Length-Lex Ordering for Set CSPs

Carmen Gervet* **and Pascal Van Hentenryck**
Brown University, Box 1910, Providence, RI 02912

Abstract

Combinatorial design problems arise in many application areas and are naturally modelled in terms of set variables and constraints. Traditionally, the domain of a set variable is specified by two sets (R,E) and denotes all sets containing R and disjoint from E. This representation has inherent difficulties in handling cardinality and lexicographic constraints so important in combinatorial design. This paper takes a dual view of set variables. It proposes a representation that encodes directly cardinality and lexicographic information, by totally ordering a set domain with a length-lex ordering. The solver can then enforce bound-consistency on all unary constraints in time $\tilde{O}(k)$ where k is the set cardinality. In analogy with finite-domain solvers, non-unary constraints can be viewed as inference rules generating new unary constraints. The resulting set solver achieves a pruning (at least) comparable to the hybrid domain of Sadler and Gervet at a fraction of the computational cost.

Introduction

Combinatorial design problems arise in a variety of applications in coding, sport scheduling, combinatorics, networking and cryptography (Colbourn, Dinitz & Stinson 1999). Many of these problems are NP-hard and are naturally modeled as set CSPs. In other words, they typically feature set-variables of fixed cardinalities and a variety of set-constraints such as inclusion, disjointness, and intersection. Moreover, they are often highly symmetric and hence lexicographic constraints are a natural vehicle to reduce the search space.

Since their inception in constraint programming (Puget 1992; Gervet 1997), set solvers have used a subset-bound representation for domains. A set domain is a pair (R, E) –where R and E are sets– that denotes the set of sets $\{s \mid R \subseteq s \subseteq U \setminus E\}$, where U is the set of all possible elements. It is convenient for implementing inclusion and disjointness constraints, but it faces inherent difficulties in handling cardinality and lexicographic constraints. Indeed, subset-bound domains are not expressive enough to represent cardinality or lexicographic information, which must be represented as constraints. Moreover, these constraints cannot be made bound-consistent and do not prune the search space effectively. The problem is further exacerbated by the inherent difficulty of breaking symmetries in set CSPs as shown in (Sellman & Van Hentenryck 2005). These limitations have been recognized by several researchers. (Azevedo & Barahona 2000) added a cardinality component to subset-domains for their digital design applications and proposed inference rules for pruning the search space using cardinality information. Unfortunately, the only interactions between the cardinality and subset-bound components occur upon instantiation. (Sadler & Gervet 2004) addressed this limitation by proposing a hybrid set domain with three components (SB, C, L), where SB is a subset-bound domain, C is the cardinality information, and L is a co-lexicographic representation of the domain. The consistency of the three domains is maintained by intra-domain constraints. The hybrid domain strengthens constraint propagation in the presence of cardinality information, allows for more effective symmetry-breaking, but has two limitations. First, the intra-domain constraints are complex and computationally expensive as they are polynomial in $n = |U|$ which is typically much greater than the cardinality k of the sets. Second, the lexicographic ordering is only partially integrated with the cardinality information, restricting the potential pruning of cardinality constraints.

Note that 0-1 matrix models in (Frisch et al. 2002; Hnich et al. 2004) can be used to encode the characteristic function of the subset-bound domain and to state cardinality and lexicographic constraints. However, the resulting representation takes $O(n)$ space and is equivalent to subset-bound domains semantically and operationally. Finally, it is important to mention the work of (Hawkins, Lagoon & Stuckey 2005) on explicit domain representation using BDDs. These representations are effective for some classes of set constraints but have difficulties handling cardinality constraints efficiently.

This paper takes a dual view of set variables. It proposes a set domain that directly represents cardinality and lexicographic information, while using constraints to reason about inclusion and disjointness. The key technical idea is to use a length-lex ordering that totally orders the sets first by length and then lexicographically. As a result, arc consistency on cardinality and lexicographic constraints can be enforced in time $O(k)$. Moreover, it is also possible to enforce bound-consistency on unary constraints for inclusion and disjoint-

*Partly funded by the Royal Academy of Engineering.

ness in time $\tilde{O}(k)$, giving an elegant integration of inclusion, disjointness, cardinality, and lexicographic constraints. Finally, in analogy with finite-domain solvers (Van Hentenryck 1989), non-basic constraints can be viewed as inference rules that generate basic constraints that once again interact through the length-lex domain and produce a pruning at least comparable to the hybrid domain.[1] As a result, the length-lex domain enjoys four fundamental advantages besides its simplicity and elegance.

1. The domains take $O(k)$ space and their bounds satisfy all unary constraints unlike subset-bound solvers.
2. The domains directly account for cardinality and lexicographic constraints critical in combinatorial design.
3. Bound-consistency on all traditional unary constraints can be enforced in time $\tilde{O}(k)$.
4. All constraint types prune the domains.

Moreover, all the algorithms presented herein can be adapted to multisets, providing similar benefits for this important structure as well.

The Length-Lex Domain

Notations We assume that sets take their value in a universe U of integers $\{1, \ldots, n\}$. Set variables are denoted by X, Y, Z, possibly subscripted. Elements of U are denoted by the letters e, x and sets are denoted by the letters m, M, s, t. A subset m of U of cardinality k is denoted $\{m_1, m_2, ..., m_k\}$ where $m_1 < m_2 < m_3 ... < m_k$. Therefore, m_j denotes the j-th smallest value in m. We call k-set any set of cardinality k.

Length-lex Ordering The length-lex ordering totally orders sets first by cardinality and then lexicographically.

Definition 1 *A length-lex ordering $\ll$ on sets of integers is defined by:*

$$s \ll t \textbf{ iff } s = \emptyset \vee |s| < |t| \vee$$
$$|s| = |t| \wedge (s_1 < t_1 \vee s_1 = t_1 \wedge s \setminus \{s_1\} \ll t \setminus \{t_1\}).$$

Example 1 *The subsets of* $\{1, 2, 3\}$ *are ordered as*

$$\emptyset \ll \{1\} \ll \{2\} \ll \{3\} \ll \{1, 2\} \ll \{1, 3\} \ll \{2, 3\} \ll \{1, 2, 3\}.$$

A length-lex domain is a pair $\langle m, M\rangle$ that denotes all the sets not smaller than m and not greater than M.

Definition 2 *A length-lex domain is a pair $\langle m, M\rangle$ satisfying $m \ll M$ and denoting the set*

$$\{s \mid m \ll s \ll M\}.$$

Observe that the cardinality of any set s in $\langle m, M\rangle$ satisfies $|m| \leq |s| \leq |M|$. As a consequence, the domain directly captures cardinality information.

Example 2 *Consider a variable X with domain $\langle\{1, 2\}, \{1, 2, 3\}\rangle$. Variable X can take the sets $\{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. If the cardinality of X is constrained to be strictly smaller than 3, its domain can be updated to $\langle\{1, 2\}, \{2, 3\}\rangle$, which denotes all sets of cardinality 2.*

[1] A formal comparison is difficult, since the hybrid model uses a co-lexicographic ordering for technical reasons.

Fundamental Operations on Domains

Once set-variables use the length-lex representation, their domains can be pruned very much like in traditional finite domains. This is a significant innovation for set solvers whose bounds do not usually represent possible (set) values and are not easily amenable to pruning. Example 2 already showed how cardinality constraints can prune a domain. In general, the pruning of a domain $\langle m, M\rangle$ consists of finding the first successor of m and the first predecessor of M satisfying some condition such as the inclusion or exclusion of a set of elements and cardinality restrictions. Set constraints use a number of *first* and *last* functions to compute these successors and predecessors and to prune the domains.

For space reasons, this paper only presents *first* functions and restricts attention to k-sets (which is typically the case in combinatorial design). The algorithms for arbitrary cardinalities can be derived in a similar fashion. The *first* functions share the same overall structure as the algorithm for finding the length-lex successor of a k-set (Kreher & Stinson 1999) (see Algorithm 1). The algorithm assumes that m has a successor and proceeds in two steps: a location phase and a reconstruction phase. The location phase (lines 1–3) determines the index i in the set as the start of the reconstruction. The reconstruction phase determines the new values for $m_i, \ldots, m_k$.

Algorithm 1 $succ_{\ll}(m)$: k-set successor algorithm

```
1: $k \leftarrow |m|,\ m' \leftarrow m,\ i \leftarrow k$;
2: while $(i \geq 0)$ and $(m_i = n - k + i)$ do
3:     $i \leftarrow i - 1$;
4: for $j = i$ to $k$ do
5:     $m'_j \leftarrow (m_i + 1) + j - i$;
6: return $m'$
```

Example 3 *Let $m = \{1, 3, 6, 7\}$ and $U = \{1, ..., 7\}$. The first phase identifies that the successor of m must start its reconstruction phase for $i = 2$, since 6 and 7 cannot be increased. The reconstruction phase fills the positions 2–4 with values $m_i + 1, \ldots, m_i + 3$ to obtain $succ_{\ll}(m) = \{1, 4, 5, 6\}$.*

The successor algorithm runs in $O(k)$ time and all the *first* functions share the same two-phase organization, although the phases are in general more complex. Function first-$r_{\ll}$(m,R) (Algorithm 3) computes the first successor of m that contains all elements in R (or m itself if it contains R). It uses an auxiliary recursive algorithm LR (Algorithm 2) to find the location of the reconstruction phase. For a call $LR(m, R, i, p)$, $R_1, \ldots, R_{p-1}$ are present in $m_1, \ldots, m_{i-1}$ and the function must determine if there exists a location $j \geq i$ for the reconstruction phase that can accommodate $R_p, \ldots, R_{|R|}$. When $p > |R|$, m contains R and no reconstruction is necessary (lines 1–2). If there is no room left for $R_p, \ldots, R_{|R|}$ or $R_p < m_i$, the reconstruction phase must start earlier than i (lines 3–4). If $m_i < R_p$, then the function is called recursively with $i + 1$ to determine if a location greater than i can be found (lines 6–8). If no such location exists, the function returns (i, p), since m_i can be increased (line 9). Finally, if $m_i = R_p$, LR is called recursively with

$i+1$ and $p+1$ since R_p is in m (line 12). This recursive call may fail, in which case a location smaller than i must be found since we cannot increase m_i without losing R_p. Once the location i is found, algorithm first-$r_{\ll}$(m,R) starts the reconstruction from i like the successor algorithm 1. However, it must make sure to include all elements of R. This explains the loop conditions on line 6 and the insertion of the remaining elements in R in lines 11–13. The function runs in time $O(k \log |R|)$, which is $\tilde{O}(k)$.

Algorithm 2 $LR(m, R, i, p)$

```
1:  if $p > |R|$ then
2:      return $(k+1, p)$;
3:  else if $k - i < |R| - p \vee m_i > R_p$ then
4:      return $(\perp, p)$;
5:  else if $m_i < R_p$ then
6:      $(i', p') \leftarrow LR(m, R, i+1, p)$;
7:      if $i' \neq \perp$ then
8:          return $(i', p')$;
9:      else
10:         return $(i, p)$;
11: else
12:     return $LR(m, R, i+1, p+1)$;
```

Algorithm 3 first-$r_{\ll}$(m,R)

```
1:  $(i, p) \leftarrow LR(m, R, 1, 1)$;
2:  if $i = \perp$ then
3:      $m' \leftarrow \perp$;
4:  else
5:      $v \leftarrow m_i + 1$;
6:      while $k - i > |R| - p \wedge i \leq k$ do
7:          $m'_i \leftarrow v$;
8:          if $m'_i = R_p$ then
9:              $p \leftarrow p + 1$
10:         $i \leftarrow i + 1, v \leftarrow v + 1$;
11:     while $i \leq k$ do
12:         $m'_i \leftarrow R_p$;
13:         $i \leftarrow j + 1, \ p \leftarrow p + 1$;
14: return $m'$
```

Example 4 *Consider $m = \{1, 3, 6, 7\}, R = \{3, 4\}$, and $U = \{1, ..., 7\}$. The call $LR(m, R, 3, 2)$ fails because $m_3 > R_2$, which implies that $LR(m, R, 2, 1)$ also fails (since $m_2 = R_1$). Since $m_1 < R_1$, $LR(m, R, 1, 1)$ returns $(1, 1)$. The reconstruction is simple in this case since all elements of R are among the smallest ones and the algorithm returns $\{2, 3, 4, 5\}$.*

Function first-$e_{\ll}$(m,E) (Algorithm 5) computes the first successor of m that does not include any element in E (or m itself if it is disjoint from E). It uses an auxiliary recursive algorithm LE (algorithm 4) to find the location and a function denoting "availability"

$$av(v) = \{e \in U \setminus E \mid e > v\}$$

Intuitively, $av(v)$ represents the set of elements in U greater than v and not in E. We will show later on how to implement the algorithms without this function that depends on U. Observe first that the location cannot be after the smallest index i such that $m_i \in E$. However, it may not be possible to start at i if there are not sufficiently many elements available in $av(m_i)$ to fill positions $i, ..., k$. To find out the location with greatest index, the algorithm proceeds recursively from left to right. If $i = k+1$, m is disjoint from E and no reconstruction is necessary. Otherwise, if $m_i \notin E$, the algorithm is called recursively with $i+1$ (line 4 in Algorithm 4). If it succeeds and returns $j \neq \perp$, j is the desired location. Otherwise, if $m_i \in E$ or the recursive call fails, the algorithm determines if there are enough elements to start at location i, which is the purpose of line 7. The reconstruction phase in Algorithm 5 fills the remaining positions with the smallest elements in $av(m_i)$. It remains to show how to implement function av efficiently. Observe first that

$$|av(m_i)| = n - m_i - |\{e \in E | e > m_i\}|.$$

If E is represented as a sorted array, it suffices to compute the index of the largest element $e \in E$ smaller or equal to m_i giving us $|\{e \in E | e > m_i\}|$ in $O(\log |E|)$ time. The reconstruction can implement lines 4–5 by starting Algorithm 5 from $m_i + 1$ and inserting elements $m_i + 2, ...$ whenever they are not in E. The overall complexity is $O(k + |E| \log |E|)$ (including the sorting) which is $\tilde{O}(k)$ when $|E|$ is $O(k)$.

Algorithm 4 $LE(m, E, i)$

```
1:  if $i = k + 1$ then
2:      return i;
3:  if $m_i \notin E$ then
4:      $j \leftarrow LE(m, E, i+1)$;
5:      if $j \neq \perp$ then
6:          return j;
7:  if $|av(m_i)| \geq k - i + 1$ then
8:      return $i$;
9:  else
10:     return $\perp$;
```

Algorithm 5 first-$e_{\ll}$(m,E)

```
1: $m' \leftarrow m$;
2: $i \leftarrow LE(m, E, 1)$;
3: if $i \neq \perp$ then
4:     for $j = i$ to $k$ do
5:         $m'_j \leftarrow av(m_i)_{j-i+1}$;
6:     return $m'$;
7: else
8:     return $\perp$;
```

Example 5 *Consider $m = \{1, 7, 8\}$, $U = \{1, ..., 8\}$, and $E = \{3, 5, 7\}$. The call to LE with $i = 2$ fails as only 8 is available to fill positions $i = 2, i = 3$. The call to LE with $i = 1$ succeeds as $av(1) = \{2, 4, 6, 8\}$ and there are only three positions to fill. The reconstruction phase gives $\{2, 4, 6\}$.*

Finally, we also show the algorithm for computing the successor to m of cardinality c, when the cardinality is not fixed. There is no location phase here and the reconstruction is simple.

Algorithm 6 first-$c_{\ll}(m, c)$

```
1: $m' \leftarrow m$
2: if $|m| = c$ then
3:     return m;
4: else if $|m| < c \land |M| \geq c$ then
5:     for $j = 1$ to $c$ do
6:         $m'_j \leftarrow j$;
7:     return $m'$
8: else
9:     return $\perp$;
```

The Set Solver

A set solver with length-lex domains may follow the architecture of fd-solvers (Van Hentenryck & Deville 1991). The solver is organized around the domain store and includes basic and non-basic constraints. The set of basic constraints is $\{s \subseteq X, s \oplus X, |X| \leq d, |X| \geq c, X \preceq Y\}$ where $\oplus$ denotes disjointness and $\preceq$ the length-lex lexicographic constraint. The set-solver maintains at least bound-consistency on basic constraints. The non-basic constraints use the domain store to generate new basic constraints that tighten the constraint store. The rest of the paper defines (a subset of) the operational semantics of the set-solver using a structural operational semantic (SOS) style. The SOS semantic manipulates configurations $\langle \gamma, \sigma \rangle$, where γ is a conjunction of constraints and σ is the domain store (i.e., a conjunction of domain constraints). The semantic is specified in terms of rewriting rules of the form

$$\frac{\text{Conditions}}{\langle \gamma, \sigma \rangle \longmapsto \langle \gamma', \sigma' \rangle}$$

that specifies that $\langle \gamma, \sigma \rangle$ can be rewritten into $\langle \gamma', \sigma' \rangle$ when the conditions hold. The SOS semantics is specified in terms of the reflexive and transitive closure of the transition relation, which is denoted by $\overset{\star}{\longmapsto}$. In other words, given a configuration $\langle \gamma, \sigma \rangle$, the set-solver returns a configuration $\langle \gamma^\star, \sigma^\star \rangle$ such that $\langle \gamma, \sigma \rangle \overset{\star}{\longmapsto} \langle \gamma^\star, \sigma^\star \rangle$.

The following rules specify the semantics of conjunction in the constraint and domain stores.

$$\frac{\langle \gamma_1, \sigma \rangle \longmapsto \sigma'}{\langle \gamma_1 \land \gamma_2, \sigma \rangle \longmapsto \langle \gamma_2, \sigma' \rangle} \qquad \frac{\langle \gamma_1, \sigma \rangle \longmapsto \langle \gamma_1', \sigma' \rangle}{\langle \gamma_1 \land \gamma_2, \sigma \rangle \longmapsto \langle \gamma_1' \land \gamma_2, \sigma' \rangle}$$

$$\frac{\langle \gamma, \sigma_1 \rangle \longmapsto \langle \gamma', \sigma_1' \rangle}{\langle \gamma, \sigma_1 \land \sigma_2 \rangle \longmapsto \langle \gamma', \sigma_1' \land \sigma_2 \rangle} \qquad \frac{\sigma \longmapsto \sigma'}{\langle \gamma, \sigma \rangle \longmapsto \langle \gamma, \sigma' \rangle}$$

The domain store must be consistent or the solver fails.

$$\frac{\neg(m \ll M)}{X \in \langle m, M \rangle \longmapsto \perp}$$

Basic Constraints

The semantics of the basic constraints is as follows.

Inclusion A constraint $s \subseteq X$ reduces the domain $\langle m, M \rangle$ of X by taking the first successor of m containing s and the first predecessor of M that contains s. Note that $s \subseteq m'$ and $s \subseteq M'$ and the constraint is bound-consistent wrt the new domain.

$$\frac{m' = \text{first-}r_{\ll}(m, s), \;\; M' = \text{last-}r_{\ll}(M, s)}{\big\langle s \subseteq X, X \in \langle m, M \rangle \big\rangle \longmapsto \big\langle s \subseteq X, X \in \langle m', M' \rangle \big\rangle}$$

Disjointness A constraint $s \oplus X$ is similar, uses first-e and last-e, and is bound-consistent.

$$\frac{m' = \text{first-}e_{\ll}(m, s), \;\; M' = \text{last-}e_{\ll}(M, s)}{\big\langle s \oplus X, X \in \langle m, M \rangle \big\rangle \longmapsto \big\langle s \oplus X, X \in \langle m', M' \rangle \big\rangle}$$

Cardinality The cardinality constraints prune the domain once and are completely solved. This is a real strength of the length-lex domain.

$$\frac{m' = \text{first-}c_{\ll}(m, c)}{\big\langle |X| \geq c, X \in \langle m, M \rangle \big\rangle \longmapsto \big\langle \emptyset, X \in \langle m', M \rangle \big\rangle}$$

$$\frac{M' = \text{last-}c_{\ll}(M, d)}{\big\langle |X| \leq d, X \in \langle m, M \rangle \big\rangle \longmapsto \big\langle \emptyset, X \in \langle m, M' \rangle \big\rangle}$$

Lexicographic Symmetries can be broken by imposing length-lex lexicographic constraints. These constraints are arc-consitent and represent another great strength of the domain.

$$\frac{m_Y' = \max(m_X, m_Y), \;\; M_X' = \min(M_Y, M_X)}{\begin{array}{l}\big\langle X \preceq Y, \{X \in \langle m_X, M_X \rangle, Y \in \langle m_Y, M_Y \rangle\} \big\rangle \longmapsto \\ \big\langle X \preceq Y, \{X \in \langle m_X, M_X' \rangle, Y \in \langle m_Y', M_Y \rangle\} \big\rangle\end{array}}$$

Note that the min and max are taken on the length-lex ordering. We are in position to present the main result on basic constraints. Observe that all constraints are monotone and contractant, so that there is a unique fixpoint when applying these rules.

Theorem 1 (Bound Consistency) *Let $\langle \gamma, \sigma \rangle$ be a configuration and $\langle \gamma^\star, \sigma^\star \rangle$ be the configuration such that $\langle \gamma, \sigma \rangle \overset{\star}{\longmapsto} \langle \gamma^\star, \sigma^\star \rangle$. Then, the basic constraints in $\gamma^\star$ are bound-consistent wrt $\sigma^\star$ and $\sigma^\star$ does not contain any unary cardinality constraints.*

Note also that each transition rule takes time $\tilde{O}(k)$.

Required and Possible Elements

It is interesting to discuss the required and excluded elements in a configuration $\langle \gamma, \sigma \rangle$. Some of the required elements can be deduced from the unary inclusion constraints in γ. However, the lexicographic, cardinality, and disjointness constraints may have reduced the domain, implying that some elements are now required. These required elements may be computed in time $O(k)$ and highlight another strength of the representation. We now specify inductively the set *req*(X,$\langle \gamma, \sigma \rangle$) of required elements of a set-variable X in $\langle \gamma, \sigma \rangle$.

$$\begin{array}{lcl} req(X, \langle \gamma, \sigma \rangle) & = & req(X, \gamma) \cup req(X, \sigma); \\ req(X, \gamma_1 \wedge \gamma_2) & = & req(X, \gamma_1) \cup req(X, \gamma_2); \\ req(X, \sigma_1 \wedge \sigma_2) & = & req(X, \sigma_1) \cup req(X, \sigma_2); \\ req(X, s \in X) & = & s; \\ req(X, X \subseteq \langle m, M \rangle) & = & \mathcal{R}(\langle m, M \rangle); \\ req(X, \bullet) & = & \emptyset \quad \text{otherwise.} \end{array}$$

It remains to show how to compute the required elements in a domain. The following rules can be used to compute $\mathcal{R}(D)$ where

$$D = \langle \{m_1, \ldots, m_i\}, \{M_1, \ldots, M_i\} \rangle \quad (i \leq k).$$

If $M_1 > m_1 + 1$, then

$$\mathcal{R}(D) = \emptyset.$$

If $M_1 = m_1 + 1 \wedge m_1 + i < n$, then

$$\mathcal{R}(D) = \emptyset.$$

If $M_1 = m_1 + 1 \wedge m_1 + i = n$, then

$$\mathcal{R}(D) = \{m_j \mid m_j = n - (i - j) \;\&\; 1 \leq j \leq i\}. \tag{1}$$

Otherwise, if $M_1 = m_1$, then

$$\mathcal{R}(D) = \{m_1\} \cup \mathcal{R}(\langle \{m_2, \ldots, m_i\}, \{M_2, \ldots, M_i\} \rangle). \tag{2}$$

Example 6 *Consider* $\langle \{1, 2, 4, 5\}, \{2, 3, 4, 5\} \rangle$ *and* $U = \{1, 2, 3, 4, 5\}$. *From (1),* $\{4, 5\}$ *is required. Finally, consider* $\langle \{1, 3, 4, 6, 7\}, \{1, 4, 5, 6, 7\} \rangle$ *and* $U = \{1, ..., 7\}$. *From (2) and (1),* $\{1, 6, 7\}$ *is required.*

It is also possible to compute the set $\mathcal{P}(D)$ of elements in U that may belong to a domain. More precisely, $\mathcal{P}(D)$ can be defined inductively as follows. Let

$$D = \langle \{m_1, \ldots, m_i\}, \{M_1, \ldots, M_i\} \rangle \quad (i \leq k).$$

Observe first that $\mathcal{P}(D)$ can never include any element smaller than m_1. If $M_1 > m_1 + 1$,

$$\mathcal{P}(D) = \{m_1, .., n\}$$

as all the elements of U belong to some sets starting with $m_1 + 1$. If $M_1 = m_1 + 1$, $\mathcal{P}(D)$ is the union of

$$\mathcal{P}(\langle \{m_1, \ldots, m_i\}, \{m_1, n - i + 2, \ldots, n\} \rangle), \tag{3}$$

i.e., the elements in the sets starting with m_1, and

$$\mathcal{P}(\langle \{m_1 + 1, \ldots, m_1 + i\}, \{M_1, \ldots, M_i\} \rangle) \tag{4}$$

i.e., the elements in the sets starting with $m_1 + 1$. Case 3 produces the sets

$$\{m_1\} \cup \{m_2, \ldots, n\}$$

while case 4 gives the element

$$\begin{array}{ll} \{m_1..M_i\} & \text{if } M_{i-1} = m_1 + i + 1 \\ \{m_1..n\} & \text{if } M_{i-1} > m_1 + i + 1. \end{array}$$

Finally, if $M_1 = m_1$,

$$\mathcal{P}(D) = \{m_1\} \cup \mathcal{P}(\langle \{m_2, \ldots, m_i\}, \{M_2, \ldots, M_i\} \rangle).$$

Observe that the above derivation generates at most $O(k)$ intervals and it is possible to design an algorithm computing $\mathcal{P}(D)$ running in $O(k)$ space and $O(k \log k)$ time (and hence in $\tilde{O}(k)$ time by sorting the intervals).

Non-Basic Constraints

Non-basic constraints can be defined as inference rules that generate basic constraints using the domain store. Space requirements prevent us from discussing all traditional constraints and we focus on the binary disjointness constraints. The goal of the section is to demonstrate that pruning rules for inclusion, disjointness, and cardinality constraints synergically cooperate in reducing the domains efficiently.

Binary Disjointness The disjointness constraint $X \oplus Y$ imposes that required elements in X cannot appear in Y and vice-versa. The following transition rule implements the traditional subset-bound pruning and generates the implied unary constraints in $O(k)$ time.

$$\frac{\begin{array}{c} \gamma_X \equiv \text{req}(Y, \langle \gamma, \sigma \rangle) \oplus X \\ \gamma_Y \equiv \text{req}(X, \langle \gamma, \sigma \rangle) \oplus Y \end{array}}{\Big\langle X \oplus Y \;\wedge\; \gamma, \sigma \Big\rangle \longmapsto \Big\langle \langle X \oplus Y \;\wedge\; \gamma_X \;\wedge\; \gamma_Y \;\wedge\; \gamma \rangle, \sigma \Big\rangle} \tag{5}$$

The disjointness constraint also implies some cardinality restrictions that prune the length-lex domains actively. Inference rules about cardinalities have been published in (Azevedo & Barahona 2000) for many constraints. In particular, the disjointness constraint implies $|X| + |Y| \leq p$, where p represents the number of values that may belong to X or Y.

$$\frac{\begin{array}{l} \sigma \Rightarrow X \in \langle m_X, M_X \rangle \;\wedge\; Y \in \langle m_Y, M_Y \rangle \\ p = |\mathcal{P}(\langle m_X, M_X \rangle) \cup \mathcal{P}(\langle m_Y, M_Y \rangle)| \\ \gamma_X \equiv |X| \leq p - |m_Y| \\ \gamma_Y \equiv |Y| \leq p - |m_X| \end{array}}{\Big\langle X \oplus Y, \sigma \Big\rangle \longmapsto \Big\langle \langle X \oplus Y \wedge \gamma_X \wedge \gamma_Y \rangle, \sigma \Big\rangle} \tag{6}$$

The inferred cardinality constraints can be generated in time $\tilde{O}(k)$ and may prune the length-lex domains, illustrating the synergy between the subset-bound and cardinality inferences.

Example 7 *Consider the disjointness constraint* $X \oplus Y$ *and the domain store*

$$\Big\langle X \in \langle \{1, 2\}, \{1, 2, 3\} \rangle \;\wedge\; Y \in \langle \{1, 2, 3\}, \{2, 3, 4, 5\} \rangle \Big\rangle$$

Rule 5 used in subset-bound solvers does not make any deduction since there are no required elements in either variable. Rule 6 produces the new domain store

$$X \in \langle \{1, 2\}, \{2, 3\} \rangle \;\wedge\; Y \in \langle \{1, 2, 3\}, \{3, 4, 5\} \rangle$$

illustrating the pruning from cardinality constraints.

Global Constraints

The strength of the length-lex domain in pruning the domains also benefits global constraints. Once again, space restrictions do not allow us to discuss this topic in depth. However, the section illustrates the significant domain reductions that global constraints and length-lex domains can jointly produce. It uses a global constraint $disjoint_{\preceq}(X_{1..q})$ combining disjointness and lexicographic constraints. Its semantics, when the X_i's are k-sets, is specified by

$$\begin{array}{l} disjoint_{\preceq}(X_{1..q}) \equiv \\ \quad \forall 1 \le i < j \le q : X_i \oplus X_j \ \wedge \ X_i \preceq X_j. \end{array}$$

Assume that the implementation first generates all the binary disjointness and lexicographic constraints for simplicity. We will present two additional pruning rules that exploit the fact that the sets are both disjoint and lexicographically ordered and dramatically improve the pruning of subset-bound solvers. The first rule is simple and removes the smallest element of X_i from X_{i+1}.

$$\frac{\begin{array}{l} \sigma \Rightarrow X_i \in \langle m, M\rangle \quad (1 \le i < q) \\ \gamma \equiv \{m_1\} \oplus X_{i+1} \end{array}}{\Big\langle disjoint_{\preceq}(X_{1..q}), \sigma\} \Big\rangle \longmapsto \Big\langle \{disjoint_{\preceq}(X_{1..q}) \ \wedge \ \gamma, \sigma \Big\rangle}$$

The second rule reduces the upper bound of the domains and ensures that sufficiently many elements are left for subsequent sets to be both lexicographically smaller and disjoint.

$$\frac{\begin{array}{l} 1 \le i < j \le q \\ \sigma \Rightarrow X_j \in D \\ v = max(\mathcal{P}(D)) - k(j - i + 1) + 1 \\ \gamma \equiv X_i \preceq \{v, n-k+2, \ldots, n\} \end{array}}{\Big\langle disjoint_{\preceq}(X_{1..q}), \sigma\} \Big\rangle \longmapsto \Big\langle \{disjoint_{\preceq}(X_{1..q}) \ \wedge \ \gamma, \sigma \Big\rangle}$$

The global disjoint and partition constraints over k-sets have been addressed by *subset bound solvers* using global propagators that run in $\mathcal{O}(qn^2)$ (Bessière & al. 2004; Ilog Solver 1998). The following example contrasts their respective pruning.

Example 8 *Let $U = \{1, \ldots, 10\}$ and $X_{1..3}$ have initial domains $\langle\{1,2,3\},\{8,9,10\}\rangle$. $disjoint_{\preceq}(X_{1..3})$ with the above rules produces the new domains*

$$\begin{array}{l} X_1 \in \langle\{1,2,3\},\{2,9,10\}\rangle \\ X_2 \in \langle\{2,3,4\},\{5,9,10\}\rangle \\ X_3 \in \langle\{3,4,5\},\{8,9,10\}\rangle. \end{array}$$

Observe that X_1 has only 64 sets in its domain. In contrast, a subset-bound solver returns the domain

$$\begin{array}{l} X_1 \in [\emptyset, \{1,..,10\}] \\ X_2 \in [\emptyset, \{2,..,10\}] \\ X_3 \in [\emptyset, \{3,..,10\}] \end{array}$$

and X_1 has 2^{10} sets in its domain. These global propagators do not prune here (they only prune when sub-partitions are discovered) and the only reduction comes from the lexicographic constraints.

This strength of the propagation in length-lex domains is derived from two fundamental properties: (1) the domain and the lexicographic constraints use the same ordering; 2) the cardinality is intrinsic to the domain ordering and precedes the lexicographic ordering.

It is also significant to mention that the length-lex ordering generalizes to multi-sets. The algorithms for finding successors and predecessors saisfying a condition can be slightly modified to account for the fact that a value may occur multiple times.

References

Azevedo, F., Barahona, P. 2000. Modelling Digital Circuits Problems with Set Constraints. in *CL*-2000.

Barnier, N., Brisset, P. 2001. Solving the Kirkman's Schoolgirl Problem in a Few Seconds. In *CP*-2001.

Bessière, C., Hnich, B., Hébrard, E., Walsh, T. 2004. Disjoint, Partition and Intersection Constraints for Sets and Multiset Variables. In *CP*-2004.

Colbourn, C. J. , Dinitz, J.H., Stinson. 1999. Applications of Combinatorial Designs to Communications, Cryptography, and Networking. In *Surveys in Combinatorics, London Mathematical Society Lecture Note Series 187*, Cambridge University Press.

Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T. 2002. Global Constraints for Lexicographic Ordering. In *CP*-2002.

Gervet, C. 1997. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. In *Constraints journal*, volume 1(3). Kluwer.

Hawkins, P., Lagoon, V., Stuckey, P. 2005. Solving Set Constraint Satisfaction Problems using ROBDDs. *JAIR Journal*, 24.

Hnich, B., Kiziltan Z., Walsh, T. Combining Symmetry Breaking with Other Constraints: lexicographic ordering with sums. In *Proc. of Int. Symposium on AI & Maths*-2004.

Ilog Solver 4.4, Reference Manual, Ilog SA, Gentilly, France. 1998.

Kreher, D.L., Stinson, D.R. 1999. *Combinatorial Algorithms*. The CRC Press Series on Discrete Mathematics and its Applications.

Puget,J-F. 1992 PECOS a High Level Constraint Programming Language In *Proc. of Spicis*.

Sadler, A., Gervet, C. 2004. Hybrid Set Domains to Strengthen Constraint Propagation and Reduce Symmetries. In *CP*-2004.

Sellman, M., Van Hentenryck, P. 2005. Structural Symmetry Breaking. In *IJCAI*-2005.

Van Hentenryck, P. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

Van Hentenryck, P., Deville, Y. *Operational Semantics of Constraint Logic Programming over Finite Domains*. in *PLILP*-2001.