2009

A Formal Concept Analysis Approach to Association Rule Mining: The QuICL Algorithms

David T. Smith
Nova Southeastern University, dtsmith@iup.edu

A Formal Concept Analysis Approach to Association Rule Mining: The QuICL Algorithms

by

David T. Smith

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2009

.

We hereby certify that this dissertation, submitted by David T. Smith, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

_____ _____

Jumping Sun, Ph.D. Date
Chairperson of Dissertation Committee

_____ _____

Michael Laszlo, Ph.D. Date
Dissertation Committee Member

_____ _____

Soundararajan Ezekiel, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Edward Lieblein, Ph.D. Date
Dean of Graduate School of Computer and Information Sciences

Graduate School of Computer and Information Sciences
Nova Southeastern University

2009

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements of a Degree of Doctor of Philosophy

A Formal Concept Analysis Approach to Association Rule Mining: The QuICL Algorithms

by
David T. Smith

May 2009

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 exploited by an end user. An alternative to FI mining can be found in formal concept analysis (FCA), a branch of applied mathematics. FCA derives a concept 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 concept lattice whose concepts contain only FI sets. Only three algorithms to construct an iceberg lattice were found in literature. Given that an iceberg concept lattice provides an analysis tool to succinctly identify association rules, this study investigated additional algorithms to construct an iceberg concept lattice. This report presents the development and analysis of the Quick Iceberg Concept Lattice (QuICL) algorithms. These algorithms provide incremental construction of an iceberg lattice. QuICL uses recursion instead of iteration to navigate the lattice and establish connections, thereby eliminating costly processing incurred by past algorithms. The QuICL algorithms were evaluated against leading FI miners and FCA construction algorithms using benchmarks cited in literature. Results demonstrate that QuICL provides performance on the order of FI miners yet additionally derive the upper covers. QuICL, when combined with known algorithms to extract a basis of association rules from a lattice, offer a "best known" ARM solution. Beyond this, the QuICL algorithms have proved to be very efficient, providing an order of magnitude gains over other incremental lattice construction algorithms. For example, on the Mushroom data set, QuICL completes in less than 3 seconds. Past algorithms exceed 200 seconds. On T10I4D100k, QuICL completes in less than 120 seconds. Past algorithms approach 10,000 seconds. QuICL is proved to be the "best known" all around incremental lattice construction algorithm. Runtime complexity is shown to be $O(\ell\, d\, i)$ where ℓ is the cardinality of the lattice, d is the average degree of the lattice, and i is a mean function on the frequent item extents.

Acknowledgements

As a Christian, I believe there is a God that is the creator of the universe and that Jesus Christ is his son. I have learned to trust Christ in times of plenty and in times of lack. I will forever give Him thanks for bringing me through.

I would like to express my thanks to my wife Diana and children Cassandra and Joel. It's been three years of working from early morning to evening, including most weekends. Their love, patience, and support have enabled me to focus on completing my degree. I look forward to re-focusing my time towards our home.

I would like to express my thanks to my father, Dr. Edwin Malcolm Ramsey Smith. His encouragement and support has enabled me to obtain my degree late in life. I am pleased that he will see his youngest son complete his Ph.D.

Finally, I would like to express my thanks to Dr. Jumping Sun, Dr. Michael Laszlo, and Dr. Soundararajan Ezekiel for their careful review and thoughtful input. Their comments are valued and have contributed to the quality of this report. I would like to further acknowledge Dr. Sun for promoting strong research activities during his classes. It was during his Knowledge Discovery in Databases course that the topic of this report was first considered.

Table of Contents

List of Tables

Tables

List of Figures

Figures

List of Algorithms

Algorithms

Chapter 1

Introduction

1.1 Problem Statement and Goal

Association rule mining is the task of identifying meaningful implication rules of the form $X \rightarrow Y$ exhibited in a data set (i.e., relation), 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, Imieliski, & Swami, 1993). The degree to which a rule is meaningful is defined by:

i) *support*, the number of times both X and Y are found 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 sets* (i.e., $X \cup Y$ that meets 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 the frequent item sets from the 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 frequent item sets of even moderate length (Han, & Kamber, 2006). A case in point, considers a data set D composed of a single tuple $\{a_1, a_2, \ldots, a_n\}$. All subsets of the items of D will be frequent. The number of subsets of the items of D will be $2^n - 1$. For all but a small n, there is not sufficient memory or processing cycles for the Apriori algorithm to reach completion. This leads to the problem: develop an efficient algorithm that can extract from a data set the frequent patterns of moderate to long length (e.g., greater than 30).

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, Stumme, & Wille, 2005). It originated in the early 1980's and was first formalized in 1982 (Wille, 1982). It has since inspired numerous publications (Priss, 2006). According to FCA, a *concept* is defined as:

> **Definition 1.1:** 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:
>
> i) $O = \{o \in \mathcal{O} \mid \forall\, i \in I, o\mathcal{R}i\}$ and
>
> ii) $I = \{i \in \mathcal{I} \mid \forall\, o \in O, o\mathcal{R}i\}$,
>
> 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 objects of a concept is called the *extent* of the concept and the set of items is called the *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 from data structures 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* derived for a relation R is depicted in Figure 1.1.

A concept lattice holds a number of interesting properties including:

Property 1.1: 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 1.2: 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 1.3: 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.

A concept lattice can be incrementally constructed using the Godin, Missaoui, and Alaoui (1995) (GMA) algorithm. GMA algorithm inserts the data for the next object into the concept lattice by partitioning all of the concepts in the lattice into three groups: modified, generator, and old. Modified concepts are those whose intent is a subset of the next object's items. Generator concepts are those whose intent intersects the object's

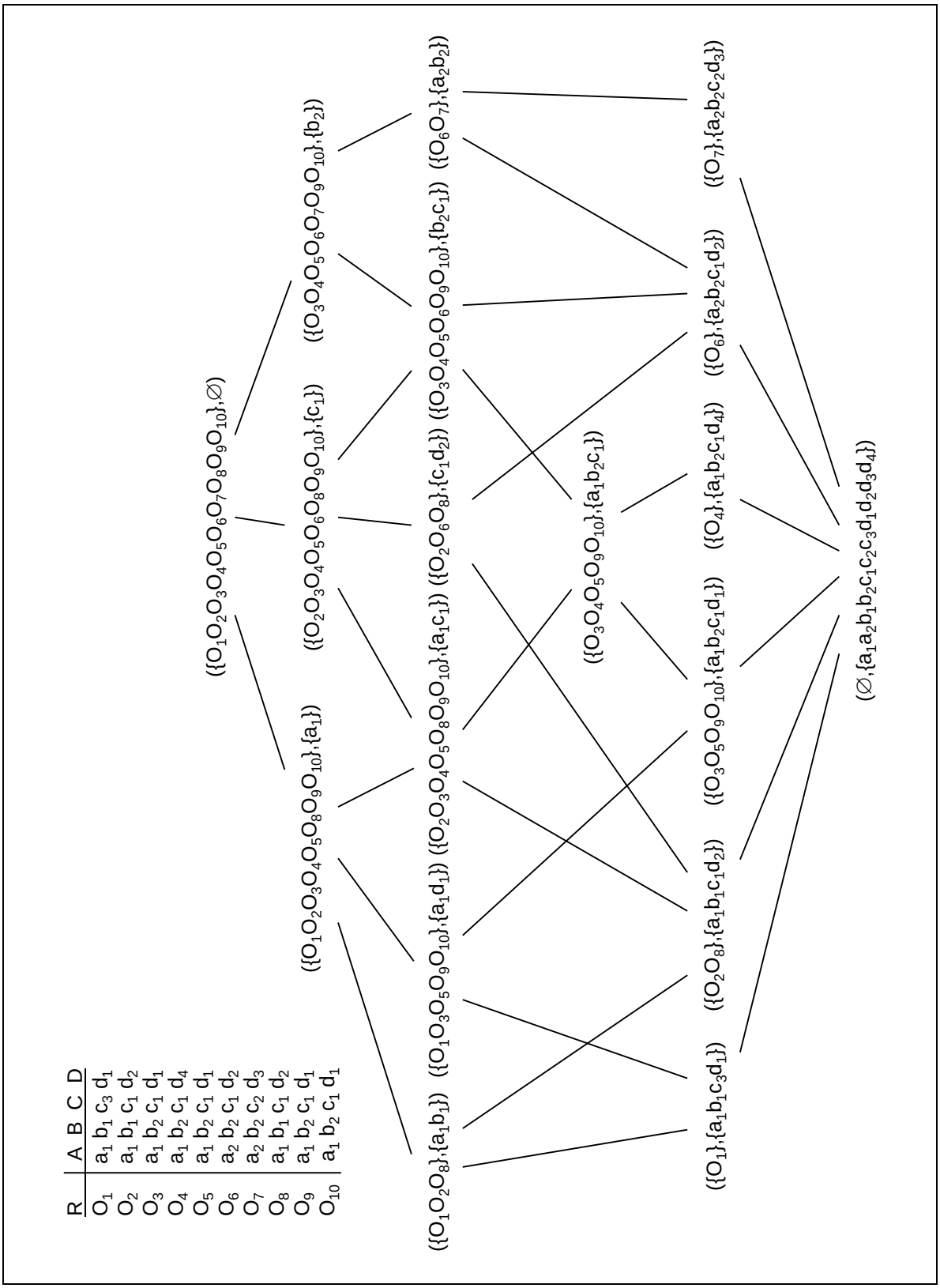


Figure 1.1: Example concept lattice.

items is non-empty and there does not exist an ancestor concept with the same intersection set. Old concepts are those that are neither modified nor generator. These concepts remain unchanged. For each modified concept, the object id of the next object is added to the concept's extent. For each generator concept, a new concept is constructed with an extent equal to the generator's extent union the next object's id, and intent equal to the generator's intent intersect with the next object's items. When generating new concepts, connections are updated accordingly.

Concept lattices are of benefit to association rule mining. 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 (Burdick, Calimlim, & Gehrke, 2001, Stumme, 2002). The concept lattice still contains the necessary and sufficient information to extract association rules and to compute both confidence and support. For example, from the concept ($\{O_1O_2O_8\}$, $\{a_1b_1\}$) of Figure 1.1 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, the GMA algorithm may incur overhead, since these concepts are essentially unnecessary artifacts relative to association rule mining.

An iceberg concept lattice is a concept lattice that contains only the concepts whose support meets a given threshold. For example, Figure 1.2 depicts the concept lattice of Figure 1.1 as an iceberg lattice for both a minimum support threshold of 60% and for 40%. As the threshold is lowered, more detail of the underlying concept lattice is revealed. Iceberg concept lattices provide a model from which association rules can be efficiently mined (Stumme, 2002). Consider the alternate notation of an iceberg lattice depicted in Figure 1.3 that corresponds to the bottom iceberg lattice of Figure 1.2. 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 the two concepts. This notation enables association rules to be directly read from the iceberg lattices. 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, the 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

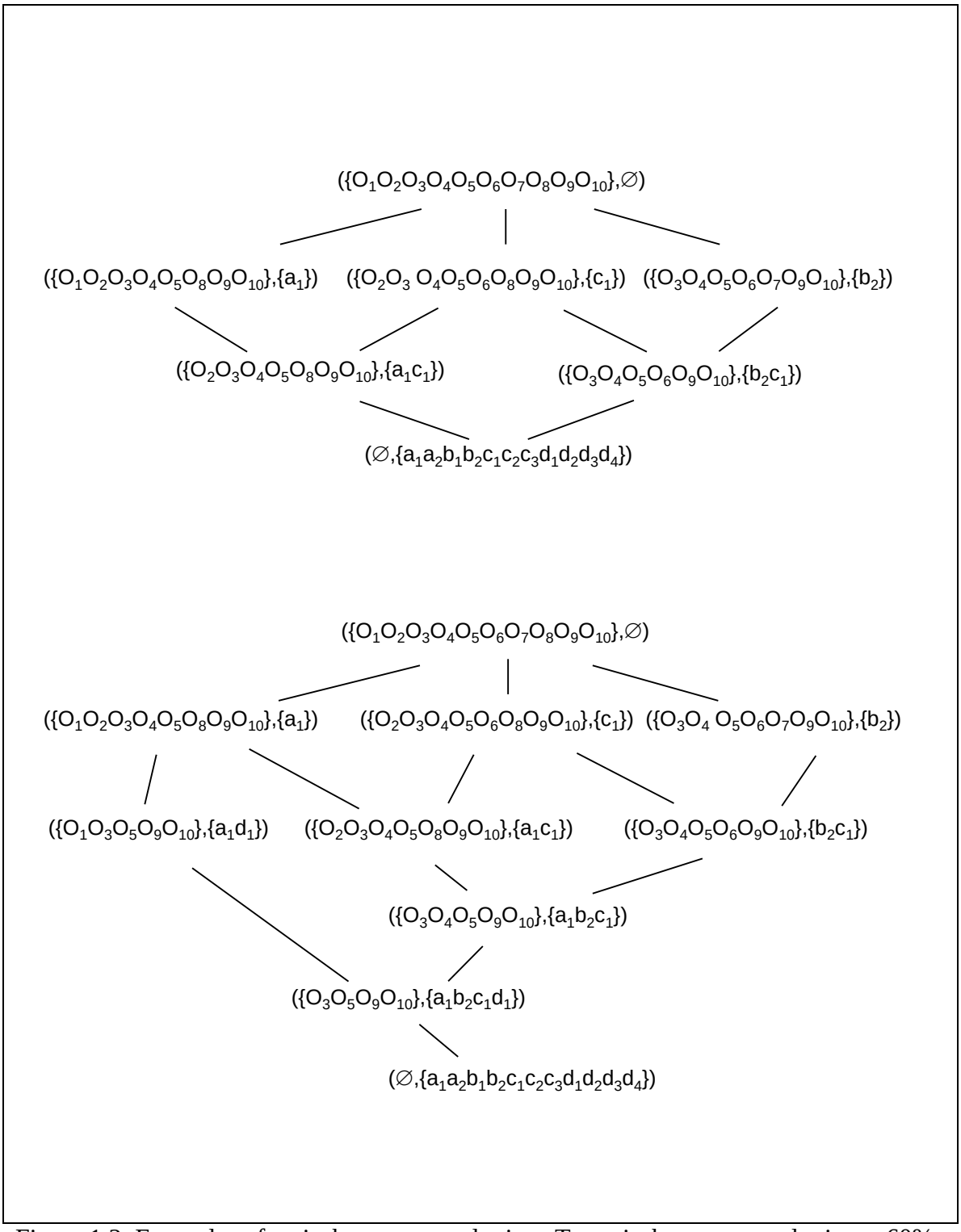


Figure 1.2: Examples of an iceberg concept lattice. Top – iceberg concept lattice at 60% support. Bottom – iceberg concept lattice at 40%. These iceberg lattices were derived from the lattice of Figure 1.1 by discarding concepts not meeting the minimum support threshold.

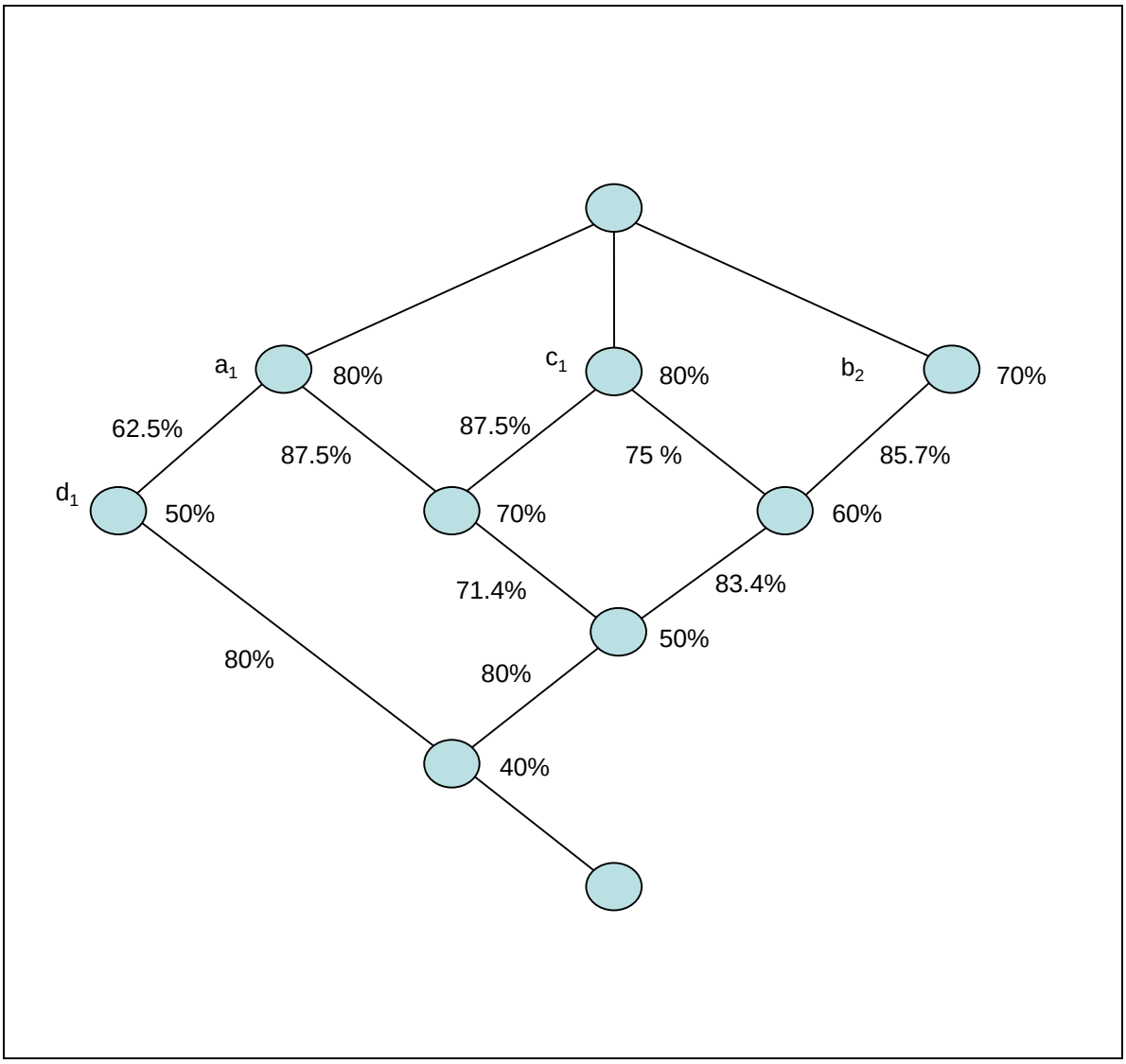


Figure 1.3: Iceberg lattice using an alternate notation. Each concept node is labeled with a percentage representing the support together with any items, if any, for which there does not exist a greater concept containing the item. Edges are labeled with a percentage indicating the effective drop in confidence between the two concepts.

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 1.3. 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[1], d_1 is an ancestor of the node label 40%).

Extracting association rules from a list of frequent item sets (i.e., intents of a set of concepts) 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

[1] The overall drop in confidence is the product of the confidences noted on the edges along the path. In this case 83.4% $\times$ 80.0%, or 66.7%.

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, Taouil, Bastide, Pasquier, and Lakhal (2001b) offer algorithms to traverse and extract the Duquenne-Guigues basis and the Luxemburger basis from an iceberg concept lattice.

Given that an iceberg concept lattice provides an analysis tool to succinctly identify a basis of association rules, additional algorithms to construct an iceberg lattice are needed. This study presents the development of efficient algorithms to construct an iceberg lattice. Its objective; to develop algorithms whose overall performance in constructing a lattice is near to the leading algorithms used for association rule mining. Since a lattice contains more information, marginally slower performance was considered acceptable. In addition to the development of the algorithms, this study:

i) presents theory of formal concept analysis as applied to association rule mining,

ii) includes a detailed analysis of performance characteristics of the developed algorithms through both theory and practice using benchmark databases (e.g., Mushroom, Chess, and T25I10D10k[2]), and

iii) enumerates comprehensive benchmarks comparing the performance of the developed algorithms to other leading algorithms.

1.2 Relevance and Significance

Association rule mining is recognized as an important area within data mining (Pasquier, Bastide, Taouil, & Lakhal, 1999a, Burdick et al., 2001, Zaki & Hsiao, 2002, and Han & Kamber, 2006). It has been applied to a wide range of domains including

[2] Mushroom, Chess, and T25I10D10k are public data sets often used in literature. See Section 4.2.

basket analysis (Agrawal et al., 1993, Brin, Motwani, & Silverstein, 1997a), intrusion detection (Lee, & Stolfo, 1998), database analysis (Huhtala, Karkkainen, Porkka, & Toivonen, 1999, Lopes, Petit, & Lakhal, 2000), geo-spatial decision support (Harms, Li, Deogun, & Tadesse, 2002), medical data analysis (Ordonez, Santana, & Braal, 2000), and organization of web pages on the World Wide Web (Cooley, Mobasher, & Srivastava, 1997). Association rule theory has extended beyond its original domain to include correlations (Brin et al., 1997a), dependency rules (Brin, Motwani, & Silverstein, 1998), episodes (Mannila, Toivonen, & Verkamo, 1997), sequential patterns (Srikant & Agrawal, 1996), and multi-dimensional patterns (Kamber, Han, & Chiang, 1997).

Since the seminal paper by Agrawal et al. (1993), techniques to derive association rules from database have be an active area of research (Agrawal, & Srikant, 1994, Park, Chen, & Yu, 1995, Brin et al., 1997a, Bayardo, 1998, Pasquier, Bastide, Taouil, & Lakhal, 1999b, Dunkel, & Soparkar, 1999, Shenoy, Haritsa, Sudarshan, Bhalotia, Bawa, & Shah, 2000, Pei, Han, & Mao, 2000, Burdick et al., 2001, Stumme, Taouil, Bastide, Pasquier, & Lakhal, 2002, Zaki, & Hsiao, 2002, Wang, Han, & Pei, 2003, Lucchese, Orlando, & Perego, 2004, Uno, Kiyomi, & Arimura, 2004, and Lucchese, Orlando, & Perego, 2006). However, this research has primarily focused on efficient and innovative theory and techniques for the extraction of frequent item sets. As such, they have fallen short of the overall task of mining association rules (Yahia, Hamrouni, & Nguifo, 2006). Key information not generated by these works is the derivation of upper covers of each frequent item set. An *upper cover* of a frequent item set I is a set of frequent item sets U such that $\forall I_u \in U$, $I_u \subset I$ and there does not exist a frequent item set I_2 where $I_u \subset I_2 \subset I$. Upper covers are needed in the production of association rules to efficiently generate

rules from the frequent item sets that are constrained to a number that can be readily exploited by an end user (Zaki, & Hsiao, 2005, Yahia et al., 2006).

Upper covers are provided in FCA. Thus, FCA has been presented as a method to mine association rules (Pasquier, 2000, Stumme, 2002, Valtchev, Missaoui, Godin, & Meridji, 2002a, and Maddouri, 2005). However, most methods involve a complete lattice and therefore are infeasible for mining association rules against large databases. Of the mentioned works, Stumme et al. provides a number of in depth papers on FCA theory that includes iceberg concept lattices (Stumme, Bastide, Pasquier, & Lakhal, 2000, Stumme, Taouil, Bastide, & Lakhal, 2001a, Stumme , Taouil, Bastide, Pasquier, & Lakhal, 2001b, Stumme, 2002 , Stumme et al., 2002, Ganter et al., 2005, and Lakha & Stumme, 2005). Stumme et al. provide a compelling argument for using iceberg concept lattices as a model from which a basis of association rules can be efficiently generated. Zaki (2000) provides similar arguments. Algorithms to generate the rules from an iceberg lattice are provided in Stumme et al. (2001b). However the offered algorithm to construct an iceberg lattice, TITANIC, fails to produce the upper covers.

After a diligent search of literature, only three algorithms to construct an iceberg lattice were found; MAGALICE (Rouane, Nehm, Valtchev, & Godin, 2004), CHARM-L (Zaki, & Hsiao, 2005), and SPROUT (Choi, 2006). MAGALICE builds upon the theory of the GMA algorithm as formalized by Valtchev, Rouane, and Missaoui (2003b). CHARM-L is an extension to the CHARM[3] algorithm (Zaki, & Hsiao, 2002) that constructs a lattice as an adjunct data structure to its core. SPROUT is similar to CHARM-L, except it uses a breadth first search.

---

[3] CHARM is a leading algorithm to mine frequent item sets. This algorithm is discussed further in Chapter 2.

This study presents an investigation into additional algorithms to construct an iceberg concept lattice. The formulation of new algorithms to efficiently construct iceberg concept lattices enables lattice traversal algorithms, such as Stumme et al. (2001b), to efficiently generate a basis of association rules that can be exploited by an end user. Therefore, the formulation of new algorithms to construct an iceberg concept lattice will contribute to the task of association rule mining. Beyond this, new efficient algorithms to construct concept lattices may provide a contribution to the wide set of areas where formal concept analysis is applied.

1.3 Barriers and Issues

The GMA algorithm (Godin et al., 1995) as introduced is not suitable for association rule mining. It is a top-down, level-wise lattice construction algorithm that cannot construct an iceberg lattice, since the supports are not fully determined until the completion of the entire lattice. Thus, concepts that do not meet the minimum support threshold are retained during lattice construction. Such concepts consume memory resources. Furthermore, the representation of concepts contains massive duplication. The same object ids and items are present in multiple concepts. This is evident in the simple concept lattice shown in Figure 1.1. Constructing a concept lattice for a moderate or even small data set could quickly exhaust available memory. Lastly, the GMA algorithm is highly dependent on subset checking and intersection operations. This raises strong concerns on runtime performance.

Chapter 3 will present algorithms to directly construct an iceberg concept lattice. The developed algorithms do not construct concepts that do not meet the minimum support threshold. Furthermore, one algorithm adopts a compressed representation to

eliminate duplicate entries of object ids and items. The algorithms, however, introduce further dependency on set operations including set difference, union, and intersection. Thus, a major challenge was developing techniques to perform fast set operations and caching of interim results. In addition, heuristics that have the potential to reduce the number of times set operations were investigated and applied. For example, insertions of items in ascending support order resulted in reducing the number of intersections performed.

1.4 Elements, Hypotheses, Theories, or Research Questions to be Investigated

The concept lattice is an elegant and well behaved data structure. It is elegant in the sense it provides concise organization of the relationships between a set of objects and a set of items. It is well behaved in that the identical data structure will be constructed regardless of the order in which the data is processed. B-trees, itemset-tidset trees of the CHARM algorithm[4], and frequent pattern trees of the CLOSET algorithm[5] on the other hand are not well-behaved. Variations of order in which data is processed may result in the construction of different structures, although the properties of the given data structure are preserved. Given the elegance and well behavior of a concept lattice, it was hypothesized that an iceberg concept lattice based algorithm will provide gains in association rule mining and will be effective in mining frequent items sets. It was expected that such algorithm will readily construct a concept lattice for a wide range of data sets and will prove to be a viable approach. It was expected that the algorithm will exhibit the same or slightly better performance with respect to memory utilization of the

[4] CHARM is a leading algorithm to mine frequent item sets. This algorithm is discussed further in Chapter 2.
[5] CLOSET is another leading algorithm to mine frequent item sets.

other leading algorithms. Furthermore, it would be resilient against variations of data characteristics and input order.

It was expected that a lattice based algorithm will exhibit runtime performance on the order of leading algorithms to mine frequent item sets, but will probably be slower due to greater dependencies on intersection, union, and set difference operations. However, the output of the proposed algorithm contains more information. A lattice provides the upper covers needed to derive a basis of association rules. Given this, it was hypothesized that the proposed algorithm will have significant gains relative to the overall task of association rule mining.

1.5 Limitations and Delimitations of the Study

An in depth analysis of the statistical properties of the various data sets used as benchmarks measures of performance were not produced, since this is not the focus of this study. Such information can be found in other works (Valtchev, Grosser, Roume, & Hacene, 2002a, Wang et al., 2003, Palmerini, Orlando, & Perego, 2004, and Zaki, & Hsiao, 2005).

In a survey comparing the performance of lattice construction algorithms, Kuznetsov and Obiedkov (2002) note a number of difficulties when attempting to perform empirical benchmarks. Problems include:

i) difficulties understanding crucial details,

ii) description of underlying data structures are often omitted,

iii) algorithms exhibit different behavior on different data sets, and

iv) different choices in programming languages impede meaningful comparisons.

To avoid these problems, this study only performed benchmarks against other algorithms that were well understood and readily implemented, or had an implementation available from the authors that is either written in Java or readily translated to Java. This enabled a common environment for performing benchmarks and thereby minimized the chances for introducing error. CHARM and CHARM-L are written in C++ and available from the authors. CHARM and CHARM-L was readily translated into Java. The MAGALICE algorithm is part of the Galicia framework (Valtchev et al., 2003a). Galicia is written in Java and is available as open source from the authors.

1.6 Definition of Terms

Ancestor Concept – an ancestor of a concept C_1 is any concept C_a such that $C_1 < C_a$. ($C_1 > C_a$ in an inverted lattice)

Anti-chain – a set of concepts that are mutually incomparable. That is an order > does not exists between any two concepts in the set. With a concept lattice, an anti-chain is formed by any horizontal cut through the lattice such that all remaining concepts are either < or > any concept in the anti-chain.

Association Rule – a meaningful implication rule of the form $X \rightarrow Y$ exhibited in a data set (i.e., relation), where X and Y are subsets of the items and $X \cap Y$ is $\varnothing$.

Association Rule Mining – the task of identifying association rules exhibited in a data set.

Basis – a minimal set of association rules that conveys all derivable association rules without loss of information.

Chain – a set of concepts that are mutually comparable. There exists an order > between any two concepts in the set.

Child Concept – a child for a concept C_1 is any concept C_c such that $C_c < C_1$ and there does not exist a concept C_2 such that $C_c < C_2 < C_1$ ($C_c > C_1$ and $C_c > C_2 > C_1$ in an inverted lattice). Operator $\succ$ in $C_1 \succ C_c$ indicates that C_c is a child of C_1. Also known as a *lower cover concept*, or an *immediate successor*.

Closed Item Set – an item set whose closure yields the same set. For an item set I the closure operation is defined as $f \circ g$ (I) where g is a function that identifies the set of objects that have a given set of items and f is a function that identifies the set of items that have a given set of objects.

Closed Set – a set derived upon performing a closure operation on some set. A set is said to be closed if the closure operation derives no further elements. That is, the closure operation derives the original set of elements.

Closure – the set derived upon performing a closure operation on some set. See Closed Set.

Closure Operation (or Function) – a set operation (or function) that derives additional elements resulting in a closed set. A closure operation, denoted as $''$, must be:

i) monotonic (i.e., $X \subseteq Y \rightarrow X'' \subseteq Y''$),

ii) extensive (i.e., $X \subseteq X''$), and

iii) idempotent (i.e., $(X'')'' = X''$).

With respect to FCA, the closure operation for an item set I is defined as $f \circ g$ (I) where g is a function that identifies set of objects that have a given set of items and f is a function that identifies set of items that have a given set of objects. Dually, the closure operation for an object set O is defined as $g \circ f$ (O).

Complete Lattice – a lattice is said to be complete iff there exists a meet and join concepts for any two concepts within the lattice.

Concept Lattice – a complete set of formal concepts derived from a formal context together with connections (i.e., edges) between any two concepts C_1 and C_2 for which an order $<$ exists and there does not exist a concept C_3 for which $C_1 < C_3 < C_2$. Also known as a *Galois lattice* (Barbut, & Monjardet, 1970).

Confidence – a measure of the degree to which an association rule is meaningful. For an association rule $X \rightarrow Y$, confidence is derived by the number of times both X and Y occurs in a data set relative to the number of times X occurs.

Data Set – an organized set of information on a set of entities of the same type. With respect to data mining, a data set is a set of objects where each object has a set of attributes. Also known as a *relation* in relational database theory.

Degree of a Lattice – the maximal number of concepts in any upper cover or lower cover within the lattice. The degree of a lattice, expressed as $deg(\mathcal{L})$, is the maximum of $(\{|Cov^{l}(c)| \mid c \in \mathcal{L}\} \cup (\{|Cov^{u}(c)| \mid c \in \mathcal{L}\})$.

Dense Data Set – a data set with a large number of items per object and a limited number of distinct items.

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

Descendent Concept – an ancestor of a concept C_1 is any concept C_d such that $C_d < C_1$. ($C_1 < C_d$ in an inverted lattice)

Extent – the set of objects having a given set of items. With respect to FCA, the extent of a concept is the set of objects.

Formal Concept – given a formal context $K\{\mathcal{I}, \mathcal{O}, \mathcal{R}\}$, a formal concept is a pair of sets $O \subseteq \mathcal{O}$ and $I \subseteq \mathcal{I}$ iff:

i) $O = \{o \in O \mid \forall\, i \in I, o\mathcal{R}i\}$ and

ii) $I = \{i \in I \mid \forall\, o \in O, o\mathcal{R}i\}$,

where $o\mathcal{R}i$ denotes object o has item i in relation $\mathcal{R}$.

Formal Concept Analysis – a branch of applied mathematics that derives theory from the definition of formal concepts.

Formal Context – a tuple $K\{\mathcal{I}, \mathcal{O}, \mathcal{R}\}$ where $\mathcal{I}$ is a set of items, $\mathcal{O}$ is a set of objects, and $\mathcal{R}$ is a relation such that $\mathcal{R} \subset \mathcal{I} \times \mathcal{O}$. The relation $\mathcal{R}$ identifies the items contained in each object.

Frequent Item Set – an item set whose support meets a specified threshold.

Horizontal Representation – data within a data set is organized as a list of the objects with each object listing its items.

Iceberg Concept Lattice – the set of concepts of a concept lattice, together with their edges, whose cardinality of extent meet a specified minimum support threshold.

Infimum (inf) – the greatest lower bounds of a set. In terms of a concept lattice $\mathcal{L}$ the infimum is the concept $C_{inf} = C \in \mathcal{L} \mid C < C' \;\forall\; C' \in \mathcal{L} \wedge C \neq C'$. It is the top concept in a concept lattice (bottom concept in an inverted lattice).

Intent – the set of common items between a set of objects. With respect to FCA, the intent of a concept is the set of items.

Item Set – a set of items.

Join – the least common ancestor of two or more concepts. For any set of concepts $\{(O_1, I_1), (O_2, I_2), \ldots, (O_n, I_n)\}$, the lattice will contain concept $(\cap^{n}_{i=1} O_i, \text{closure}(\cup^{n}_{i=1} I_i))$. Such concept is the join of $(O_1, I_1), (O_2, I_2), \ldots, (O_n, I_n)$.

Join Semi-lattice – a lattice for which only the join between any set of concepts is preserved within the lattice.

Line Diagram – a diagram depicting a concept lattice. Also known as a *Hasse diagram* (Pemmaraju, & Skiena, 1990).

Lower Cover – a lower cover of an item set I is a set of item sets U such that $\forall I_l \in U$, $I_l \supset I$ and there does not exist an item set I_2 where $I_l \supset I_2 \supset I$. With respect to FCA, a lower cover is the set of child concepts. Function $Cov^{\downarrow}(c)$ is said to map concept c to its lower cover. Also known as the set of *immediate successors*.

Maximal Concept – the greatest concept in terms of order > that contains a given item I_i. Function $\nu(I_i)$ is said to map item I_i to its maximal concept. Also known as the *item concept* of item I_i.

Maximal Frequent Item Set – a frequent item set for which there does not exist another item set whose items are a superset.

Minimal Concept – the smallest concept in terms of order > that contains a given object O_i. Function $\mu(O_i)$ is said to map object O_i to its minimal concept. Also known as the *object concept* of object O_i.

Meet – the greatest common descendent of two or more concepts. For a set of concepts $\{(O_1, I_1), (O_2, I_2), \ldots, (O_n, I_n)\}$, the lattice will contain concept $(\text{closure}(\cup^n_{i=1} O_i), \cap^n_{i=1} I_i)$. Such concept is the meet of $(O_1, I_1), (O_2, I_2), \ldots, (O_n, I_n)$.

Meet Semi-lattice – a lattice for which only the meet between any set of concepts is preserved within the lattice.

Order > – An order > is said to exist between any two concepts $C_1 = (O_1, I_1)$ and $C_2 = (O_2, I_2)$ such that $O_1 \supset O_2$ (or $I_1 \subset I_2$).

Parent Concept – a parent for a concept C_1 is any concept C_p such that $C_1 < C_p$ and there does not exist a concept C_2 such that $C_1 < C_2 < C_p$ ($C_1 > C_p$ and $C_1 > C_2 > C_p$ in an inverted lattice). Operator $\prec$ in $C_1 \prec C_p$ indicates that C_p is a parent of C_1. Also known as an *upper cover concept*, or *immediate predecessor*.

Sparse Data Set – a data set with few items per object and a large number of items.

Sub-lattice – a lattice extracted from another lattice. A sub-lattice is formed by selecting a concept together with all of its ancestors (or decedents). If the lattice from which a sub-lattice is extracted is a complete lattice, the sub-lattice will be a complete lattice.

Subsume – between two sets X_i and X_j, if $X_i \subset X_j$ and closure of X_i = closure of X_j then X_j is said to subsume X_i.

Subsumption Check – given a set of sets $\mathcal{X}$ and a new element X_i, a subsumption check ensures that X_i is not subsumed by any $X_j \in \mathcal{X}$.

Support – a measure of the degree to which an association rule is meaningful. For an association rule $X \rightarrow Y$, support is derived by the number of times $X \cup Y$ occurs in a data set. Support can be expressed as an absolute count or as a percentage of the total number of tuples in the data set. Function $\gamma(c)$ is said to map concept c to its support.

Supremum (sup) – the least upper bound of a set. In terms of a concept lattice $\mathcal{L}$ the supremum is the concept $C_{sup} = C \in \mathcal{L} \mid C > C' \; \forall \; C' \in \mathcal{L} \wedge C \neq C'$. It is the bottom concept in the concept lattice (top concept in an inverted lattice).

Trivial Join – a join concept whose extent is ∅ and intent is the set of all items. The trivial join, if present, is the topmost concept in a concept lattice.

Trivial Meet – a join concept whose extent is the set of all objects and intent is ∅. The trivial meet, if present, is the bottom most concept in a concept lattice.

Upper Cover – an upper cover of an item set I is a set of item sets U such that $\forall \; I_u \in U$, $I_u \subset I$ and there does not exists an item set I_2 where $I_u \subset I_2 \subset I$. With respect to FCA, an upper cover is the set of parent concepts. Function $Cov^u(c)$ is said to map a concept c to its upper cover. Also known as the set of *immediate predecessors*.

Vertical Representation – data within a data set is organized as a list of the items with each item listing its objects.

Width of a Lattice – the size of the maximal anti-chain present in the lattice. The width of a lattice, expressed as $w(\mathcal{L})$, = max ({|*a*| | a ∈ {anti-chains in $\mathcal{L}$}}).

1.7 Summary of Background and Problem Statement

Association rule mining is the task of identifying meaningful implication rules of the form $X \rightarrow Y$ exhibited in a data set. It has been applied to a wide range of domains including basket analysis, intrusion detection, database analysis, geo-spatial decision support, medical data analysis, and organization of pages on the World Wide Web. Furthermore, association rule theory has extended beyond its original domain to include correlations, dependency rules, episodes, sequential patterns, and multi-dimensional patterns. Association rule mining is an important area of knowledge discovery in databases and has been an active area of research.

A majority of research on association rule mining has focused on efficient techniques and innovative theory to extract frequent item sets. This is of itself an exponential problem. As such, techniques like candidate generation and frequency

counting may be intractable for even a moderate sized data set. Thus, research has been directed towards the development of efficient algorithms to prune the search space through application of theory and specialized compact data structures. While this research has made significant progress over the last fifteen years, it has focus on only part of the association rule mining problem: mining frequent item sets. In addition to identifying the set of frequent item sets, the upper covers of each frequent item set are needed to generate a set of association rules whose size is constrained to a number that can be exploited by an end user. The identification of upper covers is generally considered to be a worst case quadratic problem in terms of the number of frequent item sets.

An alternative to frequent item set mining algorithms can be found in formal concept analysis, a branch of applied mathematics. Given a formal context $\mathcal{K}$ composed of a set of objects $\mathcal{O}$, a set of items $\mathcal{I}$, and a relation $\mathcal{R} \subset \mathcal{O} \times \mathcal{I}$, formal concept analysis derives a set of concepts where each concept is a pair of sets $O \subseteq \mathcal{O}$ and $I \subseteq \mathcal{I}$ such that $O = \{o \in \mathcal{O} \mid \forall\, i \in I, o\mathcal{R}i\}$ and $I = \{i \in \mathcal{I} \mid \forall\, o \in O, o\mathcal{R}i\}$. Furthermore, between any two concepts $C_1 = (O_1, I_1)$ and $C_2 = (O_2, I_2)$ an order $<$ is said to exists between C_1 and C_2 iff $O_1 \subset O_2$. Thus, the concepts derived from $\mathcal{K}$ can be arranged into a lattice structure by defining a connection between any two concepts C_1 and C_2 for which order $<$ exists and there is no concept C_3 for which $C_1 < C_3 < C_2$. The result is a concept lattice. A concept lattice does, however, include concepts whose cardinality of O does not meet a minimum support threshold and as such may involve an excessive number of concepts. An iceberg lattice is a concept lattice whose set of concepts are restricted to those whose cardinality of O meets a minimum support threshold.

Iceberg concept lattices are of benefit to mining association rules. A concept's intent (i.e., set of I) corresponds to an item set and cardinality of extent (i.e., set of O) corresponds to its support. Furthermore, the definition of a concept embodies the mathematical notion of closure. Thus, nodes of the concept lattice represent only closed item sets, whose cardinality can be orders of magnitude lower than the cardinality of all item sets. The iceberg concept lattice still contains the necessary and sufficient information to extract association rules and to compute both confidence and support, and the connections identifying the upper covers. Furthermore, the alternate notation of an iceberg lattice depicted in Figure 1.3 enables association rules to be directly read from an iceberg lattice. This form of iceberg concept lattice can be readily traversed using the Duquenne-Guigues basis and Luxenburger basis to generate a set of association rules that can be exploited by an end user.

After a diligent search of literature, only three algorithms to construct an iceberg lattice were found; MAGALICE, CHARM-L, and SPROUT. Given that an iceberg concept lattice provides an analysis tool to succinctly identify a basis of association rules, this study investigates additional algorithms to construct an iceberg concept lattice. Formulation of new algorithms to construct iceberg concept lattices will therefore make a contribution to the task of association rule mining. Beyond this, new efficient algorithms to construct concept lattices may provide a contribution to the wide-set of areas where formal concept analysis is applied.

Chapter 2

Review of Literature

2.1 Introduction

Presented in this chapter is an overview of algorithms for association rule mining as well as concept lattice construction. Details of the CHARM (Zaki, & Hsiao, 2002), Valtchev, Missaoui, and Lebrun (2000) post-mining lattice construction , GMA (Godin et al., 1995), GALICIA-T (Valtchev et al., 2002), CHARM-L (Zaki, & Hsiao, 2005), MAGALICE (Rouane et al., 2004), and Valtchev et al. (2003) generic lattice construction algorithms are provided since these have high relevance to the objectives of this study. CHARM, with its latter CHARM-L, is the only known example of a frequent item set miner that has been extended to produce a concept lattice as an integral part of its processing. Post-mining lattice construction is an approach to generate the lattice as a subsequent step to frequent closed item set mining. It, combined with a frequent closed item set mining algorithm, will construct an iceberg lattice. GMA is an often cited incremental lattice construction algorithm that is noted for good performance. GALICIA-T is an adaption of a lattice construction algorithm specifically for association rule mining. MAGALICE is an extended GMA algorithm that constructs only an iceberg lattice. Lastly, the generic construction algorithm provides a concise statement of the tasks to be performed by an incremental lattice construction algorithm.

2.2 Classical Association Rule Mining – Mining of Frequent Item Sets

Since the Apriori algorithm was introduced in 1993, mining of association rules is an active area of research. There have been many proposed improvements to the Approri algorithm including (Agrawal & Srikant, 1994, Park et al., 1995, Brin, Motwani, Ullman, & Tsur, 1997b, Dunkel, & Soparkar, 1999, and Shenoy et al., 2000). These algorithms are effective for mining short frequent patterns, but are not viable for mining data sets involving long sets (i.e., above 10 to 15 items) (Burdick et al., 2001, Han, & Kamber, 2006).

There have been a number of algorithms developed to address the mining of long frequent item sets. Most notable are Max-Miner (Bayardo, 1998), MAFIA (Burdick et al., 2001), CLOSET (Pei et al., 2000), CHARM (Zaki, & Hsiao, 2002), and CLOSET+ (Wang et al., 2003). Max-Minor extracts *maximal frequent item sets* (i.e., item sets for which no frequent superset is present) using a combined top-down bottom-up search. It exploits the property that all subsets of a frequent item set are also frequent to rapidly prune the search space. While it has proved to be efficient, the set of maximal frequent item sets do not contain sufficient information to compute confidence. Like Max-Miner, MAFIA is also a maximal frequent item set algorithm. It employs an alternate data structure based on a *vertical* data representation (i.e., list of object ids per item) and offers a compressed bitmap vector format to address memory concerns. 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 items sets. It uses intersection and pruning to incrementally add data to the IT tree. Intersection is noted as an expensive operation that

impedes the performance of the CHARM algorithm (Wang et al., 2003). Like MAFIA, CHARM involves a vertical data representation. CHARM addresses memory concerns using a difference based representation to enumerate the sets of object ids below the first level of its tree. 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. Such context corresponds to a branch in the FP tree. 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 to identify the set of closed frequent items. Experiments using CLOSET proved it 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 several enhancements to CLOSET. A top-down pseudo projection algorithm was added to address sparse data sets, item skipping was introduced to further prune the search space, and strategies from other algorithms, such as CHARM, were incorporated. CLOSET+ is considered to be a winning algorithm for mining closed frequent item sets (Wang et al., 2003).

All three algorithms, CHARM, CLOSET, and CLOSET+ begin by performing an initial scan over the data set to obtain frequency counts for each item. This initial processing is used to discard any items that are not frequent. The counts are also used to define a sort order for the context used to construct the respective tree structures,

although a different order is used by each algorithm. Sorting the items for the context is a heuristic to minimize the branches generated during execution of the algorithm and thereby improve efficiency (e.g., CHARM sorts by support, see end of Section 2.3 for rationale).

Max-Miner, CHARM, CLOSET, and CLOSET+ algorithms have been extensively validated using experiments against real and synthetic data sets. The data sets include a wide variety of characteristics including:

i) number of tuples ranging from a few thousand to near a million,

ii) number of items ranging from low hundreds to tens of thousands,

iii) contain maximal patterns on the order of ten to a hundred and fifty, and

iv) represent both sparse and dense data.

Both CHARM and CLOSET+ demonstrate orders of magnitude performance gains over Apriori.

The Max-Miner, MAFIA, CHARM, and CLOSET+ algorithms focus on just identifying the frequent item sets through various search and pruning strategies with minimal theory drawn directly from FCA. TITANIC (Stumme et al., 2002), on the other hand, is an algorithm for identifying the intents of concepts of an iceberg concept lattice using propositions derived directly from FCA. It is still a level-wise bottom-up search and prune algorithm similar to Apriori. However, its propositions enable:

i) calculation of counts for some portion of the candidates patterns to be derived from the counts obtained at a previous level,

ii) computation of the closure sets for each candidate patterns at the previous level, and

iii) early pruning of candidate patterns that cannot be key patterns (i.e., a minimal set of items used to generate a closure).

By leveraging these propositions, the TITANIC algorithm aggressively reduces the search space and reduces the number of database scans. On completion, the algorithm reports the set of closure patterns together with their key item sets. The set of closure patterns represent the set of concept intents and therefore the set of closed item sets. TITANIC has two variants; one to compute all closure patterns and a second to compute the only frequent closure patterns (i.e., intents of concepts from an iceberg concept lattice). The TITANIC algorithm does not construct the lattice. Relationships between the concepts are not identified and therefore not retained.

Using two public data sets, the performance of the TITANIC algorithm is experimentally evaluated by the authors against the Next-Closure algorithm (Ganter, 1984), an early algorithm for computing closed sets. Limited results are reported. No comparisons against the leading algorithms for computing closed frequent items sets are offered.

A recent survey provides an analysis of algorithms for closed frequent item sets from both a theoretical and analytical viewpoint (Yahia et al., 2006). Algorithms evaluated include CLOSE (Pasquier et al., 1999a), A-CLOSE (Pasquier et al., 1999b), TITANIC, CLOSET, CLOSET+, CHARM, Linear time Closed item set Miner (LCM) (Uno et al., 2004), and DCI-Closed (Lucchese et al., 2006). The CLOSE and A-CLOSE algorithms are predecessors to TITANIC. DCI-Closed and LCM are enhancements to CHARM to avoid subsumption checking; a step involving a potential exponential asymptotic complexity. The survey first formulates the problem of association rule mining from a frequent closed item (FCI) set perspective as two steps, namely:

i) discover the FCI sets together with their keys and upper covers, then

ii) from the FCIs, keys, and upper covers derive a basis of association rules.

The algorithms evaluated are classified into one of four categories:

i) "test-and-generate" those using an iterative bottom-up test and generation of candidate sets,

ii) "divide-and-conquer" those that gathers information in a compact representation and then recursively analyze sub-contexts to search for FCIs,

iii) "hybrid" those that use a combination of the previous two, and

iv) "hybrid-without-duplication" those that extend the hybrid algorithms with techniques to avoid subsumption checks[6].

CLOSE , CLOSE-A, and TITANIC are classified as test-and-generate, CLOSET and CLOSET+ are examples of divide-and-conquer, CHARM is a hybrid, and DCI-Closed and LCM are hybrid-without-duplication. For each category, various characteristics including potential for parallelism, storage format (e.g., vertical or horizontal), and generated output (e.g., FCIs, keys) are evaluated.

In addition to the theoretical analysis, the survey presents results from empirical evaluation of the algorithms executed against real and synthetic data sets including three dense data sets, three sparse data sets, and a manufactured "worst case" data set. Experiments are performed over a full spectrum of minimum support thresholds. Results indicate that the divide-and-conquer, hybrid, and hybrid-without duplication exhibit the similar runtime performance and memory profiles that are generally within an order of magnitude differential. Divergence in excess of an order of magnitude appeared at lower minimum supports. The test-and-set algorithms, on the other hand, were overall several orders of magnitude slower.

The Yahia et al. (2006) survey draws several conclusions. There has been "frenzied activity" in developing algorithms that efficiently identify the FCI sets. These

[6] A subsumption check ensures that the closure of a candidate item set does not equal a previously identified closed item set.

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 FCI set 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 (Zaki, 2000, Valtchev, Missaoui, & Godin, 2004, Zaki, & Hsiao, 2005, and Lakha & Stumme, 2005).

2.3 CHARM Algorithm – An Example of Frequent Item Set Mining

CHARM (Zaki, & Hsiao, 2002) is an efficient algorithm for mining frequent item sets within a data set. It accomplishes this task by identifying only the set of closed frequent item sets. CHARM dynamically constructs and searches an itemset-oidset[7] tree shown in Figure 2.1. By exploiting closed set theory, CHARM prunes branches and skips levels within the tree forming a hybrid search that searches both the item set and object set search space.

The itemset-oidset search tree is comprised of nodes similar to concepts in that each node contains a set of items and a set of object identifiers (ids). Initially, the sets in the tree may not be closed. For example, the set AD in node {AD45} in Figure 2.1 is not closed since there exists nodes {ADC45}, {ADW45}, and {ADCW45} with the same set of object ids. Only node {ADCW45} represents a closed set. The CHARM algorithm thus prunes the itemset-oidset search tree such that only nodes containing closed item sets

[7] The CHARM authors use the terminology itemset-tidset (IT) tree. Itemset-oidset is being used in this discussion to maintain consistency with other presentations.

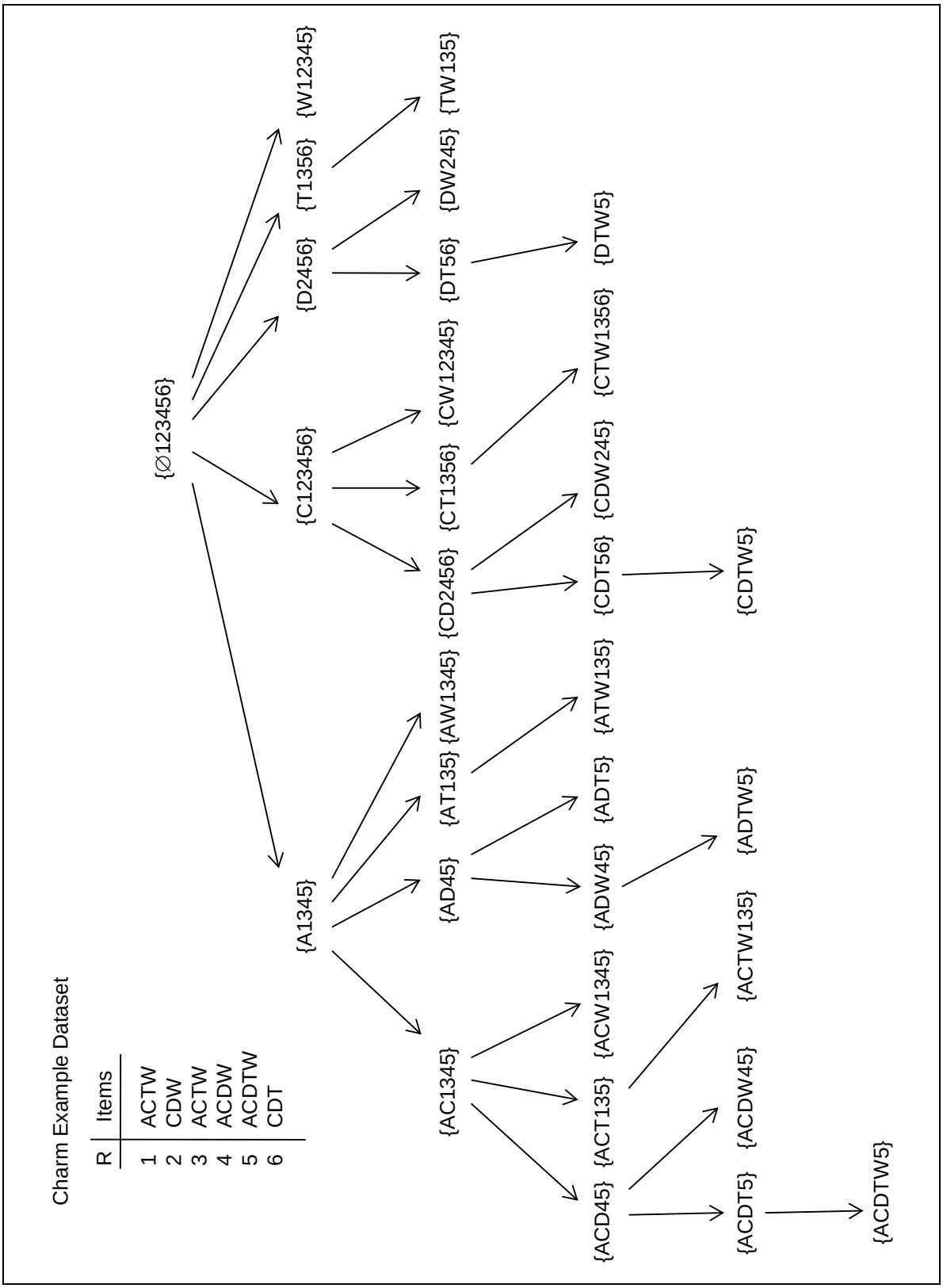


Figure 2.1: Itemset-oidset tree used by the CHARM algorithm. Example taken from (Zaki, & Hsiao, 2002).

remain. Pruning is performed dynamically during the construction process. Each remaining node identifies a potential closed set.

The itemset-oidset tree is constructed in a recursive process. The first level below the root node is constructed using a pass of the data set. The result is a set of nodes, each containing a single item and its set of object ids. Thereafter, each node is then expanded into a set of child nodes by performing a union of item sets together with an intersection on object sets for each sibling to the right of the node. For example, in Figure 2.1 the children of node {D2456} are constructed by performing a union of item sets together with an intersection on object sets against {T1356} and {W12345}. This results in children {DT56} and {DW245}. Note that the children will share a common item set prefix. Each child effectively appends another item to the prefix. Thus, the children are said to be in an equivalence class defined by the parent. If a generated node has an object set whose size is less than a specified minimum support threshold, then that node is discarded. Further pruning is achieved by exploiting the following theorem:

> **Theorem 2.3.1.** Let $I_i \times o(I_i)$ and $I_j \times o(I_j)$ be any two members of an equivalence class [P], then the following properties hold true:
>
> **Property 2.3.1**: $o(I_i) = o(I_j) \rightarrow I_i'' = I_j'' = (I_i \cup I_j)''$
>
> **Property 2.3.2**: $o(I_i) \subset o(I_j) \rightarrow I_i'' \neq I_j''$, but $I_i'' = (I_i \cup I_j)''$
>
> **Property 2.3.3**: $o(I_i) \supset o(I_j) \rightarrow I_i'' \neq I_j''$, but $I_j'' = (I_i \cup I_j)''$
>
> **Property 2.3.4**: $o(I_i) \neq o(I_j) \rightarrow I_i'' \neq I_j'' \neq (I_i \cup I_j)''$

Property 2.3.1 implies that if two children have the same set of objects then every occurrence of the first item set can be replaced with the union of the two item sets. Furthermore, the second child can be pruned since it derives the same closure. Property 2.3.2 implies that if the first child has an object set that is a subset of the other then every

occurrence of the first child's item set can be replaced with the union of the two item sets. The second child, however, cannot be pruned since it generates a different closure. Property 2.3.3 is similar in implication to property 2.3.2, however, the item set union is expressed by adding a child to the first child. That grandchild will generate a different closure. Since the grandchild expresses the union of item sets, the second child can be pruned. Property 2.3.4 implies that if two children have object sets that are neither equal nor a subset of the other, then each child leads to different closures. Furthermore, a child representing the union of the item sets is added to the first child since it leads to its own closure. Further explanation of these properties together with examples are given in (Zaki, & Hsiao, 2002).

The complete CHARM algorithm is given in Algorithm 2.1. Lines 1 through 3 initialize the top level equivalence class consisting of the top node and immediate children. A child is created for each item whose set of object ids meets the minimal threshold. After initializing the set of closed item sets to null (line 4), the function CHARM-EXTEND is called to extract the closed item sets (line 5).

CHARM-EXTEND is a recursive function that dynamically builds and prunes the itemset-oidset tree. It is passed a node representing the top of an equivalence class and the set of found closed item sets. CHARM-EXTEND is composed of two nested loops to compare and process each pair of child nodes (lines 7 and 8). Each pair whose intersection does not meet the minimum support threshold is ignored (lines 10 and 11). Each remaining pair is compared and processed according the properties 2.3.1 through 2.3.4 (lines 12 through 21). For properties 2.3.3 and 2.3.4, a grandchild node is created and added as a child of the first in the pair (lines 19 and 21 respectively). If after

Let Node be a tuple {I, O, Children} where I is a set of Items, O is a set of objects, and Children is a list of other nodes. A Node forms an equivalence class.

CHARM(K{$\mathcal{I}$, $\mathcal{O}$, $\mathcal{R}$}, MinSupp)

1. $N_{Top} \leftarrow$ new Node($\varnothing$, $\varnothing$) // top level equivalence class
2. for each $I_i \in \mathcal{I} \wedge |o(I_i)| \geq$ MinSupp: // initialize its children
3. Add new Node ($\{I_i\}$, $o(I_i)$) to N_{Top}.Children
4. $C \leftarrow \varnothing$ // set of closed item sets
5. CHARM-EXTEND(N_{Top}, C)
6. return C // all closed sets

CHARM-EXTEND(N_{Parent}, C)

7. for each $N_{Child} \in N_{Parent}$.Children:
8. for each $N_{Sibiling} \in N_{Parent}$.Children $\wedge$ $N_{Sibling}$ is a later sibling of N_{Child}
9. $I \leftarrow N_{Child}.I \cup N_{Sibling}.I$
10. $O \leftarrow N_{Child}.O \cap N_{Sibling}.O$
11. if $|O| \geq$ MinSupp: // threshold check
12. if $N_{Child}.O = N_{Sibling}.O$: // property 2.3.1
13. Remove $N_{Sibling}$ from N_{Parent}.Children
14. Replace $N_{Child}.I$ with I in all Nodes E where $N_{Child}.I \subset E.I$
15. else if $N_{Child}.O \subset N_{Sibling}.O$: // property 2.3.2
16. Replace $N_{Child}.I$ with I in all Nodes E where $N_{Child}.I \subset E.I$
17. else if $N_{Child}.O \supset N_{Sibling}.O$: // property 2.3.3
18. Remove $N_{Sibling}$ from N_{Parent}.Children
19. Add new Node (I, O) to N_{Child}.Children in order *f*
20. else if $N_{Child}.O \neq N_{Sibling}.O$: // property 2.3.4
21. Add new Node (I, O) to N_{Child}.Children in order *f*
22. if N_{Child}.Children $\neq \varnothing$:
23. CHARM-EXTEND ($N_{Child,}$ C)
24. if N_{Child} not subsumed in C:
25. Add N_{Child} to C

Algorithm 2.1: The CHARM algorithm[8]. (Zaki, & Hsiao, 2002)

[8] The authors use a different notation. Notation has been modified to be consistent with notation of this report. Furthermore, authors express lines 12-21 in a separate function CHARM-PROPERTY. This function has been placed in-line for brevity.

comparing a node with each of its later siblings the node has children, then CHARM_EXTEND is called recursively to process the node (lines 22 and 23). Furthermore, the node may represent a closed item set, provided its closure is not an item set that has been identified in the traversal of a previous branch. If the item set's closure is equal to an identified closed item set, the then item set is said to be *subsumed*. If not subsumed, the item set is added to the list of closed item sets (lines 24 and 25).

When adding children to a node, the CHARM algorithm will maintain the children in order of their supports (order *f* in lines 19 and 21). The rationale is to encounter properties 2.3.1 and 2.3.2 sooner than later. For both of these properties, a child is not generated for the first node. As a result, fewer levels are processed thereby improving performance.

As a further enhancement, CHARM uses a difference based representation to enumerate the sets of object ids below the first level of the itemset-oidset tree. These sets are termed diff sets. For example, in Figure 2.1 the object id set of {CD2456} which is a child of {C123456} will be represented as diff set {1, 3}. This results in a compact representation and improves performance. Fast determination of superset, subset, equality, and inequality is performed using differences operations on diff sets in place of intersections on object id sets.

As the CHARM algorithm proceeds, a closed item set identified in one branch of the itemset-oidset tree may be subsumed by a closed item set identified in a previous branch. Therefore, before adding an item set to the set of closed item sets a subsumption check against all found closed items sets must be performed. To avoid comparing each item set with each found item set, introducing an $O(\boldsymbol{n}^2)$ overall complexity, the found

item sets are placed into a hash table using the summation of the oids as the hash function. This hash function will leverage the fact that an item set will subsume item sets that has the same set of oids. Thus, the subsumption check can be performed in O(**1**), resulting in an O(***n***) overall complexity.

2.4 Post Mining Lattice Construction – Valtchev, Missaoui, and Lebrun Algorithm

An approach to completing the association rule mining problem is to use a closed frequent item set mining algorithm, such as CHARM, as the first part of a two step process. Then, subsequently use a lattice construction algorithm to generate the upper covers of each closed item set to form the lattice. Valtchev, Missaoui, and Lebrun (2000) (VML) offer an efficient algorithm for such purpose.

Concepts in the VML algorithm are tuples consisting of intent, parent list, and child list. The parent and child lists identify the upper and lower covers respectively. Initially, the parent and child lists of all concepts are empty. The objective of the VML algorithm is to organize a set of concepts into a lattice by populating the parent and child lists of each concept. VML algorithm asserts that the list of concepts can be first sorted into $\{C_1, C_2, \ldots, C_n\}$ such that the incremental insertion of concept C_i will only update the parent-child links between C_{prior} and C_i where prior < i (i.e., C_{prior} has already been linked into the lattice). Of the concepts linked into the lattice, there exists a subset that does not have any children. This subset forms an *anti-chain* (i.e., all concepts are mutually incomparable) that is the lower border of the lattice. This set, denoted as Border, is used to identify the parents for the next concept C_i. If for a concept $C_{Border} \in$ Border, C_{Border}.Intent $\cap$ C_i.Intent = C_{Border}.Intent then C_{Border} is a parent of C_i. However, this may not identify all of the parent concepts. The missing parent concepts may be

"shadowed" by concepts in the Border. If a concept $C_{Border} \in$ Border shadows a parent C_{parent} then $C_{parent}.Intent \subset C_{Border}.Intent$. Thus, $C_{parent}.Intent \cap C_{Border}.Intent \cap C_i.Intent \neq \varnothing$. In such case, C_{parent} is the ancestor of C_{Border} where $C_i.Intent \cap C_{Border}.Intent = C_{parent}.Intent$. Thus, a set Candidates = $\{C_{Border} \mid C_{Border} \in \text{Border} \wedge C_{Border}.Intent \cap C_i.Intent\}$ can be used to identify the parents of C_i. However, the candidate set could contain concepts that identify the same parent concept, or identify parent concepts that are actually ancestors of the true parent concepts. These cases are the result of join concepts existing between the candidates. The processing of each candidate individually can lead to violating the lattice connection property (i.e., a connection is made between any two concepts C_1 and C_2 for which order < exists and there is no concept C_3 for which $C_1 < C_3 < C_2$). This problem can be remedied by removing from the candidates those concepts that do not have a maximal intent with respect to the other candidates. After purging the non-maximal concepts, the candidate set will itself form an anti-chain.

Algorithm 2.2 provides the complete VML algorithm. Following the sort of concepts (line 1), both the lattice and Border set are initialized with the first concept (lines 3 and 4). The remaining concepts are then incrementally inserted (lines 5 through 13). Lines 6 through 8 identify the concepts in the Border that have an intersection with then next concept's intent. Such concepts together with their intersection sets are added as tuples to the Candidates set. The set of Candidate tuples are then purged of any non maximal intersection sets (line 9). Each remaining tuple in Candidates is used to identify a parent to be linked to the new concept (lines 10 through 13). After adding a link between each identified parent and the new concept, the Border set is updated by

Let Concept be a tuple {Intent, Parents, Children} where Intent is the intent of a concept, Parents is a list of parent concepts, and Children is a list of child concepts.

```
HASSE(C)                          // C is a set concepts
 1.  Sort(C)                                          // sort by |Intent| is sufficient
 2.  C_First ← First C ∈ C                            // first concept in C
 3.  L ←C_First                                       // the  lattice
 4.  Border ← {C_First}                               // anti-chain of minimal concepts
 5.  for each C_i ∈ C past C_First:
 6.      Candidates ← ∅                               // set of tuples {C_i, Intersect}
 7.      for each C_Border ∈ Border ∧ C_Border.Intent ∩ C_i.Intent ≠ ∅:
 8.          Add {C_Border, C_Border.Intent ∩ C_i. Intent} to Candidates
 9.      Candidates ← MAXIMA(Candidates)              // purges non maximal candidates
10.      for each Y_i ∈ Candidates:                   // find and link parents
11.          C_Parent ← FIND-CONCEPT(Y_i.C_i, Y_i.Intent)
12.          Add parent-child link between C_Parent and C_i
13.      Border ← (Border – C_i.Parents) ∪ {C_i}
14.  return L

MAXIMA(Candidates)        // Candidates is a set of tuples {C_i, Intersect}
15.  Sort(Candidates)                                 // sort by |Intersect| is sufficient
16.  MaxIntersects ← ∅
17.  for each Y_i ∈ Candidates:
18.      if ¬∃ Y_Max ∈ MaxIntersects | Y_i.Intersect ⊂ Y_Max.Intersect:
19.          Add Y_i to MaxIntersects
20.  return MaxIntersects

FIND-CONCEPT(C_i, Intent)
21.  while C_i.Intent ≠ Intent:
22.      for each C_Parent ∈ C_i.Parents:
23.          if Intent ⊆ C_Parent.Intent:
24.              C_i ← C_Parent
25.              break out of for each
26.  return C_i
```

Algorithm 2.2: The Valtchev, Missaoui, and Lebrun lattice construction algorithm. (Valtchev et al., 2000)

replacing all the parents of the new concept with the new concept. After all concepts have been inserted, the lattice is returned (line 14).

It is sufficient to sort the concepts (line 1) in the order of increasing size of intent, since a child concept must have a larger intent. Concepts with the same size intent are mutually incomparable. Given this, the sort can be accomplished in linear time. The sort simply arranges the concepts into an array of bucket lists using the intent size as an index, and then concatenates the buckets. The overall asymptotic complexity of the VML algorithm is $O(\ell\ w(\mathcal{L})^2\ m)$, where $\ell = |\mathcal{L}|$, $w(\mathcal{L})$ = width of lattice $\mathcal{L}$, and $m = |\mathcal{I}|$. An improved version achieves $O(\ell\ w(\mathcal{L})\ deg(\mathcal{L})\ m)$, where $deg(\mathcal{L})$ = degree of lattice $\mathcal{L}$.

2.5 Incremental Lattice Construction – Missaoui, Godin, and Alaoui Algorithm

Missaoui, Godin, and Alaoui (1995) algorithm (GMA) is a concept lattice construction algorithm that is often cited in literature. 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, the GMA algorithm will insert the new object into the lattice and produce a new concept lattice $\mathcal{L}+$. Figure 2.2 depicts the incremental insertion of the first six objects relation R of Figure 1.1. The bold text and lines identify the changes to the lattice as a result of inserting the next object. The dashed line indicates a link that is removed. As can be seen in Figure 2.2, 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 the addition of many new concepts.

The general strategy for the GMA algorithm is to partition the current concepts in the lattice into three groups: modified, generator, and old. Modified are concepts into

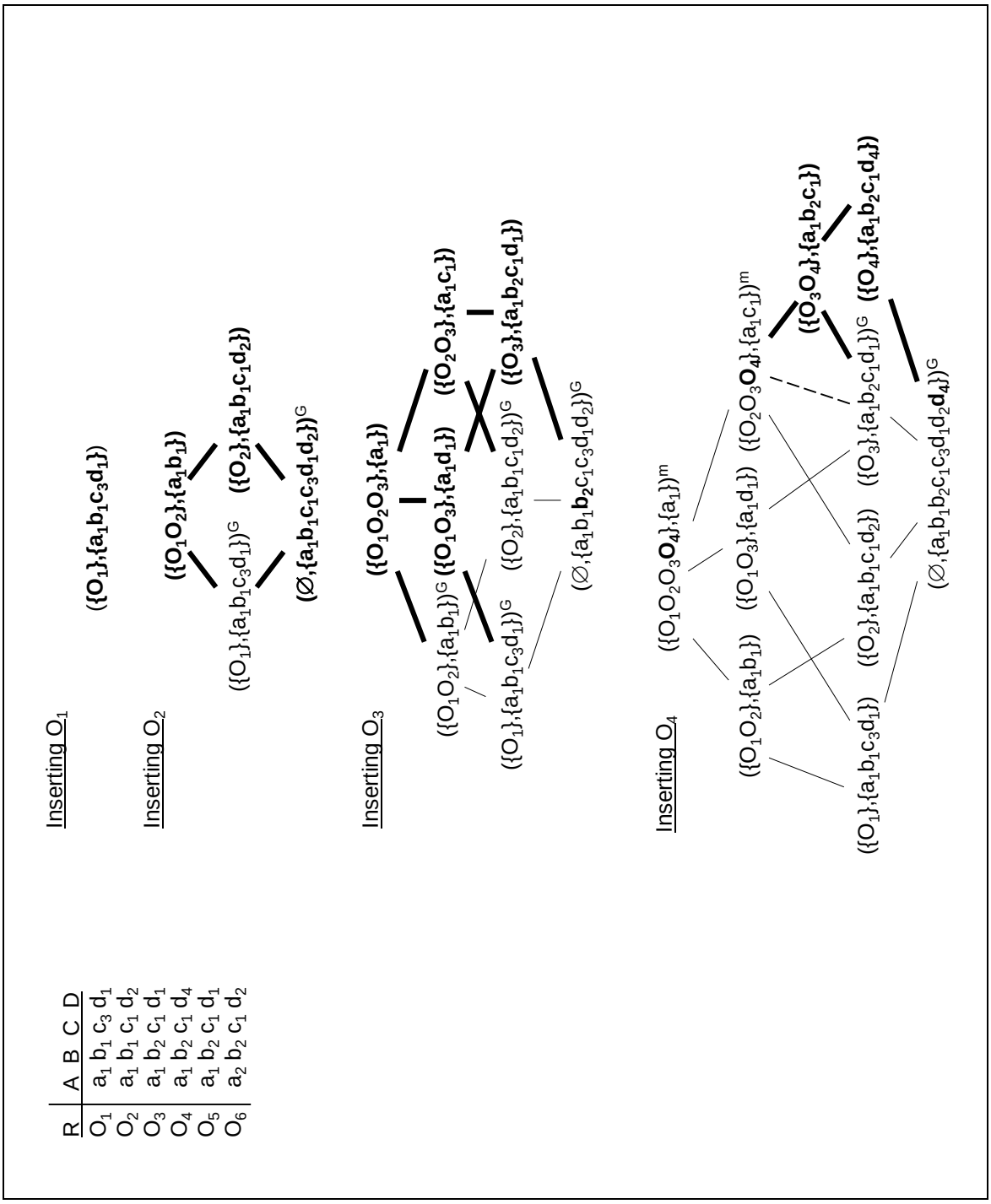


Figure 2.2: Progression of incremental object insertion into a concept lattice. Bold text indicates new concepts, inserted items or inserted objects, ^{G} a generator concept, ^{m} a modified concepts, and dashed lines are removed links.

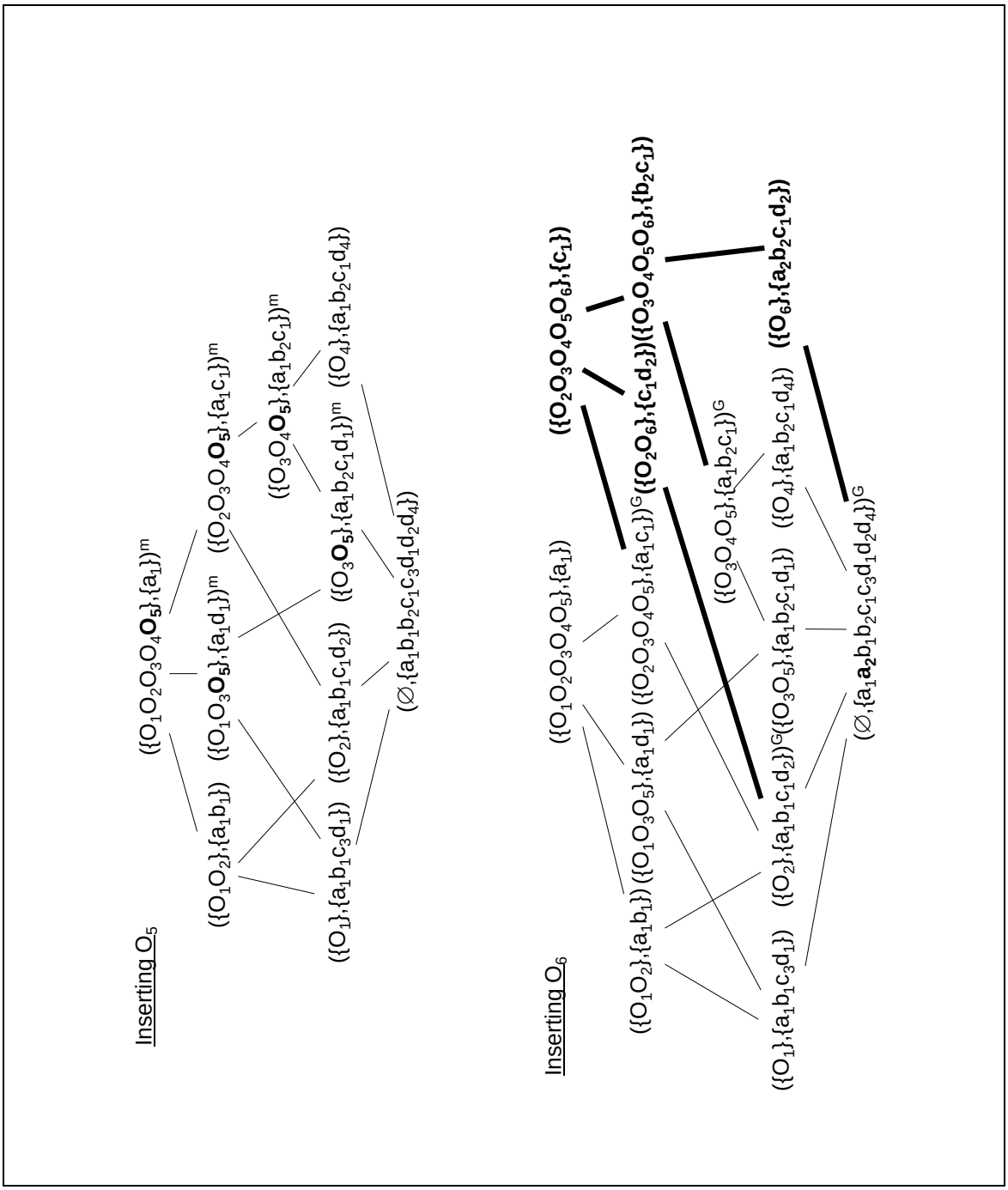


Figure 2.2 continued: Progression of incremental object insertion into a concept lattice. Bold text indicates new concepts, inserted items or inserted objects, ^{G} a generator concept, and ^{m} a modified concepts.

which the object id of the next object is added. In Figure 2.2, these are denoted with an m superscript. Generators are concepts are used to generate new concepts. These are denoted with a G superscript. All other concepts are considered old. Old concepts play no role in the insertion process. They are not changed, are not used to generate concepts, and cannot become a parent or a child of a new concept. Modified concepts are readily identified. The concepts with an intent that is a subset of the next object's items will become modified. 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 not a generator if there exists an ancestor concept whose intent when intersected with the next object's items produces the same intersection set. For example, when inserting O_6 in Figure 2.2 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 $\{a_1c_1\}$, but only ($\{O_3O_4O_5\}$, $\{a_1b_2c_1\}$) is a generator. It is an ancestor of the other two. This identification of a generator concept is expressed in Proposition 2.5.1. Each generator concept thus creates a new concept having the extent of the generator union the object id as its extent and the intersection set as its intent.

> **Proposition 2.5.1**: if $(O_x, I_x) = \inf \{(O_y, I_y) \in G \mid I_x = I_y \cap I)\}$ for some set I and there does not exist a concept $(O_z, I_z) \mid I_z = I$, then (O_y, I_y) is a generator for a concept $(O_x, I_x) \mid O_x = O_y \cup \{O\}$ and $O_y = I_y \cap I$.

In addition to creating a concept, the new concept must be linked into the lattice. Each generator concept will be a child of the concept it generated. The new concept must be further linked into the lattice by searching for its 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 that a connection exists between two concepts C_1 and C_2 provided $C_1 < C_2$ and 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. 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 2.2. In these cases the new object 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 2.3. Lines 1 and 2 bootstrap the concept lattice and are only executed upon the insertion of the first object. For the first object, the lattice is initialized with a concept whose extent is the object id and intent is the object's items. Lines 4 through 10 are pre-steps to the insertion process to ensure that the bottom concept in the lattice accounts for all the items of the object. If the next object introduces items that are not presently in the lattice, those items must first be inserted to the intent of the bottom concept. A special case exists when the bottom concept has a non-empty extent. The new items cannot be inserted into that concept. To do so would change the set of items for each object recorded in the extent of the bottom concept. To handle this case, a new bottom concept is generated.

Line 12 defines a Processed list. This list will contain references to the modified and generated concepts of the current incremental insertion. The Processed list provides a means to validate that a concept is indeed a generator and to limit the search for parents.

Let Concept be a tuple {O, I, Children} where O is a set of object ids, I is a set of items, and Children is a list of child concepts.

Let C_{Bottom} be the supremum of a concept lattice G, G contains a reference to all concepts

```
ADD(Oi, I)              // Add object Oi together with its set of items I
 1.   if CBottom = Ø:                              // special case of empty lattice
 2.       CBottom ← new Concept ({Oi}, I)
 3.   else:
 4.       if I ⊄ CBottom.I:                        // check for items not in the lattice
 5.           if CBottom.O = Ø:                    // a generated bottom
 6.               CBottom.I ← CBottom.I ∪ I
 7.           else:                                // generate the new bottom
 8.               CNew ← new Concept (Ø, I)
 9.               Add CNew to CBottom.Children
10.               CBottom ← CNew
11.
12.       Processed ← Ø    // a vector of sets of concepts indexed by the cardinality
13.                        // of each intersection set
14.       for each Ci ∈ G in ascending |I| order:
15.           if Ci.I ⊆ I:                         // modified concept
16.               Add Oi to Ci.O
17.               Add Ci to Processed[|Ci.I|]      // possible parent
18.               if Ci.I = I:                     // no more processing needed?
19.                   return
20.           else:                                // existing concept
21.               Intersect ← Ci.I ∩ I
22.                                                // intentionally left blank
23.               if ¬∃ Cj ∈ Processed[|Intersect|] | Cj.I = Intersect:  // Cj a generator?
24.                   CNew ← new Concept (Ci.O ∪ {Oi}, Intersect)
25.                   Add CNew to Processed[|Intersect|]
26.                   Add Ci to CNew.Children      // modify edges
27.                   for each Ck ∈ Processed ∧ |Ck.I| < |Intersect|:
28.                       if Ck.I ⊂ Intersect:     // is Ck a potential parent of CNew?
29.                           Parent ← TRUE
30.                           for each CChild ∈ Ck.Children ∧ Parent = TRUE:
31.                               if CChild.I ⊂ Intersect:
32.                                   Parent ←FALSE
33.                           if Parent = TRUE:
34.                               if Ci ∈ Ck.Children:
35.                                   Remove Ci from Ck.Children
36.                               Add CNew to Ck.Children
37.                   if |Intersect| = |I|:        // last valid generator ?
38.                       return                   // no more work to do
```

Algorithm 2.3: Godin, Missaoui, and Alaoui lattice construction algorithm. (Godin et al., 1995)

The Processed list is organized as a vector of concepts sets. The size of intent is used to index a given set. Thus, each set in the Processed list contains concepts whose cardinality of intent are the same. To validate a potential generator, only the set whose size of intent is the same as the size of an intersection set will be searched to ensure the real generator has not already been found and processed.

Lines 14 through 38 provide the main loop to insert the next object. All concepts currently in the lattice are processed in the order of the size of their intents. This order will ensure that valid generator concepts are processed first and that all potential parents for a new concept will exist in the Processed list at the time of inquiry. Line 15 tests if a concept of the lattice is a modified concept. If so, the next object's id is added to the modified concept's extent and the modified concept is added to the list of Processed concepts (lines 16 and 17). If a concept's intent is equal to the next object's items, then there are no further generator concepts or modified concepts to be found. Thus the algorithm exits (lines 18 and 19). If a concept in the lattice is not a modified concept, then it is an existing concept that is either a generator or old. Lines 21 and 23 provide the predicate to identify generator concepts. Old concepts are ignored. For each generator concept, a new concept is generated (line 24). The new concept is added to the Processed list and as a parent to the generator concept (lines 25 and 26). Lines 27 through 36 then search the Processed list for potential parents. A potential parent is a concept whose intent is a subset of the new concept's intent (line 28). A potential parent is a parent provided it does not have a child concept whose intent is a subset of the new concept (lines 29 through 32). If the potential parent is indeed a parent, then the parent is linked to the new concept (line 36). If the parent has the generator concept as a child then the

link between the parent and the generator concept is removed (lines 34 and 35). Lastly, lines 37 and 38 provide exit from the function on processing the generator that generates the concept with intent equal to the next object's items. There will be no valid generators beyond this concept. Therefore, there is no more work to do.

2.6 Applying FCA to Association Rule Mining – GALICIA-T Algorithm

GALICIA-T (Valtchev et al., 2002) is an enhanced GMA algorithm adapted specifically for generating frequent items sets. It uses a *trie* data structure (Knuth, 1998) to represent the set of concept intents. A trie is a tree based data structure that provides a compact representation by sharing common prefixes along branches and enables efficient search, insertion, and set operations. Each edge denotes the addition of an item in the item set. Figure 2.3 provides an example of the GALICIA-T trie data structure after inserting the first three objects in relation R (middle left), and after inserting the fourth object (middle right). The lattices depicted at the top and bottom of Figure 2.3 correspond to left and right tries respectively. The lattices, as shown, are provided for illustrative purposes and are not part of the GALICIA-T data structure. Each filled-in circle in the trie corresponds to a concept in the lattice. Theses nodes are augmented by a support count (shown at the northwest position on the circle), and list of lower covers (not shown, but can be determined from examination of the corresponding lattice). Object ids are not stored in the GALICIA-T data structure.

Given a concept lattice whose item sets, supports, and lower covers are expressed in a trie as described, the GALICIA-T algorithm inserts the next object into the lattice through a guided traversal that produces an independent trie data structure. The generated trie represents a set of new concepts. An example is depicted in the center of

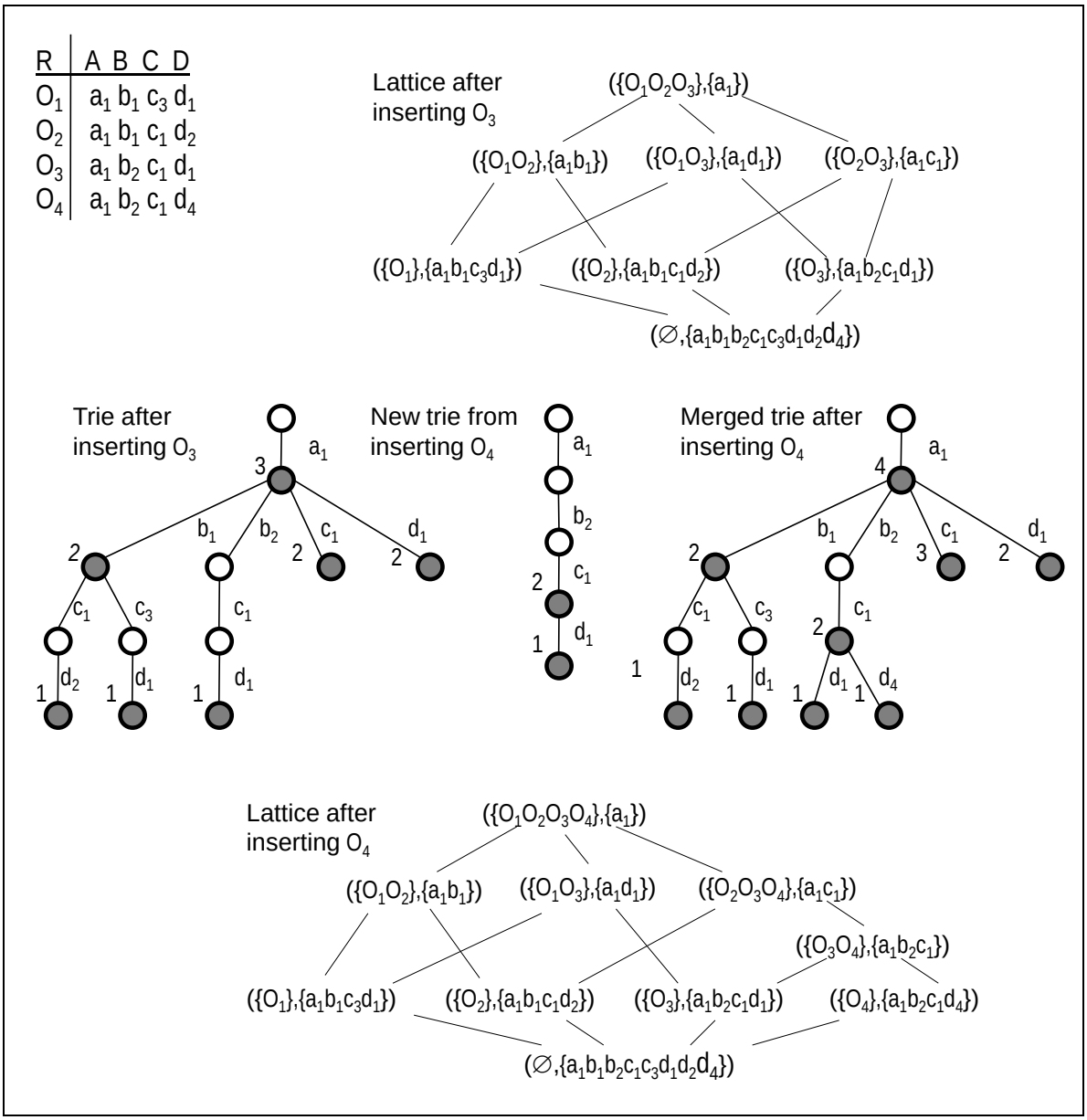


Figure 2.3: Trie data structure used by the GALICIA-T algorithm. The trie data structure before and after inserting object O_4 is depicted in the center. Edges identify an item. Filled circles correspond to concepts. Open circles are just nodes on the path identifying an item set. Each node corresponding to a concept is augmented by its support count (shown at the northwest position on the circle), and list of lower covers (not shown). Lattices at the top and bottom correspond to the trie structure and are provided of illustrative purposes only. Object ids are not stored in the trie structure.

Figure 2.3. The object's intent is used to guide the traversal and both the trie child links and lower cover links are utilized accordingly. If a terminal node is encountered during the traversal, its support is incremented. The resulting trie data structure is then merged into the source trie to produce a new trie representing the incremented lattice.

GALICIA-T was evaluated by the authors against the CLOSET algorithm. CLOSET outperforms GALICIA-T by a near two orders of magnitude. However, the authors argue that GALICIA-T enables incremental insertion and thus offers a performance gain. The cost of adding an incremental set of objects using GALICIA-T is lower than the cost of processing an entire data set by CLOSET. Valtchev et al. note a shortcoming with GALICIA-T: the approach requires generation the entire set of concepts including those infrequent with respect to cardinality of extent.

2.7 Frequent Item Set Mining with Lattice Construction – CHARM-L Algorithm

The CHARM algorithm presented earlier extracts the set of close frequent item sets from a data set without deriving the upper and lower covers. Zaki and Hsiao (2005) acknowledge that upper or lower covers are essential to efficient mining of association rules. Furthermore, they cite that a post-mining lattice construction algorithm can result in an $O(|\mathcal{L}|^2)$ asymptotic complexity, from (Nourine, & Raynaud, 1999). Zaki and Hsiao thus offer an enhanced algorithm, CHARM-L, to dynamically construct the lattice as the closed item sets are discovered.

The lattice of the CHARM-L algorithm is maintained as a separate data structure from the CHARM itemset-oidset tree. When the core CHARM processing identifies a new frequent closed item set, CHARM-L will insert the item set into the lattice as a child of the concept corresponding to the parent node in the itemset-oidset tree. What remains

is to identify concepts already in the lattice that are to become children of the new concept. These children were added to the lattice as a result of processing previous branches in the tree. To quickly identify these children, the nodes in the tree are augmented with a list of concept ids representing the concepts whose intersection of item sets equals the node's item set. A concept id is uniquely assigned to each concept as it is created. Thus, when a grandchild node is created from a child and sibling nodes, the set of concept ids for the grandchild is created by intersecting the concept ids of the child and sibling nodes.

Algorithm 2.4 provides the CHARM-L algorithm. It is essentially the same as Algorithm 2.1 except that a reference to a concept in the lattice is used in place of a set of closed item sets (lines 4, 5, 6 and 25). The nodes in the itemset-oidset tree also maintain a list of concept ids (lines 3, 11, 20, 22, and 27). Lastly, a call to function SUBSUMPTION-CHECK-LATTIC-GEN is made in place of the previous hash based subsumption check (line 23). This function will perform the subsumption check and if not subsumed, insert a new concept into the lattice. The call is moved ahead of the recursive call to CHARM-EXTEND, since the concept returned from SUBSUMPTION-CHECK-LATTIC-GEN is passed to CHARM-EXTEND.

The algorithm for SUBSUMPTION-CHECK-LATTIC-GEN is given in Algorithm 2.5. SUBSUMPTION-CHECK-LATTIC-GEN is passed a concept in the lattice representing a parent and an itemset-oidset node representing a candidate closed item set. The node's list of concept ids is used to lookup the corresponding set of potential child concepts (line 1). If the support of any potential child concepts = |O| of the node, then the node is subsumed (line 2). In such case, the parent concept is returned.

Let Node be a tuple {I, O, CIDS, Children} where I is a set of Items, O is a set of objects, CIDS is a set of concept ids, and Children is a list of child nodes. A Node forms an equivalence class.

Let Concept be a tuple {I, Supp, Parents, Children, CID} where I is a set of Items, Supp is the support of the concept, Parents is a list of parent concepts, Children is a list of child concepts, and CID a concept id (uniquely assigned on creation).

CHARM(K{$\mathcal{I}$, $\mathcal{O}$, $\mathcal{R}$}, MinSupp)

1. $N_{Top} \leftarrow$ new Node($\varnothing$, $\varnothing$, $\varnothing$) // top level equivalence class
2. for each $I_i \in \mathcal{I} \wedge |o(I_i)| \geq$ MinSupp: // initialize its children
3. Add new Node ($\{I_i\}$, $o(I_i)$, $\varnothing$) to N_{Top}.Children
4. $C_{Top} \leftarrow$ new Concept($\varnothing$,$\varnothing$) // the lattice
5. CHARM-EXTEND(N_{Top}, C_{Top})
6. Return C_{Top} // lattice w/all closed sets

CHARM-EXTEND(N_{Parent}, C_{Parent})

7. for each $N_{Child} \in N_{Parent}$.Children:
8. for each $N_{Sibiling} \in N_{Parent}$.Children $\wedge$ $N_{Sibling}$ is a later sibling of N_{Child}:
9. $I \leftarrow N_{Child}.I \cup N_{Sibling}.I$
10. $O \leftarrow N_{Child}.O \cap N_{Sibling}.O$
11. $CIDS \leftarrow N_{Child}.CIDS \cap N_{Sibling}.CIDS$
12. if $|O| \geq$ MinSupp: // threshold check
13. if $N_{Child}.O = N_{Sibling}.O$: // property 2.1
14. Remove $N_{Sibling}$ from N_{Parent}.Children
15. Replace $N_{Child}.I$ with I in all Nodes E where $N_{Child}.I \subset E.I$
16. else if $N_{Child}.O \subset N_{Sibling}.O$: // property 2.2
17. Replace $N_{Child}.I$ with I in all Nodes E where $N_{Child}.I \subset E.I$
18. else if $N_{Child}.O \supset N_{Sibling}.O$: // property 2.3
19. Remove $N_{Sibling}$ from N_{Parent}.Children
20. Add new Node (I, O, CIDS) to N_{Child}.Children in order *f*
21. else if $N_{Child}.O \neq N_{Sibling}.O$: // property 2.4
22. Add new Node (I, O, CIDS) to N_{Child}.Children in order *f*
23. $C_{New} \leftarrow$ SUBSUMPTION-CHECK-LATTIC-GEN(C_{Parent}, N_{Child})
24. if N_{Child}.Children $\neq \varnothing$:
25. CHARM-EXTEND (N_{Child}, C_{New})
26. if $C_{New} \neq C_{Parent}$:
27. Add C_{New}.CID to CIDS of appropriate nodes

Algorithm 2.4: The CHARM-L algorithm[9]. (Zaki, & Hsiao, 2005)

[9] The authors use a different notation. Notation has been modified to be consistent with notation of this report. Furthermore, authors express lines 12-22 in a separate function CHARM-PROPERTY. This function has been placed in-line for brevity. Lines 26 and 27 are expressed in the original algorithm at the top of the outer loop as a call to a function UPDATE-C. Algorithm for UPDATE-C is not provided by the authors and its details are unclear.

SUBSUMPTION-CHECK-LATTIC-GEN(C_{Parent}, N_{Child})

1. $C \leftarrow \{C_i \mid C_i \in AllConcepts \wedge C_i.CID \in N_{Child}.CIDS\}$ // lookup concepts by CID
2. if $\exists\ C_i \in C \mid C_i.Supp = |N_{Child}.O|$:
3. return C_{Parent} // N_{Child} is subsumed
4. $C_{New} \leftarrow$ new Concept ($N_{Child}.I$, $|N_{Child}.O|$) // not subsumed, create the concept
5. Add link between C_{Parent} and C_{New} // add new as a child
6. for each $C_{Child} \in C \wedge C_{Child}$ is Minimal: // adjust parent-child links
7. Add parent-child link between C_{New} and C_{Child}
8. for each $C_{ChildParent} \in C_{Child}.Parents \wedge C_{ChildParent}.I \subset C_{New}.I$:
9. Remove link between $C_{ChildParent}$ and C_{Child}
10. return C_{New}

Algorithm 2.5: The CHARM-L subsumption check algorithm[10].

If not subsumed, a new concept is created and added as a child of the parent concept (line 4 and 5). Of the potential child concepts, only the minimal concepts represent true children. These are added as children to the new concept (lines 6 and 7). This may result in existing edges now violating the lattice connection property. Thus, for each parent concept $C_{ChildParent}$ of a concept C_{Child} added as a child of the new concept, the parent-child link is removed when the intent of $C_{ChildParent} \subset$ the new concept's intent (lines 8 and 9). Finally, the new concept is returned. It will be used in the recursive call to CHARM-EXTEND.

The performance CHARM-L algorithm is evaluated using empirical tests against six of the commonly used benchmark data sets. CHARM-L is compared against CHARM and a post-FCI mining lattice construction algorithm. CHARM-L exhibits a small degradation in performance when compared to CHARM. The amount of degradation increases as the support is lowered. When compared to post-lattice construction, CHARM-L demonstrates significant gains in excess of two orders of

[10] The authors use a different notation. Notation has been modified to be consistent with notation of this report.

magnitude. The separation in performance increases dramatically as the support is lowered.

2.8 Adding Iceberg Processing to Lattice Construction – MAGALICE Algorithm

MAGALICE (Rouane et al., 2004) is an extension of an GMA based algorithm to enable incremental construction of an iceberg concept lattice. That is, given an iceberg lattice $\underline{\mathcal{L}}^{\alpha}$ [11] and a new object O, MAGALICE will insert the new object into the lattice producing $\underline{\mathcal{L}}^{\alpha+}$. After inserting an object using a GMA based algorithm, concepts that do not meet the minimum support threshold are discarded. Thus, the challenge is to regenerate concepts for extent intent pairs that were previously discarded that now meet the minimum support threshold as a result of adding object O_i. Such concepts are called *jumpers*.

The MAGALICE algorithm relies on a strong cardinality property that the extent of a jumper meets the minimum support threshold, yet its extent less object O_i does not. Thus, the cardinality of extent of all jumpers will equal the minimum support threshold. Furthermore, the parents of a jumper will already exist in the lattice prior to insertion and upon insertion of O_i will be modified to include O_i. Thus, the MAGALICE algorithm will limit its search for concepts that generate jumpers to those where O_i has been added. These concepts are called *visible*. A visible concept may generate a jumper for each item $I_i \in$ of the new object's items, when the extent of the visible concept intersects the full extent of item I_i derived from the data set yields a new extent that meets the minimum support threshold. The new extent will be the extent for a jumper. The jumper's intent will contain the visible concept's intent union $\{I_i\}$. However, the intent may be

[11] An underline with α superscript denotes an iceberg lattice with a minimum support threshold of α.

incomplete since additional items, yet to be processed in the new object's items, may generate the same extent. Thus, the extent intent pair is held in a temporary set until processing of all items of the new object is complete.

When iterating over the items of the new object, some of the items do not need to be tested. Clearly, any item $I_i \in$ the visible concept's intent will not generate a new extent. Furthermore, a $C_i \in$ child of the visible concept will not generate a new extent that meets the minimum support threshold. Omitting these items from the iteration will eliminate unneeded intersections and thereby improve performance[12].

The MAGALICE algorithm is provided in Algorithm 2.6. The ADD-OBJECT function provides the incremental insertion of a new object into an iceberg concept lattice. It first calls the ADD-OBJECT function of the underlying insertion algorithm to insert the object without regard to the minimum support (line 1). Post insertion processing then removes any concepts whose support does not meet the threshold from the lattice (lines 2 through 4). Lastly, a function FIND-FREQUENT-LOWER-COVERS is called to generate and add the jumpers (line 5).

The FIND-FREQUENT-LOWER-COVERS function begins by defining a set, denoted as Jumpers, to keep track of generated jumpers and then extract the set of visible concepts from the lattice (lines 6 and 7). The visible concepts are sorted by descending support to enable search and generation of jumpers from the bottom-up (line 8). The visible concepts are then processed. For each visible concept set of extent intent pairs, denoted as Candidates, is defined (line 10). Furthermore, a set of items, denoted as Pool, is initialized to the new object's items less the intent of the visible concept or any of its

[12] The authors provide additional tests to eliminate additional items.

Let Concept be a tuple {O, I, Children} where O is a set of object ids, I is a set of items, and Children is a list of child concepts.

```
ADD-OBJECT(G, O_i, I)   // Add object O_i together with its set of items I to lattice G
 1.   MGA-ADD-OBJECT(G, O_i, I)        // add O_i using MGA based algorithm
 2.   for each C_i ∈ G:
 3.       if |C_i.O| < MinSupp:
 4.           Remove C_i from G
 5.   FIND-FREQUENT-LOWER-COVERS(G, O_i, I)

FIND-FREQUENT-LOWER-COVERS(G, O_i, I)
 6.   Jumpers ← Ø                      // set of jumper concepts
 7.   V ← {C ∈ G | O_i ∈ C.O}          // get the visible concepts
 8.   Sort(V) by descending |I|
 9.   for each C_i ∈ V:
10.       Candidates ← Ø                // set of tuples {O, I}
11.       Th ← C_i.I                    // set of items accounted for
12.       for each C_Child ∈ C_i.Children:
13.           Th ← Th ∪ C_Child.I
14.       Pool ← I – Th                 // set of items to be processed
15.       for each I_i ∈ Pool:
16.           Extent ← C_i.O ∩ o(I_i)   // o(I_i) is the set O derived from relation ℛ
17.           if |Extent| = MinSupp:    // now meets the minimum support?
18.               C_Jumper ← C_i ∈ Jumpers ∧ C_i.O = Extent
19.               if C_Jumper ≠ ∅:      // jumper was previously created
20.                   Add  C_Jumper to C_i.Children
21.                   Pool ← Pool – C_Jumper.I
22.               else:
23.                   Candidate ←P ∈ Candidates ∧ P.O = Extent
24.                   if Candidate ≠ ∅:   // candidate was previously created
25.                       Add I_i to Candidate.I
26.                   else:
27.                       Add {Extent, C_i.I ∪ {I_i}} to Candidates
28.       for each Candidate_i ∈ Candidates:
29.           C_New ← new Concept(Candidate_i.O, Candidate_i.I)
30.           Add C_New to C_i.Children
31.           Add C_New to Jumpers
```

Algorithm 2.6: The MAGALICE algorithm[13]. (Rouane et al., 2004)

[13] At line 13, the authors include additional sets of items that do not need to be tested. Statements to derive these sets have been omitted for brevity. At line 17 the authors use a relative minimal support test instead of an absolute support test.

children (lines 11 through 14). Each item in the Pool may potentially generate a jumper. For each item in Pool, a candidate extent is generated (line 16). The candidate extent is tested to see if it meets the minimum support (line 17). If so, the candidate extent will generate a jumper provided it has not previously been encountered. Thus, the set of Jumpers is first examined. If there exists a jumper with the same extent, then that jumper will become a child of the visible concept (line 20). The items in the jumper's intent can be removed from the Pool since processing those items will identify the same jumper (line 21). If a jumper is not found, the set Candidates is examined. If there exists a candidate having an extent equal to the generated extent, then the test item is added to the candidate's intent; otherwise a new candidate pair is generated (lines 23 through 27). After processing all items in the Pool, any candidate pairs are turned into jumpers (lines 28 through 31).

The MAGALICE algorithm is evaluated against the Bordat[14] algorithm (Bordat, 1992) for two of the commonly used benchmark data sets. For one data set, MAGALICE demonstrated a fixed sized improvement which increased marginally as the supports were lowered. For the other data set, MAGALICE demonstrated increasing gains as the support were lowered. At low supports a gain by a factor of five is observed. Asymptotic complexity for a single insertion is determined to be $O(|\Delta\mathcal{L}|\ k^2 + m\ (m+k)\ \ell)$, where $\ell = |\mathcal{L}|$, $m = |\mathcal{I}|$, and $k = |\mathcal{C}|$.

[14] An early lattice construction algorithm.

2.9 Other Lattice Construction Algorithms

Lindig and Datensystene (2000) propose a simple batch algorithm to construct a concept lattice. A batch algorithm constructs a concept lattice from a complete formal context. That is, the algorithm is not constrained to object by object (or item by item) insertion and is free to query any point in the formal context as needed. The proposed algorithm 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. The asymptotic complexity of the algorithm is $O(\ell m k^2)$, where $\ell = |\mathcal{L}|$, $m = |\mathcal{I}|$, and $k = |\mathcal{C}|$. Empirical evaluation involving several synthetic generate data sets against the NextConcept[15] algorithm (Ganter, 1984) is provided.

Valtchev, Missaoui, and Lebrun (2002b) provide a divide and conquer approach to lattice construction. The input data set is first partitioned into two sets, either based on items or objects. A concept lattice is constructed for each set and the resulting lattices are merged. The asymptotic complexity is evaluated to be $O(m (m+k) \ell \log \ell)$, where $\ell = |\mathcal{L}|$, $m = |\mathcal{I}|$, and $k = |\mathcal{C}|$. However, this may only be realized for contexts that exhibit linear growth in size of lattice with respect to the number of objects.

Nourine and Raynaud (2002) offer an incremental lattice construction algorithm based on a lexicographic tree. The algorithm is an incremental version of earlier work (Nourine, & Raynaud, 1999). The lexicographic is a trie data structure similar to the one used by GALICIA-T, except the roles of intent and extent are reversed. That is, each edge in the trie denotes an object and nodes corresponding to concepts are augmented

[15] NextConcept is an early batch based algorithm.

with an item list. The common prefixes in object id lists thus share the same branch. The incremental insertion is performed on an item by item basis by using a union operation on object ids against each of the pre-existing nodes in the trie, and determining if the result is present in the trie. If the result is present and augmented with an item list, the item is added to the node; otherwise it will be a new concept. Branches and nodes will be added to the trie as needed. The node representing the new concept will be augmented with an item set consisting of the item. The insertion process also identifies a parent concept for each new concept. The remaining task is to identify the children of each new concept and link it into the lattice. Identification of children is performed by test union and count procedure for each item in $\mathcal{I}$ that is $\notin$ new concept's intent. When linking new concept into the lattice, a test is made to remove transitive edges. The Nourine and Raynaud algorithm is evaluated to have an $O(m\,(m+k)\,l)$ complexity, where $l = |\mathcal{L}|$, $m = |\mathcal{I}|$, and $k = |\mathcal{O}|$. No empirical support is provided.

Kuznetsov and Obiedkov (2002) provide a comparative survey of several lattice construction algorithms. Algorithms include: GMA (Godin et al., 1995), Lindig and Datensystene (2000), Nourine and Raynaud (2002), TITANIC (Stumme et al., 2002), Valtchev et al. (2002) divide and conquer, Bordat (1992), Close by One (Kuznetsov, 1993), Chein (1969), and Norris (1978). Bordat, Close by One, and Chein are batch based algorithms. Norris is an incremental algorithm based on essentially the same theoretical principles of the Close by One algorithm. An overview of the theory and implementation of each algorithm is provided including asymptotic complexity. A diverse set of randomly generated data sets each conforming to specified properties (e.g., number objects, number attributes, and density) are used to benchmark each algorithm in

addition to one real-world data set. Findings indicate that there is no "best" algorithm and the each algorithm exhibit different performance depending on the data set. The GMA algorithm is a good choice for sparse data sets, and batch algorithms are good for dense data sets. The Nourine and Raynaud algorithm was not the winning algorithm for any data set even though it has the best asymptotic complexity[16]. Valtchev et al. (2002) arrive at the same conclusions. Furthermore, Valtchev et al. state that comparison of lattice construction algorithms based on asymptotic complexity is a "delicate task". Their study reports that the GMA algorithm has good performance for data set with density[17] less than 0.10, but lags with densities greater than 0.50.

SPROUT (Choi, 2006) is a recent batch-based lattice construction algorithm that provides an option to build an iceberg lattice. It is similar in theory to other batch-based algorithms, such as Lindig and Datensystene (2000). It begins by creating the top concept and then generates children for a concept 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 repeats for each new concept. The author claims the algorithm is faster that any known algorithm including CHARM-L but provides no empirical evidence. Only a single test case for a small lattice ($|\mathcal{L}| = 530$) is cited.

[16] Authors note that their implementation may of impeded performance.

[17] 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.

2.10 A Generic Approach to Incremental Lattice Construction

Valtchev, Rouane, and Missaoui (2003b) developed a generic approach for the development of an incremental lattice construction algorithm. Valtchev et al. theorizes that all past incremental lattice construction algorithms involve:

i) identification concepts whose extent will be modified to include the new object,

ii) identification of generator concepts that are used to generate a new concepts,

iii) inclusion of the new object into those concepts identified to be modified,

iv) generation of the new concepts,

v) linkage of each new concept to its generator and upper covers, and

vi) the removal of resulting transitive links;

albeit the sequencing and techniques used for each task may differ between the algorithms. The identification of modified and generator concepts can be accomplished by partitioning the existing concepts of the lattice into *equivalence classes*. Each equivalence class is composed of concepts whose intent intersect the next object's items has the same value. For example, Figure 2.4 depicts the equivalence classes for the last three lattice insertions given in Figure 2.2. Each set of concepts composing an equivalence class is enclosed by a dotted line. Underlined items within each concept indicate items that are in the intersection. The maximum concept in each equivalence class is either a modifier or generator depending if its intent is a subset of the next object's items. When generating new concepts, the intent will be the equivalence class intersection set and the extent will be the generator's extent union the next object. After generating a new concept, it will be a parent of its generator and its parents can be derived from the generator's ancestors. Finally, transitive links that violate the lattice

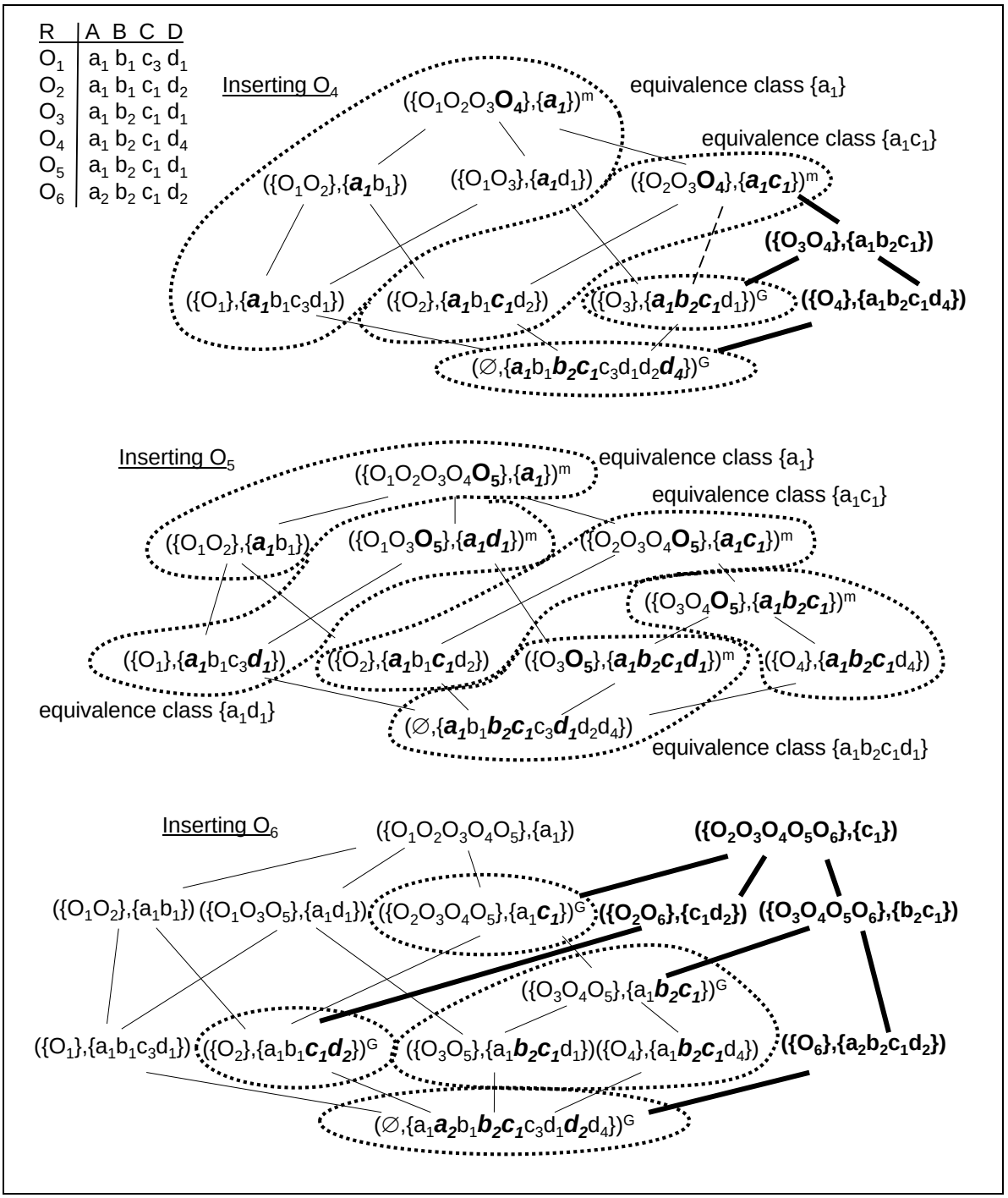


Figure 2.4: Object insertions into a concept lattice depicting equivalence classes. The lattices correspond to the progression shown in Figure 2.2 for the last three object insertions. Bold italic text within an existing concept indicates the intersection set forming the equivalence class. Bold text and lines indicates new elements, ^{G} a generator concept, ^{m} a modified concept, enclosed dotted line an equivalence class, and dashed line is a removed link.

connection property can be identified by intersecting the set of modified concepts with the parents of a generator. Such transitive links can be removed.

The complete generic algorithm for incremental insertions is given in Algorithm 2.7. Lines 6 through 13 provide identification of modified concepts and generators. Following this, the modified concepts are updated with the new object (lines 14 and 15), new concepts are generated (lines 16 through 18), the new concepts are linked into the lattice (lines 19 through 21), and any resulting transitive links are removed (lines 22 through 24). The algorithm provides a concise statement of the work to be performed as an ordered sequence of the major tasks.

Beyond formulation of the generic algorithm, Valtchev et al. provide a discussion of applying various techniques from past work to the individual tasks and derive a concrete algorithm. The theoretical asymptotic complexity for the algorithm is $O((m+k)\,k\,\ell)$, where $\ell = |\mathcal{L}|$, $m = |\mathcal{I}|$, and $k = |\mathcal{C}|$. This matches the best known theoretical complexity. At the time of the paper, the concrete algorithm had not been implemented and thus no empirical data is available.

Let Concept be a tuple {O, I, Parents} where O is a set of object ids, I is a set of items, and Parents is a list of parent concepts.

```
COMPUTE_LATTICE_INC(G, O_i, I)  // add object O_i together with its set
                                // of items I to lattice G
 1.   Q[ ] ←∅                          // vector of equivalence class sets
 2.   M ←∅                             // set of modified concepts
 3.   G ←∅                             // set of generator concepts
 4.   N ←∅                             // set of new concepts
 5.
 6.   for each C_i ∈ G:                // put concepts into their equivalence class
 7.       Add C_i to Q[I ∩ C.I]
 8.   for each Q_i ∈ Q[ ]:
 9.       C_i ← Maximal concept in Q_i
10.       if C_i ⊆ I:
11.           Add C_i to M              // C_i will be a modified concept
12.       else:
13.           Add C_i to G              // C_i is a generator concept
14.   for each C_Modified ∈ M:          // update the modified concepts
15.       C_Modified.O ← C_Modified.O ∪ {O_i}
16.   for each C_Generator ∈ G:         // generate new concepts
17.       C_New ← new Concept(C_Generator.O ∪ {O_i}, Q(C_Generator))
18.       Add C_New to N
19.   for each C_New ∈ N:               // link the new concepts into the lattice
20.       Add C_New to its C_Generator.Parents
21.       COMPUTE_UPPER_COVERS(C_New, its C_Generator)
22.   for each C_Generator ∈ G:         // identify and remove transitive links
23.       for each C_Parent ∈ C_Generator.Parents ∩ M:
24.           Remove C_Parent from C_Generator.Parents
```

Algorithm 2.7: Generic incremental lattice insertion algorithm. (Valtchev et al., 2003b)

2.11 Summary of Literature

Since surfacing in 1993, association rule mining has been a major area of research. However, a large portion of activity has been directed toward mining of frequent items sets. Notable algorithms include CHARM, CLOSET, TITANIC, and CLOSET+. While significant progress has been made, frequent item set mining has fallen short of the overall objective of mining association rules. The frequent item set miners fail to identify the upper covers of each closed frequent item set and thus are incapable of producing a reasonably sized set of association rules.

Formal concept construction algorithms have been another strong area of research. Noteworthy algorithms include GMA, Nourine and Raynaud, Lindig and Datensystene, and Valtchev et al. divide and conquer. These algorithms are effective in constructing a concept lattice. They successfully derive the set of closed items sets together with their upper covers. However, these algorithms construct a lattice for all closed item sets and not just those which are frequent. Some algorithms are incremental while others are batch. Batch-based algorithms are not constrained to object by object (or item by item) insertion and are free to inquire any point in the formal context as needed. The best asymptotic complexity is $O(\boldsymbol{m}\ (\boldsymbol{m}+\boldsymbol{k})\ \boldsymbol{\ell})$, where $\boldsymbol{\ell} = |\mathfrak{L}|$, $\boldsymbol{m} = |\mathfrak{I}|$, and $\boldsymbol{k} = |\mathfrak{O}|$. However, benchmarks have proven that asymptotic complexity may not be a good means for comparison. There is no known "best" algorithm. The GMA algorithm is considered to be a good algorithm for data sets with density[18] less than 0.10.

An approach to association rule mining is to use a frequent closed item set mining algorithm to generate the set of frequent items and then use a post-mining lattice

[18] For formal context $K\{\mathfrak{G}, \mathfrak{O}, \mathfrak{R}\}$, the density of $\mathfrak{R} = |\mathfrak{R}| / (|\mathfrak{G}| \times |\mathfrak{O}|)$ where $|\mathfrak{R}|$ is the total number of items for all objects

construction algorithm, such as the VML algorithm, to construct the upper covers. Such approach will be bounded by the asymptotic complexity of the post-mining lattice construction algorithm. For the VML algorithm, its complexity is $O(\ell \, w(\mathcal{L}) \, deg(\mathcal{L}) \, m)$, where $\ell = |\mathcal{L}|$, $w(\mathcal{L})$ = width of $\mathcal{L}$, $deg(\mathcal{L})$ = degree of $\mathcal{L}$, and $m = |\mathcal{I}|$.

Of all algorithms reviewed, only three: CHARM-L, MAGALICE, and SPROUT, provide construction of an iceberg concept lattice. Of these CHARM-L and MAGALICE are considered to be the serious contenders for this study. SPROUT is discounted due to the lack of empirical support and batch classification. CHARM-L constructs a concept lattice as a separate data structure from its itemset-oidset tree. Construction of the lattice is, however, an integral part of its processing. MAGALICE uses an underlying incremental lattice construction algorithm, such as GMA, to insert a new object without regard to frequency. Afterwards, it prunes the lattice of infrequent concepts and regenerates concepts that become frequent.

The major works presented in this chapter can be partitioned into three main camps engaged in research of association rule mining. They are:

i) Agrawal, Han, Hsiao, Pei, Srikant, Wang, and Zaki contributing research on frequent item set mining (Agrawal et al., 1993, Agrawal, & Srikant, 1994, Srikant, & Agrawal, 1996, Kamber et al., 1997, Pei et al., 2000, Zaki, 2000, Zaki, & Hsiao, 2002, Wang et al., 2003, Zaki, & Hsiao, 2005, and Han, & Kamber, 2006)

ii) Ganter, Lakha, Pasquier Taouil, and Stumme contributing theory on formal concept analysis and its application to association rule mining (Ganter 1984, Ganter, & Wille, 1997, Pasquier et al, 1999a, Pasquier et al., 1999b, Pasquier 2000, Stumme et al,, 2000, Stumme et al., 2001a, Stumme et al., 2001b, Stumme et al., 2002, Ganter et al., 2005, and Lakha, & Stumme, 2005), and

iii) Godin, Missaoui, Nourine, Raynaud, Rouane, Valtchev contributing research on lattice construction algorithms (Godin et al., 1995, Nourine, & Raynaud, 1999, Valtchev et al., 2000, Nourine, & Raynaud, 2002, Valtchev et al., 2002a, Valtchev et al., 2002b, Valtchev et al., 2003a, Valtchev et al., 2003b, Valtchev et al., 2004, and Rouane et al., 2004).

Of these, Stumme et al. provide a compelling argument for using iceberg concept lattice to mine association rules. It is interesting to note that all of Stumme's papers depict a concept lattice using a compressed notation similar to Figure 1.3. That is, items are only listed at their maximal position. This together with other information, such as support and confidence drops, is sufficient to efficiently extract a basis of association rules. A majority of papers by other authors depict concepts with their full intent and extent. While Stumme's papers provide the theoretical foundation for a compress lattice structure, there are no known algorithms that directly used the compress structure. Stumme's own offering, TITANIC (Stumme et al., 2002), does not construct a lattice, let alone one using a compressed structure.

Of the incremental lattice construction algorithms presented in this chapter, it is interesting to note that the GMA (Godin et al., 1995), Valtchev et al. (2002) divide and conquer, Valtchev et al. (2003b) generic lattice construction algorithm, and MAGALICE (Rouane et al., 2004) algorithms use the lattice data structure to drive the processing of the algorithms, where as GALICIA-T (Valtchev et al., 2002), Nourine and Raynaud (2002), and CHARM-L (Zaki, & Hsiao, 2005) use an alternate data structure to drive the algorithm and construct the lattice as a subsequent, although integrated, step. The main data structure for GALICIA-T is a Trie with items on the edges. Nourine and Raynaud use a lexicographic tree that is similar to the Trie of GALICIA-T, except the roles of intent and extent are reversed. CHARM-L uses its itemset-tidset tree.

Given this review of literature a strategy for developing new iceberg lattice construction algorithms is to:

i) Adapt the GMA lattice construction algorithm to directly construct an iceberg lattice. It is an algorithm that directly uses the lattice structure to drive its process, thereby leveraging lattice theory. Using GMA as the starting point will avoid the expensive reconstruction of discarded concepts incurred with the MAGLICE algorithm. The vertical representation of CHARM and CHARM-L may provide insights to accomplishing this task.

ii) Adopt a compressed lattice structure along the lines of the theory presented by Stumme. This may address memory concerns by minimizing the space consumed by the concepts of the lattice. Furthermore, the compressed lattice structure is of the form that basis extraction algorithms, such as Stumme et al. (2001b), can be readily used to mine a set of association rules that is constrained to a size that can be exploited by the end user.

iii) Incorporate features of other algorithms to provide further improvement. For example CHARM and Closet+ prescribe a sort order as a heuristic to improve performance. Selection of an appropriate sort order may prove effective in the new algorithm. Valtchev et al. generic lattice construction algorithm, CHARM-L, GALICIA-T, and others will be reviewed during development to aid in identifying additional opportunities for improvement.

Chapter 3

Methodology

3.1 Introduction

While GMA (Godin et al., 1995) 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 and interchanging the roles of the set of object ids (O) and the set of items (I) results in an algorithm that can directly 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. Algorithm 3.1 provides the GMA algorithm with these two modifications applied (see Section 2.5 Incremental Lattice Construction – Missaoui, Godin, and Alaoui Algorithm for a description of the algorihm theory and function). Line 22 is the predicate to ensure the minimum support threshold has been met. A complete implementation of Algorithm 3.1, written in Java, is given in Appendix A.

Let Concept be a tuple {I, O, Children} where I is a list of items, O is a list of object ids, and Children a list of child concepts.

Let C_{Bottom} be the infimum of a concept lattice G, G contain a reference to all concepts

```
ADD(I_i, O)              // Add item id I_i together with its set of object ids O
 1.  if C_Bottom = Ø:                                  // special case, empty lattice
 2.      C_Bottom ← new Concept ({I_i}, O)
 3.  else:
 4.      if O ⊄ C_Bottom.O:                            // check O ids not in lattice
 5.          if C_Bottom.I = Ø:                        // a generated bottom?
 6.              C_Bottom.O ← C_Bottom.O ∪ O
 7.          else:                                     // generate the new bottom
 8.              C_New ← new Concept (Ø, O)
 9.              Add C_New to C_Bottom.Children
10.              C_Bottom ← C_New
11.
12.      Processed ← Ø    // a vector of sets of concepts indexed by the cardinality
13.                       // of an intersection set
14.      for each C_i ∈ G in ascending |O| order:
15.          if C_i.O ⊆ O:                             // is C_i a modified concept?
16.              Add I_i to C_i.I
17.              Add C_i to Processed[|C_i.O|]         // possible parent
18.              if C_i.O = O:                         // no more processing?
19.                  return
20.          else:                                     // existing concept
21.              Intersect ← C_i.O ∩ O
22.              if |Intersect| ≥ MinSupp:
23.                  if ¬∃ C_j ∈ Processed[|Intersect|] | C_j.O = Intersect:  // is C_j a gen?
24.                      C_New ← new Concept (C_i.I ∪ {I_i}, Intersect)
25.                      Add C_New to Processed[|Intersect|]
26.                      Add C_i to C_New.Children     // modify edges
27.                      for each C_k ∈ Processed ∧ |C_k.O| < |Intersect|:
28.                          if C_k.O ⊂ Intersect:     // is C_k potential parent?
29.                              Parent ← TRUE
30.                              for each C_Child ∈ C_k.Children ∧ Parent = TRUE:
31.                                  if C_Child.O ⊂ Intersect:
32.                                      Parent ←FALSE
33.                              if Parent = TRUE:
34.                                  if C_i ∈ C_k.Children:
35.                                      Remove C_i from C_k.Children
36.                                  Add C_New to C_k.Children
37.                      if |Intersect| = |O|:         // last valid generator ?
38.                          return                    // no more work to do
```

Algorithm 3.1: The GMA algorithm modified to construct an iceberg lattice.

Preliminary tests[19] proved the modified GMA algorithm functioned correctly. The Mushroom[20] 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 seconds, 0.39 seconds, 7.17 seconds, 160.28 seconds, and 1,198.08 seconds[21]. 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 algorithm significantly degraded by an order of magnitude as support is lowered. Detailed measurements of the runtime and memory usage are provided in Chapter 4.

While the modified GMA algorithm does function correctly, its efficiency cannot compete with the leading association rule mining algorithms. This chapter describes the development of the Quick Iceberg Concept Lattice (QuICL – pronounced kwi-kəl) algorithms. These algorithms provide incremental construction of a concept lattice along the lines of the GMA algorithm, 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 instead of iteration 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.

---

[19] Prelimnary tests are executions of the algorithm during development. These tests are not performed under controlled conditions. The timings in this chapter are given to illustrate the progression during algorithm development. They are stated in seconds to provde a standard unit for comparison. See Chapter 4 Results for controlled measurements and interpretation.

[20] An often used data set for association rule mining and FCA. See Section 4.2.

[21] Execution times were obtained using an unsorted data. Sort order was later found to have a large effect on execution time.

The QuICL algorithm has three derivations: Oid-Full, Oid-Less, and Oid-Trie. In the first derivation, all of the concepts in the concept lattice retain a complete list of the object ids (oids), hence the name "Oid-Full". For the Oid-Less derivation, the concepts do not retain the list of oids between the incremental insertions. Some temporary lists of object ids are created and discarded during the insertion process. The Oid-Trie derivation is a compromise between Oid-Full and Oid-Less. Instead of eliminating the oid lists, it utilizes a trie data structure to store the oids thereby reducing memory requirements. In addition to reducing memory usage, the trie data structure also enables a few performance enhancements (e.g., intersect operations can terminate early upon encountering a common branch).

3.2 Steps Toward 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 a concept 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 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 Chapter 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 association rule mining, 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 or extent is not updated. It simply holds references to parent concepts.

The previously mentioned changes will simplify the processing in the GMA algorithm without any loss of necessary information needed to formulate association rules. The steps of the GMA algorithm 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 also be eliminated. There is, however, one small side effect. In the event an item exists common to all objects, the GMA algorithm 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.

Given the proposed modifications to the lattice structure, Figure 3.1 depicts the progression of incremental item insertions of the data in relation R into an inverted concept lattice. The final lattice of Figure 3.1 is the inverted form of the lattice given in Figure 1.1. Before presenting an initial algorithm to construct a lattice using the proposed structure, a few observations in the progression shown are noteworthy:

- Insertion of an item whose extent = the extent of a concept C within the lattice is accomplished by simply adding the item to C. C can be found by traversing the

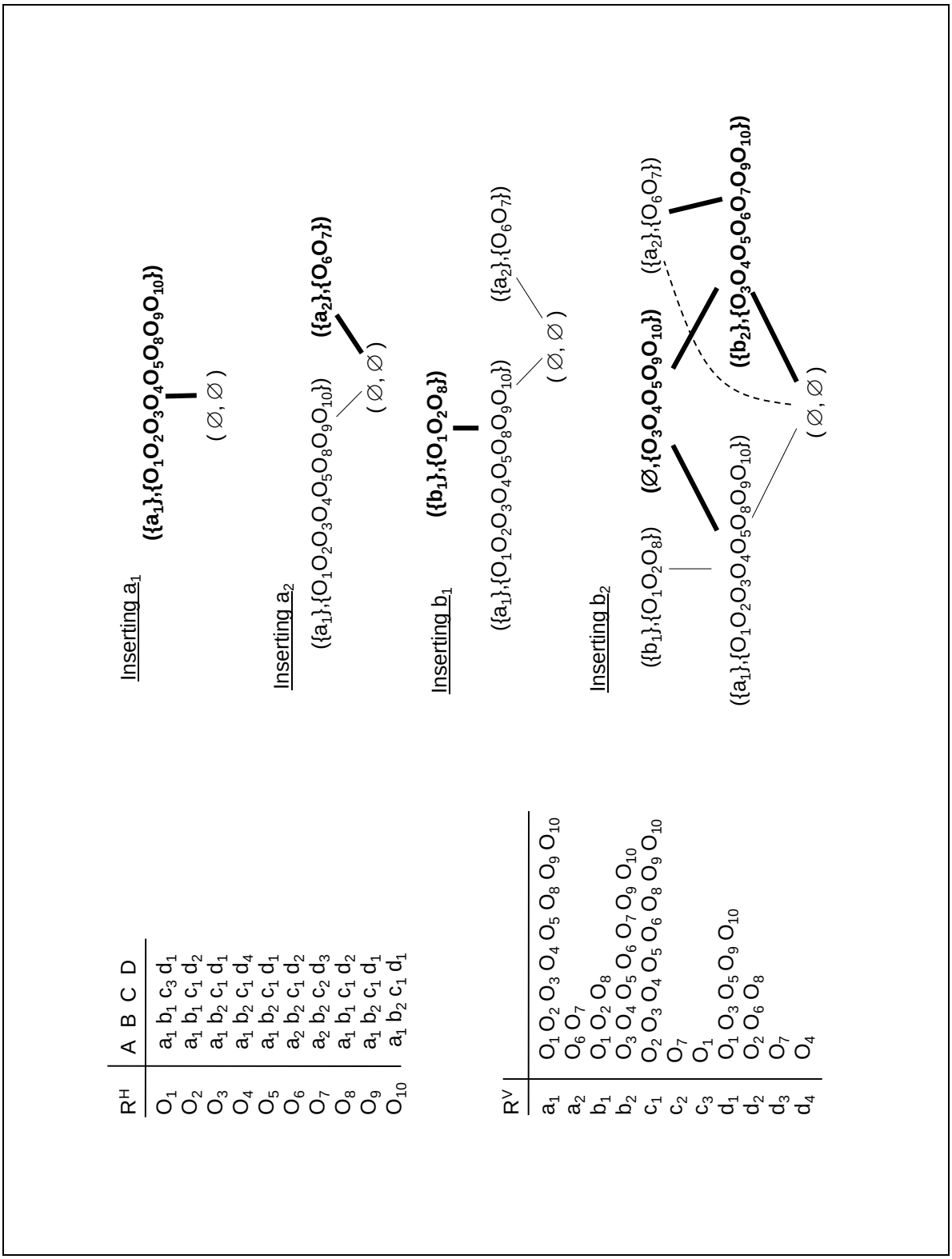


Figure 3.1: Progression of incremental item insertion into a concept lattice. Bold text and weighted lines identify new elements. Dashed lines indicate removed elements.

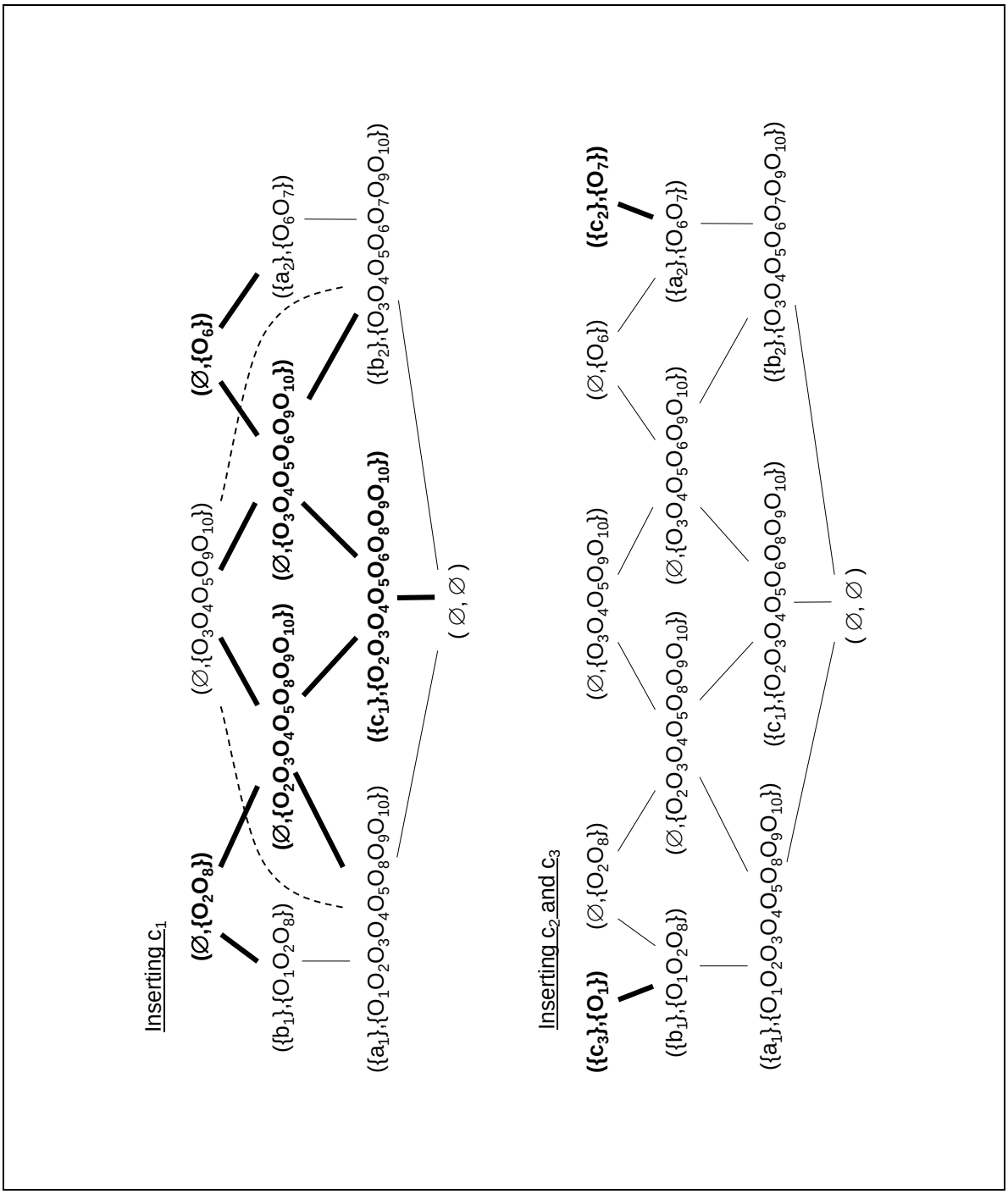


Figure 3.1 continued: Progression of incremental item insertion into a concept lattice. Bold text and lines identify new elements. Dashed lines indicate removed elements.

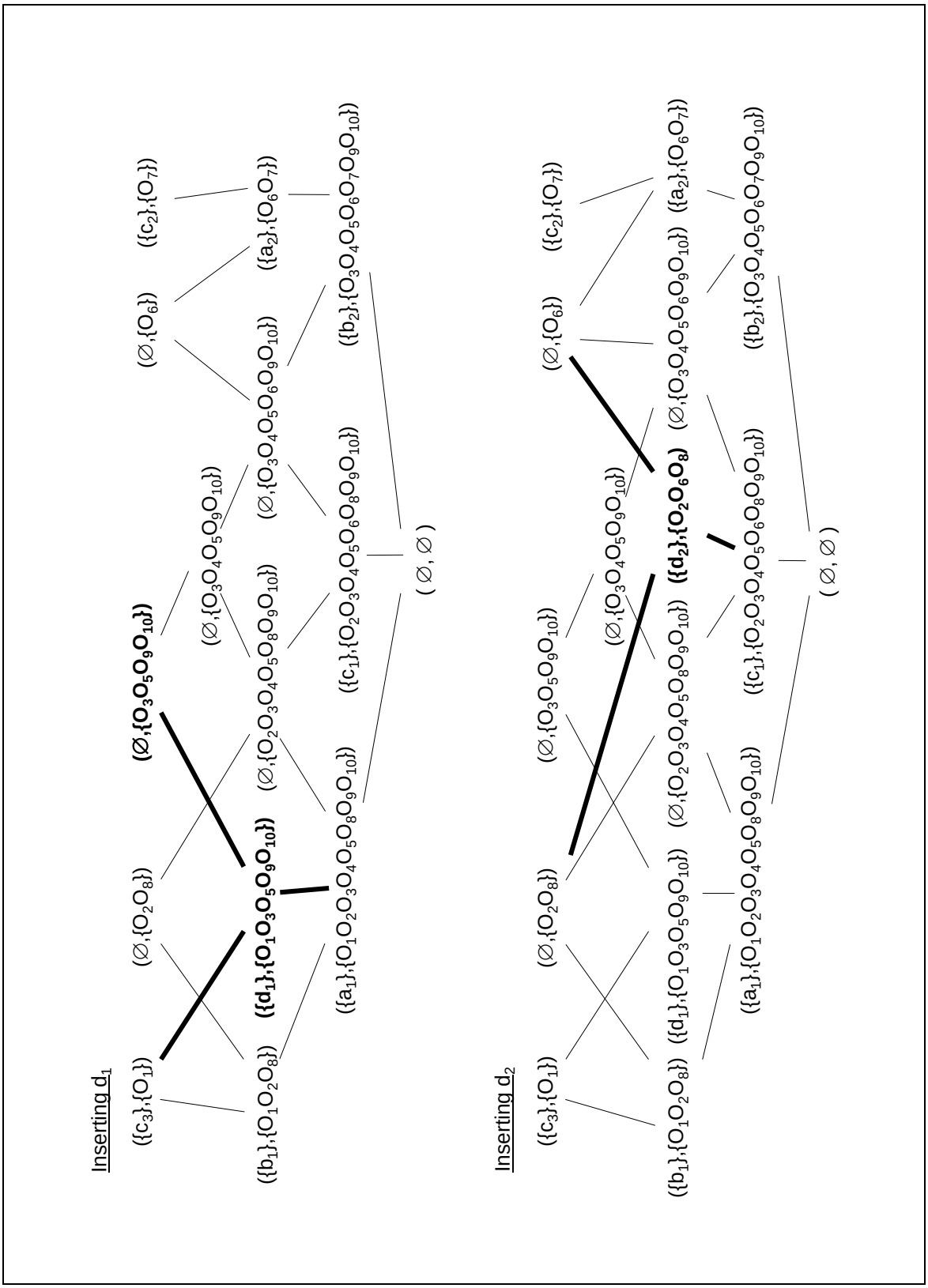


Figure 3.1 continued: Progression of incremental item insertion into a concept lattice. Bold text and lines identify new elements.

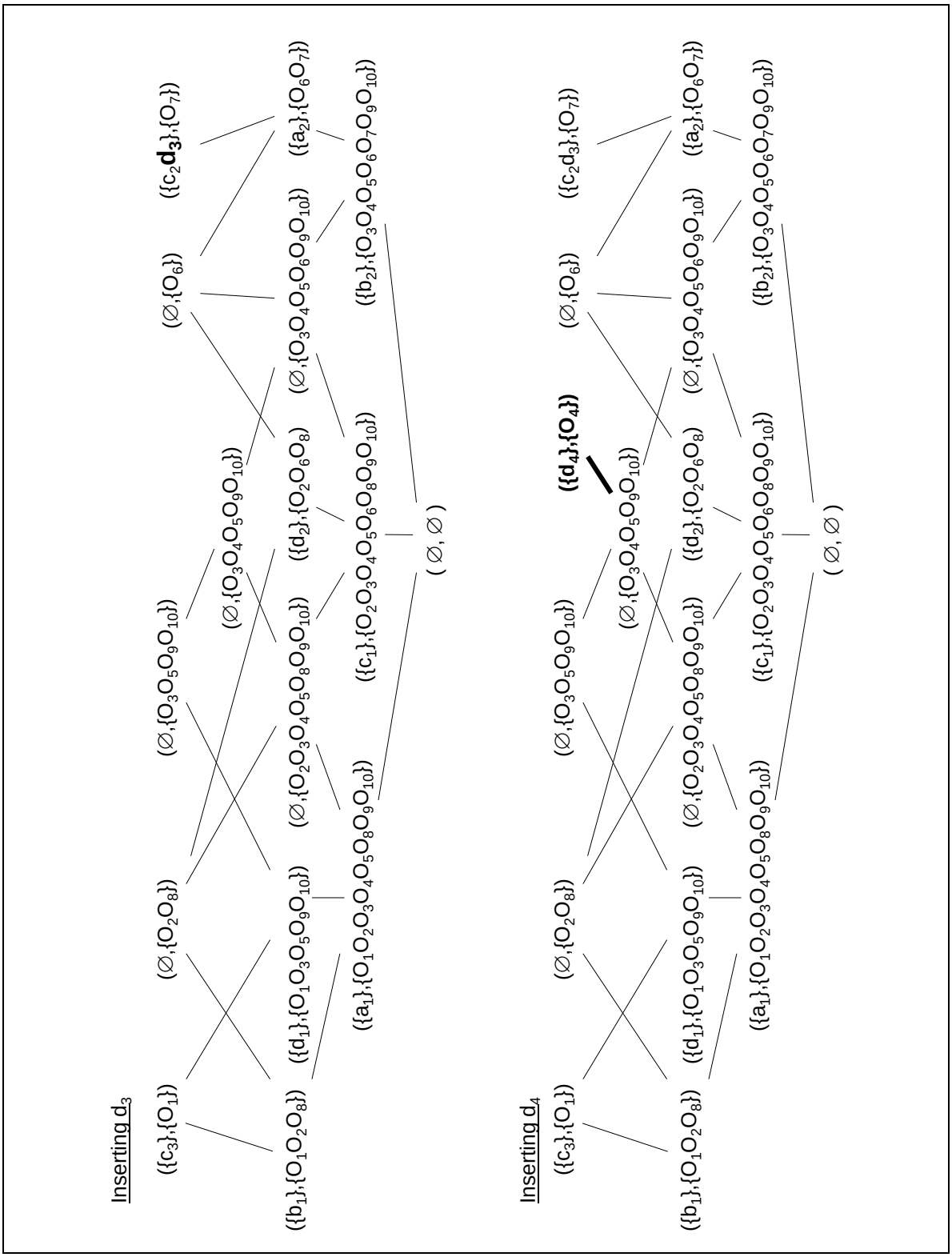


Figure 3.1 continued: Progression of incremental item insertion into a concept lattice. Bold text and lines identify new elements.

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 3.1.

- 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 3.1 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 a $C_p \cap$ 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 3.1. 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 3.1. 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 the GMA algorithm can be formulated. For each insertion, the GMA algorithm processes all concepts in a top-down, level-wise manner to modify existing concepts and to generate new concepts. The top-down traversal is used to facilitate correct identification of generators and limit the search for parent concepts. The above 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., base concept). Furthermore, recursion can be used to find, or if needed create, the parent concepts.

Algorithm 3.2 presents an incremental insertion algorithm to construct a concept lattice. For this algorithm, a concept lattice is represented as a set of concepts linked only by references to parents. The data structure for 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. 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) processes the list of concepts identified by the first phase and links the new concept into the lattice.

Both the navigation phase and link phase recursively call the INSERT function as needed.

INSERT begins by defining an empty list of tuples consisting of a type indicator with values SUPERSET or INTERSECT, an intersection set, and a reference to the

```
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                            // the lattice

INSERT(C_Base, I_i, O)
  5.    ToProcessList ← ∅  // list of tuples {Type, Concept, O} with
  6.                       // Type ∈ {SUPERSET, INTERSECT}, Concept is a
  7.                       // reference to the intersecting concept, and O
  8.                       // a set of object ids resulting from an intersection
  9.
 10.    for each C_Parent ∈ of C_Base.Parents:     // prepare-search phase
 11.        if  O = C_Parent.O:
 12.            Add I_i to C_Parent.I
 13.            return C_Parent                    // processing complete
 14.        else if O ⊂ C_Parent.O:
 15.            return INSERT (C_Parent, I_i, O)   // recurse using C_Parent as new C_Base
 16.        else if O ⊃ C_Parent.O:
 17.            Add {SUPERSET, C_Parent, C_Parent.O} to ToProcessList
 18.        else if O ∩ C.O ≠ ∅:
 19.            Add {INTERSECT, C_Parent, O ∩ C_Parent.O} to ToProcessList
 20.
 21.                                               // intentionally left blank
 22.
 23.    C_New ← New Concept({I_i}, O)              // create the new concept
 24.
 25.    for each T_i ∈ ToProcessList:              // link phase to link in C_New
 26.        if  T_i.Type = SUPERSET:
 27.            Remove T_i.Concept from C_Base.Parents
 28.            Add T_i.Concept  to C_New.Parents
 29.        else if  T_i.Type = INTERSECT:
 30.            C_Parent ← INSERT (T_i.Concept, ∅, T_i.O)
 31.            Add C_Parent to C_New.Parents
 32.
 33.    Add C_New to C_Base.Parents
 34.
 35.    return C_New
```

Algorithm 3.2: A recursive incremental lattice construction algorithm.

concept that generated the intersection set (line 5). This list is populated during the search-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 SUPERSET indicates O is a superset of the extent of the parent concept. Type INTERSECT indicates that O is neither superset nor subset. INSERT proceeds to compare O with the extent of each parent of the base concept (lines 10 through 19). 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 11 and 12). The insertion is complete. For purposes discussed later, INSERT returns a reference to the modified concept (line 13). If O is a subset the extent of any parent concept then INSERT recurses using the parent as the new base concept (lines 14 and 15). 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 SUPERSET, a reference to the parent concept, and the parent's extent is added to the ToProcessList for later processing (lines 16 and 17). 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 INTERSECT, a reference to the parent concept, and O intersect the extent of the parent is added to the ToProcessList (lines 18 and 19).

If comparison of O with the extents of all parent concepts of the base concept 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 23). 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 SUPERSET

concepts in the ToProcessList, a sibling to the INTERSECT concepts, and a parent to the base concept. The final phase establishes these parent-child links.

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 SUPERSET indicator, the parent will no longer be a parent of the base concept (line 26). 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 27) and added to the new concept's parents (line 28). 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 INTERSECT indicator. 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 30). The concept returned by the recursive call is added to the new concept's parents (line 31). Finally, the new concept is added to the parents of the base concept and the new concept is returned (lines 33 and 35)

3.3 Walk Through of the Algorithm Execution

Figure 3.2 provides a sample walkthrough of executing Algorithm 3.2. This walkthrough corresponds to the insertion of b_2 in Figure 3.1. 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 prepare-search 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 INTERSECT tuple to the ToProcessList. The tuple contains the intersection set of $\{O_3O_4O_5O_9O_{10}\}$ 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 SUPERSET tuple to the ToProcessList. Since the prepare-search phase did not encounter a parent concept where $O \subseteq$ parent's extent, a new concept $(\{b_2\}, \{O_3O_<O_5O_6O_7O_9O_{10}\})$ is created and the tuples of the ToProcessList are then processed. Processing the {INTERSECT, $(\{a_1\}, \{O_1O_2O_3O_4O_5O_8O_9O_{10}\})$, $\{O_3O_4O_5O_9O_{10}\}$} 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 prepare-search 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 {SUPERSET, ({a_2}, {O_6O_7}), {O_6O_7}} tuple involves removing ({a_2}, {O_6O_7}) from the parents of the C_{Base}, being {∅, ∅}, 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 given in Figure 3.2 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 of Figure 3.1. For example, insertion of d_3 will call INSERT with C_{Base} referencing the bottom concept {∅, ∅}, $I_i = d_3$, and $O = \{O_7\}$. The prepare-search 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 prepare-search phase of the recursive call will further recurse with C_{Base} referencing the concept ({a_2}, {O_6O_7}). The prepare-search phase of this recursive call will encounter a parent concept whose extent = 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.

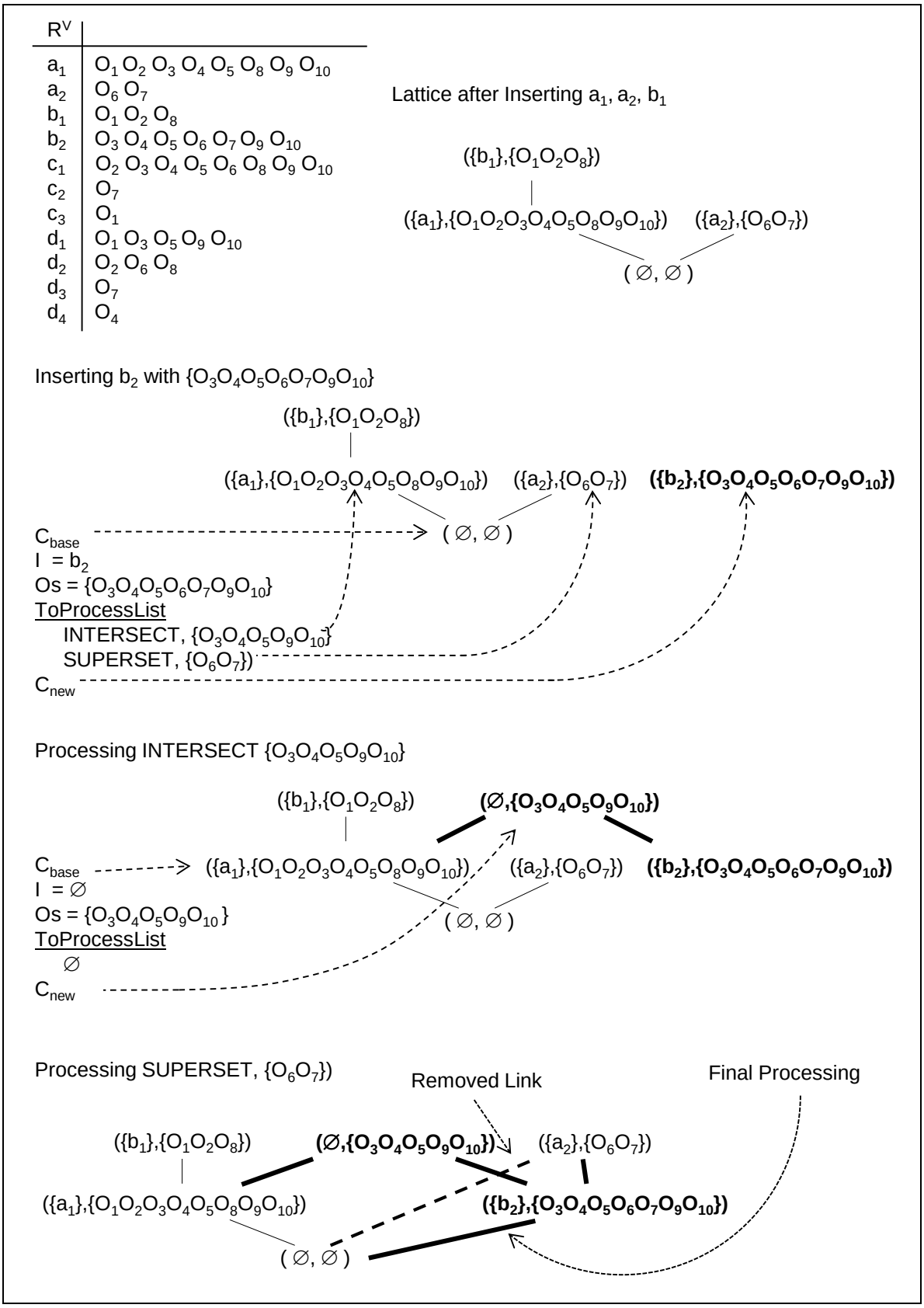


Figure 3.2: Sample walkthrough of Algorithm 3.2 execution.

3.4 Proof of Algorithm Correctness

Given a lattice $\mathcal{L}$ and a new item I_i with its set of object O, an incremental insertion algorithm is correct if it meets these requirements:

1) if $\exists\ C_i \in \mathcal{L}$ | extent of C_i = O, then insertion is completed by adding I_i to the intent of C_i, or

2) if $\neg\exists\ C_i \in \mathcal{L}$ | extent of C_i = O, then a new concept C_{New} with intent $\{I_i\}$ and extent O must be created and inserted into the lattice such that:

   i. if $\exists\ C_b \in \mathcal{L} \mid C_b > C_{New} \wedge \neg\exists\ C_3 \in \mathcal{L} \mid C_b > C_3 > C_{New}$, then C_{New} will be a parent of C_b,

   ii. if $\neg\exists\ C_b \in \mathcal{L} \mid C_b > C_{New}$, then C_{New} will be a parent of bottom concept,

   iii. $\forall\ C_p \in \mathcal{L} \mid C_{New} > C_p \wedge \neg\exists\ C_3 \in \mathcal{L} \mid C_{New} > C_3 > C_p$, C_p will be a parent of C_{New},

   iv. $\forall\ C_s \in \mathcal{L}$ | extent of $C_s \not\subset O \wedge$ extent of $C_s \cap O \neq \varnothing \wedge \neg\exists\ C_3 \in \mathcal{L}$ | $C_3 > C_s \wedge$ extent of $C_s \cap O$ = extent of $C_3 \cap O$, another new concept C_{New}' with empty intent and an extent of $C_s \cap O$ must be inserted into the lattice with C_{New}' as a parent of both C_{New} and C_s, and

   v. the resulting lattice satisfies the lattice connection property: a connection exists between two concepts C_1 and C_2 provided $C_1 < C_2$ and there is no concept C_3 for which $C_1 < C_3 < C_2$.

Algorithm 3.2 fulfills requirement 1.

Proof: Let $\mathcal{L}_b$ be a sub-lattice with concept C_b as its bottom concept. If $\exists\ C_i \in \mathcal{L}_b$ | extent of C_i = O $\wedge$ $C_i \neq C_b$ then C_i will be an ancestor of C_b. If a parent of C_b has an extent = O, then that parent concept is C_i; otherwise C_i can be found by searching the sub-lattice defined by any parent concept where extent of the concept $\supset$ O. The INSERT function of Algorithm 3.2 examines each parent concept of a base concept C_b. If a parent of the C_b has an extent = O then that parent concept is the concept to which I_i is added. If a parent concept has an extent $\supset$ O, then that parent is not the concept to which I_i is added, but the concept with extent = O will be an ancestor of that parent concept. The INSERT function recurses using the parent concept as C_b. Algorithm 3.2 initially calls the INSERT function using the bottom concept of the concept lattice as C_b.

Algorithm 3.2 creates a new concept under the conditions of requirement 2.

Proof: Let $\mathcal{L}_b$ be a sub-lattice with concept C_b as its bottom concept. If $\neg\exists\ C_i \in \mathcal{L}_b \mid$ extent of $C_i = O \wedge C_i \neq C_b$ then traversing any path where extent of a concept $\supset O$ will encounter a concept that does not have any parents whose extent $\supseteq O$. The INSERT function of Algorithm 3.2 examines each parent concept of a base concept C_b. If a parent concept has an extent $\supset O$, then the INSERT function of Algorithm 3.2 algorithm recurses using the parent concept as C_b. If none of the parents have an extent $\supseteq O$, then the INSERT function creates a new concept with an intent $\{I_i\}$ and extent O. Algorithm 3.2 initially calls the INSERT function using the bottom concept of the concept lattice as C_b.

Algorithm 3.2 fulfills requirement 2.i.

Proof: Let if $\mathcal{L}_b$ be a sub-lattice with concept C_b as its bottom concept. If $\neg\exists\ C_i \in \mathcal{L}_b \mid$ extent of $C_i = O \wedge C_i \neq C_b$, but $\exists\ C_b' \in \mathcal{L}_b \mid C_b' > C_{New} \wedge \neg\exists\ C_3 \in \mathcal{L}_b \mid C_b' > C_3 > C_{New}$, then traversing any path where extent of a concept $\supset O$ will encounter a concept that does not have any parents whose extent $\supseteq O$. The traversal thus identifies concept C_b' for which there does not exists a $C_3 \in \mathcal{L} \mid C_b' > C_3 > C_{New}$. C_{New} must become a parent of C_b'. The INSERT function of Algorithm 3.2 examines each parent concept of a base concept C_b. If a parent concept has an extent $\supset O$, then the INSERT function recurses using the parent concept as C_b. If none of the parents have an extent $\supseteq O$, then the INSERT function creates a new concept C_{New} and links it as the parent of base concept C_b. Algorithm 3.2 initially calls the INSERT function using the bottom concept of the concept lattice as C_b.

Algorithm 3.2 fulfills requirement 2.ii.

Proof: Let if $\mathcal{L}_b$ be a sub-lattice with concept C_b as its bottom concept. If $\neg\exists\ C_i \in \mathcal{L}_b \mid$ extent of $C_i = O \wedge C_i \neq C_b$, and $\neg\exists\ C_b' \in \mathcal{L}_b \mid C_b' > C_{New}$ then $\forall$ parents of C_b extent of the parent $\neg\supseteq O$. Therefore, C_{New} must become a parent of C_b. The INSERT function of Algorithm 3.2 examines each parent concept of a base concept C_b. If none of the parent concepts has an extent $\supseteq O$, the INSERT function creates a new concept C_{New} and links it as the parent of base concept C_b. Algorithm 3.2 initially calls the INSERT function using the bottom concept of the concept lattice as C_b.

Algorithm 3.2 fulfills requirement 2.iii.

Proof: Let if $\mathcal{L}_b$ be a sub-lattice with concept C_b as its bottom concept. If $\neg\exists\ C_i \in \mathcal{L}_b \mid$ extent of $C_i = O \wedge C_i \neq C_b$, then traversing any path where extent of a concept $\supset O$ will encounter a concept that does not have any parents whose extent $\supseteq O$. Let C_b' be the encountered concept that does not have any parents

whose extent $\supseteq$ O. Let C_b' define a sub-lattice $\mathcal{L}_b$'. Since C_{New} is added as a parent of C_b', any potential parent $C_p \in \mathcal{L}_b \mid C_{New} > C_p$ must also be $\in \mathcal{L}_b$'. Furthermore, each parent C_p' of C_b' will be a parent of C_{New} in the case that the C_{New} > the C_p', or it will be a descendent of the potential parent in the case that extent of C_p' $\supset$ extent of C_p' $\cap$ O. In the later case, the potential parent can be identified by traversing the path where the extent of a concept $\supseteq$ extent of C_p' $\cap$ O. The INSERT function of Algorithm 3.2 examines each parent concept of a base concept C_b. If none of the parent concepts has an extent $\supseteq$ O, the INSERT function creates a new concept C_{New}. For each parent concept C_p of base concept C_b whose extent $\subset$ O, C_p is removed from the parents of C_b and is linked as a parent of C_{New}. For each parent concept C_p of base concept C_b whose extent $\not\subset$ O but whose extent $\cap$ O $\subset$ O, the INSERT function is recursively called using C_p as the base concept to locate the parent concept. If there exists a concept whose extent = extent of C_p $\cap$ O the INSERT function will find that concept (by proof of requirement 1). The concept returned by the recursive call is linked as a parent of C_{New}. Algorithm 3.2 initially calls the INSERT function using the bottom concept of the concept lattice as C_b.

Algorithm 3.2 correctly fulfills requirement 2.iv.

Proof: Let if $\mathcal{L}_b$ be a sub-lattice with concept C_b as its bottom concept. If $\neg\exists$ $C_i \in \mathcal{L}_b \mid$ extent of C_i = O $\wedge$ $C_i \neq C_b$, then traversing any path where extent of a concept $\supset$ O will encounter a concept that does not have any parents whose extent $\supseteq$ O. Let C_b' be the encountered concept that does not have any parents whose extent $\supseteq$ O. Since C_{New} is added as a parent of C_b' then for each concept $C_s \in \mathcal{L}_b \mid C_s$ is a parent of C_b' $\wedge$ extent of $C_s \not\subset$ O $\wedge$ extent of $C_s \cap$ O $\neq$ $\varnothing \wedge \neg\exists$ $C_3 \in \mathcal{L} \mid C_3 > C_s \wedge$ extent of $C_s \cap$ O = extent of $C_3 \cap$ O a new concept C_{New}' with empty intent and extent $C_s \cap$ O must be inserted into the lattice. C_{New}' must be a parent of both C_s and C_{New} . The INSERT function of Algorithm 3.2 examines each parent concept of a base concept C_b. If none of the parent concepts has an extent $\supseteq$ O, the INSERT function creates a new concept C_{New}. For each parent concept C_p of base concept C_b whose extent $\not\subset$ O but whose extent $\cap$ O $\subset$ O, the INSERT function is recursively called using C_p as the base concept to insert a concept with empty intent and an extent of $C_s \cap$ O. If the sub-lattice defined by C_s does not contain a concept with extent $C_s \cap$ O, then the INSERT function will create and return a new concept C_{New}' for that extent. The recursive call will link C_{New}' as a parent of C_s and return C_{New}'. C_{New}' will then be linked as a parent of C_{New}. Algorithm 3.2 initially calls the INSERT function using the bottom concept of the concept lattice as C_b.

There is currently a defect in Algorithm 3.2 in that it fails to fulfill requirement 2.v. This defect is benign in the sense that the algorithm will produce the correct set of concepts[22]. However, it may generate more links than required, many of which may violate the lattice connection property. While the invalid links can be removed by a post-processing step, the invalid links will compound and generate more invalid links. The result is a significant degradation in performance and memory usage. Sample insertions demonstrating the error are presented in Figures 3.6 through 3.9. The errors occur due to relationships between the concepts referenced in the ToProcessList.

Figure 3.3 presents an error case in which a parent is linked twice to a given child. On inserting I_4 with $\{O_1O_4\}$, the prepare-search phase will add tuples {INTERSECT, ($\{I_1\}$, $\{O_1O_2\}$), $\{O_1\}$}, {INTERSECT, ($\{I_2\}$, $\{O_3O_4\}$), $\{O_4\}$}, and {INTERSECT, ($\{I_3\}$, $\{O_1O_3\}$), $\{O_1\}$} to the ToProcessList and then create a new concept ($\{I_4\}$, $\{O_1O_4\}$). Processing {INTERSECT, ($\{I_1\}$, $\{O_1O_2\}$), $\{O_1\}$} performs a recursive call to INSERT. That call will return concept ($\varnothing$, $\{O_1\}$), which is then added to the parents of C_{New}. INSERT will be recursively called a second time to process the tuple {INTERSECT, ($\{I_2\}$, $\{O_3O_4\}$), $\{O_4\}$}. This call will create concept ($\varnothing$, $\{O_4\}$), add it as a parent of ($\{I_2\}$, $\{O_3O_4\}$), and return it. The returned concept ($\varnothing$, $\{O_4\}$) is then added as a parent to C_{New}. INSERT will be recursively called a third time to process the tuple {INTERSECT, ($\{I_3\}$, $\{O_1O_3\}$), $\{O_1\}$}. That call will return a reference to concept ($\varnothing$, $\{O_1\}$) which in turn is added to the parents of C_{New}. C_{New} has two parent references to the concept ($\varnothing$, $\{O_1\}$).

[22] See section 4.3 Algorithm Validity.

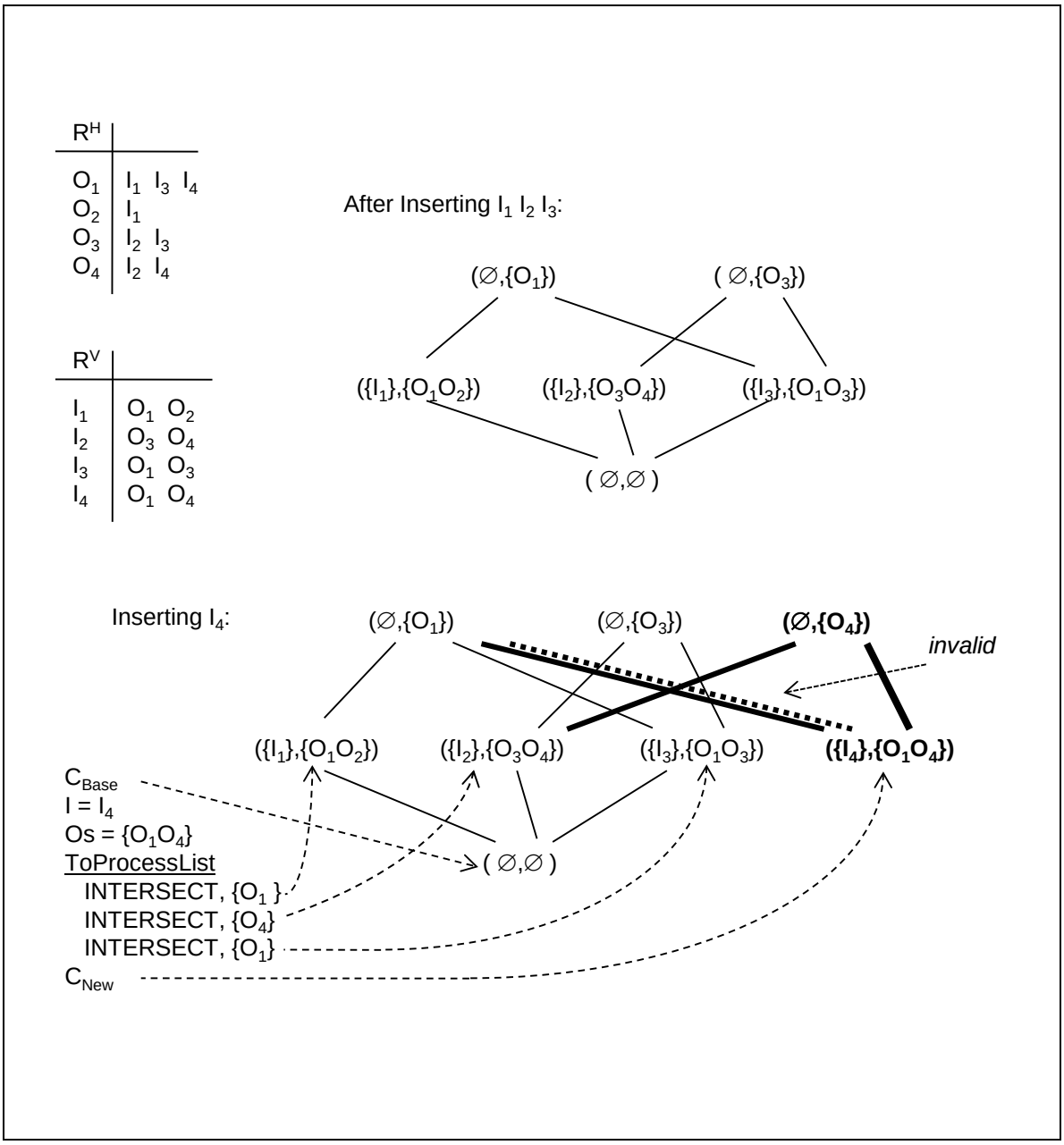


Figure 3.3: Duplicate parent-child links.

Figure 3.4 presents an error case in which the lattice connection property is violated. On inserting I_3 with $\{O_1O_2O_5\}$, the prepare-search phase will add the tuples {INTERSECT, ($\{I_1\}$, $\{O_1O_2O_3\}$), $\{O_1O_2\}$} and {INTERSECT, ($\{I_2\}$, $\{O_2O_4\}$), $\{O_1\}$} to the ToProcessList and then create new concept ($\{I_3\}$, $\{O_1O_2O_5\}$). Processing {INTERSECT, ($\{I_1\}$, $\{O_1O_2\ O_3\}$), $\{O_1O_2\}$} involves a recursive call to INSERT. The prepare-search phase of the recursive call will add a {SUPERESET, ($\varnothing$, $\{O_2\}$), $\{O_1O_2\}$} tuple in its ToProcessList and then create new concept ($\varnothing$, $\{O_1O_2\}$). Processing {SUPERESET, ($\varnothing$, $\{O_2\}$), $\{O_1O_2\}$} tuple will result in removing ($\varnothing$, $\{O_2\}$) from the parents of ($\{I_1\}$, $\{O_1O_2O_3\}$) and adding it to the parents of ($\varnothing$, $\{O_1O_2\}$). Concept ($\varnothing$, $\{O_1O_2\}$) is returned and then added as a parent of ($\{I_1\}$, $\{O_1O_2O_3\}$) by the base invocation of INSERT. INSERT will be recursively called a second time to process the tuple {INTERSECT, ($\{I_2\}$, $\{O_2O_4\}$), $\{O_1\}$}. The prepare-search phase of this call will find the concept ($\varnothing$, $\{O_2\}$) and return it. The base invocation of INSERT will then add ($\varnothing$, $\{O_2\}$) to ($\{I_1\}$, $\{O_1O_2O_3\}$) thereby violating the lattice edge property. There now exists a concept ($\varnothing$, $\{O_1O_2\}$) which is between concepts ($\{I_1\}$, $\{O_1O_2O_3\}$) and ($\varnothing$, $\{O_1\}$).

Figure 3.5 presents another error case in which the lattice edge property is violated. This case involves an INTERSECT and SUPERSET tuples in the ToProcessList. On inserting I_3 with $\{O_1O_3O_4\}$, the prepare-search phase will add the tuples {INTERSECT, ($\{I_1\}$, $\{O_1O_2\}$), $\{O_1\}$} and {SUPERSET, ($\{I_2\}$, $\{O_1O_3\}$), $\{O_1O_3\}$} to the ToProcessList and then create new concept ($\{I_3\}$, $\{O_1O_3O_4\}$). Processing {INTERSECT, ($\{I_1\}$, $\{O_1O_2\}$), $\{O_1\}$} performs a recursive call to INSERT. This recursive call will simply find and return concept ($\varnothing$, $\{O_1\}$), which is then added as a parent of ($\{I_3\}$, $\{O_1O_3O_4\}$) by the base invocation of INSERT. Processing {SUPERSET,

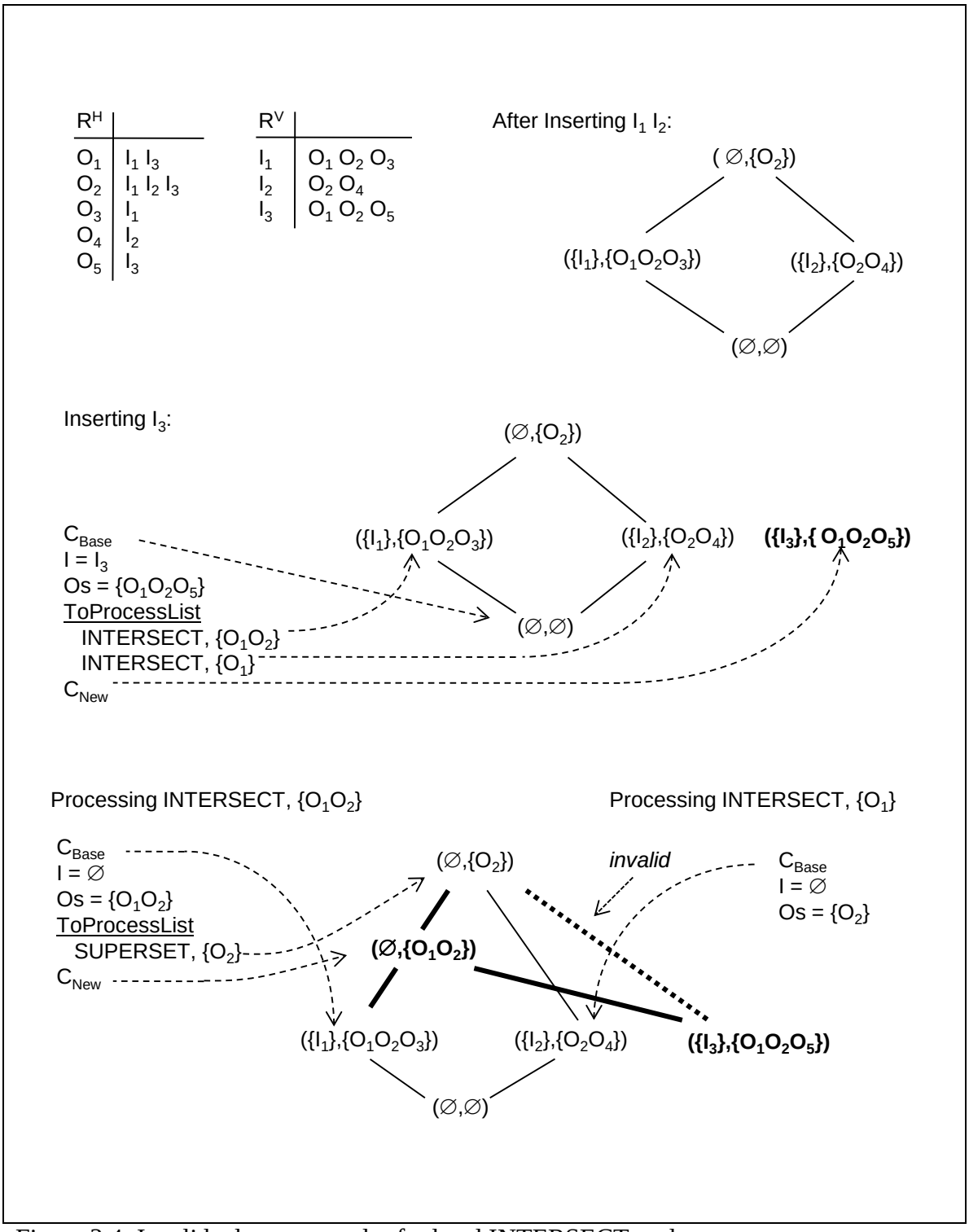


Figure 3.4: Invalid edge as a result of related INTERSECT tuples.

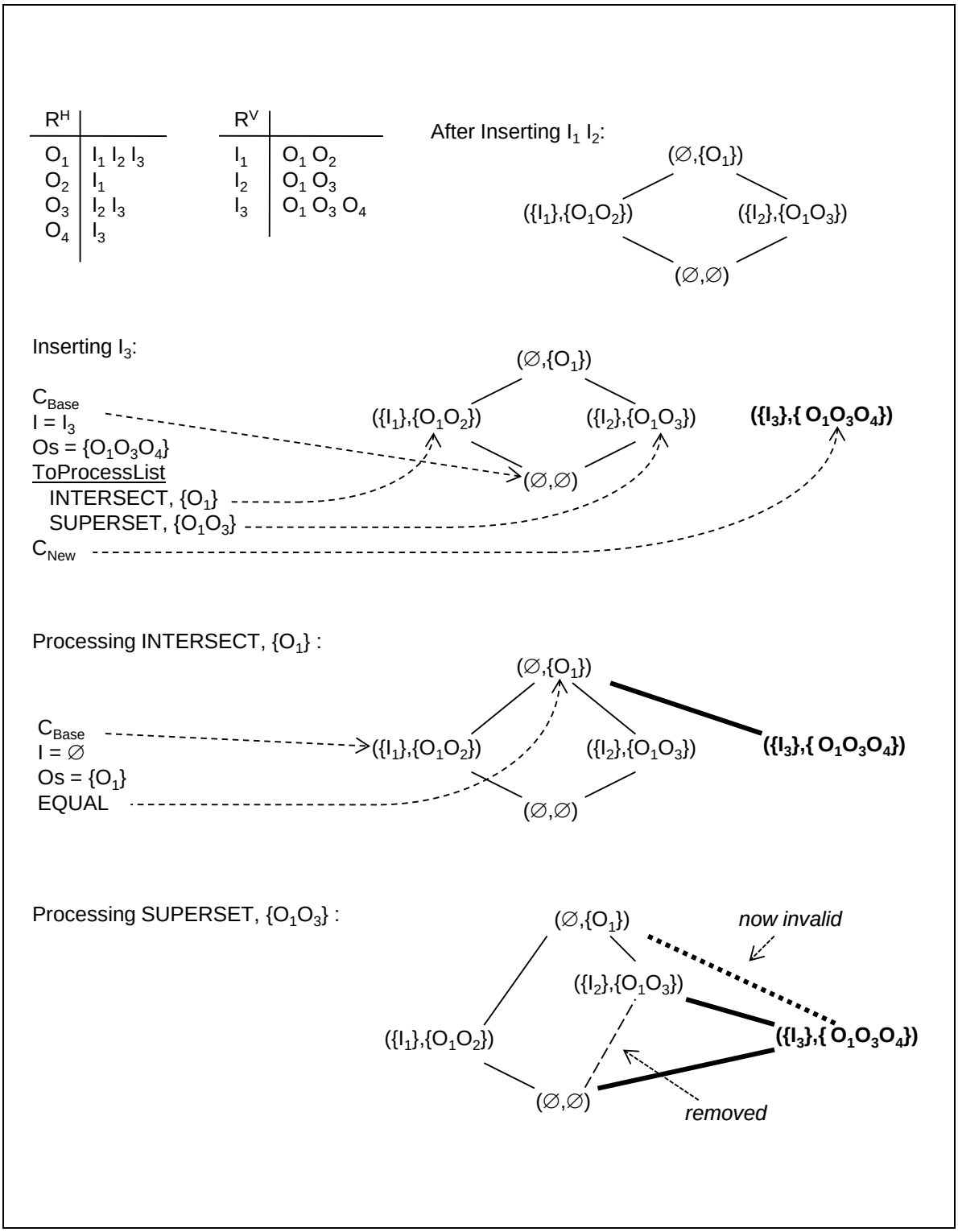


Figure 3.5: Invalid edge resulting from related INTERSECT and SUPERSET tuples.

($\{I_2\}$, $\{O_1O_3\}$), $\{O_1O_3\}$} tuple will result in removing ($\{I_2\}$, $\{O_1O_3\}$) from the parents of ($\varnothing$, $\varnothing$) and adding it to the parents of ($\{I_3\}$, $\{O_1O_3O_4\}$). The edge between ($\varnothing$, $\{O_1\}$) and ($\{I_3\}$, $\{O_1O_3O_4\}$) now violates the lattice property since concept ($\{I_2\}$, $\{O_1O_3\}$) exists between concepts ($\varnothing$, $\{O_1\}$) and ($\{I_3\}$, $\{O_1O_3O_4\}$).

3.5 Correcting the Flaw

In all error cases the extra links are a result of relationships existing between the concepts referenced in the ToProcessList. That is, there exists a non-trivial meet in the lattice between the related concepts. The recursive processing over all of the related concepts results in adding invalid parent-child links. Often the invalid link will involve the meet of the related concepts. In such cases, 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. This approach, however, is not sufficient since there exist 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. Figure 3.6 is a case where the invalid link is not the meet. In this case, 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 ($\varnothing$, $\{O_1O_2O_3\}$). Here, the invalid link is between concepts that are created during the item insertion. The invalid link will occur regardless of the order in which the tuples of the ToProcessList are processed. The processing of {INTERSECT, ($\{I_1\}$, $\{O_1O_2O_3O_4\}$), $\{O_1O_2\}$} before {INTERSECT, ($\{I_2\}$, $\{O_1O_2O_3O_4\}$), $\{O_1O_2O_5\}$}, as shown, will create the concept ($\varnothing$, $\{O_1O_2\}$) when processing {INTERSECT, ($\{I_1\}$, $\{O_1O_2O_3O_4\}$), $\{O_1O_2\}$},

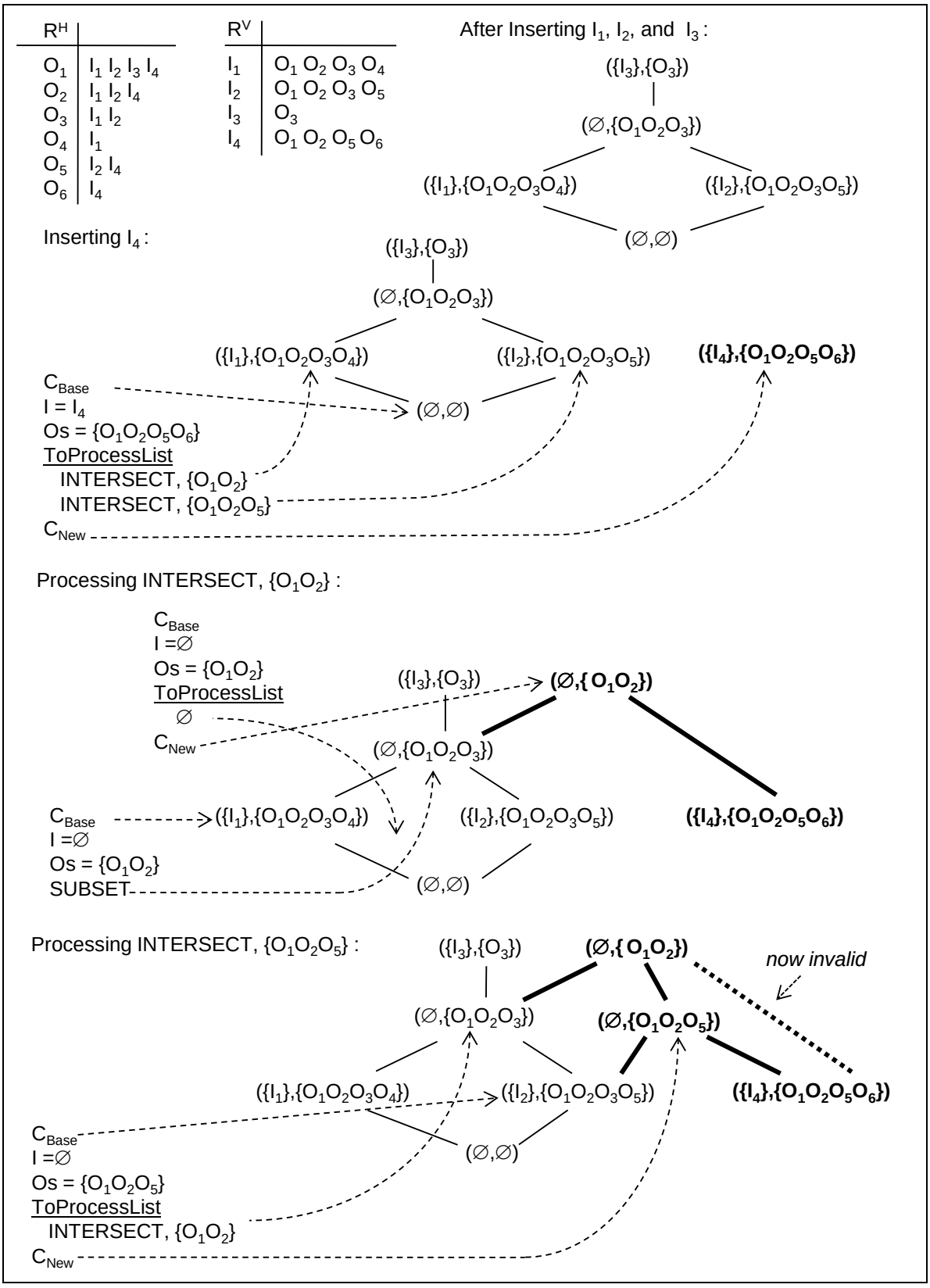


Figure 3.6: Invalid edge generated between new concepts.

then create concept (∅, {$O_1O_2O_5$}) when processing {INTERSECT, ({I_2}, {$O_1O_2O_3O_4$}), {$O_1O_2O_5$}}. On the other hand, if {INTERSECT, ({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 {INTERSECT, ({I_2}, {$O_1O_2O_3O_4$}), {$O_1O_2O_5$}}. The subsequent processing of {INTERSECT, ({I_1}, {$O_1O_2O_3O_4$}), {O_1O_2}} will simply add the violating edge. Thus, the strategy to resolve the problem is to identify and remove the tuples in the ToProcessList that will add the violating edges. These tuples will have an intersection set that is a subset of the intersection set of other tuples. Thus to correct the problem, an algorithm to purge these 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 SUPERSET tuples, since SUPERSET tuples cannot be a subset of other tuples. Furthermore, two INTERSECT 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 INTERSECT, or

ii) a superset test when the second tuple is an INTERSECT.

The later will only be performed if the first tuple is not an INTERSECT, 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 3.3 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 ∈ {NONE, SUBSET, SUPERSET, INTERSECT}, Concept a
      // reference to a concept, and O is a set of object ids

1.    for each Pi ∈ ToProcessList:
2.        for each Pj ∈ ToProcessList ∧ Pj comes after Pi:
3.            if  Pi.Type = INTERSECT ∧ Pi.O ⊂ Pj.O:
4.                Remove Pi from ToProcessList
5.            else if  Pj.Type = INTERSECT ∧ Pi.O ⊃ PjO:
6.                Remove  Pj from ToProcessList
```

Algorithm 3.3: PURGE-SUBSETS algorithm.

3.6 The Complete QuICL Oid-Full Algorithm

In addition to calling the PURGE-SUBSETS functions, there are two more minor enhancements; the first provides an additional performance improvement and the second enables a specification of a minimum support threshold in order to construct iceberg lattices. The optimization is to maintain a specific order between the parents of each concept in order to reduce the number of intersections performed. The prepare-search phase performs intersection tests between the extents of each parent concept and a given set of object ids. If during the iteration over the 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 their sizes of the extents.

The processing for iceberg lattices begins by discarding any item whose extent does not meet a minimum support threshold. In addition, the processing must prevent construction of concepts resulting from intersections with other concepts and for which the size of the concept's 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 concept. The predicate can be tested before adding an INTERSECT tuple to the ToProcessList. The result of applying these changes is the QuICL Oid-Full algorithm.

The QuICL Oid-Full algorithm is given in algorithm 3.4. The QuICL Oid-Full algorithm is Algorithm 3.2 with the stated changes. Line 21 provides the call to the

```
Let Concept be a tuple {I, O, Parents} where I a list of items, O a list of object ids,
      and Parents a list of parent concepts.

QUICL-OID-FULL(K{𝓘, 𝓞, 𝓡}, MinSupp)
 1.   C_Bottom ← new Concept (∅, ∅)
 2.   for each I_i ∈ 𝓘 ∧ |o(I_i)| ≥ MinSupp:        // o(I_i) is the set O derived from 𝓡
 3.       INSERT(C_Bottom, I_i, o(I_i))
 4.   return C_Bottom                               // the lattice

INSERT(C_Base, I_i, O)
 5.   ToProcessList ← ∅   // list of tuples {Type, Concept, O} with
 6.                       // Type ∈ {SUPERSET, INTERSECT}, Concept is a
 7.                       // reference to the intersecting concept, and O
 8.                       // a set of object ids resulting from an intersection
 9.
10.   for each C_Parent ∈ of C_Base.Parents:        // prepare-search phase
11.       if  O = C_Parent.O:
12.           Add I_i to C_Parent.I
13.           return C_Parent                       // processing complete
14.       else if O ⊂ C_Parent.O:
15.           return INSERT (C_Parent, I_i, O)      // recurse using C_Parent as new C_Base
16.       else if O ⊃ C_Parent.O:
17.           Add {SUPERSET, C_Parent, C_Parent.O} to ToProcessList
18.       else if |O ∩ C_Parent.O| ≥ MinSupp:
19.           Add {INTERSECT, C_Parent, O ∩ C_Parent.O} to ToProcessList
20.
21.   PURGE-SUBSETS(ToProcessList)
22.
23.   C_New ← New Concept({I_i}, O)                 // create the new concept
24.
25.   for each T_i ∈ ToProcessList:                 // link phase to link in C_New
26.       if  T_i.Type = SUPERSET:
27.           Remove T_i.Concept from C_Base.Parents
28.           Add T_i.Concept  to C_New.Parents
29.       else if  T_i.Type = INTERSECT:
30.           C_Parent ← INSERT (T_i.Concept, ∅, T_i.O)
31.           Add C_Parent to C_New.Parents
32.
33.   Sort C_New.Parents in order of decreasing |O|
34.
35.   Add C_New to C_Base.Parents in order of decreasing |O|
36.
37.   return C_New
```

Algorithm 3.4: The QuICL Oid-Full algorithm.

PURGE-SUBSETS function. Lines 33 and 35 specify an order for parents of a concept. Line 2 is modified to discard items that do not meet the minimum support threshold. Line 18 tests that the size of the intersection set meets the minimum support threshold. A complete implementation, written in Java, is provided in Appendix B.

3.7 An Implementation Enhancement

Testing and analysis of the QuICL Oid-Full algorithm 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. Where the same parent is encountered, each invocation has a different base concept that shares the parent. Even though each invocation may be passed a different set of object ids, intersecting the object ids with a given parent's extent will produce the same intersection set during insertion of a given item. This is the case since the intersection set ultimately is the intersection of the parent's extent and the extent of the item. Thus, an implementation enhancement is to cache[23] each intersection set with its parent concept for the duration of an item insertion. Between item insertions all cached intersection sets are discarded. This enhancement involves augmenting the tuples that represent concepts to include an intersection set, Intersect. Intersect is set the first time a parent concept is encountered during an item insertion and cleared between insertions.

While the intersection set is the same between invocations, the outcome of comparison (i.e., $=$, $\subset$, $\supset$, and $\cap$) on which the QuICL algorithm is dependent can be different. The outcome of comparison can be readily determined by performing tests on the cardinalities of the intersection set, the parent's extent, and the object id set passed to

[23] The cache is a simple reference to an intersection set from within each tuple of a concept. The intersection sets are discarded between item insertions.

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

Table 3.1: Determination of intersection outcome for Oid-Full enhancement. C_{parent}.Intersect is the cached intersection set, O is the object id set passed to INSERT, and C_{parent}.O is the extent of the parent concept.

the INSERT function. Table 3.1 provides identification of 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[24] 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 Intersect of the parent concept is assigned the object id set passed to INSERT. If the outcome of comparison is superset, then Intersect is assigned $C_{Parent}.O$. If $|O \cap C_{Parent}.O|$ < minimum support, Intersect is assigned an empty set. A new intersection set is created and cached in the parent concept only when the outcome is intersection and the intersection set meets the minimum support threshold. However, using a reference to this same set in the INTERSECT tuples of the ToProcess list will result in no additional memory consumption. This set ultimately becomes the extent of a new concept that is added to the lattice. Lastly, on creating a new concept, the Intersect is assigned a reference to the concept's extent.

In a preliminary test, the final QuICL Oid-Full algorithm successfully constructed the complete lattice for the Mushroom data set in 3.54 seconds. This represents a gain in excess of two orders of magnitude over the modified GMA algorithm (Algorithm 3.1).

[24] 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. Such sets could amount to substaintial memory consumption during the insertion process.

3.8 Asymptotic Complexity of the QuICL Oid-Full Algorithm

Determining the asymptotic complexity of lattice construction algorithms has been noted to be a "delicate task" (Valtchev et al., 2002). The review of literature of lattice construction algorithms indicates that the cardinality of the lattice (i.e., number of concepts) is a factor. However, determining the cardinality of a lattice from an input data set is of itself a #P complete[25] problem (Kuznetsov, 2001). Given this, the expression of runtime complexity for the QuICL Oid-Full algorithm will include $|\mathcal{L}|$ as a factor. Therefore, in order to postulate a runtime complexity, the cost to add a concept into the lattice must be assessed.

A concept is created at line 23 of the INSERT function of the QuICL Oid-Full algorithm (Algorithm 3.4) and then linked into the lattice. Possible terms affecting the runtime complexity of adding a concept are:

i) time to navigate to the concept above which the new concept will be created (lines 10 through 19 ultimately recursing at line 15),

ii) time to execute the prepare-search phase to identify and add entries to the ToProcessList (lines 10 through 19 not recursing at line 15 or returning at line 13),

iii) time to purge entries in the ToProcessList that are subsets of other entries (line 21),

iv) time to create the new concept (line 23),

v) time to process SUPERSET entries in the ToProcessList (lines 25 through 31 for T_i.Type = SUPERSET),

vi) time to process INTERSECT entries in the ToProcessList (lines 25 through 31 for T_i.Type = INTERSECT),

[25] #P, pronounced "sharp P" or "number P", is class of problems in computation theory which is the subset of NP related to counting (i.e., determining "how many"). #P problems are considered to be more difficult NP problems. For each #P problem, an easier "does there exists" problem may be NP. Term was first introduced by Valiant (1979).

vii) time to sort the parents of the new concept (line 33), and

viii) time to link the new concept to its base concept (line 35).

The time to navigate to the concept above which a new concept will be created involves recursion at line 15. The number of times recursion is performed for a given object id set could be, in the worst case, the height of the lattice minus one. The worst case for the height of the lattice is the cardinality of the item set. Recursion is performed using one of the parent concepts as the new base concept. To identify the parent concept on which to recurse, the INSERT function iterates through the parent concepts and performs an intersection using object id set passed in the call and the extent of each parent. The worst case for the number of parent concepts is also the cardinality of item set. The cost of intersection will be $O(k)$ where $k = |\mathcal{O}|$, provided the object ids are maintained in sorted order. Therefore, the worst case cost to navigate into the lattice to the point where a new concept will be inserted is $O(m^2 k)$, where $m = |\mathcal{I}|$ and $k = |\mathcal{O}|$. However, this analysis does not account for the fact that the vast majority of concepts are created as a result of recursive calls. This is often the case, since in a typical lattice the number of concepts far exceeds the number of items[26]. Furthermore, these recursive calls are invoked at a higher level in the lattice and the concept passed as the base is often the point above which a new concept will be created. Therefore, the time to navigate is near zero. Thus, the time to navigate will not be a dominant term in runtime complexity.

The time to execute the prepare-search phase involves iterating over the parent concepts of the base concept and performing an intersection for each. Each intersection is between the object ids passed in the call and the extent of the parent. The worst case

[26] There are at most $|\mathcal{I}|$ concepts created by non-recursive calls, since $|\mathcal{I}|$ is the number of times the INSERT function is called as a result of processing the input data set.

for the number of parent concepts is the cardinality of an item set. Furthermore, the worst case cost of intersection will be O(k) where $k = |\mathcal{O}|$, provided the object ids are in sorted order. Therefore, the worst case cost execute the prepare-search phase is O(mk), where $m = |\mathcal{I}|$, and $k = |\mathcal{O}|$. However, this worst case cost would only be realized with data sets of extreme density[27] and are not representative of real world data sets. For real world data sets the average number of parent concepts of the base concept will be far less than the cardinality of the item set. For example, Table 3.2 presents the average degree (i.e., average number of parents in the upper cover of all concepts) of the lattices generated by four benchmark data sets often cited in literature. Beyond this, using the cardinality of the object id set as the factor representing the cost of intersection will be excessive. For some data sets such as the T10I4D100k, the largest set of object ids will represent only a fraction of the objects. Furthermore, the cardinality of the object ids sets (i.e. extents) in a vast majority of concepts will be much smaller. However, the cardinality of these sets must at least meet the minimum support. Thus, the intersection cost will be some factor that is greater than the minimum support (best case), less than the cardinality of the largest object id set (worst case), and probably skewed towards a lower value depending on the density of the data set (expected case). This factor will be

Data Set	$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert\mathcal{L}\rvert$	Avg Deg	Max Deg
Mushroom	8,124	119	238,709	5.71	33
Pumsb at 75%$_{supp}$	49,046	7,116	101,047	7.02	21
T10I4D100k	100,000	999	2,347,374	4.29	846
T25I10D10k	9,219	1,000	2,557,928	4.30	996

Table 3.2: Sample data set and lattice characteristics[28].

[27] 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.

[28] See Section 4.2 for further description of these data sets.

some type of mean function on the cardinality of the extents of frequent items. This mean will take into account the density of the data set, since the density will affect the probability of common values between object id sets and therefore generate a greater number intersection sets whose cardinality is large. The explicit type of mean (e.g., arithmetic, quadratic, weighted, etc.) is indeterminate due to the dependency on density. The term *density weighted mean* will be used to reference this mean. Given this, an expected cost to execute the prepare-search phase is $O(\boldsymbol{d}\,\boldsymbol{i})$, where $\boldsymbol{d} = \deg_{avg}(\mathcal{L})$, and $\boldsymbol{i}$ is a density weighted mean on the cardinality of frequent item extents.

The time to purge entries in the ToProcessList that are subsets of other entries is the time to execute the PURGE-SUBSETS algorithm. This algorithm compares each entry with every other entry. The cost to remove an entry can be done in constant time and therefore is not a factor. Thus, the runtime complexity is $O(\boldsymbol{n}^2\,\boldsymbol{c})$ where $\boldsymbol{n}$ is the number of entries and $\boldsymbol{c}$ is the cost of comparison. The number of entries in the ToProcessList will at most be number of parents of a concept, which in the worst case is the cardinality of the set of items. The worst case cost of comparison will be $O(\boldsymbol{k})$ where $\boldsymbol{k} = |\mathcal{O}|$, provided the object ids are maintained in sorted order. Therefore, the worst case time to purge entries in the ToProcessList is $O(\boldsymbol{m}^2\,\boldsymbol{k})$, where $\boldsymbol{m} = |\mathcal{I}|$ and $\boldsymbol{k} = |\mathcal{O}|$. The argument to use average degree of the lattice in place of $|\mathcal{I}|$ cannot be clearly made due to the quadratic expression. While the average degree is indeed a fraction of $|\mathcal{I}|$, there can exist concepts whose number of parents approaches $|\mathcal{I}|$ as evident by the maximum degree given in Table 3.2. However, since the average degree on most data sets is small (i.e., < 10), there must exist a large number for concepts less than the average to

compensate for any outliers. Furthermore, since the degree for a majority of the concepts must be at least two[29], an even greater number of small concepts must be present. Therefore, average degree of the lattice will be a better expression than $|\mathcal{I}|$. The cost of comparison will be less than the cost of intersection. In the cases where neither set is a subset of the other, this outcome is often determined within a few iterations into each set. An outcome can be determined as soon as an object id not present in the other set is found in each. Given that a vast majority of the entries in the ToProcessList are not subsets of each other, the cost of comparison will probably be near a constant which is some small fraction of $|\mathcal{O}|$. However, the density of data set may have an effect on this constant, since a higher density increase the probability of common values between sets. Therefore, the cost purge entries is $O(\boldsymbol{d}^2\, \boldsymbol{c})$, where $\boldsymbol{d} = \deg_{avg}(\mathcal{L})$ and $\boldsymbol{c}$ is a small fraction of $|\mathcal{O}|$ depending density.

Of the remaining terms, only the time to process INTERSECT entries has the potential to be a dominant term. Time to create the new concept is $O(\mathbf{1})$, time to process SUPERSET entries is $O(\boldsymbol{d})$, time to sort parents is $O(\boldsymbol{d} \log \boldsymbol{d})$, and time to link the new concept to its base is $O(\log \boldsymbol{d})$, where $\boldsymbol{d} = \deg_{avg}(\mathcal{L})$. Processing the INTERSECT entries involves performing a recursive call to the INSERT function. Many of these calls will result in creating a new concept. The cost to create a new concept, during the processing of INTERSECT entries, does not need to be consider at this point. This cost will be accounted for in the overall runtime complexity, since the overall runtime complexity has factor $|\mathcal{L}|$ which includes all concepts. This leaves calls that search for concepts already present in the lattice. The cost of performing such call will be very similar to cost of

[29] There can be at most $|\mathcal{I}|$ interior concepts of degree one. Since $|\mathcal{L}|$ greatly exceeds $|\mathcal{I}|$, most interior concepts have a degree greater than one.

prepare-search, since the same statements are performed (lines 10 through 19 of Algorithm 3.4). However, these calls may recurse a number of times through line 15 before ultimately reaching line 13. The number of times recursion is performed will be in the worse case the height of the lattice. However, the insertion of a new concept is often at higher levels in the lattice. Furthermore, the parent is often found within a few levels of recursion and in many cases just one[30]. Therefore, the expected number of times recursion is performed is a sub-linear function (e.g., log, sqrt) on the height of the lattice. Thus, the runtime complexity of performing calling INSERT to find a concept already present in the lattice will be $O(\boldsymbol{d}\, \boldsymbol{i}\, \boldsymbol{h})$, where $\boldsymbol{h}$ is a sub-linear function on the height of the lattice, and $\boldsymbol{d}$ and $\boldsymbol{i}$ as previously stated. The number of such calls will be a fraction of the number of parents of a concept depending on density, since density has an effect on the probability the extent already exists in the lattice. Therefore, the time to process INTERSECT entries that find concepts already present in the lattice is $O(\boldsymbol{d}\, \boldsymbol{d}'\, \boldsymbol{i}\, \boldsymbol{h})$, where $\boldsymbol{d} = \deg_{avg}(\boldsymbol{\mathcal{L}})$, $\boldsymbol{d}'$ is a fraction of $\boldsymbol{d}$ depending on density, $\boldsymbol{i}$ is a density weighted mean on the cardinality of frequent item extents, and $\boldsymbol{h}$ is a sub-linear function on the height of the lattice.

Of all the possible terms affecting the runtime complexity of adding a concept into the lattice, only the time to execute the prepare-search phase, time to purge subsets, and time to process INTERSECT entries are feasible dominant terms. The complexity for these terms is $O(\boldsymbol{d}\, \boldsymbol{i})$, $O(\boldsymbol{d}^2\, \boldsymbol{c})$, and $O(\boldsymbol{d}\, \boldsymbol{d}'\, \boldsymbol{i}\boldsymbol{h})$ respectively. Given this, it is postulated that the runtime complexity for the QuICL Oid-Full algorithm will be either $O(\boldsymbol{\ell}\, \boldsymbol{d}\, \boldsymbol{i})$, $O(\boldsymbol{\ell}\, \boldsymbol{d}^2\, \boldsymbol{c})$, or $O(\boldsymbol{\ell}\, \boldsymbol{d}\, \boldsymbol{d}'\, \boldsymbol{i}\boldsymbol{h})$, where $\boldsymbol{\ell} = |\boldsymbol{\mathcal{L}}|$ and the rest as previously stated. While $O(\boldsymbol{\ell}\, \boldsymbol{d}^2\, \boldsymbol{c})$

[30] This behavior was observered by instrumentation added during the benchmarks of Chapter 4. The number times the calls recurse appear to grow at a slower rate than the height of the lattice.

or O($\ell\, d\, d'\, i\, h$) on the surface appear to be greater than O($\ell\, d\, i$), the probability of a real effect on the performance by these terms is doubtful, since both c and d' will often be very small fractions. Only O($\ell\, d\, i$) is assured. Thus, the expected asymptotic complexity of the QuICL Oid-Full algorithm will at least be O($\ell\, d\, i$), but could approach O($\ell\, d^2\, c$) or O($\ell\, d\, d'\, i\, h$), where $\ell = |\mathcal{L}|$, $d = \deg_{avg}(\mathcal{L})$, i is density weighted mean on the cardinality of frequent item extents, c is a small fraction of $|\mathcal{O}|$ depending density, d' is a fraction of d depending on density, and h is a sub-linear function on the height of the lattice.

Determination of the space complexity for the QuICL Oid-Full algorithm will also include $|\mathcal{L}|$ as a factor. Therefore, in order to postulate a space complexity, the space consumed by each a concept must first be assessed. Each concept is a tuple {I, O, Parents} where I is a list of items, O is a list of object ids, and Parents is a list of parent concepts. Given this, a space complexity for the QuICL Oid-Full algorithm of O($\ell\, m^2\, k$) could be considered, where $\ell = |\mathcal{L}|$, $m = |\mathcal{I}|$, and $k = |\mathcal{O}|$. However, the QuICL Oid-Full algorithm only stores a given item in only one place within the lattice. The sum of the |I| for all concepts will be at most $|\mathcal{I}|$. Thus, one m can be removed as a factor. The other m can be replaced by d, being the average degree of the lattice, for the same arguments stated during the analysis of runtime complexity. Likewise, k can be replaced with i, being a density weighted mean on the cardinality of frequent item extents. Therefore, it is postulated that the memory complexity of the QuICL Oid-Full algorithm is O($\ell\, d\, i$) where $\ell = |\mathcal{L}|$, $d = \deg_{avg}(\mathcal{L})$, and i is a density weighted mean on the cardinality of frequent item extents.

3.9 Discussion for an Alternate QuICL

The execution times from preliminary tests of the QuICL Oid-Full algorithm are very promising. Performance against the Mushroom data set and the T25I10D10k[31] data set appear to be far better than any known algorithms. However, the performance gains do not hold against some other data sets, such as Chess[32]. An issue for the QuICL Oid-Full algorithm is storage of the complete list of object ids in each concept. The same object ids can be repeated in multiple concepts. This is evident in Figure 3.1. Since the number of concepts can grow exponentially with respect to the size of the data set, the size and storage of concept lattice has potential to exceed the capacity of the main memory. To address this concern, an alternate compressed data structure is sought. Such alternate notation can be obtained by exploiting 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. An object O_i need only be recorded in a concept at its minimal position (i.e., highest position in the inverted lattice). The complete set of O for a concept can be computed by traversing all ancestors.

Figure 3.7 depicts the same lattice construction progress of Figure 3.1 using the compressed lattice data structure. In addition to the same observations concerning the progression given in Figure 3.1, an additional observation is noteworthy:

- As new concepts are inserted into the lattice, the object ids of the children of the new concept will percolate up into the new concept for any object ids that are common between the children.

[31] T25I10D10k is another test data set often used in studies of association rule mining. See Section 4.2.
[32] Chess is data set often used in studies of association rule mining. See Section 4.2.

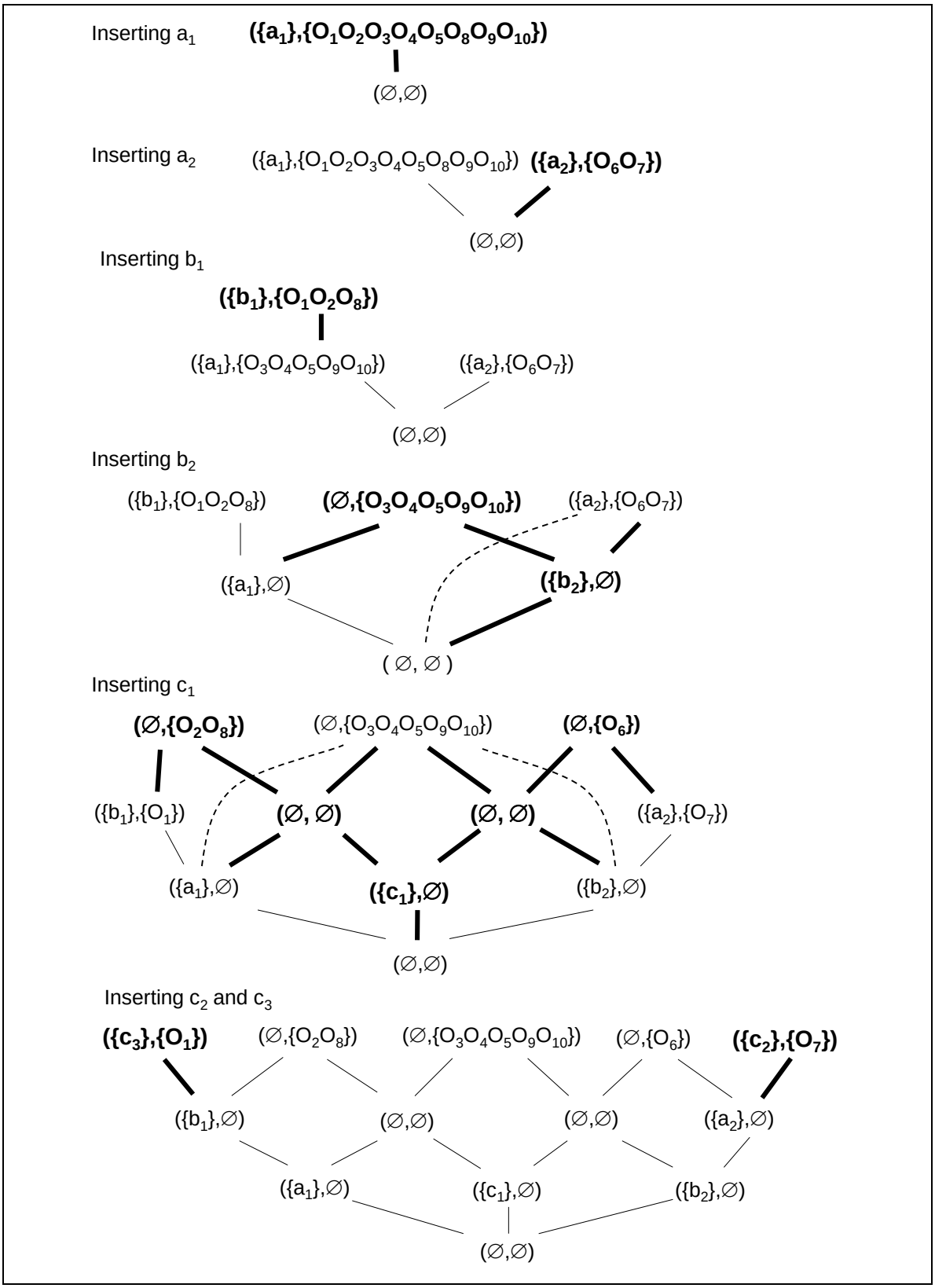


Figure 3.7: Progression of incremental insertion into a compressed lattice. Bold text and lines identify added new elements. Dashed lines are removed elements.

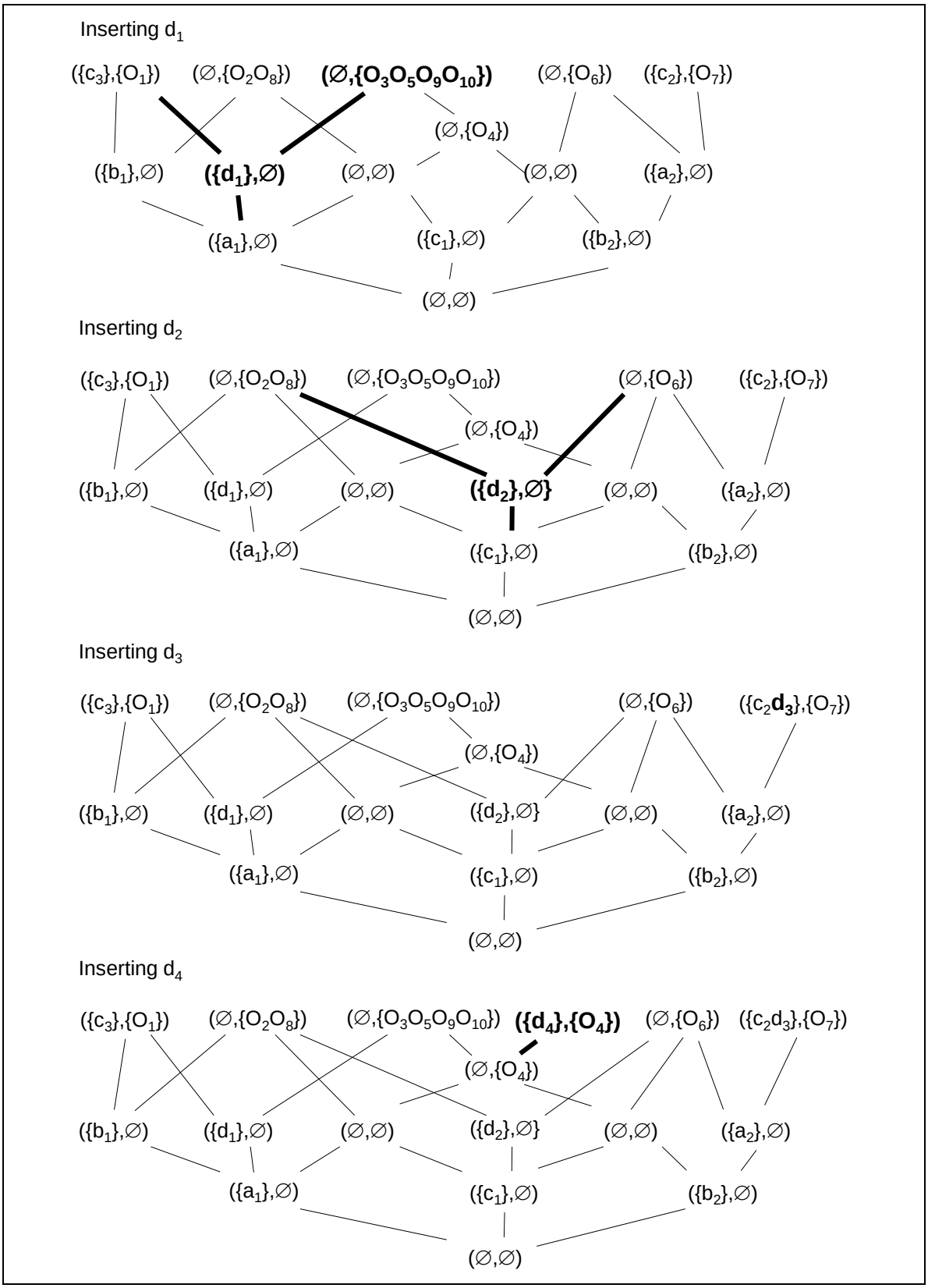


Figure 3.7 continued: Progression of incremental insertion into a compressed lattice. Bold text and lines identify added new elements.

3.10 An Incremental Insertion Algorithm Using a Compressed Lattice

Algorithm 3.5 presents an incremental insertion algorithm that uses the compressed lattice data structure. This is the QuICL algorithm of Algorithm 3.4 with a few modifications:

i) The tuples now include field Support, since support can no longer be determined by |O|. Changes to set and use Support are made accordingly (lines 20, 24, 38, and 40).

ii) The predicates to support construction of an iceberg lattice have been removed. Support for iceberg lattices using the compressed data structure will be discussed later in this chapter.

iii) A call is made to a function GET-O to obtain the complete list of object ids for a concept (line 7). The GET-O algorithm, given in Algorithm 3.6, traverses the ancestors in the lattice to obtain a concept's object ids. The returned set is used instead of the O of a concept when performing subset, superset, equal, and intersect operations (lines 8, 11, 13, and 15).

iv) Object ids in O that are not at, or will no longer be at, the minimal position[33] are removed from O (lines 21 and 22) prior to constructing the new concept (line 24). This performs part of the percolation of ids. Removing the percolated ids from the base concept completes the process (line 25).

GET-O-EXTEND within GET-O traverse all ancestors of a concept recursively. The VisitedSet (line 2) is used to ensure that an ancestor concept is only processed once. The set O is sorted (line 4) to enable fast subset, superset, equal, and intersect operations on the returned set.

In a preliminary test, Algorithm 3.5 took 1,560 seconds to execute against the Mushroom data set. While this execution time is two orders of magnitude greater than the QuICL Oid-Full, it is interesting to note that it matches the GMA algorithm. Profile analysis revealed that near 99% of the execution time is consumed by the GET-O function. Thus, alternate strategies to perform the comparison tests are needed.

[33] Minimal position in the highest concept in an inverted lattice that is to hold a given object id.

```
Let Concept be a tuple {I, O, Support, Parents} where I a list of items, O a list of
      object ids, Support is the support of the concept, and Parents a list of parent
      concepts.

INSERT(C_Base, I_i, O) :
 1.   ToProcessList ← ∅   // list of tuples {Type, Concept, O} with
 2.                       // Type ∈ {SUPERSET, INTERSECT}, Concept a
 3.                       // reference to the intersecting concept, and O
 4.                       // a set of object ids resulting from an intersection
 5.
 6.   for each C_Parent ∈ of C_Base.Parents:          // prepare-search phase
 7.       O_Parent ← GET-O(C_Parent)
 8.       if  O = O_Parent:
 9.           Add I_i to C_Parent.I
10.           return C_Parent
11.       else if  O ⊂ O_Parent:
12.           return INSERT (C_Parent, I_i, O)
13.       else if  O ⊃ ConceptO:
14.           Add {SUPERSET, C_Parent, O_Parent} to ToProcessList
15.       else if O ∩ ConceptO ≠ ∅:
16.           Add {INTERSECT, C_Parent, O ∩ O_Parent} to ToProcessList
17.
18.   PURGE-SUBSETS(ToProcessList)
19.
20.   Support ← |O|
21.   for each T_i ∈ ToProcessList:                  // remove ids not at min position
22.       O ← O – T_i.O
23.
24.   C_New ← New Concept({I_i}, O, Support)         // create the new concept
25.   C_Base.O ←  C_Base.O – O                       // percolate object ids
26.
27.   for each T_i ∈ ToProcessList:                  // link phase - process intersections
28.       if  T_i.Type = SUPERSET:
29.           Remove T_i.Concept from C_Base.Parents
30.           Add T_i.Concept  to C_New.Parents
31.       else if  T_i.Type = INTERSECT:
32.           C_Parent ← INSERT (T_i.Concept, ∅, T_i.O)
33.           Add C_Parent to C_New.Parents
34.
38.   Sort C_New.Parents in order of decreasing Support
39.
40.   Add C_New to C_Base.Parents  in order of decreasing Support
41.
42.   return C_New
```

Algorithm 3.5: Incremental item insertion algorithm for a compressed lattice.

```
GET-O(Concept)
 1.  O ←∅
 2.  VistedSet ←∅
 3.  GET-O-EXTEND(Concept, O, VisitedSet)
 4.  Sort O
 5.  Returns O

GET-O-EXTEND(Concept, O, VisitedSet)
 6.  if Concept ∉ VistedSet:
 7.      Add Concept to VistedSet
 8.      O ←O ∪ Concept.O
 9.      for each C_Parent ∈ Concept.Parents:
10.          GET-O-EXTEND(C_Parent, O, VisitedSet)
```

Algorithm 3.6: Supporting algorithms to extract an object id set.

3.11 A Strategy to Intersect a Concept Lattice

Instead of performing subset, superset, equal, and intersect operations against the object ids obtained for each individual concept, an alternate strategy is to intersect the extent of an item with the complete lattice. This strategy exploits the compress lattice data structure. That is, all object ids are physically stored at only their minimal positions. As a result, only one concept in the lattice contains a given object id. A vector indexed by object id can be used to locate the concept. Intersection with the lattice can be performed by iterating over the item's extent, lookup each concept using the object id as an index, and then add the object to the set of intersect object ids stored in the concept. The intersect object ids are a temporal set that is cleared between item insertions. This represents the set of object ids at its minimal position that intersects the item's extent. It does not represent the extent of a given concept that intersects the item's extent. The full intersection set for a concept can be obtained by traversing all ancestors of a concept and accumulating all object ids in the object id intersection sets.

Figure 3.8 illustrates an example of lattice intersection. For the new item I_3, the concept for each object in the extent $\{O_3O_4O_6\}$ is identified using each object id as an index into the O2C vector, and the object id is then added to the referenced concept's intersection set. In Figure 3.8 the object ids highlighted in bold represent the object ids in the intersection set. To obtain the full intersection set for a given concept, the concept and all of its ancestors are traversed accumulating the object ids of the intersection sets. For example, the full intersection set for concept ($\{I_5\}$, $\varnothing$) is $\{O_3O_4O_6\}$ and the full intersection set for concept ($\{I_6\}$, $\varnothing$) is $\{O_3\}$.

Algorithm 3.7 presents the incremental item insertion algorithm for a compressed lattice that uses lattice intersection. The related supporting functions used in Algorithm 3.7 are given in Algorithm 3.8. The BUILD-LATTICE function now calls a function INTERSECT-LATTICE (line 3) to intersect an item's extent with the concept lattice. This will set the intersection sets of the concepts referenced by the O2C lookup vector. INTERSECT-LATTICE is executed once for each item prior to calling INSERT to add the item into the lattice. The call to function GET-INTERSECT (line 12) is used to obtain the full set of object ids of a concept that intersect the items extent. However, this set by itself is not sufficient to determine if a concept's object ids is a subset, superset, or equal to the item's extent. The function HAS-SUPERSET, called on line 13, returns a boolean indicating if the concept has at least one object id that does not intersect with the item's extent. From the Intersect returned by INTERSECT and HasSuper returned by HAS-SUPERSET the relationship between an item's extent and a concept can be readily determined as indicated in Table 3.3.

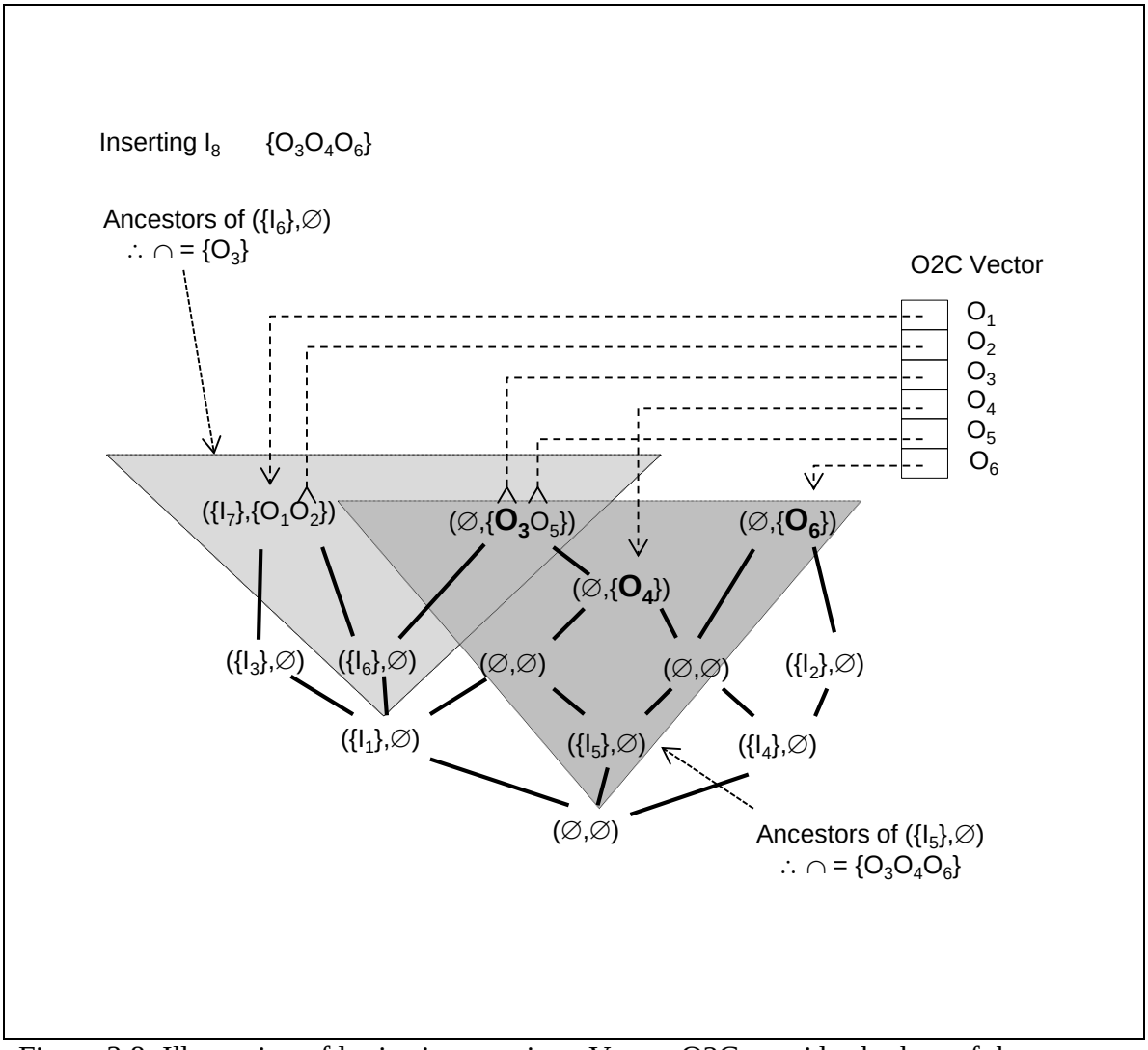


Figure 3.8: Illustration of lattice intersection. Vector O2C provides lookup of the concept holding a given object id. The shaded triangles denote the ancestors of a given concept. Bolded text denotes the intersecting object ids. For a given concept C, the full set of intersecting object ids are the intersection object ids in that concept and all ancestor concepts.

Test on Intersect	HasSuper	Outcome
|Intersect| = 0		No relationship
|Intersect| = |O|	FALSE	Item's Extent = Concept's Extent
|Intersect| = |O|	TRUE	Item's Extent $\subset$ Concept's Extent
|Intersect| < |O|	FALSE	Item's Extent $\supset$ Concept's Extent
|Intersect| < |O|	TRUE	Item's Extent $\cap$ Concept's Extent

Table 3.3: Determination of intersection outcome.

The prepare-search phase in the INSERT function uses above tests in place of previous =, $\subset$, $\supset$, and $\cap$ set operations (lines 14, 16, 19, 21, and 23). In addition, function ADJUST is called following creation of a new concept (line 30). The ADJUST function is needed to perform adjustments between C_{New} and C_{Base} concepts in order to maintain integrity of the compressed lattice data structure. The rest of the INSERT function is the same as before.

Algorithm 3.8 provides the algorithms for functions supporting Algorithm 3.7. For these functions, the tuples defining concepts in the lattice are augmented with a set of temporal fields to hold the results of intersecting an item with the lattice. The fields include Intersect, FullIntersect, and HasSuper. Intersect are the object ids at their minimal position that intersect an item's extent. The FullIntersect is the full list of object ids of a concept that intersect an item's extent. HasSuper is a boolean indicating that there exists at least one object id of the concept that does not intersect with the item's extent. Intersect is set by calling the INTERSECT-LATTICE function. FullIntersect and HasSuper are derived as needed by the GET-INTERSECT and HAS-SUPERSET functions, respectively. All of the temporal fields are retained during insertion of a given item and discarded between item insertions. These fields can be implemented using hash tables or as additional fields in the tuples. In either case, clearing the fields involves a marginal amount of bookkeeping and performance overhead.

```
Let Concept be a tuple {I, O, Support, Parents} where I a list of items, O a list of
      object ids, Support is the support of the concept, and Parents a list of parent
      concepts.

Let O2C be a vector whose index O_i identifies the concept C_i | O_i ∈ C.O

BUILD-LATTICE(K{ℐ, 𝒪, ℛ})
 1.   C_Bottom ← new Concept (∅, ∅)
 2.   for each I_i ∈ ℐ:                          // o(I_i) is the set O derived from ℛ
 3.       INTERSECT-LATTICE(C_Bottom, o(I_i))
 4.       INSERT(C_Bottom, I_i, o(I_i))
 5.   return C_Bottom                            // the lattice

INSERT(C_Base, I_i, O)
 6.   ToProcessList ← ∅   // list of tuples {Type, Concept, O} with
 7.                       // Type ∈ {SUPERSET, INTERSECT}, Concept a
 8.                       // reference to the intersecting concept, and O
 9.                       // a set of object ids resulting from an intersection
10.
11.   for each C_Parent ∈ of C_Base.Parents:     // prepare-search phase
12.       Intersect ← GET-INTERSECT(C_Parent)
13.       HasSuper ← HAS-SUPERSET(C_Parent)
14.       if  Intersect = ∅:                          // no relationship
15.           continue for each with next C_Parent
16.       else if |Intersect| = |O| ∧ HasSuper = FALSE:     // equal case
17.           Add I_i to C_Parent.I
18.           return C_Parent
19.       else if |Intersect| = |O| ∧ HasSuper = TRUE:      // subset case
20.           return INSERT (C_Parent, I_i, O)
21.       else if |Intersect| < |O| ∧ HasSuper = FALSE:     // superset case
22.           Add {SUPERSET, C_Parent, Intersect} to ToProcessList
23.       else if |Intersect| < |O| ∧ HasSuper = TRUE:      // intersect case
24.           Add {INTERSECT, C_Parent, Intersect} to ToProcessList
25.
26.   PURGE-SUBSETS(ToProcessList)
27.
28.   C_New ← New Concept({I_i}, C_Base.Intersect, |O|)       // create the new concept
29.
30.   ADJUST(C_Base, C_New)
            . . .
```

Algorithm 3.7: Incremental item insertion algorithm using lattice intersection.

Let tuples of Concepts be augmented with the temporal fields {Intersect, FullIntersect, HasSuper} where Intersect a list of the concept's O that intersect with an item's extent, FullIntersect a full set of object ids of a concept that intersect with an item's extent, and HasSuper a boolean indicating at least one object id of a concept does not intersect with an item's extent.

```
INTERSECT-LATTICE(C_Bottom, O)
 1.  All temporal fields of all concepts within the lattice ←∅
 2.  for each O_i ∈ O:                    // perform the intersection
 3.      C_Min ← O2C[O_i]                 // C_Min is the concept with O_i at minimal position
 4.      if C_Min ≠ ∅:
 5.          Add O_i to C_Min.Intersect
 6.      else:
 7.          Add O_i to C_Bottom.Intersect

GET-INTERSECT(Concept)
 8.  if Concept.FullIntersect = ∅:
 9.      VisitedSet ←∅
10.      GET-INTERSECT-EXTEND(Concept, Concept.FullIntersect, VisitedSet)
11.      Sort Concept.FullIntersect
12.  return Concept.FullIntersect

GET-INTERSECT-EXTEND(Concept, FullIntersect, VisitedSet)
13.  if Concept ∉ VisitedSet:
14.      Add Concept to VisitedSet
15.      Add Concept.Intersect to FullIntersect
16.      for each C_Parent ∈ Concept.Parents:
17.          GET-INTERSECT-EXTEND(C_Parent, FullIntersect, VisitedSet)

HAS-SUPERSET(Concept)
18.  if Concept.HasSuper = ∅:
19.      if |Concept.Intersect| < |Concept.O|:
20.          Concept.HasSuper ← TRUE
21.      else:
22.          for each C_Parent ∈ Concept.Parents ∧ Concept.HasSuper = ∅:
23.              if HAS-SUPERSET (C_Parent):
24.                  Concept.HasSuper ← TRUE
25.          if Concept.HasSuper = ∅:
26.              Concept.HasSuper ← FALSE
27.  return Concept.HasSuper

ADJUST(C_Base, C_New)
31.  for each O_i ∈ C_Base.Intersect:
32.      O2C[O_i] ← C_New
33.  C_Base.O ← C_Base.O − C_Base.Intersect
34.  C_New.Intersect ← C_Base.Intersect
35.  C_Base.Intersect ← ∅
```

Algorithm 3.8: Algorithms of supporting functions for lattice intersection.

The INTERSECT-LATTICE function begins by clearing the values of all temporal fields of the previous execution (line 1) and then performs the lattice intersection using the O2C vector (lines 2 through 7). If a concept for an object id is not found in the O2C vector, then the object id is added to the Intersect temporal field of the bottom concept. Upon completion of INTERSECT-LATTICE all concepts whose minimal object ids intersect with the extent of the new item will have their Intersect set accordingly.

Function GET-INTERSECT returns FullIntersect of a concept. It calls GET-INTERSECT-EXTEND to assign FullIntersect the Intersect of the concept and all ancestor concepts (line 10). Following the call to GET-INTERSECT-EXTEND, the object ids of FullIntersect are sorted. Sorted ids are needed to enable fast set operations. GET-INTERSECT-EXTEND is the same recursive algorithm as GET-O-EXTEND previously presented, except it adds the Intersect to the resulting set instead of O. The assignment to FullIntersect effectively caches the result. This eliminates lattice traversals in the event that GET-INTERSECT is called more than once for the same concept during insertion of an item. It would be desirable to cache the interim results gathered during execution of GET-INTERSECT-EXTEND. However, the algorithm and data structure is not conducive to such approach.

Function HAS-SUPERSET first checks if a value for the HasSuper field for the concept has previously been derived (line 18). If so, returns the value of HasSuper, otherwise it compares the number of object ids between the concept's intersection and object id sets. If the number of objects in the intersection set is less, then there exists at least one object id that is not in the extent of the item. In such case, HasSuper for the

concept is set to TRUE and that value is returned. If the size of the intersection set equals the object id set, then HAS-SUPERSET recurses on each parent concept. If any of the recursive calls returns TRUE, HasSuper for the concept is set to TRUE and that value is returned. If none of the recursive calls returns TRUE, HasSuper for the concept is set to FALSE and that value is returned. These recursive calls will set the HasSuper field for each parent, thereby caching the intermediary results. The ADJUST function will update the O2C vector to reference C_{New} for each of the percolated object ids (lines 31 and 32) and percolate the intersection sets from C_{Base} to C_{New} (lines 33 through 35).

In a preliminary test, Algorithm 3.7 (with supporting algorithms of Algorithm 3.8) took 446 seconds to execute against the Mushroom data set. Thus, Algorithm 3.7 provides a performance gain of a factor of three over Algorithm 3.5. While this is substantial, the executions times are is still far short of the performance of QuICL Oid-Full. Profile analysis revealed that 98% of execution time was spent executing the GET-INTERSECT and HAS-SUPERSET. Virtually all execution time is still spent performing the lattice intersection.

3.12 A Push Instead of Pull Intersection

Algorithm 3.8 uses a pull-down strategy to complete the intersection process for a given concept. That is, the GET-INTERSECT function derives FullIntersect by pulling down the Intersect from all ancestor concepts. This pull-down is performed as needed the first time the full set of intersect object ids is accessed for a given concept. An alternate approach is to push the object ids down into the lattice during that lattice intersect operation. This approach is given in Algorithm 3.9. Here, the LATTICE-INTERSECT function performs a call to LATTICE-INTERSECT-EXTEND to push an

object id into the lattice (line 6). LATTICE-INTERSECT-EXTEND is a recursive algorithm. It first checks whether the object ids have not already been added to a concept's FullIntersect. If so the recursion halts, otherwise the object id is added to a concept's FullIntersect and LATTICE-INTERSECT-EXTEND is called for each parent. Since a concept's FullIntersect is set during the lattice intersection, the GET-INTERSECT function for this algorithm simply returns FullIntersect (line 13).

In a preliminary test, Algorithm 3.7 using Algorithm 3.9 took 139 seconds to execute against the Mushroom data set. This is performance improvement of a factor of three over Algorithm 3.7 using Algorithm 3.8 and an order of magnitude improvement over Algorithm 3.5. However, performance of this algorithm is still an order of magnitude slower than QuICL Oid-Full. Profile analysis revealed that 93% of the execution time is now consumed executing the INTERSECT-LATTICE function. A large portion of execution time is still spent performing the lattice intersection. There are no further apparent improvements to this push approach.

3.13 A Hybrid Pull-Down and Bottom-up Intersection

As a lattice grows, the support functions of both Algorithms 3.8 and 3.9 exhibits a degradation of performance. The problem is that the object ids percolate to the top concepts in the lattice, yet the QuICL algorithms need the results of intersection for the concepts from the bottom-up. Regardless of pull-down or push-down approach, a traversal through the body of the lattice is required. A strategy to improve performance is to reduce the number of times the lattice is traversed. One means of achieving this strategy is to exploit proposition 3.1.

Proposition 3.1: If a concept has more than one child, then the set of object ids for the concept that intersects with an item's extent will be the intersection of the set of object ids that intersects with an item's extent of all child concepts.

Proof: A parent concept is the join concept (i.e., least common ancestor in an inverted lattice) of all of its children. Therefore, the parent's extent is the intersection of the extents of its children. Let O_i be an object id in the extent of a child that intersects with the items extent, then

i) if O_i is in the extent of all other children, then O_i is in each child's set of object ids that intersects the item's extent, or

ii) if O_i is not in the extent of at least one child, then O_i cannot be in the parent's set of object ids that intersects an item's extent since that set is a subset of the parent's extent.

Therefore, the set of object ids that intersects with an item's extent for a parent concept will be the intersection of the set of object ids that intersects with an item's extent of all child concepts.

Given proposition 3.1, the algorithm to pull-down object ids through the lattice is only required for concepts that have less than two children. For concepts with two or more children, the full intersection set of object ids of the concept can be derived by intersecting the full intersection sets of its children. In doing so, traversal of the lattice can be limited to only concepts with less than two children. For concepts with two or more children, only the immediate children are traversed. This savings in lattice traversal is at the expense of introducing a k-way intersection. Algorithm 3.10 provides an enhanced GET-INTERSECT function to supersede the one in Algorithm 3.8.

A preliminary test of Algorithms 3.7 and 3.8 using the GET-INTERSECT function of Algorithm 3.10 took 143 seconds to execute against the Mushroom data set. This performance is slightly slower than Algorithm 3.9. Thus, the hybrid algorithm may appear to be a fruitless approach. However, there exists an opportunity for further

```
INTERSECT-LATTICE(C_Bottom, O)
 1.  All temporal fields of all concepts within the lattice ←∅
 2.  for each O_i ∈ O:
 3.      C_Min ← O2C[O_i]                    // C_Min is concept with O_i at minimal
     position
 4.      if C_Min ≠ ∅:
 5.          Add O_i to C_Min.Intersect
 6.          INTERSECT-LATTICE-EXTEND(C_Min, O_i)
 7.      else:
 8.          Add O_i to C_Bottom.Intersect

INTERSECT-LATTICE-EXTEND(Concept, O_i)
 9.   if O_i ∉ Concept. FullIntersect:      // need only test last O_i in FullIntersect
10.      Add O_i to Concept. FullIntersect
11.      for each C_Child ∈ Concept.Children:
12.          INTERSECT-LATTICE-EXTEND(C_Child, O_i)

GET-INTERSECT(Concept)
13.  return Concept.FullIntersect
```

Algorithm 3.9: A push down algorithm for lattice intersection.

```
GET-INTERSECT(Concept)
 1.  if Concept.FullIntersect = ∅:
 2.      if |Concept.Children| < 2:
 3.          VisitedSet ←∅
 4.          GET-INTERSECT-EXTEND(
 5.                          Concept, Concept.FullIntersect, VisitedSet)
 6.          Sort Concept.FullIntersect
 7.      else:
 8.          Concept.FullIntersect ←
 9.                          ∩ GET-INTERSECT(C_i) ∀ C_i∈ Concept.Children
10.
11.  return Concept.FullIntersect
```

Algorithm 3.10: Hybrid pull-down and bottom-up intersection algorithm.

improvement. Profile analysis revealed 93% of the time is spent executing the hybrid GET-INTERSECT function. Of this time, 72% is consumed in the pull-down of object ids. The remaining time is spent performing the k-way intersections together with a marginal amount of overhead. The time consumed in the pull-down of object ids is a considerable percentage, yet it is performed only on a limited number of concepts. Herein lays the opportunity for improvement.

Consider the lattice illustrated in Figure 3.9. As items are inserted, the number of concepts in the lattice will grow exponentially and thus become very large. Of the concepts in the lattice, let the concepts that hold object ids at their minimal position be defined as *supports* and the concepts that have only one child be defined as *dependents*. A concept can be both a support and a dependent. Even though the number of concepts in the lattice can become very large, the number of supports and dependents is limited as stated by Propositions 3.2 and 3.3.

> **Proposition 3.2**. The number of support concepts in a compressed concept lattice data structure can be at most $|\mathcal{O}|$.
>
> *Proof*: Within a compressed lattice a given object id is stored in only one concept, its minimal concept. Therefore of all the concepts in the lattice, at most $|\mathcal{O}|$ number of concepts can hold an object id. Since a support concept is defined to be those concepts that hold object ids, the number of support concepts in the lattice is at most $|\mathcal{O}|$.
>
> **Proposition 3.3**. The number of dependent concepts in a concept lattice can be at most $|\mathcal{I}|$.
>
> *Proof*: A dependent concept is a concept with only one child. If a concept has only one child then it exists in the lattice only as a result of inserting an item having a set of objects that is a $\supset$ extent of the child concept. $|\mathcal{I}|$ are inserted into the lattice. Therefore, the number of dependent concept in a concept lattice can be at most $|\mathcal{I}|$.

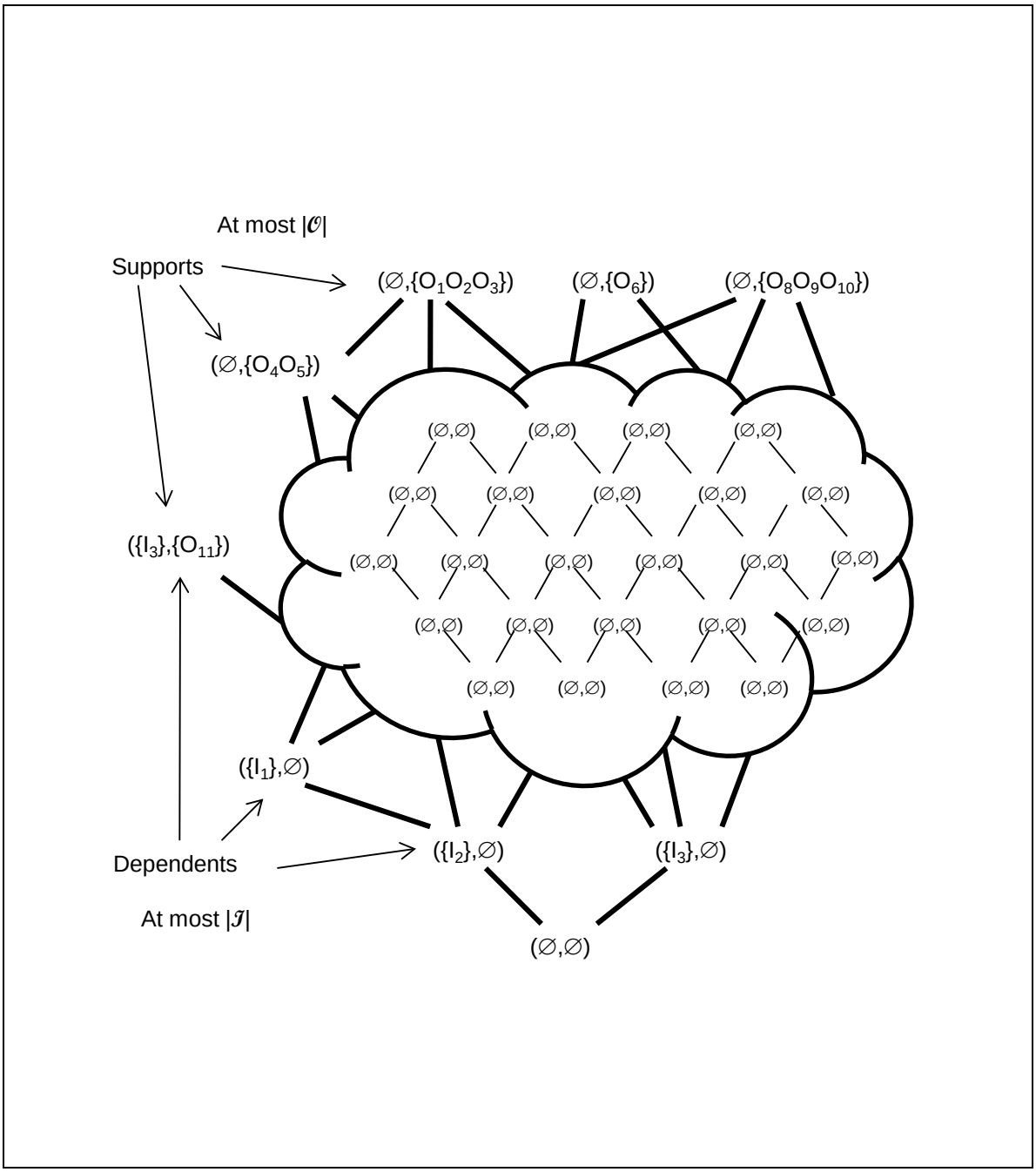


Figure 3.9: Lattice illustrating support and dependent concepts.

An outcome of proposition 3.2 is that a dependent concept is dependent upon at most $|\mathcal{C}|$ concepts. A performance gain can be achieved by maintaining a list of support concepts within each dependent concept. The pull-down of intersecting object ids can be performed by consulting the list of support concepts thereby bypassing a lattice traversal. Since by proposition 3.3 the number of dependent concepts is also limited, the overhead to maintain these support lists in each dependent concept is small. To enable maintenance of the list of supports, the support concepts will maintain a list of dependent concepts. As object ids percolate up though the concept lattice, the ADJUST function can update the dependent and support lists as needed.

Algorithm 3.11 provides modified GET-INTERSECT and ADJUST functions to utilize and maintain the support and dependent concept lists. GET-INTERSECT is modified to use the list of support concepts to pull-down the intersecting object ids (lines 3 and 4). The ADJUST function is modified to update the dependent and support lists as object ids are moved from concept C_{Base} to C_{New} (lines 13 through 18). While these modifications correctly maintain the existing support and dependent lists, they do not create the initial support lists for a concept representing the new item. However, to enable such functionality, the ADJUST function adds a reference to C_{New} from C_{Base} in a temporal field AdjustedTo (line 12).

Algorithm 3.12 provides modified BUILD-LATTICE and INTERSECT-LATTICE functions to initialize the support list for a new concept resulting from inserting an item into the lattice. Also provided is an ADD-LINK function to discard a support list when a concept is no longer a dependent. The INTERSECT-LATTICE function now returns a set of concepts that are the potential supports for the new item.

The set is initialized to be empty (line 13) and populated with a given concept the first time an object id is added to its intersection set (lines 17 and 18). The set is returned after iterating through all of the object ids in the item's extent (line 24). The BUILD-LATTICE function retains a reference to the returned set (line 3).

The INSERT function was defined to return the concept whose extent equals the set of object ids passed in the call. The rationale for returning a concept is to enable a parent-child link to be created when recursively calling the INSERT function. Since the returned concept can be a new concept, it can be exploited to provide initialization of a new concept's support list. The INSERT function can also return an existing concept. Therefore, the returned concept must be tested to determine if it is a new concept that needs to be initialized (line 5). During execution of the INSERT function, object ids of a potential support concept may percolate into other new concepts. In such case the AdjustedTo field will be set to reference the concept into which object ids have been percolated (line 11 of Algorithm 3.11). Therefore, the list of potential supports must first be processed before assigning the list to the new concept. Any concepts with an assigned AdjustedTo field are replaced by the concept referenced in the field (lines 6 through 8). In addition, the loop through the support concepts can add the new concept to the list of dependent concepts of each support (line 9). The corrected list of support concepts is then assigned to the new concept (line 10).

In a preliminary test, Algorithms 3.7 and 3.8 using the supporting functions defined of Algorithms 3.11 and 3.12 took 49 seconds to execute against the Mushroom data set. This represents a three times improvement over Algorithm 3.9 Profile analysis revealed that 78% of the execution time was spent executing the GET-INTERSECT and

Let tuples of dependent Concepts be augmented with the field {Supports}. Supports is a list of Concepts holding object ids that support a Dependent

Let tuples of support Concepts be augmented with the field {Dependents}. Dependents is a list of Concepts that have only one child and are dependent upon the Concept.

Let tuples of all Concepts be augmented with the temporal field {AdjustedTo}. AdjustedTo is the concept generated from a base concept. AdjustedTo is discarded following each item insertion.

```
GET-INTERSECT(Concept) :
 1.  if Concept.FullIntersect = ∅:
 2.      if |Concept.Children| < 2:
 3.          for each C_i ∈ Concept.Supports:
 4.              Add C_i.Intersect to Concept.FullIntersect
 5.          Sort Concept.FullIntersect
 6.      else
 7.          Concept.FullIntersect ←
 8.                  ∩ GET-INTERSECT(C_i) ∀ C_i ∈ Concept.Children
 9.  return Concept.FullIntersect

ADJUST(C_Base, C_New) :
10.  for each O_i ∈ C_Base.Intersect:
11.      O2C[O_i] ← C_New
12.  C_Base.AdjustedTo ← C_New
13.  if C_Base.Intersect ≠ ∅ ∧ C_Base.Dependents ≠ ∅:
14.      for each C_Dependent ∈ C_Base.Dependents:
15.          Add C_New to C_Dependent.Supports
16.          Add C_Dependent to C_New.Dependents
17.          if C_Base.O = ∅:
18.              Remove C_Base from C_Dependent.Supports
19.  C_Base.O ← C_Base.O – C_Base.Intersect
20.  C_New.Intersect ← C_Base.Intersect
21.  C_Base.Intersect ← ∅
```

Algorithm 3.11: Algorithm modifications to maintain supports and dependents.

```
BUILD-LATTICE(K{𝓘, 𝓞, 𝓡})
 1.  C_Bottom ← new Concept (∅, ∅)
 2.  for each I_i ∈ 𝓘:                          // o(I_i) is the set O derived from 𝓡
 3.      Supports ←INTERSECT-LATTICE(C_Bottom, o(I_i))
 4.      C_New ← INSERT(C_Bottom, I_i, o(I_i))
 5.      if |C_New.Children| < 2 ∧ C_New.Supports ≠ ∅:      // not an existing concept
 6.          for each C_Support ∈ Supports:                 // prepare the supports
 7.              if C_Support.AdjustedTo ≠ ∅:
 8.                  Replace C_Support with C_Support.AdjustedTo
 9.              Add C_New to C_Support.Dependents
10.          C_New.Supports ←Supports                       // set the support concepts
11.  return C_Bottom                                        // the lattice

INTERSECT-LATTICE(C_Bottom, O)
12.  All temporal fields of all concepts within the lattice ←∅
13.  Supports ← ∅
14.  for each O_i ∈ O:              // perform the intersection
15.      C_Min ← O2C[O_i]           // C_Min is concept with O_i at minimal position
16.      if C_Min ≠ ∅:
17.          if C_Min.Intersect = ∅:
18.              Add C_Min to Supports
19.          Add O_i to C_Min.Intersect
20.      else:
21.          if C_Bottom.Intersect = ∅:
22.              Add C_Bottom to Supports
23.          Add O_i to C_Bottom.Intersect
24.  return Supports

ADD-LINK(C_Parent, C_Child)
25.  Add C_Child to C_Parent.Children
26.  Add C_Parent to C_Child.Parents
27.  if |C_Parent.Children| > 1 ∧ C_Parent.Supports ≠ ∅:      // no longer a dependent ?
28.      for each C_Support ∈ C_Parent.Supports:
29.          Remove C_Parent from C_Support.Dependents
30.      C_Parent.Supports ← ∅
```

Algorithm 3.12: Algorithms to initialize supports and dependents of a new concept.

HAS-SUPERSET functions. The performance of these algorithms is approaching the QuICL Oid-Full algorithm. There is still one more step that can be made to improve performance.

3.14 The QuICL Oid-Less Algorithm

The development of an alternate QuICL algorithm as presented in Algorithms 3.5 through 3.12 maintains sets of object ids and utilizes set operations against the sets of object ids. Since the support concepts effectively provide a level of indirection to the object ids sets, the next step is to perform the set operations against the support concepts themselves. The GET-INTERSECT function will perform a k-way intersect of the support concepts of each child, the tuples in the ToProcessList retain sets of support concepts instead of sets of object ids, and the PURGE-SUBSET will purge tuples whose set of support concepts are a subset of the other tuples. This combined with a few additional changes results in the QuICL Oid-Less algorithm. Except for object ids in the temporal field Intersect, the concepts do not hold any object ids.

The complete QuICL Oid-Less algorithm, except for iceberg processing, is given in Algorithm 3.13. The concept tuples are composed of a list of items I, a list of parent concepts Parents, a list of child concepts Children, and a concept id (CID). The concepts do not include a list of object ids since they are no longer needed. However, the concept tuples do include the support (i.e., |all O|) and a record of the number of object ids that would be stored in the concept of the compress lattice structure. Both the parent and children lists are needed since both the parent and child traversals are performed. The concept id, uniquely assigned when a concept is created, is needed to enable fast execution of set operations. The tuples of support concepts are augmented with a list of

dependents concepts. Dually, tuples of dependent concepts are augmented with a list of support concepts. Lastly, all tuples are augmented with temporal fields AdjustedTo, Intersect, IntersectSupports, and HasSuper. IntersectSupports is a list of concepts that support a concept and have an intersection with the item's extent. The other fields are the same as previously defined. All temporal fields are discarded between item insertions.

The QuICL Oid-Less algorithm begins with the QUICL-OID-LESS function. It accepts a formal context and returns a constructed lattice. QUICL-OID-LESS is similar to the BUILD-LATTICE function of Algorithm 3.12 with a few changes. In order to correctly intersect lists of concepts, the derivation of the concepts supporting the dependent concepts must be consistent. Therefore, the list of support concepts that intersect an item's extent is derived for each dependent concept prior to inserting the item into the lattice. In algorithm 3.13, the list AllDependents is used to keep track of all the dependent concepts. Function GET-SUPPORTS-FOR-DEPENDENTS, called on line 4, is used to initialize each dependent concept with the list of support concepts that have an intersection with the item's extent.

The INSERT function for the QuICL Oid-Less algorithm given in Algorithm 3.13 is similar to the INSERT function previously presented. INSERT now accepts the concept's support and a list of support concepts. While Supports is not currently used, it will be used to add support for an iceberg lattice (presented in Section 3.15). INSERT calls function GET-INTERSECT-SUPPORTS instead of GET-INTERSECT. GET-INTERSECT-SUPPORTS checks if the intersection supports have already been derived, if not performs a k-way intersection on the intersection supports of the child concepts. For dependent concepts these intersection supports have already been set by

GET-SUPPORTS-FOR-DEPENDENTS. Following the call to GET-INTERSECT-SUPPORTS, INSERT obtains the number of intersecting object ids by summation of the intersection size of all intersection support concepts and sets the variable ISize (line 19). Conditions on ISize, HasSuper, and the Support passed in the call are used to identify if an $=$, $\subset$, $\supset$, or $\cap$ relationship exists between a concept's extent and the item's extent (lines 21, 23, 26, 28, and 30). The remainder of INSERT is the same as before except the tuples in the ToProcessList now contain sets of intersecting support concepts in place of sets of intersecting object ids. The tuples also retain ISize which is used as the support value when performing a recursive call to INSERT.

Let Concept be a tuple {I, NoOids, Support, Parents, Children, CID} where
I a list of items, NoOids the number object ids that would be at the minimal position within this concept, Support the support of a concept, Parents a list of parent concepts, Children a list of child concepts, and CID a concept id (uniquely assigned on creation)

Let tuples of dependent Concepts be augmented with the field {Supports} where
Supports is a list of Concepts holding object ids that support a Dependent

Let tuples of support Concepts be augmented with the field {Dependents} where
Dependents is a list of Concepts that have only one child and are dependent upon the Concept.

Let tuples of Concepts be augmented with the temporal fields {AdjustedTo, Intersect, IntersectSize, IntersectSupports, HasSuper} where:
AdjustedTo references a generated concept into which object ids have been percolated
Intersect a list of the concept's object ids at their minimal position that intersect with an item's extent
IntersectSize the size of Intersect at time of lattice intersection
IntersectSupports a list of ancestor Concepts that support the Concept and have an object id that intersects an items extent
HasSuper a boolean indicating at least one object id of a concept that does not intersect with an item's extent.
All temporal fields are discarded following each item insertion.

Let O2C be a vector whose index O_i identifies the concept $C \mid O \in C.O$

Let AllDependents be a list of dependent Concepts

QUICL-OID-LESS(K{$\mathcal{I}$, $\mathcal{O}$, $\mathcal{R}$})

1. $C_{Bottom} \leftarrow$ new Concept ($\varnothing$)
2. for each $I_i \in \mathcal{I}$: // $o(I_i)$ is the set O derived from $\mathcal{R}$
3. Supports $\leftarrow$INTERSECT-LATTICE(C_{Bottom}, $o(I_i)$)
4. GET-SUPPORTS-FOR-DEPENDENTS( )
5. $C_{New} \leftarrow$ INSERT(C_{Bottom}, I_i, $|o(I_i)|$, Supports)
6. if $|C_{New}.Children| < 2 \wedge C_{New}.Supports \neq \varnothing$: // not an existing concept
7. for each $C_{Support} \in$ Supports: // prepare the supports
8. if $C_{Support}.AdjustedTo \neq \varnothing$:
9. Replace $C_{Support}$ with $C_{Support}.AdjustedTo$
10. Add C_{New} to $C_{Support}.Dependents$
11. $C_{New}.Supports \leftarrow$Supports // set the support concepts
12. Add C_{New} to AllDependents
13. return C_{Bottom} // the lattice

Algorithm 3.13: The QuICL Oid-Less algorithm.

```
INSERT(C_Base, I_i, Support, Supports)
14.   ToProcessList ← ∅  // list of tuples {Type, Concept, Supp, IntersectSupports}
15.                       // with Type ∈ {SUPERSET, INTERSECT}
16.
17.   for each C_Parent ∈ of C_Base.Parents:                    // prepare-search phase
18.       ISupports ← GET-INTERSECT- SUPPORTS (C_Parent)
19.       ISize ← + C_Support.IntersectSize ∀ C_Support ∈ IntersectSupports
20.       HasSuper ← HAS-SUPER(C_Parent)
21.       if  ISize = 0:                                        // no relationship
22.           continue for each with next C_Parent
23.       else if ISize = Support ∧ HasSuper = FALSE:     // equal case
24.           Add I_i to C_Parent.I
25.           return C_Parent
26.       else if  ISize = Support ∧ HasSuper = TRUE:     // subset case
27.           return INSERT (C_Parent, I_i, Support, Supports)
28.       else if ISize < Support ∧ HasSuper = FALSE:     // superset case
29.           Add {SUPERSET, CParent, ISize, ISupports} to ToProcessList
30.       else if ISize < Support ∧ HasSuper = TRUE:      // intersect case
31.           Add {INTERSECT, C_Parent, ISize, ISupports} to ToProcessList
32.
33.   PURGE-SUBSETS(ToProcessList)
34.
35.   C_New ← New Concept({I_i}, Support)                   // create the new concept
36.
37.   ADJUST(C_Base, C_New)
38.
39.   for each T_i ∈ ToProcessList:                         // link phase to link in C_New
40.       if  T_i.Type = SUPERSET:
41.           Remove  parent-child link between T_i.Concept and C_Base
42.           ADD-LINK(T_i.Concept, C_New)
43.       else if  T_i.Type = INTERSECT:
44.           C_Parent ← INSERT (T_i.Concept, ∅, T_i.Supp, T_i.IntersectSupports)
45.           ADD-LINK(C_Parent, C_New)
46.
47.   // Intentionally left blank
48.
49.
50.   Sort C_New.Parents in order of decreasing Support
51.
52.   ADD-LINK(C_New, C_Base) maintaining decreasing Support order of C_Base.Parents
53.
54.   return C_New
```

Algorithm 3.13 continued: The QuICL Oid-Less algorithm.

```
GET-SUPPORTS-FOR-DEPENDENTS( )
 55.    for each C_Dependent ∈ AllDependents:
 56.        for each C_Support ∈ C_Dependent.Supports:
 57.            if C_Support.Intersect ≠ ∅:
 58.                Add C_Support to C_Dependent.IntersectSupports
 59.        Sort C_Dependent.IntersectSupports

INTERSECT-LATTICE(C_Bottom, O)
        ( Same as in Algorithm 3.12 except increments IntersectSize of a concept
          for which an object id is added to Intersect )

GET-INTERSECT-SUPPORTS(Concept)
 60.    if Concept.IntersectSupports = ∅:
 61.        Concept.IntersectSupports ←
 62.                ∩ GET- INTERSECT-SUPPORTS (C_i) ∀ C_i ∈ Concept.Children
 63.    return Concept. IntersectSupports

HAS-SUPERSET(Concept)
        ( Same as in Algorithm 3.8 except uses NoOids instead of |O| )

ADJUST(C_Base, C_New)
        ( same as in Algorithm 3.11 )

PURGE-SUBSETS(ToProcessList)
            // ToProcessList is a list of tuples {Type, Concept, ISize, IntersectCs} with
            // Type ∈ {NONE, SUBSET, SUPERSET, INTERSECT}
        ( Same as in Algorithm 3.3 except uses IntersectCs in place of Intersect )

COMPARE(C1, C2, Type)
            // C1 and C2 are vectors of Concepts, Type as defined above is an assumed
            // state of comparison. Returns a Type indicating the result of comparison.
        ( Same as in Algorithm 3.3 except compares the CID's of concepts )

ADD-LINK(C_Parent, C_Child)
        ( same as in Algorithm 3.12 )
```

Algorithm 3.13 continued: The QuICL Oid-Less algorithm.

3.15 Adding Iceberg Processing

The QuICL Oid-Less algorithm presented in Algorithm 3.13 constructs a complete concept lattice and thus include nodes that do not meet the minimum support threshold. To produce an iceberg lattice, the algorithm cannot simply discard the concepts in the lattice that do not meet the minimum support threshold, since these concepts may represent object ids stored at their minimal position. For example, Figure 3.10 depicts an iceberg lattice within the context of a full lattice. Bold text identifies the valid concepts that meet the minimum support threshold. The object ids in the remaining concepts will need to be represented in the iceberg lattice; otherwise the lattice will lose information. Some of the object ids can be correctly placed into the valid concept and maintain a compressed lattice structure (i.e., a given object id is stored in only one concept). For example, O_1 can be stored into the concept ($\{a_1\}$, $\varnothing$). However, there may exist object ids where this is not possible. In Figure 3.10, the object ids O_3, O_4, O_5, O_9, and O_{10} cannot be stored into valid concepts such that the lattice maintains a compressed lattice structure. To accommodate the representation of such object ids, a single layer of place holder concepts will be used. These concepts will be denoted as *iced*. The lattice at the top of Figure 3.11 depicts the iceberg lattice of Figure 3.10 with an extra iced concept to represent the needed object ids. While this lattice involves only one iced concept, additional iced concepts may appear as further insertions are performed. The lattice at the bottom of Figure 3.11 depicts such case.

R^V	
a_1	**O_1 O_2 O_3 O_4 O_5 O_8 O_9 O_{10}**
a_2	O_6 O_7
b_1	O_1 O_2 O_8
b_2	**O_3 O_4 O_5 O_6 O_7 O_9 O_{10}**
c_1	**O_2 O_3 O_4 O_5 O_6 O_8 O_9 O_{10}**
c_2	O_7
c_3	O_1
d_1	O_1 O_3 O_5 O_9 O_{10}
d_2	O_2 O_6 O_8
d_3	O_7
d_4	O_4

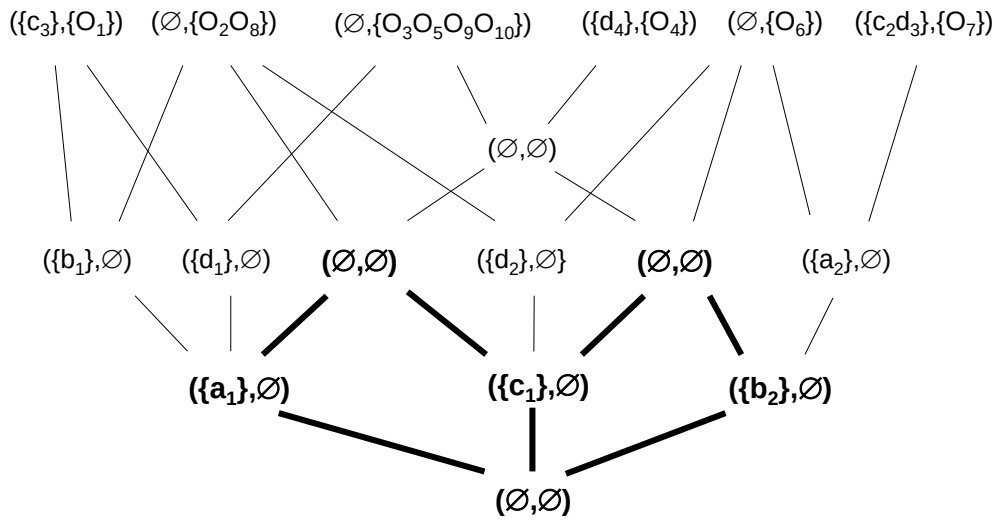


Figure 3.10: Iceberg lattice within a full lattice using a 60% threshold. The iceberg lattice is in bold text and lines. Grayed out text are concepts that do not meet the minimum support threshold.

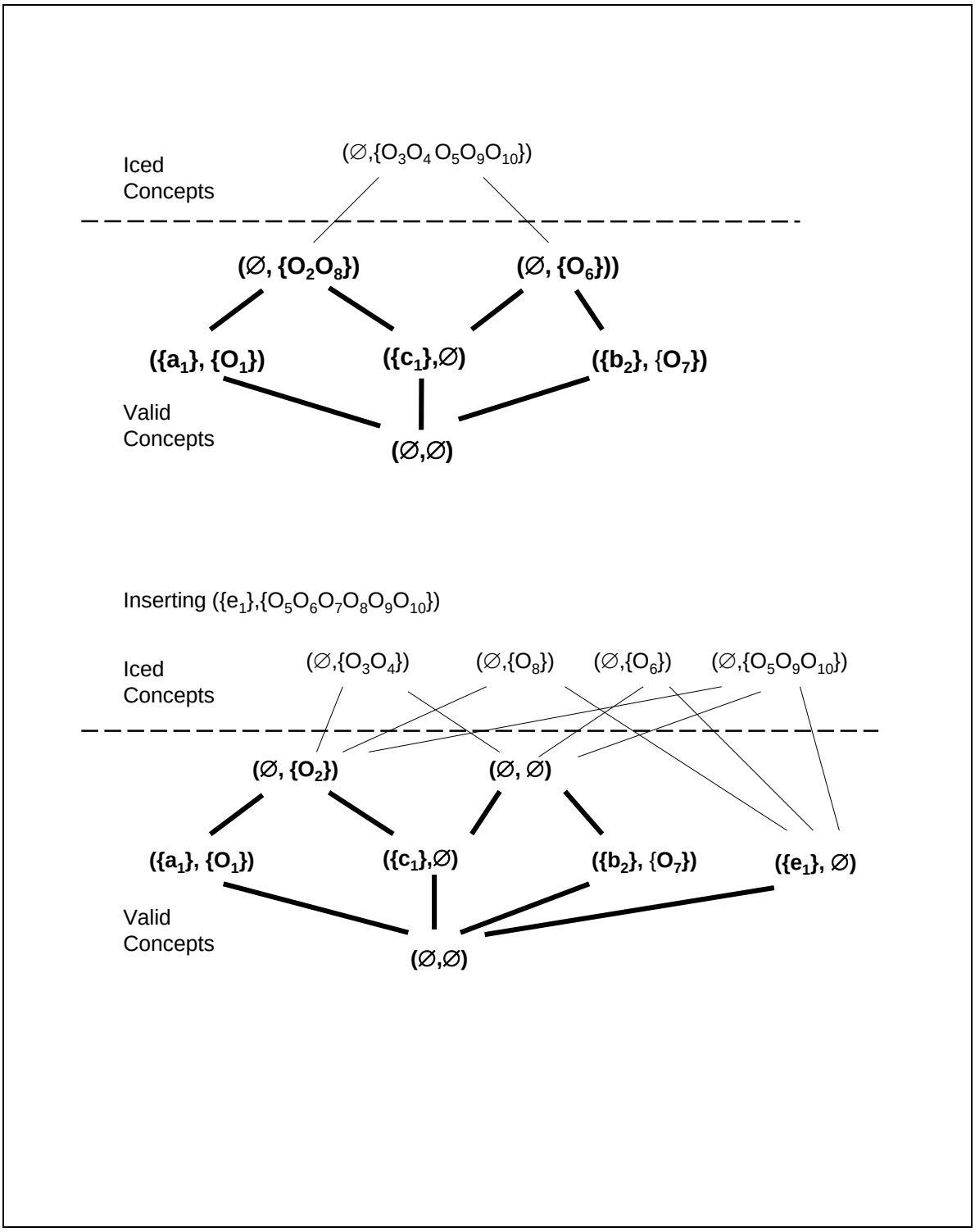


Figure 3.11: Iceberg lattice using a compressed structure. Top – iceberg lattice of Figure 3.10. Bottom – iceberg lattice with subsequent insertion of item e_1 with objects $\{O_6O_7O_8O_9O_{11}\}$.

Consider the insertion of item e_1 with objects $\{O_6O_7O_8O_9O_{10}\}$ as shown in Figure 3.11. The QuICL INSERT function will be called with the bottom concept (∅, ∅) as the base. The prepare-search phase will proceed to compare $\{O_6O_7O_8O_9O_{10}\}$ with each parent. The comparison of ($\{a_1\}$, $\{O_1\}$) will identify concepts (∅, $\{O_2O_8\}$) and (∅, $\{O_3O_4O_5O_9\ O_{10}\}$) as the intersection support concepts representing an intersection set of $\{O_8O_9O_{10}\}$. Since the size of the intersection set does not meet the minimum support threshold, these object ids cannot be represented in a valid concept. Instead, these object ids must be represented as a set of iced concepts. In order for the new concept to correctly represent the complete set of objects, in this case $\{O_6O_7O_8O_9O_{10}\}$, each identified iced concept must be linked as a parent. Similar scenarios are encountered when comparing $\{O_6O_7O_8O_9O_{10}\}$ to the other parent concepts of the base concept.

An approach to adding iceberg processing is to check that the size of the intersection set meets the minimum support threshold before recursively calling INSERT to process an INTERSECT tuple (line 31 of Algorithm 3.13). If the threshold is not met, then alternate processing to search through the ancestors is invoked to find or create iced concepts and link them to the new concept. This approach, however, may result in traversing the lattice. An alternate approach involves leveraging the set of intersection support concepts that are passed as an argument to the INSERT function. When INSERT completes by creating a new concept, the new concept must account for all of the support concepts. Each of the support concepts will either be the concept itself or a support concept of a parent. Since all parent concepts have been assigned prior to completion of the INSERT function, an alternate approach is to ignore iced concepts and intersections that result in iced concepts during the prepare-search phase. Then, at the end of the

INSERT function, perform a check to see if all support concepts have been accounted. Any support concepts that have not been accounted will either be a found iced concept or a concept from which an iced concept is to be extracted. A found iced concept is a concept whose size of intersect object ids equals the concept's support. The final step is linking the iced concepts, either found or extracted, to the new concept. This alternate approach has the advantage of identifying the iced concepts by traversing only the immediate parents, thereby avoiding potential traversals through the lattice. This approach also reduces the size of the ToProcessList, thereby reducing the execution time of the PURGE-SUBSETS function (Algorithm 3.13).

Algorithm 3.14 provides a modified INSERT function that supports construction of iceberg processing. The prepare-search phase will only process valid concepts (line 4). Furthermore, the prepare-search phase ignores any parent concept for which the size of intersection does not meet the minimum support threshold (lines 8 and 9). Lastly, INSERT calls the function ICED-INSERT in the event that either an iceberg concept or a parent concept whose intersection set size did not meet the minimum support threshold was encountered during the prepare-search phase (lines 34 and 35). ICED-INSERT is called after processing the ToProcessList.

Algorithm 3.15 provides the functions ICED-INSERT and EXTRACT-ICED-CONCEPT. ICED-INSERT begins by removing the intersection support concepts of all parent concepts from the support concepts (lines 1 and 2). The new concept is also removed from the support concepts (line 3). The remaining support concepts are either found iced concepts or are concepts from which an iced concept is to be extracted. Note that the AdjustedTo field will, when set, provide an indirection to a found iced concept.

Each found iced concept is added as a parent to the new concept (lines 6 through 9). From each support concept that is not a found iced concept, an iced concept representing the intersection set will be extracted. The extracted iced concept is added as a parent to the new concept (lines 11 and 12).

The EXTRACT-ICED-CONCEPT function, given in Algorithm 3.15, is used to extract an iced concept from another concept. The concept from which an iced concept is extracted can be a valid concept or another iced concept. In the case of another iced concept, the extraction effectively splits the iced concept. When splitting, the children of the split concept will become children of the extracted concept (lines 15 and 16). Also, the support of the split concept is adjusted accordingly (line 17). For a valid concept, the concept will be a child of the extracted concept (line 19). In both cases the ADJUST function is called to complete further adjustments to the lattice structure (line 21)

The last step in supporting iceberg processing is to only insert items whose number of objects meets the minimum support threshold. Algorithm 3.16 provides an updated QUICL-OID-LESS function with this change (line 2). A complete implementation, written in Java, of the QuICL Oid-Less algorithm with support for iceberg lattices is provided in Appendix C.

```
INSERT(C_Base, I_i, Support, Supports)
 1.  ToProcessList ← ∅  // list of tuples {Type, Concept, Supp, IntersectSupports}
 2.                     // with Type ∈ {SUPERSET, INTERSECT}
 3.
 4.  for each C_Parent ∈ of C_Base.Parents ∧ C_Parent.Support ≥ MinSupp:  // prepare-search
 5.      ISupports ← GET-INTERSECT-SUPPORTS(C_Parent)
 6.      ISize ← + C_Support.IntersectSize ∀ C_Support ∈ ISupports
 7.      HasSuper ← HAS-SUPERSET(C_Parent)
 8.      if  ISize < MinSupp:                          // no relationship or to be iced
 9.          continue for each with next C_Parent
10.      else if  ISize = Support ∧ HasSuper = FALSE:   // equal case
11.          Add I_i to C_Parent.I
12.          return C_Parent
13.      else if  ISize = Support ∧ HasSuper = TRUE:    // subset case
14.          return INSERT (C, I_i, Supp, Supports)
15.      else if ISize < Support ∧ HasSuper = FALSE:    // superset case
16.          Add {SUPERSET, C_Parent, ISize, ISupports} to ToProcessList
17.      else if ISize < Support ∧ HasSuper = TRUE:     // intersect case
18.          Add {INTERSECT, C_Parent, ISize, ISupports} to ToProcessList
19.
20.  PURGE-SUBSETS(ToProcessList)
21.
22.  C_New ← New Concept({I_i}, Support)                // create the new concept
23.
24.  ADJUST(C_Base, C_New)
25.
26.  for each T_i ∈ ToProcessList:                      // link phase to link in C_New
27.      if T_i.Type = SUPERSET:
28.          Remove parent-child link between T_i.Concept and C_Base
29.          ADD-LINK(T_i.Concept, C_New)
30.      else if T_i.Type = INTERSECT:
31.          C_Parent ← INSERT (T_i.Concept, ∅, T_i.Supp, T_i.IntersectSupports)
32.          ADD-LINK(C_Parent, C_New)
33.
34.  if encountered C_Parent.Support < MinSupp ∨ 0 < ISize < MinSupp:
35.      ICED-INSERT(C_New, Supports)
36.
37.  Sort C_New.Parents in order of decreasing Support
38.
39.  ADD-LINK(C_New, C_Base) maintaining decreasing Support order of C_Base.Parents
40.
41.  return C_New
```

Algorithm 3.14: Modified INSERT algorithm for iceberg processing.

```
ICED-INSERT(C_New, Supports)
 1.  for each C_Parent ∈ C_New.Parents:            // remove accounted supports
 2.      Supports ← Supports – GET-INTERSECT-SUPPORTS(C_Parent)
 3.  Remove C_New from Supports
 4.
 5.  for each C_Support ∈ Supports:                // process remaining supports as iced
 6.      if C_Support.AdjustedTo ≠ ∅:              // a found iced concept by indirection
 7.          ADD-LINK(C_Support.AdjustedTo, C_New)
 8.      else if not HAS-SUPERSET(C_Support):      // a found iced concept
 9.          ADD-LINK(C_Support, C_New)
10.      else:
11.          C_Iced ← EXTRACT-ICED-CONCEPT(C_Support)
12.          ADD-LINK(C_Iced, C_New)

EXTRACT-ICED-CONCEPT(C_Support)
13.  C_Iced ← New Concept(∅, C_Support.IntersectSize)
14.  if C_Support.Support < MinSupp):             // split an iceberg concept
15.      for each C_Child ∈ C_Support.Children:
16.          ADD-LINK(C_Iced, C_Child)
17.      C_Support.Support ← C_Support.Support – C_Iced.Support
18.  else:                                        // extracting from a valid concept
19.      ADD-LINK(C_Iced, C_Support)
20.
21.  ADJUST(C_Support, C_Iced)
```

Algorithm 3.15: Algorithms supporting iceberg processing.

```
QUICL-OID-LESS(K{ℐ, 𝒪, ℛ})
 1.  C_Bottom ← new Concept (∅)
 2.  for each I_i ∈ ℐ ∧ |o(I_i)| ≥ MinSupp:        // o(I_i) is the set O derived from ℛ
 3.      Supports ←INTERSECT-LATTICE(C_Bottom, o(I_i))
 4.      GET-SUPPORTS-FOR-DEPENDENTS( )
 5.      C_New ← INSERT(C_Bottom, I_i, |o(I_i)|, Supports)
 6.      if |C_New.Children| < 2 ∧ C_New.Supports ≠ ∅:   // not an existing concept
 7.          for each C_Support ∈ Supports:              // prepare the supports
 8.              if C_Support.AdjustedTo ≠∅:
 9.                  Replace C_Support with C_Support.AdjustedTo
10.              Add C_New to C_Support.Dependents
11.          C_New.Supports ←Supports                    // set the support concepts
12.          Add C_New to AllDependents
13.  return C_Bottom                                     // the lattice
```

Algorithm 3.16: Modified QuICL Oid-Less algorithm for iceberg processing.

3.16 Discussion for a Third QuICL Algorithm

The motivation for the QuICL Oid-Less algorithm is to address memory concerns with the storage of object ids in the concepts, whose number grows exponentially. The QuICL Oid-Less algorithm is successful in eliminating repeated object ids between item insertions, however, this is achieved at the expense of considerable complexity. Where the concepts of the QuICL Oid-Full algorithm simply maintain a list of items, list of object ids, and a list of parent references, the QuICL Oid-Less algorithm includes with each concept a parent concept list, a concept child list, count of object ids, support, and concept id; and augments many concepts with a support concept list, a dependent concept list, a temporary intersect object id list, a temporary intersect concept id list, adjusted to reference, intersect size, and a superset indicator. Thus, the savings in storing object ids are at the expense of consuming memory elsewhere. Furthermore, the QuICL Oid-Less algorithm may incur runtime overhead beyond its worth.

A compromise between the QuICL Oid-Full and QuICL Oid-Less algorithms may be found by considering the trie data structure which has surfaced several times literature (Valtchev et al., 2002, Nourine, & Raynaud 2002). A *trie* data structure (Knuth, 1998) is a tree based data structure that provides a compact representation by sharing common prefixes along branches, and enables efficient search, insertion, and set operations. Each edge denotes the addition of an element in a set. Thus for the QuICL algorithm, a trie can be employed to store the object ids of the concepts. Figure 3.12 depicts the use of a trie to store the object ids. At the top is the concept lattice of Figure 3.1 after inserting items a_1 though c_3 of relation in Figure 3.1. In this lattice, the ten object ids of the formal context are present in the concept lattice multiple times resulting in 51 entries. The

bottom of Figure 3.12 depicts a trie to represent all of the oid lists of the lattice. Due to common prefixes shared between the object id lists, the trie contains only 35 object id entries. Each concept of the lattice references a position in trie (as shown using dotted lines). For example, the concept with item set $\{a_1\}$ references the bottom left node in the trie. This node identifies the object id set $\{O_1O_2O_3O_4O_5O_8O_9O_{10}\}$ as indicated by the labeled edges on path from the root to the node. Each leaf in the trie will be referenced by a concept in the lattice. The number of leaves in the trie may be less than the number of concepts, since a concept may reference an interior node already present in the trie at the time a concept is generated. For example, the concept with item set $\{c_3\}$ references the top left node in the trie. This node identifies the object id set $\{O_1\}$.

3.17 Implementing a Trie in the QuICL Algorithm

While the trie data structure does provide savings in storing the number of object id entries, the trie must be carefully implemented to insure the savings in storing object ids is not outweighed by the overhead of the trie data structure. For example, each node in trie could be represented as a tuple consisting of a parent reference, child reference, sibling reference, and the object id of the parent edge. Child and sibling references facilitate representation of general trees (i.e., any number of children). Both forward and backward references are required. Forward references are needed to insert an object id list into the trie. Backward references are needed to identify an object id list. This representation defeats the objective to reduce space. Assuming 4 bytes to represent a reference, each object id in the trie would incur an additional 12 bytes overhead. Thus, the trie in Figure 3.12 using this representation would consume 560 bytes ($(12 + 4) \times 35$). This exceeds the space used by the original sets of object ids which is 204 bytes (4×51).

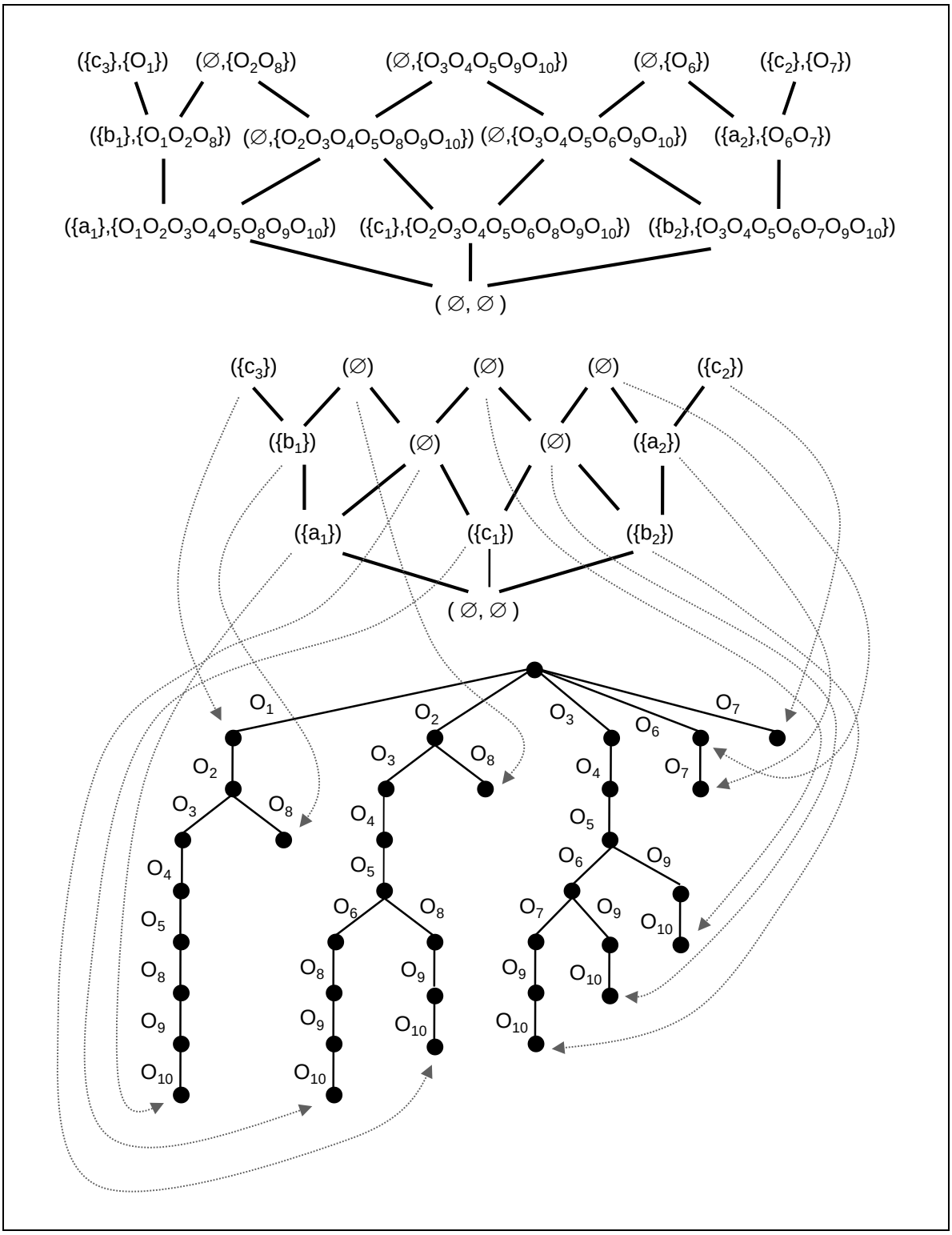


Figure 3.12: Concept lattice using a trie data structure to store object ids. Top – concept lattice of Figure 3.1 after inserting items a_1 through c_3. Bottom – same concept lattice with references into a trie data structure holding the object ids.

An alternative implementation is to place all of the object ids of a trie into a compound trie node. In this representation each node contains a vector of object ids and a parent reference. Parent references include a reference to a parent trie node and the index of the parent object id within its vector. Child references can be implemented with the aid of a global hash table. The reference to a parent trie node, index of the parent object id, and the first object id of the child trie node comprise a composite key that uniquely identifies a child trie node. Figure 3.13 provides an example of a trie using compound trie nodes.

This alternate implementation of the trie has the potential to realize savings in memory, although not apparent in the example of Figure 3.12. Assuming 4 bytes for each reference or integer value, the overhead for the compound trie node is at least 24 bytes. This is based on 4 bytes for the reference to the parent trie node, 4 bytes for the parent index position, and 16 bytes for the hash table entry. Furthermore, each concept incurs an additional 8 bytes overhead, since they now indirectly reference the oids through a reference to a trie node and an index position. Using these assumptions and estimates, Table 3.4 presents calculated memory savings (or excess) in using a trie data structure. Savings (or excess) are calculated for both the simple trie implementation and the compound trie node implementation. The lattice of Figure 3.12 and the lattice generated by the Mushroom data set are used in the analysis. For both cases, the overhead of the simple trie implementation exceeds the savings offered by the reduced the number of object id entries. On the other hand, the compound trie node implementation offers a real savings of 42 MBs for the Mushroom data set, cutting the memory requirements to store object ids by nearly half.

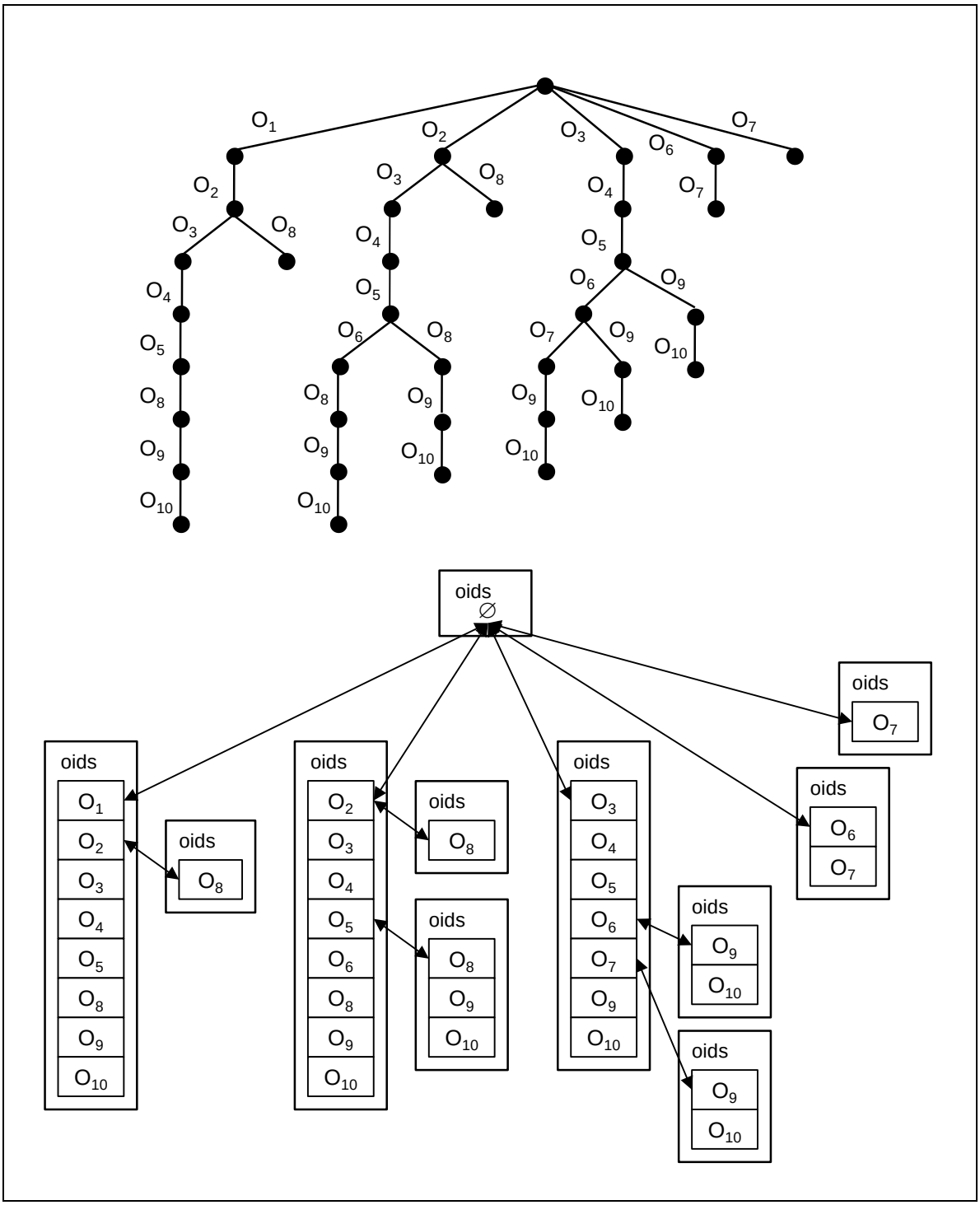


Figure 3.13: QuICL trie representation. Top – the trie of Figure 3.12. Bottom – depiction of same trie using compound trie nodes.

	Lattice of	
	Figure 3.11	Mushroom
No object Ids in formal context	10	8,124
No concepts	12	238,709
No object Id entries in the lattice	51	21,936,050
Total bytes to store object ids in concepts	204	87,744,200
No Object id entries in corresponding trie	35	9,577,434
No of compound trie nodes (excluding root)	10	218,034
Bytes to store object ids in a trie	140	38,309,736
Overhead bytes of using simple trie	420	114,929,208
Total bytes for object ids of simple trie	560	153,238,944
Overhead bytes of using compound trie	336	7,142,488
Total bytes for object ids of compound trie	476	45,452,224
Savings (excess) in using simple trie	(356)	(65,494,744)
Savings (excess) in using compound trie	(272)	42,291,976

Calculations are:

Total bytes to store object ids in concepts =
No object Id entries in the lattice × 4

Bytes to store object ids in a trie =
No Object id entries in corresponding trie × 4

Overhead bytes of using simple trie =
No Object id entries in corresponding trie × 12

Total bytes for object ids of simple trie =
Bytes to store object ids in a trie + Overhead bytes of using simple trie

Overhead bytes of using compound trie =
No of concepts × 8 + No of compound trie nodes × 24

Total bytes for object ids of compound trie =
Bytes to store object ids in a trie + Overhead bytes of using compound trie

Savings (excess) in using simple trie =
Total bytes to store object ids in concepts - Total bytes for object ids of simple trie

Savings (excess) in using compound trie =
Total bytes to store object ids in concepts - Total bytes for object ids of compound trie

Table 3.4: Sample calculations of memory savings (excess) of trie implementations.

3.18 The QuICL Oid-Trie Algorithm

In addition to savings in memory, the trie data structure enables two performance enhancements. The first is an optimization when performing intersections. Intersections are accomplished by processing two references into the trie. Typically the references start in different branches. One reference, the other reference, or both are then advanced to a parent object id position depending if the two referenced object ids are greater than, less than, or equal respectively. In the case of equal, the common object id is placed into an intersection buffer that will be used to create a new compound trie node upon completing the intersection process. The optimization is to terminate the intersection processing if at any time the two references reference the same position in the trie. At that point the remaining object ids of the two lists will be the same. The object ids in the intersection buffer can then be inserted into the trie using either reference as the insertion point. The second enhancement is to enable bi-directional traversal of the edge between a concept and the position in the trie of its object ids. The reference from a trie position to a concept can be implemented using a hash table. By provided a reference from a trie position to a concept, the INSERT function of the QuICL algorithm can directly determine if a concept for a given object id set already exists. If so, the item passed to the INSERT function is added to the concepts item list and the concept is returned. The reference to a concept effectively provides a short cut directly to an existing concept, thereby eliminating the recursive calls used to navigate into the lattice.

The bi-directional enhancement has further significance with respect to the asymptotic runtime complexity. By providing a direct lookup from a trie position to a concept, the time to process INTERSECT entries in the ToProcessList that lookup

concept in the lattice will be $O(\boldsymbol{d}')$, where $\boldsymbol{d}'$ is a fraction of $\deg_{avg}(\mathcal{L})$ depending on density. Thus, the term representing the time to process INTERSECT entries in the ToProcessList will not be a dominant factor in inserting a concept into the lattice. Therefore, the asymptotic complexity of the QuICL Oid-Trie algorithm will at least be $O(\boldsymbol{\ell}\, \boldsymbol{d}\, \boldsymbol{i})$ but could approach $O(\boldsymbol{d}^2\, \boldsymbol{c})$, where $\boldsymbol{\ell} = |\mathcal{L}|$, $\boldsymbol{d} = \deg_{avg}(\mathcal{L})$, $\boldsymbol{i}$ a density weighted mean on the cardinality of frequent item extents, and $\boldsymbol{c}$ is a small fraction of $|\mathcal{O}|$ depending density.

The QuICL Oid-Trie algorithm is given in Algorithms 3.17 and 3.18. Like the QuICL Oid-Full algorithm, a concept lattice is represented as a set of concepts linked only by references to parents. In addition to the lattice, a trie data structure is represented as a set of linked TrieNodes. A TrieNode is a tuple composed of a position within a parent TrieNode and a vector of object ids. A position within a TrieNode, TriePos, is a tuple composed of a reference to a TrieNode and an index into its vector of object ids. Thus, each concept in the QuICL Oid-Trie concept lattice is a tuple composed of a list of items, a trie position that identifies a set of object ids, and list of parents.

The QUICL-OID-TRIE function is similar to the QUICL-OID-FULL with a few modifications. In addition to creating a Concept to represent an empty lattice, a TrieNode is created to represent the root of an empty trie (line 2). Before calling the INSERT function to add an item and its extent into the lattice, a TRIE-INSERT function is used to add the extent into the trie (line 4). The returned trie position is then passed to the INSERT function in place of an object id set (line 5).

The TRIE-INSERT is passed a reference to a TrieNode identifying a branch in the trie passed where an object id set may be grafted, an object id set, and a index into the

object id set identifying a subset of object ids yet to be processed. The TRIE-INSERT function begins by traversing to object ids within the TrieNode while equal to the object ids yet to be processed (line 7 through 10). If the end of the object ids is reached, the object id set is already present in the trie. Therefore, a trie position for the object id last traversed in the TridNode is created and returned (line 12 and 13); otherwise a lookup in a hash table, TrieChildren, is used to determine if a child branch exists for the remaining object ids (line 15). If a child branch is found, TRIE-INSERT recurses (line 17 and 18). If not, a new TrieNode is created for the remaining object ids, the node is added to the hash table, and a trie position referencing the last object id in the new TrieNode is returned (lines 20 through 22).

The algorithm for the QuICL Oid-Trie INSERT function, Algorithm 3.18, is the same as the QuICL Oid-Full algorithm with appropriate changes. References to trie positions are used in place of object id lists. Lines 18 through 21 provide the insertion of a new object id set, resulting from an intersection, into the trie. A hash table, TrieConcepts, is used to provide the reference from a trie position to a concept. Line 26 adds an entry to TrieConcepts upon creating a new concept. Lines 1 through 4 perform a lookup into TrieConcepts to determine if a concept for a set of object ids already exists. If so, then item I_i is added to the concept's item list and the concept is then returned. The prepare-search phase no longer needs to process the case object ids equal to a parent's object ids (lines 11 through 13 of Algorithm 3.4), since this case is handled by the lookup into TrieConcepts (lines 1 through 4).

```
Let TrieNode be a tuple {ParentPos, O} where ParentPos a TriePos tuple representing
      a position in the trie, and O a vector of object ids.

Let TriePos be a tuple {TrieNode, Inx} where TrieNode a reference to a trie node
      within the trie and Inx an index position to an object id of its O.

Let Concept be a tuple {I, TriePos, Parents} where I a list of items, TriePos a position
      in the trie representing a list of object ids, and Parents a list of parent concepts.

TrieChildren← new hash table

QUICL-OID-TRIE(K{ℐ, 𝒪, ℛ}, MinSupp)
 1.   C_Bottom ← new Concept (∅, ∅)
 2.   Node_Root ← new TrieNode (∅, ∅)
 3.   for each I_i ∈ ℐ ∧ |o(I_i)| ≥ MinSupp:        // o(I_i) is the set O derived from ℛ
 4.       TP ← TRIE-INSERT(TN_Root, o(I_i), 0)
 5.       INSERT(C_Bottom, I_i, TP)
 6.   return C_Bottom                               // the lattice

TRIE-INSERT(Node, O, Inx)  // returns a TriePos for the O
 7.   NInx ← 0
 8.   while NInx < |Node.O| ∧ Inx < |O| ∧ Node.O[NInx] = O[Inx]:
 9.       NInx ← NInx + 1
10.       Inx ← Inx + 1
11.
12.   if |O| = Inx:                                // O already exists in the trie
13.       return new TriePos(Node, NInx)
14.
15.   N_Parent ← TrieChildren.lookup({Node, NInx, O[Inx]})
16.
17.   if N_Parent ≠ ∅:
18.       return TRIE-INSERT(N_Parent, O, Inx)
19.   else:
20.       N_Parent ← new TrieNode(new TriePos(Node, NInx), SUBSET(O, Inx, |O|))
21.       TrieChildren.put({Node, NInx, O[Inx]}, N_Parent)
22.       return new TriePos(N_Parent, |N_Parent.O| – 1)
```

Algorithm 3.17: The QuICL Oid-Trie algorithm.

```
TrieConcepts ← new hash table

INSERT(C_Base, I_i, TriePos)
1.  C_Ancestor ← TrieConcepts.lookup(TriePos)  // see if a concept already exists
2.  if C_Ancestor ≠ ∅:                         // for the object id set
3.      Add I_i to C_Ancestor.I
4.      return C_Ancestor
5.
6.  ToProcessList ← ∅  // list of tuples {Type, Concept, TriePos} with
7.                     // Type ∈ {SUPERSET, INTERSECT}, Concept a
8.                     // reference to the intersecting concept, and TriePos
9.                     // a position in the trie representing the set of object ids
10.                    // resulting from an intersection
11.
12. for each C_Parent ∈ of C_Base.Parents:                  // prepare-search phase
13.     if  O(TriePos) ⊂ O(C_Parent.TriePos):               // O( ) is the set of object ids
14.         return INSERT(C_Parent, I_i, TriePos)           // identified by a TriePos
15.     else if  O(TriePos) ⊃ O(C_Parent.TriePos):
16.         Add {SUPERSET, C_Parent, C_Parent.TriePos} to ToProcessList
17.     else if |O(TriePos) ∩ O(C_Parent.TriePos)| ≥ MinSupp:
18.         TP_Conv ← position in the trie where TriePos and C_Parent.TriePos converge
19.         O_New ← O(TriePos) ∩ O(C_Parent.TriePos) after converge point
20.         TriePos_New ← TRIE-INSERT(TP_Conv.TrieNode, TP_Conv.Inx, O_New, 0)
21.         Add {INTERSECT, C_Parent, TriePos_New)} to ToProcessList
22.
23. PURGE-SUBSETS(ToProcessList)
24.
25. C_New ← New Concept({I_i}, TriePos)                    // create the new concept
26. TrieConcepts.put(TriePos, C_New)
27.
28. for each T_i ∈ ToProcessList:                          // link phase to link in C_New
29.     if  T_i.Type = SUPERSET:
30.         Remove T_i.Concept from C_Base.Parents
31.         Add T_i.Concept  to C_New.Parents
32.     else if  T.Type = INTERSECT:
33.         C_Parent ← INSERT(T_i.Concept, ∅, T_i.TriePos)
34.         Add C_Parent to C_New.Parents
35.
36. Sort C_New.Parents in order of decreasing |O|
37.
38. Add C_New to C_Base.Parents in order of decreasing |O|
39.
40. return C_New
```

Algorithm 3.18: INSERT function of the QuICL Oid-Trie algorithm.

A complete implementation of the QuICL Oid-Trie algorithm, written in Java, is provided in Appendix D. The implementation incorporates the QuICL Oid-Full implementation enhancement to cache the results of intersection as described in Section 3.7.

3.19 Converting a Data Set to a Vertical Representation

Most data sets are organized in a horizontal representation. That is, a list of objects each with a set of attributes. The QuICL algorithms are dependent upon a vertical representation, whereby an item together with its set of objects are incrementally inserted into the lattice. Therefore, an algorithm to transpose a horizontal representation into a vertical representation is needed.

The CHARM algorithm has the same requirement of a vertical representation. Thus, the CHARM algorithm provides a transpose algorithm. The algorithm performs two passes over the data set and executes in linear time with respect to the number of objects. The first pass identifies the list of items and obtains counts of the number of objects for each. Between the passes, a buffer is allocated for each item. Each buffer is assigned an offset within an output data set where the object ids for the item will be stored. The second pass performs the transpose. For each object, the buffers represented by the set of items are identified and appended with the object id. As each buffer fills, it is flushed to the data set. After completing the second pass, all buffers are flushed to ensure the object ids are written to the data set. The result is a data set in vertical representation. An implementation of the transpose algorithm is provided in Appendix E along with a test harness used to execute the QuICL and GMA algorithms.

This algorithm is used as a pre-step to the QuICL algorithm. In addition, an order for the items can be specified. The order is used when assigning the file offsets to the buffers. While the transpose involves additional processing overhead, its asymptotic complexity is linear. Therefore, it will not be the dominant term in the overall process.

3.20 Summary of Methodology

This chapter presented the development of the Quick Iceberg Concept Lattice (QuICL – pronounced kwi-kəl) algorithms. These algorithms provide incremental construction of a concept lattice along the lines of the GMA algorithm, but approach the insertion process from the bottom of the lattice rather than 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 instead of iteration to facilitate the location of additional points of change and enable linkage between parent and child concepts. To support construction of iceberg lattices, the QuICL algorithms add data on an item by item basis and interchange the roles of the set of object ids and the set of items. These changes effectively invert the lattice. Furthermore, the QuICL algorithms exploit 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, each item is recorded in only one concept within the lattice (i.e., lowest position in the inverted lattice). This representation conserves space and enables direct extraction of association rules (see Chapter 1).

The QuICL algorithm has three derivations: Oid-Full, Oid-Less, and Oid-Trie. In the first derivation, all of the concepts in the concept lattice retain a complete list of the object ids (oids), hence the name “Oid-Full”. While preliminary results of the QuICL Oid-Full algorithm were very promising for some data sets, the performance gains do not hold against some others. An issue for the QuICL Oid-Full algorithm is the storage of

the complete list of object ids in each concept. The same object ids can be repeated in multiple concepts. Thus, an alternate algorithm, termed Oid-Less, was derived to eliminate the storage of object ids between the incremental insertions, although temporary lists of object ids are created and discarded during the insertion process. The QuICL Oid-Less algorithm is successful in eliminating repeated object ids between item insertions, however, this is achieved at the expense of considerable complexity. Therefore, the Oid-Trie derivation was developed as a compromise between Oid-Full and Oid-Less. Instead of eliminating the oid lists, it utilizes a trie data structure to store the object ids, thereby reducing memory requirements. In addition to gains in memory usage, the trie data structure also enabled a few performance enhancements.

Given a lattice $\mathcal{L}$ and a new item I_i with its set of object O, an incremental insertion algorithm such as QuICL is correct if it meets these requirements:

1) if $\exists\ C_i \in \mathcal{L} \mid$ extent of C_i = O, then insertion is completed by adding I_i to the intent of C_i, or

2) if $\neg\exists\ C_i \in \mathcal{L}, \mid$ extent of C_i = O, then a new concept C_{New} with intent $\{I_i\}$ and extent O must be created and inserted into the lattice such that:

   i. if $\exists\ C_b \in \mathcal{L} \mid C_b > C_{New} \wedge \neg\exists\ C_3 \in \mathcal{L} \mid C_b > C_3 > C_{New}$, then C_{New} will be a parent of C_b,

   ii. if $\neg\exists\ C_b \in \mathcal{L} \mid C_b > C_{New}$, then C_{New} will be a parent of bottom concept,

   iii. $\forall\ C_p \in \mathcal{L} \mid C_{New} > C_p \wedge \neg\exists\ C_3 \in \mathcal{L} \mid C_{New} > C_3 > C_p$, C_p will be a parent of C_{New},

   iv. $\forall\ C_s \in \mathcal{L} \mid$ extent of $C_s \not\subset$ O $\wedge$ extent of $C_s \cap$ O $\neq \varnothing \wedge \neg\exists\ C_3 \in \mathcal{L} \mid C_3 > C_s \wedge$ extent of $C_s \cap$ O = extent of $C_3 \cap$ O, another new concept C_{New}' with empty intent and an extent of $C_s \cap$ O must be inserted into the lattice with C_{New}' as a parent of both C_{New} and C_s, and

v. the resulting lattice satisfies the lattice connection property: a connection exists between two concepts C_1 and C_2 provided $C_1 < C_2$ and there is no concept C_3 for which $C_1 < C_3 < C_2$.

An initial formulation of the QuICL algorithm was proved to meet requirements 1, 2, 2.i, 2.ii, 2.iii, and 2.iv, but failed to meet requirement 2.v. This was corrected in the QuICL algorithm by removing non-maximal entries (i.e., entries containing object ids subsets of other entries) from an internal list of SUPERSET and INTERSECT entries. SUPERSET entries identify concepts that become immediate parents of a new concept. INTERSECT entries identify concepts are siblings to a new concept that find or generate other parent concepts.

It is postulated that the asymptotic runtime complexity for the QuICL Oid-Full algorithm will be at least $O(\boldsymbol{\ell}\, \boldsymbol{d}\, \boldsymbol{i})$, but could approach $O(\boldsymbol{\ell}\, \boldsymbol{d}^2\, \boldsymbol{c})$ or $O(\boldsymbol{\ell}\, \boldsymbol{d}\, \boldsymbol{d}'\, \boldsymbol{i}\, \boldsymbol{h})$, where $\boldsymbol{\ell} = |\mathcal{L}|$, $\boldsymbol{d} = \deg_{avg}(\mathcal{L})$, $\boldsymbol{i}$ is a density weighted mean on the cardinality of frequent item extents, $\boldsymbol{c}$ is a small fraction of $|\mathcal{O}|$ depending density, $\boldsymbol{d}'$ is a fraction of $\boldsymbol{d}$ depending on density, and $\boldsymbol{h}$ is a sub-linear function on the height of the lattice. An enhancement of the QuICL Oid-Trie algorithm eliminates $O(\boldsymbol{d}\, \boldsymbol{d}'\, \boldsymbol{i}\, \boldsymbol{h})$ from consideration. The asymptotic memory complexity is postulated to be $O(\boldsymbol{\ell}\, \boldsymbol{d}\, \boldsymbol{i})$.

Table 3.4 provides a summary of the advantages and disadvantages of the QuICL Algorithms. Table 3.4 also notes the strategy to store the object ids representing concept extents.

QuICL Derivation	Storage of Object Ids	Advantages	Disadvantages
Oid-Full	Object ids are replicated throughout lattice as needed to represent concept extents	• Simple data structure • Very good performance	• Memory usage for the lattice may be excessive due to repeated object ids
Oid-Less	Each object id is only stored once at its minimal position	• Reduction in memory consumption for storage of object ids • Intersection of concept ids, instead of object ids, may provide a gain in performance	• Added complexity may impact performance • Many additional fields in the representation of a concept may impact memory usage
Oid-Trie	Object ids are stored in a trie data structure to share common prefixes between concept extents	• Reduction in memory usage without the complexity of Oid-Less • Avoids a possible O($\ell d d' i h$) runtime complexity • Early halt of iterations when performing set operations may provide a performance gain	• Traversals between trie nodes when performing set operations may impact performance

Table 3.5: Comparison of QuICL derivations.

Chapter 4

Results

4.1 Introduction

This chapter presents the results of empirical evaluations and analysis of the QuICL algorithms. Seven public data sets often cited in other studies are used as the benchmarks. Included in this chapter is:

i) a presentation on the characteristics of each data set,

ii) an analysis of algorithm validity for the QuICL algorithms,

iii) results on the effect of input sort order on both runtime and memory usage for QuICL and GMA algorithms,

iv) runtime results of algorithms against the seven data sets over a spectrum of minimum supports,

v) memory results of algorithms against the seven data sets over a spectrum of minimum supports,

vi) performance analysis of the QuICL algorithms,

vii) empirical evidence to support asymptotic runtime complexity analysis,

viii) performance analysis of the GMA algorithm, and

ix) a report on the number of intersections performed by CHARM, QuICL, and GMA algorithms.

All three variants of the QuICL algorithms are compared against the CHARM, CHARM-L, GMA, and MAGALICE algorithms. The C version of the CHARM and CHARM-L algorithms were downloaded from the author's web site (Zaki, 2008) and

translated to Java. 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 itemset-oidset tree. The implementation was enhanced to cache in memory the object id sets. MAGALICE is part of the GALICIA open source project (Valtchev et al., 2003) and was downloaded from the GALICIA website (Valtchev, Godin, Missaoui, Huchard, Napoli, Grosser, et al., 2008). MAGALICE is written in Java. The GMA algorithm with modifications for iceberg processing, Algorithm 3.1, was directly implemented in Java. Source listing of the implementation is provided in Appendix A. Likewise the QuICL algorithms: Oid-Full, Oid-Less, and Oid-Trie, were implemented in Java. The source listing of each implementation is provided in Appendix B, C, and D respectively. Appendix E provides a source listing of a transpose algorithm used to convert a data set from a horizontal representation to a vertical representation. The source listing also includes a test harness to execute the QuICL and GMA algorithms.

All benchmarks are executed on an Intel Core 2 Duo CPU at 2.99 GHz with 3.0 GBs memory running Windows XP 2002 with service pack 3. All benchmarks are executed using the Java JRE version 1.5.0_07 with maximum heap size set to 1.6 GBs. 1.6 GB is the maximum setting allowed by the 1.5 version of the Java VM.

To measure the execution time, all algorithms were instrumented to time the lattice construction functions exclusive of any I/O time. Time before and after each execution point is obtained using Java's System.currentTimeMillis function. The

difference between the after and before times divided by 1000 is used to determine an execution time in seconds. All execution times reported in this chapter are in seconds rounded to two decimal places. 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 memory usage reported in this chapter are in megabytes (MBs).

Association rule mining and formal concept analysis are exponential problems. As such, the runtime performance and memory usage will grow exponentially as the minimum support for a data set is reduced. Thus, all charts in this chapter (except Figure 4.1) will use a logarithmic scale. By using a logarithmic scale the differences between the algorithms are more apparent. Furthermore, differences by order of magnitude can be readily observed. For example, Figure 4.1 displays the runtime performance of the algorithms against the Mushroom data set using both a fixed scale (top) and logarithmic scale (bottom). The logarithmic scale provides a clearer depiction of the differences between the algorithms. Differences in order of magnitude can be easily identified. For example, the QuICL Oid-Full algorithm is an order of magnitude better than GMA at $10\%_{supp}$ and two orders of magnitude better than MAGALICE. All charts display minimum support on the horizontal axis and runtime, or memory usage, on the vertical axis.

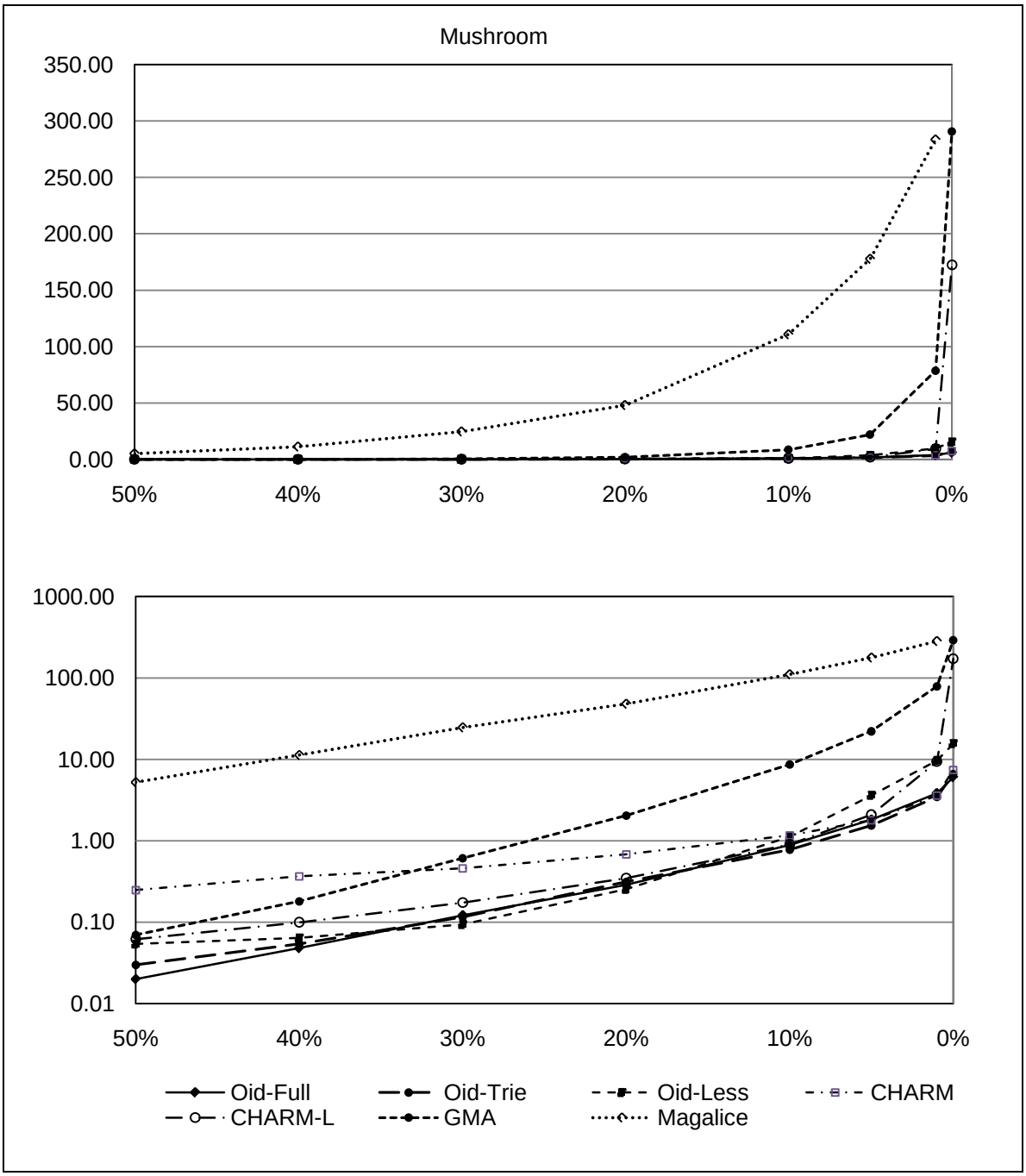


Figure 4.1: Logarithmic vs. fixed scale axis. Top – runtime performance using a fixed scale. Bottom – runtime performance using a logarithmic scale.

4.2 Data Set and Lattice Characteristics

The data sets used for the benchmarks are:

i) Mushroom,

ii) Chess,

iii) Pumsb,

iv) Pumsb*,

v) T10I4D100k,

vi) T25I10D10k, and

vii) 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 (2001). 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 ("Frequent Itemset Mining Dataset Repository", 2008). The T25I10D10k and T25I20D100k data sets were downloaded from the High Performance Computing Laboratory of The Institute of Information Science and Technologies, Pisa, Italy ("DCI: A hybrid algorithm for frequent set counting datasets", 2008).

The characteristics of these data sets together with characteristics of their generated concept lattices are given in Table 4.1. Density is calculated by $|\mathcal{R}| / (|\mathcal{O}| \times |\mathcal{I}|)$ where $|\mathcal{R}|$ is the total number of items for all objects found in the data set, $|\mathcal{O}|$ is number of objects, and $|\mathcal{I}|$ is the number of items. As can be observed in Table 4.1, the Chess and Mushroom data sets are dense, Pumsb and Pumsb* are marginal, and the remaining data sets are sparse. While the density measure does provide insight into predicting the behavior of mining algorithms, additional insight can be gained by a density profile. A density profile is a sorted plot of the number of objects per item. The axes are expressed in fractions of the total number of objects and total number of items. Therefore, the Y axis shows the support of an item. The density profiles for each data set are given in Figure 4.2. The Mushroom data set has about 10% of items with support in excess of $60\%_{supp}$, the support drops off linearly to $10\%_{supp}$ the over the next 50% of items, and exhibits a decay curve thereafter. The Chess data set has an approximate linear drop in support over the span of all items. The Chess data set contains 38 fields of which each has 2 values, thus representing 76 items. The near linear drop from $100\%_{supp}$ to $0\%_{supp}$ over the span of all items indicates a linear distribution of selecting one item of field over the other from being completely biased to being even. The Pumsb data set has a sharp linear drop from near $100\%_{supp}$ to $5\%_{supp}$ in the first few items. Since for the Pumsb, $|\mathcal{O}|$ is near 50,000, these few items will have sizable object id sets and thus represent a challenge for mining algorithms. The Pumsb* thus omits items whose support is 80% or more from the Pumsb data set. For both the Pumsb and Pumsb* there is a decay curve from $5\%_{supp}$ to $0\%_{supp}$ over the next 25% of items. The density profiles for both plot no items passed the approximate 30% position. This indicates that there are approximately

70% of items that are not represented in the data set. The density profiles for the T10I4D100K and T25I20D100K data sets both exhibit a decay curve beginning around $8\%_{supp}$ and have a fair percentage of items not represented in the data set. Since both these data sets contain 100,000 objects, the object id sets for the initial items are around 8,000 but quickly drop to under a 1,000. The T25I10D10K data set has a decay curve beginning at $10\%_{supp}$ and dropping to $5\%s_{upp}$ over the first 10% of items and then a near linear drop over all remaining items.

In addition to characteristics of the data sets, Table 4.1 includes characteristics of the generated lattices for a spectrum of minimum supports. Included is the number of concepts, average degree, maximum degree, and height. These values were obtained by executing QuICL and/or CHARM-L algorithms. The $20\%_{supp}$ entry for Pumsb was obtained using the CHARM algorithm (hence the omission of average degree and maximum degree for that entry). The 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. Table 4.1 clearly indicates the exponential growth in size of the lattice as the minimum support is lowered. The average degree and height tend to grow at a small slow rate as the minimum support is lowered, although the average degree exhibit a few anomalies (e.g., $0.01\%_{supp}$ of the T10I4D100K data set, $0.05\%_{supp}$ of the T25I20D100K data set, and $0.01\%_{supp}$ of the T25I20D100K data set). While for the T10I4D100K, T25I20D100K, T25I20D100K data sets the maximum degree can approach $|\mathcal{I}|$ at very low supports, the maximum degree for the lattices of the other data sets is only a fraction of $|\mathcal{I}|$.

Data Set	$\|\mathcal{O}\|$	$\|\mathcal{I}\|$	Density	Min Supp	$\|\mathfrak{L}\|$	Avg Degree	Max Degree	Height
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
				65%	496,069	8.31	28	15
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
				20%	122,262	5.94		
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	1.00%	5,256	3.53	800	9
				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

Table 4.1: Data set and lattice characteristics. $|\mathcal{O}|$ is number of objects, $|\mathcal{I}|$ is the number of items, and $|\mathfrak{L}|$ 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.

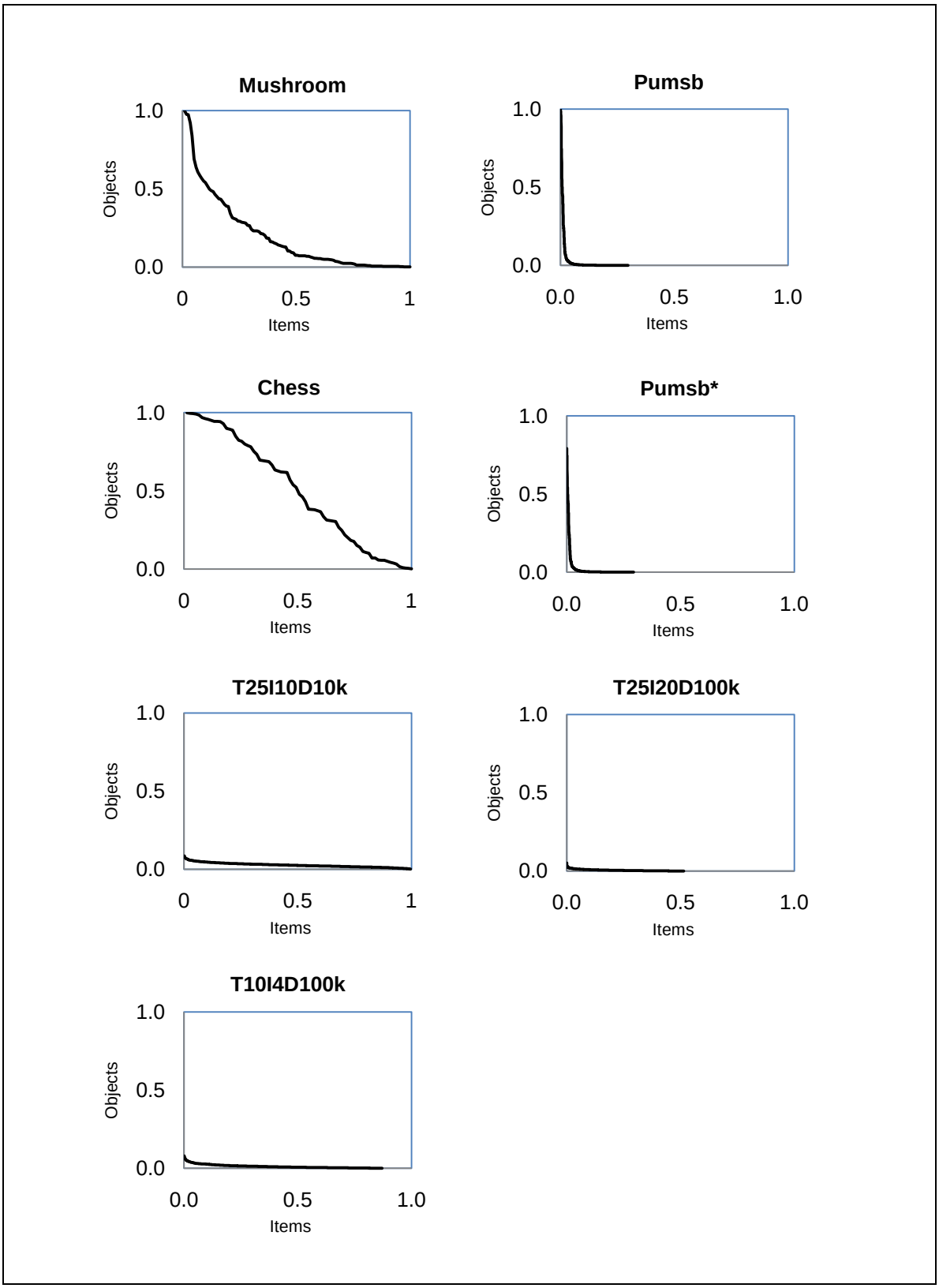


Figure 4.2: Density profiles of benchmark data sets.

4.3 Algorithm Validity

The QuICL algorithms were validated through a two pronged approach. First is the manual examination of execution paths of each during the incremental construction of small lattices. The small data sets of the examples of this report and small subsets (e.g., 10 items and 20 objects) of the Mushroom data set were used as the test cases. Manual examination was performed by single step and appropriately set breakpoints within a debugger. In addition to constructing complete lattices, the iceberg processing was also tested by setting the minimum support. For all test cases, the QuICL algorithm correctly constructed the concept lattices.

The second prong of validating the QuICL algorithms is to execute all algorithms including GMA, CHARM, CHARM-L, and MAGALICE against the benchmark data sets and compare the characteristics of the generated lattices. If the algorithms produced the same characteristics then the QuICL algorithms are deemed valid since;

i) the GMA, CHARM, CHARM-L, and MAGALICE are already considered valid by the research community as evidence by citations found in literature,

ii) data sets used are of sufficient size and variety to encompass special cases, and

iii) there is a near zero probability that different lattices generated from the same input will have the same characteristics.

The characteristics include the number of concepts in the lattices and the average degree of the lattice. The average degree is the average of the number concepts in the upper cover of each concept. The number of concepts by itself is not sufficient to claim that two lattices generated from the same input are equivalent. For example, omitting the PURGE-SUBSETS function in any of the QuICL algorithms will generate a lattice with

the same number concepts, but there will exist a number of invalid parent-child links resulting in a different average degree. Table 4.2 provides four such cases.

Data Set	Min Supp	$\|\mathcal{L}\|$	Average Degree	
			Valid	Invalid
Mushroom	10%	4,897	3.8365	4.7121
Mushroom	0%	238,709	5.7093	6.7231
T10I4D100k	0.05%	46,993	3.0998	3.1577
T10I4D100k	0.00%	2,347,374	4.2880	17.8058

Table 4.2: Cases of invalid average degree. Valid average degree is the average degree reported by the QuICL algorithms. Invalid average degree is the average degree reported by the QuICL algorithms without PURGE-SUBSETS. $|\mathcal{L}|$ is the number concepts reported by QuICL algorithms with and without PURGE-SUBSETS.

Appendix F provides the measurements of lattice characteristics for each algorithm. Table F.1 reports the number of concepts in each lattice, by each benchmark data set and selected minimum supports. Omitted entries in the tables are result of an algorithm exceeding the maximum heap size supported by the Java virtual machine. The QuICL algorithms together with CHARM and CHARM-L report the same number of concepts for all cases. The GMA algorithm reports a value one greater than the QuICL and CHARM algorithms for all data sets except Mushroom. This discrepancy is explained by the difference in lattice representations. The QuICL and CHARM algorithms do not count the bottom concept when reporting the number of concepts since, for these algorithms, the bottom concept represents an empty item set. The bottom concept only serves as an entry point into the lattice. GMA, on the other hand, includes the bottom concept since it may utilize this concept in the event of items that have all objects. Such is the case of the Mushroom data set. The MAGALICE algorithm reports the same number of concepts as GMA for many of the cases. This is expected since the

MAGALICE algorithm utilizes the GMA algorithm. The MAGALICE algorithm does, however, report in some cases a close but erroneous number. Such numbers are highlighted in Table F.1. These results indicate possible problems with the processing of jumpers[34] within the MAGALICE algorithm. Either MAGALICE has a different interpretation of minimum support, an error exists in the implementation supplied by the author, or MAGALICE is invalid.

Table F.2 reports the average degree of each lattice generated by each algorithm, by each benchmark data set and selected minimum supports. Values are not provided for the CHARM algorithm since it does not generate upper covers. The MAGALICE algorithm is also omitted since the number of concepts may be in error. All other algorithms report the same average degree for all cases.

Given the explainable consistency between the QuICL algorithms and the CHARM, CHARM-L, and GMA algorithms in both number of concepts and average degree, the QuICL algorithms are deemed valid.

4.4 Effect of Sort Order for the QuICL and GMA Algorithms

Before comparing the performance and memory usage of the QuICL algorithms against the CHARM, GMA, and MAGALICE algorithms, experiments were conducted to determine if the order of item insertion has an effect on performance and memory usage. If an effect is realized, then the ordering providing the best performance and memory usage will be used when comparing the algorithms. For these experiments, the items are sorted in ascending and descending support order. In addition, each data set as

[34] Jumper is the term used by the Magalice algorithm for a concept that is regenerated as a result of adding a new object. The addition of an object will change the supports of concepts. A concept that was discarded due to lack of a support may now meet the minimum support threshold. Such concept is given the term jumper.

downloaded from the respective repository is used to represent an unsorted state. The Chess, Mushroom, Pumsb, and T10I4D100k data sets were used in these experiments. The CHARM algorithms are not included since they perform sorting as an integral part of their processing. MAGALICE is not included since it loads the entire formal context into memory and then operates on the formal context using both an object index and item index lookup.

Figures 4.3, 4.4, and 4.5 provide the results on the effect of sort order on the runtime execution for the QuICL Oid-Full, QuICL Oid-Less, and QuICL Oid-Trie algorithms respectively. Figure 4.6 provides the results on the effect on the runtime execution of the GMA algorithm. Furthermore, Figures 4.7, 4.8, 4.9, and 4.10 provide the results on the effect on memory usage for the respective algorithms. The QuICL Oid-Full and Oid-Trie algorithms provide the best performance by incrementally inserting items in ascending support order. For QuICL Oid-Full the gain in performance ranges from a marginal amount to near two times (e.g., T10I4D100k at 0.005% $_{supp}$) depending the data set and selected minimum support. Greater gains are generally realized at lower supports. Similar gains are realized by QuICL Oid-Trie. The differences in execution time for these algorithms are attributed to the number of intersections. By inserting concepts in ascending item support order the lattice initially grows at small rate that accelerates towards later insertions. On the other hand, by inserting in descending order, the lattice grows rapidly as the initial items are inserted with the growth rate diminishing over subsequent items. Appendix G provides supporting evidence. While the QuICL algorithms use the lattice structure to navigate to the points of change within the lattice, a larger lattice will involve more intersections to locate those points. Therefore, by

inserting the items in ascending support order the number of intersections performed over the course of constructing the entire lattice will be restrained. For both QuICL Oid-Full and QuICL Oid-Trie the sort order has no effect on memory usage. This is due to the well-behavior of lattice construction.

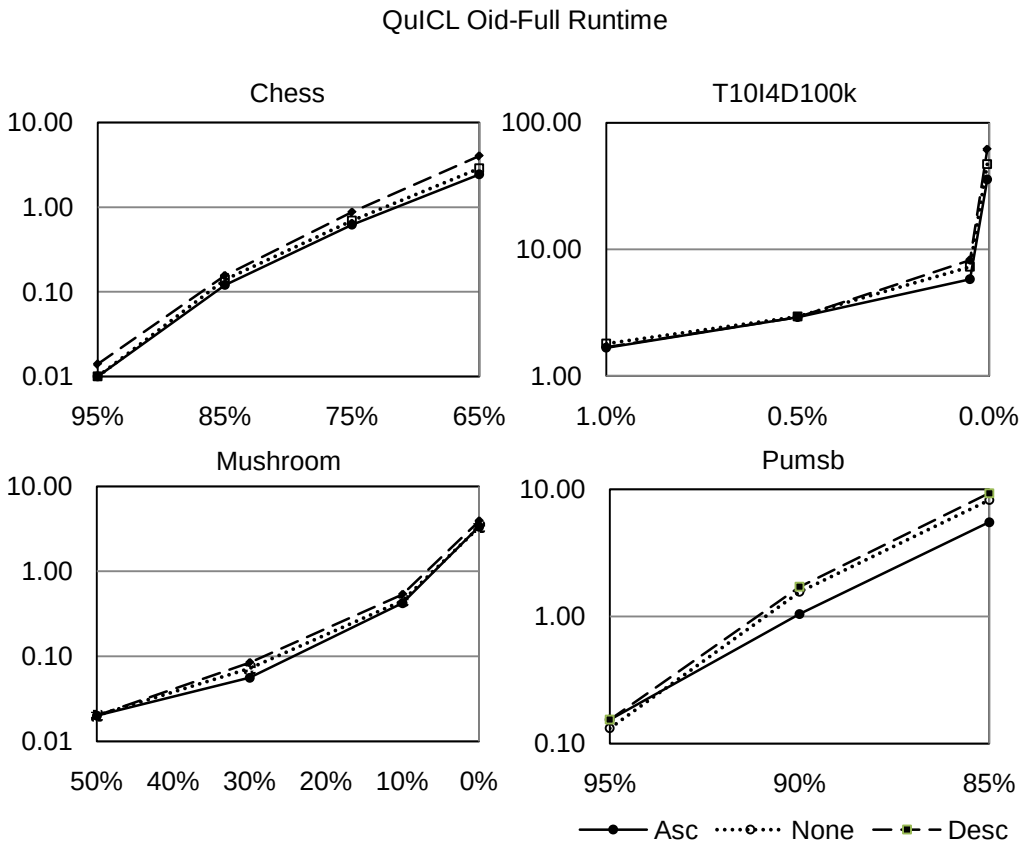
For the QuICL Oid-Less algorithm, the best performance and memory usage is attained by incrementally inserting items in descending support order. Even though the QuICL Oid-Less algorithm has the same fundamental structure as the QuICL Oid-Full and QuICL Oid-Trie algorithms, there are two factors that contribute to this conflicting preference. First, the QuICL Oid-Less performs intersection on sets of concepts rather than sets of object ids. By inserting items in descending support order, the concepts hold larger sets of object ids for longer initial period of time. During this time the cost of intersections is greatly reduced. Second, for each item insertion an intersection with the lattice as a whole is effectively performed. This results in creating numerous temporary object id and concept id sets. By inserting the items in descending support order the space consumed by these sets is restrained. While the lattice is still small, the intersection with a large object id set will be limited. Likewise, as the lattice gets large, intersection with small object id set is desired; otherwise the temporary sets will get exceedingly large (e.g., Chess data set of Figure 4.8). The size of the temporary sets not only impacts memory consumption, but also performance since it is these sets that are internally created and processed.

The QuICL Oid-Less algorithm exhibits a runtime preference for ascending order in the case the T10I4D100k data set. The T10I4D100k data set is a sparse data set. As a result, there is little gain in the performing intersections at the concept level. Any sets of

object ids are quickly broken into small sets as the algorithm proceeds. The effect of sort order on the number of intersections performed, as stated for the QuICL Oid-Full and QuICL Oid-Trie algorithms, has a greater impact on performance. The QuICL Oid-Less algorithm still exhibits a small memory preference for descending order. Given that the gain in memory is small, the amount of memory being consumed is well within the range of available memory, and the overall objective is to improve runtime performance; the best sort order for QuICL Oid-Less will be considered to be ascending when executed against sparse data sets.

The GMA algorithm also exhibits significant gains in performance by inserting the items in descending support order for dense data sets and moderate gains by inserting in ascending order for the T100I4D00k data set. On dense data sets, the differences in performance can be attributed to the cost of linking new concepts into the lattice. On creating a new concept, GMA must search the lattice to find its parents. This search is limited to the set of generated and modified concepts that are identified in the course of processing an item insertion until the point where a new concept is generated. When inserting the items in ascending support order, the generated concepts for the next item will be towards the bottom of the lattice. Thus, the number of generated and modified concepts encountered before that point will be large. This results in excessive time spent searching for parents. Inserting items in ascending support order effectively builds the lattice from the top-down and thus incurs a greater cost when searching for parents. Inserting items in descending order builds the lattice from the bottom up, greatly reducing the search cost. Section 4.9 provides evidence to the fact. On sparse data sets, the number of modified and generated concepts for each item insertion is drastically smaller

than on dense data sets. Therefore, the time to search for parents is significantly smaller and is less than costs of intersections. Thus on sparse data sets, the effect of sort order on the number of intersections performed, as stated for the QuICL Oid-Full and QuICL Oid-Trie algorithms, has a greater impact on performance. For the GMA algorithm, the sort order has no effect on memory usage.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	0.01	0.01	0.01
85%	1,885	0.12	0.14	0.16
75%	11,525	0.62	0.70	0.87
65%	49,240	2.44	2.89	4.04

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	385	1.67	1.80	1.69
0.500%	1,073	2.91	2.95	2.91
0.050%	46,993	5.80	7.28	8.22
0.005%	769,777	35.58	47.02	61.40

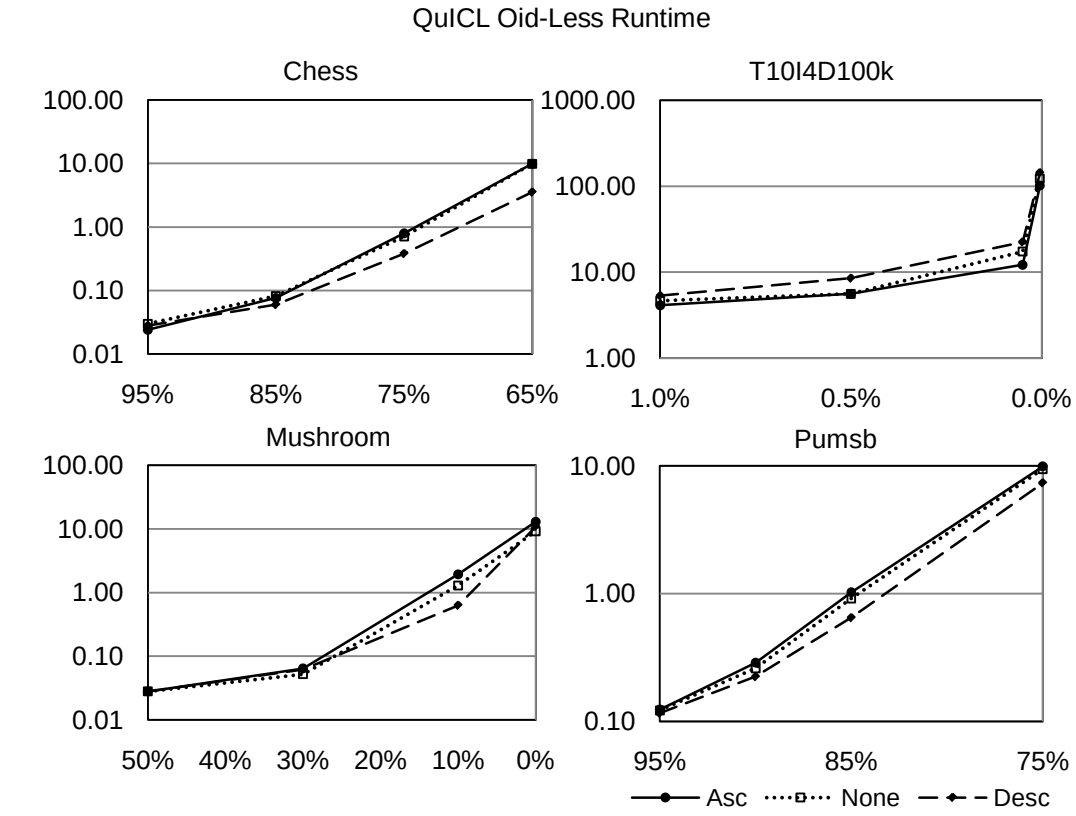
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	0.02	0.02	0.02
30%	427	0.06	0.07	0.08
10%	4,897	0.42	0.45	0.53
0%	238,709	3.36	3.32	3.90

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	110	0.15	0.13	0.15
90%	1,466	1.04	1.56	1.71
85%	8,513	5.50	8.20	9.29

Figure 4.3: Effect of item sort order on the QuICL Oid-Full runtime execution.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	0.02	0.03	0.03
85%	1,885	0.08	0.08	0.06
75%	11,525	0.80	0.71	0.38
65%	49,240	9.89	9.87	3.54

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	385	4.12	4.65	5.32
0.500%	1,073	5.61	5.61	8.51
0.050%	46,993	12.19	17.38	22.45
0.005%	769,777	101.59	122.70	142.91

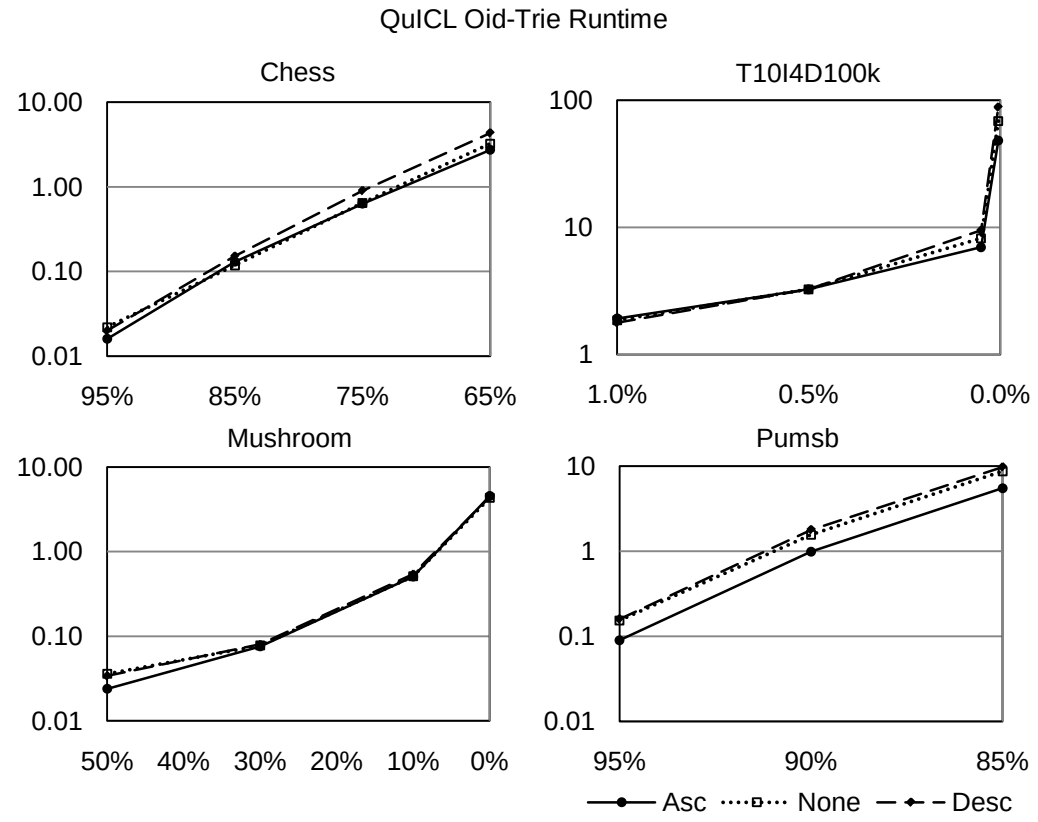
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	0.03	0.03	0.03
30%	427	0.06	0.05	0.06
10%	4,897	1.94	1.29	0.63
0%	238,709	12.97	9.18	10.80

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	110	0.12	0.12	0.12
90%	1,466	0.29	0.26	0.22
85%	8,513	1.02	0.91	0.65
75%	101,047	9.86	9.41	7.33

Figure 4.4: Effect of item sort order on the QuICL Oid-Less runtime execution.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	0.02	0.02	0.02
85%	1,885	0.13	0.12	0.15
75%	11,525	0.63	0.65	0.90
65%	49,240	2.73	3.24	4.33

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	385	1.92	1.86	1.78
0.500%	1,073	3.25	3.26	3.26
0.050%	46,993	6.97	8.22	9.48
0.005%	769,777	48.35	68.73	88.48

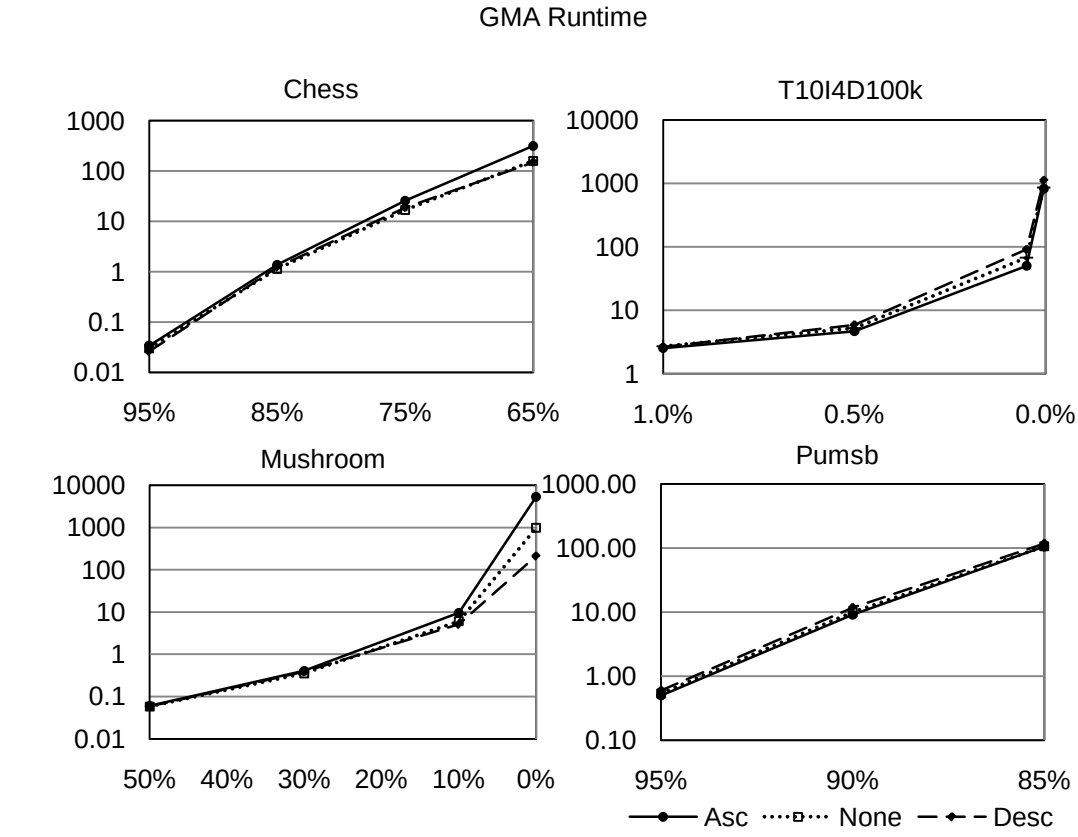
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	0.02	0.04	0.03
30%	427	0.08	0.08	0.08
10%	4,897	0.51	0.51	0.54
0%	238,709	4.55	4.32	4.42

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	110	0.09	0.15	0.16
90%	1,466	0.99	1.57	1.79
85%	8,513	5.50	8.70	9.78

Figure 4.5: Effect of item sort order on the QuICL Oid-Trie runtime execution.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	75	0.03	0.03	0.03
85%	1,886	1.37	1.13	1.23
75%	11,526	25.61	16.84	18.85
65%	49,241	314.97	159.01	148.68

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	386	2.53	2.71	2.65
0.500%	1,074	4.69	5.25	5.91
0.050%	46,994	49.86	67.25	90.90
0.005%	769,778	822.74	861.79	1107.74

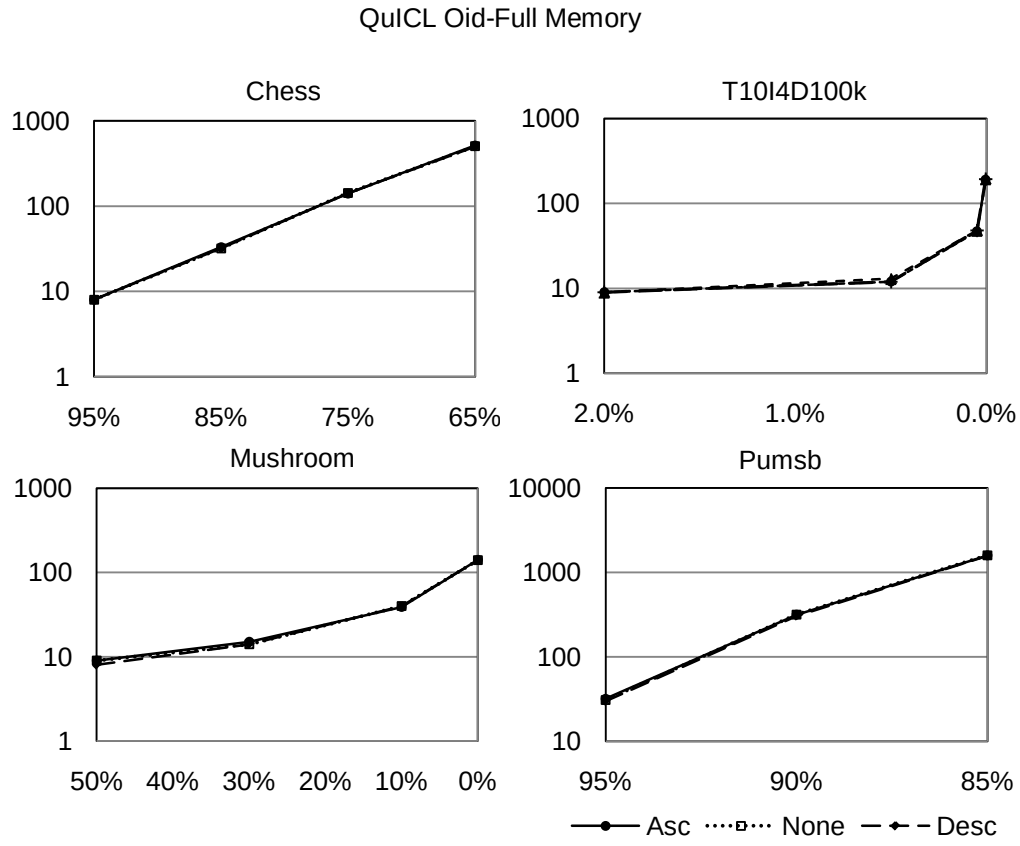
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	0.06	0.06	0.06
30%	427	0.40	0.35	0.39
10%	4,897	9.50	6.05	5.09
0%	238,709	5278.02	978.76	210.29

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	111	0.50	0.54	0.60
90%	1,467	9.06	9.95	11.73
85%	8,514	105.06	105.77	117.44

Figure 4.6: Effect of item sort order on the GMA runtime execution.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	8	8	8
85%	1,885	30	30	30
75%	11,525	138	137	136
65%	49,240	492	491	519

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	385	11	11	11
0.500%	1,073	13	12	13
0.050%	46,993	48	49	49
0.005%	769,777	230	219	201

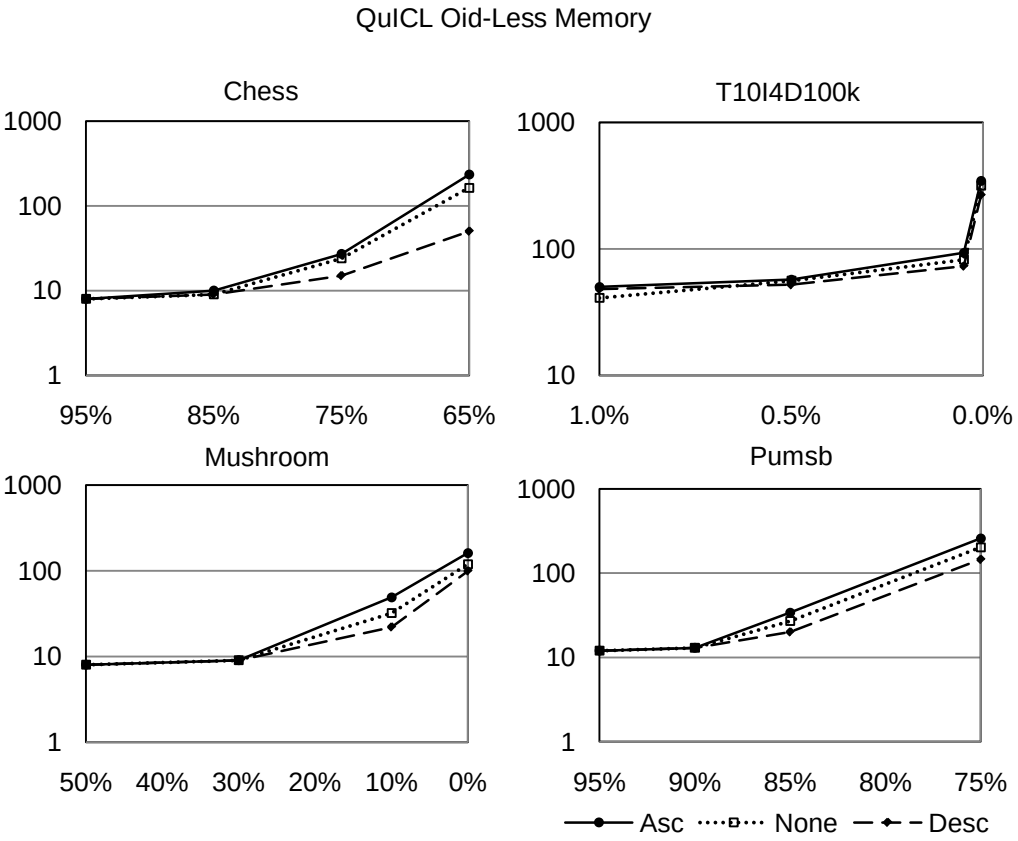
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	8	8	8
30%	427	13	13	13
10%	4,897	42	42	41
0%	238,709	153	149	145

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	110	29	29	28
90%	1,466	287	295	295
85%	8,513	1557	1577	1577

Figure 4.7: Effect of item sort order on the QuICL Oid-Full memory usage.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	8	8	8
85%	1,885	10	9	9
75%	11,525	27	24	15
65%	49,240	234	163	50

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	385	50	41	48
0.500%	1,073	57	56	52
0.050%	46,993	93	82	73
0.005%	769,777	345	315	265

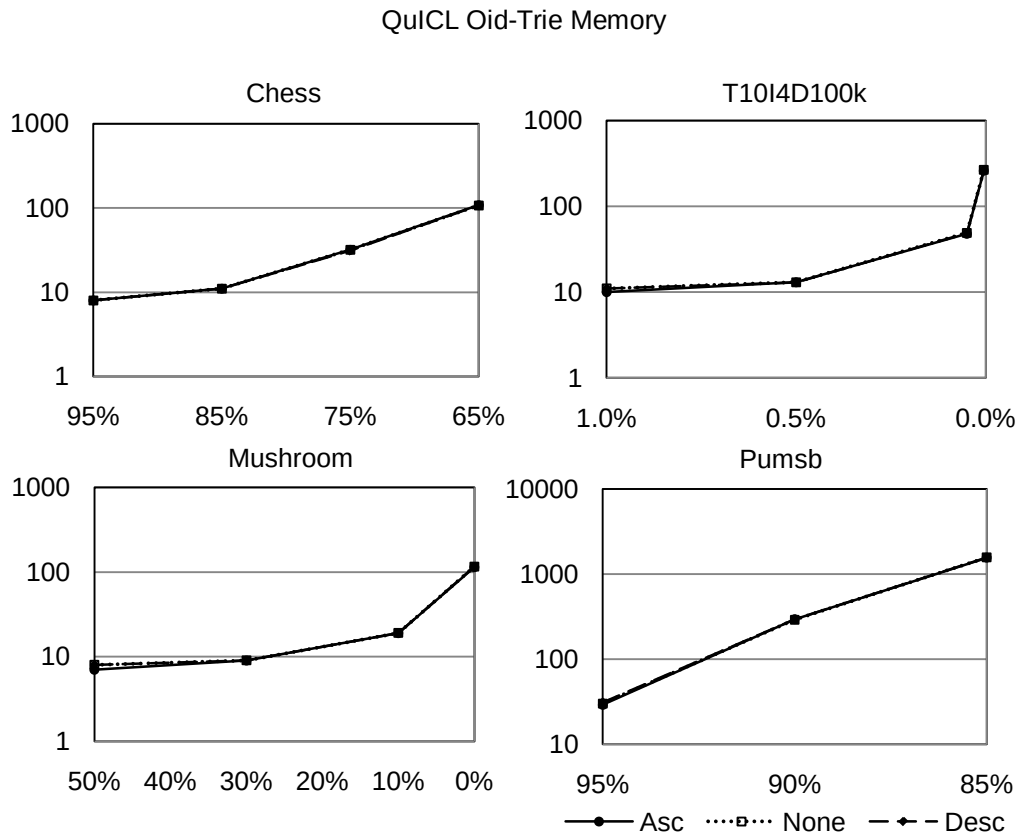
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	8	8	8
30%	427	9	9	9
10%	4,897	49	32	22
0%	238,709	161	119	99

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	110	12	12	12
90%	1,466	13	13	13
85%	8,513	34	27	20
75%	101,047	258	202	145

Figure 4.8: Effect of item sort order on the QuICL Oid-Less memory usage.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	8	8	8
85%	1,885	11	11	11
75%	11,525	32	32	31
65%	49,240	108	107	107

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	385	10	11	11
0.500%	1,073	13	13	13
0.050%	46,993	48	49	49
0.005%	769,777	261	268	269

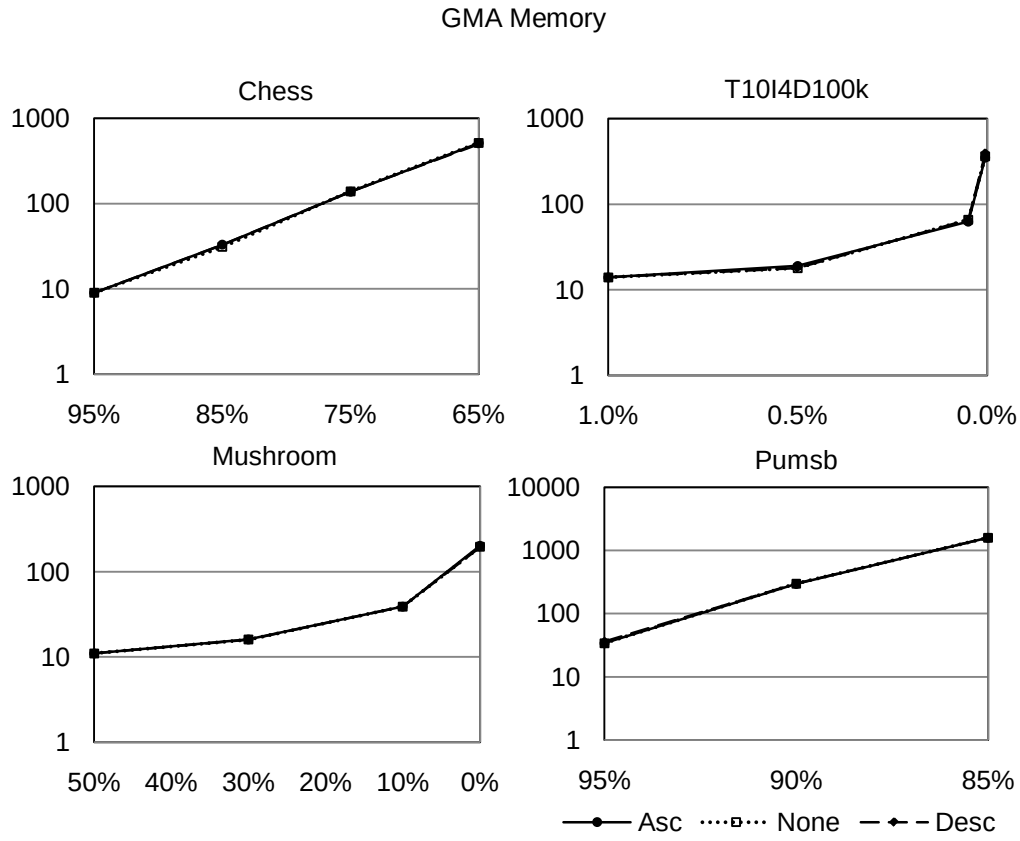
Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	7	8	8
30%	427	9	9	9
10%	4,897	19	19	19
0%	238,709	114	116	116

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	110	29	30	31
90%	1,466	295	291	295
85%	8,513	1557	1573	1573

Figure 4.9: Effect of item sort order on the QuICL Oid-Trie memory usage.



Chess

Min Supp	|ℒ|	Asc	None	Desc
95%	74	9	9	9
85%	1,885	33	31	33
75%	11,525	138	139	139
65%	49,240	513	515	500

T10I4D100k

Min Supp	|ℒ|	Asc	None	Desc
1.000%	386	14	14	14
0.500%	1,074	19	18	18
0.050%	46,994	63	66	66
0.005%	769,778	352	362	394

Mushroom

Min Supp	|ℒ|	Asc	None	Desc
50%	45	11	11	11
30%	427	16	16	16
10%	4,897	39	39	39
0%	238,709	202	196	194

Pumsb

Min Supp	|ℒ|	Asc	None	Desc
95%	111	34	34	36
90%	1,467	295	297	298
85%	8,513	1582	1582	1583

Figure 4.10: Effect of item sort order on the GMA memory usage.

4.5 Comparison of Algorithm Execution Time

The runtime results of executing the algorithms are given in Figures 4.11 through 4.17 for the Chess, Mushroom, Pumsb, Pumsb*, T10I4D100k, T25I10D10K, and T25I20D100k data sets respectively. All reported times are the average of five separate test executions of the same test case. Ascending item support order is used for the QuICL Oid-Full and QuICL Oid-Trie algorithms for all data sets, and for the GMA and QuICL Oid-Less algorithms on sparse data sets. Descending order is used for GMA and QuICL Oid-Less on dense data sets. All other algorithms are unsorted since either the sort order is an integral part of the algorithm (e.g., CHARM) or is not applicable (e.g., MAGALICE). All times are in seconds. The plots in each figure display seconds on the vertical axis and minimum support on the horizontal axis.

GMA is generally slower than the QuICL and CHARM algorithms 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. All concepts in ascending support order from the top of the lattice through the concept that is the generator for the next item's object ids are visited. As a result, the GMA algorithm visits and intersects more concepts than needed. The QuICL algorithms use the lattice structure to navigate to a more limited subset of concepts thereby reducing the number of intersections. The CHARM algorithms provide pruning on its itemset-oidset trie as its means to limit the number of intersections. Secondly, the 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. For parent concepts already

present in the lattice the QuICL Oid-Full and QuICL Oid-Less algorithms navigate the lattice structure as needed. The QuICL Oid-Trie algorithm alternatively uses a hash table to directly locate these parents. Other parents are dynamically created and directly linked using recursive insertion. The CHARM-L algorithm uses set operations on sets of generator concepts associated with each concept to identify and link parent concepts.

The MAGALICE algorithm demonstrates the worst performance of all the algorithms. Its performance when compared to the GMA algorithm is at least an order of magnitude slower, in some cases three order of magnitude (e.g., Pumsb*) at high supports. The performance does however, appear converge with the GMA as the support is lowered. The plots for the Chess, Mushroom, and T25I10D10K data sets indicate that the performance of MAGALICE will be comparable to GMA at $0\%_{supp}$, although no conclusive measurements were attainable. These results indicate considerable processing is involved with the generation of jumpers[35]. As the support is lowered this processing diminishes since there are fewer discarded concepts that need to be regenerated. Indeed at $0\%_{supp}$, no concepts will be discarded and thus there will be no jumpers. At $0\%_{supp}$ MAGALICE reverts to the underlying GMA processing.

The CHARM algorithm 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 frequent closed item sets. Furthermore, CHARM uses a difference based representation for the object ids below the

[35] Jumper is the term used by the MAGLICE algorithm for a concept that is regenerated as a result of adding a new object. The addition of an object will change the supports of concepts. A concept that was discarded due to lack of a support may now meet the minimum support threshold. Such concept is given the term jumper.

first level in its itemset-oidset tree. Since these data sets contain a number of items having large object id sets (e.g., greater than 200), 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 frequent item sets) and equate to an absolute support of 100, 28, and 20 respectively. As the support is reduced below these points, the CHARM algorithm is outperformed by the QuICL Oid-Full and QuICL Oid-Trie algorithms. For the remaining data set, Mushroom, CHARM is outperformed by QuICL Oid-Full and QuICL Oid-Trie over all supports.

The CHARM-L algorithm 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. The CHARM-L algorithm still outperforms the QuICL Oid-Full and QuICL Oid-Trie algorithms on the Pumsb and Pumsb* algorithms over near all supports, and the T10I4D100k, T25I10D10k, and T25I20D100k at relatively high supports (although higher than with CHARM). This indicates that the difference based representation of the underlying CHARM algorithm is still providing a gain. However, the gain is significantly diminished by the processing required to derive the upper covers. The gain in

performance of CHARM-L over QuICL on Pumsb* is approximately a factor of two. For the Mushroom and Chess data sets, CHARM-L is outperformed by both QuICL Oid-Full and QuICL Oid-Trie over all supports.

The QuICL Oid-Full algorithm 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 at only 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. While CHARM-L does outperform the QuICL Oid-Full and QuICL Oid-Trie algorithms at relatively high supports on the T10I4D100k, T25I10D10k, and, T25I20D100k data sets, the gain is limited to a few seconds (e.g., 2.53 seconds in T10I4D100k at $0.5\%_{supp}$ and 1.05 seconds in T25I10D10k at $1.0\%_{supp}$), although the T25I20D100k exhibits a gain of 57.8 seconds at $1.0\%_{supp}$. In all cases, the gain quickly turns into a large loss as the support is lowered (e.g., loss of 1,534 seconds in T25I10D10k at $0.0\%_{supp}$ and 1,847 seconds in T10I4D100k at $0.0\%_{supp}$). At low supports QuICL Oid-Full outperforms CHARM-L by an excess of an order of magnitude on the Mushroom, T10I4D100K, T25I10D10k, and T25I20D100k[36] data sets, and a factor greater than five for Chess.

The QuICL Oid-Full and QuICL Oid-Trie algorithms exhibit the near same runtime complexity for all data sets. QuICL Oid-Trie exhibits around 25% loss over

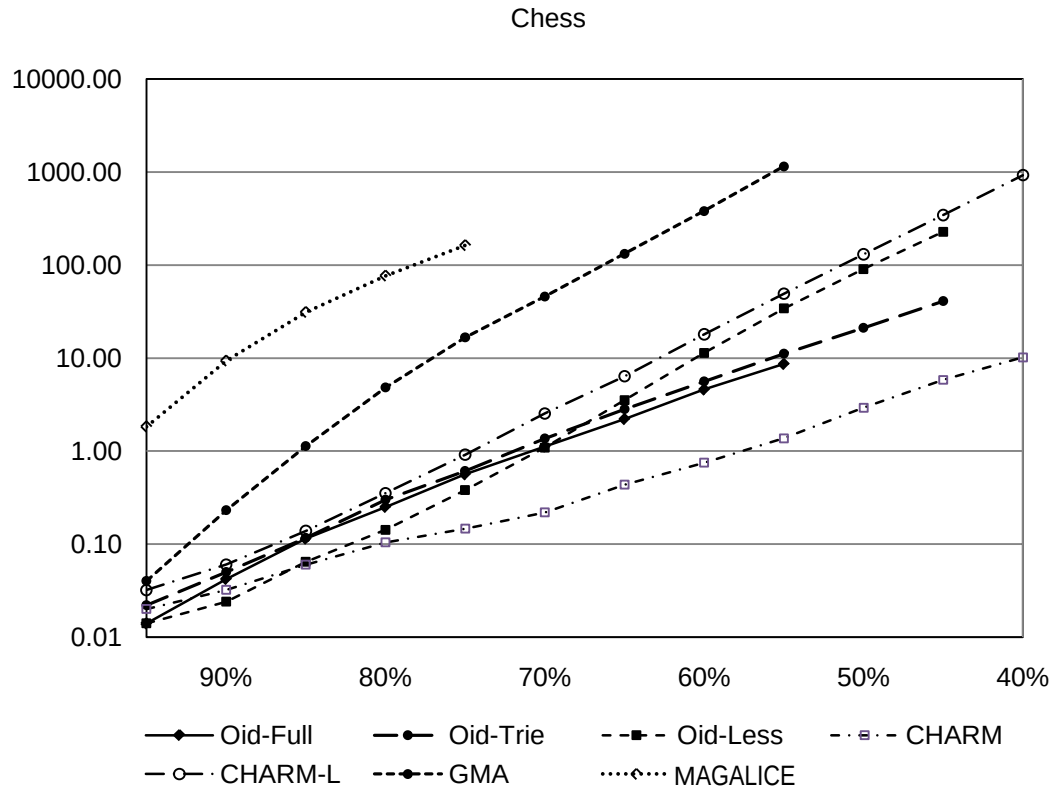
[36] A measurement of an order of magnitude greater was not obtained for the T25I20D100k data set due to the heap size limit on the Java VM. However, at $0.01\%_{supp}$ a value in excess of an order of magnitude can be readily inferred by both the trend in CHARM-L's measurements and by relationship to the CHARM measurements. At $0.01\%_{supp}$, CHARM consumes 1,924.78 seconds. The CHARM-L measurements are in excess $3 \times$ CHARM. Therefore, a value greater than 6,000 seconds is expected for CHARM-L at $0.01\%_{supp}$.

QuICL Oid-Full algorithm, although 45% loss is exhibited for Mushroom at $10\%_{supp}$ and at most a 12% loss for Pumsb* over all supports. This loss in performance is expected. The functions to compare and intersect object id sets at the heart of the QuICL Oid-Trie algorithm will encounter a performance impact, since they must traverse between trie nodes and not just a simple array. Therefore, the cost of advancing the intersection indices within these functions will be more than for QuICL Oid-Full. A loss in performance can also be attributed to cost of adding the object id sets into the trie. To add an object id set involves walking the trie to identify the point where a new trie node[37] will be grafted. Adding object id sets into the Trie may have a minor effect, if any, on the runtime complexity. To counteract the runtime overhead introduced by the trie data structure, the QuICL Oid-Trie algorithm incorporates two performance enhancements. One provides early termination of intersection and compare when the traversals of the two sets encounter the same trie node. The second provides direct lookup of concepts that exist in the lattice, thereby avoiding the intersect and compare functions when searching for such concepts. While these enhancements provide minor gain, they are not sufficient to overcome the increase in intersection cost.

The remaining algorithm, QuICL Oid-Less, provides the best performance of the lattice construction algorithms on the Pumsb data set outperforming CHARM-L by more than a factor of two over all supports. QuICL Oid-Less is QuICL's answer to handling data set containing items with large object id sets. Instead of directly intersecting object ids, QuICL Oid-Less intersects support concepts (i.e., concepts that logically hold object ids in a compressed lattice). For concept lattices where the support concepts hold large

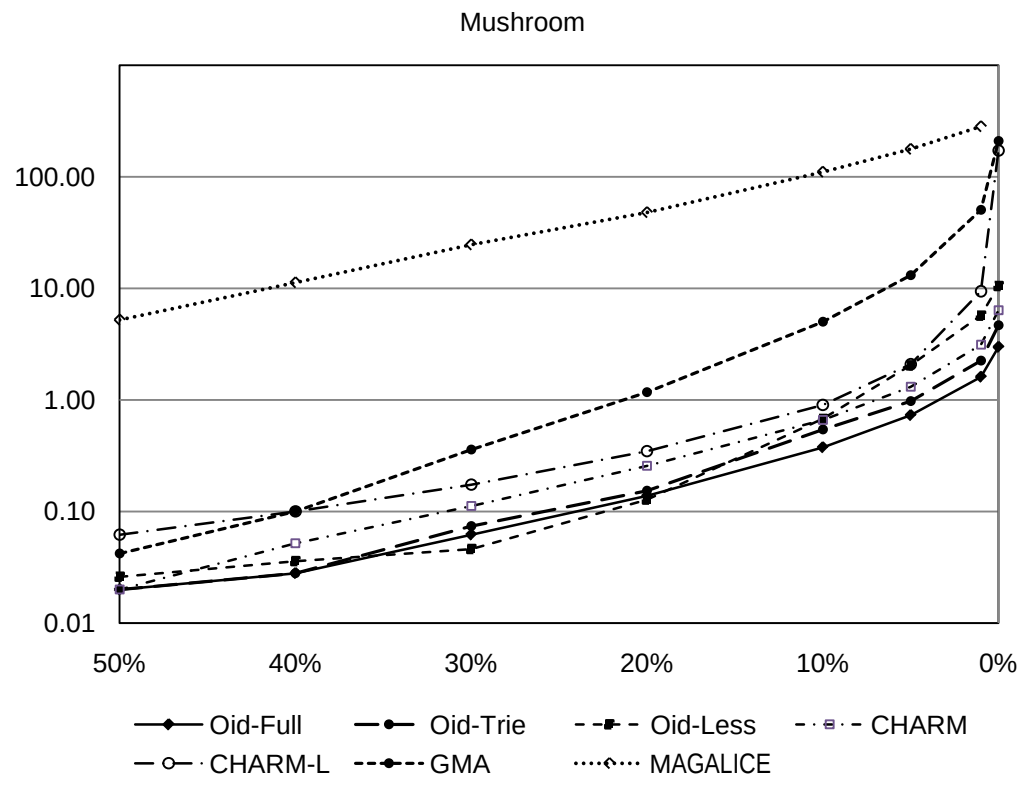
[37] The QuICL Oid-Trie algorithm uses a compound trie node implementation. At most one trie node will be created for each inserted object id set.

sets of object ids, the QuICL Oid-Less algorithm realizes a significant performance gain. Such is the case for the Pumsb lattice. This gain is also realized for an initial period of time during the algorithm execution on other data sets. During this time period, the number of support concepts is limited and the cardinality of object ids they represent is large. This behavior is observed in the Chess, Mushroom, and Pumsb* data sets. Over the course of execution, the support concepts fragment resulting degradation of performance. Indeed, the worst case is the point where all support concepts are iced concepts with each representing a single object id. For sparse data sets, the execution rapidly approaches this worst case at which time QuICL Oid-Less exhibits a performance loss of a factor between two and four when compared to QuICL Oid-Full.



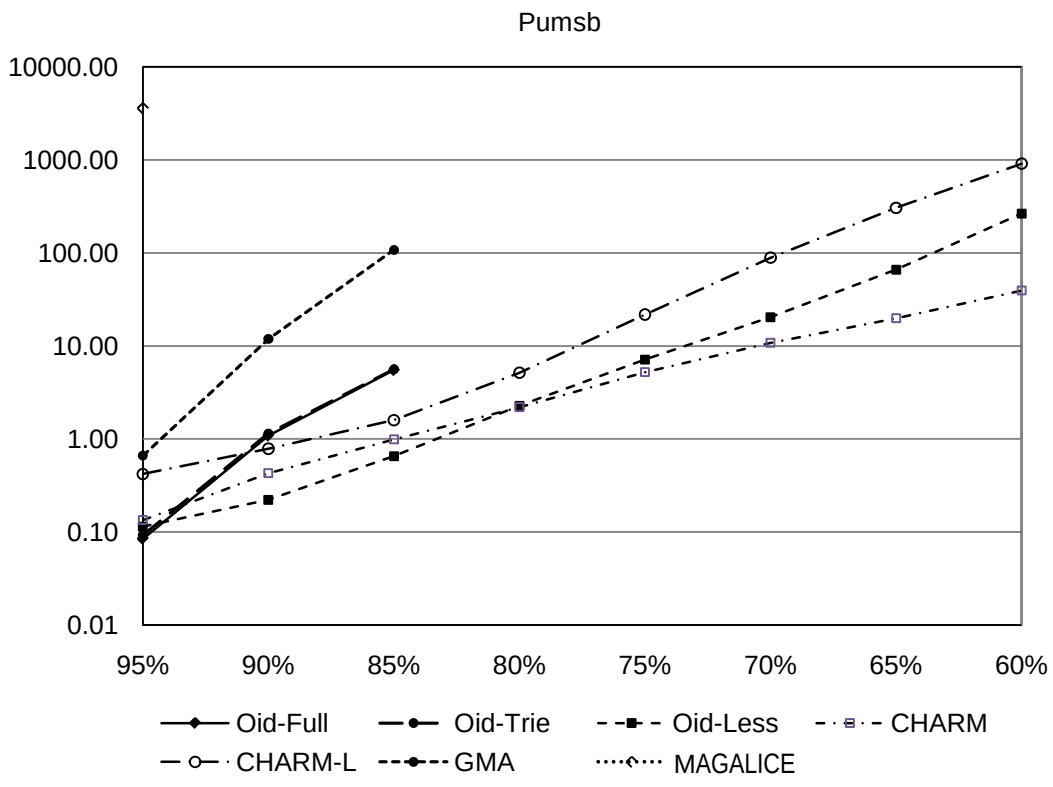
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
95%	74	0.01	0.02	0.01	0.02	0.03	0.04	1.83
90%	503	0.04	0.05	0.02	0.03	0.06	0.23	9.32
85%	1,885	0.11	0.12	0.06	0.06	0.14	1.13	31.02
80%	5,083	0.25	0.30	0.14	0.10	0.35	4.81	76.32
75%	11,525	0.56	0.61	0.38	0.15	0.91	16.62	162.16
70%	23,991	1.11	1.36	1.09	0.22	2.54	45.92	
65%	49,240	2.20	2.83	3.52	0.43	6.36	131.83	
60%	98,392	4.57	5.61	11.29	0.75	17.99	381.31	
55%	192,863	8.60	11.13	34.01	1.37	49.04	1148.32	
50%	369,450		21.12	90.35	2.92	130.30		
45%	707,964		40.89	227.01	5.82	343.79		
40%	1,366,833				10.16	924.48		

Figure 4.11: Comparison of runtime execution time using the Chess data set.



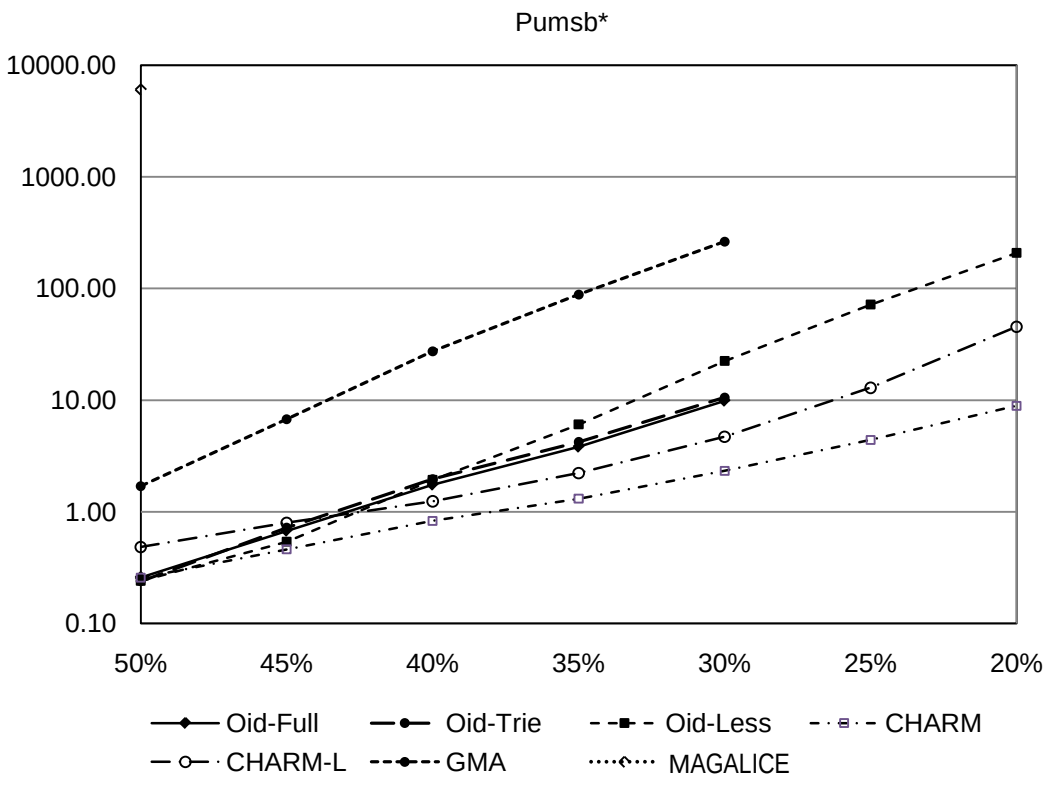
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
50%	45	0.02	0.02	0.03	0.02	0.06	0.04	5.23
40%	140	0.03	0.03	0.04	0.05	0.10	0.10	11.29
30%	427	0.06	0.07	0.05	0.11	0.17	0.36	24.69
20%	1,197	0.14	0.15	0.13	0.26	0.35	1.17	48.11
10%	4,897	0.38	0.54	0.68	0.66	0.90	5.04	110.89
5%	12,854	0.73	0.98	2.05	1.32	2.09	13.13	177.82
1%	51,672	1.61	2.25	5.71	3.12	9.43	50.68	283.66
0%	238,709	2.99	4.67	10.54	6.36	172.46	209.65	

Figure 4.12: Comparison of runtime execution time using the Mushroom data set.



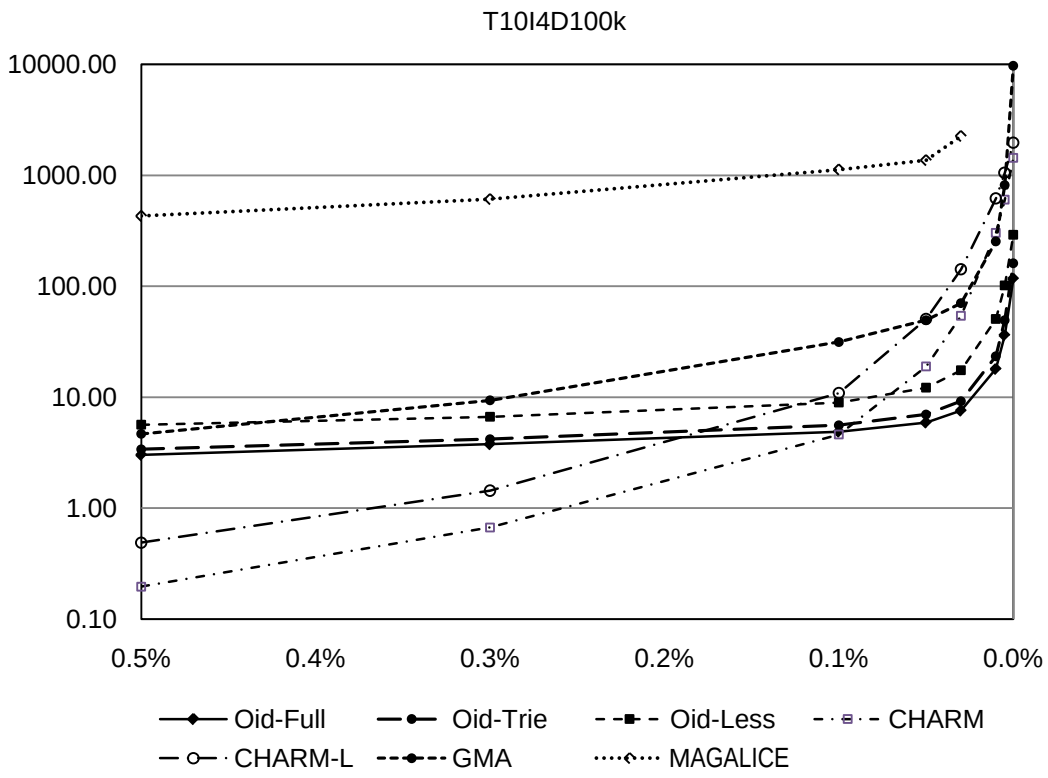
Min Supp	|ℒ|	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
95%	110	0.08	0.09	0.11	0.13	0.42	0.66	3574.71
90%	1,466	1.08	1.14	0.22	0.43	0.79	11.88	
85%	8,513	5.48	5.60	0.65	0.98	1.59	107.61	
80%	33,295			2.27	2.19	5.15		
75%	101,047			7.11	5.23	21.74		
70%	241,258			20.36	10.77	88.84		
65%	496,069			65.90	19.84	304.76		
60%	1,074,627			263.83	39.28	908.79		

Figure 4.13: Comparison of runtime execution time using the Pumsb data set.



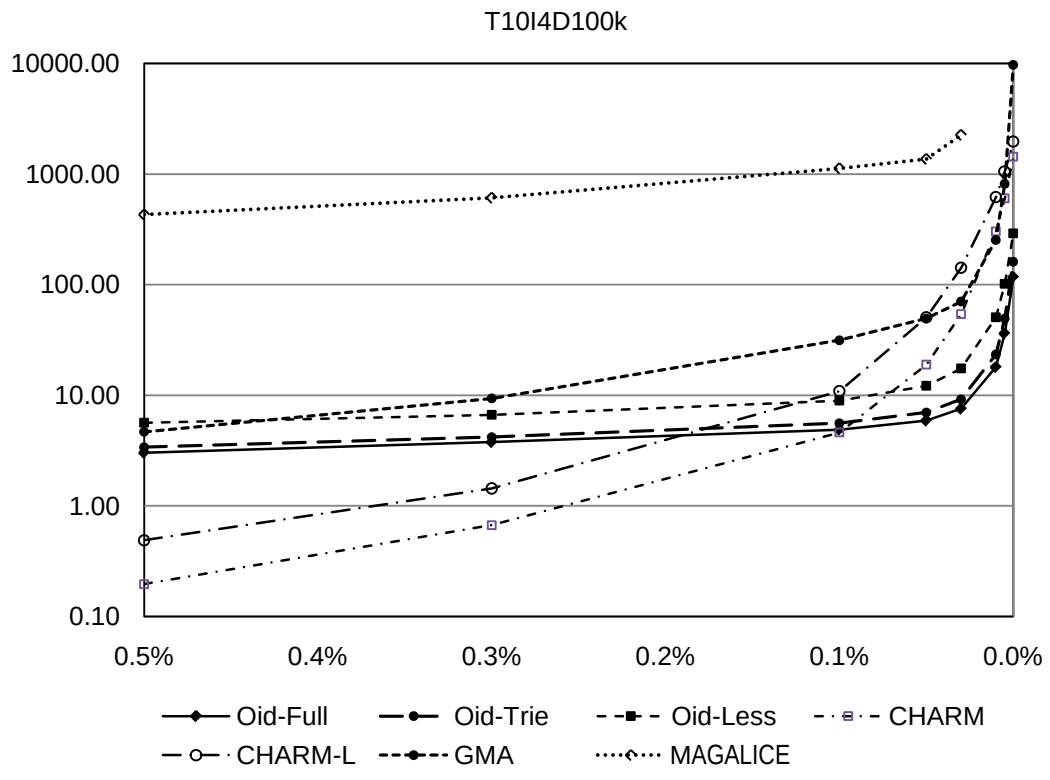
Min Supp	|ℒ|	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
50%	248	0.26	0.24	0.24	0.26	0.48	1.69	6057.85
45%	713	0.67	0.72	0.54	0.46	0.80	6.74	
40%	2,610	1.74	1.96	1.93	0.83	1.24	27.40	
35%	6,133	3.81	4.22	6.04	1.31	2.22	87.94	
30%	16,154	9.85	10.55	22.37	2.33	4.70	262.68	
25%	42,756			71.84	4.41	12.91		
20%	122,262			207.89	8.90	45.40		

Figure 4.14: Comparison of runtime execution time using the Pumsb* data set.



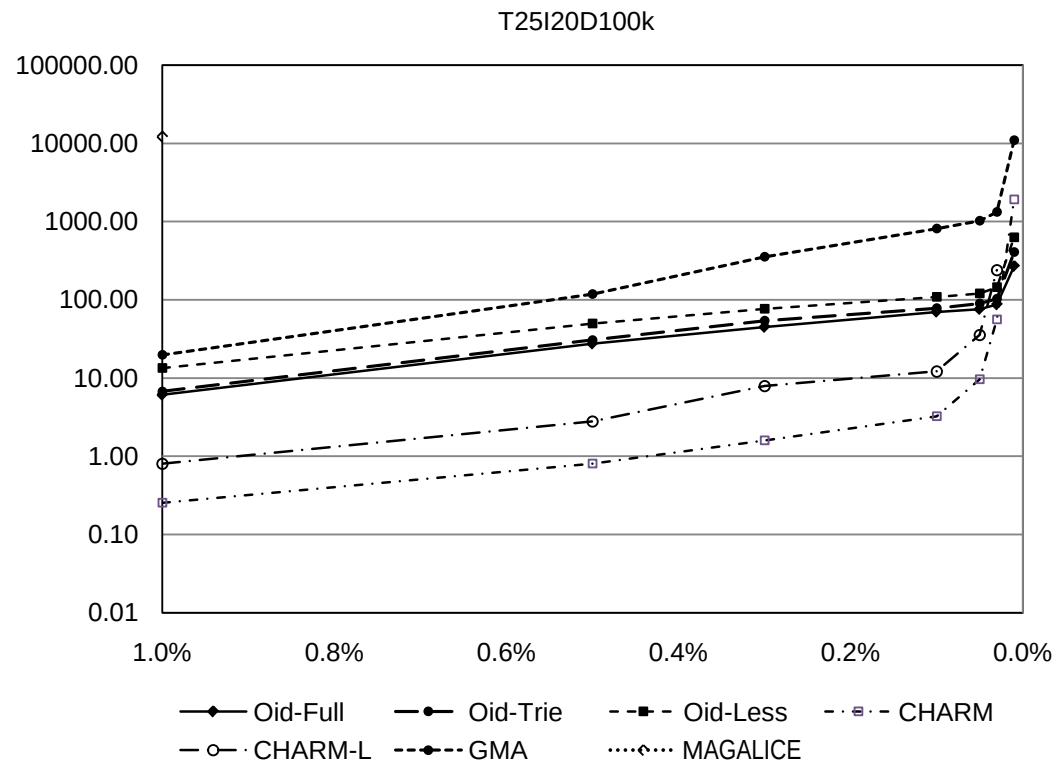
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
0.500%	1,073	3.02	3.39	5.64	0.20	0.49	4.66	430.27
0.300%	4,509	3.76	4.18	6.62	0.67	1.44	9.35	609.43
0.100%	26,806	4.88	5.58	8.95	4.60	10.89	31.42	1124.73
0.050%	46,993	5.90	7.00	12.13	18.87	50.55	49.47	1364.72
0.030%	71,265	7.54	9.23	17.49	54.00	142.11	70.32	2247.53
0.010%	283,397	17.94	23.34	50.45	302.08	618.76	253.56	
0.005%	769,777	36.17	49.35	101.30	602.99	1053.38	815.71	
0.000%	2,347,374	116.84	161.47	290.16	1433.06	1963.54	9717.73	

Figure 4.15: Comparison of runtime execution time using the T10I4D100k data set.



Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
1.00%	5,582	1.49	1.63	2.26	0.19	0.44	2.68	115.02
0.50%	23,394	1.72	2.00	2.94	0.61	1.34	8.62	214.44
0.30%	44,925	2.07	2.43	3.62	1.31	3.41	15.51	302.50
0.10%	209,436	11.29	15.40	29.29	52.89	179.12	60.47	816.84
0.05%	576,021	31.63	45.58	85.79	207.04	528.18	200.41	
0.03%	1,438,054	65.53	95.09	180.36	434.07	916.80	823.92	
0.00%	2,557,928	145.35	195.55	466.42	1018.31	1679.37	5003.55	

Figure 4.16: Comparison of runtime execution time using the T25I10D10k data set.



Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
1.00%	5,256	6.09	6.76	13.39	0.25	0.80	19.74	12117.89
0.50%	27,067	27.41	30.61	49.77	0.81	2.78	117.71	
0.30%	72,640	44.71	53.99	76.74	1.59	7.90	353.11	
0.10%	150,970	69.91	78.25	108.43	3.24	12.11	816.62	
0.05%	212,765	75.74	89.57	119.77	9.63	35.52	1024.13	
0.03%	461,138	86.85	102.92	144.93	56.02	239.18	1321.71	
0.01%	3,519,933	267.86	407.07	626.32	1924.78		10961.24	

Figure 4.17: Comparison of runtime execution time using the T25I20D100k data set.

4.6 Comparison of Algorithm Memory Usage

The memory results of executing the algorithms are given in Figures 4.18 through 4.24 for the Chess, Mushroom, Pumsb, Pumsb*, T10I4D100k, T25I10D10K, and T25I20D100k data sets respectively. 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 the QuICL Oid-Full and QuICL Oid-Trie algorithms for all data sets, and for the GMA and QuICL Oid-Less algorithms on sparse data sets. Descending order is used for GMA and QuICL Oid-Less on dense data sets. All other algorithms are unsorted since either the sort order is an integral part of the algorithm (e.g., CHARM) or is not applicable (e.g., MAGALICE). All measurements are in megabytes (MBs). The plots in each figure display MBs on the vertical axis and minimum support on the horizontal axis.

In addition to the memory results, the algorithms were separately instrumented to report various characteristics on the internal data structures, consisting of;

i) cardinality of object id entries in lattice for the GMA and QuICL Oid-Full algorithms,

ii) cardinality of the object id entries in the trie of the QuICL Oid-Trie algorithm,

iii) cardinality of the item entries in the lattice for the GMA and CHARM-L algorithms, and

iv) the number of parent-child links of the lattice for all lattice construction algorithms.

These characteristics provide insight into the exhibited memory usage. Results from executing the instrumented algorithms against the benchmark data sets over a relevant subset of supports are given in Table 4.3. $|\mathcal{L}|$ is the number of concepts in the resulting lattice. $|\mathcal{O}'|$ is the number of object ids in a full lattice. Such value is reflected in the

QuICL Oid-Full and GMA algorithms. $|\mathcal{O}''|$ is the number of object id entries in the QuICL Oid-Full trie. No object ids are retained in the QuICL Oid-Less or CHARM-L lattices. $|\mathcal{I}'|$ is the number of items in a full lattice. Such value applies to GMA and CHARM-L. $|\mathcal{I}''|$ is the number of items in the QuICL lattices. $|\mathcal{P}|$ is the number of parent-child links. QuICL Oid-Full and QuICL Oid-Trie maintain only one set of parent-child links to traverse from children to parents. The other algorithms maintain two sets of parent-child links to enable bi-directional traversals required of the respective algorithm.

From Table 4.3 rough approximations for memory to represent the internal lattice can be calculated using factors;

i) 4 bytes for each object id in all lattices or item in QuICL lattice,

ii) 6 bytes for each item in GMA or CHARM-L lattice,

iii) 6 bytes for each parent-child link in QuICL Oid-Full or QuICL Oid-Trie lattice,

iv) 12 bytes for each parent-child link in all other lattices,

v) 124 bytes for each concept in QuICL Oid-Full lattice,

vi) 216 bytes for each concept in QuICL Oid-Trie lattice,

vii) 320 bytes for each concept in QuICL Oid-Less lattice,

viii) 276 bytes for each concept in GMA lattice, and

ix) 356 bytes for each concept in CHARM-L lattice.

Details of these factors are given in Appendix H. Determination of concept size for MAGALICE was not performed due to exhibited gross memory consumption. Calculation of memory usage for each algorithm by applying these factors to the applicable characteristics of Table 4.3 is given in Appendix I. In addition, Appendix I includes a comparison to the actual observed memory usage. For the QuICL Oid-Full

and QuICL Oid-Trie, and the GMA algorithms, the calculated memory usage is within a reasonable neighborhood of the observed memory usage. As expected, the observed memory usage is greater to account for the Java VM, loaded class files, execution stack, and data structures beyond the lattice. For the QuICL Oid-Less algorithm, the observed memory usage greatly exceeds that required by the lattice. This excess is a result of the temporary sets constructed and discarded during each item insertion. Similarly, for the CHARM-L algorithm the observed memory usage greatly exceeds that of the lattice. In this case, the excess is for object ids stored in its itemset-oidset tree during processing. Furthermore, the CHARM-L calculations may not be account for all the factors contributing to the size of the lattice. A factor that is omitted is a list of generator concepts associated with each concept.

The MAGALICE exhibits the worst memory usage. Except for a few cases, the memory usage is well in excess of an order of magnitude for all data sets and supports. The reason for this excess was not investigated.

QuICL Oid-Full and GMA exhibit similar memory usage for small lattices (i.e., less than 200,000 concepts) and diverge for larger lattices (i.e., greater than 500,000). For both of these algorithms, 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 Oid-Full) to 72% (e.g., T25I10D10K at 0.0% using QuICL Oid-Full) of the memory consumed. Furthermore, for large lattices the number of parent-

child links account for another 15%. Since the overhead of a concept in the GMA lattice is greater than in the QuICL Oid-Full lattice and the GMA lattice uses two references for each parent-child link, the GMA lattice consumes more memory for large lattices. As the lattice becomes very large GMA exceeds QuICL Oid-Full by a factor approaching two.

For most data sets, the QuICL Oid-Trie algorithm is realizing significant reduction in the number of object id entries within its trie over the number of object id entries present in QuICL Oid-Full's lattice. In only the Pumsb data set is the reduction less than 1.0%. For the Pumsb*, T10I4D100k, and T25I20D100k the reduction is around 15%. A reduction of 30%, 60%, and 80% is realized for the T25I10D10k, Mushroom, and Chess data sets respectively. For small lattices, the reduction in object id entries translates to a significant reduction in memory usage over the QuICL Oid-Full algorithm. For Pumsb* the reduction in memory usage is around 15% [38]. A 50% and 75% reduction is realized for the Mushroom and Chess data sets, although only a 25% reduction is realized from Mushroom at $0.0\%_{supp}$. At $0.0\%_{supp}$ the lattice is of sufficient size that the additional overhead for concepts in the QuICL Oid-Trie lattice is impacting the gain realized by a reduction in object id entries. For the Chess data set, the reduction in memory usage enables QuICL Oid-Trie to construct lattices for smaller minimum supports. For the T10I4D100k, T25I10D10k, and T25I20D100k data sets any gain in reducing the number of object id entries is nullified or outweighed by the overhead in the concepts. For very large lattices, the overhead results in loss greater than 50% (e.g., T25I10D10k at $0.0\%_{supp}$).

[38] Savings exceeding 15% is observed at the $30\%_{supp}$ and thus exceeds the calculated savings. This anomaly may be attributed to unidentified runtime factors experienced with the QuICL Oid-Full execution.

The QuICL Oid-Less algorithm provides further reduction in memory usage by eliminating the object id entries from permanent storage within the lattice or any auxiliary data structure. Temporary object id entries are constructed and then discarded for each item insertion. QuICL Oid-Less eliminates permanent object ids from the lattice at the expense of further overhead in each concept (e.g., 2.4 times QuICL Oid-Full and 1.3 times QuICL Oid-Trie). The net result is a reduction in memory usage by a factor of two on the Mushroom data set, factor of five on the Chess data set, and an order of magnitude on the Pumsb* data set. These gains are sustained over most supports but diminish slightly for large lattices. On the Chess data set, gains of an order of magnitude are exhibited at higher supports then settles to a factor of five at lower supports. This reduction in memory usage enables QuICL Oid-Less to process smaller supports than QuICL Oid-Full on the Chess and Pumsb* data sets. On the Pumsb data set, the growth in memory usage as the support is lowered is drastically restrained. This enables QuICL Oid-Less to process significantly lower supports (e.g., $60\%_{supp}$ as opposed to $85\%_{supp}$ with QuICL Oid-Full). On the T10I4D100K and T25I10D10K QuICL Oid-Less exhibits a loss between 5% and 60%. For these data sets, the lattice quickly degrades into a worst case where there are many iced concepts each representing a single object id[39]. As a result, the size of the temporary sets effectively matches the size of the object id entries in concepts of the QuICL Oid-Full lattice. Thus for these data sets, QuICL Oid-Less will exhibit the same asymptotic memory complexity as QuICL Oid-Full with around a 50% overhead. For the T25I20D100k, this worst case state is not reached until very low supports. Thus, the memory usage of QuICL Oid-Less for this data set is less than

[39] The number of object ids is limited to the formal context. This number should not be confused with the number of object id entries in a lattice. The number of object id entries can exponentially exceed the number of object ids in the formal context.

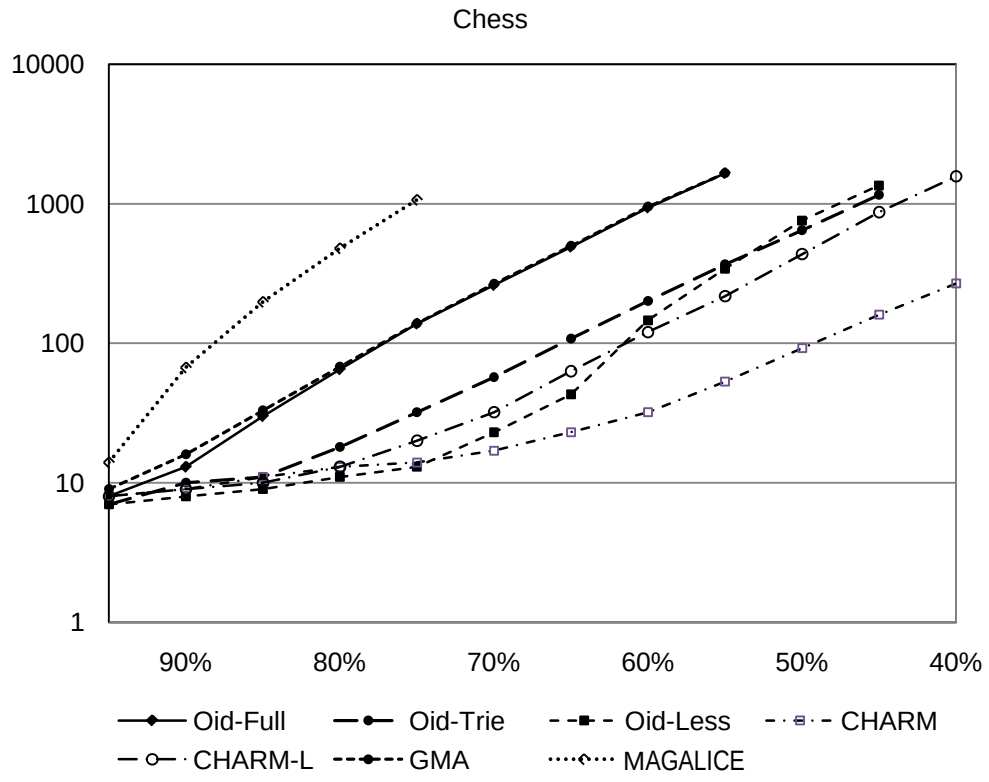
QuICL Oid-Full for most supports. QuICL Oid-Less does, however, exhibit somewhat erratic growth as the support is lowered. This is a result of inserting the items in ascending support order. Such order exacerbates the size of the temporary sets resulting from intersection. The reclamation of space consumed by these sets is attributing to this behavior.

Like QuICL Oid-Less, the CHARM-L algorithm provides a reduction in memory by eliminating the object id entries from the lattice. CHARM-L does maintain object id entries in its itemset-oidset tree. These entries are dynamically constructed during the traversal of the itemset-oidset 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 Oid-Full. It exceeds the memory consumed by the concepts of QuICL Oid-Less. Due to very different approaches, the reduction or gain in memory usage when compared against QuICL algorithms is varied. CHARM-L exhibits the best asymptotic memory complexity on the Pumsb* data sets resulting in a reduction near a factor of five over QuICL Oid-Less. On Pumsb the complexity is slightly better than QuICL Oid-Less. However, for most supports QuICL Oid-Less provides greater reduction. Only at $60\%_{supp}$ does CHARM-L consume less memory. CHARM-L has the same memory complexity as QuICL Oid-Full and QuICL Oid-Trie on the Chess, T10I4D100k, and T25I10D100k data sets. It provides an 85% gain over QuICL Oid-Full (40% gain over QuICL Oid-Trie) on the Chess data set, but a loss around a factor of three on the T10I4D100k and T25I10D100k data sets. The

memory consumption of CHARM-L somewhat matches the memory consumption of QuICL Oid-Less on the Mushroom and T25I20D100k data sets[40].

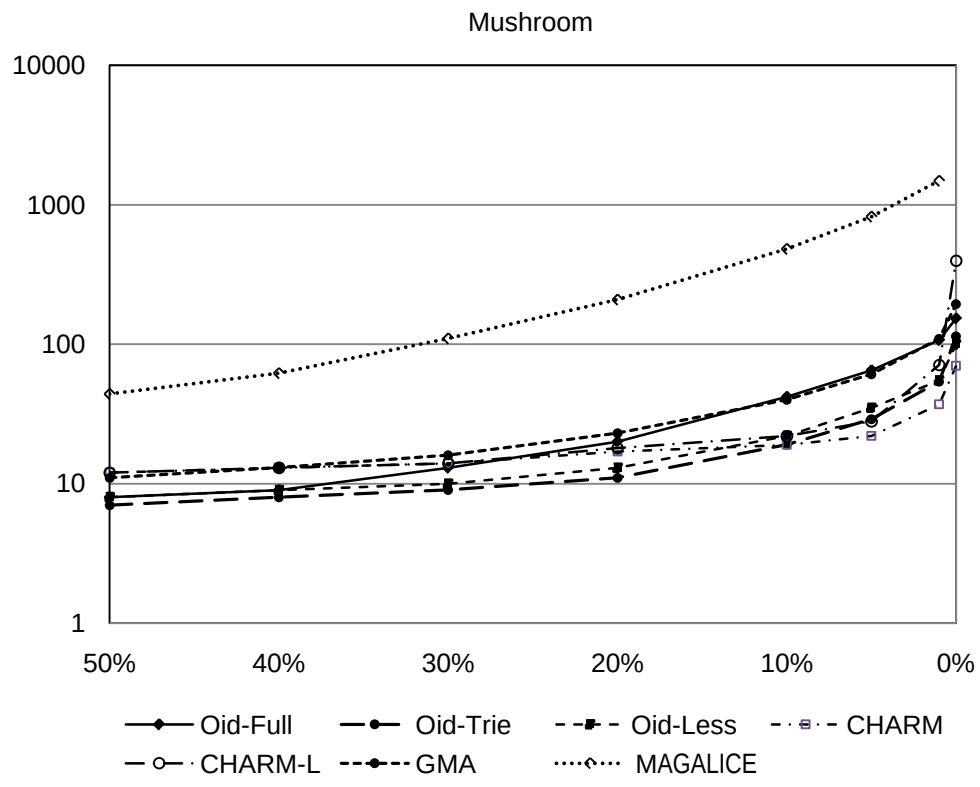
The CHARM algorithm does not construct a lattice. As such, its memory consumption is for processing its itemset-oidset tree and construction its list of frequent item sets. Since CHARM-L is an extension to CHARM that constructs a concept lattice the memory consumption of CHARM is expected to be less 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 itemset-oidset tree is released upon processing a branch and may be reused for the lattice.

[40] Except for $0.0\%_{supp}$ on Mushroom.



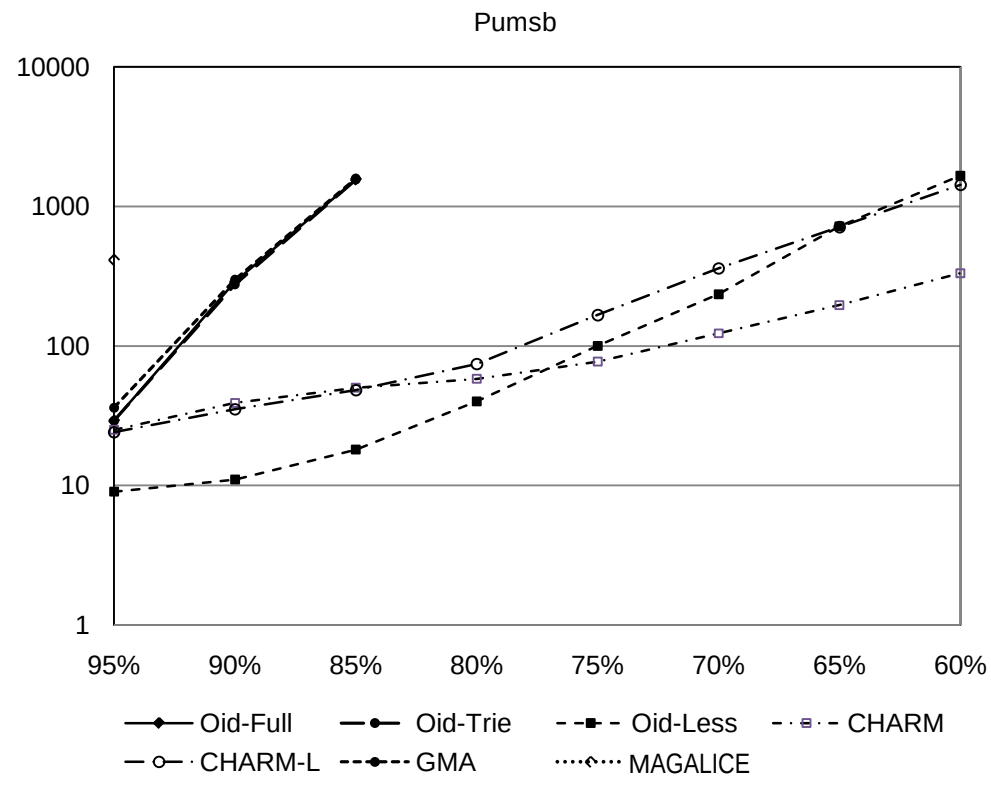
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
95%	74	8	7	7	8	8	9	14
90%	503	13	10	8	9	9	16	67
85%	1,885	30	11	9	11	10	33	198
80%	5,083	65	18	11	13	13	68	479
75%	11,525	138	32	13	14	20	139	1066
70%	23,991	261	57	23	17	32	267	
65%	49,240	492	108	43	23	63	500	
60%	98,392	939	201	146	32	120	951	
55%	192,863	1656	367	343	53	217	1656	
50%	369,450		646	760	92	435		
45%	707,964		1156	1353	160	868		
40%	1,366,833				268	1568		

Figure 4.18: Comparison of memory usage using the Chess data set.



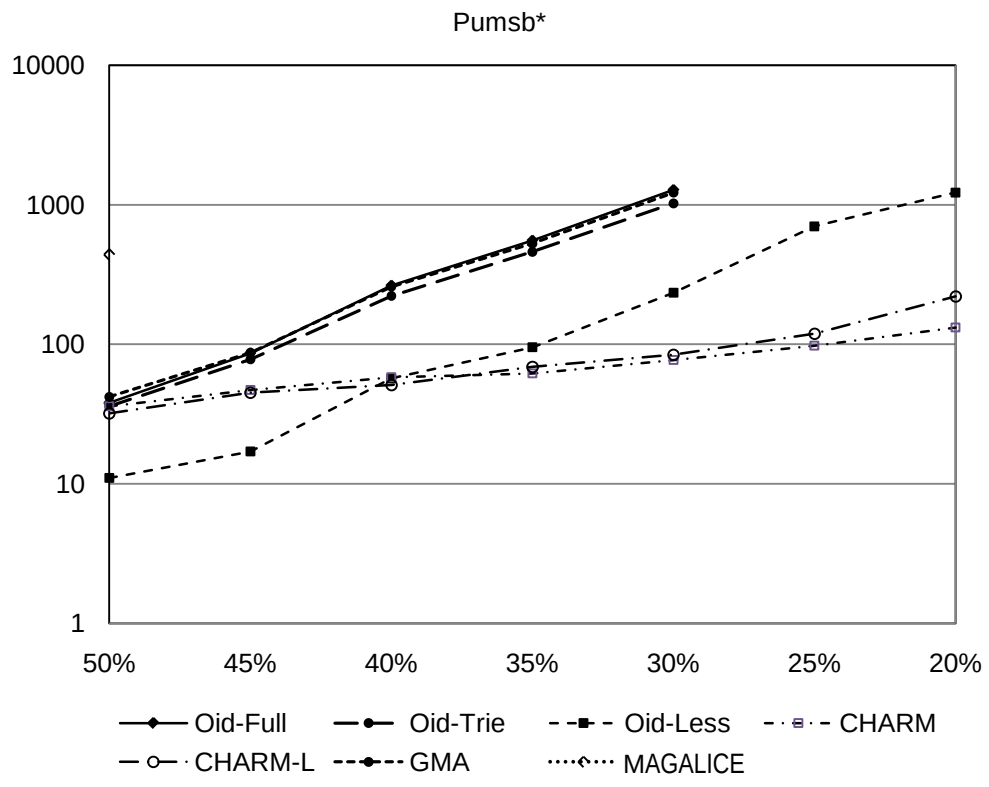
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
50%	45	8	7	8	12	12	11	44
40%	140	9	8	9	13	13	13	62
30%	427	13	9	10	14	14	16	110
20%	1,197	20	11	13	17	18	23	209
10%	4,897	42	19	22	19	22	40	483
5%	12,854	65	29	35	22	28	61	823
1%	51,672	107	54	55	37	71	109	1488
0%	238,709	153	114	100	70	397	194	

Figure 4.19: Comparison of memory usage using the Mushroom data set.



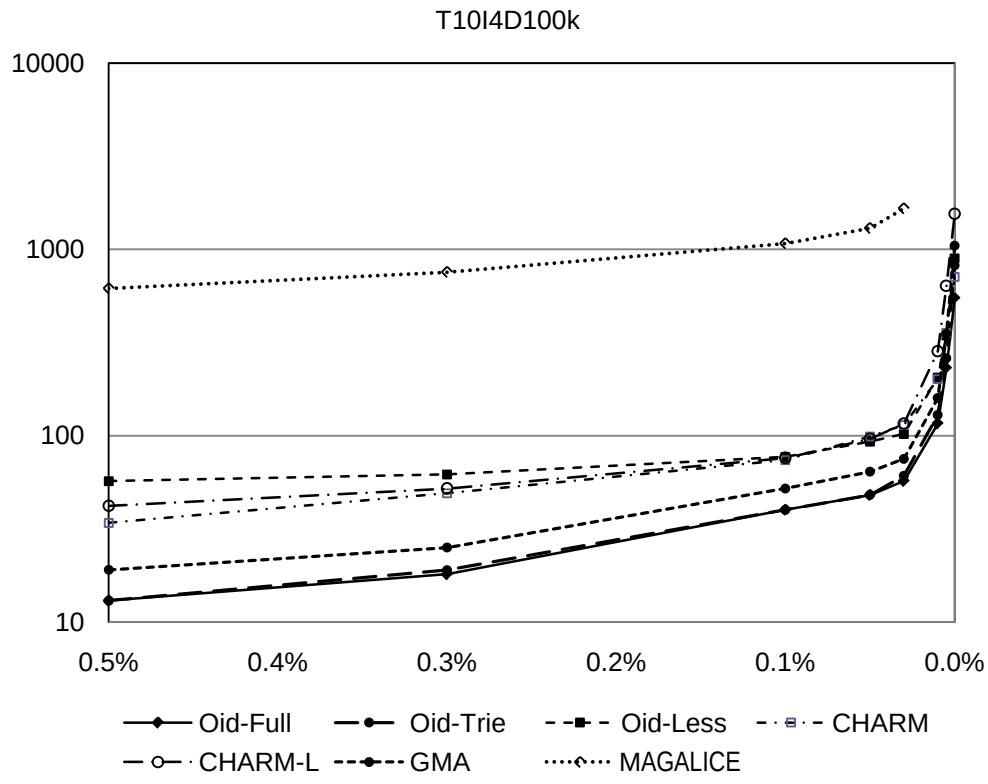
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
95%	110	29	29	9	25	24	36	412
90%	1,466	287	275	11	39	35	298	
85%	8,513	1557	1552	18	50	48	1582	
80%	33,295			40	58	74		
75%	101,047			100	77	166		
70%	241,258			234	123	359		
65%	496,069			717	196	709		
60%	1,074,627			1658	332	1423		

Figure 4.20: Comparison of memory usage using the Pumsb data set.



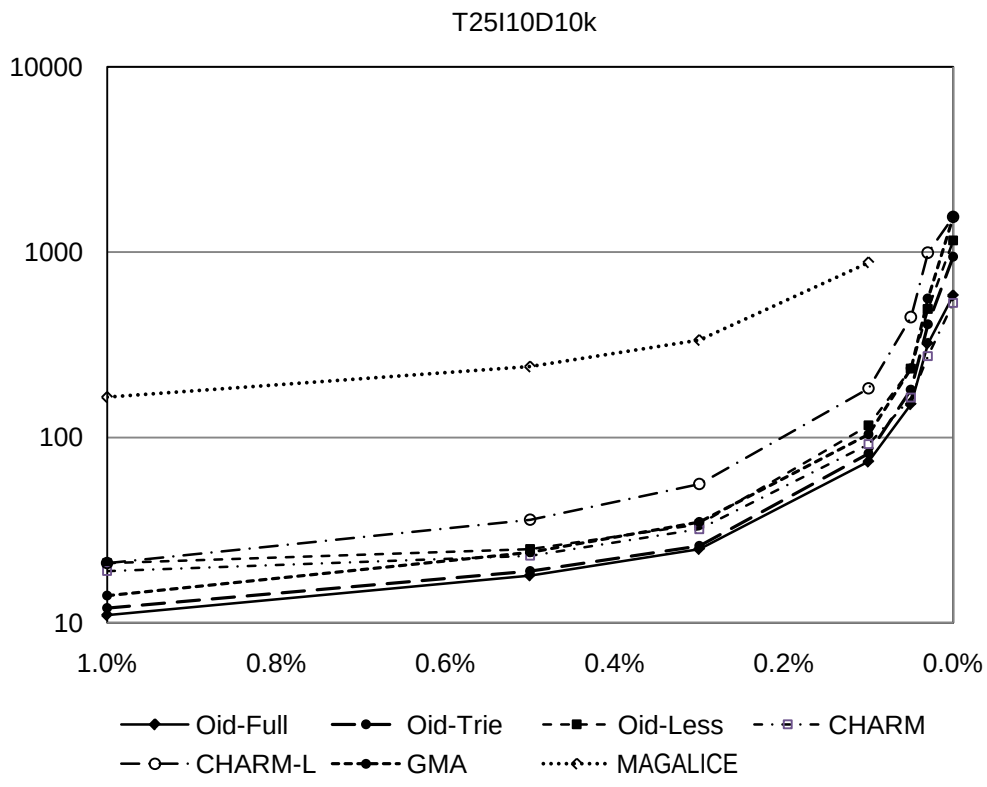
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
50%	248	38	36	11	36	32	42	439
45%	713	86	78	17	47	45	87	
40%	2,610	263	222	57	58	51	259	
35%	6,133	553	460	95	62	69	528	
30%	16,154	1276	1024	234	77	84	1222	
25%	42,756			701	98	119		
20%	122,262			1224	132	220		

Figure 4.21: Comparison of memory usage using the Pumsb* data set.



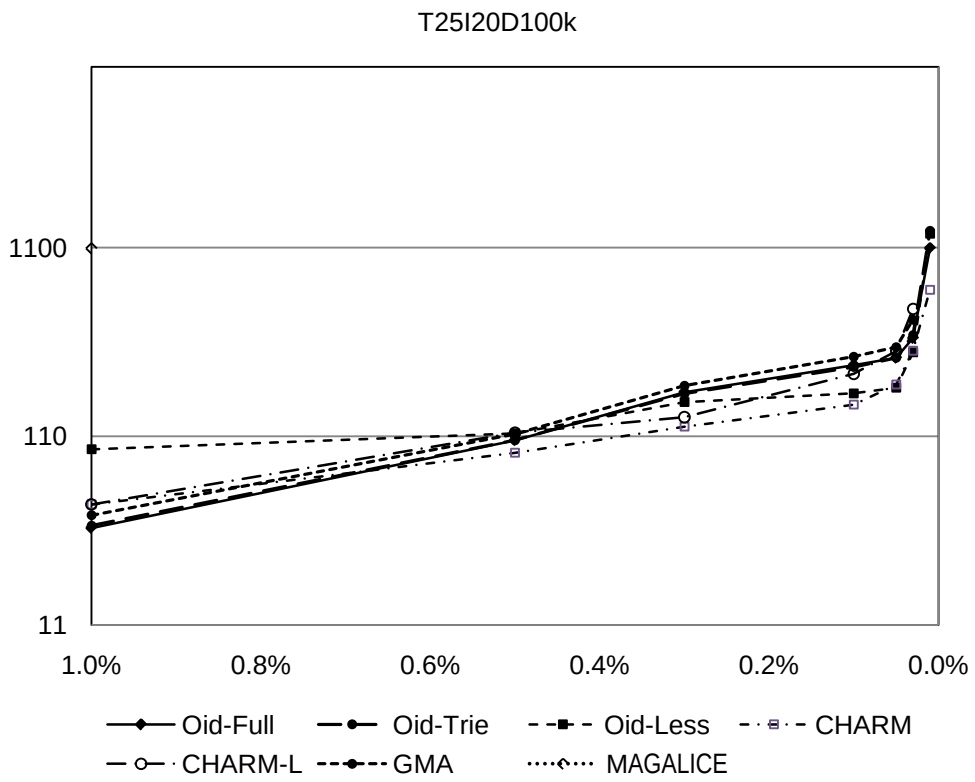
Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
0.500%	1,073	13	13	57	34	42	19	617
0.300%	4,509	18	19	62	49	52	25	755
0.100%	26,806	40	40	77	74	76	52	1076
0.050%	46,993	48	48	93	99	96	64	1300
0.030%	71,265	57	61	102	114	116	75	1658
0.010%	283,397	116	129	205	201	284	159	
0.005%	769,777	230	261	345	355	637	352	
0.000%	2,347,374	546	820	896	711	1552	1048	

Figure 4.22: Comparison of memory usage using the T10I4D100k data set.



Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
1.00%	5,582	11	12	21	19	21	14	165
0.50%	23,394	18	19	25	23	36	24	241
0.30%	44,925	25	26	34	32	56	35	335
0.10%	209,436	74	82	116	92	184	104	879
0.05%	576,021	151	181	235	164	446	236	
0.03%	1,438,054	321	408	493	275	992	561	
0.00%	2,557,928	583	945	1156	533	1548	1554	

Figure 4.23: Comparison of memory usage using the T25I10D10k data set.



Min Supp	$\|\mathcal{L}\|$	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
1.00%	5,256	36	37	94	48	48	42	1090
0.50%	27,067	105	105	114	90	115	113	
0.30%	72,640	189	185	167	124	139	204	
0.10%	150,970	263	256	186	162	236	291	
0.05%	212,765	287	287	199	207	311	327	
0.03%	461,138	365	379	307	313	521	457	
0.01%	3,519,933	1094	1349	1306	658			

Figure 4.24: Comparison of memory usage using the T25I20D100k data set.

	Min Supp	$\|\mathcal{L}\|$	$\|\mathcal{O}'\|$	$\|\mathcal{O}''\|$	$\|\mathcal{I}'\|$	$\|\mathcal{I}''\|$	$\|\mathcal{P}\|$
Chess	95%	74	232,224	50,036	202	9	198
	85%	1,885	5,341,131	933,670	9,018	16	8,302
	75%	11,525	29,631,737	5,099,269	70,729	23	63,270
	65%	49,240	112,611,870	19,857,410	346,926	29	303,912
	55%	192,863	383,151,148	69,458,506	1,535,601	35	1,321,976
Mushroom	50%	45	238,282	106,077	152	13	88
	30%	427	1,384,872	494,819	2,115	28	1,282
	10%	4,897	6,961,076	2,490,957	33,322	56	18,790
	0%	238,709	21,936,050	9,577,434	3,982,989	119	1,362,867
Pumsb	95%	110	5,245,186	5,182,774	310	13	279
	90%	1,466	66,249,882	65,982,078	6,446	20	5,887
	85%	8,513	366,699,694	365,244,637	48,761	24	43,993
	80%	33,295	1,363,773,560		231,559	25	203,885
Pimsb*	50%	248	7,009,886	6,756,195	924	27	703
	40%	2,610	57,137,840	49,131,427	15,422	46	11,028
	30%	16,154	278,455,886	235,356,264	120,185	60	82,959
	20%	122,262	1,490,739,095		1,123,645	86	726,335
T10I4D100k	0.500%	1,073	1,357,945	1,251,699	1,800	569	1,800
	0.100%	26,806	6,638,031	5,705,060	88,164	797	87,564
	0.050%	46,993	8,040,209	6,810,733	149,459	839	145,671
	0.010%	283,397	12,091,976	10,140,956	833,624	867	796,040
	0.005%	769,777	15,204,154	12,220,186	2,491,078	869	2,307,800
	0.000%	2,347,374	19,216,860	14,140,919	9,315,849	870	10,065,478
T25I10D10k	1.00%	5,582	797,573	631,166	20,314	919	19,992
	0.50%	23,394	1,904,796	1,307,129	89,585	982	86,092
	0.10%	209,436	4,882,207	3,504,219	588,817	996	551,827
	0.05%	576,021	7,079,426	4,930,114	1,718,173	996	1,575,898
	0.00%	2,557,928	12,175,844	7,120,716	9,802,820	996	10,992,589
T25I20D100k	1.00%	5,256	6,733,629	6,402,159	18,543	800	18,543
	0.50%	27,067	21,851,821	20,495,570	110,606	2,131	110,606
	0.10%	150,970	53,984,829	47,581,685	701,476	4,329	700,913
	0.05%	212,765	58,082,074	50,199,243	967,539	4,729	958,619
	0.01%	3,519,933	114,598,522	77,861,760		5,072	12,925,176

Table 4.3: Characteristics of internal data structures. $|\mathcal{L}|$ is the number of concepts in the lattice, $|\mathcal{O}'|$ is the number of object id entries in a full lattice, $|\mathcal{O}''|$ is the number of object id entries in the QuICL Oid-Full trie, $|\mathcal{I}'|$ is the number of item entries in a full lattice, $|\mathcal{I}''|$ is the number of item entries in the QuICL lattices, and $|\mathcal{P}|$ is the number of parent-child links.

4.7 Performance Analysis of the QuICL Algorithms

To gain a better understanding of time spent within the QuICL algorithms and to provide empirical evidence for rationale concerning its runtime behavior, the QuICL algorithms was separately instrumented to time key sections of respective INSERT functions (Algorithms 3.4, 3.14, and 3.18). The INSERT function performs the incremental insertion of the next item. For the QuICL Oid-Full algorithm (Algorithm 3.4), instrumentation includes:

i) Insert – total time spent in the INSERT function,

ii) Prep – time to execute the prepare-search phase (lines 10 through 19),

iii) Prep Navigate – time to execute the prepare-search that terminates upon reaching line 15,

iv) Prep Link – time to execute the prepare-search that terminates upon reaching line 13,

v) Prep Create – time to execute the prepare-search that completes without reaching lines 13 or 15,

vi) Purge – time to execute the PURGE-SUBSETS function (line 21),

vii) Superset Link - time to link a new concept to parent superset concepts (lines 25 through 31 processing SUPERSET tuples), and

viii) Intersect Link - time to link a new concept to parent intersect concepts (lines 25 through 31 processing INTERSECT tuples, excluding time consumed in the recursive call to INSERT).

Prep, Purge, Superset Link, and Intersect Link are exclusive of each other and account for the majority of time spent in the INSERT function. Likewise, Prep Navigate, Prep Link, and Prep Create are exclusive of each other and account for the time spent in the prepare-search phase.

Results from executing the instrumented QuICL Oid-Full algorithm against all benchmark data sets for relevant subset of supports are given in Table 4.4 and 4.5.

Ascending support order is used for all test data sets. Insert is the total seconds spent in the INSERT function. Prep, Purge, Superset Link, and Intersect Link are times spent executing the respective sections of the INSERT function. Other accounts for the remaining time spent in the INSERT function. Find Link is derived by summing half of the Prep Navigate, all of Prep Link, and half of Intersect Link. This value represents, to some degree, the time to find and link parent concepts already present in the lattice. Prep, Prep Navigate, Prep Link, Prep Create, Purge, Superset Link, Intersect Link, Other, and Find Link are given as a percentage of Insert time with actual seconds subscripted.

For QuICL Oid-Full algorithm, the majority of execution time is spent in the prepare-search phase, in most cases over 75%. The proportion of time does decrease as the support is lowered and the size of the lattice increases. Most executions of the prepare-search phase complete with the creation of a new concept, since Prep Create time accounts for near all Prep time. Thus, time to execute the prepare-search phase is the dominant part of the overall execution time. However, the time to purge subset entries is also significant and its proportion of time generally increases with the size of the lattice. For example, on T25I10D10k the Purge time grows from 2% at $0.3\%_{supp}$ to 32% at $0.0\%_{supp}$, whereas on Mushroom the Purge time fluctuates around 21% over the spectrum of supports. There are no cases where Purge time exceeds Prep time. This seems to indicate that the time to purge subset entries will not be dominant. However, the time to purge subset entries is a significant amount of the overall time that cannot be discounted. The Prep Navigate time is negligible reaching between 1% and 3% in a few cases (e.g., 1% in T10I4D100k at $0.005\%_{supp}$, 3% in T20I10D10k at $0.0\%_{supp}$). This indicates that the time to navigate into the lattice is an insignificant term in runtime complexity. Some

portion of the Prep Navigate time, Intersect Link time, combined with all of the Prep Link time will be representative of the time to find and link parents already present in the lattice. The Find Link time, is to some degree, a representation of this value. In most cases, the Find Link time is just a few percent. However, it can exceed 10% of the overall execution time on large lattices. For example, on both T10I4D100k and T25I10D10k the Find Link time is 11% at $0.0\%_{supp}$. On Mushroom the Find Link is 11% at $1.0\%_{supp}$ and 21% at $0.0\%_{supp}$. This indicates that the time to process INTERSECT entries that find and link existing concepts can become a significant portion of time. Like time to purge subsets, it cannot be discounted. The remaining time, expressed by Other, is generally a few percent[41]. This time accounts for creating a concept, sorting parents of a new concept, linking a concept to its base concept, and other minor terms. These are negligible with respect to runtime complexity.

The QuICL Oid-Trie algorithm was instrumented in the same manner as QuICL Oid-Full except the time for Prep Link is now for the time to look up a concept given a trie position (lines 1 through 4 of Algorithm 3.18). Results from executing the instrumented QuICL Oid-Trie algorithm against all benchmark data sets for a relevant subset of supports are given in Tables 4.6 and 4.7. Ascending support order is used for all data sets. Like Tables 4.4 and 4.5, Insert is the total number of seconds spent in the QuICL Oid-Trie's INSERT function and Prep, Prep Navigate, Prep Link, Prep Create, Purge, Superset Link, Intersect Link, Other, and Find Link are given as a percentage of Insert time with actual seconds subscripted.

The portions of time spent executing the sections of the QuICL Oid-Trie INSERT function is comparable to the portions reported for QuICL Oid-Full's INSERT function

[41] Chess at $85\%_{supp}$ is an exception.

with a few changes. The actual seconds of QuICL Oid-Trie's Prep Create, and therefore Prep, is slightly less than QuICL Oid-Full on the Chess, Pumsb, and Pumsb* data sets but more on Mushroom, T100I4D100k, T25I10D10k, and T25I20D100k data sets. At low supports on the latter data sets, QuICL Oid-Trie's Prep Create takes twice as much as QuICL Oid-Full. The Prep Create is slower since QuICL Oid-Trie's intersect function must traverse between trie nodes. However, an enhancement incorporated into QuICL Oid-Trie is providing a gain on Chess, Pumsb, and Pumsb* data sets. The enhancement halts intersection processing when the traversals of the object id sets are within the same trie node. Thus for Chess, Pumsb, and Pumsb*, the underlying trie has a sufficient number of object id sets sharing common prefixes to realize a gain. The actual seconds for Purge time is slower on QuICL Oid-Trie[42]. Since comparisons start at the leaves in the trie and terminate as soon as an object id is found in each set that is not in the other set, the common trie nodes are rarely reached. Thus, the additional cost of traverse between tree nodes, although limited, is resulting in slower purge times. Times for Superset Link and Intersect Link are basically the same between QuICL Oid-Trie and QuICL Oid-Full. For QuICL Oid-Trie, the Prep Navigate time is very small resulting in 0% being reported in all cases. This is a result of the direct lookup of existing concepts by oid trie position. The time now remaining in Prep Navigate is reflective of the actual time to navigate to the concept above which a new concept will be created. Clearly, this time is negligible. Furthermore, the lookup is providing an 80% reduction in the Prep Find time. Thus, Find Link time is limited to less than 5% in all test cases. Therefore for QuICL Oid-Trie, the time to find and link existing concepts is not a dominant term of the runtime complexity. The gains achieved in some sections of the QuICL Oid-Trie

[42] Pumsb* at $30\%_{supp}$ is an exception.

algorithm are, however, not sufficient to offset the overhead incurred in traversing the trie nodes.

The QuICL Oid-Less algorithm was instrumented in the same manner as QuICL Oid-Full with the additional instrumentation:

i) Iceberg – time to process iced concepts (lines 34 and 35 of Algorithm 3.14).

Results from executing the instrumented QuICL Oid-Less algorithm against all benchmark data sets for a relevant subset of supports are given in Tables 4.8 and 4.9. Descending support order is used for all data sets to provide consistent analysis. As shown in Table 4.4 and 4.5, Insert is the total number of seconds spent in the QuICL Oid-Less's INSERT function and Prep, Prep Navigate, Prep Link, Prep Create, Purge, Superset Link, Intersect Link, Iceberg, Other, and Find Link are given as a percentage of Insert time with actual seconds subscripted.

The portions of time spent in the sections of the QuICL Oid-Less INSERT function is different to the portions reported in QuICL Oid-Full. The portion of time spent in Prep increases as the support is lowered. For all data sets, the portion of time exceeds 74% at low supports. The actual seconds spent in QuICL Oid-Less Prep is less than QuICL Oid-Full at high supports, but is generally more at low supports. For example, QuICL Oid-Less Prep time is 11.95 seconds on T25I20D100k at $0.1\%_{supp}$ where QuICL Oid-Full is 66.04 seconds, but at $0.01\%_{supp}$ QuICL Oid-Less is 447.88 seconds where QuICL Oid-Full is 180.99 seconds. At high supports, QuICL Oid-Less is benefiting from intersecting sets of support concepts rather than sets of object ids, but as the support is lowered the support concepts fragment result in greater runtime consumption. For Pumsb, T25I20D100k, and to some degree Chess, the point at which to gain turns into a loss is significantly lower (e.g., T25I20D100K at $0.01\%_{supp}$). These

data sets contain items with large object id sets and thus sustain a benefit for a greater spectrum of supports. The actual seconds for QuICL Oid-Less' Purge is slightly less than QuICL Oid-Full on Mushroom, Chess, Pumsb, and Pumsb* data sets, and slightly more on T10I4D100k, T25I10D10k, and T25I20D100K data sets. The portion of time spent in Purge is less due to additional time consumed elsewhere. An additional time component incurred by QuICL Oid-Less is the time to process iced concepts. The Iceberg time accounts for around 30% at high supports but drops as the support is lowered. At $0.0\%_{supp}$ the Iceberg time is zero since there are no iced concepts. The actual seconds for Superset Link time is about the same as QuICL Oid-Full. The actual seconds for Intersect Link and Other vary since the QuICL Oid-Less algorithm involves additional adjustments to information retained in the concepts (e.g., dependent and support concept lists, support, etc.). Overall, the QuICL Oid-Less algorithm provides gain at high supports and suffers a loss as the support is lowered. For data sets that contain items having large object sets the gain turns into a loss at significantly lower supports.

	Min Supp	$\|\mathcal{L}\|$	Insert	Prep		Purge		Superset Link		Intersect Link		Other	
Mushroom	10%	4,897	0.46	78%	0.36	17%	0.08	0%	0.00	2%	0.01	2%	0.01
	5%	12,854	0.88	70%	0.62	23%	0.20	0%	0.00	1%	0.01	6%	0.05
	1%	51,672	1.86	69%	1.28	24%	0.45	1%	0.02	3%	0.05	3%	0.06
	0%	238,709	3.30	65%	2.14	20%	0.67	2%	0.07	7%	0.22	6%	0.20
Chess	85%	1,885	0.12	75%	0.09	8%	0.01	0%	0.00	0%	0.00	17%	0.02
	75%	11,525	0.63	83%	0.52	11%	0.07	0%	0.00	2%	0.01	5%	0.03
	65%	49,240	2.50	80%	2.00	14%	0.36	1%	0.02	3%	0.07	2%	0.05
	55%	192,863	9.94	79%	7.87	17%	1.65	0%	0.04	2%	0.15	2%	0.23
Pumsb	90%	1,466	0.96	98%	0.94	1%	0.01	0%	0.00	0%	0.00	1%	0.01
	85%	8,513	5.30	99%	5.23	1%	0.05	0%	0.00	0%	0.00	0%	0.02
Pumsb*	50%	248	0.23	96%	0.22	4%	0.01	0%	0.00	0%	0.00	0%	0.00
	40%	2,610	1.69	89%	1.50	9%	0.16	0%	0.00	0%	0.00	2%	0.03
	30%	16,154	9.53	84%	8.02	15%	1.46	0%	0.01	0%	0.01	0%	0.03
T10I4D100k	0.100%	26,806	4.85	99%	4.78	1%	0.03	0%	0.00	0%	0.01	1%	0.03
	0.050%	46,993	6.00	97%	5.79	2%	0.11	0%	0.01	0%	0.02	1%	0.07
	0.010%	283,397	15.86	85%	13.44	10%	1.62	0%	0.04	1%	0.12	4%	0.64
	0.005%	769,777	29.50	77%	22.75	16%	4.68	1%	0.24	2%	0.48	5%	1.35
	0.000%	2,347,374	95.99	58%	55.72	31%	29.32	2%	1.78	2%	2.18	7%	6.99
T25I10D10k	0.30%	44,925	1.93	95%	1.83	2%	0.04	1%	0.01	2%	0.03	1%	0.02
	0.10%	209,436	8.61	77%	6.66	15%	1.32	0%	0.02	2%	0.14	5%	0.47
	0.05%	576,021	21.53	72%	15.59	19%	4.19	1%	0.22	1%	0.21	6%	1.32
	0.03%	1,438,054	43.22	68%	29.54	21%	9.11	1%	0.61	2%	0.77	7%	3.19
	0.00%	2,557,928	114.58	58%	66.55	32%	37.08	2%	2.31	2%	2.84	5%	5.80
T25I20D100k	0.30%	72,640	44.19	99%	43.94	0%	0.11	0%	0.01	0%	0.04	0%	0.09
	0.10%	150,970	66.56	99%	66.04	0%	0.17	0%	0.02	0%	0.09	0%	0.24
	0.05%	212,765	72.88	99%	72.24	0%	0.32	0%	0.02	0%	0.09	0%	0.21
	0.03%	461,138	83.14	97%	80.49	2%	1.80	0%	0.11	0%	0.29	1%	0.45
	0.01%	3,519,933	226.18	80%	180.99	16%	35.96	1%	1.47	1%	2.16	2%	5.60

Table 4.4: Timings of the main QuICL Oid-Full sections. Insert is the total seconds spent in the INSERT function. Prep, Purge, Superset Link, Intersect Link is the time spent in the respective sections. Other accounts for the remaining time spent in the INSERT function. Prep, Purge, Superset Link, Intersect Link and Other are given as a percentage of Insert time with actual seconds subscripted.

	Min Supp	$\|\mathcal{L}\|$	Insert	Prep		Prep Navigate		Prep Find		Prep Create		Find Link	
Mushroom	10%	4,897	0.46	78%	0.36	0%	0.00	0%	0.00	78%	0.36	2%	0.01
	5%	12,854	0.88	70%	0.62	1%	0.01	2%	0.02	68%	0.60	3%	0.03
	1%	51,672	1.86	69%	1.28	0%	0.00	9%	0.17	59%	1.10	11%	0.20
	0%	238,709	3.30	65%	2.14	1%	0.02	17%	0.57	45%	1.50	21%	0.69
Chess	85%	1,885	0.12	75%	0.09	0%	0.00	0%	0.00	75%	0.09	0%	0.00
	75%	11,525	0.63	83%	0.52	0%	0.00	2%	0.01	83%	0.52	3%	0.02
	65%	49,240	2.50	80%	2.00	0%	0.00	2%	0.04	78%	1.95	3%	0.08
	55%	192,863	9.94	79%	7.87	0%	0.00	3%	0.32	76%	7.51	4%	0.40
Pumsb	90%	1,466	0.96	98%	0.94	0%	0.00	1%	0.01	97%	0.93	1%	0.01
	85%	8,513	5.30	99%	5.23	0%	0.00	0%	0.02	98%	5.22	0%	0.02
Pumsb*	50%	248	0.23	96%	0.22	0%	0.00	0%	0.00	96%	0.22	0%	0.00
	40%	2,610	1.69	89%	1.50	0%	0.00	0%	0.00	89%	1.50	0%	0.00
	30%	16,154	9.53	84%	8.02	0%	0.00	0%	0.04	84%	7.98	1%	0.05
T10I4D100k	0.100%	26,806	4.85	99%	4.78	0%	0.00	0%	0.01	98%	4.77	0%	0.02
	0.050%	46,993	6.00	97%	5.79	0%	0.00	1%	0.04	96%	5.75	1%	0.05
	0.010%	283,397	15.86	85%	13.44	0%	0.01	2%	0.29	83%	13.13	2%	0.36
	0.005%	769,777	29.50	77%	22.75	1%	0.32	4%	1.29	71%	21.08	6%	1.69
	0.000%	2,347,374	95.99	58%	55.72	2%	2.05	9%	8.28	47%	45.12	11%	10.40
T25I10D10k	0.30%	44,925	1.93	95%	1.83	0%	0.00	2%	0.04	92%	1.78	3%	0.06
	0.10%	209,436	8.61	77%	6.66	0%	0.00	1%	0.09	76%	6.56	2%	0.16
	0.05%	576,021	21.53	72%	15.59	1%	0.12	3%	0.74	68%	14.69	4%	0.91
	0.03%	1,438,054	43.22	68%	29.54	1%	0.49	5%	2.18	62%	26.75	7%	2.81
	0.00%	2,557,928	114.58	58%	66.55	3%	3.52	8%	9.69	46%	53.01	11%	12.87
T25I20D100k	0.30%	72,640	44.19	99%	43.94	0%	0.00	1%	0.23	99%	43.70	1%	0.25
	0.10%	150,970	66.56	99%	66.04	0%	0.00	0%	0.30	99%	65.72	1%	0.35
	0.05%	212,765	72.88	99%	72.24	0%	0.00	1%	0.40	99%	71.82	1%	0.45
	0.03%	461,138	83.14	97%	80.49	0%	0.02	1%	0.81	96%	79.61	1%	0.97
	0.01%	3,519,933	226.18	80%	180.99	0%	1.03	2%	5.24	77%	174.37	3%	6.84

Table 4.5: Additional timings of QuICL Oid-Full sections. Insert is the total seconds spent in the INSERT function. Prep, Prep Navigate, Prep Link, Prep Create, and Find Link are given as a percentage of Insert time with actual seconds subscripted.

	Min Supp	$\|\mathcal{L}\|$	Insert	Prep		Purge		Superset Link		Intersect Link		Other	
Mushroom	10%	4,897	0.44	70%	0.31	23%	0.10	2%	0.01	0%	0.00	5%	0.02
	5%	12,854	0.87	70%	0.61	24%	0.21	0%	0.00	2%	0.02	3%	0.03
	1%	51,672	2.06	65%	1.33	28%	0.57	1%	0.02	3%	0.06	4%	0.08
	0%	238,709	4.51	64%	2.90	20%	0.88	2%	0.08	3%	0.14	11%	0.51
Chess	85%	1,885	0.10	80%	0.08	10%	0.01	0%	0.00	0%	0.00	10%	0.01
	75%	11,525	0.58	74%	0.43	17%	0.10	0%	0.00	0%	0.00	9%	0.05
	65%	49,240	2.58	77%	1.98	18%	0.47	0%	0.01	2%	0.04	3%	0.08
	55%	192,863	10.25	70%	7.19	24%	2.43	0%	0.05	2%	0.16	4%	0.42
	45%	707,964	38.75	65%	25.11	27%	10.34	1%	0.24	2%	0.60	6%	2.46
Pumsb	90%	1,466	0.89	98%	0.87	2%	0.02	0%	0.00	0%	0.00	0%	0.00
	85%	8,513	4.97	98%	4.86	2%	0.09	0%	0.01	0%	0.01	0%	0.00
Pumsb*	50%	248	0.22	86%	0.19	9%	0.02	0%	0.00	0%	0.00	5%	0.01
	40%	2,610	1.72	83%	1.43	16%	0.28	0%	0.00	0%	0.00	1%	0.01
	30%	16,154	3.71	80%	2.98	19%	0.70	0%	0.01	0%	0.01	0%	0.01
T10I4D100k	0.100%	26,806	5.06	98%	4.97	1%	0.03	0%	0.00	0%	0.01	1%	0.05
	0.050%	46,993	6.23	96%	5.98	2%	0.13	0%	0.01	0%	0.02	1%	0.09
	0.010%	283,397	20.87	85%	17.66	9%	1.97	0%	0.04	2%	0.40	4%	0.80
	0.005%	769,777	44.72	80%	35.80	13%	5.89	1%	0.25	1%	0.35	5%	2.43
	0.000%	2,347,374	150.97	68%	103.35	24%	36.72	1%	1.73	1%	2.13	5%	7.04
T25I10D10k	0.30%	44,925	2.22	93%	2.07	2%	0.04	1%	0.02	1%	0.02	3%	0.07
	0.10%	209,436	13.54	82%	11.13	13%	1.70	0%	0.06	1%	0.07	4%	0.58
	0.05%	576,021	39.01	80%	31.28	13%	5.11	1%	0.23	1%	0.20	6%	2.19
	0.03%	1,438,054	87.50	78%	68.61	14%	12.26	1%	0.62	1%	0.70	6%	5.31
	0.00%	2,557,928	181.21	66%	120.16	25%	45.19	1%	2.44	2%	3.06	6%	10.36
T25I20D100k	0.30%	72,640	45.79	99%	45.31	0%	0.15	0%	0.01	0%	0.04	1%	0.28
	0.10%	150,970	69.25	99%	68.51	0%	0.22	0%	0.06	0%	0.12	0%	0.34
	0.05%	212,765	76.34	98%	75.16	1%	0.46	0%	0.24	0%	0.10	0%	0.38
	0.03%	461,138	89.81	96%	85.96	2%	2.22	0%	0.11	0%	0.22	1%	1.30
	0.01%	3,519,933	365.21	83%	304.58	12%	45.52	0%	1.58	1%	2.25	3%	11.28

Table 4.6: Timings of the main QuICL Oid-Trie sections. Insert is the total seconds spent in the INSERT function. Prep, Purge, Superset Link, Intersect Link is the time spent in the respective sections. Other accounts for the remaining time spent in the INSERT function. Prep, Purge, Superset Link, Intersect Link and Other are given as a percentage of Insert time with actual seconds subscripted.

	Min Supp	$\|\mathcal{L}\|$	Insert	Prep	Prep Navigate	Prep Find	Prep Create	Find Link
Mushroom	10%	4,897	0.44	70% 0.31	0% 0.00	0% 0.00	70% 0.31	0% 0.00
	5%	12,854	0.87	70% 0.61	0% 0.00	0% 0.00	70% 0.61	1% 0.01
	1%	51,672	2.06	65% 1.33	0% 0.00	1% 0.02	63% 1.30	2% 0.05
	0%	238,709	4.51	64% 2.90	0% 0.00	3% 0.12	61% 2.76	4% 0.19
Chess	85%	1,885	0.10	80% 0.08	0% 0.00	0% 0.00	80% 0.08	0% 0.00
	75%	11,525	0.58	74% 0.43	0% 0.00	0% 0.00	74% 0.43	0% 0.00
	65%	49,240	2.58	77% 1.98	0% 0.00	2% 0.04	75% 1.93	2% 0.06
	55%	192,863	10.25	70% 7.19	0% 0.00	2% 0.19	68% 6.97	3% 0.27
	45%	707,964	38.75	65% 25.11	0% 0.00	2% 0.93	62% 24.05	3% 1.23
Pumsb	90%	1,466	0.89	98% 0.87	0% 0.00	0% 0.00	97% 0.86	0% 0.00
	85%	8,513	4.97	98% 4.86	0% 0.00	0% 0.01	98% 4.85	0% 0.02
Pumsb*	50%	248	0.22	86% 0.19	0% 0.00	0% 0.00	86% 0.19	0% 0.00
	40%	2,610	1.72	83% 1.43	0% 0.00	0% 0.00	83% 1.43	0% 0.00
	30%	16,154	3.71	80% 2.98	0% 0.00	0% 0.01	80% 2.97	1% 0.02
T10I4D100k	0.100%	26,806	5.06	98% 4.97	0% 0.00	0% 0.00	98% 4.96	0% 0.01
	0.050%	46,993	6.23	96% 5.98	0% 0.00	0% 0.00	96% 5.97	0% 0.01
	0.010%	283,397	20.87	85% 17.66	0% 0.01	0% 0.07	84% 17.56	1% 0.28
	0.005%	769,777	44.72	80% 35.80	0% 0.11	0% 0.19	79% 35.45	1% 0.42
	0.000%	2,347,374	150.97	68% 103.35	0% 0.24	1% 1.81	67% 101.11	2% 3.00
T25I10D10k	0.30%	44,925	2.22	93% 2.07	0% 0.00	0% 0.00	93% 2.07	0% 0.01
	0.10%	209,436	13.54	82% 11.13	0% 0.00	0% 0.02	82% 11.09	0% 0.06
	0.05%	576,021	39.01	80% 31.28	0% 0.03	0% 0.11	80% 31.10	1% 0.23
	0.03%	1,438,054	87.50	78% 68.61	0% 0.31	1% 0.53	77% 67.67	1% 1.04
	0.00%	2,557,928	181.21	66% 120.16	0% 0.48	1% 1.97	65% 117.44	2% 3.74
T25I20D100k	0.30%	72,640	45.79	99% 45.31	0% 0.00	0% 0.02	99% 45.29	0% 0.04
	0.10%	150,970	69.25	99% 68.51	0% 0.00	0% 0.04	99% 68.46	0% 0.10
	0.05%	212,765	76.34	98% 75.16	0% 0.00	0% 0.05	98% 75.08	0% 0.10
	0.03%	461,138	89.81	96% 85.96	0% 0.01	0% 0.13	96% 85.78	0% 0.25
	0.01%	3,519,933	365.21	83% 304.58	0% 0.62	0% 1.28	83% 302.44	1% 2.72

Table 4.7: Additional timings of QuICL Oid-Trie sections. Insert is the total seconds spent in the INSERT function. Prep, Prep Navigate, Prep Link, Prep Create, and Find Link are given as a percentage of Insert time with actual seconds subscripted.

	Min Supp	$\|\mathcal{L}\|$	Insert	Prep	Purge	Iceberg	Superset Link	Intersect Link	Other
Mushroom	10%	4,897	0.60	57% 0.34	3% 0.02	28% 0.17	0% 0.00	0% 0.00	12% 0.07
	5%	12,854	1.95	78% 1.52	2% 0.03	16% 0.31	0% 0.00	1% 0.01	4% 0.08
	1%	51,672	5.61	87% 4.86	2% 0.13	7% 0.38	0% 0.01	1% 0.06	3% 0.17
	0%	238,709	10.70	84% 9.02	3% 0.32	0% 0.05	1% 0.07	2% 0.22	10% 1.02
Chess	85%	74	0.04	25% 0.01	0% 0.00	25% 0.01	0% 0.00	25% 0.01	25% 0.01
	75%	1,885	0.38	63% 0.24	3% 0.01	13% 0.05	0% 0.00	0% 0.00	21% 0.08
	65%	11,525	3.68	81% 2.98	3% 0.11	8% 0.29	0% 0.01	1% 0.04	7% 0.25
	55%	49,240	34.36	87% 29.77	2% 0.82	8% 2.72	0% 0.06	1% 0.22	2% 0.77
	45%	192,863	231.60	89% 205.35	3% 5.90	5% 11.96	0% 0.21	0% 1.14	3% 7.04
Pumsb	90%	1,466	0.13	46% 0.06	0% 0.00	31% 0.04	0% 0.00	8% 0.01	15% 0.02
	80%	33,295	2.36	74% 1.74	3% 0.06	9% 0.21	0% 0.00	2% 0.04	13% 0.31
	70%	241,258	21.65	69% 14.89	3% 0.61	19% 4.01	0% 0.07	2% 0.37	8% 1.70
	60%	1,074,627	271.26	75% 202.63	2% 4.25	16% 42.95	0% 0.50	1% 1.88	7% 19.05
Pumsb*	50%	248	0.12	25% 0.03	0% 0.00	58% 0.07	0% 0.00	0% 0.00	17% 0.02
	40%	2,610	1.80	71% 1.27	4% 0.07	18% 0.33	0% 0.00	0% 0.00	7% 0.13
	30%	16,154	22.41	87% 19.56	3% 0.72	5% 1.16	0% 0.00	0% 0.01	4% 0.96
T10I4D100k	0.100%	26,806	3.48	53% 1.83	1% 0.03	32% 1.11	0% 0.01	1% 0.02	14% 0.48
	0.050%	46,993	6.60	68% 4.48	2% 0.13	23% 1.50	0% 0.02	1% 0.04	7% 0.43
	0.010%	283,397	45.05	86% 38.52	5% 2.20	5% 2.41	0% 0.05	1% 0.23	4% 1.64
	0.005%	769,777	96.73	87% 84.62	6% 6.08	3% 3.38	0% 0.24	1% 0.54	2% 1.87
	0.000%	2,347,374	283.28	85% 239.56	12% 32.82	0% 0.94	1% 1.89	1% 3.62	2% 4.45
T25I10D10k	0.30%	44,925	1.67	62% 1.04	3% 0.05	19% 0.31	1% 0.02	2% 0.04	13% 0.21
	0.10%	209,436	28.35	89% 25.10	6% 1.57	3% 0.76	0% 0.04	0% 0.11	3% 0.77
	0.05%	576,021	87.99	90% 78.98	5% 4.78	1% 1.27	0% 0.12	0% 0.31	3% 2.53
	0.03%	1,438,054	180.65	90% 162.45	6% 10.63	1% 2.25	0% 0.65	1% 0.94	2% 3.73
	0.00%	2,557,928	462.36	88% 408.33	8% 38.81	0% 1.32	1% 2.96	1% 4.29	1% 6.65
T25I20D100k	0.30%	72,640	11.34	64% 7.27	1% 0.12	23% 2.64	0% 0.01	1% 0.07	11% 1.23
	0.10%	150,970	17.59	68% 11.95	1% 0.23	15% 2.68	0% 0.02	1% 0.16	14% 2.55
	0.05%	212,765	21.65	72% 15.52	2% 0.45	15% 3.33	0% 0.05	1% 0.14	10% 2.16
	0.03%	461,138	45.30	77% 34.94	5% 2.36	10% 4.42	0% 0.09	1% 0.43	7% 3.06
	0.01%	3,519,933	526.15	85% 447.88	9% 47.66	2% 12.77	0% 1.27	1% 2.88	3% 13.69

Table 4.8: Timings of the main QuICL Oid-Less sections. Insert is the total seconds spent in the INSERT function. Prep, Purge, Superset Link, Intersect Link is the time spent in the respective sections. Other accounts for the remaining time spent in the INSERT function. Prep, Purge, Superset Link, Intersect Link and Other are given as a percentage of Insert time with actual seconds subscripted.

	Min Supp	$\|\mathcal{L}\|$	Insert	Prep		Prep Navigate		Prep Find		Prep Create		Find Link	
Mushroom	10%	4,897	0.60	57%	0.34	2%	0.01	5%	0.03	50%	0.30	6%	0.04
	5%	12,854	1.95	78%	1.52	7%	0.14	6%	0.11	65%	1.27	9%	0.18
	1%	51,672	5.61	87%	4.86	6%	0.31	7%	0.40	74%	4.14	10%	0.59
	0%	238,709	10.70	84%	9.02	4%	0.44	13%	1.40	67%	7.15	16%	1.73
Chess	85%	74	0.04	26%	0.01	0%	0.00	0%	0.00	26%	0.01	13%	0.01
	75%	1,885	0.38	64%	0.24	0%	0.00	20%	0.08	44%	0.17	20%	0.08
	65%	11,525	3.68	81%	2.98	0%	0.00	11%	0.40	70%	2.57	11%	0.42
	55%	49,240	34.36	87%	29.77	0%	0.00	7%	2.43	79%	27.31	7%	2.54
	45%	192,863	231.60	89%	205.35	0%	0.37	6%	14.84	82%	190.02	7%	15.60
Pumsb	90%	1,466	0.13	45%	0.06	0%	0.00	12%	0.02	32%	0.04	16%	0.02
	80%	33,295	2.36	74%	1.74	0%	0.01	13%	0.31	60%	1.42	14%	0.33
	70%	241,258	21.65	69%	14.89	0%	0.01	13%	2.80	56%	12.03	14%	2.99
	60%	1,074,627	271.26	75%	202.63	0%	0.57	11%	30.83	63%	170.99	12%	32.06
Pumsb*	50%	248	0.12	21%	0.03	0%	0.00	0%	0.00	21%	0.03	0%	0.00
	40%	2,610	1.80	71%	1.27	6%	0.11	1%	0.01	64%	1.15	4%	0.07
	30%	16,154	22.41	87%	19.56	1%	0.24	0%	0.11	86%	19.21	1%	0.24
T10I4D100k	0.100%	26,806	3.48	53%	1.83	0%	0.00	3%	0.11	49%	1.72	3%	0.12
	0.050%	46,993	6.60	68%	4.48	0%	0.00	2%	0.12	66%	4.36	2%	0.14
	0.010%	283,397	45.05	85%	38.52	0%	0.05	1%	0.45	84%	38.00	1%	0.59
	0.005%	769,777	96.73	87%	84.62	0%	0.33	2%	2.09	85%	82.14	3%	2.53
	0.000%	2,347,374	283.28	85%	239.56	1%	3.17	19%	54.91	64%	181.21	21%	58.30
T25I10D10k	0.30%	44,925	1.67	62%	1.04	0%	0.00	5%	0.09	57%	0.95	7%	0.11
	0.10%	209,436	28.35	89%	25.10	0%	0.03	1%	0.16	88%	24.91	1%	0.23
	0.05%	576,021	87.99	90%	78.98	0%	0.31	1%	1.09	88%	77.53	2%	1.40
	0.03%	1,438,054	180.65	90%	162.45	1%	1.07	2%	4.40	87%	156.89	3%	5.41
	0.00%	2,557,928	462.36	88%	408.33	1%	6.14	47%	218.91	40%	182.99	48%	224.13
T25I20D100k	0.30%	72,640	11.34	64%	7.27	0%	0.00	3%	0.35	61%	6.92	3%	0.38
	0.10%	150,970	17.59	68%	11.95	0%	0.00	3%	0.46	65%	11.47	3%	0.54
	0.05%	212,765	21.65	72%	15.52	0%	0.00	5%	1.09	67%	14.41	5%	1.16
	0.03%	461,138	45.30	77%	34.94	0%	0.03	4%	1.91	73%	32.96	5%	2.14
	0.01%	3,519,933	526.15	85%	447.88	1%	4.32	2%	11.44	82%	431.86	3%	15.04

Table 4.9: Additional timings of QuICL Oid-Less sections. Insert is the total seconds spent in the INSERT function. Prep, Prep Navigate, Prep Link, Prep Create, and Find Link are given as a percentage of Insert time with actual seconds subscripted.

4.8 Empirical Evidence to Support Asymptotic Complexity Analysis

To provide empirical evidence in support of the asymptotic complexity analysis, the QuICL Oid-Full algorithm was separately instrumented to report $|\mathcal{L}|$, $\deg_{avg}(\mathcal{L})$, number of intersections performed, and the number of iterations in the innermost loop of the intersection function. Applying these values in the formulas expressed in the runtime complexity will result in a calculated time that can be compared against the actual timings reported by the instrumented QuICL Oid-Full algorithm of Section 4.7. Any correlations found between the calculated times and actual times provide evidence supporting the runtime complexity.

For QuICL Oid-Full, the dominant term affecting runtime complexity is time to execute the prepare-search phase to identify and add entries to the ToProcessList. However, the time to purge subset entries is also significant. These times are reported by the instrumented QuICL Oid-Full as Prep Create and Purge, respectively. Furthermore, the runtime complexity of each is $O(\ell\, d\, i)$ and $O(\ell\, d^2\, c)$, where $\ell = |\mathcal{L}|$, $d = \deg_{avg}(\mathcal{L})$, i is a density weighted mean on the cardinality of frequent item extents, and c is a small fraction of $|\mathcal{O}|$ depending density. Table 4.10 presents empirical evidence in support of these complexities. Abs Supp is the absolute support derived by $|\mathcal{O}| \times$ Min Supp. Factor i must be ≥ Abs Supp. Avg Inter is an attempt to represent factor i. It is the average cost of intersection derived by number of iterations in the innermost loop / number of intersections. This value is indeed ≥ Abs Supp, is less than the cardinality of the maximum object id set, and appears to skew towards Abs Supp depending on density. Avg Deg is the average degree of the lattice. Prep Create C is a constant used to derive Prep Create Calc time. This value represents a constant unit of work incurred by the

algorithm. Prep Create Calc is the calculated time. Its value = $|\mathcal{L}| \times$ Avg Inter $\times$ Avg Deg $\times$ Prep Create C. Prep Create Actual is the Prep Create time reported by the instrumented QuICL Oid-Full algorithm. Purge C is a constant used to derive Purge Calc time. In this case Purge C represents both a constant unit of work incurred by the algorithm and factor c. Thus, Purge Calc = $|\mathcal{L}| \times (\text{Avg Deg})^2 \times$ Purge C. Purge Actual is the Purge time reported by the instrumented QuICL Oid-Full algorithm. Prep Create C and Purge C were adjusted for each data set, until the calculated times are near the reported actual times.

For most data sets, a Purge C constant could be found such that the calculated times are well correlated to the actual times. This indicates the purge time is indeed $O(\ell d^2 c)$. The T25I20D100k is the only exception. Here the actual times indicate either a greater complexity or the use of average degree in place of $|\mathcal{I}|$ is not a good fit. Alternative means such as quadratic (i.e., RMS) or average of averages at each level in the lattice were substituted, but did not offer better correlations.

For dense data sets a Prep Create C could be found such that the calculated times are well correlated to the actual times. This indicates that the prepare-search time is indeed $O(\ell d i)$. However, on sparse data sets a Prep Create C could not be found. In these data sets the actual times have a greater growth rate than $O(\ell d i)$. This does not necessarily disprove an $O(\ell d i)$. Simply, the value for i derived by number of iterations in the innermost loop / number of intersections is not a good mean function. On sparse data sets, the Avg Inters drops too quickly as the support is lowered. The discrepancy between dense and sparse data does indicate that density is a factor in computing the mean. Also, use of average degree in place of $|\mathcal{I}|$ may not be the good fit.

	Min Supp	$\|\mathcal{L}\|$	Abs Supp	Avg Inter	Avg Deg	Prep Create			Purge		
						C	Calc	Actual	C	Calc	Actual
Mushroom	50%	45	4,062	7,232	1.98	5E-09	0.00	0.02	1E-07	0.00	0.00
	40%	140	3,250	6,390	2.33	5E-09	0.01	0.03	1E-07	0.00	0.00
	30%	427	2,437	5,100	3.00	5E-09	0.03	0.05	1E-07	0.00	0.01
	20%	1,197	1,625	3,950	3.32	5E-09	0.08	0.12	1E-07	0.00	0.02
	10%	4,897	812	2,611	3.84	5E-09	0.25	0.34	1E-07	0.01	0.07
	5%	12,854	406	1,748	4.17	5E-09	0.47	0.63	1E-07	0.02	0.19
	1%	51,672	203	774	4.75	5E-09	0.95	1.15	1E-07	0.12	0.47
	0%	238,709	0	263	5.71	5E-09	1.80	1.65	1E-07	0.78	0.67
Chess	95%	74	3,036	3,169	2.68	2.5E-09	0.00	0.01	2E-07	0.00	0.00
	90%	503	2,876	3,093	3.64	2.5E-09	0.01	0.05	2E-07	0.00	0.00
	85%	1,885	2,717	2,992	4.40	2.5E-09	0.06	0.10	2E-07	0.01	0.00
	80%	5,083	2,557	2,894	5.03	2.5E-09	0.19	0.23	2E-07	0.03	0.03
	75%	11,525	2,397	2,804	5.49	2.5E-09	0.44	0.54	2E-07	0.07	0.06
	70%	23,991	2,237	2,712	5.84	2.5E-09	0.95	1.05	2E-07	0.16	0.13
	65%	49,240	2,077	2,607	6.17	2.5E-09	1.98	1.98	2E-07	0.38	0.36
	60%	98,392	1,918	2,490	6.51	2.5E-09	3.99	4.23	2E-07	0.84	0.75
	55%	192,863	1,758	2,361	6.85	2.5E-09	7.80	7.52	2E-07	1.81	1.69
Pumsb	95%	110	46,594	48,525	2.54	2.5E-09	0.03	0.10	1E-07	0.00	0.00
	90%	1,466	44,141	47,035	4.02	2.5E-09	0.69	0.94	1E-07	0.00	0.00
	85%	8,513	41,689	45,455	5.17	2.5E-09	5.00	5.23	1E-07	0.02	0.03
Pumsb*	50%	248	24,523	35,780	2.83	5E-09	0.13	0.24	3E-06	0.01	0.01
	45%	713	22,071	31,836	3.38	5E-09	0.38	0.63	3E-06	0.02	0.02
	40%	2,610	19,618	27,614	4.23	5E-09	1.52	1.51	3E-06	0.14	0.16
	35%	6,133	17,166	24,987	4.69	5E-09	3.60	3.16	3E-06	0.41	0.45
	30%	16,154	14,714	22,221	5.14	5E-09	9.22	8.13	3E-06	1.28	1.45

Table 4.10: Empirical evidence of asymptotic runtime analysis. Abs Supp = $|\mathcal{O}| \times$ Min Supp. Avg Inter = number of iterations in the innermost loop / number of intersections. Avg Deg is the average degree of the lattice. Prep Create C is a constant used to derive Prep Create Calc time. Prep Create Calc = $|\mathcal{L}| \times$ Avg Inter $\times$ Avg Deg $\times$ Prep Create C. Prep Create Actual is the Prep Create time reported by the instrumented QuICL Oid-Full. Purge C is a constant used to derive Purge Calc time. Purge Calc = $|\mathcal{L}| \times (\text{Avg Deg})^2 \times$ Purge C. Purge Actual is the Purge time reported by the instrumented QuICL Oid-Full.

	Min Supp	$\|\mathcal{L}\|$	Abs Supp	Inter Avg	Avg Deg	Prep Create			Purge		
						C	Calc	Actual	C	Calc	Actual
T10I4D100k	2.000%	155	2,000	6,128	1.00	5E-08	0.05	0.47	7E-07	0.00	0.00
	1.000%	385	1,000	4,196	1.03	5E-08	0.08	1.75	7E-07	0.00	0.00
	0.500%	1,073	500	3,253	1.68	5E-08	0.29	2.97	7E-07	0.00	0.00
	0.300%	4,509	300	2,739	2.57	5E-08	1.58	3.71	7E-07	0.02	0.00
	0.100%	26,806	100	1,761	3.27	5E-08	7.71	4.86	7E-07	0.20	0.02
	0.050%	46,993	50	960	3.10	5E-08	6.99	5.90	7E-07	0.32	0.10
	0.030%	71,265	30	518	2.88	5E-08	5.32	7.15	7E-07	0.41	0.27
	0.010%	283,397	10	142	2.81	5E-08	5.66	14.04	7E-07	1.57	1.54
	0.005%	769,777	5	72	3.00	5E-08	8.31	23.36	7E-07	4.84	4.78
	0.000%	2,347,374	0	33	4.29	5E-08	16.51	51.83	7E-07	30.21	29.29
T25I10D10k	5.00%	72	461	1,055	1.00	5E-08	0.00	0.02	7E-07	0.00	0.00
	3.00%	389	277	755	1.00	5E-08	0.01	0.39	7E-07	0.00	0.00
	1.00%	5,582	92	531	3.58	5E-08	0.53	1.45	7E-07	0.05	0.01
	0.50%	23,394	46	468	3.68	5E-08	2.02	1.64	7E-07	0.22	0.01
	0.30%	44,925	28	366	3.59	5E-08	2.95	1.79	7E-07	0.40	0.03
	0.10%	209,436	9	46	2.63	5E-08	1.27	7.67	7E-07	1.02	1.34
	0.05%	576,021	5	26	2.74	5E-08	2.04	18.11	7E-07	3.02	4.13
	0.03%	1,438,054	3	18	3.01	5E-08	3.91	32.73	7E-07	9.12	9.26
	0.00%	2,557,928	0	14	4.30	5E-08	7.88	63.06	7E-07	33.07	36.76
T25I20D100k	3.00%	19	3,000	6,787	1.00	5E-08	0.01	0.01	7E-07	0.00	0.00
	2.00%	143	2,000	4,823	1.10	5E-08	0.04	0.25	7E-07	0.00	0.00
	1.00%	5,256	1,000	3,000	3.53	5E-08	2.78	6.22	7E-07	0.05	0.00
	0.50%	27,067	500	2,000	4.09	5E-08	11.06	27.92	7E-07	0.32	0.02
	0.30%	72,640	300	1,636	4.74	5E-08	28.18	45.27	7E-07	1.14	0.10
	0.10%	150,970	100	1,254	4.64	5E-08	43.94	67.32	7E-07	2.28	0.16
	0.05%	212,765	50	1,122	4.51	5E-08	53.77	73.62	7E-07	3.02	0.30
	0.03%	461,138	30	766	4.14	5E-08	73.11	81.74	7E-07	5.53	1.72
	0.01%	3,519,933	10	108	3.67	5E-08	69.96	190.44	7E-07	33.22	35.72

Table 4.10 continued: Empirical evidence of asymptotic runtime analysis.

4.9 Performance Analysis of the GMA Algorithm

To gain an understanding of time spent within the GMA algorithm and to provide empirical evidence for rationale concerning its runtime behavior, GMA was separately instrumented to time key sections of its ADD function (Algorithm 3.1). The ADD function performs the incremental insertion of the next item. Instrumentation includes:

i) Add – total time spent in the ADD function,

ii) Sort – time to sort concepts prior to iteration (line 14),

iii) Intersect – time spent performing intersections (line 21, also determines outcome of comparison for lines 15 and 18),

iv) Generator Test – time spent checking that a potential generator is indeed a generator (loop expressed by the $\forall$ of line 23), and

v) Link – time to find parents and link new concepts into the lattice (lines 27 through 36).

The results from executing the instrumented GMA algorithm against all benchmark data sets for relevant subset of supports are given in Table 4.11 (ascending support order) and Table 4.12 (descending support order). Column Add is the total seconds spent in the ADD function. Columns Sort, Intersect, Generator Test, and Link show the time spent executing respective sections of the ADD function. Other accounts for the remaining time spent in the ADD function. Sort, Intersect, Generator Test, Link, and Other are given as a percentage of Add time with actual seconds subscripted.

For GMA, considerable amount of time is consumed searching for parents and linking new concepts into the lattice. On dense data sets using descending order, the time to link parents is the dominant term ranging between 50% and 96% of the total time[43]. Furthermore, use of ascending order greatly exacerbates the Link time, up to 29 times

[43] Pumsb* at $50\%_{supp}$ is an exception.

more (e.g., Mushroom at $0.0\%_{supp}$) resulting in the Link time exceeding 90% of the total time in all cases. On sparse data set using descending order, the Link time is generally limited to a small percentage value[44]. For ascending order, the Link time ranges from 4% to 52% of the total time with the higher percents generally encountered at lower supports. For all data sets, the actual seconds for link time is greater on ascending order.

Beyond the Link time, the time to perform intersections is next major consumer. It is generally the dominant term of the runtime complexity on sparse data sets ranging between 61% to 89% when using ascending order, and 62% to 90% when using descending order. The only exceptions are T10I4D100k at $0.0\%_{supp}$, T25I10D10k at $0.03\%_{supp}$ and $0.0\%_{supp}$, and T25I20D100k at $0.01\%_{supp}$ on ascending order; and T25I10D10k at $0.0\%_{supp}$ on descending order. In these test cases, the Generator Test time is significant and has an impact on the total time. On dense data sets, the Intersect time is between 0% and 25% when using ascending order, and 4% to 56% when using descending order. In most cases[45], the actual seconds for Intersect is more on descending order, up to seven times (e.g., Chess at less than $75\%_{supp}$). This is reflective of the fact that insertion in descending order will result in more intersections.

In general, the Generator Test time has a small percentage value, except in the cases previously mentioned. In such cases, the time consumed for Generator Test is exacerbated by ascending order. In general, time spent on sorting concepts is negligible, but might exceed 10% of the overall execution time in a few cases (e.g., T25I10D10k at $0.1\%_{supp}$ and $0.05\%_{supp}$ in ascending order and all supports in descending order). Here the sorting time is exacerbated by descending order.

---

[44] 14% on T10I4D100k at $0.0\%_{supp}$ and 21% on T25I10D10k at $0.0\%_{supp}$ are a couple of exceptions.
[45] Mushroom at $0.0\%_{supp}$ is the only exception.

	Min Supp	$\|\mathcal{L}\|$	Add	Sort		Intersect		Generator Test		Link		Other	
Mushroom	10%	4,897	9.43	0%	0.00	9%	0.89	0%	0.01	90%	8.48	1%	0.05
	5%	12,854	37.44	0%	0.02	6%	2.29	0%	0.06	93%	34.91	0%	0.16
	1%	51,672	369.67	0%	0.08	2%	8.70	0%	0.72	97%	359.40	0%	0.77
	0%	238,709	5,254.37	0%	0.70	1%	38.11	1%	38.67	99%	5176.76	0%	0.13
Chess	85%	1,886	1.36	0%	0.00	4%	0.05	0%	0.00	95%	1.29	1%	0.02
	75%	11,526	25.51	0%	0.01	1%	0.30	0%	0.02	98%	25.11	0%	0.07
	65%	49,241	315.32	0%	0.03	0%	1.50	0%	0.26	99%	313.11	0%	0.42
	55%	192,864	3,794.69	0%	0.13	0%	6.82	0%	4.16	100%	3781.96	0%	1.62
Pumsb	90%	1,467	8.88	0%	0.00	7%	0.63	0%	0.01	91%	8.08	2%	0.16
	85%	8,514	104.24	0%	0.01	4%	3.67	0%	0.08	96%	99.94	1%	0.54
Pumsb*	50%	249	1.27	0%	0.00	25%	0.32	0%	0.00	70%	0.89	5%	0.06
	40%	2,611	31.93	0%	0.01	10%	3.24	0%	0.01	89%	28.48	1%	0.19
	30%	16,155	457.21	0%	0.02	4%	17.19	0%	0.25	96%	439.01	0%	0.74
T10I4D100k	0.100%	26,807	31.94	1%	0.36	89%	28.31	0%	0.00	6%	1.99	4%	1.28
	0.050%	46,994	50.00	1%	0.63	90%	44.90	0%	0.01	6%	3.04	3%	1.42
	0.010%	283,398	255.73	2%	4.79	84%	214.86	1%	1.83	13%	32.03	1%	2.22
	0.005%	769,778	819.77	2%	15.08	67%	545.45	4%	34.82	27%	219.93	1%	4.49
	0.000%	2,347,375	9,723.21	1%	67.25	16%	1508.63	32%	3087.01	52%	5043.36	0%	16.96
T25I10D10k	0.30%	44,926	15.89	6%	0.95	81%	12.94	0%	0.02	10%	1.60	2%	0.38
	0.10%	209,437	61.69	10%	5.98	77%	47.42	1%	0.35	11%	6.78	2%	1.16
	0.05%	576,022	203.48	10%	21.22	61%	123.91	7%	13.54	20%	41.53	2%	3.28
	0.03%	1,438,055	829.72	7%	56.12	32%	269.52	25%	206.92	35%	288.76	1%	8.40
	0.00%	2,557,929	5,066.23	2%	108.58	9%	453.26	46%	2319.62	43%	2162.48	0%	22.29
T25I20D100k	0.30%	72,641	360.82	2%	5.73	89%	321.55	0%	0.06	8%	28.11	1%	5.37
	0.10%	150,971	831.99	3%	21.50	92%	763.89	0%	0.11	4%	37.43	1%	9.06
	0.05%	212,766	1,021.30	3%	30.43	92%	934.73	0%	0.16	4%	45.67	1%	10.31
	0.03%	461,139	1,314.02	3%	39.56	84%	1109.13	0%	2.94	12%	151.51	1%	10.88
	0.01%	3,519,934	10,809.01	2%	251.71	43%	4623.86	4%	441.33	50%	5452.81	0%	39.30

Table 4.11: Timings of GMA algorithm sections using ascending order. Add is the total seconds spent in the ADD function. Sort, Intersect, Generator Test, and Link is the time spent in respective sections of the ADD function. Other accounts for the remaining time spent in the ADD function. Sort, Intersect, Generator Test, Link, and Other are given as a percentage of Add time with actual seconds subscripted.

	Min Supp	$\|\mathcal{L}\|$	Add	Sort		Intersect		Generator Test		Link		Other	
Mushroom	10%	4,897	4.98	1%	0.04	48%	2.41	0%	0.02	50%	2.51	0%	0.00
	5%	12,854	12.94	1%	0.09	42%	5.40	1%	0.11	57%	7.34	0%	0.00
	1%	51,672	50.46	1%	0.65	24%	12.36	1%	0.46	73%	36.66	1%	0.33
	0%	238,709	206.72	2%	3.50	11%	21.72	2%	4.31	86%	176.98	0%	0.21
Chess	85%	1,886	1.22	1%	0.01	15%	0.18	1%	0.01	81%	0.99	2%	0.03
	75%	11,526	18.61	0%	0.03	11%	2.02	0%	0.04	88%	16.43	0%	0.09
	65%	49,241	146.51	0%	0.15	7%	10.93	0%	0.27	92%	134.84	0%	0.32
	55%	192,864	1,247.66	0%	1.01	4%	49.79	0%	1.49	96%	1195.31	0%	0.06
Pumsb	90%	1,467	11.24	0%	0.01	32%	3.58	0%	0.00	68%	7.64	0%	0.01
	85%	8,514	114.78	0%	0.02	19%	22.35	0%	0.02	80%	91.71	1%	0.68
Pumsb*	50%	249	1.45	0%	0.00	56%	0.81	0%	0.00	44%	0.64	0%	0.00
	40%	2,611	27.20	0%	0.02	42%	11.38	1%	0.15	58%	15.64	0%	0.01
	30%	16,155	270.79	0%	0.10	30%	80.38	0%	1.01	70%	189.14	0%	0.16
T10I4D100k	0.100%	26,807	56.07	3%	1.96	91%	51.17	0%	0.01	2%	1.27	3%	1.66
	0.050%	46,994	91.31	5%	4.43	91%	83.21	0%	0.03	2%	1.69	2%	1.95
	0.010%	283,398	453.17	10%	44.59	86%	391.82	0%	0.31	2%	10.64	1%	5.81
	0.005%	769,778	1,132.01	12%	140.23	82%	929.01	0%	4.00	4%	46.01	1%	12.76
	0.000%	2,347,375	3,743.05	12%	441.53	62%	2317.73	12%	436.21	14%	506.44	1%	41.14
T25I10D10k	0.30%	44,926	30.59	12%	3.55	82%	25.02	0%	0.02	4%	1.28	2%	0.72
	0.10%	209,437	132.98	20%	26.24	75%	99.09	0%	0.12	3%	3.75	3%	3.78
	0.05%	576,022	339.38	24%	82.33	67%	227.33	1%	2.70	5%	17.20	3%	9.82
	0.03%	1,438,055	870.10	25%	218.59	57%	500.02	5%	47.37	9%	78.83	3%	25.29
	0.00%	2,557,929	2,277.33	18%	399.62	35%	801.83	25%	558.24	21%	467.04	2%	50.60
T25I20D100k	0.30%	72,641	740.54	3%	23.08	92%	683.00	0%	0.09	4%	26.99	1%	7.38
	0.10%	150,971	1,383.49	5%	72.99	91%	1262.74	0%	0.12	2%	33.82	1%	13.82
	0.05%	212,766	1,960.86	7%	135.88	90%	1760.38	0%	0.17	2%	43.45	1%	20.98
	0.03%	461,139	4,547.71	9%	431.78	87%	3964.11	0%	0.93	2%	97.30	1%	53.59
	0.01%	3,519,934	30,885.52	26%	8024.57	69%	21456.19	0%	49.94	3%	1020.69	1%	334.13

Table 4.12: Timings of GMA algorithm sections using descending order. Add is the total seconds spent in the ADD function. Sort, Intersect, Generator Test, and Link is the time spent in respective sections of the ADD function. Other accounts for the remaining time spent in the ADD function. Sort, Intersect, Generator Test, Link, and Other are given as a percentage of Add time with actual seconds subscripted.

4.10 Comparison of Intersections

As a final comparison between algorithms, CHARM, GMA, QuICL Oid-Full, and QuICL Oid-Trie were instrumented to report the number of intersections performed during the construction of a lattice. For GMA (Algorithm 3.1), the number of intersections is the number of times through the main loop of line 14. Each time through the loop, one intersection is performed. The result of intersection is used to determine the outcome of $\subseteq$ at line 15, outcome of = at line 18, and the intersection set of line 21. The number of intersections does not include any comparison tests involved with testing for generators or searching for parent concepts. For QuICL Oid-Full (Algorithm 3.4), the number of intersections equals the number of times through the prepare-search loop of line 10. Each time through the loop, one intersection is performed. The result of intersection is used to determine the outcome of =, $\subset$, and $\supset$ (lines 11, 14, and 16 respectively), and the intersection set of lines 18 and 19. The number of intersections does not include any comparison tests involved with purging subset entries. For CHARM, the number of intersections is the number performed at line 10 of Algorithm 2.1. Ascending support order is used for QuICL Oid-Full and QuICL Oid-Trie. Both ascending and descending support order is used for GMA. CHARM internally performs a sort.

Table 4.13 provides the results for the purpose of comparison. In comparing CHARM to QuICL Oid-Full, QuICL Oid-Full performs 20% to 45% more intersections on Chess with the percentage increasing as the support is lowered, 18% to 65% more on Pumsb* with the percentage decreasing as the support is lowered, and 12% more (at $10\%_{supp}$) to 19% less (at $0.0\%_{supp}$) on Mushroom. On T10I4D100k and T25I10D10k,

QuICL Oid-Full performs approximately 5 times more intersections at high supports but 16% and 39% less, respectively, at $0.0\%_{supp}$. On T25I20D100K, the number intersections for QuICL Oid-Full are more than sixty times that of the CHARM algorithm at high supports, but drops to 64% more at $0.01\%_{supp}$. QuICL Oid-Trie performs 0% to 4% less intersections than QuICL Oid-Full. This small gain is the result of the direct look-up of parent concepts already present in the lattice. The look-up effectively bypasses a small number of intersections. In comparing QuICL Oid-Full with GMA ascending, GMA performs more intersections ranging from a few percent (e.g., on Mushroom) to over an order of magnitude (e.g., on T25I20D100k). GMA algorithm performs about time five times more intersections in descending order of the input than in ascending. This clearly indicates that the number of intersections performed by incremental insertion is significantly restrained using ascending support order over descending order.

	Min Supp	$\|\mathcal{L}\|$	CHARM	Oid-Full	Oid-Trie	GMA Ascending	GMA Descending
Chess	85%	1,885	2,040	2,453	2,453	2,469	8,872
	75%	11,525	12,300	16,194	16,194	16,494	101,650
	65%	49,240	52,680	73,012	73,012	76,422	580,043
	55%	192,863	206,058	299,452	299,451	325,385	2,776,025
	45%	707,964	764,897		1,147,942		
Mushroom	10%	4,897	13,755	15,411	15,060	28,274	159,598
	5%	12,854	41,559	42,475	41,337	84,328	559,654
	1%	51,672	204,831	171,773	165,005	433,931	2,658,520
	0%	238,709	722,998	585,509	561,327	3,008,711	13,294,536
Pumsb*	50%	248	386	636	631	1,046	2,681
	40%	2,610	4,094	5,724	5,715	10,468	44,619
	30%	16,154	29,763	35,214	35,197	62,533	373,966
T10I4D100k	0.100%	26,806	84,549	492,326	492,324	2,121,378	11,065,654
	0.050%	46,993	450,339	1,076,219	1,076,189	3,540,405	22,135,262
	0.001%	283,397	11,425,072	15,273,903	15,270,100	18,980,003	158,563,214
	0.001%	769,777	26,845,219	41,432,480	41,400,882	53,023,943	429,938,356
	0.000%	2,347,374	135,897,608	113,860,447	110,427,801	186,602,383	1,321,116,356
T25I10D10k	0.300%	44,926	122,921	741,001	740,938	6,109,812	20,184,635
	0.100%	209,436	12,213,130	17,410,550	17,407,832	27,460,247	113,839,759
	0.050%	576,022	45,080,919	57,975,763	57,941,620	81,843,625	303,212,996
	0.030%	1,438,054	87,667,671	112,295,967	112,054,097	196,187,133	751,113,513
	0.000%	2,557,928	255,133,119	155,603,121	150,763,040	362,230,207	1,349,704,766
T25I20D100k	0.300%	72,640	72,439	4,628,183	4,628,183	31,933,715	99,224,142
	0.100%	150,970	152,995	9,525,668	9,525,668	106,471,253	279,288,120
	0.050%	212,765	388,754	11,837,524	11,837,501	143,330,271	503,391,739
	0.030%	461,138	3,068,334	19,085,665	19,083,580	168,701,767	1,602,853,580
	0.010%	3,519,933	148,382,639	243,262,618	243,019,880		

Table 4.13: Comparison of intersections by algorithm.

4.11 Summary of Results

This chapter has presented the results of empirical evaluations and analysis of the QuICL algorithms against the GMA, CHARM, CHARM-L, and MAGALICE algorithms. The CHARM and CHARM-L algorithms were downloaded from the author's web site and converted to Java. MAGALICE was downloaded from the Galicia project. Its source is in Java. An iceberg enhanced GMA algorithm and the QuICL algorithms were directly implemented in Java. Thus, all algorithms are in Java enabling all tests to be performed on the same platform and environment.

The Mushroom, Chess, Pumsb, Pumsb*, T10I4D100k, T25I10D10k, and T25I20D100k data sets were used as the benchmarks. Mushroom and Chess are examples of dense data sets, Pumsb and Pumsb* are marginal, and the rest are sparse. The characteristics of the lattices generated from these data sets indicate that the size of the lattice grows exponentially as the support is lowered, but the average degree and height of the lattice grows at a slow steady rate. The maximum degree on sparse data sets quickly approaches $|\mathcal{I}|$, where on dense data sets the maximum degree is much closer to the average. The density of a data set, calculated by $|\mathcal{R}| / (|\mathcal{O}| \times |\mathcal{I}|)$, can be readily observed in a density profile.

The QuICL algorithms were validated by a two prong approach; first through manual inspection of the construction of small lattices. All test lattices, including iceberg lattices, were correctly constructed. The second prong involved executing all algorithms including GMA, CHARM, CHARM-L, and MAGALICE against the benchmark data sets and comparing the characteristics of the generated lattices. The QuICL algorithms reported the same $|\mathcal{L}|$ as CHARM and CHARM-L, and a difference of at most one for

GMA on all data sets and selected supports. The difference is readily explained by the underlying representation of each lattice. The QuICL algorithms reported the same $deg_{avg}(\mathcal{L})$ as the CHARM-L and GMA algorithms. Based on the consistent characteristics of the lattices constructed by the QuICL, CHARM, CHARM-L, and GMA algorithms, the QuICL algorithms were deemed valid.

Before comparing the performance and memory usage of the QuICL algorithms against the CHARM, GMA, and MAGALICE algorithms, experiments were conducted to determine if the order of item insertion has an effect on performance and memory usage. If an effect is realized, then the order providing the best performance and memory usage will be used when comparing the algorithms. The results of experiments indicate that QuICL Oid-Full and QuICL Oid-Trie provide the best performance by incrementally inserting items in ascending support order. By inserting concepts in ascending item support order, the lattice initially grows at small rate that accelerates towards later insertions, thereby restraining the number of intersections performed. For QuICL Oid-Less, the best performance on dense data sets is attained by incrementally inserting items in descending support order. This conflicting preference is the result of intersecting sets of support concepts instead sets of object ids. On sparse data sets, the intersection of support concepts had reduced effects. Therefore on sparse data sets, ascending order provides the best performance. The GMA algorithm also exhibits significant gains in performance by inserting the items in descending support order for dense data sets, up to an order of magnitude. Here the conflicting preference is attributed to the cost to link new concepts into the lattice. However, the cost of searching for parents has a reduced effect on sparse data sets. Thus, on sparse data sets the ascending order provides the best

performance. For QuICL Oid-Full, QuICL Oid-Trie, and GMA, the sort order had no effect on memory usage. This is due to the well behavior of lattice construction. For QuICL Oid-Less, descending order resulted in less memory usage. Descending order restrains the size of temporary sets created during item insertion.

Using the sort order providing the best performance and memory usage, the performance of QuICL algorithms were compared to CHARM, CHARM-L, GMA, and MAGALICE through empirical tests against the seven benchmark data sets. The QuICL Oid-Full algorithm provided the best overall performance for constructing iceberg lattices. It is only outperformed by CHARM-L on the Pumsb and Pumsb* data sets and for sparse data at only relatively high supports. The Pumsb and Pumsb* data sets contain items with very large object id sets. Thus, CHARM-L is benefiting from its difference based representation of object id sets. While CHARM-L does outperform QuICL Oid-Full at relatively high supports on sparse data, the gain is generally limited to a few seconds. In all cases, the gain quickly degenerates into a large loss as the support is lowered. At low supports, QuICL Oid-Full outperforms CHARM-L by an excess of an order of magnitude on the four data sets, and a factor greater than five on a fifth. QuICL Oid-Trie exhibits the near same runtime complexity as QuICL Oid-Full for all data sets with a small performance overhead. This loss in performance is expected since the compare and intersect functions must traverse between trie nodes. QuICL Oid-Less, provides the best performance of the lattice construction algorithms on the Pumsb data set outperforming CHARM-L by more than a factor of two over all supports. QuICL Oid-Less is QuICL's answer to handling data set containing items with large object id sets. On data sets containing items with large object sets, such as Pumsb, QuICL Oid-

Less realizes a significant performance gain. This gain may also be realized for an initial period of time during the algorithm execution on other data sets.

CHARM provides the best performance for the Chess, Pumsb, and Pumsb* data sets. These results are expected since CHARM does not derive the upper covers and it uses a difference based representation for the sets of object ids. On these data sets the difference based representation provides a real gain in both in memory and runtime execution. The CHARM algorithm provides the best performance on sparse data sets at relatively high supports, but is outperformed by QuICL Oid-Full and QuICL Oid-Trie algorithms at low supports. CHARM is outperformed by QuICL Oid-Full and QuICL Oid-Trie on Mushroom over all supports. CHARM-L exhibits performance along the lines of CHARM but degrades by a factor of two to an excess of an order of magnitude as the support is lowered. These results are expected since the CHARM-L is an extension to CHARM that additionally derives the upper covers. CHARM-L is outperformed by QuICL Oid-Full and QuICL Oid-Trie on Chess over all supports.

GMA is generally slower than the QuICL and CHARM algorithms by an order of magnitude with greater divergence at lower minimum supports. There are the two main factors attributing to its degradation of performance. First, it performs more intersections than needed. Second, it incurs an expensive search for parents when linking a new concept into the lattice. MAGALICE exhibits the worst performance of all the algorithms. When compared to the GMA algorithm, MAGALICE is at least an order of magnitude slower.

QuICL Oid-Full and GMA exhibit similar memory usage for small lattices and diverge for larger lattices. For both algorithms, the number of object id entries stored in a

lattice is a major consumer of memory space, accounting for more than 95% of the memory used on small lattices. As the lattice becomes large, the overhead of storing the concepts can become a dominant term. Furthermore, for large lattices the number of parent-child links account for another 15%. Since the overhead of GMA concepts is greater and GMA uses two references for each parent-child link, a GMA lattice can consume two times more memory than QuICL Oid-Full on large lattices. QuICL Oid-Trie algorithm realizes a savings in the number of object id entries its trie between 15% and 80% over the QuICL Oid-Full lattice. For small lattices, these savings translate to a significant reduction in memory usage enabling QuICL Oid-Trie to process lower supports. However, on large lattices any gain in reducing the number of object id entries is outweighed by the overhead in the concepts. The QuICL Oid-Less algorithm provides further reduction in memory by eliminating the object id entries from permanent storage within the lattice. The permanent object ids are eliminated at the expense of additional overhead in each concept. The net change is a reduction in memory space between a factor of two to an order of magnitude. For dense data sets, these gains are sustained over most supports but diminish slightly for large lattices. Thus, QuICL Oid-Less is able to process lower supports than QuICL Oid-Full. However, on some sparse data sets, QuICL Oid-Less exhibits an excess between 5% and 60%. For these data sets, the lattice quickly degrades to a worst case where there are many iced concepts each representing a single object id. Like QuICL Oid-Less, the CHARM-L algorithm provides reduction in memory usage by eliminating the object id entries from the lattice. The overhead for concepts in the CHARM-L lattice is about three times QuICL Oid-Full and thus exceeds QuICL Oid-Less. Due to very different approaches, the reduction or gain in memory

space when compared against QuICL algorithms is varied. The CHARM algorithm does not construct a lattice. As such, it generally provides the best memory usage. Memory is temporarily consumed during processing of its itemset-oidset tree and permanently consumed by the list of found frequent item sets. The MAGACLICE algorithm exhibited gross memory consumption.

Performance analysis indicates that QuICL Oid-Full spends a majority of time in the prepare-search phase. In most cases the proportion of time is over 75% but decreases as the support is lowered. Thus, the prepare-search phase is the dominant part of the overall execution time. However, the time to purge subset entries is also significant and its proportion of time generally increases with the size of the lattice. Likewise, time to find and link parent concepts already in the lattice is also significant and increases with the size of the lattice. The remaining sections of the QuICL Oid-Full algorithm consume a negligible amount of time and are therefore insignificant.

Portions of time spent in the sections of the QuICL Oid-Trie are similar to QuICL Oid-Full with a few changes. On the Chess, Pumsb, and Pumsb* data sets, the actual seconds in the prepare-search phase is slightly less, but up to two times more on the other data sets. Traversal between trie nodes during intersection is accounting for the additional time. For Chess, Pumsb, and Pumsb, an enhancement to halt intersection processing when the traversals of the object id sets are within the same trie node is achieving a gain. For these data sets, the trie has a large number of object id sets sharing common prefixes. A second enhancement to directly lookup existing concepts limits the proportion of time to find and link existing concepts to less than 5%. Thus for QuICL Oid-Trie, this time is not a dominant term.

The portions of time spent in the sections of QuICL Oid-Less is different from QuICL Oid-Full. The portion of time spent executing the prepare-search increases as the support is lowered. For all data sets, this portion of time still exceeds 74% at low supports. The actual seconds spent is less than QuICL Oid-Full at high supports, but is generally more at low supports. At high supports QuICL Oid-Less is benefiting from intersecting sets of support concepts, but as the support is lowered the support concepts fragment resulting in greater runtime consumption. An additional time component incurred by QuICL Oid-Less is the time to process iced concepts. This time is around 30% at high supports but drops as the support is lowered. Overall, the QuICL Oid-Less algorithm provides a gain at high supports and realizes a loss as the support is lowered.

To provide empirical evidence in support of the asymptotic complexity analysis QuICL Oid-Full was separately instrumented to report $|\mathcal{L}|$, $\deg_{avg}(\mathcal{L})$, number of intersections performed, and the number of iterations in the innermost loop of the intersection function. From the reported values, a calculated time can be derived using the formulas expressed in the runtime complexity. Correlations found between the calculated times and actual times provide evidence supporting the runtime complexity. For dense data sets a prepare-search time constant could be found such that the calculated times are well correlated to the actual times. This indicates the prepare search time is indeed $O(\ell d i)$. However, on sparse data sets such constant could not be found. This does not necessarily disprove an $O(\ell d i)$. Instead, it indicates that the mean used for calculating i is not appropriate. The discrepancy between dense and sparse data does, however, indicate that density is a factor in computing the mean. For most data sets, a

purge time constant could be found such that the calculated purge times are well correlated to the actual times. This indicates the purge time is indeed $O(\ell d^2 c)$.

Performance analysis of the GMA algorithm indicates that it spends considerable time on searching for parents and linking new concepts into the lattice. On dense data sets using descending order, the time ranges between 50% and 96%. Furthermore, use of ascending order greatly exacerbates the search and link time. On sparse data using descending order, the time is generally limited to a small percentage. Beyond the search and link time, the time to perform intersections is next major consumer. It is generally the dominant term on sparse data sets, and between 0% and 25% when using ascending order on dense data sets. In most cases, the actual time spent on performing intersections is more when using descending order. The generator test time and time to sort concepts is a small percent, except in a few cases. Generator test is exacerbated by ascending order whereas sort time is exacerbated by descending order.

As a final comparison between algorithms, CHARM, GMA, QuICL Oid-Full, and QuICL Oid-Trie were instrumented to report the number of intersections performed during the construction of a lattice. QuICL Oid-Full performs 20% to 45% more intersections than CHARM on Chess, 18% to 65% more on Pumsb*, but up to 19% less on Mushroom. On T10I4D100k and T25I10D10k QuICL Oid-Full performs around five times more intersections at high supports but 16% and 39% less, respectively, at $0.0\%_{supp}$. QuICL Oid-Trie performs 0% to 4% less intersections than QuICL Oid-Full. In comparing QuICL Oid-Full with GMA ascending, GMA performs more intersections ranging from a few percent to over an order of magnitude. GMA descending performs around five times more intersections than GMA ascending.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

5.1 Conclusions

It was hypothesized that an iceberg concept lattice based algorithm will provide gains in association rule mining and will be effective in mining frequent items sets. This is found to be true. All QuICL algorithms correctly construct iceberg concept lattices for the specified minimum support threshold. The concepts of the iceberg lattices identify the frequent item sets together with their supports. The QuICL algorithms were validated by comparing the characteristics of the lattices generated by QuICL against those generated by GMA, CHARM[46], CHARM-L, and MAGALICE. All differences were explained. Therefore, the QuICL algorithms are deemed valid. Furthermore, the lattices constructed by QuICL are of the form corresponding to Figure 1.3. That is, each item is represented in a single concept, its maximal concept, each concept readily identifies its support, and the drop in confidence along an edge can be easily computed (i.e., support(parent) / support(child)). This notation enables association rules to be directly read from the lattice. Furthermore, a basis of association rules can be generated by traversing the lattice. The lattice is of the form whereby the Stumme et al. (2001b)

[46] CHARM does not construct a lattice. However, the number of frequent item sets identified by CHARM can be compared to the number of concepts in the lattice generated by QuICL.

algorithms can extract the Duquenne-Guigues basis and the Luxemburger basis. Therefore, the QuICL algorithms provide gains in association rule mining.

It was hypothesized that an iceberg concept lattice based algorithm will readily construct a concept lattice for a wide range of data sets and will prove to be a viable approach. This is found to be effectively true. The QuICL algorithms were tested against seven data sets using a spectrum of supports. The data sets represent a variety of characteristics including:

i) number of tuples ranging from a few thousand to a hundred thousand,
ii) number of items ranging from below a hundred to ten thousand,
iii) both sparse and dense data, and
iv) items with large object sets.

The QuICL algorithms constructed lattices for all supports on three of the data sets (Mushroom, T10I4D100k, and T25I10D10k), all but very near $0.0\%_{supp}$ on another (T25I20D100k), and medium to high supports on the other three (Chess, Pumsb, Pumsb*). The QuICL algorithms were unable to produce lattices over all supports for four of the data sets due to memory constraints. However, CHARM, CHARM-L, GMA, and MAGALICE were also unable to produce lattices over all supports for the same data sets due to memory constraints. CHARM was able to identify the frequent item sets for supports lower than QuICL on three of the data sets. But, CHARM does not produce the upper covers and is therefore deficient in the overall goal of identifying a useable set of association rules. CHARM-L did produce a lattice at a slightly lower support on one data set, but on another, failed at a support where the QuICL algorithms succeeded. On data sets containing items with large object sets, QuICL Oid-Full could only produce a lattice at high supports where CHARM-L could construct lattices for significantly lower

supports. On these data sets, QuICL Oid-Less was able to construct lattices for the same supports as CHARM-L. All QuICL algorithms were able to construct lattices for all data sets and supports where the GMA and MAGALICE succeeded. Given that the QuICL algorithms were able to construct lattices for all supports on three of the data sets, were more or less able to construct lattices at supports where CHARM-L was successful, and they were able construct lattices for all supports where GMA and MAGALICE succeeded, the QuICL algorithms have proved to be a viable approach.

It was hypothesized that an iceberg concept lattice based algorithm will exhibit the same or slightly better memory utilization than other leading algorithms. With respect to CHARM, a leading frequent item set miner, this is found to be false. CHARM does not construct a lattice. Therefore, there is no concept overhead, no parent-child links, and no retention of object id sets. Furthermore, CHARM uses a difference based representation for its temporary object id sets and these sets are released as soon as possible. Thus, CHARM consumes significantly less memory on dense data sets. While at high supports CHARM is only slightly less, CHARM is factor of two to over an order of magnitude better than QuICL Oid-Full at low supports. On sparse data sets, QuICL Oid-Full consumed slightly more over all supports. However, on one sparse data set, CHARM consumed more memory. For the dense data sets, QuICL Oid-Trie and QuICL Oid-Less derivations did provide a reduction is memory space over QuICL Oid-Full, but the reduction was not sufficient to match CHARM. QuICL Oid-Trie and QuICL Oid-Less still consume 50% to ten times more than CHARM at low supports.

With respect to leading lattice construction algorithms, the hypothesis concerning memory was found to be mostly true. Memory consumption for QuICL Oid-Full was the

same as GMA on dense data sets, and the QuICL Oid-Trie and QuICL Oid-Less derivations generally provided a 50% to 80% reduction in memory usage. On sparse data sets, QuICL Oid-Full was 10% to 40% less over most supports, although greater reduction was provided at low supports. When compared against CHARM-L, the QuICL algorithms provide similar reduction on sparse data sets. However, on dense data sets CHARM-L consumes less memory. The difference based representation of the underlying CHARM algorithm is providing a benefit. Therefore, CHARM-L is realizing a gain, especially on data sets having items with large object sets. However, QuICL Oid-Trie and QuICL Oid-Less do challenge CHARM-L and at some supports provide a reduction in memory usage.

It was hypothesized that an iceberg concept lattice based algorithm will exhibit runtime performance on the order of leading algorithms to mine frequent item sets, but will probably be slower due to greater dependencies on intersection, union, and set difference operations. The QuICL algorithms were found to have performance on the order of leading algorithms to mine frequent item sets and, in a few of cases, provided the best performance. On the Mushroom, a dense data set, QuICL Oid-Full was faster than CHARM by a factor of two over all supports. For example, at $0.0\%_{supp}$ QuICL Oid-Full was less than three seconds where CHARM took over six seconds. On Chess, another dense data set, QuICL was slower by a factor of four. On two sparse data sets, QuICL was slower by a few seconds at high supports, but as the support was lowered QuICL provided gains around an order of magnitude. For example, on T10I4D100k at $0.5\%_{supp}$, QuICL Oid-Full took three seconds where CHARM was under a second. At $0.0\%_{supp}$, QuICL was under 120 seconds where CHARM was over 1,400. However, on a third

sparse data set, CHARM maintained a significant lead over most supports. Only at $0.01\%_{supp}$, did QuICL prevail. On data sets containing items with large object sets, CHARM was faster by an approximate factor of four (less at low supports and more at high supports). While for these data sets the QuICL Oid-Less derivation did provide some savings and was able to match or beat CHARM at high to medium supports, it lost by a factor of four to an order of magnitude at low supports.

QuICL Oid-Full provided the best all around performance of the lattice based algorithms. On both dense and sparse data sets, QuICL Oid-Full provided approximately 50% savings over GMA at high supports and an order of magnitude savings at low supports. For example, QuICL Oid-Full constructed the entire lattice for Mushroom in less than three seconds where GMA took over 200 seconds. For T10I4D100k, a sparse data set, QuICL Oid-Full completed in less than 120 seconds where GMA is near 10,000 seconds. QuICL Oid-Full provided near two orders of magnitude savings over MAGALICE. When compared against CHARM-L, QuICL generally provided the best performance. On Mushroom, CHARM-L is at least three times slower. At $0.0\%_{supp}$, QuICL Oid-Full was less than three seconds, whereas CHARM-L approached 200 seconds. Likewise on Chess, CHARM-L was at least three times slower. Only on data sets containing items with very large object sets did CHARM-L prevail. On these data sets, CHARM-L was five times faster. However, on one such data set the QuICL Oid-Less derivation was four times faster over all supports. On sparse data sets, QuICL Oid-Full was slower by a few seconds at high supports, but as the support was lowered QuICL Oid-Full provided gains around an order of magnitude. For example, on T10I4D100k at $0.5\%_{supp}$, QuICL Oid-Full took three seconds where CHARM-L was

under one second. At 0.0% $_{supp}$, QuICL was under 120 seconds whereas CHARM-L was near 2,000 seconds.

It was hypothesized that an iceberg concept lattice based algorithm will be resilient against variations of data characteristics and input order. This is found to be partially true. QuICL Oid-Full provided the best all around performance of the lattice based algorithms. It performed well on both dense and sparse data sets, and for small and large data sets. The QuICL Oid-Less and QuICL Oid-Trie derivations only offer gains in memory and or performance on dense data sets. The input order did indeed have no effect on the memory consumption on QuICL Oid-Full and QuICL Oid-Trie. This is due to the well behavior of lattice construction. Only for QuICL Oid-Less did input order have an effect. Insertion in descending support order restrained the size of temporary sets. The input order did have a large effect on the performance for all QuICL derivations. All QuICL algorithms are subject to a natural preference for ascending support order. Such order impedes the initial growth of the lattice, thereby reducing the number of required intersections. QuICL Oid-Less did exhibit a conflicting preference on dense data sets. In this case, QuICL Oid-Less offers significant savings in the cost of each intersection by inserting items in descending order.

In conclusion, this study investigated the development of efficient algorithms to construct an iceberg lattice. Its objective was to develop an algorithm whose overall performance in constructing a lattice is comparable to the leading algorithms used for association rule mining. Furthermore, it was proposed that such algorithm would provide gains relative to the overall task of association rule mining. This objective has been met. The performance of QuICL algorithms is on the order of leading algorithms to mine

frequent item sets, and QuICL additionally derives the upper covers. The lattices constructed by QuICL are of the form whereby association rules can be directly read and a basis can be readily generated. The Stumme et al. (2001b) algorithms can be used to extract the Duquenne-Guigues basis and the Luxemburger basis. Thus, the QuICL algorithms provide significant gains in the overall task of association rule mining. They enable the generation of association rules whose size is constrained to a number that can be exploited by the end user. Therefore, the QuICL algorithms offer a significant contribution to association rule mining. Beyond this, it was proposed that new efficient algorithms to construct concept lattices may present a contribution to formal concept analysis. The QuICL algorithms provide 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 Oid-Full provides the best all-around performance. QuICL Oid-Trie provides a reasonable tradeoff between performance and memory enabling it to create lattices for lower supports. QuICL Oid-Less addresses a special class of data sets containing items with large object sets. These derivations allow

construction of lattices for cases where GMA fails. Therefore, the QuICL algorithms offer a significant contribution to formal concept analysis.

5.2 Implications

Association rule mining is a challenging task due to the exponential nature of the problem. Small to moderate data sets can readily generate millions of frequent item sets. From a technical perspective, association rule mining presents challenges in both runtime execution and memory usage. Attention to efficiency is needed to ensure algorithms are successful within space and time constraints. However, efficiency is only one factor in assessing the effectiveness of association rule mining. Critical to the effectiveness, is a means to constrain the number of found association rules to a size that can be exploited by the end user. It is here where most past work has fallen short. The QuICL algorithms have maintained an attention to efficiency while at the same time derived the missing information needed to generate a basis of association rule. This has been achieved through a formal concept analysis approach. It is therefore postulated that QuICL algorithms offer the best solution to mining frequent items together with the upper covers. The QuICL algorithms combined with algorithms to extract a basis of association rules from a lattice, such as Stumme et al. (2001b), will provide the most efficient path to derive a set of association rules whose size is constrained to an exploitable number.

Frequent item set mining and lattice construction algorithms derived from formal concept analysis has in the past been two separate areas of research. Very few studies compare results of frequent item set miners against lattice construction algorithms, yet these areas are clearly related. Results included in this study, together with analysis of work performed in each area, indicate lattice construction algorithms are typically an

order of magnitude slower than frequent item set miners. Only CHARM-L provides good results in reducing the gap. The QuICL algorithms have gone a step further. The QuICL algorithms are true incremental lattice construction algorithms that have efficiency on the order of frequent item set miners. The QuICL algorithms have effectively bridged the gap.

The QuICL algorithms have incorporated the best features of a number of algorithms. The main loop of the QuICL algorithms is very similar to the main loop of CHARM. Both compare an incoming object id set against an id set present in its data structure and perform different actions depending if the sets are $=$, $\subset$, $\supset$, or $\cap$ meeting the minimum support threshold, although the actions of each are adapt to the respective data structure. The purge subsets function is the same as MAXIMA function of the Valtchev et al. (2000) (VML) lattice construction algorithm. Both serve the same purpose of preventing invalid parent-child links. The QuICL algorithms conform to the general principles of the Valtchev et al. (2003) generic lattice construction algorithm, although a different ordering of steps is utilized. The idea to use a trie structure for QuICL Oid-Trie was borrowed from the GALICIA-T (Valtchev et al., 2002) and Nourine and Raynaud (2002) algorithms. The rationale to maintain parent concepts id descending support order was borrowed from CHARM (Zaki, & Hsiao, 2002). CHARM sorts its child nodes in order to increase the probability of detecting, sooner than later, a case which conserves processing. Likewise, the order of parent concepts increases the probability of conserving processing.

QuICL algorithms differ from past work in several notable ways;

i) The QuICL algorithm is a pure incremental lattice construction algorithm. That is, its sole data structure driving its processing is a lattice. Many other algorithms are driven by some other data structure and construct the lattice as an integral sub-task. Such algorithms include: GALICIA-T (Valtchev et al., 2002), Nourine and Raynaud (2002), and CHARM-L (Zaki, & Hsiao, 2005). By being a pure incremental lattice construction algorithm, the foundation of QuICL is based solely on formal concept analysis. Additional theory derived from FCA may provide for further improvements to QuICL.

ii) The QuICL algorithms have recognized that it is sufficient to store an item in only its maximal position. There is no need to include the item in all descendent concepts. Thus, the only modified concepts will be those where an item is inserted at its maximal position. This eliminates the need to modify a substantial number of concepts thereby significantly improving performance.

iii) In comparing QuICL to GMA (Godin et al., 1995), both identify generator concepts. For QuICL, the generators are base concepts that do not have a parent whose object id set that 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.

iv) While QuICL does conform to the general principles of the Valtchev et al. (2003) generic lattice construction algorithm, it has effectively, through recursion, folded the sequential steps into an interleaved process. The main outcome is, again, the direct identification of parents when linking a new concept into the lattice.

Given this discussion and the results presented in this report, it is postulated that QuICL is the “best known” all around incremental lattice construction algorithm.

5.3 Recommendations

Given the favorable results and conclusions of this report, the QuICL algorithms are recommended for use in association rule mining and formal concept lattice construction. They are proven to be correct and highly efficient. For association rule mining, the QuICL algorithms provide the missing information needed to extract a basis of association rules. They are ready to be combined with basis extraction algorithms to form a complete solution for association rule mining. Furthermore, the QuICL algorithms are ready to be included in lattice construction and analysis suites, such as Galicia (Valtchev el al., 2003).

An obvious next step is to combine QuICL with basis extraction algorithms, such as Stumme et al. (2001b), in order to further validate the claim of the "best known" association rule mining solution. Another step is evaluation against a broader set of data sets and other lattice construction algorithms. This is needed to further validate the claim of "best known" all around lattice construction algorithm.

An issue for QuICL, as well as frequent item set miners and lattice construction algorithms in general, is memory consumption. The exponential nature of the problem can quickly exhaust memory space. 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. The QuICL Oid-Less and QuICL Oid-Trie derivations were able the construct lattices at lower supports than QuICL Oid-Full and GMA, but still failed at some point. The CHARM algorithm was able to process even lower supports. Further investigation into CHARM's difference based representation may shed light on additional improvements to QuICL. Another avenue for reducing memory may be found in

Algorithm 3.9. This algorithm pushes lattice intersected object ids into the descendents of each support concepts that logically represent the intersected object id. This algorithm was abandoned in place of a hybrid pull-down bottom-up algorithm, since no further performance enhancements were apparent. However, this algorithm does hold a key to saving memory. The QuICL algorithms determine if two sets are =, $\subset$, $\supset$, or $\cap$ using the cardinality of intersection sets. With the exception of the purge subset function, the actual ids are not needed. Thus, Algorithm 3.9 could simply increment an intersection count in all descendents instead of appending an object id. Thus, the temporary sets of QuICL Oid-Less would not be needed. This approach is dependent upon finding an alternate solution to purge subsets.

Further improvements to QuICL may be found by closer examination of number of intersections performed by QuICL and CHARM. QuICL is on par with CHARM in a number of cases, yet there are still further cases where QuICL performs significantly more intersections. Studying these cases may shed light on other enhancements. Also, investigations in the cost of intersection may prove fruitful.

The MAGALICE (Rouane et al., 2004) algorithm exhibited the worst performance of all algorithms, however, its intent has merit. Its intent is to enable incremental insertion of an object to an existing iceberg lattice. The rationale is to facilitate the addition of new set of objects to an already constructed lattice. For example, in a retail system it may be desired to add transactions for the previous day into a lattice derived from the all past transactions. The downfall of the MAGACLICE algorithm is that the adjustments are made for each individual object. Instead, some method to adjust the lattice for a set of new objects as a whole is needed. This may

involve retention of information about object sets that did not meet the minimum support threshold, construction of a separate complete lattice for the new objects, assessment of the concepts in the new lattice relative to the retained information, and integration of selected new concepts into the lattice. This is area for future research.

5.4 Summary

Association rule mining 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 and $X \cap Y$ is $\varnothing$. It has been applied to a wide range of domains including basket analysis, database analysis, and organization of pages on the World Wide Web. Furthermore, association rule theory has extended beyond its original domain to include correlations, dependency rules, episodes, sequential patterns, and multi-dimensional patterns. Association rule mining has thus been a major area of research. However, a large portion of activity has been focused on efficient techniques and innovative theory to extract frequent item (FI) sets. Notable algorithms include CHARM, CLOSET, TITANIC, and CLOSET+. While significant progress has been made, FI mining has fallen short of the overall objective of mining association rules. The FI miners fail to identify the upper covers of each closed FI set. The upper covers are needed to generate a set of association rules whose size is constrained to a number that can be exploited by an end user. The identification of upper covers is generally considered to be a worst case quadratic problem in terms of the number of FI sets.

An alternative to FI mining algorithms can be found in formal concept analysis (FCA), a branch of applied mathematics. Given a formal context composed of a set of objects $\mathcal{O}$, a set of items $\mathcal{I}$, and a relation $\mathcal{R} \subset \mathcal{O} \times \mathcal{I}$, FCA derives a set of concepts

where each concept is a pair of sets $O \subseteq \mathcal{O}$ and $I \subseteq \mathcal{I}$ such that $O = \{o \in \mathcal{O} \mid \forall i \in I, o\mathcal{R}i\}$ and $I = \{i \in \mathcal{I} \mid \forall o \in O, o\mathcal{R}i\}$. Furthermore, between any two concepts $C_1 = (O_1, I_1)$ and $C_2 = (O_2, I_2)$ an order $<$ is said to exists between C_1 and C_2 iff $O_1 \subset O_2$. Thus, the derived concepts can be arranged into a lattice structure by defining a connection between any two concepts C_1 and C_2 for which order $<$ exists and there is no concept C_3 for which $C_1 < C_3 < C_2$. The result is a lattice whose concepts identify the set of closed FIs (I) together with their support (|O|), and its connections identify the upper covers.

The study of FCA has been a strong area of research. Noteworthy algorithms include Godin, Missaoui, and Alaoui (Godin et al., 1995) (GMA), Nourine and Raynaud (2002), Lindig and Datensystene (2000), and Valtchev et al. (2002) divide and conquer. Some are batch while others are incremental (i.e., insert object by object or item by item). The best known asymptotic complexity is $O(\boldsymbol{m}(\boldsymbol{m}+\boldsymbol{k})\boldsymbol{\ell})$, where $\boldsymbol{\ell} = |\mathcal{L}|$, $\boldsymbol{m} = |\mathcal{I}|$, and $\boldsymbol{k} = |\mathcal{O}|$. However, benchmarks have proven that asymptotic complexity may not be the best measurement for comparison. To date there is no known "best" algorithm. GMA, an incremental algorithm, is considered to be a good algorithm for data sets with density less than 0.10.

Most FCA construction algorithms construct a complete lattice whose concepts identify all closed item sets and not just those that are frequent. An iceberg lattice, on the other hand, is a concept lattice whose concepts are restricted to those where |O| meets a minimum support threshold. An iceberg lattice contains the necessary and sufficient information to extract association rules. Furthermore, the alternate notation of an iceberg lattice depicted in Figure 1.3 enables association rules to be directly read from iceberg

lattices. This form of iceberg concept lattice can be readily traversed to extract a basis of association rules that can be exploited by an end user. Only three algorithms to construct an iceberg lattice were found in literature; MAGALICE (Rouane et al., 2004), CHARM-L (Zaki, & Hsiao, 2005), and SPROUT (Choi, 2006). Given that an iceberg concept lattice provides an analysis tool to succinctly identify a basis of association rules, this study investigated additional algorithms to construct an iceberg concept lattice.

This report presented the development and analysis of the Quick Iceberg Concept Lattice (QuICL – pronounced kwi-kəl) algorithms. These algorithms provide incremental construction of a concept lattice along the lines of GMA, but approach the insertion process from the bottom of the lattice rather than top-down. The structure of the lattice is used to navigate to a point of change. Recursion is used instead of iteration to identify additional points of change and to enable connections between parent and child concepts. To support construction of iceberg lattices, the QuICL algorithms add data on an item by item basis and interchange the roles of the set of object identifiers (ids) and the set of items. These changes effectively invert the lattice. Furthermore, the lattice of the QuICL algorithms conforms to the notation of Figure 1.3.

QuICL has three derivations; Oid-Full, Oid-Less, and Oid-Trie. In the first derivation, all of the concepts in the concept lattice retain a complete list of the object ids (oids), hence the name "Oid-Full". While results of QuICL Oid-Full were promising for some data sets, the performance gains do not hold against others. An issue for QuICL Oid-Full is storage of the complete list of object ids in each concept. The same object ids can be repeated in multiple concepts. Thus, an alternate algorithm, termed Oid-Less, was derived to eliminate the permanent storage of object ids. QuICL Oid-Less is successful

in eliminating the object ids, however, this is achieved at the expense of considerable complexity. Therefore, the Oid-Trie derivation was developed as a compromise between QuICL Oid-Full and QuICL Oid-Less. Instead of eliminating the object ids, it utilizes a trie data structure to store the ids in a compressed structure, thereby reducing memory requirements. The QuICL algorithms were proved to be correct and validated by comparing the characteristics of the lattices generated by QuICL against lattices generated by other algorithms. The runtime complexity for the QuICL Oid-Full algorithm is postulated to be at least O($\ell\, d\, i$), but could approach O($\ell\, d^2\, c$) or O($\ell\, d\, d'\, i\, h$), where $\ell = |\mathcal{L}|$, $d = \deg_{avg}(\mathcal{L})$, i a density weighted mean on the cardinality of frequent item extents, c is a small fraction of $|\mathcal{O}|$ depending density, d' is a fraction of d depending on density, and h is a sub-linear function on the height of $\mathcal{L}$. An enhancement of the QuICL Oid-Trie algorithm eliminates O($d\, d'\, h$) from consideration. The memory complexity is postulated to be O($\ell\, d\, i$).

Evaluations of the QuICL algorithms against GMA, CHARM, CHARM-L, and MAGALICE were conducted using seven public data sets. The data sets include both sparse and dense data, and some contain items with large object sets. Before comparing QuICL against the other algorithms, experiments were conducted to determine if the order of item insertion has an effect on performance and memory usage. Best performance for QuICL Oid-Full and QuICL Oid-Trie was attained by incrementally inserting items in ascending support order. This order inhibits the initial growth of the lattice, thereby reducing the number of required intersections. For QuICL Oid-Less and GMA, descending support order provides the best performance on dense data sets. Other

factors in each algorithm contribute to this conflicting preference. Except for QuICL Oid-Less, the sort order had no effect on memory.

In comparing QuICL to CHARM, an FI miner, CHARM provides the best performance on most dense data sets. These results are expected since CHARM does not derive the upper covers and it uses a difference based representation for the sets of object ids. However, on sparse data sets, CHARM is outperformed by QuICL Oid-Full as the support is lowered. CHARM is also outperformed by QuICL Oid-Full on Mushroom, a dense data set, over all supports. The QuICL algorithms consume significantly more memory than CHARM on dense data sets and slightly more memory on sparse data sets. On dense data sets, QuICL Oid-Trie and QuICL Oid-Less derivations provide a reduction in memory usage over QuICL Oid-Full, but the reduction is not sufficient enough to match CHARM.

QuICL Oid-Full provided the best overall performance for constructing iceberg lattices. It outperforms GMA by an order of magnitude and MAGALICE by two orders of magnitude. It is only outperformed by CHARM-L on data sets containing items with large object sets, and for the sparse data sets at relatively high supports. However, on sparse data sets, the gain of CHARM-L is generally limited to a few seconds which quickly turns into a large loss as the support is lowered. At low supports, QuICL Oid-Full outperforms CHARM-L in excess of an order of magnitude on most data sets. QuICL Oid-Trie exhibits the near same runtime complexity as QuICL Oid-Full for all data sets with a small performance overhead. It provides a reasonable tradeoff between performance and memory. QuICL Oid-Less is QuICL's answer to handling data set that contains items with large object id sets. By intersecting concept sets instead of object id

sets, QuICL Oid-Less realizes a significant performance gain on such data sets and outperforms CHARM-L by more than a factor of two. With respect to memory usage, QuICL Oid-Full was the same as GMA on dense data sets, but provided 10% to 40% reduction on sparse data. QuICL Oid-Trie and QuICL Oid-Less derivations generally provide additional reduction in memory usage. When compared against CHARM-L, the QuICL algorithms provide similar reduction in memory usage on sparse data sets. However, on dense data sets CHARM-L consumes less memory. The difference based representation of the underlying CHARM algorithm is providing a benefit. QuICL Oid-Trie and QuICL Oid-Less do, however, challenge CHARM-L and at some supports provide a reduction in memory usage.

Empirical evidence supporting asymptotic runtime complexity for the O($\ell\, d\, i$) and O($\ell\, d^2\, c$) was provided. In all except sparse data, strong correlations between observed and calculated execution times were present for O($\ell\, d\, i$). The lack of correlation on sparse data does not necessarily disprove an O($\ell\, d\, i$) complexity. Instead, it indicates that the mean used for calculating i is not appropriate. The discrepancy between dense and sparse data does, however, indicate that density is a factor in computing the mean.

In conclusion, this study has met its objective to develop a lattice based algorithm whose overall performance is near the leading algorithms used for association rule mining. Furthermore, the constructed lattices are of the form whereby association rules can be directly read and a basis can be readily extracted. Therefore, the QuICL algorithms offer a significant contribution to association rule mining. Beyond this, the QuICL algorithms have proved to be very efficient, providing an order of magnitude gains over prior incremental lattice construction algorithm. For example, on the

T10I4D100k data set, GMA takes near 10,000 seconds where QuICL Oid-Full completes in less than 120 seconds. On Chess at $55\%_{supp}$, GMA is over 1,000 seconds where Oid Full is less than ten seconds. QuICL Oid-Full provides the best all around performance on both dense and sparse data. QuICL Oid-Trie provides a reasonable tradeoff between performance and memory, enabling it to create lattices for lower supports. QuICL Oid-Less addresses a special class of data sets that contain items with large object id sets. Therefore, the QuICL algorithms offer a significant contribution to formal concept analysis.

Epilogue

While this report presented the QuICL derivations in the order of QuICL Oid-Full, QuICL Oid-Less, and QuICL Oid-Trie, the QuICL Oid-Less derivation was actually developed first. At the start of this study, it was assumed that the storage of object ids within all concepts would exhaust available memory. Therefore, attention was focused on deriving an algorithm that used the compressed lattice structure. After attaining the best possible results through a sequence of enhancements, the early algorithms were reconstructed to confirm the preliminary timings for this report. When re-implementing Algorithm 3.5, an error was introduced. The call to clear the temporary set of pull-down object ids was omitted. On executing the re-implemented Algorithm 3.5, execution times of a few hours were expected. As a result of the omission the algorithm executed in 80 seconds, and it produced a correct lattice. Analysis revealed that the concepts retained the complete list of object ids. Thus, the QuICL Oid-Full algorithm was discovered. Additional enhancements reduced the time to those given in this report. This is a lesson learned. Test assumptions, they may lead to great discoveries. The QuICL Oid-Less derivation still had merit with a special class of data sets.

Appendix A

Implementation of the Modified GMA Algorithm

```
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * The Godin Missaoui Alaoui algorithm to construct a Concept Lattice.
 *
 * This is an implementation of algorithm one found in the article: "Incremental concept formation
 * algorithms based on Galois (concept) lattices", R. Godin, R. Missaoui, and H Alaoui.
 *
 * This implementation has swapped the roles of items and objects and has added a minimal support test.
 * As a result this algorithm constructs an iceberg concept lattice.
 *
 * A concept lattice consists of a bottom concept. Concepts
 * are discovered then added as ancestors using the method insert.
 *
 * @author David T. Smith
 */
public class GMAConceptLattice implements ConceptLattice {
	/**
	 * A concept within the lattice.  A concept contains a complete list of item ids and object ids together
	 * with parent and child lists.  The parents and children lists provide the edges between the concepts.
	 */
	public static class Concept {
		IntArray iids = new IntArray();;
		int[] oids;
		List<Concept> children = new ArrayList<Concept>();
		ArrayList<Concept> parents = new ArrayList<Concept>();

		public Concept(int iid, int[] oids) {
			addAttr(iid);
			this.oids = oids;
		}

		public void addAttr(int iid) {
			iids.add(iid);
		}
	}

	private int minSupport = 0;
	private Concept bottom = new Concept(-1, new int[0]);
	private List<Concept> allConcepts = new ArrayList<Concept>();
	private int[] intersectBuff;
	private ArrayList<ArrayList<Concept>> processedList = new ArrayList<ArrayList<Concept>>();

	/**
	 * Construct an empty concept lattice. The minimum support is specified.
	 *
	 * @param minSupport - the minimum support
```

```
 */
public GMAConceptLatticeAdd(int minSupport, int buffSize) {
	this.minSupport = minSupport;
	this.intersectBuff = new int[buffSize];
	allConcepts.add(bottom);
}

/**
 * The GMA incremental insert algorithm to insert a new item into the lattice
 * @param iid - item id to be added
 * @param oids - object ids of objects holding the item
 */
public void insert(int iid, int[] oids) {
	// Special case for an empty lattice - bottom concept is empty
	if (bottom.iids.size() == 0 && bottom.oids.length == 0) {
		// Just add the item ids and object ids to the empty bottom concept
		bottom.addAttr(iid);
		bottom.oids = oids;
		return;
	}

	// Union the oids with the bottom oids
	int inx1 = 0;
	int inx2 = 0;
	int usize = 0;
	while (inx1 < oids.length && inx2 < bottom.oids.length) {
		if (oids[inx1] == bottom.oids[inx2]) {
			intersectBuff[usize++] = oids[inx1];
			inx1++;
			inx2++;
		} else if (oids[inx1] < bottom.oids[inx2]) {
			intersectBuff[usize++] = oids[inx1++];
		} else {
			intersectBuff[usize++] = bottom.oids[inx2++];

		}
	}

	while (inx1 < oids.length) {
		intersectBuff[usize++] = oids[inx1++];
	}

	while (inx2 < bottom.oids.length) {
		intersectBuff[usize++] = bottom.oids[inx2++];
	}

	// Test if the oids contain ids that are do not exist in the lattice.
	if (usize > bottom.oids.length) {
		int[] noids = new int[usize];
		System.arraycopy(intersectBuff, 0, noids, 0, usize);
		if (bottom.iids.size() == 0) {
			bottom.oids = noids;
		} else {
			Concept newC = new Concept(-1, noids);
			allConcepts.add(newC);
			bottom.children.add(newC);
```

```
            newC.parents.add(bottom);
            bottom = newC;
        }
    }

    // Insure the process list has a bucket allocated for the current oids size
    while (processedList.size() > oids.length) {
        processedList.remove(processedList.size() - 1);
    }

    // Empty the processed list
    for (ArrayList<Concept> list : processedList) {
        list.clear();
    }

    processedList.ensureCapacity(oids.length);

    while (processedList.size() <= oids.length) {
        processedList.add(new ArrayList<Concept>());
    }

    // Process, in ascending order of support, all concepts in the current concept list
    Collections.sort(allConcepts, conceptComparator);

    int end = allConcepts.size();
    for (int i = 0; i < end; i++) {
        Concept c = allConcepts.get(i);
        int[] oids1 = c.oids;
        int[] oids2 = oids;
        inx1 = 0;
        inx2 = 0;
        int i1;
        int i2;
        boolean subset = true;
        boolean superset = true;
        int isize = 0;

        if (inx1 < oids1.length && inx2 < oids2.length) {
            i1 = oids1[inx1++];
            i2 = oids2[inx2++];
            for (;;) {
                if (i1 == i2) {
                    intersectBuff[isize++] = i1;
                    if (inx1 < oids1.length && inx2 < oids2.length) {
                        i1 = oids1[inx1++];
                        i2 = oids2[inx2++];
                    } else {
                        break;
                    }
                } else if (i1 < i2) {
                    subset = false;
                    if (inx1 < oids1.length) {
                        i1 = oids1[inx1++];
                    } else {
                        superset = false;
                        break;
```

```
                }
            } else {
                superset = false;
                if (inx2 < oids2.length) {
                    i2 = oids2[inx2++];
                } else {
                    subset = false;
                    break;
                }
            }
        }
    }

    if (isize < minSupport) {
        continue;
    }

    if (inx1 < oids1.length) {
        subset = false;
    }

    if (inx2 < oids2.length) {
        superset = false;
    }

    if (subset && superset) {
        c.addAttr(iid);
        return;
    }

    if (subset) {
        c.addAttr(iid);
        processedList.get(isize).add(c);
    } else if (isize >= minSupport){       // Additional test for min support threshold
        List<Concept> bkt = processedList.get(isize);
        boolean isGen = true;
        for (Concept p : bkt) {
            if (isize == p.oids.length) {
                int inx = 0;
                for (inx = 0; inx < isize && intersectBuff[inx] == p.oids[inx]; inx++) {
                }
                if (inx == isize) {        // equal?
                    isGen = false;
                    break;
                }
            }
        }

        if (isGen) {
            int[] noids = new int[isize];
            System.arraycopy(intersectBuff, 0, noids, 0, isize);
            Concept newC = new Concept(iid, noids);
            allConcepts.add(newC);
            processedList.get(isize).add(newC);

            newC.children.add(c);
```

```
                    c.parents.add(newC);

                    outer:  for (List<Concept> bkt2 : processedList) {
                        for (Concept p : bkt2) {
                            if (p.oids.length >= isize) {
                                break outer;
                            }

                            if (isSubset(p.oids, newC.oids)) {
                                boolean isParent = true;
                                for (Concept ch : p.children) {
                                    if (isSubset(ch.oids, newC.oids)) {
                                        isParent = false;
                                        break;
                                    }
                                }
                                if (isParent) {
                                    p.children.remove(c);
                                    c.parents.remove(p);
                                    p.children.add(newC);
                                    newC.parents.add(p);
                                }
                            }
                        }
                    }

                    if (isize == oids.length) {
                        return;
                    }
                }
            }
        }
    }

    /**
     * Test for subset
     * @param ids1
     * @param ids2
     * @return true if ids1 subset of ids2
     */
    private boolean isSubset(int[] ids1, int[] ids2) {
        int inx1 = ids1.length - 1;
        int inx2 = ids2.length - 1;
        int i1 = ids1[inx1];
        int i2 = ids2[inx2];

        for (;;) {
            if (inx1 > inx2) {
                return false;
            }
            if (i1 == i2) {
                if (inx1 == 0) {
                    return true;
                }
                i1 = ids1[--inx1];
                i2 = ids2[--inx2];
```

```
                } else if (i1 > i2) {
                    return false;
                } else {
                    if (inx2 == 0) {
                        return false;
                    }
                    i2 = ids2[--inx2];
                }
            }
        }

        /**
         * Comparator used to sort concepts in ascending order of support
         */
        private static Comparator<Concept> conceptComparator = new Comparator<Concept>() {
            public int compare(Concept o1, Concept o2) {
                return o1.oids.length - o2.oids.length;
            }
        };

        public int getNoConcepts() {
            return allConcepts.size();
        }
}
```

Appendix B

Implementation of the QuICL Oid-Full Algorithm

```
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * The QuICL Oid-Full algorithm to construct a Concept Lattice.
 *
 * A concept lattice consists of a bottom concept. Discovered concepts are then added as ancestors using
 * the method insert.
 *
 * @author David T. Smith
 */
public class OidfullConceptLattice implements ConceptLattice {
    /**
     * A concept within the lattice.  A concept contains a list of physical item ids and a complete list
     * of object ids together with parents lists.  The parents provide the edges between the concepts.
     */
    public static class Concept  {
        int[] aids;
        int[] oids;
        int[] intersectOids;
        List<Concept> parents;

        public Concept(int aid, int[] oids, int noParents) {
            addAttr(aid);
            this.oids = oids;
            parents = new ArrayList<Concept>(noParents);
        }

        public void addAttr(int aid) {
            if (aid < 0) {
                return;
            }
            if (aids == null) {
                aids = new int[1];
            } else {
                int[] naids = new int[aids.length + 1];
                System.arraycopy(aids, 0, naids, 0, aids.length);
                aids = naids;
            }
            aids[aids.length - 1] = aid;
        }
    }

    /**
     * A class defining a tuple that is placed into the ToProcessList.
     */
    private static class IntersectInfo {
```

```
        SetCmp type;
        Concept concept;

        public IntersectInfo(SetCmp type, Concept concept) {
            this.type = type;
            this.concept = concept;
        }
}

private int minSupport = 0;
private Concept bottom = new Concept(-1, null, 16);
private enum SetCmp { UNKNOWN, EQ, SUBSET, SUPERSET, INTERSECT };
private List<Concept> allConcepts = new ArrayList<Concept>();
private int[] intersectBuff;
private static final int[] emptyOids = new int[0];

/**
 * Construct an empty concept lattice. The minimum support is specified.
 *
 * @param minSupport - the minimum support
 * @param buffSize - the size for an intersection buffer
 */
public OidfullConceptLattice(int minSupport, int buffSize) {
        this.minSupport = minSupport;
        this.intersectBuff = new int[buffSize];
}

/**
 * Insert a new item into the concept lattice
 *
 * @param iid - item id to be added
 * @param oids - the list of object ids - passed as an int array
 */
public void insert(int iid, int[] oids) {
        if (oids.length >= minSupport) {
            insert(bottom, iid, oids);
            for (Concept c : allConcepts) {
                c. intersectOids = null;
            }
        }
}

/**
 * The QuICL Oid-Full incremental insert algorithm to insert a new item into the lattice
 *
 * @param baseC - concept above which a new concept is found or inserted
 * @param iid - item id to be added
 * @param oids - object ids of objects holding the item
 * @return the found or created concept
 */
private Concept insert(Concept bottom, int iid, int[] oids) {
        // create the ToProcessList to hold tuples
        List<IntersectInfo> toProcessList = new ArrayList<IntersectInfo>();

        for (Concept parentC : bottom.parents) {   // prepare-search phase
            // Intersect and compare oids w/testC.oids
```

```
if (parentC.intersectOids == null) {
	int[] poids = parentC.oids;
	int inx1 = 0;
	int inx2 = 0;
	int i1;
	int i2;
	int isize = 0;
	i1 = oids[inx1++];
	i2 = poids[inx2++];

	for (;;) {
		if (i1 == i2) {
			intersectBuff[isize++] = i1;
			if (inx1 < oids.length && inx2 < poids.length) {
				i1 = oids[inx1++];
				i2 = poids[inx2++];
			} else {
				break;
			}
		} else if (i1 < i2) {
			if (inx1 < oids.length) {
				i1 = oids[inx1++];
			} else {
				break;
			}
		} else {
			if (inx2 < poids.length) {
				i2 = poids[inx2++];
			} else {
				break;
			}
		}
	}

	// cache the result in the parent concept
	if (isize == oids.length) {
		parentC.intersectOids = oids;
	} else if (isize == parentC.oids.length) {
		parentC.intersectOids = parentC.oids;
	} else {
		if (isize < minSupport) {
			parentC.intersectOids = emptyOids;
		} else {
			parentC.intersectOids = new int[isize];
			System.arraycopy(intersectBuff, 0, parentC.intersectOids, 0, isize);
		}
	}
}

if (parentC.intersectOids.length < minSupport) {
	continue;
}

// process the outcome of the intersection
if (parentC.intersectOids.length == oids.length) {
	if (parentC.oids.length == parentC.intersectOids.length) {	// Equal
```

```
                    parentC.addAttr(iid);
                    return parentC;
                } else {                                                    // Subset
                    return insert(parentC, iid, oids);
                }
            } else {
                if (parentC.oids.length <= parentC.intersectOids.length) {      // Superset
                    toProcessList.add(new IntersectInfo(SetCmp.SUPERSET, parentC));
                } else {                                                    // Intersect
                    toProcessList.add(new IntersectInfo(SetCmp.INTERSECT, parentC));
                }
            }
        }

        purgeSubsets(toProcessList);

        Concept newC = new Concept(iid, oids, toProcessList.size());
        allConcepts.add(newC);

        for (IntersectInfo p : toProcessList) {   // link phase
            if (p.type == SetCmp.SUPERSET) {
                bottom.parents.remove(p.concept);
                newC.parents.add(p.concept);

            } else if (p.type == SetCmp.INTERSECT) {
                Concept parentC = insert(p.concept, -1, p.concept. intersectOids);
                    newC.parents.add(parentC);
            }
        }

        Collections.sort(newC.parents, conceptComparator);

        int inx = Collections.binarySearch(bottom.parents, newC, conceptComparator);
        if (inx < 0) {
            inx = -inx - 1;
        }

        bottom.parents.add(inx, newC);

        return newC;
    }

    /**
     * Comparator used to sort concepts in descending order of support
     */
    private static Comparator conceptComparator = new Comparator() {
        public int compare(Object o1, Object o2) {
            return ((Concept) o2).oids.length - ((Concept) o1).oids.length;
        }
    };

    /**
     * Purge tuples in the ToProcesList that have intersection sets that are subsets of other tuples.
     * Purged tuples are marked as UNKNOWN
     *
     * @param toProcessList
```

```
 */
private void purgeSubsets(List<IntersectInfo> toProcessList) {
      for (int i = 0; i < toProcessList.size() - 1; i++) {
            IntersectInfo interInfo1 = toProcessList.get(i);
            if (interInfo1.type != SetCmp.UNKNOWN) {
                  for (int j = i + 1; j < toProcessList.size(); j++) {
                        IntersectInfo interInfo2 = toProcessList.get(j);
                        if (interInfo2.type != SetCmp.UNKNOWN) {
                              int[] oids1 = interInfo1.concept.intersectOids;
                              int[] oids2 = interInfo2.concept.intersectOids;
                              int inx1 = oids1.length - 1;;
                              int inx2 = oids2.length - 1;;
                              int i1 = oids1[inx1];
                              int i2 = oids2[inx2];

                              boolean subset = true;
                              boolean superset = true;

                              if (interInfo1.type == SetCmp.INTERSECT) {
                                    for (;;) {
                                          if (inx1 > inx2) {
                                                subset = false;
                                                break;
                                          }
                                          if (i1 == i2) {
                                                if (inx1 == 0) {
                                                      break;
                                                }
                                                i1 = oids1[--inx1];
                                                i2 = oids2[--inx2];
                                          } else if (i1 > i2) {
                                                subset = false;
                                                break;
                                          } else {
                                                superset = false;
                                                if (inx2 == 0) {
                                                      subset = false;
                                                      break;
                                                }
                                                i2 = oids2[--inx2];
                                          }
                                    }
                                    if (subset) {
                                          interInfo1.type = SetCmp.UNKNOWN;
                                    }
                              } else {
                                    subset = false;
                              }

                              if (interInfo2.type == SetCmp.INTERSECT && superset && !subset) {
                                    for (;;) {
                                          if (inx1 < inx2) {
                                                superset = false;
                                                break;
                                          }
                                          if (i1 == i2) {
```

```
                                        if (inx2 == 0) {
                                            break;
                                        }
                                        i1 = oids1[--inx1];
                                        i2 = oids2[--inx2];
                                    } else if (i1 < i2) {
                                        superset = false;
                                        break;
                                    } else {
                                        if (inx1 == 0) {
                                            superset = false;
                                            break;
                                        }
                                        i1 = oids1[--inx1];
                                    }
                                }
                                if (superset) {
                                    interInfo2.type = SetCmp.UNKNOWN;

                                }
                            }
                        }
                    }
                }
            }

    public int getNoConcepts() {
        return allConcepts.size();
    }
}
```

Appendix C

Implementation of the QuICL Oid-Less Algorithm

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;

/**
 * The QuICL Oid-Less algorithm to construct a Concept Lattice.
 *
 * A concept lattice consists of a bottom concept. Concepts are discovered and added as ancestors
 * using the method insert.
 *
 * @author David T. Smith
 */
public class OidlessConceptLattice implements ConceptLattice {
	/**
	 * A concept within the lattice.  A concept contains a list of physical item ids, count of object ids,
	 * support, and other temporal fields used during item insertion.  The parents and children provide the
	 * edges between the concepts.
	 */
	public static class Concept implements Comparable {
		private static int nextConceptId = 0;
		private int id = nextConceptId++;
		private int[] aids;
		private int noOids;
		private int support;
		private ConceptArray parents= new ConceptArray();
		private ConceptArray children= new ConceptArray();
		private Concept[] intersectSupportConcepts;
		private int intersectSize = 0;
		private int noIntersectOids = 0;
		private IntArray intersectOids = null;
		private HasSupers hasSuper = HasSupers.UNKNOWN;
		private Concept adjusted;
		private static ConceptArray hasSuperProcessedConcepts = new ConceptArray();
		private static ConceptArray intersectProcessedConcepts = new ConceptArray();
		private static ConceptArray intersectBaseConcepts = new ConceptArray();
		private static ConceptArray adjustedFromConcepts = new ConceptArray();

		public Concept(int aid, int noOids, int support) {
			addAttr(aid);
			this.noOids = noOids;
			this.support = support;
		}

		public static void clearTemporalFields() {
			for (Concept concept : intersectProcessedConcepts) {
				concept.intersectSupportConcepts = null;
				concept.intersectSize = 0;
			}
```

```
        for (Concept concept : hasSuperProcessedConcepts) {
            concept.hasSuper = HasSupers.UNKNOWN;
            concept.intersectSize = 0;
        }

        for (Concept concept : intersectBaseConcepts) {
            concept.noIntersectOids = 0;
            concept.intersectOids = null;
        }

        for (Concept concept : adjustedFromConcepts) {
            concept.adjusted = null;
        }

        intersectProcessedConcepts.clear();
        hasSuperProcessedConcepts.clear();
        intersectBaseConcepts.clear();
        adjustedFromConcepts.clear();
    }

    public void addAttr(int aid) {
        if (aid < 0) {
            return;
        }
        if (aids == null) {
            aids = new int[1];
        } else {
            int[] naids = new int[aids.length + 1];
            System.arraycopy(aids, 0, naids, 0, aids.length);
            aids = naids;
        }
        aids[aids.length - 1] = aid;
    }

    public void subtractNoOids(int noOids) {
        this.noOids -= noOids;
    }

    public void subtractSupport(int adjustment) {
        support -= adjustment;
    }

    public void addIntersectOid(int oid) {
        if (intersectOids == null) {
            intersectOids = new IntArray();
            intersectBaseConcepts.add(this);
        }

        intersectOids.add(oid);
        noIntersectOids++;
    }

    public void setIntersectOids(IntArray intersectOids) {
        this.intersectOids = intersectOids;
        intersectBaseConcepts.add(this);
    }
```

```
public Concept[] getIntersectSupportConcepts() {
    if (intersectSupportConcepts == null) {
        intersectSize = 0;

        Concept[] newIntersectionConcepts = kwayIntersect(children);
        for (Concept c : newIntersectionConcepts) {
            intersectSize += c.noIntersectOids;
        }

        setIntersectSupportConcepts(newIntersectionConcepts);
    }
    return intersectSupportConcepts;
}

public void setIntersectSupportConcepts(Concept[] intersectionConcepts) {
    intersectProcessedConcepts.add(this);
    this.intersectSupportConcepts = intersectionConcepts;
}

public int getIntersectSize() {
    getIntersectSupportConcepts();
    return intersectSize;
}

public void setIntersectSize(int intersectSize) {
    this.intersectSize = intersectSize;
}

public Concept[] kwayIntersect(ConceptArray children) {
    Concept[][] iters = new Concept[children.size()][];
    int[] inxs = new int[children.size()];
    int i = 0;

    for (Concept child : children) {
        iters[i++] = child.getIntersectSupportConcepts();
    }

    int p1 = 0;
    int p2 = 1;
    int kwayBuffInx = 0;
    if (inxs[p1] < iters[p1].length) {
        int cid1 = iters[p1][inxs[p1]++].id;
        while (inxs[p2] < iters[p2].length) {
            int cid2 = iters[p2][inxs[p2]++].id;
            if (cid1 < cid2) {
                cid1 = cid2;
                p1 = p2;
                p2 = (p2 + 1) % iters.length;
            } else if (cid1 == cid2) {
                p2 = (p2 + 1) % iters.length;
                if (p1 == p2) {
                    Concept c = iters[p2][inxs[p2] - 1];
                    intersectBuff[kwayBuffInx++] = c;
                    p2 = (p2 + 1) % iters.length;
                }
```

```
                    }
                }
            }

            Concept[] intersectConcepts = new Concept[kwayBuffInx];
            System.arraycopy(intersectBuff, 0, intersectConcepts, 0, kwayBuffInx);

            return intersectConcepts;
        }

        public boolean hasSuperset() {
            if (hasSuper == HasSupers.UNKNOWN) {
                IntArray intersection = intersectOids;
                boolean hasSuperset = noOids > (intersection == null ? 0 : intersection.size());

                if (!hasSuperset) {
                    for (Concept parent : parents) {
                        if (parent.hasSuperset()) {
                            hasSuperset = true;
                            break;
                        }
                    }
                }

                if (hasSuperset) {
                    hasSuper = HasSupers.YES;
                } else {
                    hasSuper = HasSupers.NO;
                }

                hasSuperProcessedConcepts.add(this);
            }
            return hasSuper == HasSupers.YES;
        }

        public int compareTo(Object o) {
            return id - ((Concept) o).id;
        }

        public void setNoIntersectOids(int noIntersectOids) {
            this.noIntersectOids = noIntersectOids;
        }

        public void setInsersectSize(int intersectSize) {
            this.intersectSize = intersectSize;
        }

        public void setAdjusted(Concept generatedConcept) {
            this.adjusted = generatedConcept;
            adjustedFromConcepts.add(this);
        }
    }

    /**
     * A utility class for performance.  ConceptArray provides similar function ArrayList<Concept> with
    * a few performance enhancements (e.g., binary sort based removal).
```

```
 */
public static class ConceptArray extends FastArrayList<Concept>  {
      public ConceptArray(int noParents) {
            super(noParents);
      }

      public ConceptArray() {
            super();
      }

      public ConceptArray(Concept[] concepts) {
            super(concepts);
      }
}

/**
 * Class defining a tuple that is placed into the ToProcessList.
 */
private static class IntersectInfo {
      SetCmp type;
      Concept concept;
      Concept[] intersectSupportConcepts;

      public IntersectInfo(SetCmp type, Concept concept, Concept[] intersectSupportConcepts) {
            this.type = type;
            this.concept = concept;
            this.intersectSupportConcepts = intersectSupportConcepts;
      }
}

private enum SetCmp { UNKNOWN, EQ, SUBSET, SUPERSET, INTERSECT };
private enum HasSupers {UNKNOWN, YES, NO };

private ConceptArray oid2Concept = new ConceptArray();
private ConceptArray allDependents = new ConceptArray();
private ConceptArray allConcepts = new ConceptArray();
private Concept bottom = new Concept(-1, 0, Integer.MAX_VALUE, 0);
private static Concept[] intersectBuff;
private int minSupport = 0;
private HashMap<Concept, ConceptArray> supportCsMap =
                                                new HashMap<Concept, ConceptArray>();

private HashMap<Concept, ConceptArray> dependentCsMap =
                                                new HashMap<Concept, ConceptArray>();

/**
 * Construct an empty concept lattice. The minimum support is specified.
 *
 * @param minSupport - the minimum support
 */
public OidlessConceptLattice(int minSupport, int buffSize) {
      this.minSupport = minSupport;
      intersectBuff = new Concept[buffSize];
}

/**
```

```
 * Insert a new item into the concept lattice
 *
 * @param iid - item id to be added
 * @param oids - the list of object ids - passed as an int array
 */
public void insert(int aid, int[] oids) {
      if (oids.length >= minSupport) {
            Concept[] supportConcepts = intersectLattice(oids);

            Concept newC = insert(bottom, aid, oids.length, supportConcepts);

            if (newC.children.size() == 1 && !hasSupportCs(newC)) {
                  ConceptArray adjustedSupports = new ConceptArray(supportConcepts.length);

                  for (Concept support : supportConcepts) {
                        if (support.adjusted != null) {
                              support = support.adjusted;
                        }

                        adjustedSupports.add(support);
                        ConceptArray dependents = dependentCsMap.get(support);
                        if (dependents == null) {
                              dependents = new ConceptArray();
                              dependentCsMap.put(support, dependents);
                        }
                        dependents.add(newC);
                  }

                  adjustedSupports.sort();
                  supportCsMap.put(newC, adjustedSupports);
                  allDependents.add(newC);
            }
      }
}

/**
 * The QuICL Oid-Less incremental insert algorithm to insert a new item into the lattice
 *
 * @param baseC - concept above which a new concept is found or inserted
 * @param iid - item id to be added
 * @param support - the support for the new concept
 * @param supportConcepts - array of Concepts that are the supports.
 * @return the found or created concept
 */
private Concept insert(Concept baseC, int iid, int support, Concept[] supportConcepts) {
      // create the ToProcessList to hold tuples
      List<IntersectInfo> toProcessList = new ArrayList<IntersectInfo>();
      boolean hasIceberg = false;

      for (Concept parentC : baseC.parents) { // prepare-search phase
            if (parentC.support < minSupport) {       // test for iceberg concept
                  hasIceberg = true;
                  continue;
            }

            Concept[] intersectSupportConcepts = parentC.getIntersectSupportConcepts();
```

```
        if (intersectSupportConcepts.length == 0) {
            continue;
        }

        int isize = parentC.getIntersectSize();
        boolean hasSuper = parentC.hasSuperset();

        if (isize == support) {
            if (!hasSuper) {                                    // equal
                parentC.addAttr(iid);
                return parentC;
            } else {                                            // subset
                return insert(parentC, iid, support, supportConcepts);
            }
        } else {
            if (!hasSuper) {                                    // superset
                toProcessList.add(new IntersectInfo(SetCmp.SUPERSET, parentC,
                                                        intersectSupportConcepts));
            } else {                                            // intersect
                if (parentC.getIntersectSize() < minSupport) {
                    hasIceberg = true;
                } else {
                    toProcessList.add(new IntersectInfo(SetCmp.INTERSECT, parentC,
                                                        intersectSupportConcepts));
                }
            }
        }
    }

    purgeSubsets(toProcessList);

    Concept newC = new Concept(iid, baseC.noIntersectOids, support);
    allConcepts.add(newC);

    adjust(baseC, newC);

    for (IntersectInfo interInfo : toProcessList) {             // link phase
        if (interInfo.type == SetCmp.SUPERSET) {
            removeLink(interInfo.concept, baseC);
            addLink(interInfo.concept, newC);
        } else if (interInfo.type == SetCmp.INTERSECT) {
            int isize = interInfo.concept.getIntersectSize();
            Concept parentC = insert(interInfo.concept, -1, isize,
                                                    interInfo.intersectSupportConcepts);
            addLink(parentC, newC);
        }
    }

    if (hasIceberg) {                                           // iceberg processing
        Concept[][] iters = new Concept[newC.parents.size() + 1][];
        int[] inxs = new int[iters.length + 1];
        int i = 0;

        for (Concept t : newC.parents) {
```

```
                iters[i++] =  t.getIntersectSupportConcepts();
            }
            iters[i] = new Concept[] { baseC };

            int supportConceptInx = 0;
            outer: while (supportConceptInx < supportConcepts.length) {
                for (i = 0; i < iters.length; i++) {
                    for(;;) {
                        if (inxs[i] == iters[i].length) {
                            break;
                        }
                        int r = iters[i][inxs[i]].id - supportConcepts[supportConceptInx].id;
                        if (r < 0){
                            inxs[i]++;
                        } else if (r == 0) {
                            supportConceptInx++;
                            continue outer;
                        } else {
                            break;
                        }
                    }
                }
                icebergLink(supportConcepts[supportConceptInx++], newC);
            }
        }

        baseC.setAdjusted(newC);
        newC.parents.sort(conceptComparator);

        int inx = baseC.parents.binarySearch(newC, conceptComparator);
        if (inx < 0) {
            inx = -inx - 1;
        }

        addLink(newC, baseC, inx);

        return newC;
    }

    /**
     * Comparator to sort concepts in descending support order
     */
    private static Comparator<Concept> conceptComparator = new Comparator<Concept>() {
        public int compare(Concept o1, Concept o2) {
            return o2.support - o1.support;
        }
    };

    /**
     * Extract and link up iceberg concepts to a new concept
     *
     * @param supportC - concept that is/will become an iceberg concept
     * @param newC
     */
    private void icebergLink(Concept supportC, Concept newC) {
        if (supportC.adjusted != null) {
```

```
            addLink(supportC.adjusted, newC);
        } else if (supportC.support == supportC.noIntersectOids) {
            addLink(supportC, newC);
        } else {
            Concept icebergConcept = icebergInsert(supportC);
            adjust(supportC, icebergConcept);
            addLink(icebergConcept, newC);
        }
    }

    /**
     * Construct an iceberg concept from another concept
     *
     * @param fromC - concept from which an iceberg is extracted
     */
    private Concept icebergInsert(Concept fromC) {
        int isize = fromC.intersectOids.size();

        Concept icebergConcept = new Concept(-1, isize, isize, 0);

        if (fromC.support < minSupport) {
            for (Concept child : fromC.children) {    // split an iceberg concept
                addLink(icebergConcept, child);
            }
            fromC.subtractSupport(isize);
        } else {
            addLink(icebergConcept, fromC);          // extracting from a non iceberg concept
        }
        return icebergConcept;
    }

    /**
     * Perform a lattice intersection
     *
     * @param oids - object ids for the intersection
     * @return a list of concepts that are referenced by the object ids
     */
    public Concept[] intersectLattice(int[] oids) {
        Concept.clearTemporalFields();
        int buffInx = 0;
        for (int i = 0; i < oids.length; i++) {
            int oid = oids[i];
            Concept concept = getConcept(oid);
            if (concept == null) {
                if (bottom.noIntersectOids == 0) {
                    intersectBuff[buffInx++] = bottom;
                }
                bottom.addIntersectOid(oid);
            } else {
                if (concept.noIntersectOids == 0) {
                    intersectBuff[buffInx++] = concept;
                }
                concept.addIntersectOid(oid);
            }
        }
```

```
        Concept[] supportConcepts = new Concept[buffInx];
        System.arraycopy(intersectBuff, 0, supportConcepts, 0, buffInx);
        Arrays.sort(supportConcepts);

        getSupportsForDependents();

        return supportConcepts;
    }

    /**
     * Pull-down the support concept list for concepts that have an intersection with a new item
     */
    private void getSupportsForDependents() {
        for (Concept dependent : allDependents) {
            ConceptArray supports = supportCsMap.get(dependent);
            int buffInx = 0;
            int intersectSize = 0;

            for (Concept supportC : supports) {
                if (supportC.noIntersectOids > 0) {
                    intersectBuff[buffInx++] = supportC;
                    intersectSize += supportC.noIntersectOids;
                }
            }

            Concept[] intersectSupportConcepts = new Concept[buffInx];
            System.arraycopy(intersectBuff, 0, intersectSupportConcepts, 0, buffInx);
            Arrays.sort(intersectSupportConcepts);
            dependent.setIntersectSupportConcepts(intersectSupportConcepts);
            dependent.setInsersectSize(intersectSize);
        }
    }

    /**
     * Adjust the temporal field to account for a new concept
     *
     * @param fromC
     * @param newC
     */
    public void adjust(Concept fromC, Concept newC) {
        fromC.setAdjusted(newC);

        setOid2Concept(fromC.intersectOids, newC);

        if (fromC.intersectSupportConcepts != null) {
            Concept[] newIntersectionSupportConcepts =
                                            new Concept[fromC.intersectSupportConcepts.length];
            System.arraycopy(fromC.intersectSupportConcepts, 0,
                             newIntersectionSupportConcepts, 0, fromC.intersectSupportConcepts.length);
            newC.setIntersectSupportConcepts(newIntersectionSupportConcepts);
            newC.setIntersectSize(fromC.getIntersectSize());
        }

        if (fromC.intersectOids == null) {
            return;
        }
```

```
        fromC.subtractNoOids(fromC.noIntersectOids);

        if (hasDependentCs(fromC)) {
            for (Concept dependent : getDependentCs(fromC)) {
                addSupportC(dependent, newC);
                addDependentC(newC, dependent);
                if (fromC.noOids == 0) {
                    removeSupportC(dependent, fromC);
                }
            }
            if (fromC.noOids == 0) {
                removeDependentCs(fromC);
            }
        }

        newC.setIntersectOids(fromC.intersectOids.cloneArray());
        newC.setNoIntersectOids(fromC.noIntersectOids);
        IntArray bottomIntersect = fromC.intersectOids;
        bottomIntersect.clear();
    }

    /**
     * Get a concept using the object id to concept map
     *
     * @param oid the object id
     * @return concept holding the object id
     */
    public Concept getConcept(int oid) {
        if (oid < oid2Concept.size()) {
            return oid2Concept.get(oid);
        } else {
            return null;
        }
    }

    /**
     * Update entries in the object id to concept map
     *
     * @param oids - list of object ids held by a new concept
     * @param newC - new concept holding the ids
     */
    public void setOid2Concept(IntArray oids, Concept newC) {
        if (oids != null) {
            for (IntIterator iter = oids.iterator(); iter.hasNext();) {
                int oid = iter.next();
                while (oid >= oid2Concept.size()) {
                    oid2Concept.add(null);
                }
                oid2Concept.set(oid, newC);
            }
        }
    }

    /**
     * Purge tuples in the ToProcesList that have concept support sets that are subsets of other concept
```

```
 * support sets of other tuples
 *
 * @param toProcessList
 */
private void purgeSubsets(List<IntersectInfo> toProcessList) {
      for (int i = 0; i < toProcessList.size() - 1; i++) {
            IntersectInfo interInfo1 = toProcessList.get(i);
            if (interInfo1.type != SetCmp.UNKNOWN) {
                  for (int j = i + 1; j < toProcessList.size(); j++) {
                        IntersectInfo interInfo2 = toProcessList.get(j);
                        if (interInfo2.type != SetCmp.UNKNOWN) {
                              Concept[] oids1 = interInfo1.intersectSupportConcepts;
                              Concept[] oids2 = interInfo2.intersectSupportConcepts;
                              int inx1 = oids1.length - 1;;
                              int inx2 = oids2.length - 1;;
                              int i1 = oids1[inx1].id;
                              int i2 = oids2[inx2].id;
                              boolean subset = true;
                              boolean superset = true;

                              if (interInfo1.type == SetCmp.INTERSECT) {
                                    for (;;) {
                                          if (inx1 > inx2) {
                                                subset = false;
                                                break;
                                          }
                                          if (i1 == i2) {
                                                if (inx1 == 0) {
                                                      break;
                                                }
                                                i1 = oids1[--inx1].id;
                                                i2 = oids2[--inx2].id;
                                          } else if (i1 > i2) {
                                                subset = false;
                                                break;
                                          } else {
                                                superset = false;
                                                if (inx2 == 0) {
                                                      subset = false;
                                                      break;
                                                }
                                                i2 = oids2[--inx2].id;
                                          }
                                    }
                                    if (subset) {
                                          interInfo1.type = SetCmp.UNKNOWN;

                                    }
                              } else {
                                    subset = false;
                              }

                              if (interInfo2.type == SetCmp.INTERSECT && superset && !subset) {
                                    for (;;) {
                                          if (inx1 < inx2) {
                                                superset = false;
```

```
                                        break;
                                    }
                                    if (i1 == i2) {
                                        if (inx2 == 0) {
                                            break;
                                        }
                                        i1 = oids1[--inx1].id;
                                        i2 = oids2[--inx2].id;
                                    } else if (i1 < i2) {
                                        superset = false;
                                        break;
                                    } else {
                                        if (inx1 == 0) {
                                            superset = false;
                                            break;
                                        }
                                        i1 = oids1[--inx1].id;
                                    }
                                }
                                if (superset) {
                                    interInfo2.type = SetCmp.UNKNOWN;

                                }
                            }
                        }
                    }
                }
            }

    private void addLink(Concept parent, Concept child) {
        addLink(parent, child, -1);
    }

    private void addLink(Concept parent, Concept child, int inx) {
        parent.children.add(child);
        if (inx < 0) {
            child.parents.add(parent);
        } else {
            child.parents.add(inx, parent);
        }
        if (getSupportCs(parent) != null && parent.children.size() > 1) {
            for (Concept supportConcept : getSupportCs(parent)) {
                removeDependentC(supportConcept, parent);
            }
            removeSupports(parent);
            allDependents.removeSorted(parent);
        }
    }

    private void removeLink(Concept parent, Concept child) {
        parent.children.removeSorted(child);
        child.parents.remove(parent);
    }

    public boolean hasDependentCs(Concept concept) {
```

```
        return dependentCsMap.get(concept) != null;
    }

    public ConceptArray getDependentCs(Concept concept) {
        return dependentCsMap.get(concept);
    }

    public void addDependentC(Concept toConcept, Concept dependent) {
        ConceptArray dependentCs = dependentCsMap.get(toConcept);

        if (dependentCs == null) {
            dependentCs = new ConceptArray();
            dependentCsMap.put(toConcept, dependentCs);
        }

        dependentCs.add(dependent);
    }

    public boolean hasSupportCs(Concept concept) {
        return supportCsMap.get(concept) != null;
    }

    public void addSupportC(Concept toConcept, Concept concept) {
        ConceptArray supportCs = supportCsMap.get(toConcept);
        supportCs.add(concept);
    }

    public void removeSupportC(Concept fromConcept, Concept concept) {
        ConceptArray supports = supportCsMap.get(fromConcept);
        supports.removeSorted(concept);
    }

    public ConceptArray getSupportCs(Concept concept) {
        return supportCsMap.get(concept);
    }

    public void removeSupports(Concept fromConcept) {
        supportCsMap.remove(fromConcept);
    }

    public void removeDependentC(Concept fromConcept, Concept concept) {
        ConceptArray dependents = dependentCsMap.get(fromConcept);
        dependents.removeSorted(concept);
    }

    public void removeDependentCs(Concept fromConcept) {
        dependentCsMap.remove(fromConcept);
    }

    public int getNoConcepts() {
        return allConcepts.size();
    }
}
```

Appendix D

Implementation of the QuICL Oid-Trie Algorithm

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;

/**
 * The QuICL Oid-Trie algorithm to construct a Concept Lattice.  This is the Oid-Full algorithm
 * only using a trie data structure to represent object ids.
 *
 * A concept lattice consists of a bottom concept. Concepts are discovered and added as ancestors
 * using the method insert.
 *
 * @author David T. Smith
 */
public class OidTrieConceptLattice implements ConceptLattice {

    /**
     * A position in the Trie representing an object set.  A TriePos references a TrieNode
     * and an offset within that node which is the last object in the object set.
     */
    private static class TriePos {
        TrieNode node;
        int      offset;

        TriePos(TrieNode node, int offset) {
            this.node = node;
            this.offset = offset;
        }

        int getLength() {
            return node.length + offset + 1;
        }

        public int hashCode() {
            return node.hashCode() + offset;
        }

        public boolean equals(Object o) {
            TriePos r = (TriePos) o;
            return r.node == node && r.offset == offset;
        }
    }

    /**
     * A Trie child reference.  A TrieChildRef is used a key to lookup a child trie node.
     */
    private static class TrieChildRef extends TriePos {
        int baseOid;

        TrieChildRef(TriePos pos, int baseOid) {
```

```
            super(pos.node, pos.offset);
            this.baseOid = baseOid;
        }

        public TrieChildRef(TrieNode node, int lastOffset, int baseOid) {
            super(node, lastOffset);
            this.baseOid = baseOid;
        }

        public int hashCode() {
            return node.hashCode() + (offset + 1) * (baseOid + 1);
        }

        public boolean equals(Object o) {
            TrieChildRef r = (TrieChildRef) o;
            return r.node == node && r.offset == offset && r.baseOid == baseOid;
        }
    }

    /**
     * A node within the Trie.  Each node is linked to child nodes using a hashtable.  A TrieChildRef
     * will serve as the key.
     */
    private static class TrieNode {
        int oids[];
        TriePos parent;
        int length;

        TrieNode(int oids[], int offset, TriePos child, int length) {
            this.oids = new int[oids.length - offset];
            System.arraycopy(oids, offset, this.oids, 0, oids.length - offset);
            this.parent = child;
            this.length = length;
        }

        TriePos insert(int[] oids, int offset) {
            TriePos r =  insert(this, oids, offset);
            return r;
        }

        static TriePos insert(TrieNode node, int[] oids, int offset) {
            int nodeOffset = 0;
            int lastOffset = 0;
            while (nodeOffset < node.oids.length && offset < oids.length &&
                                                    node.oids[nodeOffset] == oids[offset]) {
                lastOffset = nodeOffset;
                nodeOffset++;
                offset++;
            }

            if (offset == oids.length) {
                return new TriePos(node, lastOffset);
            }

            TrieChildRef key = new TrieChildRef(node, lastOffset, oids[offset]);
```

```
            TrieNode parent = trieChildren.get(key);

            if (parent != null) {
                return insert(parent, oids, offset);
            }

            parent = new TrieNode(oids, offset, new TriePos(node, lastOffset),
                                                                    node.length + nodeOffset);
            trieChildren.put(key, parent);
            TriePos pos = new TriePos(parent, oids.length - offset - 1);
            return pos;
        }
    }

    /**
     * A concept within the lattice. A concept contains a reference into a trie representing a list of object
     * ids, list of item ids, and a list of parents. The parents provide the edges between the concepts.
     */
    public static class Concept {
        int[] aids;
        TriePos oids;
        TriePos intersectOids;
        List<Concept> parents;

        public Concept(int aid, TriePos oids, int noParents) {
            addAttr(aid);
            this.oids = oids;
            this. intersectOids = oids;
            parents = new ArrayList<Concept>(noParents);
        }

        public void addAttr(int aid) {
            if (aid < 0) {
                return;
            }
            if (aids == null) {
                aids = new int[1];
            } else {
                int[] naids = new int[aids.length + 1];
                System.arraycopy(aids, 0, naids, 0, aids.length);
                aids = naids;
            }
            aids[aids.length - 1] = aid;
        }
    }

    /**
     * A class defining a tuple that is placed into the ToProcessList.
     */
    private static class IntersectInfo {
        SetCmp type;
        Concept concept;

        public IntersectInfo(SetCmp type, Concept concept) {
            this.type = type;
            this.concept = concept;
```

```
        }
}


private int minSupport = 0;
private Concept bottom = new Concept(-1, null, 16);
private TrieNode root = new TrieNode(new int[0], 0, null, 0);
private enum SetCmp {UNKNOWN, EQ, SUBSET, SUPERSET, INTERSECT};
private List<Concept> allConcepts = new ArrayList<Concept>();
private int[] intersectBuff;
private static HashMap<TrieChildRef, TrieNode> trieChildren =
                                                    new HashMap<TrieChildRef, TrieNode>();
private static HashMap<TriePos, Concept> trieConcepts = new HashMap<TriePos, Concept>();

/**
 * Construct an empty concept lattice. The minimum support is specified.
 *
 * @param minSupport - the minimum support
 * @param buffSize -the size for an intersection buffer
 */
public OidTrieConceptLattice(int minSupport, int buffSize) {
        this.minSupport = minSupport;
        this.intersectBuff = new int[buffSize];
}

/**
 * Insert a new item into the concept lattice.
 *
 * @param iid - item id to be added
 * @param oids - the list of object ids - passed as an int array
 */
public void insert(int iid, int[] oids) {
        if (oids.length >= minSupport) {
                insert(bottom, iid, root.insert(oids, 0));
                for (Concept c : allConcepts) {
                        c. intersectOids = null;
                }
        }
}

/**
 * The QuICL Oid-Trie incremental insert algorithm to insert a new item into the lattice
 *
 * @param baseC - concept above which a new concept is found or inserted
 * @param iid - item id to be added
 * @param oids - a trie position representing a set of object ids of objects holding the item
 * @return the found or created concept
 */
private Concept insert(Concept bottom, int iid, TriePos oids) {
        Concept trieConcept = trieConcepts.get(oids);
        if (trieConcept != null) {
                trieConcept.addAttr(iid);
                return trieConcept;
        }

        // create the ToProcessList to hold tuples
```

```
List<IntersectInfo> toProcessList = new ArrayList<IntersectInfo>();

for (Concept parentC : bottom.parents) { // prepare-search phase
    if (parentC.intersectOids == null) {
        // Intersect and compare oids w/parentC.oids
        TrieNode trie1 = oids.node;
        TrieNode trie2 = parentC.oids.node;
        int inx1 = oids.offset;
        int inx2 = parentC.oids.offset;
        int[] oids1 = trie1.oids;
        int[] oids2 = trie2.oids;
        int i1 = oids1[inx1];
        int i2 = oids2[inx2];
        int ipos = intersectBuff.length;

        if (trie1 != trie2) {
            for (;;) {
                if (i1 == i2) {
                    intersectBuff[--ipos] = i1;
                    if (inx1 == 0) {
                        inx1 = trie1.parent.offset;
                        trie1 = trie1.parent.node;
                        if (trie1 == root) {
                            if (inx2 > 0 || trie2.parent.node != root) {
                            }
                            break;
                        }
                        oids1 = trie1.oids;
                        i1 = oids1[inx1];
                        if (inx2 > 0 && trie1 == trie2) {
                            i2 = oids2[--inx2];
                            break;
                        }
                    } else {
                        i1 = oids1[--inx1];
                    }
                    if (inx2 == 0) {
                        inx2 = trie2.parent.offset;
                        trie2 = trie2.parent.node;
                        if (trie2 == root) {
                            break;
                        }
                        oids2 = trie2.oids;
                        i2 = oids2[inx2];
                        if (trie1 == trie2) {
                            break;
                        }
                    } else {
                        i2 = oids2[--inx2];
                    }
                } else if (i1 > i2) {
                    if (inx1 == 0) {
                        inx1 = trie1.parent.offset;
                        trie1 = trie1.parent.node;
                        if (trie1 == root) {
                            break;
```

```
                    }
                    oids1 = trie1.oids;
                    i1 = oids1[inx1];
                    if (trie1 == trie2) {
                        break;
                    }
                } else {
                    i1 = oids1[--inx1];
                }
            } else {
                if (inx2 == 0) {
                    inx2 = trie2.parent.offset;
                    trie2 = trie2.parent.node;
                    if (trie2 == root) {
                        break;
                    }
                    oids2 = trie2.oids;
                    i2 = oids2[inx2];
                    if (trie1 == trie2) {
                        break;
                    }
                } else {
                    i2 = oids2[--inx2];
                }
            }
        }
    }

    int isize;

    if (trie1 == trie2) {
        if (inx1 < inx2) {
            inx2 = inx1;
        } else if (inx1 > inx2) {
            inx1 = inx2;
        }
        isize= trie1.length + inx1 + 1 + (intersectBuff.length - ipos);
    }else {
        isize = intersectBuff.length - ipos;
    }

    // cache the result in the parent concept
    if (isize == oids.getLength()) {
        parentC.intersectOids = oids;
    } else if (isize == parentC.oids.getLength()) {
        parentC.intersectOids = parentC.oids;
    } else {
        if (isize < minSupport) {
            parentC.intersectOids = rootPos;
        } else {
            if (trie1 == trie2) {
                while (inx1 >= 0) {
                    intersectBuff[--ipos] = trie1.oids[inx1--];
                }
                parentC.intersectOids = trie1.insert(intersectBuff, ipos);
            } else {
```

```
                                parentC.intersectOids = root.insert(intersectBuff, ipos);

                            }
                        }
                    }
                }

                if (parentC.intersectOids.getLength() < minSupport) {
                    continue;
                }

                // process the outcome of the intersection
                if (parentC.intersectOids.getLength() == oids.getLength()) {
                    if (parentC.oids.getLength() != parentC.intersectOids.getLength()) {     // Subset
                        return insert(parentC, iid, oids);
                    }
                } else {
                    if (parentC.oids.getLength() <= parentC.intersectOids.getLength()) {     // Superset
                        toProcessList.add(new IntersectInfo(SetCmp.SUPERSET, parentC));
                    } else {                // Intersect
                        toProcessList.add(new IntersectInfo(SetCmp.INTERSECT, parentC));
                    }
                }
            }
        }

        /**
         * Comparator used to sort concepts in descending order of support
         */
        private static Comparator conceptComparator = new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Concept) o2).oids.getLength() - ((Concept) o1).oids.getLength();
            }
        };

        /**
         * Purge tuples in the ToProcesList that have intersection sets that are subsets of other intersection
         * sets of other tuples
         *
         * @param toProcessList
         */
        private void purgeSubsets(List<IntersectInfo> toProcessList) {
            for (int i = 0; i < toProcessList.size() - 1; i++) {
                IntersectInfo interInfo1 = toProcessList.get(i);
                if (interInfo1.type != SetCmp.UNKNOWN) {
                    for (int j = i + 1; j < toProcessList.size(); j++) {
                        IntersectInfo interInfo2 = toProcessList.get(j);
                        if (interInfo2.type != SetCmp.UNKNOWN) {
                            TrieNode trie1 = interInfo1.concept.intersectOids.node;
                            TrieNode trie2 = interInfo2.concept.intersectOids.node;
                            int inx1 = interInfo1.concept.intersectOids.offset;
                            int inx2 = interInfo2.concept.intersectOids.offset;
                            int[] oids1 = trie1.oids;
                            int[] oids2 = trie2.oids;
                            int i1 = oids1[inx1];
                            int i2 = oids2[inx2];
```

```
boolean subset = true;
boolean superset = true;

if (interInfo1.type == SetCmp.INTERSECT) {
      if (trie1 != trie2) {
            for (;;) {
                  if (trie1.length + inx1 > trie2.length + inx2) {
                        subset = false;
                        break;
                  }
                  if (i1 == i2) {
                        if (inx1 == 0) {
                              if (trie1.parent.node == root) {
                                    break;
                              }
                              inx1 = trie1.parent.offset;
                              trie1 = trie1.parent.node;
                              oids1 = trie1.oids;
                              i1 = oids1[inx1];

                              if (inx2 > 0 && trie1 == trie2) {
                                    i2 = oids2[--inx2];
                                    break;
                              }
                        } else {
                              i1 = oids1[--inx1];;
                        }
                        if (inx2 == 0) {
                              inx2 = trie2.parent.offset;
                              trie2 = trie2.parent.node;
                              oids2 = trie2.oids;

                              i2 = oids2[inx2];
                              if (trie1 == trie2) {
                                    break;
                              }
                        } else {
                              i2 = oids2[--inx2];
                        }
                  } else if (i1 > i2) {
                        subset = false;
                        break;
                  } else {
                        superset = false;
                        if (inx2 == 0) {
                              if (trie2.parent.node == root) {
                                    subset = false;
                                    break;
                              }

                              inx2 = trie2.parent.offset;
                              trie2 = trie2.parent.node;
                              oids2 = trie2.oids;
                              i2 = oids2[inx2];
```

```
                        if (trie1 == trie2) {
                            break;
                        }
                    } else {
                        i2 = oids2[--inx2];
                    }
                }
            }
        }
        if (trie1 == trie2 && inx1 > inx2) {
            subset = false;
        }

        if (subset) {
            interInfo1.type = SetCmp.UNKNOWN;

        }
    } else {
        subset = false;
    }

    if (interInfo2.type == SetCmp.INTERSECT && superset && !subset) {
        if (trie1 != trie2) {
            for (;;) {
                if (trie1.length + inx1 < trie2.length + inx2) {
                    superset = false;
                    break;
                }
                if (i1 == i2) {
                    if (inx2 == 0) {
                        if (trie2.parent.node == root) {
                            break;
                        }
                        inx2 = trie2.parent.offset;
                        trie2 = trie2.parent.node;
                        oids2 = trie2.oids;

                        i2 = oids2[inx2];

                        if (inx1 > 0 && trie1 == trie2) {
                            i1 = oids1[--inx1];
                            break;
                        }
                    } else {
                        i2 = oids2[--inx2];
                    }
                    if (inx1 == 0) {
                        inx1 = trie1.parent.offset;
                        trie1 = trie1.parent.node;
                        oids1 = trie1.oids;
                        i1 = oids1[inx1];
                        if (trie1 == trie2) {
                            break;
                        }

                    } else {
```

```
                                        i1 = oids1[--inx1];
                                    }

                                } else if (i1 < i2) {
                                    superset = false;
                                    break;
                                } else {
                                    if (inx1 == 0) {
                                        if (trie1.parent.node == root) {
                                            superset = false;
                                            break;
                                        }

                                        inx1 = trie1.parent.offset;
                                        trie1 = trie1.parent.node;
                                        oids1 = trie1.oids;
                                        i1 = oids1[inx1];

                                        if (trie1 == trie2) {
                                            break;
                                        }
                                    } else {
                                        i1 = oids1[--inx1];
                                    }
                                }
                            }
                        }
                        if (trie1 == trie2 && inx1 < inx2) {
                            superset = false;
                        }
                        if (superset) {
                            interInfo2.type = SetCmp.UNKNOWN;

                        }
                    }
                }
            }
        }
    }

    public int getNoConcepts() {
        return allConcepts.size();
    }
}
```

Appendix E

Implementation of Supporting Functions

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.io.RandomAccessFile;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * A test harness to run the QuICL algorithms and the iceberg modified GMA algorithm.
 * This harness includes function to transpose a data set from horizontal to vertical representation.
 *
 * Synopsis:
 *
 * java BuildLattice [options]
 *
 * where options are:
 *
 *     -i [inputFile] - the input file
 *     -f [format] - the file format:
 *               ibm - horizontal text base series of numbers, tid oid no_ids [iid ...] per row
 *               txt - horizontal text base series of numbers [iid ...] per row
 *               vert - vertical binary series of numbers produced by this harnes
 *               (default ibm)
 *          -a [algorithm] - the selected agorithm:
 *               oidfull - QuICL Oid-Full
 *               oidless - QuICL Oid-Less
 *               oidtrie - QuICL Oid-Trie
 *               gma - GMA
*                transpose – only create transpose file from horizontal input
 *               (default oidfull)
 *          -g [support] - the relative support, a real number between 1 and 0 (default 0).
*           -G [support] - the absolute support.
 *          -r [transposeFile] - a intermediate file in vertical format produced by this harness
 *          -s [sortOrder] - items sorted by support (ignored if format is vert)
 *               asc - ascending support order
 *               desc -descending support order
 *               (default unsorted)
 *          -n [no rows] – Process only the first n objects in the data set (default all objects).
 *          -m [no columns] – Process only the first m items in data set (default all items).
 *
 * @author David T. Smith
 */
public class BuildLattice {
      // internal interface used to hide disk vs. memory based input
      public interface RandomAccess {
            void seek(long offset) throws IOException;
            void writeInt(int i) throws IOException;
            int readInt() throws IOException;
```

```
        void setLength(long l) throws IOException;
        String readLine() throws IOException;
        long length() throws IOException;
        long getFilePointer() throws IOException;
}

// disk based implementation of RandomAccess
private static class RandomAccessFileI extends RandomAccessFile implements RandomAccess {
        public RandomAccessFileI(String fileName, String mode) throws FileNotFoundException {
                super(fileName, mode);
        }
}

// memory based implementation of RandomAccess
private static class RandomAccessByteI implements RandomAccess {
        byte[] buffer = new byte[0];
        int pos = 0;

        public void seek(long offset) throws IOException {
                pos = (int) offset;
        }

        public void writeInt(int i) throws IOException {
                for (int k = 3; k >= 0; k--) {
                        write((i >> (8 * k)) & 0xFF);
                }
        }

        public int readInt() throws IOException {
                return ((read() << 24) | (read() << 16) | (read() << 8) | read());
        }

        public int read() throws IOException {
                if (pos < buffer.length) {
                        return buffer[pos++] & 0XFF;
                } else if (pos == buffer.length){
                        pos++;
                        return -1;
                } else {
                        throw new IOException("Read past EOF");
                }
        }

        public void write(int i)  {
                buffer[pos++] = (byte) i;
        }

        public void setLength(long l) throws IOException {
                byte[] newBuffer = new byte[(int) l];
                System.arraycopy(buffer, 0, newBuffer, 0, Math.min((int) l,
                                buffer.length));
                buffer = newBuffer;
        }

        public String readLine() throws IOException {
                StringBuffer input = new StringBuffer();
```

```
            int c;

            loop: for(;;) {
                switch (c = read()) {
                case -1:
                case '\n':
                    break loop;
                case '\r':
                    long cur = pos;
                    if ((read()) != '\n') {
                        seek(cur);
                    }
                    break loop;
                default:
                    input.append((char) c);
                    break;
                }
            }

            if ((c == -1) && (input.length() == 0)) {
                return null;
            }

            return input.toString();
        }

        public long length() throws IOException {
            return buffer.length;
        }

        public long getFilePointer() throws IOException {
            return pos;
        }
    }

    /**
     * Internal class to transpose a horizontal file to vertical format
     */
    private static class TransposeDB {
        private static final int bufferSize = 1024;

        private static class OidWriter {
            RandomAccess out;
            int aid;
            int offset;
            int support;
            int[] oidBuffer = new int[bufferSize];
            int size = 0;

            public OidWriter(int aid, RandomAccess out) {
                this.aid = aid;
                this.out = out;
            }

            public void incrSupport() {
                support++;
```

```
    }

    public int getSize() {
        return (2 + support) * (Integer.SIZE / Byte.SIZE);
    }

    public int setOffset(int offset) throws IOException {
        this.offset = offset;
        out.seek(offset);
        out.writeInt(aid);
        this.offset += Integer.SIZE / Byte.SIZE;
        out.writeInt(support);
        this.offset += Integer.SIZE / Byte.SIZE;
        return this.offset + (support * (Integer.SIZE / Byte.SIZE));
    }

    public void appendOid(int oid) throws IOException {
        if (size == bufferSize) {
            flush();
        }
        oidBuffer[size++] = oid;
    }

    public void flush() throws IOException {
        if (size > 0) {
            out.seek(offset);
            for (int i = 0; i < size; i++) {
                out.writeInt(oidBuffer[i]);
            }
            offset += size * (Integer.SIZE / Byte.SIZE);
            size = 0;
        }
    }
}

ArrayList<OidWriter> oidWriters = new ArrayList<OidWriter>();
ArrayList<Integer> oidWritersIndex = new ArrayList<Integer>();
private boolean sortAsc;
private boolean sortDesc;
private int prefix;
private int oidInx;
private int nCols;
private int nRows;
private boolean hasTid;
private boolean hasOid;
private boolean hasCount;
private int countInx;

public TransposeDB(boolean hasTid, boolean hasOid, boolean hasCount, boolean sortAsc,
                                    boolean sortDesc, int nCols, int nRows) throws IOException {
    this.hasTid = hasTid;
    this.hasOid = hasOid;
    this.hasCount = hasCount;
    this.sortAsc = sortAsc;
    this.sortDesc = sortDesc;
    this.nCols = nCols;
```

```
        this.nRows = nRows;
        prefix = 0;
        countInx = -1;
        oidInx = -1;

        if (hasTid) {
            prefix++;
        }
        if (hasOid) {
            oidInx = prefix;
            prefix++;
        }
        if (hasCount) {
            countInx = prefix;
            prefix++;
        }
    }

    public void rotate(RandomAccess in, RandomAccess out) throws IOException {
        out.setLength(0);
        prepVertical(in, out);
        in.seek(0);
        writeVertical(in);
    }

    private void prepVertical(RandomAccess in, RandomAccess out) throws IOException {
        String line;
        int noid = 0;
        int nRows = 0;
        while ((line = in.readLine()) != null) {
            if (this.nRows >= 0 && nRows >= this.nRows) {
                break;
            }
            nRows++;
            line = line.replaceAll("[ ]+"," ");
            String[] parts = line.split("[, ]");
            if (oidInx < 0) {
               noid++;
            } else {
                int oid = Integer.parseInt(parts[oidInx]);
                if (oid > noid) {
                    noid = oid;
                }
            }
            for (int i = prefix; i < parts.length; i++) {
                if (nCols > 0 && i - prefix >= nCols) {
                    break;
                }
                String part = parts[i];
                int aid = Integer.parseInt(part);
                while (oidWriters.size() <= aid) {
                    oidWriters.add(null);
                }

                OidWriter oidWriter = oidWriters.get(aid);
                if (oidWriter == null) {
```

```
                oidWriter = new OidWriter(aid, out);
                oidWriters.set(aid, oidWriter);
            }

            oidWriter.incrSupport();
        }
}

for (int i = 0; i < oidWriters.size(); i++) {
        oidWritersIndex.add(i);
}

if (sortAsc) {
Collections.sort(oidWritersIndex, new Comparator() {
        public int compare(Object o1, Object o2) {
            int inx1 = (Integer) o1;
            int inx2 = (Integer) o2;
            OidWriter oidWriter1 = oidWriters.get(inx1);
            OidWriter oidWriter2 = oidWriters.get(inx2);
            if (oidWriter2 == null) {
                return 1;
            }
            if (oidWriter1 == null) {
                return -1;
            }
            return oidWriter1.support - oidWriter2.support;
        }
});
}

if (sortDesc) {
        Collections.sort(oidWritersIndex, new Comparator() {
            public int compare(Object o1, Object o2) {
                int inx1 = (Integer) o1;
                int inx2 = (Integer) o2;
                OidWriter oidWriter1 = oidWriters.get(inx1);
                OidWriter oidWriter2 = oidWriters.get(inx2);
                if (oidWriter1 == null) {
                    return 1;
                }
                if (oidWriter2 == null) {
                    return -1;
                }
                return oidWriter2.support - oidWriter1.support;
            }
        });

}

 long fileSize = 2 * (Integer.SIZE / Byte.SIZE);
for (int oidWriterIndex : oidWritersIndex) {
        OidWriter oidWriter = oidWriters.get(oidWriterIndex);
        if (oidWriter != null) {
            fileSize += oidWriter.getSize();
        }
}
```

```
            out.setLength(fileSize);

            int offset = 0;
            out.seek(offset);
            out.writeInt(noid);
            out.seek(offset);
            int i = out.readInt();
            offset += Integer.SIZE / Byte.SIZE;
            out.writeInt(oidWriters.size() - 1);
            offset += Integer.SIZE / Byte.SIZE;

            for (int oidWriterIndex : oidWritersIndex) {
                OidWriter oidWriter = oidWriters.get(oidWriterIndex);
                if (oidWriter != null) {
                    offset = oidWriter.setOffset(offset);
                }
            }
        }

        private void writeVertical(RandomAccess in) throws IOException {
            String line;
            int oid = 0;
            int nRows = 0;
            while ((line = in.readLine()) != null) {
                if (this.nRows >= 0 && nRows >= this.nRows) {
                    break;
                }
                nRows++;
                line = line.replaceAll("[ ]+"," ");
                String[] parts = line.split("[, ]");
                if (oidInx < 0) {
                   oid++;
                } else {
                    oid = Integer.parseInt(parts[oidInx]);
                }
                for (int i = prefix; i < parts.length; i++) {
                    if (nCols > 0 && i - prefix >= nCols) {
                        break;
                    }
                    int aid = Integer.parseInt(parts[i]);
                    OidWriter oidWriter = oidWriters.get(aid);
                    if (oidWriter == null) {
                        System.out.println("error");
                    }
                    oidWriter.appendOid(oid);
                }
            }

            for (OidWriter oidWriter : oidWriters) {
                if (oidWriter != null) {
                    oidWriter.flush();
                }
            }
        }
    }
```

```
public static void main(String[] args) throws IOException {
    ParmParser parmParser = new ParmParser(args, "i: a:r:s:n:m:f:g:G ");

    String inFileName = null;
    String rotateFileName = null;
    String sort = "none";
    String format = "vert";
    String algo = "oidfull";
    boolean sortDesc = false;
    boolean sortAsc = false;
    boolean hasOid = true;
    boolean hasTid = true;
    int     nCols = -1;
    int     nRows = -1;
    float   relativeSupport = 0;
    int     absSupport = 0;

    int c;
    while ((c = parmParser.getopt()) != -1) {
        switch (c) {
        case 'f':
            format = parmParser.getOptarg();
            if (format.equals("ibm")) {
                hasTid = true;
                hasOid = true;
                hasCount = true;
            }
            if (format.equals("txt")) {
                hasTid = false;
                hasOid = false;
                hasCount = false;
            }
            if (format.equals("vert")) {
                hasTid = false;
                hasOid = false;
                hasCount = false;
            }
            break;
        case 'i':
            inFileName = parmParser.getOptarg();
            break;
        case 'r':
            rotateFileName = parmParser.getOptarg();
            break;
        case 'a':
            algo = parmParser.getOptarg();
            break;
        case 's':
            sort = parmParser.getOptarg();
            if (sort.equals("asc")) {
                sortAsc = true;
            }
            if (sort.equals("desc")) {
                sortDesc = true;
            }
```

```
            break;
        case 'g':
            relativeSupport = Float.parseFloat(parmParser.getOptarg());
            break;
        case 'G':
            absSupport = Integer.parseInt(parmParser.getOptarg());
            break;
        case 'n':
            nCols = Integer.parseInt(parmParser.getOptarg());
            break;
        case 'm':
            nRows = Integer.parseInt(parmParser.getOptarg());
            break;
        }
}

RandomAccess in;
RandomAccess rotate;

in = new RandomAccessFileI(inFileName,"r");

if (!format.equals("vert")) {
        if (rotateFileName == null) {
            rotate = new RandomAccessByteI();
        } else {
            rotate = new RandomAccessFileI(rotateFileName, "rw");
        }
        TransposeDB rotator = new TransposeDB(hasTid, hasOid, hasCount, sortAsc,
                sortDesc, nCols, nRows);
        rotator.rotate(in, rotate);
        in = rotate;
        in.seek(0);
}

long latTime = 0;;

int noids = in.readInt();
int naids = in.readInt();

int minSupport = absSupport;

if (minSupport == 0) {
        minSupport= (int) (noids * relativeSupport + .5);
}

if (minSupport == 0) {
        minSupport = 1;
}

ConceptLattice cl = null;

if (algo.equals("gma ")) {
        cl = new GMAConceptLattice(minSupport, noids);
} else if (algo.equals("oidfull")) {
        cl = new OidfullConceptLattice(minSupport, noids);
} else if (algo.equals("oidless ")) {
```

```
            cl = new OidlessConceptLattice(minSupport, noids);
        } else if (algo.equals("oidtrie")) {
            cl = new OidTrieConceptLattice(minSupport, noids);
        } else if (algo.equals("transpose")) {
            System.exit(0);
        }

        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);

        while (in.getFilePointer() < in.length()) {
            markTime = System.currentTimeMillis();

            int aid = in.readInt();
            int support = in.readInt();

            if (support < minSupport) {
                in.seek(in.getFilePointer() + (support * 4));
                continue;
            }

            int[] oids = new int[support];
            for (int i = 0; i < support; i++) {
                int oid = in.readInt();
                oids[i] = oid;
            }

            long markTime = System.currentTimeMillis();
            cl.insert(aid, oids);
            latTime+= System.currentTimeMillis() - markTime;
        }

        double time = (System.currentTimeMillis() - startBuildTime) / 1000.0;
        System.out.print(algo + "\t" + inFileName + "\t" + sort + "\t" + relativeSupport + "\t" +
                                                cl.getNoConcepts() + "\t" + (latTime/1000.0));
    }
                                                    }
```

Appendix F

Empirical Data in Support of Algorithm Validity

	Min Supp	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
Chess	95%	74	74	74	74	74	75	74
	90%	503	503	503	503	503	504	499
	85%	1,885	1,885	1,885	1,885	1,885	1,886	1,886
	80%	5,083	5,083	5,083	5,083	5,083	5,084	5,084
	75%	11,525	11,525	11,525	11,525	11,525	11,526	11,526
	70%	23,991	23,991	23,991	23,991	23,991	23,992	
	65%	49,240	49,240	49,240	49,240	49,240	49,241	
	60%	98,392	98,392	98,392	98,392	98,392	98,393	
	55%	192,863	192,863	192,863	192,863	192,863	192,864	
	50%		369,450	369,450	369,450	369,450		
	45%		707,964		707,964	707,964		
	40%				1,366,833	1,366,833		
Mushroom	50%	45	45	45	45	45	45	45
	40%	140	140	140	140	140	140	140
	30%	427	427	427	427	427	427	427
	20%	1,197	1,197	1,197	1,197	1,197	1,197	1,197
	10%	4,897	4,897	4,897	4,897	4,897	4,897	4,885
	5%	12,854	12,854	12,854	12,854	12,854	12,854	12,843
	1%	51,672	51,672	51,672	51,672	51,672	51,672	51,640
	0%	238,709	238,709	238,709	238,709	238,709	238,709	238,709
Pumsb	95%	110	110	110	110	110	111	111
	90%	1,466	1,466	1,466	1,466	1,466	1,467	
	85%	8,513	8,513	8,513	8,513	8,513	8,514	
	80%			33,295	33,295	33,295		
	75%			101,047	101,047	101,047		
	70%			241,258	241,258	241,258		
	65%			496,069	496,069	496,069		
	60%				1,074,627	1,074,627		
Pumsb*	50%	248	248	248	248	248	249	253
	45%	713	713	713	713	713	714	
	40%	2,610	2,610	2,610	2,610	2,610	2,611	
	35%	6,133	6,133	6,133	6,133	6,133	6,134	
	30%	16,154	16,154	16,154	16,154	16,154	16,155	
	25%			42,756	42,756	42,756		
	20%				122,262	122,262		

Table F.1: Algorithm validity as assessed by number of concepts. Highlighted values are considered to be in error.

	Min Supp	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
T10I4D100k	2.000%	155	155	155	155	155	156	156
	1.000%	385	385	385	385	385	386	386
	0.500%	1,073	1,073	1,073	1,073	1,073	1,074	1,073
	0.300%	4,509	4,509	4,509	4,509	4,509	4,510	4,509
	0.100%	26,806	26,806	26,806	26,806	26,806	26,807	26,564
	0.050%	46,993	46,993	46,993	46,993	46,993	46,994	46,253
	0.030%	71,265	71,265	71,265	71,265	71,265	71,266	69,117
	0.010%	283,397	283,397	282,397	283,397	283,397	283,398	
	0.005%	769,777	769,777	769,777	769,777	769,777	769,778	
	0.000%	2,347,374	2,347,374	2,347,374	2,347,374	2,347,374	2,347,375	
T20I10D10k	5.00%	72	72	72	72	72	73	73
	3.00%	389	389	389	389	389	390	390
	1.00%	5,582	5,582	5,582	5,582	5,582	5,583	5,451
	0.50%	23,394	23,394	23,394	23,394	23,394	23,395	22,538
	0.30%	44,925	44,925	44,925	44,925	44,925	44,926	44,926
	0.10%	209,436	209,436	209,436	209,436	209,436	209,437	176,749
	0.05%	576,021	576,021	576,021	576,021	576,021	576,022	
	0.03%	1,438,054	1,438,054	1,438,054	1,438,054	1,438,054	1,438,055	
	0.00%	2,557,928	2,557,928	2,557,928	2,557,928	2,557,928	2,557,929	
T20I20D100k	3.00%	19	19	19	19	19	20	20
	2.00%	143	143	143	143	143	144	144
	1.00%	5,256	5,256	5,256	5,256	5,256	5,257	5,257
	0.50%	27,067	27,067	27,067	27,067	27,067	27,068	
	0.30%	72,640	72,640	72,640	72,640	72,640	72,641	
	0.10%	150,970	150,970	150,970	150,970	150,970	150,971	
	0.05%	212,765	212,765	212,765	212,765	212,765	212,766	
	0.03%	461,138	461,138	461,138	461,138	461,138	461,139	
	0.01%	3,519,933	3,519,933	3,519,933	3,518,933		3,519,933	

Table F.1 continued: Algorithm validity as assessed by number of concepts. Highlighted values are considered to be in error.

	Min Supp	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
Chess	95%	2.6400	2.6400	2.6400		2.6400	2.6400	
	90%	3.6329	3.6329	3.6329		3.6329	3.6329	
	85%	4.4019	4.4019	4.4019		4.4019	4.4019	
	80%	5.0313	5.0313	5.0313		5.0313	5.0313	
	75%	5.4893	5.4893	5.4893		5.4893	5.4893	
	70%	5.8376	5.8376	5.8376		5.8376	5.8376	
	65%	6.1719	6.1719	6.1719		6.1719	6.1719	
	60%	6.5147	6.5147	6.5147		6.5147	6.5147	
	55%	6.8544	6.8544	6.8544		6.8544	6.8544	
	50%							
	45%							
	40%							
Mushroom	50%	1.9348	1.9348	1.9348		1.9348	1.9348	
	40%	2.3121	2.3121	2.3121		2.3121	2.3121	
	30%	2.9977	2.9977	2.9977		2.9977	2.9977	
	20%	3.3222	3.3222	3.3222		3.3222	3.3222	
	10%	3.8365	3.8365	3.8365		3.8365	3.8365	
	5%	4.1670	4.1670	4.1670		4.1670	4.1670	
	1%	4.7521	4.7521	4.7521		4.7521	4.7521	
	0%	5.7093	5.7093	5.7093		5.7093	5.7093	
Pumsb	95%	2.5135	2.5135	2.5135		2.5135	2.5135	
	90%	4.0130	4.0130	4.0130		4.0130	4.0130	
	85%	5.1671	5.1671	5.1671		5.1671	5.1671	
	80%			6.1234		6.1234		
	75%			7.0167		7.0167		
	70%			7.8129		7.8129		
	65%			8.3051		8.3051		
	60%							
Pumsb*	50%	2.8233	2.8233	2.8233		2.8233	2.8233	
	45%	3.3768	3.3768	3.3768		3.3768	3.3768	
	40%	4.2237	4.2237	4.2237		4.2237	4.2237	
	35%	4.6920	4.6920	4.6920		4.6920	4.6920	
	30%	5.1352	5.1352	5.1352		5.1352	5.1352	
	25%			5.5662		5.5662		
	20%					5.9408		

Table F.2: Algorithm validity as assessed by average degree.

	Min Supp	Oid-Full	Oid-Trie	Oid-Less	CHARM	CHARM-L	GMA	MAGALICE
T10I4D100k	2.000%	0.9936	0.9936	0.9936		0.9936	0.9936	
	1.000%	1.0259	1.0259	1.0259		1.0259	1.0259	
	0.500%	1.6760	1.6760	1.6760		1.6760	1.6760	
	0.300%	2.5656	2.5656	2.5656		2.5656	2.5656	
	0.100%	3.2665	3.2665	3.2665		3.2665	3.2665	
	0.050%	3.0998	3.0998	3.0998		3.0998	3.0998	
	0.030%	2.8808	2.8808	2.8808		2.8808	2.8808	
	0.010%	2.8089	2.8089	2.8089		2.8089	2.8089	
	0.005%	2.9980	2.9980	2.9980		2.9980	2.9980	
	0.000%	4.2880	4.2880	4.2880		4.2880	4.2880	
T25I10D10k	5.00%	0.9863	0.9863	0.9863		0.9863	0.9863	
	3.00%	0.9974	0.9974	0.9974		0.9974	0.9974	
	1.00%	3.5809	3.5809	3.5809		3.5809	3.5809	
	0.50%	3.6799	3.6799	3.6799		3.6799	3.6799	
	0.30%	3.5860	3.5860	3.5860		3.5860	3.5860	
	0.10%	2.6348	2.6348	2.6348		2.6348	2.6348	
	0.05%	2.7358	2.7358	2.7358		2.7358	2.7358	
	0.03%	3.0100	3.0100	3.0100		3.0100	3.0100	
	0.00%	4.2975	4.2975	4.2975		4.2975	4.2975	
T25I20D100k	3.00%	0.9500	0.9500	0.9500		0.9500	0.9500	
	2.00%	1.0903	1.0903	1.0903		1.0903	1.0903	
	1.00%	3.5273	3.5273	3.5273		3.5273	3.5273	
	0.50%	4.0862	4.0862	4.0862		4.0862	4.0862	
	0.30%	4.7434	4.7434	4.7434		4.7434	4.7434	
	0.10%	4.6427	4.6427	4.6427		4.6427	4.6427	
	0.05%	4.5055	4.5055	4.5055		4.5055	4.5055	
	0.03%	4.1401	4.1401	4.1401		4.1401	4.1401	
	0.01%	3.6720	3.6720	3.6720			3.6720	

Table F.2 continued: Algorithm validity as assessed by average degree.

Appendix G

Effect of Item Sort Order on Lattice Growth

N	Ascending Insertion			Descending Insertion		
	Item	|Item O|	$\|\mathcal{L}\|$	Item	|Item O|	$\|\mathcal{L}\|$
1	50	1,975	1	58	3,195	1
2	19	1,980	2	52	3,185	3
3	68	1,984	3	29	3,181	7
4	70	2,007	4	40	3,170	15
5	15	2,026	5	60	3,149	31
6	11	2,129	6	36	3,099	63
7	38	2,196	7	7	3,076	127
8	27	2,205	8	62	3,060	191
9	54	2,216	9	34	3,040	383
10	21	2,225	10	56	3,021	767
11	72	2,345	11	66	3,021	1,151
12	74	2,407	13	48	3,013	2,303
13	17	2,500	15	5	2,971	4,607
14	31	2,526	18	9	2,874	8,063
15	46	2,556	21	25	2,860	16,071
16	44	2,612	26	3	2,839	21,319
17	64	2,631	32	42	2,714	36,203
18	42	2,714	46	64	2,631	47,537
19	3	2,839	69	44	2,612	61,462
20	25	2,860	99	46	2,556	72,178
21	9	2,874	140	31	2,526	80,468
22	5	2,971	219	17	2,500	87,314
23	48	3,013	349	74	2,407	92,913
24	56	3,021	568	72	2,345	93,407
25	66	3,021	925	21	2,225	94,768
26	34	3,040	1,600	54	2,216	96,010
27	62	3,060	2,737	27	2,205	96,972
28	7	3,076	4,626	38	2,196	97,867
29	36	3,099	8,182	11	2,129	98,176
30	60	3,149	12,484	15	2,026	98,264
31	40	3,170	22,000	70	2,007	98,316
32	29	3,181	35,377	68	1,984	98,383
33	52	3,185	69,518	19	1,980	98,383
34	58	3,195	98,392	50	1,975	98,392

Table G.1: Effect of sort order on lattice growth using Chess data set at $60\%_{supp}$. Item and |Item $\mathcal{O}$| is the item id and size of the item's extent of the N^{th} item. $|\mathcal{L}|$ is the size of lattice after insertion of the 1 through N^{th} item of the given sort order.

N	Ascending Insertion			Descending Insertion		
	Item	|Item O|	$\|\mathcal{L}\|$	Item	|Item O|	$\|\mathcal{L}\|$
1	7032	15,223	1	7072	38,749	1
2	4526	15,463	2	161	36,856	3
3	7046	16,487	3	197	36,746	7
4	4799	16,666	4	4502	36,492	15
5	2354	16,736	5	84	36,475	23
6	7052	17,230	7	4499	36,386	47
7	7026	17,486	9	168	36,267	63
8	167	18,513	10	4933	35,782	112
9	0	18,629	11	4937	35,387	210
10	6857	19,226	12	4496	34,214	418
11	4953	19,349	13	277	34,042	605
12	5946	19,349	13	4493	32,200	851
13	6856	19,349	13	4503	29,661	927
14	4786	19,414	14	4798	29,632	959
15	70	19,930	15	4413	29,349	1,459
16	7036	21,490	20	2297	29,189	1,674
17	2299	21,636	21	7057	28,619	2,318
18	155	21,690	22	4807	27,370	2,331
19	4525	21,851	28	4833	27,370	2,331
20	4680	21,916	40	6867	26,769	3,674
21	4780	21,916	40	4527	26,481	3,682
22	4518	22,007	63	4627	26,481	3,709
23	6869	22,277	67	4727	26,481	3,709
24	6922	22,277	67	4785	26,481	3,709
25	66	22,339	68	2301	25,362	3,771
26	7022	22,502	74	2401	25,362	3,771
27	2300	23,684	76	15	24,822	3,908
28	7042	24,150	83	4946	24,445	6,338
29	111	24,206	84	14	24,224	6,353
30	252	24,206	84	163	24,224	6,353
31	14	24,224	85	111	24,206	6,372
32	163	24,224	85	252	24,206	6,372
33	4946	24,445	103	7042	24,150	6,398
34	15	24,822	104	2300	23,684	6,493
35	2301	25,362	106	7022	22,502	6,539

Table G.2: Effect of sort order on lattice growth using Pumsb* data set at $30\%_{supp}$. Item and |Item $\mathcal{O}$| is the item id and size of the item's extent of the N^{th} item. $|\mathcal{L}|$ is the size of lattice after insertion of the 1 through N^{th} item of the given sort order.

N	Ascending Insertion			Descending Insertion		
	Item	|Item O|	|$\mathcal{L}$|	Item	|Item O|	|$\mathcal{L}$|
36	2401	25,362	106	66	22,339	6,721
37	4527	26,481	125	6869	22,277	6,776
38	4627	26,481	153	6922	22,277	6,776
39	4727	26,481	153	4518	22,007	7,100
40	4785	26,481	153	4680	21,916	8,761
41	6867	26,769	205	4780	21,916	8,761
42	4807	27,370	230	4525	21,851	11,130
43	4833	27,370	230	155	21,690	11,235
44	7057	28,619	273	2299	21,636	11,337
45	2297	29,189	282	7036	21,490	12,593
46	4413	29,349	314	70	19,930	12,613
47	4798	29,632	343	4786	19,414	13,671
48	4503	29,661	385	4953	19,349	13,685
49	4493	32,200	553	5946	19,349	13,685
50	277	34,042	740	6856	19,349	13,685
51	4496	34,214	881	6857	19,226	13,885
52	4937	35,387	1,106	0	18,629	13,908
53	4933	35,782	1,416	167	18,513	13,910
54	168	36,267	2,596	7026	17,486	14,900
55	4499	36,386	4,928	7052	17,230	14,940
56	84	36,475	6,396	2354	16,736	14,942
57	4502	36,492	12,245	4799	16,666	15,303
58	197	36,746	12,825	7046	16,487	16,104
59	161	36,856	14,395	4526	15,463	16,110
60	7072	38,749	16,154	7032	15,223	16,154

Table G.2 continued: Effect of sort order on lattice growth using Pumsb* data set at $30\%_{supp}$. Item and |Item $\mathcal{O}$| is the item id and size of the item's extent of the N^{th} item. |$\mathcal{L}$| is the size of lattice after insertion of the 1 through N^{th} item of the given sort order.

Appendix H

Size of the QuICL, GMA, and CHARM-L Data Elements

Table H.1 provides the memory consumption of the data elements in the QuICL, GMA and CHARM-L implementations. These are based upon examination of the sources and applying the Java memory sizes given in Table H.2

Data structure element	Memory in bytes
Object id	4
Item in QuICL lattice	4
Item in GMA and CHARM lattice Item references are stored in an ArrayList. Therefore a factor of 1.5 is applied to account for unused capacity	6
Parent child link in QuICL Oid-Full or Oid-Trie lattice Parent child links are stored in an ArrayList. Therefore a factor of 1.5 is applied to account for unused capacity. Only a single reference is used to traverse from child to parent.	6
Parent child link in QuICL Oid-Less, GMA, or CHARM-L lattice Parent child links are stored in an ArrayList. Therefore a factor of 1.5 is applied to account for unused capacity. Furthermore, two references are used for each to enable bi-directional traversal.	12
Concept in QuICL Oid-Full lattice As determined by: 3 × Object reference + 1 × ArrayList overhead + 1 × Object overhead + 2 × array overhead.	124
Concept in QuICL Oid-Trie lattice (includes TrieNode overhead) Assume ratio of TrieNode to Concept is near 1.0 : 5 × int + 9 × Object reference + 7 × Object overhead + 2 × array overhead + 1 × ArrayList overhead. The seven objects are: a Concept, two TriePos, TrieNode, TrieChildRef, and two HashMapEntry.	216
Concept in QuICL Oid-Less lattice As determined by: 5 × int + 7 × Object reference + 1 × Object overhead + 2 × array overhead + 3 × ArrayList overhead.	320
Concept in GMA lattice As determined by: 4 × Object reference + 1 × Object overhead + 1 × array overhead + 3 × ArrayList overhead.	276
Concept in CHARM-L lattice As determined by: 2 × int + 1 × long + 3 × Object reference + 1 × Object overhead + 4 × ArrayList overhead. One ArrayList is a List of ArrayLists. At least one ArrayList is assumed to be present in the List.	356

Table H.1: Memory consumption of QuICL, GMA, and CHARM-L data elements.

Java primitive or data structure	Memory in bytes
int	4
long	8
Object reference	4
array overhead	12
Object overhead	8
ArrayList overhead (or similar structure) Value is based on assumption that a large number of ArrayLists will have a maximum size over the course of execution that is less than the default initial capacity.	80

Table H.2: Memory consumption of Java data elements.

Appendix I

Calculated Memory Consumption

	Min Supp	Calculated Memory Consumption of QuICL Oid-Full in MBs (% of total)										Actual
		Concepts		Object Ids		Items		P-C Links		Total		
Chess	85%	0	(1%)	21	(99%)	0	(0%)	0	(0%)	22		30
	75%	1	(1%)	119	(98%)	0	(0%)	0	(0%)	120		138
	65%	6	(1%)	450	(98%)	0	(0%)	2	(0%)	458		492
	55%	24	(2%)	1533	(98%)	0	(0%)	8	(1%)	1564		1656
Mushroom	50%	0	(1%)	1	(99%)	0	(0%)	0	(0%)	1		8
	30%	0	(1%)	6	(99%)	0	(0%)	0	(0%)	6		13
	10%	1	(2%)	28	(97%)	0	(0%)	0	(0%)	29		42
	0%	30	(24%)	88	(70%)	0	(0%)	8	(7%)	126		153
Pumsb	95%	0	(0%)	21	(100%)	0	(0%)	0	(0%)	21		29
	90%	0	(0%)	265	(100%)	0	(0%)	0	(0%)	265		287
	80%	1	(0%)	1467	(100%)	0	(0%)	0	(0%)	1468		1557
Pimsb*	50%	0	(0%)	28	(100%)	0	(0%)	0	(0%)	28		38
	40%	0	(0%)	229	(100%)	0	(0%)	0	(0%)	229		263
	30%	2	(0%)	1114	(100%)	0	(0%)	0	(0%)	1116		1276
T10I4D100k	0.500%	0	(2%)	5	(97%)	0	(0%)	0	(0%)	6		13
	0.100%	3	(11%)	27	(87%)	0	(0%)	1	(2%)	30		40
	0.050%	6	(15%)	32	(83%)	0	(0%)	1	(2%)	39		48
	0.010%	35	(40%)	48	(55%)	0	(0%)	5	(5%)	88		116
	0.005%	95	(56%)	61	(36%)	0	(0%)	14	(8%)	170		230
	0.000%	291	(68%)	77	(18%)	0	(0%)	60	(14%)	428		546
T25I10D10k	1.000%	1	(17%)	3	(80%)	0	(0%)	0	(3%)	4		11
	0.500%	3	(26%)	8	(69%)	0	(0%)	1	(5%)	11		18
	0.100%	26	(53%)	20	(40%)	0	(0%)	3	(7%)	49		74
	0.050%	71	(65%)	28	(26%)	0	(0%)	9	(9%)	109		151
	0.000%	317	(73%)	49	(11%)	0	(0%)	66	(15%)	432		583
T25I20D100k	1.000%	1	(2%)	27	(97%)	0	(0%)	0	(0%)	28		36
	0.500%	3	(4%)	87	(96%)	0	(0%)	1	(1%)	91		105
	0.100%	19	(8%)	216	(90%)	0	(0%)	4	(2%)	239		263
	0.050%	26	(10%)	232	(88%)	0	(0%)	6	(2%)	264		287
	0.010%	436	(49%)	458	(51%)	0	(0%)	0	(0%)	895		1094

Table I.1: Calculated memory consumption of QuICL Oid-Full lattice. Actual is the memory usage reported by the "Mem Usage" field of the Windows Task Manager upon termination of algorithm execution.

	Min Supp	Calculated Memory Consumption of QuICL Oid-Trie in MBs (% of total)										Actual
		Concepts		Object Ids		Items		P-C Links		Total		
Chess	85%	0	(10%)	4	(89%)	0	(0%)	0	(1%)	4		11
	75%	2	(11%)	20	(88%)	0	(0%)	0	(2%)	23		32
	65%	11	(12%)	79	(86%)	0	(0%)	2	(2%)	92		108
	55%	42	(13%)	278	(85%)	0	(0%)	8	(2%)	327		367
Mushroom	50%	0	(2%)	0	(98%)	0	(0%)	0	(0%)	0		7
	30%	0	(4%)	2	(95%)	0	(0%)	0	(0%)	2		11
	10%	1	(9%)	10	(89%)	0	(0%)	0	(1%)	11		19
	0%	52	(53%)	38	(39%)	0	(0%)	8	(8%)	98		114
Pumsb	95%	0	(0%)	21	(100%)	0	(0%)	0	(0%)	21		29
	90%	0	(0%)	264	(100%)	0	(0%)	0	(0%)	264		275
	80%	2	(0%)	1461	(100%)	0	(0%)	0	(0%)	1463		1552
Pimsb*	50%	0	(0%)	27	(100%)	0	(0%)	0	(0%)	27		36
	40%	1	(0%)	197	(100%)	0	(0%)	0	(0%)	197		222
	30%	3	(0%)	941	(100%)	0	(0%)	0	(0%)	945		1024
T10I4D100k	0.500%	0	(4%)	5	(95%)	0	(0%)	0	(0%)	5		13
	0.100%	6	(20%)	23	(78%)	0	(0%)	1	(2%)	29		40
	0.050%	10	(27%)	27	(71%)	0	(0%)	1	(2%)	38		48
	0.010%	61	(57%)	41	(38%)	0	(0%)	5	(4%)	107		129
	0.005%	166	(73%)	49	(21%)	0	(0%)	14	(6%)	229		261
	0.000%	507	(81%)	57	(9%)	0	(0%)	60	(10%)	624		820
T25I10D10k	1.000%	1	(31%)	3	(66%)	0	(0%)	0	(3%)	4		12
	0.500%	5	(47%)	5	(48%)	0	(0%)	1	(5%)	11		19
	0.100%	45	(72%)	14	(22%)	0	(0%)	3	(5%)	63		82
	0.050%	124	(81%)	20	(13%)	0	(0%)	9	(6%)	154		181
	0.000%	553	(85%)	28	(4%)	0	(0%)	66	(10%)	647		945
T25I20D100k	1.000%	1	(4%)	26	(95%)	0	(0%)	0	(0%)	27		37
	0.500%	6	(7%)	82	(93%)	0	(0%)	1	(1%)	89		105
	0.100%	33	(14%)	190	(84%)	0	(0%)	4	(2%)	227		256
	0.050%	46	(18%)	201	(80%)	0	(0%)	6	(2%)	253		287
	0.010%	760	(71%)	311	(29%)	0	(0%)	0	(0%)	1072		1349

Table I.2: Calculated memory consumption of QuICL Oid-Trie lattice. Actual is the memory usage reported by the "Mem Usage" field of the Windows Task Manager upon termination of algorithm execution.

	Min Supp	Calculated Memory Consumption of QuICL Oid-Less in MBs (% of total)										Actual
		Concepts		Object Ids		Items		P-C Links		Total		
Chess	85%	1	(86%)			0	(0%)	0	(14%)	1		9
	75%	4	(83%)			0	(0%)	1	(17%)	4		13
	65%	16	(81%)			0	(0%)	4	(19%)	19		43
	55%	62	(80%)			0	(0%)	16	(20%)	78		343
Mushroom	50%	0	(93%)			0	(0%)	0	(7%)	0		8
	30%	0	(90%)			0	(0%)	0	(10%)	0		10
	10%	2	(87%)			0	(0%)	0	(13%)	2		22
	0%	76	(82%)			0	(0%)	16	(18%)	93		100
Pumsb	95%	0	(91%)			0	(0%)	0	(9%)	0		8
	90%	0	(87%)			0	(0%)	0	(13%)	1		18
	80%	3	(84%)			0	(0%)	1	(16%)	3		100
Pimsb*	50%	0	(90%)			0	(0%)	0	(10%)	0		11
	40%	1	(86%)			0	(0%)	0	(14%)	1		47
	30%	5	(84%)			0	(0%)	1	(16%)	6		234
T10I4D100k	0.500%	0	(94%)			0	(1%)	0	(6%)	0		57
	0.100%	9	(89%)			0	(0%)	1	(11%)	10		77
	0.050%	15	(90%)			0	(0%)	2	(10%)	17		93
	0.010%	91	(90%)			0	(0%)	10	(10%)	100		205
	0.005%	246	(90%)			0	(0%)	28	(10%)	274		345
	0.000%	751	(86%)			0	(0%)	121	(14%)	872		896
T25I10D10k	1.000%	2	(88%)			0	(0%)	0	(12%)	2		21
	0.500%	7	(88%)			0	(0%)	1	(12%)	9		25
	0.100%	67	(91%)			0	(0%)	7	(9%)	74		116
	0.050%	184	(91%)			0	(0%)	19	(9%)	203		235
	0.000%	819	(86%)			0	(0%)	132	(14%)	950		1156
T25I20D100k	1.000%	2	(88%)			0	(0%)	0	(12%)	2		94
	0.500%	9	(87%)			0	(0%)	1	(13%)	10		114
	0.100%	48	(85%)			0	(0%)	8	(15%)	57		186
	0.050%	68	(86%)			0	(0%)	12	(14%)	80		199
	0.010%	1126	(100%)			0	(0%)	0	(0%)	1126		1306

Table I.3: Calculated memory consumption of QuICL Oid-Less lattice. Actual is the memory usage reported by the "Mem Usage" field of the Windows Task Manager upon termination of algorithm execution.

	Min Supp	Calculated Memory Consumption of GMA in MBs (% of total)									Actual
		Concepts		Object Ids		Items		P-C Links		Total	
Chess	85%	1	(2%)	21	(97%)	0	(0%)	0	(0%)	22	33
	75%	3	(3%)	119	(97%)	0	(0%)	1	(1%)	122	139
	65%	14	(3%)	450	(96%)	0	(0%)	4	(1%)	468	500
	55%	53	(3%)	1533	(96%)	0	(0%)	16	(1%)	1602	1656
Mushroom	50%	0	(1%)	1	(99%)	0	(0%)	0	(0%)	1	11
	30%	0	(2%)	6	(98%)	0	(0%)	0	(0%)	6	16
	10%	1	(5%)	28	(95%)	0	(0%)	0	(1%)	29	40
	0%	66	(39%)	88	(52%)	0	(0%)	16	(10%)	170	194
Pumsb	95%	0	(0%)	21	(100%)	0	(0%)	0	(0%)	21	36
	90%	0	(0%)	265	(100%)	0	(0%)	0	(0%)	265	298
	80%	2	(0%)	1467	(100%)	0	(0%)	1	(0%)	1470	1582
Pimsb*	50%	0	(0%)	28	(100%)	0	(0%)	0	(0%)	28	42
	40%	1	(0%)	229	(100%)	0	(0%)	0	(0%)	229	259
	30%	4	(0%)	1114	(100%)	0	(0%)	1	(0%)	1119	1222
T10I4D100k	0.500%	0	(5%)	5	(94%)	0	(0%)	0	(0%)	6	19
	0.100%	7	(21%)	27	(76%)	0	(0%)	1	(3%)	35	52
	0.050%	13	(28%)	32	(69%)	0	(0%)	2	(4%)	47	64
	0.010%	78	(57%)	48	(36%)	0	(0%)	10	(7%)	136	159
	0.005%	212	(71%)	61	(20%)	0	(0%)	28	(9%)	301	352
	0.000%	648	(77%)	77	(9%)	0	(0%)	121	(14%)	846	1048
T25I10D10k	1.000%	2	(31%)	3	(64%)	0	(0%)	0	(5%)	5	14
	0.500%	6	(43%)	8	(50%)	0	(0%)	1	(7%)	15	24
	0.100%	58	(69%)	20	(23%)	0	(0%)	7	(8%)	84	104
	0.050%	159	(77%)	28	(14%)	0	(0%)	19	(9%)	206	235
	0.000%	706	(80%)	49	(5%)	0	(0%)	132	(15%)	887	1156
T25I20D100k	1.000%	1	(5%)	27	(94%)	0	(0%)	0	(1%)	29	42
	0.500%	7	(8%)	87	(91%)	0	(0%)	1	(1%)	96	113
	0.100%	42	(16%)	216	(81%)	0	(0%)	8	(3%)	266	291
	0.050%	59	(19%)	232	(77%)	0	(0%)	12	(4%)	303	327
	0.010%	972	(68%)	458	(32%)	0	(0%)	0	(0%)	1430	

Table I.4: Calculated memory consumption of GMA lattice. Actual is the memory usage reported by the "Mem Usage" field of the Windows Task Manager upon termination of algorithm execution.

	Min Supp	Calculated Memory Consumption of CHARM-L in MBs (% of total)								Actual
		Concepts		Object Ids	Items		P-C Links		Total	
Chess	85%	1	(83%)		0	(4%)	0	(12%)	1	10
	75%	4	(80%)		0	(5%)	1	(15%)	5	20
	65%	18	(78%)		1	(6%)	4	(16%)	23	63
	55%	69	(76%)		6	(7%)	16	(17%)	91	217
Mushroom	50%	0	(91%)		0	(3%)	0	(6%)	0	12
	30%	0	(86%)		0	(5%)	0	(9%)	0	14
	10%	2	(83%)		0	(6%)	0	(11%)	2	22
	0%	85	(72%)		16	(14%)	16	(14%)	117	397
Pumsb	95%	0	(90%)		0	(3%)	0	(8%)	0	24
	90%	1	(84%)		0	(4%)	0	(11%)	1	35
	80%	3	(81%)		0	(5%)	1	(14%)	4	48
Pimsb*	50%	0	(88%)		0	(4%)	0	(8%)	0	32
	40%	1	(83%)		0	(5%)	0	(12%)	1	51
	30%	6	(80%)		0	(7%)	1	(14%)	7	84
T10I4D100k	0.500%	0	(93%)		0	(2%)	0	(5%)	0	42
	0.100%	10	(87%)		0	(3%)	1	(10%)	11	76
	0.050%	17	(88%)		1	(3%)	2	(9%)	19	96
	0.010%	101	(89%)		3	(3%)	10	(8%)	114	284
	0.005%	274	(88%)		10	(3%)	28	(9%)	312	637
	0.000%	836	(84%)		37	(4%)	121	(12%)	994	1552
T25I10D10k	1.000%	2	(86%)		0	(4%)	0	(10%)	2	21
	0.500%	8	(86%)		0	(4%)	1	(11%)	10	36
	0.100%	75	(89%)		2	(3%)	7	(8%)	84	184
	0.050%	205	(89%)		7	(3%)	19	(8%)	231	446
	0.000%	911	(84%)		39	(4%)	132	(12%)	1082	1548
T25I20D100k	1.000%	2	(86%)		0	(3%)	0	(10%)	2	48
	0.500%	10	(84%)		0	(4%)	1	(12%)	11	115
	0.100%	54	(83%)		3	(4%)	8	(13%)	65	162
	0.050%	76	(83%)		4	(4%)	12	(13%)	91	207
	0.010%	1253	(100%)		0	(0%)	0	(0%)	1253	

Table I.5: Calculated memory consumption of CHARM-L lattice. Actual is the memory usage from the "Mem Usage" field of the Windows Task Manager upon termination of algorithm execution.

Reference List

Agrawal, R., Imieliski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 207-216, Washington, D.C., United States: ACM Press.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases, *Proceedings of the 20th International Conference on Very Large Databases*, 487-499, Santiago, Chile.

Barbut, M., & Monjardet, B. (1970). Ordre et classification: Algebre et combinatoire.: Hachette.

Bayardo, R. J. Jr. (1998). Efficiently mining long patterns from databases, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 85-93, Seattle, Washington, United States: ACM.

Bordat, J. P. (1992). *Sur i'algorithmique combinatoire d'ordres finis.* Université de Montpellier.

Brin, S., Motwani, R., & Silverstein, C. (1998). Beyond market baskets: Generalizing association rules to dependence rules, *Data Mining and Knowledge Discovery, 2*(1), 39-68.

Brin, S., Motwani, R., & Silverstein, C. (1997a). Beyond market baskets: Generalizing association rules to correlations, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 265-276, Tucson, Arizona, United States: ACM.

Brin, S., Motwani, R., Ullman, J. D., & Tsur, S. (1997b). Dynamic itemset counting and implication rules for market basket data, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 255-264, Tucson, Arizona, United States: ACM.

Burdick, D., Calimlim, M., & Gehrke J. (2001). Mafia: A maximal frequent itemset algorithm for transactional databases, *Proceedings of the 17th International Conference on Data Engineering*, 443-452: IEEE Computer Society.

Chein, M. (1969). Algorithme de recherche des sous-matrices premières d'une matrice. *Bull. Math. Soc. Sci. Math. R.S. Roumanie, 13*, 21-25.

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

Cooley, R., Mobasher, B., & Srivastava, J. (1997). Web mining: Information and pattern discovery on the world wide web, *Proceedings of the 9th International Conference on Tools with Artificial Intelligence*, 558-567: IEEE Computer Society.

DCI: A hybrid algorithm for frequent set counting - datasets. (2008). Retrieved 2008, from http://miles.cnuce.cnr.it/~palmeri/datam/DCI/datasets.php

Dunkel, B., & Soparkar, N. (1999). Data organization and access for efficient data mining, *Proceedings of the 15th International Conference on Data Engineering*, 522-529: IEEE Computer Society.

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.

Frequent Itemset Mining Dataset Repository. (2008). Retrieved 2008, from http://fimi.cs.helsinki.fi/data/

Ganter, B. (1984). *Two basic algorithms in concept analysis*. TU Darmstadt, Germany.

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.

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

Harms, S., Li, D., Deogun, J., & Tadesse, T. (2002). Efficient rule discovery in a geo-spatial decision support system, *Proceedings of the 2002 Annual National Conference on Digital Government Research*, 1-7. Los Angeles, California: Digital Government Society of North America.

Huhtala, Y., Karkkainen, J., Porkka, P., & Toivonen, H. (1999). Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal, 42*(2), 100-111.

IBM synthetic data generator. (2001). from http://www.almaden.ibm.com/software/quest/Resources/

Kamber, M., Han, J., & Chiang, J. Y. (1997). Metarule-guided mining of multi-dimensional association rules using data cubes, *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, 207-210.

Knuth, D. E. (1998). *The Art of Computer Programming, Vol. 3, Sorting and Searching* (2nd. ed.). Reading, MA: Addison-Wesley.

Kuznetsov, S. O. (1993). A fast algorithm for construction of all intersections of objects from a finite semilattice. *Automatic Documentation and Mathematical Linguistics, 27*(5), 11-21.

Kuznetsov, S. O. (2001). On computing the size of a lattice and related decision problems. *Order, 18*(4), 13-21.

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.

Lakha, l., & Stumme, G. (2005). Efficient mining of association rules based on formal concept analysis. *Lecture Notes in Computer Science, 3626*, 180-195.

Lee, W., & Stolfo, S. J. (1998). Data mining approaches for intrusion detection, *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*. San Antonio, Texas: USENIX Association.

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

Lopes, S., Petit, J. M., & Lakhal, L. (2000). Efficient discovery of functional dependencies and armstrong relations, *Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology*, 350-364: Springer-Verlag.

Lucchese, C., Orlando, S., & Perego, R. (2004). *DCI Closed: A fast and memory efficient algorithm to mine frequent closed itemsets*.

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.

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, *Proceedings of the 6th International Conference on Concept Lattices and Their Applications*, 10-21. Olomouc, Czech Republic.

Mannila, H., Toivonen, H., & Verkamo, A. I. (1997). Discovery of frequent episodes in event sequences. *Data Mining Knowledge Discovery, 1*(3), 259-289.

Norris, E. M. (1978). An algorithm for computing the maximal rectangles in a binary relation. *Rev. Roumaine Math. Pures Appl., 23*(2), 243-250.

Nourine, L., & Raynaud, O. (1999). A fast algorithm for building lattices. *Inf. Process. Lett., 71*(5-6), 199-204.

Nourine, L., & Raynaud, O. (2002). A fast incremental algorithm for building lattices. *Journal of Experimental & Theoretical Artificial Intelligence, 14*(2&3), 217-227.

Ordonez, C., Santana, C. A., & Braal, L. (2000). Discovering interesting association rules in medical data, *Proceedings of the ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 78-85.

Palmerini, P., Orlando, S., & Perego, R. (2004). Statistical properties of transactional databases, *Proceedings of the 2004 ACM Symposium on Applied Computing*, 515-519. Nicosia, Cyprus: ACM.

Park, J. S., Chen, M. S., & Yu, P. S. (1995). An effective hash-based algorithm for mining association rules, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 175-186. San Jose, California, United States: ACM.

Pasquier, N. (2000). Mining association rules using formal concept analysis, *Proceedings of the 8th International Conference on Conceptual Structures,* 259-264. Darmstadt, Germany: Springer-Verlag.

Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999a). Efficient mining of association rules using closed itemset lattices. *Inf. Syst., 24*(1), 25-46.

Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999b). Discovering frequent closed itemsets for association rules, *Proceedings of the 7th International Conference on Database Theory*, 398-416: Springer-Verlag.

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

Pemmaraju, S., & Skiena, S. (1990). *Computational discrete mathematics: Combinatorics and graph theory with mathematica*. Reading, MA: Addison-Wesley.

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

Rouane, M. H., Nehm´e, K., Valtchev, P., & Godin, R. (2004). On-line maintenance of iceberg concept lattices, *Contributions to the 12th International Conference on Conceptual Structures*. Huntsville, AL: Shaker Verlag.

Shenoy, P., Haritsa, J. R., Sudarshan, S., Bhalotia, G., Bawa, M., & Shah, D. (2000). Turbo-charging vertical mining of large databases. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data,* 22-33. Dallas, Tx: ACM

Srikant, R., & Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements, *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, 3-17: Springer-Verlag.

Stumme, G. (2002). Efficient data mining based on formal concept analysis, *Proceedings of the 13th International Conference on Database and Expert Systems Applications*, 534-546: Springer-Verlag.

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

Stumme, G., Taouil, R., Bastide, Y., & Lakhal, L. (2001a). Conceptual clustering with iceberg concept lattices, *Proceedings of GI-Fachgruppentreffen Maschinelles Lernen'01, Universität Dortmund, 763*.

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

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

Valiant, L. (1979). The Complexity of Computing the Permanent. *Theoretical Computer Science, 8*, 189-201: Elsevier.

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

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

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

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

Valtchev, P., Missaoui, R., & Lebrun, P. (2002b). A partition-based approach towards constructing galois (concept) lattices. *Discrete Mathematics, 256*(3), 801-829.

Valtchev, P., Rouane, H., & Missaoui, R. (2003b). A generic scheme for the design of efficient on-line algorithms for lattices. *Lecture Notes in Computer Science, 2746,* 282-295 : Springer-Berlin/Heidelberg.

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

Wille, R. (1982). Restructuring lattice theory: An approach based on hierarchies of concepts, *Ordered Sets,* 445-470. Dordrecht Boston.

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.

Zaki, M. J. (2000). Generating non-redundant association rules, *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 34-43. Boston, Massachusetts, United States: ACM.

Zaki, M. J. (2008). Mohammed J. Zaki. 2008, from http://www.cs.rpi.edu/~zaki/

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