Computer and Information Science; Vol. 7, No. 1; 2014
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

A Formal Concept Analysis Approach to Data Mining: The QuICL Algorithm for Fast Iceberg Lattice Construction

David T. Smith[1]

[1] Indiana University of Pennsylvania, Indiana, Pennsylvania, USA

Correspondence: David T. Smith, Indiana University of Pennsylvania, Indiana, Pennsylvania, USA. E-mail: dtsmith@iup.edu

Received: May 29, 2013 Accepted: September 16, 2013 Online Published: October 28, 2013

doi:10.5539/cis.v7n1p10 URL: http://dx.doi.org/10.5539/cis.v7n1p10

Abstract

Association rule mining (ARM) is the task of identifying meaningful implication rules exhibited in a data set. Most research has focused on extracting frequent item (FI) sets and thus fallen short of the overall ARM objective. The FI miners fail to identify the upper covers that are needed to generate a set of association rules whose size can be readily exploited by an end user. An alternative to FI mining can be found in formal concept analysis (FCA). FCA derives a lattice whose concepts identify closed FI sets and connections identify the upper covers. However, most FCA algorithms construct a complete lattice and therefore include item sets that are not frequent. An iceberg lattice, on the other hand, is a lattice whose concepts contain only FI sets. This paper presents the development of the Quick Iceberg Concept Lattice (QuICL) algorithm. QuICL uses recursion instead of iteration to navigate the lattice and establish connections, thereby eliminating costly processing incurred by past algorithms. The QuICL algorithm was evaluated against a leading FI miner and lattice construction algorithms using cited benchmarks. Results demonstrate that QuICL provides performance on the order of FI miners yet additionally derive the upper covers. Beyond this, QuICL has proved to be very efficient, providing an order of magnitude gains over other lattice construction algorithms.

Keywords: data mining, concept lattice, knowledge discovery, association rule mining

1. Introduction

Association rule mining (ARM) is the task of identifying meaningful implication rules of the form $X \rightarrow Y$ exhibited in a data set, where X and Y are subsets of the *items* (i.e., possible distinct values of columns of a data set) and $X \cap Y$ is $\varnothing$ (Agrawal et al., 1993). The degree to which a rule is meaningful is defined by: i) *support*, the number of times both X and Y occur in the data set, and ii) *confidence*, the number of times that $X \rightarrow Y$ holds true relative to all occurrences of X. Mining association rules typically involves two steps: i) identifying *frequent item* (FI) sets (i.e., $X \cup Y$ that meet a minimum support threshold), and ii) deriving association rules from the item sets that meet a level of confidence.

A well known algorithm to extract FI sets from a data set is Apriori (Agrawal & Srikant, 1994). Apriori searches the space of all patterns in an iterative bottom-up breadth-first manner. Each iteration obtains counts for its current set of candidate patterns and removes from further consideration any candidate patterns that are not frequent or cannot be frequent. Apriori has proved to be efficient for mining frequent patterns of small length. However, for long patterns Apriori can be I/O intensive since each iteration requires a full scan of the data set. Furthermore, a bottom-up algorithm must obtain counts for each set in the power set of all items composing each frequent pattern. Thus, Apriori may be an intractable solution for FI sets of even moderate length (Han & Kamber, 2006).

ARM has be an active area of research (Pei et al., 2000; Zaki & Hsiao, 2002; Wang et al., 2003, Uno et al., 2004; Lucchese et al., 2006). However, this research has primarily focused on innovative theory and techniques for efficient extraction of FI sets. As such, they have fallen short of the overall task of mining association rules (Yahia et al., 2006). Key information not generated by these works is the derivation of upper covers of each FI set. An *upper cover* of a FI set I is a set of FI sets U such that $\forall\ I_u \in U$, $I_u \subset I$ and there does not exist a FI set I_2 where $I_u \subset I_2 \subset I$. Upper covers are needed to produce a set of association rules whose size is constrained to a number that can be readily exploited by an end user (Zaki & Hsiao, 2005; Yahia et al., 2006).

An alternate approach to frequency counting can be found in formal concept analysis (FCA) (Ganter & Wille, 1997). FCA is a branch of applied mathematics that has been applied to a wide variety of applications including linguistics, text retrieval, and economics (Ganter et al., 2005). It originated in the early 1980's and was first formalized in 1982 (Wille). It has since inspired numerous publications (Priss, 2006). According to FCA, a *concept* is defined as:

> **Definition 1:** Given a set of object identifiers (ids) $\mathcal{O}$, a set of items $\mathcal{I}$, and a relation $\mathcal{R}$ such that $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{I}$, a formal concept is a pair of sets $O \subseteq \mathcal{O}$ and $I \subseteq \mathcal{I}$ iff: $O = \{o \in \mathcal{O} \mid \forall\ i \in I, o\mathcal{R}i\}$ and $I = \{i \in \mathcal{I} \mid \forall\ o \in O, o\mathcal{R}i\}$, where $o\mathcal{R}i$ denotes object o has item i in relation $\mathcal{R}$.

Furthermore, between any two concepts $C_1 = (O_1, I_1)$ and $C_2 = (O_2, I_2)$ an order < exists between C_1 and C_2 iff $O_1 \subset O_2$ (or equivalently $I_1 \supset I_2$). The set of object ids of a concept is its *extent* and the set of items is its *intent*.

Let $\mathcal{L}$ be the set of all concepts derived from a data set where the attribute-values define the set of items and the tuple ids define the set of object ids. The concepts of $\mathcal{L}$ can be arranged in a lattice such that a connection (i.e., edge) is made between any two concepts C_1 and C_2 for which order < exists and there is no concept C_3 for which $C_1 < C_3 < C_2$. Given this property, tree terminology can be applied to a lattice. An *ancestor* concept C_a of concept C_1 is any concept for which an order $C_1 < C_a$ exists. A *descendent* concept C_d of concept C_1 is any concept for which an order $C_1 > C_d$ exists. A *parent* concept C_p of concept C_1 is ancestor concept for which there is no concept C_3 such that $C_1 < C_3 < C_p$. A *child* concept C_c of concept C_1 is descendent concept for which there is no concept C_3 such that $C_1 > C_3 > C_c$. An example of a concept lattice is depicted in Figure 1.

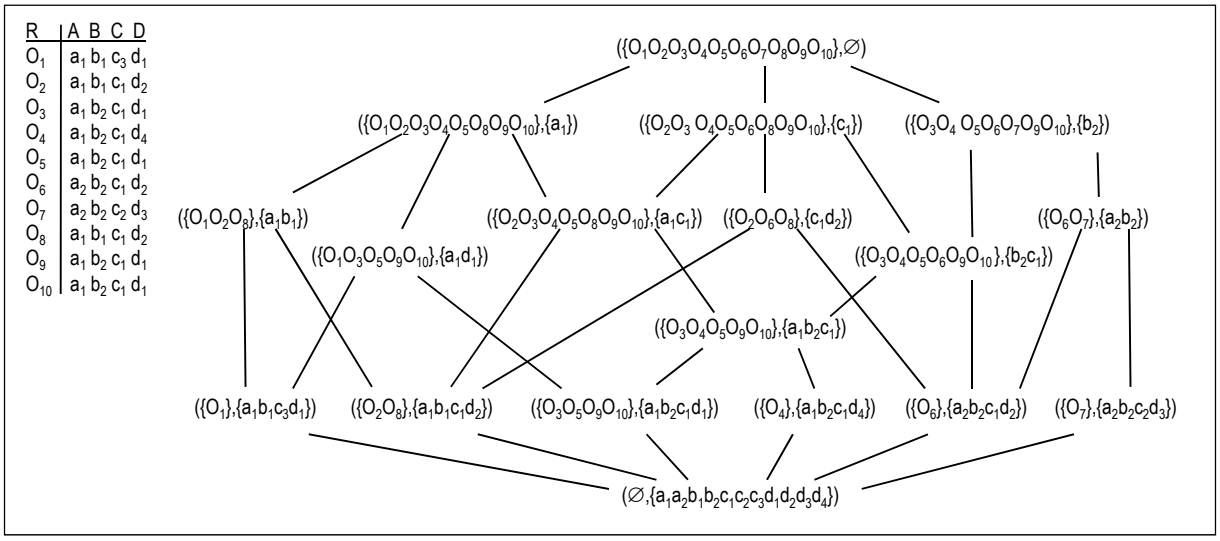

Figure 1. Example concept lattice

A concept lattice holds a number of interesting properties including:

> **Property 1:** Extent of concept C is the $\cap$ of sets of O defined by each $I_i \in I$ of C; dually the intent of C is the $\cap$ of the sets of I defined by each $O_i \in O$ of C.
>
> **Property 2:** If $I_i \in I$ of concept C_1 then $\forall\ C_2 \mid C_2 < C_1$, $I_i \in I$ of C_2; dually if $O_i \in O$ of concept C_1 then $\forall\ C_3 \mid C_3 > C_1$, $O_i \in O$ of C_3.
>
> **Property 3:** Extent of concept C is the $\cap$ of the O of all parent concepts of C, $\cap$ with the set of O defined by each $I_i \in I$ of C that is not $\in I$ of a parent concept of C; dually the intent of a concept C is the $\cap$ of the I of all child concepts of C, $\cap$ with the set of I defined by each $O_i \in O$ of C that is not $\in O$ of any child concept of C.

Concept lattices are of benefit to ARM. A concept's intent corresponds to an item set and the cardinality of extent corresponds to the item set support. Furthermore, the definition of a concept embodies the mathematical notion of closure. Thus, nodes of the concept lattice represent only *closed item sets* (i.e., an item set whose closure yields the same set), whose number can be orders of magnitude lower than the number of all item sets (Stumme, 2002). The concept lattice still contains the necessary and sufficient information to extract association rules and to compute both support and confidence. For example, from the concept ($\{O_1O_2O_8\}$, $\{a_1b_1\}$) of Figure 1,

the association rule $a_1 \rightarrow b_1$ can be mined. The support for $a_1 \rightarrow b_1$ can be extracted from the lattice by traversing any path from the bottom of the lattice through concepts where $\{a_1b_1\}$ is a subset of a concept's intent. Support is the size of the extent of the highest concept where $\{a_1b_1\}$ is a subset of a concept's intent. In this case, support for $a_1 \rightarrow b_1$ is 3, or 30%. Likewise support for a_1, the antecedent of $a_1 \rightarrow b_1$, can be extracted. The support for a_1 is 8, or 80%. Confidence is computed as support(rule) / support(antecedent(rule)). Thus, the confidence of $a_1 \rightarrow b_1$ is 37.5%. On the other hand, the confidence for $b_1 \rightarrow a_1$ is 100%, since the antecedent, now b_1, has a support of 30%. In the same manner the association rules $a_1 \rightarrow b_2$ $50\%_{supp}$ $62.5\%_{conf}$, $b_2 \rightarrow a_1$ $50\%_{supp}$ $71.4\%_{conf}$, and $a_1b_2 \rightarrow c_1$ $50\%_{supp}$ $100\%_{conf}$ can be mined from the concept $(\{O_3O_4O_5O_9O_{10}\}, \{a_1b_2c_1\})$. While a concept lattice contains the necessary and sufficient information to compute confidence and support, it includes concepts that do not meet the minimum support. Thus, use of a lattice construction algorithm for ARM a may incur substantial overhead, since such concepts are essentially unnecessary artifacts.

An iceberg lattice is a lattice that contains only the concepts whose support meets a given threshold. For example, Figure 2 depicts the concept lattice of Figure 1 as an iceberg lattice for both a minimum support threshold of 60% and 40%. As the threshold is lowered, more detail of the underlying concept lattice is revealed.

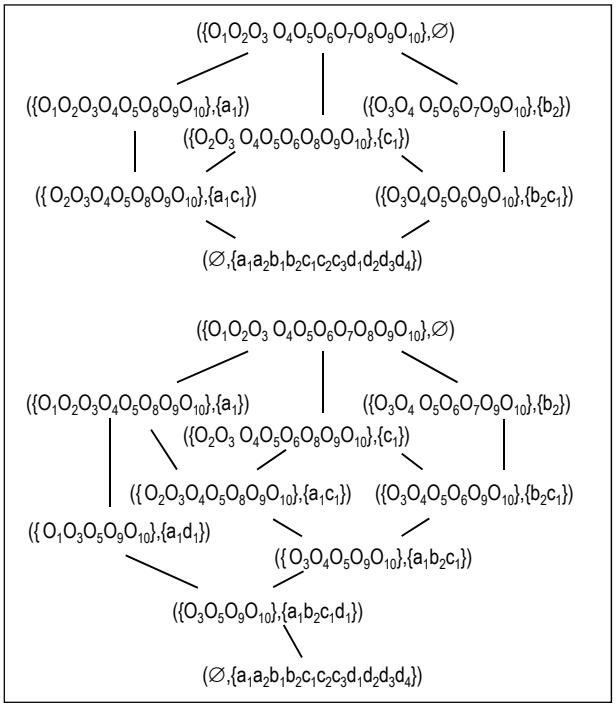

Figure 2. Examples of an iceberg concept lattice. Top – iceberg concept lattice at 60% support. Bottom – iceberg concept lattice at 40%. These are derived from the lattice of Figure 1 by discarding concepts not meeting the minimum support threshold

An iceberg lattice provides a model from which association rules can be efficiently mined (Stumme, 2002). Consider the alternate notation of an iceberg lattice depicted in Figure 3 that corresponds to the bottom iceberg lattice of Figure 2. Each concept node is labeled with a percentage representing the support together with any items, if any, for which there does not exist a greater concept containing the item. The edges are labeled with a percentage indicating the effective drop in confidence between two concepts. This notation enables association rules to be directly read from an iceberg lattice. An association rule $\alpha_1 \rightarrow \alpha_2$ will hold with 100% confidence for any concepts C_1 and C_2 where C_1 is labeled with α_1, C_2 is labeled with α_2, and $C_1 < C_2$. The support for the association rule is the support of C_1. For example, association rule $d_1 \rightarrow a_1$ $50\%_{supp}$ $100\%_{conf}$ can be read from lattice. Furthermore, an association rule $\alpha_1\alpha_2 \rightarrow \alpha_3$ will hold with 100% confidence for any concepts C_1, C_2, and C_3 where C_1 is labeled with α_1, C_2 is labeled with α_2, C_3 is labeled with α_3, and $C_3 >$ *meet* (i.e., greatest common sub-concept) of C_2 and C_1. The support of the association rule is the support of the meet concept. For example, the association rule $a_1b_2 \rightarrow c_1$ $50\%_{supp}$ $100\%_{conf}$ can be read. An association rule $\alpha_1 \rightarrow \alpha_2$ with less than 100% confidence can be read from any concepts C_1 and C_2 where C_1 is labeled with α_1, C_2 is labeled with α_2, and $C_1 <$

C_2. The support will be the support of C_2. The confidence will be the product of the confidences noted on the edges along the path from C_1 to node C_2. For example, the association rule $a_1 \rightarrow d_1$ $50\%_{supp}$ $62.5\%_{conf}$ can be read from the lattice of Figure 3. By a combination of the previous steps further association rules can be read. For example, the association rule $a_1b_2 \rightarrow d_1$ $40\%_{supp}$ $80\%_{conf}$ can be read from the lattice (the meet of $a_1b_2 \rightarrow$ the node labeled 40% support with $80\%_{conf}$, and d_1 is an ancestor of that node). Similarly, $c_1b_2 \rightarrow d_1$ $40\%_{supp}$ $66.7\%_{conf}$ (the meet of $c_1b_2 \rightarrow$ the node label 60% support with $100\%_{conf}$, the node label 60% $\rightarrow$ the node label 50% with $83.4\%_{conf}$, the node label 50% $\rightarrow$ the node label 40% with $80\%_{conf}$, therefore $c_1b_2 \rightarrow$ the node label 40% with a $66.7\%_{conf}$ drop in overall confidence (Note 1), d_1 is an ancestor of the node label 40%).

Extracting association rules from a list of FI sets may yield an excessive number, even when applying strict thresholds to both support and confidence. The rules may contain highly redundant information, for example $\alpha_1 \rightarrow \alpha_2, \alpha_2 \rightarrow \alpha_3, \alpha_1 \rightarrow \alpha_3, \alpha_1 \rightarrow \alpha_4, \alpha_1 \rightarrow \alpha_2\alpha_4$. The excessive size and redundancy impedes the usefulness of the extracted rules. What is desired is a meaningful subset that can be exploited by an end user. A *basis* is a minimal subset of association rules that can be combined to form all association rules without any loss of information. A basis can be extracted from an iceberg concept lattice using a systematic traversal of the lattice. The Duquenne-Guigues (1986) basis provides extraction of a minimal set of association rules with 100% confidence and the Luxenburger (1991) basis provides extraction of a minimal set of association rules with less than 100% confidence. Stumme et al. (2001) offer algorithms to traverse and extract the Duquenne-Guigues basis and the Luxemburger basis from an iceberg concept lattice.

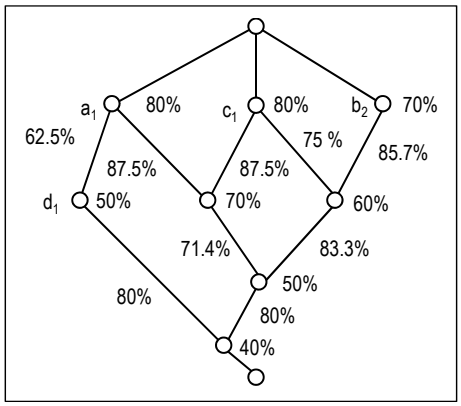

Figure 3. Iceberg lattice using an alternate notation. Each concept node is labeled with a percentage representing the support together with any items, if any, for which there does not exist a greater concept containing the item. Edges are labeled with a percentage indicating the effective drop in confidence between the two concepts

Given that an iceberg concept lattice provides an analysis tool to succinctly identify a basis of association rules, algorithms to construct an iceberg lattice are needed. This paper presents the Quick Iceberg Concept Lattice (QuICL) algorithm used to efficiently construct an iceberg lattice. When combined with lattice traversal algorithms, such as Stumme et al. (2001), QuICL provides an efficient ARM solution to generate of a basis of association rules that can be exploited by an end user. Beyond application to ARM, QuICL is a very efficient lattice construction algorithm that offers orders of magnitude improvement over past algorithms.

2. Background

2.1 Classical Association Rule Mining – Mining of Frequent Item Sets

There have been a number of algorithms developed to address the mining of long FI sets. Most notable are CLOSET (Pei et al., 2000), CHARM (Zaki & Hsiao, 2002), and CLOSET+ (Wang et al., 2003). CHARM constructs an itemset-tidset (IT) tree whose nodes are similar to the nodes of a concept lattice. It is a top-down, depth-first search that exploits a notion of equivalence classes to skip levels in order to quickly identify closed FI sets. CHARM involves a *vertical* data representation (i.e., list of object ids per item) and uses a difference based representation to enumerate the sets of object ids below the first level of its IT tree. It uses intersection to incrementally add data to its IT tree. The IT tree is dynamically pruned during processing using properties of set union and closure. Intersection is noted as an expensive operation that impedes the performance of the CHARM algorithm (Wang et al., 2003). Alternatively, CLOSET uses a frequent pattern (FP) tree to provide a compact representation of the data in memory. The FP tree is a *horizontal* representation that maintains counts, each

relative to a context of an ordered list of frequent items. Branches are added to the FP tree upon processing an object whose items omit one or more items in the path of an existing branch. Following construction of the FP tree, a divide and conquer algorithm that performs physical bottom-up projections on the FP tree together with item set merging and sub-item set pruning is used to identify the set of closed FI sets. CLOSET is shown to be effective for *dense* data sets (i.e., many items per transaction with few distinct items), but CLOSET's performance degrades rapidly on *sparse* data sets (i.e., few items per transaction with many distinct items) as the minimum support threshold is lowered. CLOSET+ offers enhancements to CLOSET; top-down pseudo projection algorithm to address sparse data sets and item and skipping to further prune the search space.

A survey provides an analysis of algorithms for mining closed FI sets from both a theoretical and analytical viewpoint (Yahia et al., 2006). Algorithms evaluated include TITANIC (Stumme et al., 2002), CLOSET, CLOSET+, and CHARM. TITANIC is a test and generate algorithm along the lines of Apriori that leverages theory from FCA. Yahia et al. draw several conclusions. There has been "frenzied activity" in developing algorithms to efficiently mine FI sets. These algorithms have made significant progress by leveraging theory in combination with carefully designed compact data structures. However, this activity has lost sight of the overall goal of producing a set of association rules that is "of exploitable size by end users". All algorithms fail to produce the upper covers and therefore unable to generate a reasonable basis of association rules. Without the upper covers, the derivation of association rules from the FI sets of even a modest context will generate an excessive number of rules that cannot be reasonably comprehended by end users. Other studies derive the same conclusion (Valtchev et al., 2004; Zaki & Hsiao, 2005; Lakha & Stumme, 2005).

2.2 Missaoui, Godin, and Alaoui Algorithm

Missaoui, Godin, and Alaoui (1995) algorithm (GMA) is an often cited lattice construction algorithm. It is an incremental algorithm. That is, given a concept lattice $\mathcal{L}$ and a new object O_i with its set of items I, GMA will insert the new object into the lattice to produce a new concept lattice $\mathcal{L}$+. Figure 4 depicts the incremental insertion of the first six objects relation R of Figure 1. As seen in Figure 4, the insertion of an object can result in modifying the extent of several existing concepts, generation of several new concepts, addition of links, and occasional removal of links. The insertion of a single object may result in numerous modified concepts and addition of many new concepts.

The strategy for GMA is to partition the current set of concepts into three groups: modified, generator, and old. Modified are concepts into which the object id of the next object is added. Generators are concepts are used to generate new concepts. All other concepts are considered old and play no role in the insertion process. Modified concepts are readily identified. They are concepts with an intent that is a subset of the next object's items. The identification of generator concepts, on the other hand, is more involved. Any concept whose intent intersects with, but not a subset of, the object's items is potentially a generator. However, not all such concepts are generators. A concept is a generator provided there does not exist an ancestor whose intent when intersected with the next object's items produces the same intersection set. For example, when inserting O_6 in Figure 4 the concepts ($\{O_3O_4O_5\}$, $\{a_1b_2c_1\}$), ($\{O_3O_5\}$, $\{a_1b_2c_1d_1\}$), and ($\{O_4\}$, $\{a_1b_2c_1d_4\}$) all have an intersection set of $\{a_1c_1\}$, only ($\{O_3O_4O_5\}$, $\{a_1b_2c_1\}$) is a generator. Each generator concept is used to create a new concept having the extent of the generator union the object id as its extent and the intersection set as its intent.

New concepts must be further linked into the lattice by searching for the parents. A potential parent is any concept, existing or generated, whose intent is a subset of the new concept's intent. In order to preserve the lattice property (i.e., connection exists between two concepts C_1 and C_2 provided there is no concept C_3 for which $C_1 < C_3 < C_2$), the potential parent is a parent only if it does not have a child whose intent is a subset of the new concept.

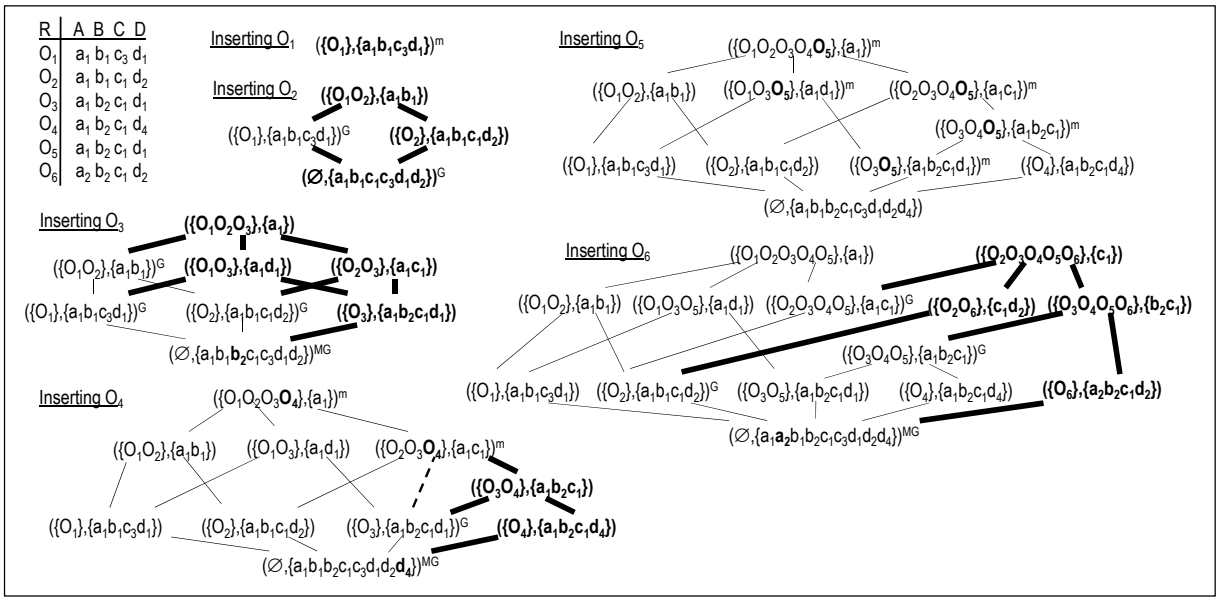

Figure 4. Progression of incremental object insertion into a concept lattice. Bold text indicates new concepts, inserted items or inserted objects, ^{G} a generator concept, ^{m} a modified concept, and dashed lines are removed links

The search for potential parents can be constrained to only consider concepts that are modified or generated. Occasionally a link between a parent and a child must be removed. This occurs when a parent for a concept is found and that parent is currently the parent of the generator concept that created the new concept. An example is the insertion of object O_4 shown in Figure 4. In these cases the new concept is being inserted between the parent and the generator. The removal of the link is required to preserve the lattice connection property.

The complete GMA algorithm is given in Algorithm 1. Lines 1 through 10 bootstrap the lattice upon insertion of the first object; for subsequent objects ensure the bottom concept contains all of the object's items. Line 11 declares a vector of sets that is used to: i) verify a potential generator is valid, and ii) limit the search for parents. Only modified and generated concepts are placed into this vector (lines 15 and 22). Line 12 provides the main loop to iterate over all concepts in the lattice in a top-down breath-first order (Note 2). Lines 13 through 17 identify and process modified concepts. Line 20 tests if a concept is a generator. If so, a new concept is generated (line 21) and linked into the lattice (line 23). Lines 24 through 33 search the Processed vector for the parents and links them to the new concept.

2.3 Other Lattice Construction Algorithms

Other notable lattice construction algorithms include Valtchev, Missaoui, and Lebrun divide and conquer (2002b), GALICIA-T (Valtchev et al., 2002), and Nourine and Raynaud (2002). Lindig and Datensystene begins by constructing a known concept, such as the top (or bottom concept), and then proceeds to generate its children (or parents). The process repeats for each found concept until the lattice is complete. Valtchev et al. divide and conquer recursively partitions the input data set into two sets, either based on items or objects. At each level a concept lattice is constructed for each set and the resulting lattices are then merged. GALICIA-T uses a *trie* data structure to represent the set of concept intents whereby each edge of the trie denotes the addition of an item in an item set. GALICIA-T algorithm inserts the next object into the lattice through a guided traversal of the trie to produce an independent trie data structure. The generated trie represents a set of new concepts that are then merged back into the source trie. The lattice is thus an adjunct to the core trie structure. Similarly, Nourine and Raynaud use a trie to represent its lexicographic tree. Each edge in the lexicographic tree denotes an object and nodes corresponding to concepts are augmented with an item list. The incremental insertion is performed on an item by item basis by using a union operation on object ids of concepts represented in the trie. If the result of union is present in the trie and augmented with an item list, the item is added to the node; otherwise it will be a new concept. For new concepts, the extent will be will be added to the trie as needed. Identification of children is performed by a test union and count procedure for each item in $\mathcal{I}$ that is $\notin$ new concept's intent.

```
Let Concept be a tuple {O, I, Children} where O is a set of object ids,
    is a set of items, and Children is a list of child concepts.
Let C_Bottom be a reference to the supremum of a concept lattice G
ADD(O_i, I)
 1.  if C_Bottom = Ø:
 2.      C_Bottom ←    new Concept ({O_i}, I)
 3.  else:
 4.      if    I ⊄ C_Bottom.I:
 5.          if    C_Bottom.O = Ø:
 6.              C_Bottom.I ← C_Bottom.I ∪ I
 7.          else:
 8.              C_New ← new Concept (Ø, I)
 9.              Add C_New to C_Bottom.Children
10.              C_Bottom ← C_New
11.  Processed ← Ø
12.  for each C_i ∈ G in ascending |I| order:
13.      if C_i.I ⊆ I:
14.          Add O_i to C_i.O
15.          Add C_i to Processed[|C_i.I|]
16.          if C_i.I = I:
17.              return
18.      else:
19.          Intersect ← C_i.I ∩ I
20.          if ¬∃ C_j ∈ Processed[|Intersect|] | C_j.I = Intersect:
21.              C_New ← new Concept (C_i.O ∪ {O_i}, Intersect)
22.              Add C_New to Processed[|Intersect|]
23.              Add C_i to C_New.Children
24.              for each C_k ∈ Processed ∧ |C_k.I| < |Intersect|:
25.                  if C_k.I ⊂ Intersect:
26.                      IsParent ← TRUE
27.                      for each C_Child ∈ C_k.Children ∧ IsParent:
28.                          if C_Child.I ⊂ Intersect:
29.                              IsParent ←FALSE
30.                      if IsParent:
31.                          if C_i ∈ C_k.Children:
32.                              Remove C_i from C_k.Children
33.                          Add C_New to C_k.Children
34.              if |Intersect| = |I|:
35.                  return
```

Algorithm 1. Godin, Missaoui, and Alaoui (GMA) lattice construction algorithm

Kuznetsov and Obiedkov (2002) provide a comparative survey of several lattice construction algorithms. Algorithms include: GMA, Nourine andRaynaud, and Valtchev et al. divide and conquer. Findings indicate that

there is no "best" algorithm and the each algorithm exhibit different performance depending on the data set. GMA is a good choice for sparse data sets, and batch algorithms are good for dense data sets. Valtchev et al. (2002) arrive at the same conclusions. Their study reports that GMA has good performance for data set with density (Note 3) less than 0.10, but lags with densities greater than 0.50.

2.4 Iceberg Lattice Construction Algorithms

Three algorithms to construct an iceberg lattice were found in literature: CHARM-L (Zaki & Hsiao, 2005), SPROUT (Choi, 2006), and Martin and Eklund (2008). CHARM-L, an extension to the CHARM algorithm, is an example of a lattice construction algorithm that is an integrated (Note 4) extension of a FI set miner. The lattice of the CHARM-L algorithm is maintained as an adjunct data structure from CHARM's IT tree. When the core CHARM processing identifies a new potential FI set, CHARM-L will attempt to insert a new concept representing the FI set into the lattice, as a child of the concept corresponding to the parent node in the IT tree. What remains is to identify concepts already in the lattice that are to become children of the new concept. Such identification is performed by intersecting concept id sets that are maintained within each node of the IT tree.

SPROUT is a lattice construction algorithm that provides an option to build an iceberg lattice. It begins by creating the top concept and then generates children by appending each object not in the concept's extent and inquiring the formal context for the item sets. Generated concepts are tested for closure and pre-existence. If not closed, the concept is discarded. If pre-existent, a parent-child link is added. The process repeated for each new concept.

Martin and Eklund is another algorithm that generates a lattice from a set of closed FI set found by a FI set miner. It maintains a border set of concepts that have been inserted into the lattice thereby limiting the concepts that must be examined during the insertion of the next closed FI set.

3. Methodology

While GMA and like algorithms are not directly suitable to construct an iceberg lattice, adapting the algorithm to add data incrementally on an item by item basis (i.e., vertical representation) and interchanging the roles of the set of object ids (O) and the set of items (I), results in an algorithm that can construct an iceberg concept lattice. The algorithm still performs a top-down level-wise search and insert process; however, these changes effectively invert the lattice. The addition of a predicate to ensure that the minimum support threshold has been met is the only remaining change needed to construct an iceberg lattice.

Preliminary tests (Note 5) validated the modified GMA algorithm functioned correctly. The Mushroom (Note 6) data set was used as the test case. The converted algorithm was tested with minimum supports of 50%, 30%, 10%, 1%, and 0%. The algorithm reported a number of concepts of 45, 427, 4,897, 51,672 and 238,709 respectively with execution times of 0.04, 0.39, 7.17, 160.28, and 1,198.08 seconds. The reported number of concepts is the same as found by the CHARM-L algorithm. While the execution time for high supports was comparable to CHARM-L, the performance significantly degraded by an order of magnitude as support is lowered. Thus, the modified GMA algorithm cannot compete with the leading ARM algorithms.

This section describes the development of the Quick Iceberg Concept Lattice (QuICL – pronounced kwi-kəl) algorithm. QuICL provides incremental construction of a concept lattice along the lines of GMA, but approach the insertion process from the bottom of the lattice as opposed to a top-down, level-wise search for generators. The structure of the lattice is used to navigate to a point of change. Recursion is used to facilitate the location of additional points of change and enable linkage between parent and child concepts. The result is an algorithm that constructs all 238,709 concepts derived from the Mushroom data set in less than three seconds, a performance improvement over GMA that is near three orders of magnitude.

3.1 A Step Towards an Efficient Incremental Algorithm

A step towards an efficient incremental insertion algorithm for an iceberg lattice is to apply a few minor modifications to the representation of the lattice. In addition to interchanging the roles of the set of object ids (O) and the set of items (I) to invert the lattice, the cardinality of I in a given concept can be significantly reduced by exploiting the lattice property: if $I_i \in I$ of concept C_1 then $\forall\ C_2 \mid C_2 < C_1, I_i \in I$ of C_2. Thus, an item $I_i \in I$ of concept C_1 does not need to be physically recorded in a concept if there exists a concept C_2 such that $C_2 > C_1$ and $I_i \in I$ of concept C_2. Instead, the item I_i is implied by the lattice structure. An item I_i need only be recorded in a concept at its *maximal* position (i.e., lowest position in the inverted lattice). This representation is also desirable for direct extraction of association rules (see Section 1). Another modification is to omit a topmost concept whose intent is the set of all items in the concept lattice. As a result, the concept lattice becomes a *semi-lattice*. The semi-lattice can be readily converted to a complete lattice by a post-construction step to add a common

topmost parent for all concepts in the lattice that do not have parents. For the purpose of ARM, this post-step is not needed. The final modification is to redefine the bottom concept simply as an entry point into the lattice. Thus, the bottom concept does not hold any objects or items. It is created upon initial construction of an empty lattice and its intent and extent are not updated.

The previously mentioned changes will simplify the processing in the GMA algorithm without any loss of necessary information. The steps of GMA that add an item to the intent of concepts whose extent is a proper subset of the next item's objects are not needed, since the lattice structure will imply the item. As a result, concepts whose extent is a proper subset of the next item's objects will not need to be visited. Furthermore, the pre-steps to ensure the extent of the bottom concept includes new object ids can be eliminated. There is, however, one small side effect. In the event an item exists common to all objects, GMA would place that item and its object ids into the bottom concept. With the proposed changes, the item and object ids will be in a new concept that is the sole parent to the bottom concept.

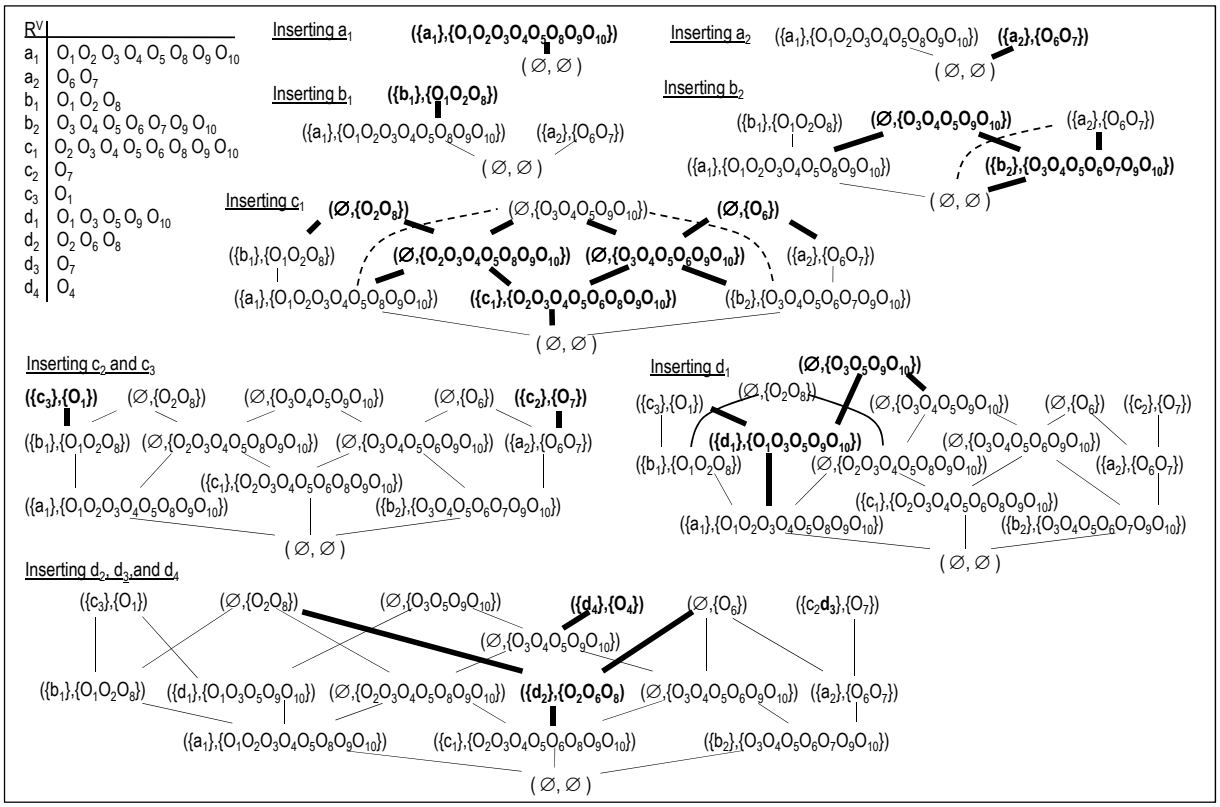

Figure 5. Progression of incremental item insertion into a concept lattice. Bold text and weighted lines identify new elements. Dashed lines indicate removed links. R^v is the vertical representation of R of Figure 1

Given the proposed modifications to the lattice structure, Figure 5 depicts the progression of incremental item insertions of the data in relation R of Figure 1 into an inverted concept lattice. The final lattice of Figure 5 is the inverted form of the lattice given in Figure 1. Before presenting an algorithm to construct a lattice using the proposed structure, a few observations are noteworthy:

- Insertion of an item whose extent = the extent of a concept C_i within the lattice is accomplished by simply adding the item to C_i. C_i can be found by traversing the lattice from the bottom along any path where the item's extent $\subseteq$ a concept's extent. An example is inserting d_3 in Figure 5.
- Except for the previous case, a new concept C_{New} will be added to the lattice. That concept will forever hold the item.
- If an empty lattice is defined as a bottom concept with an empty intent and extent, then any subsequent insertion of the new concept C_{New} will always be performed above another concept. Let the concept above which C_{New} is to be inserted be denoted as C_{Base}. C_{Base} can be identified by traversing the lattice along any path

where the item's extent is $\subset$ of a concept's extent. For example, when inserting d_4 with object id set $\{O_4\}$ into the lattice of Figure 5 the base concept will be $(\varnothing, \{O_3O_4O_5O_9O_{10}\})$.

- For all parent concepts C_p of C_{Base} such that the extent of C_p is not =, $\subset$, or $\supset$ of new item's extent, the new concept C_{New} will be a sibling of each C_p. C_{Base} will be a child of the new concept. If the extent of a $C_p \cap$ item's extent is not empty then another new concept with an extent = extent of a $C_p \cap$ item's extent must be found or inserted above C_p. Such concept can be found, or if needed created, by recursing using a null item and extent of $C_p \cap$ item's extent as the set of object ids. The concept returned from the recursive call will also be a parent of C_{New}. An example of finding already existing concepts in the recursive call is inserting d_2 in Figure 5. An example of creating a new concept in the recursive call is inserting b_2.
- For all parent concepts C_p of C_{Base} such that the extent of C_p is $\subset$ of extent of C_{New}, C_{New} will be inserted between C_{Base} and C_p. C_{Base} will no longer be a child of C_p. Instead the C_{New} will be a child of C_p and C_{Base} will be a child of the C_{New}. An example is inserting c_1 with object id set $\{O_2O_3O_4O_5O_6O_8O_9O_{10}\}$ into the lattice of Figure 5. The object ids are a superset of the extent of concept $(\varnothing, \{O_3O_4O_5O_9O_{10}\})$. Thus, concept $(\varnothing, \{O_2O_3O_4O_5O_8O_9O_{10}\})$ is inserted between the base concept $(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_9O_{10}\})$ and concept $(\varnothing, \{O_3O_4O_5O_9O_{10}\})$.

Given these observations, an alternative algorithm to GMA can be formulated. For each insertion, GMA processes all concepts in a top-down, breath-first manner to modify existing concepts and to generate new concepts. The top-down traversal is used to facilitate identification of generators and limit the search for parent concepts. The noted observations, however, suggest alternate approach. The identification of generator concepts can be performed from the bottom up using the lattice structure to navigate to a generator (i.e., a base concept). Furthermore, recursion can be used to find, or if needed create, the parent concepts.

Algorithm 2 presents an incremental insertion algorithm to construct a concept lattice. For this algorithm, each concept is a tuple composed of a list of items, a list of object ids, and a list of parent concepts. A designated empty concept named C_{Bottom} provides an entry point into the lattice. The algorithm begins with the BUILD-LATTICE function. This function accepts a formal context $K\{\mathcal{I}, \mathcal{O}, \mathcal{R}\}$. BUILD-LATTICE creates an empty concept lattice consisting of the bottom concept (line 1) and then incrementally adds each item into the lattice using the INSERT function (lines 2 and 3). After inserting all items, the bottom concept is returned (line 4). The INSERT function provides the incremental insertion of an item into the lattice or sub-lattice. INSERT is passed a reference to a concept, referred to as the base concept C_{Base}, above which an item id I_i together with its extent O is to be inserted. The item id can and will often be omitted when inserting into a *sub-lattice* (i.e., a subset of a lattice consisting of a concept and all its ancestors). INSERT involves three phases; i) navigate into the lattice and identify a list of concepts to be further processed, ii) if needed, construct a new concept, and iii) process the list of concepts identified by the first phase and link the new concept into the lattice. Both the navigation phase and link phase recursively call the INSERT function as needed.

INSERT first defines an empty list of tuples consisting of a type indicator with values SUP or ISET, an intersection set, and a reference to the concept that generated the intersection set (line 5). This list is populated during a navigate-prepare phase and is processed during the link phase. The intersection set is the result of intersecting the object set O passed to the INSERT function with the extent of a parent concept. A type SUP indicates O is a superset of the extent of the parent concept. Type ISET indicates that O is neither superset nor subset. INSERT proceeds to compare O with the extent of each parent of the base concept (lines 8 through 17). If O is equal to a parent concept's extent then the item I_i, if supplied, is added to the concepts list of items (lines 9 and 10). The insertion is complete. For purposes discussed later, INSERT returns a reference to the modified concept (line 11). If O is a subset the extent of any parent concept then INSERT recurses using the parent as the new base concept (lines 12 and 13). This effectively navigates into the concept lattice to locate the position above which the item will be inserted. If O is superset of the extent of a parent then a tuple composed of SUP, a reference to the parent concept, and the parent's extent is added to the ToProcessList for later processing (lines 14 and 15). If O is neither equal to, subset, nor superset of the extent of a parent concept, and O intersect the extent of a parent is non-empty, then a tuple composed of ISET, a reference to the parent concept, and O intersect the extent of the parent is added to the ToProcessList (lines 16 and 17).

If comparison of O with the extents of all parent concepts does not encounter a parent concept where O is equal to or a subset of the parent's extent, then a new concept node will be constructed (line 18). The new concept will contain the item I_i in its intent and O as its extent. The new concept will be a child of all SUP concepts in the ToProcessList, a sibling to the ISET concepts, and a parent to the base concept.

```
Let Concept be a tuple {I, O, Parents} where I a list of Items, O is a
      list of Object Ids, and Parents a list of parent concepts.
BUILD-LATTICE(K{I, O, R})
 1.  C_Bottom ← new Concept (∅, ∅)
 2.  for each I_i ∈ I:
 3.        INSERT(C_Bottom, I_i, o(I_i))   // o(I_i) is the set O derived from R
 4.  return C_Bottom
INSERT(C_Base, I_i, O)
 5.  ToProcessList ← ∅      // list of tuples {Type, Concept, O} where
 6.  // Type ∈ {SUP, ISET}, Concept the intersecting concept, and
 7.  // O a set of object ids resulting from intersection
 8.  for each C_Parent ∈ of C_Base.Parents:     // Navigate-prepare phase
 9.     if   O = C_Parent.O:
10.        Add I_i to C_Parent.I
11.        return C_Parent
12.     else if O ⊂ C_Parent.O:
13.        return INSERT (C_Parent, I_i, O)
14.     else if O ⊃ C_Parent.O:
15.        Add {SUP, C_Parent, C_Parent.O} to ToProcessList
16.     else if O ∩ C.O ≠ ∅:
17.        Add {ISET, C_Parent, O ∩ C_Parent.O} to ToProcessList
18.  C_New ← New Concept({I_i}, O)
19.  for each T_i ∈ ToProcessList:              // Link phase
20.     if   T_i.Type = SUP:
21.        Remove T_i.Concept from C_Base.Parents
22.        Add T_i.Concept   to C_New.Parents
23.     else if   T_i.Type = ISET:
24.        C_Parent ← INSERT (T_i.Concept, ∅, T_i.O)
25.        Add C_Parent to C_New.Parents
26.  Add C_New to C_Base.Parents
27.  return C_New
```

Algorithm 2. A recursive incremental lattice construction algorithm

After creating the new concept, the final phase of the algorithm processes the concepts in the ToProcessList and links the new concept into the lattice. For a parent concept in the ToProcessList with a SUP type, the parent will no longer be a parent of the base concept (line 20). Instead it will be the parent of the new concept. Thus, the parent concept is removed from the base concept's list of parents (line 21) and added to the new concept's parents (line 22). Each parent concept for which O is neither equal to, a subset of, nor superset of the parent's extent will be a sibling to the new concept. Furthermore, if O intersect the extent of a sibling is not empty then additional processing is required to add the information about O intersect the extent of a sibling into the lattice. Such siblings are the concepts in the ToProcessList that have an ISET type. A concept representing O intersect the extent of a sibling must be found within the lattice, or if absent created, and added as a parent of the new concept. To do this, the algorithm recurses using the sibling as the base concept, a null item, and O intersect the extent of the sibling as the set of object ids (line 24). The concept returned by the recursive call is added to the new concept's parents (line 25). Finally, the new concept is added to the parents of the base concept and the new concept is returned (lines 26 and 27).

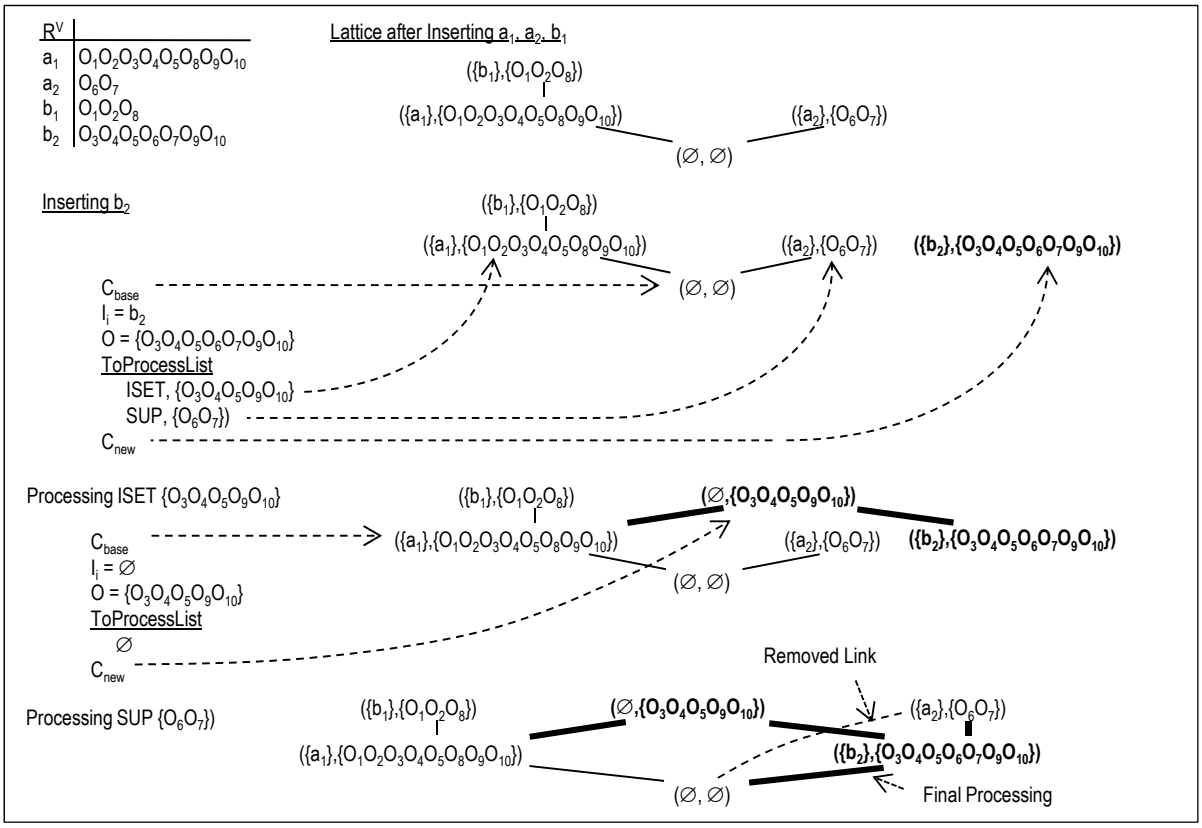

Figure 6. Sample walkthrough of Algorithm 2 execution

3.2 Walk Through of the Algorithm Execution

Figure 6 provides a sample walkthrough of executing Algorithm 2. This walkthrough corresponds to the insertion of b_2 in Figure 1. Execution begins with a call to INSERT with the C_{Base} referencing the bottom concept $\{\varnothing,\varnothing\}$, $I_i = b_2$, and $O = \{O_3O_4O_5O_6O_7O_9O_{10}\}$. The navigate-prepare phase tests the intersection of O with each parent of $\{\varnothing,\varnothing\}$. The intersection test of O intersect the extent of $(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_9O_{10}\})$ results in adding an ISET tuple to the ToProcessList. The tuple contains the intersection set $\{O_3O_4O_5O_9O_{10}\}$ and the concept $(\{a_1\},\{O_1O_2O_3O_4O_5O_8O_9O_{10}\})$. The intersection test of O intersect the extent of $(\{a_2\},\{O_6O_7\})$ results in adding a SUP tuple to the ToProcessList. Since the navigate-prepare phase did not encounter a parent concept where $O \subseteq$ parent's extent, a new concept $(\{b_2\},\{O_3O_4O_5O_6O_7O_9O_{10}\})$ is created and the tuples of the ToProcessList are then processed. Processing the $\{ISET,(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_9O_{10}\}),\{O_3O_4O_5O_9O_{10}\}\}$ tuple involves a recursive call to INSERT with the C_{Base} referencing the concept $(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_1O_{10}\})$, $I_i = \varnothing$, and $O = \{O_3O_4O_5O_9O_{10}\}$. The navigate-prepare phase of the recursive call to INSERT produces an empty ToProcessList since the extent of $(\{b_1\}, \{O_1O_2O_8\})$, the sole parent of concept $(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_9O_{10}\})$, has an empty intersection with $\{O_3O_4O_5O_9O_{10}\}$. Thus, the recursive call completes by creating the concept $(\varnothing, \{O_3O_4O_5O_9O_{10}\})$ and adding it as a parent of $(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_9O_{10}\})$. The new concept is returned from the recursive call. The returned concept is added as a parent of $(\{b_2\}, \{O_3O_4O_5O_6O_7O_9O_{10}\})$ by the base invocation of INSERT.

Processing the $\{SUP, (\{a_2\}, \{O_6O_7\}), \{O_6O_7\}\}$ tuple involves removing $(\{a_2\}, \{O_6O_7\})$ from the parents of C_{Base}, being $\{\varnothing,\varnothing\}$, and adding it as a parent to the C_{New}, being $(\{b_2\}, \{O_3O_4O_5O_6O_7O_9O_{10}\})$. At this time all tuples in the ToProcessList have been processed. The first invocation of INSERT completes by adding C_{New} as a parent to C_{Base} and returning a reference to C_{New}.

The walkthrough of b_2 inserting given in Figure 6 demonstrates a majority of the execution paths through the algorithm. However, the walkthrough did not execute the paths where the O in the call to INSERT are equal to or a subset of the parent's extent. Such execution paths are readily apparent in many of the other insertions depicted in Figure 5. For example, insertion of d_3 will call INSERT with C_{Base} referencing the bottom concept $\{\varnothing,\varnothing\}$, $I_i = d_3$, and $O = \{O_7\}$. The navigate-prepare phase will recurse with C_{Base} referencing the concept $(\{b_2\}, \{O_3O_4O_5O_6O_7O_9O_{10}\})$,

since the O is a subset of the extent. The navigate-prepare phase of the recursive call will further recurse with C_{Base} referencing the concept ({a_2}, {O_6O_7}). The navigate-prepare phase of this recursive call will encounter a parent concept whose extent equals O. That concept is ({c_2}, {O_7}). In this case I_i, being d_3, is inserted into the intent of ({c_2}, {O_7}) and a reference to this concept is returned back through all invocations.

3.3 A Shortcoming and a Correction

There is currently a defect in Algorithm 2 in that it may violate the lattice connection property (i.e., edge is made between any two concepts C_1 and C_2 for which order < exists and there is no concept C_3 for which $C_1 < C_3 < C_2$). These errors are due to relationships between the concepts referenced in the ToProcessList; either between two ISET tuples or between an ISET and SUP tuple.

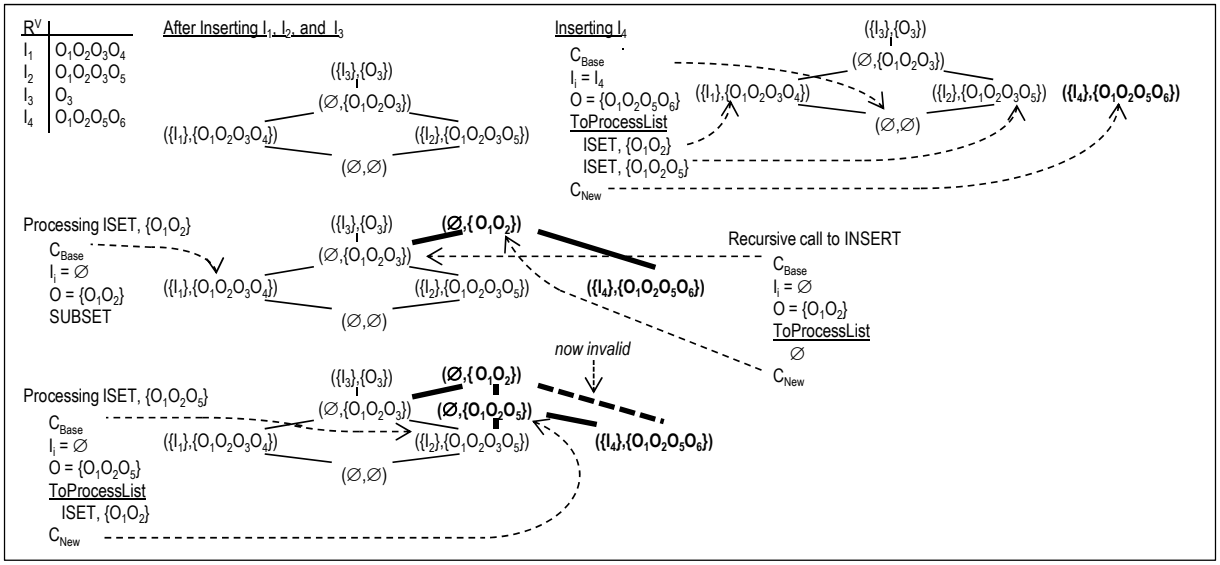

Figure 7. Invalid edge generated between new concepts

The processing of all related tuples results in adding invalid parent-child links. If there exists a non-trivial meet in the lattice between the referenced concepts then the intersection sets recorded in the tuples of ToProcessList of the related concepts will be the extent of the meet, and therefore the intersection sets will be the same. Thus, an approach to correcting the flaw is to remove all but one of the tuples in the ToProcessList of any tuples having the same intersection set. However, this approach is not sufficient since there exists cases where the invalid link does not involve a concept that is currently in the lattice. These cases are still the result of a relationship between concepts in the ToProcessList. A case is depicted in Figure 7. Here, the related concepts referenced in the ToProcessList are ({I_1}, {$O_1O_2O_3O_4$}) and ({I_2}, {$O_1O_2O_3O_5$}), and the meet concept is (∅, {$O_1O_2O_3$}). The invalid link will occur regardless of the order in which the tuples of the ToProcessList are processed. The processing of {ISET, ({I_1}, {$O_1O_2O_3O_4$}), {O_1O_2}} before {ISET, ({I_2},{$O_1O_2O_3O_4$}), {$O_1O_2O_5$}}, as shown, will create the concept (∅, {O_1O_2}) when processing {ISET, ({I_1}, {$O_1O_2O_3O_4$}), {O_1O_2}}, then create concept (∅, {$O_1O_2O_5$}) when processing {ISET, ({I_2}, {$O_1O_2O_3O_4$}), {$O_1O_2O_5$}}. On the other hand, if {ISET, ({I_2}, {$O_1O_2O_3O_4$}), {$O_1O_2O_5$}} is processed first, then both concepts (∅, {O_1O_2}) and (∅, {O_1O_2 O_5}) will be created upon processing {ISET, ({I_2}, {$O_1O_2O_3O_4$}), {$O_1O_2O_5$}}. The subsequent processing of {ISET, ({I_1}, {$O_1O_2O_3O_4$}), {O_1O_2}} will simply add the violating edge. Therefore a solution is to identify and remove all the tuples in the ToProcessList that have an intersection set that is a subset of the intersection set of other tuples. Thus to fully correct the problem, an algorithm to purge such tuples from the ToProcessList is needed.

A purge subsets algorithm involves comparing the intersection set of each tuple with the intersection set of every other tuple in the ToProcessList. This will introduce a potential $O(\boldsymbol{n}^2\ \boldsymbol{m})$ asymptotic complexity when $\boldsymbol{n}$ is the number of tuples in the ToProcessList and $\boldsymbol{m}$ is the size of the intersection sets. While the number of tuples in a given ToProcessList is bounded by the number of parent concepts of a given base concept, it is desired that the purge subsets algorithm be highly efficient and avoid any unneeded processing. There is no need to compare two SUP tuples, since SUP tuples cannot be a subset of other tuples. Furthermore, two ISET tuples cannot be both a subset and superset of each other. Therefore, the only tests needed between any two tuples are: i) a subset test when the first tuple is an ISET, and ii) a superset test when the second tuple is an ISET. The later will only be

performed if the first tuple is not an ISET, or if the result of the subset test is false. Furthermore, to obtain an $O(\boldsymbol{n}^2 \boldsymbol{m})$ complexity but not $O(\boldsymbol{n}^2 \boldsymbol{m}^2)$ the sets of object ids must be maintained in sorted order. This is necessary for fast determination of subset and superset operations. These operations can be optimized to determine an outcome as soon as possible. A subset operation on sorted lists can report false if at any time an id is found in the first set that does not exist in the second, or the number of ids yet to be examined in the first set is greater than the number of ids yet to be examined in the second. Dually, a superset operation can report false if at any time an id is found in the second set that does not exist in the first, or the number of ids yet to be examined in the first set is less than the number of ids yet to be examined in the second.

Algorithm 3 presents an efficient algorithm to purge tuples in the ToProcessList. Function PURGE-SUBSETS accepts the ToProcessList tuples. Lines 1 and 2 provide loops to compare each tuple with every other tuple. Lines 3 through 6 perform the comparisons between the tuples and removal of the subset tuples as needed.

```
PURGE-SUBSETS(ToProcessList)
  // ToProcessList is a list of tuples {Type, Concept, O} with
// Type ∈ {SUP, ISET}, Concept a reference to a concept,
// and O a set of object ids
1.  for each Pi ∈ ToProcessList:
2.      for each Pj ∈ ToProcessList ∧ Pj comes after Pi:
3.          if   Pi.Type = ISET ∧ Pi.O ⊂ Pj.O:
4.              Remove Pi from ToProcessList
5.          else if   Pj.Type = ISET ∧ Pi.O ⊃ PjO:
6.              Remove   Pj from ToProcessList
```

Algorithm 3. PURGE-SUBSETS algorithm

3.4 The QuICL Algorithm

In addition to calling the PURGE-SUBSETS function, there are three more enhancements; the first maintains the parent concepts in an order that may potentially improve performance, the second enables a specification of a minimum support threshold in order to construct iceberg lattices, and the third removes redundant intersect operations thereby further improving performance. The rationale to maintain parent concepts in a sorted order is to reduce the number of times the body of the navigate-prepare loop is executed. If during the iteration over parents, a parent concept whose extent is equal to or subset of the set of object ids is encountered the algorithm returns without testing the remaining parents. To increase the probability that such parent concepts are encountered sooner than later, the parents are maintained in descending order of the cardinality of extents.

To construct an iceberg lattice the insertion must discard any item whose extent does not meet a minimum support threshold. In addition, the processing must prevent construction of concepts whose extent would not meet the threshold. Since the extent for a new concept resulting from an intersection with another concept is the intersection set that is stored in the tuples of the ToProcessList, a predicate on the size of the intersection set can be used to prevent construction of such concepts. The predicate can be tested before adding an ISET tuple to the ToProcessList.

Testing and analysis of the Algorithm 2 revealed that more intersections are being performed than needed. This is the result of the same parent concepts being intersected from multiple invocations of the INSERT function. In such case, each invocation has a different base concept that shares the given parent. Even though each invocation may be passed a different set of object ids, the resulting intersection set will be the same during insertion of a given item. This is the case since the intersection set is ultimately the intersection of the parent's extent and the extent of the item being inserted. Thus, an enhancement is to cache (Note 7) each intersection set with its parent concept for the duration of an item insertion. Between item insertions all cached intersection sets are discarded.

While the intersection set of each invocation is the same, the outcome of comparison (i.e., $=$, $\subset$, $\supset$, and $\cap$) on which Algorithm 2 is dependent can be different. The outcome of comparison can be readily determined by performing tests on the cardinalities of the cached intersection set, the parent's extent, and the object id set passed to the INSERT function. Table 1 provides identification of an outcome based on the cardinality of these

sets.

In caching the intersection set in the parent concept, care must be taken to avoid incurring a penalty (Note 8) in memory consumption. A penalty can be avoided by using the appropriate reference as the intersection set. If the outcome of comparison is equal or a subset, then cached intersection set is set to the object id set passed to INSERT. If the outcome of comparison is superset, then cached set is set to $C_{Parent}.O$. If $|O \cap C_{Parent}.O| <$ minimum support, the cached set is set to an empty set. Only when the outcome of comparison is intersect and the intersection set meets the minimum support threshold will the new intersection set be cached. However, using a reference to this same set in the ISET tuples of the ToProcessList will result in no additional memory consumption. This set ultimately becomes the extent of a new concept that is added to the lattice.

Table 1. Determination of intersection outcome. $C_{parent}.IS$ is the cached intersection set of a parent concept, O is the object id set passed to INSERT, and $C_{parent}.O$ is the extent of a parent concept

Test on Intersect	Outcome
$\|C_{parent}.IS\| = 0$	No relationship
$\|C_{parent}.IS\| = \|O\| \wedge \|C_{parent}.IS\| = \|C_{parent}.O\|$	$O = C_{Parent}.O$
$\|C_{parent}.IS\| = \|O\| \wedge \|C_{parent}.IS\| < \|C_{parent}.O\|$	$O \subset C_{Parent}.O$
$\|C_{parent}.IS\| < \|O\| \wedge \|C_{parent}.IS\| = \|C_{parent}.O\|$	$O \supset C_{Parent}.O$
$\|C_{parent}.IS\| < \|O\| \wedge \|C_{parent}.IS\| < \|C_{parent}.O\|$	$O \cap C_{Parent}.O$

The applying these three changes to Algorithm 2 together with a call to PURGE-SUBSET is the QuICL algorithm, given in Algorithm 4. Line 30 provides the call to the PURGE-SUBSETS function. Lines 38 and 39 specify an order for parents of a concept. Line 2 discard items that do not meet the minimum support threshold. Line 14 tests that the size of the intersection set meets the minimum support threshold. Lines 9 through 12 obtain a reference to the intersection set IS. If the intersection set was previously computed then it is obtained from the cache, otherwise it is computed. Lines 16, 20, 23, and 26 use tests against the cardinality of the intersection set to determine an outcome of comparison. Lines 17, 21, 24 and 27 cache the intersection set during insertion of a given item. Line 3 clears the cached intersection sets following item insertion.

In a preliminary test, the QuICL algorithm constructed the complete lattice for the Mushroom data set in three seconds. This represents a gain in excess of two orders of magnitude over the GMA algorithm.

4. Results

QuICL was empirically evaluated against the CHARM, CHARM-L, and the iceberg enhanced GMA algorithms. The C version of the CHARM and CHARM-L were downloaded from the author's web site and translated to Java (Note 9). The CHARM implementation utilized memory mapped I/O to read the object ids from a vertical representation of a data set. On translating to Java, the memory mapped I/O was converted to the available random access classes. This introduced a performance problem since the CHARM implementation re-reads the sets of object ids multiple times when generating the first level of CHARM's IT tree. The implementation was enhanced to cache in memory the object id sets. The GMA algorithm with modifications for iceberg processing and QuICL (Algorithm 4) were directly implemented in Java.

```
Let Concept be a tuple {I, O, Parents, IS} where I is a list of items, O is a list of object ids,
    Parents is a list of parent concepts, IS is a reference to a cached intersection set.
QUICL (K{I, O, R}, MinSupp)
 1.  C_Bottom ← new Concept (∅, ∅)
 1.  for each I_i ∈ I ∧ |o(I_i)| ≥ MinSupp:
 2.      INSERT(C_Bottom, I_i, o(I_i)) // o(I_i) is the set O derived from R
 3.      ∀ C_i ∈ C_Bottom | C_i.IS not ∅, C_i.IS ← ∅
 4.  return C_Bottom
INSERT(C_Base, I_i, O)
 5.  ToProcessList ← ∅      // list of tuples {Type, Concept, O} where
 6.  // Type ∈ {SUP, // ISET}, Concept the intersecting
 7.  // concept, and O a set of object ids resulting from intersection
 8.  for each C_Parent ∈ of C_Base.Parents:      // Navigate-prepare phase
 9.      if C_Parent.IS = ∅:
10.          IS ← O ∩ C_Parent.O
11.      else:
12.          IS ← C_Parent.IS
13.
14.      if |IS| < MinSupp:
15.          C_Parent.IS ← {∅}
16.      else if    |IS| = |O| ∧ |IS| = |C_parent.O|:
17.          C_Parent.IS ← C_parent.O
18.          Add I_i to C_Parent.I
19.              return C_Parent
20.      else if |IS| = |O| ∧ |IS| < |C_parent.O|:
21.          C_Parent.IS ← O
22.          return INSERT (C_Parent, I_i, O)
23.      else if |IS| < |O| ∧ |IS| = |C_parent.O|:
24.          C_Parent.IS ← C_parent.O
25.          Add {SUP, C_Parent, C_Parent.O} to ToProcessList
26.      else if |C_parent.IS| < |O| ∧ |C_parent.IS| < |C_parent.O|:
27.          C_Parent.IS ← IS
28.          Add {ISET, C_Parent, IS} to ToProcessList
29.  C_New ← New Concept({I_i}, O)
30.  PURGE-SUBSETS(ToProcessList)
31.  for each T_i ∈ ToProcessList:                  // Link phase
32.      if    T_i.Type = SUP:
33.          Remove T_i.Concept from C_Base.Parents
34.          Add T_i.Concept    to C_New.Parents
35.      else if    T_i.Type = ISET:
36.          C_Parent ← INSERT(T_i.Concept, ∅, T_i.O)
37.          Add C_Parent to C_New.Parents
38.  Sort C_New.Parents in order of decreasing |O|
39.  Add C_New to C_Base.Parents in order of decreasing |O|
40.  return C_New
```

Algorithm 4. The QuICL algorithm

Table 2. Data set and lattice characteristics. $|\mathcal{O}|$ is number of objects, $|\mathcal{I}|$ is the number of items, and $|\mathcal{L}|$ is number of concepts. Average degree is the average number of concepts in the upper cover of each given concept. Maximum degree is the maximum number of concepts in the upper cover of any concept

Data Set	|O|	|I|	Density	Min Supp	|L|	Avg Deg	Max Deg	H
Chess	3,196	76	0.4933	95%	74	2.64	8	5
				85%	1,885	4.40	13	8
				75%	11,525	5.49	20	11
				65%	49,240	6.17	24	13
				55%	192,863	6.85	27	15
Mushroom	8,124	120	0.1933	50%	45	1.93	9	5
				30%	427	3.00	21	9
				10%	4,897	3.84	31	14
				0%	238,709	5.71	33	22
Pumsb	49,046	7,117	0.0104	95%	110	2.51	12	4
				85%	8,513	5.17	19	9
				75%	101,047	7.02	21	12
Pumsb*	49,046	7,117	0.0071	50%	248	2.82	18	8
				40%	2,610	4.22	29	12
				30%	16,154	5.14	36	15
T10I4D100k	100,000	1,000	0.0101	0.50%	1,073	1.68	569	5
				0.10%	26,806	3.27	796	10
				0.05%	46,993	3.10	832	10
				0.01%	283,397	2.81	846	11
				0.00%	2,347,374	4.29	846	14
T25I10D10k	9,219	1,000	0.0278	1.00%	5,582	3.58	919	10
				0.50%	23,393	3.68	982	12
				0.10%	209,436	2.63	996	13
				0.05%	576,020	2.74	996	14
				0.00%	2,557,927	4.30	996	17
T25I20D100k	100,000	10,000	0.0028	0.50%	27,067	4.09	2,131	12
				0.10%	150,970	4.64	4,325	14
				0.05%	212,765	4.51	4,703	14
				0.01%	3,519,933	3.67	4,889	18

Seven public data sets were used as the benchmarks consisting of Mushroom, Chess, Pumsb, Pumsb*, T10I4D100k, T25I10D10k, and T25I20D100k. The Mushroom data set contains characteristics of various species of mushrooms. The Chess data set is sequence of steps recorded for a game of chess. Pumsb data set contains census data. The Pumsb* data set is the Pumsb data set with removal of items whose support is greater than or equal to 80%. The T10I4D100k, T25I10D10k, and, T25I20D100k are synthetic data sets generated by the IBM Synthetic Data Generator. It generates data sets that emulate retail transactions according to a set of input parameters (e.g., number items, number transactions, average transaction length). The Mushroom, Chess, Pumsb, Pumsb* and T10I4D100k data sets were downloaded from the University of Helsinki Frequent Item Set Mining Data set Repository. The T25I10D10k and T25I20D100k data sets were downloaded from the High Performance Computing Laboratory of The Institute of Information Science and Technologies, Pisa, Italy. The characteristics of these data sets together with characteristics of their generated concept lattices are given in

Table 2. Density is calculated by |R| / (|O| × |I|) where |R| is the total number of items for all objects found in the data set, |O| is number of objects, and |I| is the number of items. As can be observed in Table 2, the Chess and Mushroom data sets are dense, and the remaining data sets are sparse.

All benchmarks are executed on an Intel Core 2 Duo CPU at 2.99 GHz with 3.0 GBs memory running Windows XP 2002. All benchmarks are executed using the Java JRE version 1.5. To measure the execution time, all algorithms were instrumented to time the lattice construction functions exclusive of any I/O time. All reported execution times are in seconds. Furthermore, to minimize error all reported execution times are the average of five separate measurements for each test case.

To measure the memory usage, all algorithms pause before termination. The memory usage is then obtained from the "Mem Usage" field of the Windows Task Manager dialog at the time of pause. Thus, the memory usage includes space consumed by the Java virtual machine, class files of each algorithm, and the heap allocation. All reported memory usages are in megabytes.

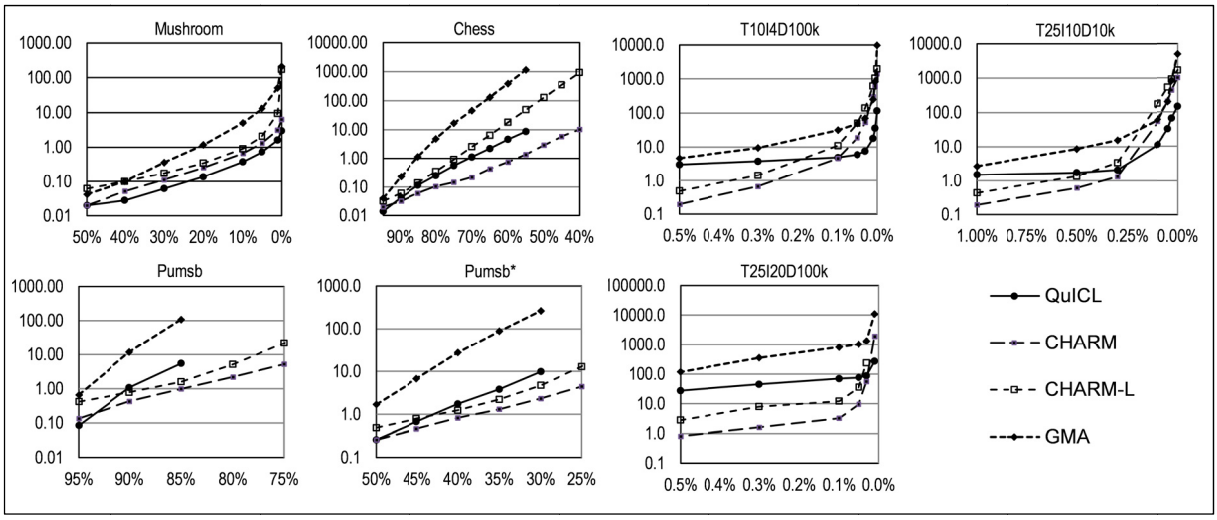

Figure 8. Comparison of algorithm execution times

4.1 Comparison of Algorithm Execution Time

The runtime results of executing the algorithms are given in Figure 8. All data points are the average of five separate test executions of the same test case. Ascending item support order is used for QuICL for all data sets and GMA on sparse data sets. Descending order is used for GMA on dense data sets. All other algorithms are unsorted since sorting is an integral part of the algorithm. All times are in seconds. The plots display seconds on the vertical axis and minimum support on the horizontal axis.

GMA is generally slower than QuICL and CHARM by an order of magnitude. GMA diverges further for small minimum supports. There are two factors that impact the performance of the GMA algorithm. First, each item insertion intersects the next item's object ids with a large portion of the concepts currently in the lattice. GMA visits all concepts in ascending support order from the top of the lattice through the lowest generator. Thus, GMA visits and intersects more concepts than needed. QuICL uses the lattice structure to navigate to a more limited subset of concepts thereby reducing the number of intersections. CHARM performs pruning on its IT tree as its means to limit the number of intersections. Second, GMA links each new concept into the lattice by searching for parents. While this search is restricted to generated and modified concepts, the number can still be excessive. QuICL, on the other hand, will navigate the lattice structure. CHARM-L uses set operations on sets of generator concepts associated with each concept to identify and link parent concepts.

CHARM provides the best performance for the Chess, Pumsb, and Pumsb* data sets. As the support is lowered, the performance gain often exceeds an order of magnitude over the other algorithms. These results are expected since CHARM does not derive the upper covers. It only identifies the closed FI sets. Furthermore, CHARM uses a difference based representation for the object ids below the first level in its IT tree. Since these data sets contain a number of items having large object id sets (e.g., greater than 100), this difference based representation provides a real gain in both in memory and runtime execution. The CHARM algorithm provides the best

performance for the T10I4D100k, T25I10D10k, and T25I20D100k data sets, but only for supports greater than 0.1%, 0.3%, and 0.02% respectively. For these data sets, these supports are relatively high (i.e., will produce only a small fraction of all possible FI sets) and equate to an absolute support of 100, 28, and 20 respectively. As the support is reduced below these points, CHARM is outperformed by QuICL. For Mushroom, a dense dataset, CHARM is outperformed by QuICL over all supports.

CHARM-L exhibits performance along the lines of CHARM but degrades as the support is lowered and the number of concepts increases. For dense data sets the degradation can readily diverge in excess of an order of magnitude. For the sparse data sets, the divergence is between a factor of two (e.g., T25I10D10k) and a factor of five (e.g., T25I20D100k). These results are expected since the CHARM-L is an extension to CHARM that additionally derives the upper covers.

QuICL provides the best overall performance for constructing iceberg lattices. It is only outperformed by CHARM-L on the Pumsb and Pumsb* data sets and for the T10I4D100k, T25I10D10k, and T25I20D100k only at relatively high supports. The Pumsb and Pumsb* are data sets that contain items with very large object id sets (e.g., greater than 10,000) and thus CHARM-L is sufficiently benefiting from its difference based representation to maintain a lead. The gain in performance of CHARM-L over QuICL on Pumsb* is approximately a factor of two. CHARM-L does outperform QuICL at relatively high supports on the T10I4D100k, T25I10D10k, and T25I20D100k data sets. However the gain is limited to a few seconds (e.g., three seconds on T10I4D100k at $0.5\%_{supp}$ and one second on T25I10D10k at $1.0\%_{supp}$), although the T25I20D100k exhibits a gain of 60 seconds at $1.0\%_{supp}$. In all cases, the gain quickly turns into a large loss as the support is lowered (e.g., loss greater than 1,500 seconds on T25I10D10k at $0.0\%_{supp}$ and 1,800 seconds on T10I4D100k at $0.0\%_{supp}$). At low supports QuICL outperforms CHARM-L by an excess of an order of magnitude on the Mushroom, T10I4D100K, T25I10D10k, and T25I20D100k data sets, and a factor greater than five for Chess.

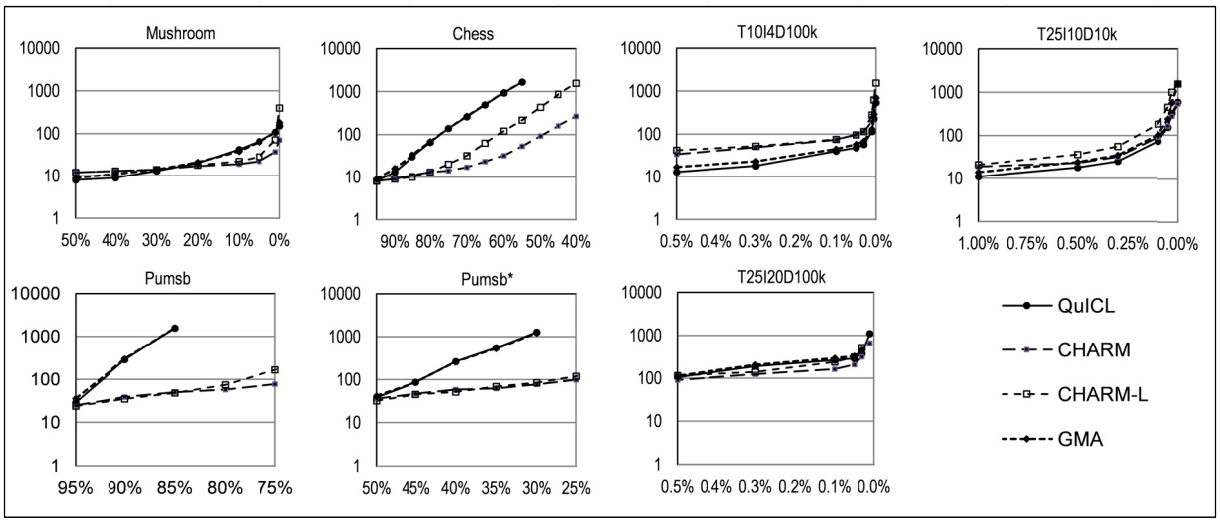

Figure 9. Comparison of algorithm memory usage

4.2 Comparison of Algorithm Memory Usage

The memory results of executing the algorithms are given in Figure 9. All memory measurements were obtained from the "Mem Usage" field of the Windows Task Manager dialog upon termination of the algorithm. Ascending item support order is used for QuICL for all data sets and for GMA on sparse data sets. Descending order is used for GMA on dense data sets. All other algorithms are unsorted since sorting is an integral part of the algorithm. All measurements are in megabytes (MBs). The plots display MBs on the vertical axis and minimum support on the horizontal axis.

QuICL and GMA exhibit similar memory usage for small lattices (i.e., less than 200,000 concepts) and diverge a marginal amount for larger lattices (i.e., greater than 500,000). For both of these, the number of object id entries stored in the lattice is a major consumer of memory. For small lattices on dense data sets the space for object ids can account for more than 95% of the memory consumed. As the lattice becomes large, the overhead of the concepts will become a dominant term. On large lattices, memory for the concepts, excluding object ids and

items, can account for 38% (e.g., T10I4D100k at 0.01%$_{supp}$ using QuICL) to 72% (e.g., T25I10D10K at 0.0%$_{supp}$ using QuICL) of memory. Furthermore, for large lattices the number of parent-child links account for another 15%.

CHARM-L provides a reduction in memory by not retaining object ids in its lattice. However, CHARM-L does maintain object ids in its IT tree. These entries are dynamically constructed during the traversal of the CHARMS IT tree and discarded upon completion of a branch. The memory consumed for each concept in the CHARM-L lattice is about three times the memory consumed by the concepts of QuICL. Due to very different approaches, the reduction or gain in memory usage when compared against QuICL is varied. CHARM-L exhibits the best memory usage on the Pumsb, Pumsb*, and chess data sets. These data sets contain large object id sets. Thus the difference based representation is providing significant reduction in memory usage. CHARM-L has comparable memory consumption on the Mushroom and T25I20D100K data sets, but a loss around a factor of three on the T10I4D100k and T25I10D100k data sets. For the later two datasets, the overhead to represent a concept is degrading the memory consumption as these involve very large lattices containing concepts whose cardinality of extent is small.

The CHARM algorithm does not construct a lattice. As such, its memory consumption is for processing its IT tree and construction its list of FI sets. Since CHARM-L is an extension to CHARM that constructs a concept lattice the memory consumption of CHARM is expected to be less than CHARM-L. This is indeed the case. However, the difference between the exhibited memory consumption of CHARM-L and CHARM should not be interpreted to be the memory for the lattice, since memory used for the IT tree is released upon processing a branch and may be reused for the lattice.

5. Conclusions

This paper has presented the QuICL algorithm, used to incrementally to construct an iceberg concept lattice. Its objective was to offer a lattice based algorithm whose overall performance in constructing a lattice is comparable to algorithms used for ARM. Furthermore, it was proposed that such algorithm would provide gains relative to the overall task of ARM. This objective has been met. The performance of QuICL is on the order of CHARM, a leading algorithm to mine FI sets, and QuICL additionally derives the upper covers. The lattices constructed by QuICL are of a form whereby association rules can be directly read and a basis can be readily generated. As such, the Stumme et al. (2001) algorithms can be used to extract the Duquenne-Guigues basis and Luxemburger basis. Thus, it is postulated that QuICL provides a significant gain in the overall task of ARM. QuICL enables the generation of association rules whose size is constrained to a number that can be exploited by the end user. Beyond this, it was proposed that new efficient algorithms to construct concept lattices may present a contribution to formal concept analysis. QuICL provides an order of magnitude gains in performance over GMA, an often cited incremental lattice construction algorithm. It is noted that GMA provides good performance on data sets whose density is less than 0.10. QuICL provides excellent performance on both sparse and dense data sets. For example, on the T10I4D100k, a sparse data set, QuICL provides a gain over GMA of two orders of magnitude (e.g., less than 120 seconds verses near 10,000 seconds at 0.0%$_{supp}$). On Mushroom, a dense data set, the same two order magnitude gain is realized (e.g., three seconds verses 200 seconds at 0.0%$_{Supp}$), likewise on Chess (e.g., less than ten second verses over 1,000 seconds at 55%$_{supp}$). Literature has noted there is no known "best" algorithm for lattice construction and that each algorithm demonstrates different performance on different data sets, yet QuICL provides the best all-around performance.

QuICL differs from past lattice construction algorithms in three notable ways. First, QuICL is a pure incremental lattice construction algorithm. That is, its sole data structure driving its processing is the lattice. Many other algorithms are driven by some other data structure and separately construct the lattice, although as an integral sub-task. For example GALICIA-T (Valtchev et al., 2002) uses a trie, Nourine and Raynaud (2002) uses its lexicographic tree, and CHARM-L uses its IT tree as it primary data structure. By being a pure incremental lattice construction algorithm, the foundation of QuICL is based solely on FCA. Additional theory derived from FCA may provide for further improvements to QuICL. Second, QuICL has recognized that it is sufficient to store an item at only its maximal position. There is no need to include the item in all descendent concepts. Thus for a given item insertion, the only modified concept will be the one where an item is inserted. This eliminates the need to modify a substantial number of concepts thereby significantly improving performance. Third, in comparing QuICL to GMA, both identify generator concepts. For QuICL, the generators are the base concepts that do not have a parent whose extent is a superset of the incoming object id set. QuICL differs from GMA in that it identifies the lowest generator concepts first, whereas GMA first identifies the highest. Thus, QuICL eliminates the step to validate a candidate generator is indeed a generator, a potentially time consuming process. Furthermore, since QuICL approaches the lattice from the bottom up, its recursion directly identifies the parent

concepts. This eliminates the very expensive task of searching for parents incurred by GMA. This task is exacerbated on dense data sets. Given this discussion and the results presented herein, it is postulated that QuICL is the "best known" all around incremental lattice construction algorithm. An evaluation against a broader set of data sets and other lattice construction algorithms is needed to validate this claim.

An issue for QuICL, as well as FI set miners and lattice construction algorithms in general, is memory consumption. The exponential nature of the problem can quickly exhaust available memory. All algorithms used in this study failed to produce a complete lattice for four of the seven data sets. In each case the failure was due to memory constraints. CHARM and CHARM-L was able to process lower supports than QuICL and GMA. Further investigation into CHARM's difference based representation may shed light on additional improvements to QuICL.

Another issue for QuICL is seen in the runtime execution on the Chess, Pumsb, and Pumsb* datasets. These datasets contain items having very large object id sets (e.g., on Chess half the items exceed 1,500 object ids, on Pumsb some items have over 40,000 object ids). As a result many concepts have large extents. The time to perform intersections for these large sets is a considerable portion of execution time. Again, further investigation into CHARM's difference based representation may provide insights in addressing this issue.

An enhancement to QuICL that addresses both the memory consumption and large object ids sets is to exploit the lattice property: if $O_i \in$ extent of concept C_1 then $\forall\ C_2 \mid C_2 > C_1$, $O_i \in$ extent of C_2. Thus, an $O_i \in O$ of concept C_2 does not need to be physically recorded in a concept if there exists a concept C_1 such that $C_1 < C_2$ and $O_i \in O$ of concept C_1. Instead, a given object O_i need only be recorded in a concept at its minimal position (i.e., highest position in the inverted lattice). This forms a compressed lattice structure. The savings in memory is at the cost of a penalty in performance. Analysis of this tradeoff is a subject of future research.

QuICL was developed in 2009 (Smith). Since then a recent algorithm to generate an iceberg lattice for ARM has been developed (Szathmary et. al. 2011). Szathmary et. al. include an empirical evaluation of their algorithm, Snow-Touch, against CHARM-L using a number of the same datasets used to evaluate QuICL. A comparison of the results of Szathmary et al. against the results presented herein indicates that QuICL will exhibit leading performance. For example, on the Mushroom dataset at $25\%_{supp}$ the reported execution time of Snow-Touch is around 40 seconds whereas QuICL is under a tenth of a second (at $0\%_{supp}$ QuICL is around 3 seconds). Furthermore, the times of QuICL are better than the reported times for Snow-Touch on the Chess and T2510D10k datasets.

References

Agrawal, R., Imieliński, T., & Swami, A. (1993, June). Mining association rules between sets of items in large databases. In *ACM SIGMOD Record* (Vol. 22, No. 2, pp. 207-216). ACM.

Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB* (Vol. 1215, pp. 487-499).

Choi, V. (2006). Faster algorithms for constructing a Galois/concept lattice. In *Proceedings of SIAM Conference on Discrete Mathematics*. Victoria, British Columbia, Canada.

Duquenne, V., & Guigues, J. L. (1986). Famille minimale d'implications informatives re'sultant d'un tableau de donne'es binaires. *Mathe'matiques et Sciences Humaines, 24*(95), 8-18.

Ganter, B., Stumme, G., & Wille, R. (2005). *Formal Concept Analysis: Foundations and Applications*. Springer-Verlag.

Ganter, B., & Wille, R. (1997). *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, New York, Inc.

Godin, R., Missaoui, R., & Alaoui, H. (1995). Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence, 11*(2), 246-267. http://dx.doi.org/10.1111/j.1467-8640.1995.tb00031.x

Han, J., & Kamber, M. (2006). *Data Mining Concepts and Techniques* (2nd ed.). Morgan Kaufmann.

Kuznetsov, S. O., & Obiedkov, S. A. (2002). Comparing performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence, 14*(2/3), 189-216. http://dx.doi.org/10.1080/09528130210164170

Lakha, L., & Stumme, G. (2005). Efficient mining of association rules based on formal concept analysis. *Lecture Notes in Computer Science, 3626*, 180-195. http://dx.doi.org/10.1007/11528784_10

Lindig, C., & Datensystene, G. (2000). Fast concept analysis. *Working with Conceptual Structures – Contributions to ICCS 2000*, 152-161. Shaker Verlag.

Lucchese, C., Orlando, S., & Perego, R. (2006). Fast and memory efficient mining of frequent closed itemsets. *IEEE Transactions on Knowledge and Data Enginering, 18*(1), 21-36. http://dx.doi.org/10.1109/TKDE.2006.10

Luxenburger, M. (1991). Implications partielles dans un contexte. *Mathe'matiques, Informatique et Sciences Humaines, 29*(113), 35-55.

Maddouri, M. (2005). A formal concept analysis approach to discover association rules from data. In *Proceedings of the 6th International Conference on Concept Lattices and Their Applications* (pp. 10-21). Olomouc, Czech Republic.

Martin, B., & Eklund, P. W. (2008). From Concepts to Concept Lattice: A Border Algorithm for Making Covers Explicit.In *ICFCA 2008. LNCS (LNAI)* (Vol. 4933, pp. 78-89). Springer, Heidelberg.

Nourine, L., & Raynaud, O. (2002). A fast incremental algorithm for building lattices. *Journal of Experimental & Theoretical Artificial Intelligence, 14*(2&3), 217-227. http://dx.doi.org/10.1080/09528130210164152

Pei, J., Han, J., & Mao, R. (2000). Closet: An efficient algorithm for mining frequent closed itemsets. In *Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (pp. 21-30).

Priss, U. (2006). *A formal concept analysis bibliography*. Retrieved from http://www.upriss.org.uk/fca/bibliography.html

Smith, D. T. (2009). *A Formal Concept Analysis Approach to Data Mining: The QuICL Algorithms*. ProQuest, UMI Dissertations Publishing.

Stumme, G. (2002). Efficient data mining based on formal concept analysis. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications* (pp. 534-546). Springer-Verlag. http://dx.doi.org/10.1007/3-540-46146-9_53

Stumme, G., Bastide, Y., Pasquier, N., & Lakhal, L. (2000). Fast computation of concept lattices using data mining techniques. In *Proceedings of 7th International Workshop on Knowledge Representation Meets Databases* (pp. 129-139).

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., & Lakhal, L. (2001). Intelligent structuring and reducing of association rules with formal concept analysis. In *Proceedings of the Joint German/Austrian Conference on AI: Advances in Artificial Intelligence* (pp. 335-350). Springer-Verlag.

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., & Lakhal, L. (2002). Computing iceberg concept lattices with TITANIC. *Data Knowledge Engineering, 42*(2), 189-222. http://dx.doi.org/10.1016/S0169-023X(02)00057-5

Szathmary, L., Valtchev, P., Napoli A., Godin, R., Boc, A., & Makarenkov, V. (2011). Fast Mining of Iceberg Lattices: A Modular Approach Using Generators. *8th International Conference of Concept Lattices and Their Applications*. Nancy, France.

Uno, T., Kiyomi, M., & Arimura, H. (2004). LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*. Brighton, UK.

Valtchev, P., Godin, R., Missaoui, R., Huchard, M., Napoli, A., Grosser, D., et al. (2008). *Project Galicia*. Retrieved from http://www.iro.umontreal.ca/~galicia/

Valtchev, P., Grosser, D., Roume, C., & Hacene, M. R. (2003). Galicia: An open platform for lattices. In *Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures* (pp. 241-254). Dresden, Germany.

Valtchev, P., Missaoui, R., & Godin, R. (2004). Formal concept analysis for knowledge discovery and data mining: The new challenges. *Lecture Notes in Computer Science, 2961*, 352-371. Springer-Berlin/Heidelberg. http://dx.doi.org/10.1007/978-3-540-24651-0_30

Valtchev, P., Missaoui, R., Godin, R., & Meridji, M. (2002). Generating frequent itemsets incrementally: Two novel approaches based on Galois lattice theory. *Journal of Experimental & Theoretical Artificial Intelligence, 14*(2-3), 115-142. http://dx.doi.org/10.1080/09528130210164198

Valtchev, P., Missaoui, R., & Lebrun, P. (2000). A fast algorithm for building the Hasse diagram of a Galois lattice. In *Proceedings of Colloque LaCIM 2000* (pp. 293-306). Montréal.

Wang, J., Han, J., & Pei, J. (2003). Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 236-245). Washington, D.C.: ACM Press.

Wille, R. (1982). Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets* (pp. 445-470). Dordrecht Boston. http://dx.doi.org/10.1007/978-94-009-7798-3_15

Yahia, S. B., Hamrouni, T., & Nguifo, E. M. (2006). Frequent closed itemset based algorithms: A thorough structural and analytical survey. *ACM SIGKDD Explorations, 8*(1), 93-104. http://dx.doi.org/10.1145/1147234.1147248

Zaki, M. J., & Hsiao, C. (2002). Charm: An efficient algorithm for closed itemset mining. In *Proceedings of SIAM International Conference on Data Mining* (pp. 457-473).

Zaki, M. J., & Hsiao, C. (2005). Efficient algorithms for mining closed itemsets and their lattice structure. *IEEE Transactions on Knowledge and Data Engineering, 17*(4), 462-478. http://dx.doi.org/10.1109/TKDE.2005.60

Notes

Note 1. The overall drop in confidence is derived by 83.4% × 80.0%.

Note 2. Iterating in ascending |I| order effectively performs a top-down breath-first traversal.

Note 3. Density is a measure of the completeness of a data set. For formal context K{$\mathcal{G}$, $\mathcal{O}$, $\mathcal{R}$}, the density of $\mathcal{R}$ = $|\mathcal{R}|$ / ($|\mathcal{G}| \times |\mathcal{O}|$) where $|\mathcal{R}|$ is the total number of items for all objects.

Note 4. Construction of the lattice occurs during its core processing. Alternatively, a lattice can be constructed as subsequent step to identifying closed FI sets. Such approach, however, impacts efficiency (Zaki & Hsiao, 2005).

Note 5. Preliminary tests are executions of the algorithm during development. These tests are not performed under controlled conditions. The timings are given to illustrate the progression during algorithm development.

Note 6. An often used data set for ARM and FCA.

Note 7. The cache is a simple reference to an intersection set from each tuple representing a concept. The intersection sets are discarded between item insertions.

Note 8. Failure to use the appropriate object id sets that are either already present in memory or will be subsequently used in the algorithm, will result in the storage of many additional object ids sets.

Note 9. CHARM and CHARM-L were translated to Java in order to provide a consistent test environment.