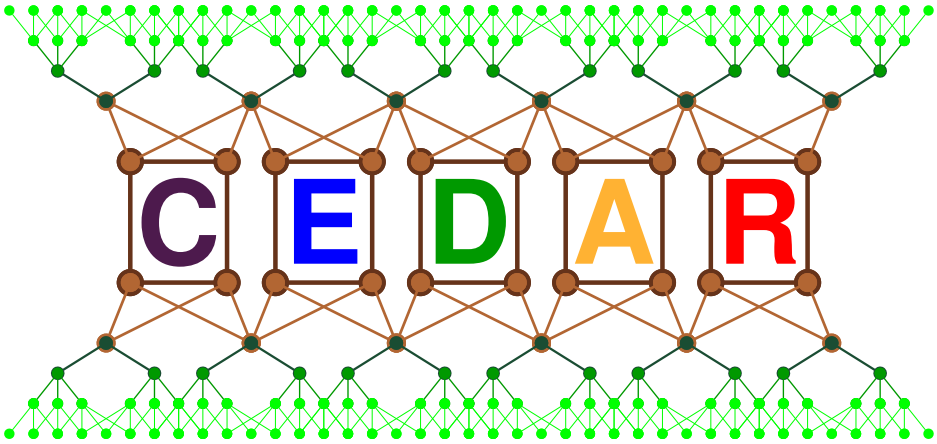


Technical Report Number 4

Efficient Encoding of Very Large Partial Orders

A Specification

Hassan Aït-Kaci

September 2013

Publication Note

Author's address:

LIRIS - UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)4 27 46 57 08
Email: hassan.ait-kaci@univ-lyon1.fr

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL).

CEDAR **Technical Report Number 4**

Efficient Encoding of Very Large Partial Orders
A Specification

Hassan Aït-Kaci
hassan.ait-kaci@univ-lyon1.fr

September 2013

Abstract

This document defines a data structure and lattice operations thereon for representing bit vectors of very large size that saves space while retaining efficient operations.

Keywords: Transitive closure, partial-order encoding, lattice operations

Résumé

Ce document définit une structure de données dotée d'opérations de treillis pour représenter des vecteurs de bits de très grand taille qui utilise moins d'espace tout en conservant des opérations efficaces.

Mots-Clés: Fermeture transitive, encodage d'ordre partiel, operations de treillis

Table of Contents

1 Introduction

In [3], a method is presented for encoding elements of a partially ordered set based on transitive closure. The data structure it relies on is that of a binary word—*i.e.*, a bit vector. With such a structure, all boolean operations—*and*, *or*, *not*—are thus very efficient. This representation also eases computation of the transitive closure since setting a bit on or off is trivially accommodated.

However, while this representation is convenient and time-efficient for relatively small posets of the order of a few hundred elements, it quickly becomes space-inefficient for large posets of hundreds of thousands, or millions of elements.

This document defines an alternative representation of indexed bit sets that offers the advantage of being more compact than bit vectors while retaining time-efficient boolean and bit-setting operations.

In Section 2, the basic data structure is defined. In Section 3, bit setting and unsetting operations are defined. In Section 4, the three boolean operations—conjunction, disjunction, and negation—are defined. In Section 5, some implementation considerations are discussed.

2 Compact Bit Codes

The ideas is intuitively simple. It consists of representing a bit vector as a finite array of k ($k \in \mathbb{N}$) pairs of integer indices $\langle l_i, u_i \rangle$, for $i = 0, \ldots, k-1$, such that, for all indices $i = 0, \ldots, k-2$:

$$0 \leq l_i < u_i < l_{i+1} < u_{k-1}. \tag{1}$$

We shall refer to such a sequence $\{\ \langle l_i, u_i \rangle \mid i = 0, \ldots, k-1\ \}$ of k pairs, where $k \in \mathbb{N}$, as a *compact code*. For $k = 0$, this is written as the empty sequence $\{\}$.

Given a compact code representation of a bit vector $\mathtt{V}$, each pair $\langle l, u \rangle$ represents a *maximal* contiguous sequence of 1's (hereafter referred to as a "*packet*") in $\mathtt{V}$. Thus, the i-th packet of a bit vector is represented as the pair of indices $\langle l_i, u_i \rangle$ such that l_i is the index of the lowest bit in the packet, and u_i is the index of the first $\mathtt{0}$-bit following the packet.

For example, the compact code of the bit vector $\mathtt{0011111001111000000110000}$ corresponds to the sequence of packet pairs:[1] $\{\ \langle 4, 6\rangle, \langle 12, 16\rangle, \langle 18, 23\rangle\ \}$.

The empty bit vector (containing all 0's) is represented as the empty sequence $\{\}$. The *length* of a bit vector represented by a compact code sequence of k pairs (or packets) is u_k. The *size* of a compact code C of k pairs (or packets) is k (*i.e.*, its number of packets).

[1] Recall that a bit vector is written with its lowest bit to the right.

3 Bit Operations

Let $C = \{ \langle l_i, u_i \rangle \mid i = 0, \ldots, k \}$ be a compact code of k packets ($k > 0$). Given a number $n \in \mathbb{N}$, and a compact code C of k packets as defined above, we say that:[2]

- *n is within a packet of C* iff $\exists i \in [0, k-1]$ such that $l_i \leq n < u_i$—in which case we shall write $C.\mathbf{packet}(n) = i$;
- *n is between packets of C* iff either one of the three statements holds:
  1. $n < l_0$; or,
  2. $u_{k-1} \leq n$; or,
  3. $\exists i \in [0, k-2]$ such that $u_i \leq n < l_{i+1}$.

If a number n is between packets of a compact code C of size k, we define two functions $C.\mathbf{prev}(n)$ and $C.\mathbf{next}(n)$ for each of the three possible respective cases above as follows (where the symbol '`?`' means "undefined"):

1. $C.\mathbf{prev}(n) \stackrel{\text{def}}{=}$ `?` and $C.\mathbf{next}(n) \stackrel{\text{def}}{=} l_0$;
2. $C.\mathbf{prev}(n) \stackrel{\text{def}}{=} u_k$ and $C.\mathbf{next}(n) \stackrel{\text{def}}{=}$ `?`;
3. $C.\mathbf{prev}(n) \stackrel{\text{def}}{=} u_i$ and $C.\mathbf{next}(n) \stackrel{\text{def}}{=} l_{i+1}$.

For such a number n, we say that:

- n is left-adjacent in C if $n = C.\mathbf{next}(n) - 1$;
- n is right-adjacent in C if $n = C.\mathbf{prev}(n)$;
- n is adjacent in C if it is both left-adjacent and right-adjacent in C.

Note that if n is between packets and adjacent, this necessarily means that the two packets on each side are only separated by a single `0`-bit (the bit in position n in the denoted bit vector).

N.B.: In all the compact code expressions to follow, we use the implicit convention that a packet with undefinable bounds is simply omitted. Thus, we will always use the notation $\{ \langle l_0, u_0 \rangle, \ldots, \langle l_{k-1}, u_{k-1} \rangle \}$ to denote a compact code, where $k \geq 0$ up to the above conventions regardless of the actual number of packets. For example, for $k = 0$ this will correspond to the empty code $\{\}$, and for $k = 1$, this will correspond to the single-packet code $\{ \langle l_0, u_0 \rangle \}$.

We define the following bit-setting operations on C. These methods operate "in place" by modifying a code C that invokes them.

- $C.\mathbf{set}(n)$, for $n \in \mathbb{N}$, which sets the n-th bit of the bit vector denoted by C to 1.
- $C.\mathbf{set}(n, m)$, for $n, m \in \mathbb{N}, n < m$, which sets to 1 all the bits from position n (*inclusive*) to position m (*exclusive*) of the bit vector denoted by C.

[2] In what follows, we shall use the "dot" notation of object-oriented methods to denote all functions or operations on codes.

- $C.\mathbf{unset}(n)$, for $n \in \mathbb{N}$, which sets the n-th bit of the bit vector denoted by C to 0.
- $C.\mathbf{unset}(n, m)$, for $n, m \in \mathbb{N}, n < m$, which sets to 0 all the bits from position n (*inclusive*) to position m (*exclusive*) of the bit vector denoted by C.

For $m \leq n$, both $C.\mathbf{set}(n, m)$ and $C.\mathbf{unset}(n, m)$ are no_ops—*i.e.*, they leave C unchanged. Since $\mathbf{set}(n)$ is equivalent to $\mathbf{set}(n, n+1)$, we will just give the methods for $\mathbf{set}(n, m)$ and similarly for $\mathbf{unset}(n)$.

3.1 Bit setting

There are four cases to consider for which performing $C.\mathbf{set}(n, m)$ modifies C as follows.

1. If n is within a packet in C (say, $C.\mathbf{packet}(n) = i$) and m is within a packet in C (say, $C.\mathbf{packet}(m) = j$), then, if $i = j$, $C.\mathbf{set}(n, m)$ leaves C unchanged. Otherwise (if $i < j$),[3] then C becomes:

$$\{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \}. \tag{2}$$

2. If n is within a packet in C (say, $C.\mathbf{packet}(n) = i$) and m is between packets in C, then C becomes:

$$\begin{cases} \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } m \text{ is left-adjacent in } C \text{ and } C.\mathbf{next}(m) = l_j; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, m\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{otherwise}. \end{cases} \tag{3}$$

3. If n is between packets in C and m is within a packet in C (say, $C.\mathbf{packet}(m) = j$), then C becomes:

$$\begin{cases} \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n \text{ is right-adjacent in } C \text{ and } C.\mathbf{prev}(n) = u_i; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle n, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{otherwise}. \end{cases} \tag{4}$$

[3] *N.B.*: By Condition (1) and since $n < m$, then $i \neq j$ implies necessarily that $i < j$.

4. If both n and m are between packets in C, then C becomes:

$$\begin{cases} \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n \text{ is right-adjacent in } C \text{ and } C.\mathbf{prev}(n) = u_i, \text{ and} \\ \qquad \text{if } m \text{ is left-adjacent in } C \text{ and } C.\mathbf{next}(m) = l_j; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, m\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n \text{ is right-adjacent in } C \text{ and } C.\mathbf{prev}(n) = u_i, \text{ and} \\ \qquad \text{if } m \text{ is not left-adjacent in } C; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle n, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n \text{ is not right-adjacent in } C, \text{ and} \\ \qquad \text{if } m \text{ is left-adjacent in } C \text{ and } C.\mathbf{next}(m) = l_j; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle n, m\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{otherwise.} \end{cases} \tag{5}$$

3.2 Bit unsetting

Here again, there are four cases to consider for which performing $C.\mathbf{unset}(n, m)$ modifies C as follows.

1. If both n and m are between packets in C: if $C.\mathbf{prev}(n) = C.\mathbf{prev}(m)$ (or, equivalently, if $C.\mathbf{next}(n) = C.\mathbf{next}(m)$), then $C.\mathbf{unset}(n, m)$ leaves C unchanged; otherwise, C becomes:

$$\{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, C.\mathbf{prev}(n)\rangle, \langle C.\mathbf{next}(m), u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \}. \tag{6}$$

2. If n is within a packet in C (say, $C.\mathbf{packet}(n) = i$) and m is between packets in C, then C becomes:

$$\begin{cases} \{\ \langle l_0, u_0\rangle, \ldots, \langle l_{i-1}, u_{i-1}\rangle, \langle C.\mathbf{next}(m), u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n = l_i, \text{ where } C.\mathbf{next}(m) = l_j; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, n\rangle, \langle C.\mathbf{next}(m), u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{otherwise (\textit{i.e.}, if } n > l_i), \text{ where } C.\mathbf{next}(m) = l_j. \end{cases} \tag{7}$$

3. If n is between packets in C and m is within a packet in C (say, $C.\mathbf{packet}(m) = j$), then C becomes:

$$\begin{cases} \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, C.\mathbf{prev}(n)\rangle, \langle l_{j+1}, u_{j+1}\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } m = u_j - 1, \text{ where } C.\mathbf{prev}(m) = u_i; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, C.\mathbf{prev}(n)\rangle, \langle m, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{otherwise (\textit{i.e.}, if } m < u_j - 1), \text{ where } C.\mathbf{prev}(m) = u_i. \end{cases} \tag{8}$$

4. If both n is within a packet (say, $C.\mathbf{packet}(n) = i$), and m is within a packet (say, $C.\mathbf{packet}(m) = i$) in C, then C becomes:

$$\begin{cases} \{\ \langle l_0, u_0\rangle, \ldots, \langle l_{i-1}, u_{i-1}\rangle, \langle l_{j+1}, u_{j-1}\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n = l_i, \text{ and} \\ \qquad \text{if } m = u_{j-1}; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_{i-1}, u_{i-1}\rangle, \langle m, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n = l_i, \text{ and} \\ \qquad \text{if } m < u_{j-1}; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, n\rangle, \langle l_{j+1}, u_{j-1}\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n > l_i, \text{ and} \\ \qquad \text{if } m = u_{j-1}; \\ \{\ \langle l_0, u_0\rangle, \ldots, \langle l_i, n\rangle, \langle m, u_j\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \} \\ \qquad \text{if } n > l_i, \text{ and} \\ \qquad \text{if } m < u_{j-1}. \end{cases} \tag{9}$$

4 Boolean Operations

Let:

$$C = \{\ \langle l_i, u_i\rangle \mid i = 0, \ldots, k-1\ \} \tag{10}$$

and:

$$C' = \{\ \langle l'_i, u'_i\rangle \mid i = 0, \ldots, k'-1\ \} \tag{11}$$

be two compact code pair sequences, with $k \geq 0$ and $k' \geq 0$.

4.1 Conjunction

Invoking $C.\mathbf{and}(C')$ will modify C according to C' by unsetting all the bits in C that are between packets in C', leaving C' unchanged.

- If $C = \{\}$, then C is left unchanged; else, if $C' = \{\}$, then C becomes $\{\}$.
- Otherwise (*i.e.*, if $k > 0$ and $k' > 0$), C is modified by invoking:
  - $C.\mathbf{unset}(0, l'_0)$; and,
  - $C.\mathbf{unset}(u'_i, l'_{i+1})$, for $i = 0$ up to $i = k' - 1$; and,
  - $C.\mathbf{unset}(u'_{k'-1}, u_{k-1})$.

  Note that in practice, when proceeding in the above order, as soon as the first argument of any $\mathbf{unset}(\ldots)$ is greater than or equal to u_{k-1}, there is no need to perform the unsetting nor proceed any further.

4.2 Disjunction

Invoking $C.\mathbf{or}(C')$ will modify C according to C' by setting all the bits in C that are within packets in C', leaving C' unchanged.

- If $C' = \{\}$, then C is left unchanged; else, if $C = \{\}$, then C becomes (a copy of) C'.
- Otherwise (*i.e.*, if $k > 0$ and $k' > 0$), C is modified by invoking:
  - $C.\mathbf{set}(l'_i, u'_i)$, for $i = 0$ up to $i = k' - 1$.

4.3 Negation

Since a bit vector is open-ended, we may define its negation only up to a length at least greater than its highest `1`-bit position. This operation is denoted as $C.\mathbf{not}(n)$. Thus, $\{\}.\mathbf{not}(n)$, is undefined for any $n \geq 0$.

Otherwise, for a non-empty code $C = \{\ \langle l_0, u_0\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \}$ and $n \geq u_{k-1}$, $C.\mathbf{not}(n)$ modifies C to become:

$$\{\ \langle 0, l_0\rangle, \ldots, \langle u_i, l_{i+1}\rangle, \ldots, \langle u_{k-1}, n\rangle\ \}. \tag{12}$$

Again, following our convention, if $l_0 = 0$, $\langle 0, l_0\rangle$ being undefinable, the first element of $C.\mathbf{not}(n)$ is$\langle u_0, l_1\rangle$. Similarly, if $n = u_{k-1}$, then $\langle u_{k-1}, n\rangle$ is undefinable and the last element of $C.\mathbf{not}(n)$ is $\langle u_{k-2}, l_{k-1}\rangle$.

5 Implementation Considerations

We need to come up with a data structure for representing compact code that would enable retaining maximal efficiency in the bit setting and unsetting operations, and hence in the boolean operation that rely on them.

Most frequently used operations on such a data structure C for an integer n are:

- $C.\mathbf{packet}(n)$—for n inside a packet in C, returning that packet number;
- $C.\mathbf{prev}(n)$—for n between packets in C, returning the upper index of the packet preceding n;
- $C.\mathbf{next}(n)$—for n between packets in C, returning the lower index of the packet following n;
- adding/removing a packet.

Because the elements of a code sequence are ranges rather than integers, one cannot expect hashed $\mathcal{O}(1)$ time access to find out whether a given integer lies within or between packets. So structures such as defined by the Java classes `java.util.HashSet` or `java.util.LinkedHashSet` cannot be used.

In order to make these operations at most $\mathcal{O}(\log(k))$ time for a compact code of size k, one way is to represent a compact code as a balanced binary tree of pairs of bit position spans $\langle l_i, u_i \rangle$, taking advantage of the ordering imposed by condition (1).

Thus, the `java.util.TreeSet` class looks like a convenient choice, since it offers the required data structure properties in addition to defining methods such as `first`, `last`, `higher`, `lower`, `add`, `remove`, *etc.*, as well as an order-respecting iterator.

On the other hand, the `java.util.TreeSet` is missing a *replace* method—which is critically needed for setting and unsetting bits. It is also missing an *insert* method that splices a new sequence of packet pairs into an existing compact code, which may also be often used. One must resort to several `add`/`remove` method invocations to replace or insert elements, which incur new searches (and possible intermediate rebalancing of the tree) each time. This is a waste since replacing and inserting can be done in $\mathcal{O}(1)$ time when having already found the required elements, and only one (final) $\mathcal{O}(\log(k))$ tree rebalancing.

Hence, rather than relying on the ready-to-use `java.util.TreeSet` class, it may be more beneficial to implement a new specific class for a compact code as a doubly linked list and a balanced binary tree adapted for the specific nature of its pair elements. This would make transparent the double links of each pair element and ease replacement and insertion.[4]

6 Discussion

A similar data structure was proposed by researchers in data and knowledge bases in [1]. However, the authors did not use that representation for lattice operations as we do here. Instead, they focused on using it for obtaining more compact range-sequence codes for the transitive closure of the "is-a" relation of a taxonomy. That representation is equivalent to the one we specify here and to the one in [3]. Contrary to [3], they define an element's code as the union of index ranges from the post-order arrangement of the a spanning tree of the "is-parent-of" relation of a taxonomy. Each concept in the taxonomy (*i.e.*, each element in the poset) of post-order index j is then encoded as the interval $\langle i, j \rangle$ where i is the smallest post-order index of all its descendants. Although they did not do it, it is easy to show that their representation is equivalent to bit vectors. But they did not specify lattice operations on their data structures as we do for ours in this document. What they focused on was minimizing the total number of packets in range codes. In order to do so, they suggest generating codes based on the "optimal" spanning tree for generating the most compact set of codes. The data structure and algorithm for what they call "compressed transitive closure" do not maintain dynamic interval consistency caused by potential adjacency as we do here.[5]

[4] Actually, the `java.util.TreeSet` does maintain a doubly-linked list for its elements in order to ensure its two ordered iterators (ascending and descending). But this structure is not made public and one cannot splice in new elements from a given found element. So a good start would be to modify the source code of `java.util.TreeSet.java` and adapt it to what we need.

[5] In fact, they see that only as a possible *a posteriori* optimization, but one that would cause their optimal spanning-tree finding algorithm to be incorrect if applied incrementally while it is executed.

Although they do cite [3], Agrawal *et al.* do so only in the conclusion as they had just noticed its publication. They suggest that their approach and that exposed in [3] might be combined for processing large taxonomies. As far as this author knows, no follow-up on this suggestion was carried out.

It is clear to this author how the work of [1], although orthogonal to ours, could be adapted to our needs as well in order to improve its space consumption. However, it is to be noted that their code-compaction method requires a topologically ordered poset. For very large taxonomies (over 1 million elements) the price of sorting the taxonomy may only be worth spending as a once-for-all prepocessing prior to query time [2]. This, of course, would make incrementality impossible, or too costly (*i.e.*, outweighing the benefits).

Finally, although this work has been motivated to obtain a compact representation for bit vectors, it should be noted that the data structure, and operations on it, specified in this document can be used to represent *arbitrary* sets of integers (or integer-indexed totally-ordered sets of any type) seen as sequences of intervals (and thus as sequences of these intervals' bounds). Set intersection is realized as the conjunction described in Section 4.1; set union as the disjunction described in Section 4.2; and, set complementation as the negation described in Section 4.3.

7 Conclusion

In this document, I described a specification of a way to represent bit vectors of very large size (hundreds of thousand, or millions of bits), and operations thereon. The basic idea is to see a bit vector as a sequence of set-bit position ranges. I gave methods for in-place bit-range setting and unsetting, and boolean operations on such sequences. I also discussed some implementation considerations.

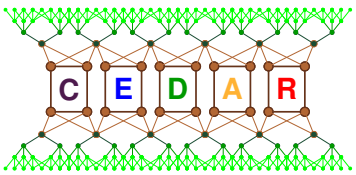
References

[1] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 253–262, Portland, Oregon, May/June 1989. ACM, SIGMOD Record 18(2). [See online[6]].

[2] Hassan Aït-Kaci and Samir Amir. Classifying and querying very large taxonomies—a comparative study to the best of our knowledge. $\mathcal{CEDAR}$ Technical Report Number 2, $\mathcal{CEDAR}$ Project, LIRIS, Département d'Informatique, Université Claude Bernard Lyon 1, Villeurbanne, France, May 2013. [See online[7]].

[3] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989. [See online[8]].

[6] http://dbs.informatik.uni-halle.de/Lehre/DBS_IIa_SS02/p253-agrawal.pdf
[7] http://cedar.liris.cnrs.fr/papers/ctr2.pdf
[8] http://www.hassan-ait-kaci.net/pdf/encoding-toplas-89.pdf



Technical Report Number 4

Efficient Encoding of Very Large Partial Orders

Hassan Aït-Kaci

September 2013