm to filter the problem C_z leading to new filtered subset+card domains D_i^z for each variable S_i. It remains to compile all these domains D_i^z into new length-lex domain for variables S_i. Let $z_i^j = \{z | z_i = j\}$. To compute the new length-lex domain of S_i, find the first sub-interval f in which it has at least a support: $f = \mathrm{argmin}_j \left\{ z \in z_i^j \middle| Consistent(C_z) \right\}$. Then, the minimal length-lex support among all constraints using this sub-interval is the new lower-bound of S_i: [10]

$$L_i \leftarrow \min_{\preceq} \left\{ \min_{\preceq}(D_i^z) \, | z \in z_i^f \right\}$$

Note that $\min_{\preceq}([L, U]\#[l, u])$ can be directly implemented using algorithm 2 with cardinality l and feasibility test for the constraint $C(S) \equiv S \in [L, U]\#[l, u]$.

Our generic algorithm calls the subset-card filtering algorithm $O(n^k)$ times and then needs $O(n^{k+1})$ to compile the results. For example, for the $PARTITION(S_1, S_2)$ constraint defined as $S_1 \cap S_2 = \emptyset \wedge S_1 \cup S_2 = N$, this reduction runs in $O(n^3)$ because the subset+card $PARTITION$ constraint can be filtered in linear time.

The main problem of this algorithm is the exponent k in the complexity. To get ride of it, we consider another way to represent a length-lex interval $I = [L, U]$ in subset+card, by relaxing it: $I \subseteq [Req(I), Poss(I)]\#[|L|, |U|]$ where Req and $Poss$ denote required and possible elements in the interval (see [3] for a complete characterization). A natural relaxation of a constraint then consists in considering the interval decomposition of only one variable at a time, relaxing the intervals of other variables: that is, to filter the i-th interval, consider successively the n_i subintervals of $[L_i, U_i]$, and relax intervals $D(S_h)$ for $h \neq i$.

An application of this idea is the conjunction of open GCCs that has been studied in [5]. We consider a global constraint whose semantic is $GCC_1(S_1, X) \wedge \cdots \wedge GCC_k(S_k, X) \wedge (S_1, \ldots, S_k$ form a partition of $X)$. In [5] this constraint is modeled with a general network flow. We can use their model together with the relaxed decomposition idea above. $O(kn)$ calls to the GCC algorithm achieve a partial consistency of this global constraint at least as good as the propagation of each individual GCC thanks to the complete filtering for length-lex open GCC.

8 Conclusion

Recently, van Hoeve and Régin described a beautiful filtering algorithm for an open GCC when the set variable has a subset-bound domain. Our first contribution is to present filtering algorithms for the case where the set variable uses a subset-card domain and a length-lex domain. For the length-lex domain, we propose two approaches. The first approach uses the cost-based version of the (closed) constraint and well-chosen weights to map the order of set weights with the length-lex order. It provides a partial

[10] The upper-bound is symmetrical.

filtering algorithm with a linear overhead with respect to the weighted GCC, resulting in a total time complexity of $O(n^2(m+n)\log^2 n)$. The second approach partitions the length-lex interval in a linear number of subset+card intervals on which the filtering algorithm of van Hoeve and Régin is applied, achieving complete filtering in $O(n^2 m)$ time.

Our second significant contribution is to show that these techniques are general and apply to other constraints. The cost-based technique applies to any non-strict constraint for which a cost-based filtering algorithm is available. The partitioning technique is fully generic and is able to lift any subset+card constraint into a length-lex constraint, adding a $O(n^k)$ factor with k sets, to achieve complete filtering. We also present a natural relaxation with an $O(nk)$ time overhead.

Our work also provides, as a fundamental corollary, an upper-bound for the complexity of any filtering algorithm for constraints over length-lex sets, providing a first step towards the implementation of a feature-full length-lex solver. Future work will investigate the exploitation of the inherent structure of constraints in order to decrease the number of sets in the partition or to consider new types of partitions or reductions.

References

1. Francisco Azevedo and Pedro Barahona. Modelling digital circuits problems with set constraints. In *Proceedings of CL2000*, pages 414–428, 2000.
2. Boi Faltings and Santiago Macho-Gonzalez. Open constraint programming. *Artif. Intell.*, 161(1-2):181–208, 2005.
3. C. Gervet and P. van Hentenryck. Length-lex ordering for set csps. In *Proceedings of AAAI 2006*. AAAI Press, 2006.
4. Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
5. W.-J. van Hoeve and J.-C. Régin. Open constraints in a closed world. In *Proceedings of CP-AI-OR 2006*, volume 3990 of *LNCS*, pages 244–257. Springer, 2006.
6. G Kreweras. Sur quelques problemes relatifs au vote pondere, [some problems of weighted voting]. *Math. Sci. Humaines*, 84:45–63, 1983.
7. J.-F Puget. Pecos a high level constraint programming language. In *Proceedings of Spicis 92*, 1992.
8. Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI/IAAI, Vol. 1*, pages 209–215, 1996.
9. Jean-Charles Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3-4):387–405, 2002.