Constructing concept lattices and compressed pseudo-lattices

by

F.J. (Dean) van der Merwe

Submitted in fulfilment of the requirements for the degree
M.Sc. Computer Science
in the Faculty Engineering, Built Environment
and Information Technology

University of Pretoria

Pretoria, South Africa

May 2003

© University of Pretoria

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Constructing concept lattices and compressed pseudo-lattices

By F.J. (Dean) van der Merwe
Supervisor: Prof. D.G. Kourie
Department of Computer Science
University of Pretoria
M.Sc. Computer Science

Summary

Concept lattices are well-defined mathematically structures used in a variety of areas of application. Concept lattices are exponential in size in the worst case and therefore constructing concept lattices may be time consuming. This dissertation approaches the efficiency problems associated with constructing concept lattices from two points of view and proposes complementary solutions for each. Firstly, a new lattice construction algorithm, called AddAtom, unrelated to well known and published algorithms is proposed. Secondly, a data structure called a compressed pseudo-lattice is proposed. Compressed pseudo-lattices involve the construction of sublattices instead of the complete lattice of all concepts.

The AddAtom algorithm efficiently constructs lattices and is proposed as a general purpose lattice construction algorithm. The algorithm is an incremental algorithm and has a theoretic complexity bound of $O(||L||.||O||^2.\max(||O'||))$ when constructing the concept lattice, L of a context with a set of objects O each of which can possess a maximum of $\max(||O'||)$ attributes from a set of attributes A. In experimental comparisons AddAtom outperformed other published algorithms in a range of contexts. In randomly generated contexts that had very high or very low densities its performance was the second best. It was however the best performing incremental lattice construction algorithm.

The notion of a compressed pseudo-lattice is introduced and suggested as a data structure in areas of application related to Formal Concept Analysis (FCA). It is closely related to the line diagram of a lattice and its use as a computational tool in applications such as machine learning, information retrieval and knowledge discovery in databases is discussed. The data structure, essentially a bipartite graph that incorporates an embedded sublattice, combines desirable features of concept lattices in a structure that allows for a flexible mechanism of scaling the size of the embedded sublattice, using defined operations that compress and expand it by removing or adding atoms and coatoms. A compressed pseudo-lattice essentially represents a complete sublattice from which a number of atoms and/or coatoms have been removed. Additionally the relation of the sublattice to the context from which it was derived is preserved. An application-dependent compression strategy or criterion is required to guide this process. The intent- and extent representative operations of a lattice are defined as substitutes for the infimum and supremum operations in compressed pseudo-lattices. It is argued that the removal of concepts from a concept lattice may hold advantages over traditional approaches.

Acknowledgements

My sincere thanks to:

- My family for their support and encouragement.
- Deon Oosthuizen for introducing me to the subject during many enthusiastic discussions and his support and assistance during his tenure as an associate professor at the University of Pretoria during which he also acted as supervisor.
- Prof. Derrick Kourie for his assistance, patience and invaluable advice as supervisor.
- Sergei Obiedkov for his collaborative assistance and for making available the data of his experimental comparisons of AddAtom and other lattice construction algorithms as well as helpful comments and suggestions.

Table of contents

Acknowledgements ... 3
Table of contents ... 4
Chapter 1: Introduction ... 6
Chapter 2: Order theoretic and FCA lattice definitions and notation ... 9
2.1 List and set notation ... 9
2.2 Order theoretic lattice definitions ... 9
2.3 FCA definitions ... 11
2.4 Why EA-lattices? ... 13
2.5 EA-lattice definition ... 15
2.6 Boolean lattices ... 18
2.7 Augmented lattices ... 19
2.8 Different views of a lattice ... 19
2.8.1 Set-theoretic view of lattices ... 19
2.8.2 Graph-theoretic view of lattices ... 20
2.9 Intent and extent operations ... 21
2.10 Summary ... 23
Chapter 3: Lattice construction ... 24
3.1 Algorithmic lattice construction ... 24
3.2 Incremental vs. batch lattice construction algorithmns ... 24
3.3 Constructing the line diagram ... 24
3.4 An inefficient batch lattice construction algorithm ... 25
Chapter 4: The AddAtom lattice construction algorithm ... 28
4.1 Informal description ... 28
4.2 Intent- and extent representative operations and lattice construction ... 33
4.3 Definition of the AddAtom algorithm ... 36
4.4 AddAtom example ... 39
4.5 An algorithm for AIR and EIR ... 41
4.6 Efficient AddAtom algorithm ... 45
4.7 Discussion ... 48
Chapter 5: AddAtom algorithmic performance ... 49
5.1 A survey of concept lattice construction algorithms ... 50
5.2 A Theoretical performance bound for AddAtom ... 52
5.2.1 Notation ... 52
5.2.2 Concept lattice size formulae ... 53
5.2.3 Complexity of set operations ... 57
5.2.4 AddAtom theoretical performance ... 58
5.3 Empirical performance: pilot study ... 61
5.4 Empirical performance: wide study ... 69
5.5 Conclusions ... 76
Chapter 6: Compressed pseudo-lattices ... 77
6.1 A bipartite database and query operation ... 77
6.2 A concept lattice database and query operation ... 79
6.3 An adapted sublattice database and query operation ... 79
6.4 Removing concepts from a lattice ... 80
6.5 The use of the intent- and extent representative operations ... 83
6.6 The CompressLattice operation ... 84
6.7 Definition and properties of compressed pseudo-lattices ... 85
6.8 The ExpandLattice operation ... 88
6.9 Interpretation of compressed pseudo-lattices ... 89
6.10 Why compressed pseudo-lattices? ... 91
6.11 Compression strategies and criteria ... 94
6.12 Implementation and discussion of preliminary results ... 96
Chapter 7: AddAtom implementation ... 98

7.1 Evolution of code 98
7.2 Set class 99
7.3 Lattice class 99
7.4 Compressed pseudo-lattice implementation 101
7.5 Implementation issues 102
7.5.1 Time 102
7.5.2 Space / memory 102
7.5.3 Object-oriented implementation 103
7.5.4 UNIX and Windows 103
7.5.5 Testing 104
7.5.6 User interface 105
7.5.7 Continued advances in hardware 105
7.6 Comparison with other lattice construction algorithms 105
Chapter 8: Summary and future work 106
8.1 Positioning and related research 106
8.2 Further work 108
References 110
Appendix A: Further optimised AddAtom algorithm 114
Appendix B: AddAtom algorithmic complexity bounds 118

Chapter 1: Introduction

Formal Concept Analysis (FCA) is an established area of research in the computer sciences with many areas of application especially in branches of artificial intelligence. Central to FCA is the notion of a formal concept lattice (or concept lattice for short). A concept lattice is a mathematically well-defined structure that comprises of a number of concepts describing a context. Each concept has an extent consisting of a number of objects from the context as well as the attributes that these objects have in common within the context. The concepts are ordered in a partial order and form an order-theoretic lattice. Chapter 2 defines and gives examples of concept lattices.

Concept lattices have been proposed and studied in a number of areas of application:

- Data analysis (Vogt and Wille (1995)).
- Discovery of association rules (Stumme et al. (1998, 2000), Pasquier et al. (1999)).
- Information retrieval (Godin et al. (1995a), Carpineto and Romano (1996a)).
- Exploration of attributes and attribute relationships in data (Ganter (1999), Duquenne (1999, 2001)).
- Conceptual clustering and classification (Godin et al. (1991), Carpineto and Romano (1996b)).
- Machine learning (Oosthuizen (1994b), Mephu Nguifo and Njiwoua (2001), Xie et al. (2002)).
- Computer assisted human browsing and dissemination of data (Cole and Stumme (2000), Cole and Eklund (2001)).

FCA was proposed in the early eighties by Wille (1982). Ganter and Wille (1999) now serves as the foundation of FCA. FCA builds on the work of Birkhoff (1973) and Barbut and Monjardet (1970). In FCA, the problem of generating the set of concepts of a concept lattice and then constructing the line diagram to represent the concept lattice has been well-studied (refer to Kuznetsov and Obiedkov (2002) for an overview and comparison). Chapter 3 describes the basic challenges of this problem.

In the worst case, concept lattices are however exponential in size in terms of the input context. Although, in general, natural data does not realize the worst case, in practical applications very large lattices can still result. This creates time and space performance issues for applications using lattices and therefore every effort should be made to more efficiently construct lattices. This dissertation approaches the efficiency problems from two points of view and proposes complementary solutions for each. First a new lattice construction algorithm, called AddAtom, unrelated to well known and published algorithms is proposed. The algorithm efficiently constructs lattices and is proposed as a general purpose lattice construction algorithm that outperforms other published algorithms in a wide range of contexts (although not in all types of contexts). The second proposed approach to managing the time taken to build the lattice is to construct sub-lattices instead of the complete lattice of all concepts. A generic framework and the necessary operations for building such lattices are proposed and defined. The resulting data structure is called a

compressed pseudo-lattice. Not only does this approach allows for the incremental scaling of the lattice size in relation to the amount of time that an application can afford to spend on constructing lattices, but initial evidence suggest a number of other advantages, despite the removal of a large number of concepts from the lattice.

AddAtom

The AddAtom algorithm is explained and defined in chapter 4. When inserting a new object into a lattice L_n, the basic strategy of the algorithm is to start at the zero concept in L_n and then recursively search its parent concepts for generator concepts in an elegant and tightly focussed search. After creating the required new concepts and arcs associated with the generator concepts and modifying the relevant concepts a new lattice L_{n+1} is produced.

The algorithm is characterised by a number of features that differentiates it from some of the other construction algorithms. Firstly, it is an incremental lattice construction algorithm and therefore constructs a lattice L_{n+1} from an input lattice L_n by adding an additional object to the context of L_n. Secondly, it creates the set of concepts and the line diagram of the lattice. Thirdly, it is defined on concept lattices as well as on concept sublattices, which makes it suitable to be used in generating compressed pseudo-lattices.

Fourthly, AddAtom is defined in terms of a class of lattice operations, called the intent- and extent representative operations of a lattice. These operations elegantly define the nature of lattice construction and their use to traverse and inspect a lattice is significantly different from most other approaches that use the supremum and infimum operations. These operations are in some sense second order supremum and infimum operations and are useful in situations where, for example, the infimum or supremum of a set of attributes in a lattice is trivial (i.e. either unit or the zero concept). Throughout the dissertation it is argued that these operations are very useful in traversing and searching the lattice for areas of interest. This is underpinned by the fact that the intent- and extent representative operations are key to the definition of AddAtom as well as compressed pseudo-lattices. Chapter 6 also provides examples of how these operations may be useful in information retrieval (IR) and machine learning.

During the past few decades a number of lattice construction algorithms have been proposed and although not all the algorithms are directly comparable due to the fact that all do not produce the same outputs (e.g. some generate the line diagrams while others merely generate the set of concepts). In Chapter 5 a worst-case complexity bound of AddAtom is established as $O(||L||.||O||^2.\max(||O'||))$. This bound is cubic in nature relative to the lattice size. Although this bound is not of the same order of magnitude as that of the lowest known for lattice construction algorithms, this does not necessarily mean that the algorithm has a worse performance.

The results of experimental comparisons of AddAtom with other lattice construction algorithms (chapter 5) show that AddAtom is a very good lattice construction algorithm and compares well with other algorithms from an experimental point of view. It does however not perform the best across all types of contexts. In random contexts with either very low or very high densities other algorithms perform slightly better than AddAtom. In these contexts AddAtom is still the second best performer. These results are consistent with the claim that the theoretical bound is not very sharp, confirming that the algorithmic performance of AddAtom is very good and that it is a worthy candidate for use as a general purpose lattice construction algorithm in that it is efficient (compared to other algorithms) over the range of artificial and natural data sets albeit not over all types of contexts.

Compressed pseudo-lattices

Compressed pseudo-lattices are defined in chapter 6 and provide a formal framework within which concepts can be removed from lattices to create sublattices. The data structure, essentially a bipartite graph that incorporates an embedded sublattice, combines desirable features of concept lattices in a structure that allows for a flexible mechanism of scaling the size of the embedded sublattice, using defined operations that compress and expand it by removing or adding atoms and coatoms. A compressed pseudo-lattice essentially represents a lattice from which a number of atoms and/or coatoms have been removed. Additionally the relation of the sublattice to the context from which it was derived is preserved. An application-dependent compression strategy or criterion is required to guide this process. It is argued that the removal of concepts from a concept lattice may hold advantages over traditional approaches.

The implementation of the algorithms developed for supporting the AddAtom and compressed pseudo-lattice implementations are discussed in chapter 7. A number of implementation issues and considerations as well as the strategies to deal with these are also discussed.

Finally, chapter 8 summarise the findings of the dissertation and ends by identifying areas of further research related to AddAtom and compressed pseudo-lattices.

Chapter 2: Order theoretic and FCA lattice definitions and notation

This chapter defines the basic lattice terminology and notation used throughout this text. The basic order-theoretic lattice definitions are well known and may be found in many standard mathematical texts, for example in Gratzer (1971). Building on the order-theoretic definitions the notation and definitions of Formal Concept Analysis (FCA) are then introduced (Ganter and Wille (1999)) and the basic FCA building block, the *formal concept lattice*, is defined.

Next an *EA-lattice* is defined. An EA-lattice is closely related to a FCA lattice and is described as a substantially equivalent lattice to a formal concept lattice. EA-lattices have some desirable features that make it suitable for the use in compressed pseudo-lattices (chapter 6). The different views of a lattice are also explored. Both the EA-lattice and the different views of a lattice are concepts that are frequently used in the rest of the chapters.

The chapter concludes by defining the *intent- and extent representative operations* on a lattice.

An important aspect of the intent- and extent representative operations is that they are defined specifically for use in concept sub-lattices, such as those formed by removing atoms and/or coatoms from formal concept lattices. In chapter 6 the notion of removing concepts from lattices is formalised and generalised into additional concept lattice operations and the definition of a data structure called a compressed pseudo-lattice. The removal of concepts from concept lattices have been proposed by other authors (refer to chapter 8 for a discussion) but the approach taken here is more general. The concept of an EA-lattice (section 2.5) is defined and compared to a formal concept lattice (section 2.3).

2.1 LIST AND SET NOTATION

In this text, sets are denoted by capitals e.g. S. whilst the set elements are in lower case e.g. x, y or z. A set is indicated by the notation $\{x, y, z\}$ or $\{a_1, a_2,\ldots, a_n\}$.

The cardinality of set S is denoted by $\|S\|$.

Ordered lists are shown as $\langle x, y, z\rangle$.

2.2 ORDER THEORETIC LATTICE DEFINITIONS

A *binary relation on a set* is an association between pairs of elements of the set.

Consider a set S and arbitrary elements x, y and z in S. A partial ordering relation, $\leqslant$, on S is a binary relation that is reflexive ($x \leqslant x$), antisymmetric ($x \leqslant y \wedge y \leqslant x \Rightarrow x = y$) and transitive ($x \leqslant y \wedge y \leqslant z \Rightarrow x \leqslant z$). The set S in conjunction with an associated partial

ordering relation, $\leqslant$, is called a *partially ordered set*[1] or *poset* and is denoted by $\langle S, \leqslant \rangle$. For $x, y \in S, x \neq y$, x is said to *cover* y, denoted by $y \prec x$ when $y \leqslant x$ and there is no $z \in S$, $z \neq x, z \neq y$ such that $y \leqslant z$ and $z \leqslant x$.

When $y \prec x$, some texts refer to x as the *parent*, *predecessor, upper cover* or *upper neighbour* of y. Similarly y is referred to as the *child*, *successor, lower cover or lower neighbour* of x.

One way of visually representing a poset is by means of a directed graph called a *line diagram* in which elements of the poset form the nodes and a directed arc (or edge) is drawn from node y to x iff $y \prec x$. Line diagrams are often referred to as a *Hasse diagrams.* They provide a natural data structure for visually representing posets. By convention, instead of showing the direction of arcs explicitly in the line diagram, node x is shown above node y if $y \prec x$. By virtue of the transitivity of the partial ordering relation, line diagrams are directed acyclic graphs.

Two elements x, y of a poset $\langle S, \leqslant \rangle$ are called *comparable* if $y \leqslant x$ or $x \leqslant y$. If these conditions are not met they are said to be *not comparable*.

Consider a poset L. $x \in L$ is an *upper bound* of $H \subseteq L$ iff $y \leqslant x$ for all $y \in H$. Out of all the upper bounds of H in L, the *least upper bound* of H (if it exists) is called its *supremum* and is denoted by Sup(H) or Sup(L, H). Likewise, $x \in L$ is a *lower bound* of a set $H \subseteq L$ iff $x \leqslant y$ for all $y \in H$. The *greatest lower bound* of H (if it exists) is called its *infimum* and is denoted by Inf(H) or Inf(L, H). A poset $\langle L, \leqslant \rangle$ is a *lattice* iff Sup({x, y}) and Inf({x, y}) exist for all pairs $x, y \in L$. It is not difficult to show that if Sup(H) exists then it is unique, and likewise for Inf(H). Some texts refer to a supremum as a *join* and the infimum as a *meet.* Two or more elements of a poset are also said to *meet* at their infimum. A poset S is a *complete lattice* if the supremum and infimum exist for all subsets of S. It can be shown that all non-empty finite posets that are lattices are complete. A subset U of a complete lattice V that is closed under both suprema and infima is called a *complete sublattice.*

A complete lattice L has a largest element called the *unit element*, denoted by 1_L, and a smallest element called the *zero element*, denoted by 0_L. The elements in a lattice covering the zero element are often called *atoms* whilst the elements covered by the unit element are called *coatoms*.

The *upward closure* of any element c, indicated by UpwardClosure(L, c), is the set of elements greater than or equal to c in terms of the partial order. The *downward closure* of c is the set of elements that are less than or equal to c, and is indicated by DownwardClosure(L, c).

Figure 2.1 is the line diagram of the poset $\langle \{1, 2, 3, 4, 6, 8, 12, 24\}, | \rangle$ where $m|n$ means that m is a factor of (or divides) n. In the figure, 24 is the supremum of {3, 8} whilst 2 is the infimum of {8, 2}. It is easy to verify that both the supremum and infimum of any pair of elements exist, that in each case they are unique and this poset is therefore a lattice. The upward closure of 6 is {6, 12, 24} whilst its downward closure is {1, 2, 3, 6}. Since 24 is the largest element of the poset it is the unit element whilst 1, being the smallest element, is the zero element of the poset. Elements 2 and 3 are the atoms of the lattice whilst 8 and 12 are the coatoms.

---

[1] Ganter and Wille (1999) also refers to a partially ordered set as an ordered set but we avoid this terminology since it may cause confusion with that of a *completely ordered set* in which all the elements can be ordered from smallest to largest.

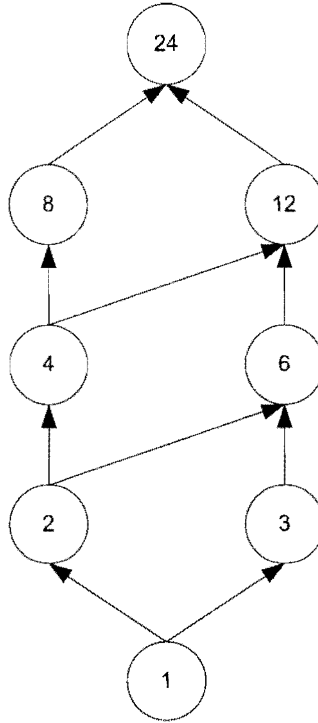


Figure 2.1: Lattice of ⟨{1, 2, 3, 4, 6, 8, 12, 24}, |⟩ where $m|n$ means m is a factor of (or divides into) n

In what follows the direction of the arcs will not be shown since there is no loss of generality in doing so.

Not all elements of a poset are necessarily related or comparable (hence the term *partial* order). A subset, S, of a poset whose elements are all comparable (i.e. $x \leqslant y$ or $y \leqslant x$ for all $x, y, \in S, x \neq y$) is called a *chain*. A subset, S, of a poset, none of whose elements are comparable is called an *anti-chain*.

The *width* of a finite poset is defined as the maximal size of an antichain in that poset. The *length* of a poset is defined as the supremum of the sizes of chains in the poset.

In a line diagram of a lattice, the nodes above a given element x are said to be *spanned* by x if there is a path from the x to the nodes. A node that spans a set of nodes is therefore a *lower bound* of the set of nodes. If for a node y there is a set of nodes with paths that end in y then y an *upper bound* of the set of lattice elements.

2.3 FCA DEFINITIONS

Consider a domain of discourse in which each element of a set of *objects*, $O = \{o_1, o_2, \ldots, o_j\}$, possesses one or more observable *attributes* from a set of attributes $A = \{a_1, a_2, \ldots, a_k\}$. We also refer to objects as *entities*, whilst attributes are sometimes referred to as *features* or *descriptors*. The triple $C = \langle O, A, I\rangle$, where I is a *binary relation* between O and A, $I \subseteq O \times A$, is referred to as a *context* and denotes this domain of discourse. The binary relation I, also called an *incidence relation*, identifies the attributes of each object. The notation oIa is used to indicate that object o possesses the attribute a. For any $E \subseteq O$ and $F \subseteq A$ the following operators are defined:

$E' = \{ a : A \mid (\forall o \in E)\ oIa \}$ – the set of attributes common to the objects in E

$F' = \{ o : O \mid (\forall a \in F)\ oIa \}$ – the set of objects common to the attributes in F

FCA studies posets known as *formal concept lattices* (also referred to as *Galois lattices*) that are induced by a binary relation over a pair of sets of objects and attributes. In FCA the context $C = \langle O, A, I \rangle$ is known as a *formal context*. A *formal concept* of a formal context is a couple $\langle E, F \rangle$ from $\mathcal{P}(O) \times \mathcal{P}(A)$ with $E \subseteq O$ and $F \subseteq A$ (where $\mathcal{P}(X)$ is the power set of the set X). In addition, the following property is satisfied:

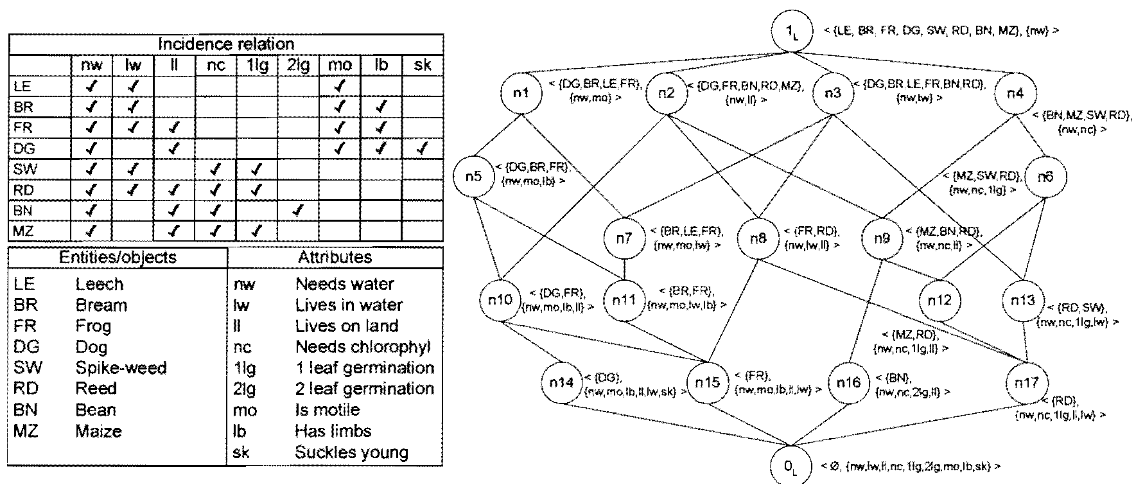
$F = E'$ and $E = F'$

A formal concept (henceforth referred to simply as a concept) is thus a pair consisting of a set of related objects having some attributes in common and the set of precisely those attributes that all the objects have in common. E is also called the *extent* of the concept $c = \langle E, F \rangle$ while F is called the concept's *intent* denoted by Extent(c) and Intent(c) respectively. The set F with $F \subseteq A$ is the intent of some concept if and only if $(F')' = F$, in which case the concept of which F is the intent is precisely $\langle F', F \rangle$. Similarly $E \subseteq O$ is the extent of a concept $\langle E, E' \rangle$ iff $(E')' = E$. The *support* of a concept is defined as the number of objects in its extent.

The set of all formal concepts in a context can be shown (according to the basic theorem on concept lattices – see Ganter and Wille 1999) to constitute a lattice with respect to the partial ordering relation $\leq_c$ defined as:

$\langle E_1, F_1 \rangle \leq_c \langle E_2, F_2 \rangle$ iff $E_1 \subseteq E_2$ (or equivalently iff $F_1 \supseteq F_2$) for two concepts $c_1 = \langle E_1, F_1 \rangle$ and $c_2 = \langle E_2, F_2 \rangle$

This lattice is known as a *formal concept lattice* in FCA. Since only formal concepts are part of a concept lattice and since there is a direct relationship between E and E', and therefore either extent or intent of a concept in a formal concept lattice uniquely identifies the concept.



Incidence relation									
	nw	lw	ll	nc	1lg	2lg	mo	lb	sk
LE	✓	✓					✓		
BR	✓	✓					✓	✓	
FR	✓	✓	✓				✓	✓	
DG	✓		✓				✓	✓	✓
SW	✓	✓		✓	✓				
RD	✓	✓	✓	✓	✓				
BN	✓		✓	✓		✓			
MZ	✓		✓	✓	✓				

Entities/objects		Attributes	
LE	Leech	nw	Needs water
BR	Bream	lw	Lives in water
FR	Frog	ll	Lives on land
DG	Dog	nc	Needs chlorophyl
SW	Spike-weed	1lg	1 leaf germination
RD	Reed	2lg	2 leaf germination
BN	Bean	mo	Is motile
MZ	Maize	lb	Has limbs
		sk	Suckles young

Figure 2.2: Context and formal concept lattice of the Living Context

A context can easily be represented by a *cross table*, i.e. by a matrix where the rows are labelled by the objects in the context and the columns by the attributes. A cross (or tick) in row g and column m indicates that the object g possesses the attribute m.

Figure 2.2 is the formal concept lattice of a small context. This simple context, called the Living Context, is taken from Ganter et al. (1986) and was originally used in a Hungarian educational film. The context is a simple ecological description of some living organisms.

Although very simplistic, it is useful for illustrative purposes. Each concept in the figure is numbered and is also labelled with its intent and extent.

The lattice shows a number of characteristics of the Living Context. For example the unit concept $1_L = \langle\{LE, BR, FR, DG, SW, RD, BN, MZ\}, \{nw\}\rangle$ has all the objects of the context in its extent but also has $\{nw\}$ as intent. This indicates that all the objects in the Living Context possess the attribute nw. Alternatively it can be said that all living things/objects (in the context) need water. By virtue of the partial ordering relation, $\leq_c$, it is clear that each concept y that is covered by x possesses at least all the attributes of x and at least one attribute in addition in its intent. Similarly the extent of x possesses at least the objects of the intent of y and at least one object in addition. Since there is no concept in the lattice above concept n_6 that has llg in its intent and both nw and nc are also contained in n_6's intent it shows that any object with llg as an attribute also has nc and nw as attributes. Concept n_4 with intent $\{nw, nc\}$ indicates that the converse is however not true, in that there are objects with nw and nc as attributes but not possessing llg for example BN.

A formal concept lattice is a useful structure since the formal concept lattice of a particular context contains encoded within its concepts all "meaningful" concepts in that only combinations of objects that actually have a particular set of attributes in common are grouped together in a concept. (This is as a result of the definition of formal concepts.) Similarly all groupings of attributes that does have common objects are represented in the lattice. Thus a set of attributes such as $\{nc, sk\}$ is not a "meaningful" set since there is no set of objects that has this particular and only this particular set of attributes in common. Another way of viewing this is that there is no evidence in the context to suggest that $\{nc, sk\}$ is a meaningful concept in the context. (Note however, that if the context is expanded in some way, this might not continue to be the case.) Similarly, the set of attributes $F = \{ll, lw\}$ is also not a grouping of attributes supported by the context. Using the operator defined earlier $F' = \{FR, RD\}$ and therefore the objects FR and RD are the only two having ll and lw in common. However applying the operator $(F')' = \{nw, ll, lw\}$ we see that whenever ll and lw are present for an object of the context, the attribute nw is *always* present. F is thus not a grouping of attributes supported by the evidence but $\{nw, ll, lw\}$ is and corresponds to concept n_8 of the lattice. This type of reasoning makes lattices a particularly useful tool in machine learning and in knowledge discovery in databases (KDD). Some of the many types of reasoning (e.g. abductive reasoning, inductive reasoning, unsupervised learning, supervised learning etc.) supported by a lattice are discussed in Oosthuizen (1994b).

2.4 WHY EA-LATTICES?

When algorithmically constructing formal concept lattices, for machine learning purposes based on using "real" or natural data, there are however a number of drawbacks. These drawbacks are the reason for defining EA-lattices, a class of lattices closely related to FCA lattices, in section 2.5.

- The same concept may simultaneously represent different objects and attributes in the context. This can happen when incrementally building the lattice and objects already represented in the lattice do not sufficiently differentiate the attributes or when there are duplicate objects (perhaps as a result of data errors or incomplete data). It is desirable to have each object and attribute represented by a separate concept because this correspond to the natural world where distinct objects are indeed acknowledged as being different although the initial information about their features might not be sufficient to indicate the precise nature of the differences. The same can be said in regard to differentiating features of objects. A "real world" example would be when two closely related books are described by a number of

keywords. Because the texts of the books are so closely related the list of keywords describing each book could be the same for both books although they are still two separate books. In this example a concept lattice of books will typically be augmented with meta-information describing the location of the book in for example a library. If more than one object are represented by one concept a separate data structure needs to be created to store this meta-information. Should the different books be represented by different concepts, the lattice data structure could directly be used for this purpose without additional data structures. In the Living Context, concept n_{14} is clearly the most precise representative of the object DG. However, it is also the only representative of the attribute sk (sk occurs in no concept higher up in the lattice). Thus, despite the fact that object DG and attribute sk are different real-world concepts, they have the same representative in the formal concept lattice. The problem could conceivably be avoided by carefully choosing the attributes of each object or introducing more attributes, but this is not always possible.

- When incrementally building a lattice, the concepts corresponding to particular objects or attributes change as the lattice grows. When numbering the concepts in the data structure representing the lattice it is desirable to have the line diagram node corresponding to the particular object or attributes stay the same throughout the lifetime of the data structure. The node can then be used as an index into the data structure. A concept corresponds to an object if the extent of the concept contains only that object. A concept corresponds to an attribute if its intent contains only that attribute.

- The objects, attributes and so-called intermediate concepts of the lattice are not clearly partitioned in a formal concept lattice. In figure 2.2 it is difficult identifying the concepts corresponding to particular objects or attributes since they may be located on any level of the lattice. Indeed, as was pointed out above a concept, such as concept n_{14} may even “correspond” to both an object and an attribute.

- Attributes that are not present in any of the objects are not represented in a formal concept lattice. This can happen when the lattice is built incrementally and initially contains only a few objects, none of which contain the particular attributes. Similarly, if the lattice is being constructed by incrementally introducing new attributes, the objects without any attributes introduced into the lattice to date will not be represented in the lattice.

For the above reasons and for reasons related to the definition of a compressed pseudo-lattice (chapter 6), a related lattice called an *EA-lattice* or *entity attribute lattice* is defined. (Oosthuizen (1994b) and Kourie and Oosthuizen (1998) previously made mention of such lattices, but never formally defined them.) *Each* object and *each* attribute in an EA-lattice is represented as a separate concept that is not associated (or co-labelled with) with any other object or attribute. (Note that this need not be true in general for a formal concept lattice, as will be discussed below.) As a result, the concepts (excluding 1_L and 0_L) in an EA-lattice can be partitioned into three sets: the attribute concepts, the object concepts and the intermediate concepts respectively. (Note that such a partitioning is not necessarily possible in a formal concept lattice as, for example, when the extent of one attribute is a subset of another. The Living Context introduced in figure 2.2 is an example of a context where such a partition is not possible.) This corresponds to the real world in the sense that an object (or attribute) is acknowledged to be unique, even if there is initially insufficient evidence to support its uniqueness in the light of the data examined up until that time. Due to the one-to-one mapping from objects and attributes to concepts, it is thus permissible in an EA-lattice to talk of an *object concept* and an *attribute concept*.

2.5 EA-LATTICE DEFINITION

Assume that $\|A\| > 1$, $\|O\| > 1$. A concept $\langle E, F\rangle$, where $E \subseteq O$ and $F \subseteq A$, is called an *EA-formal concept* in a context $\langle O, A, I\rangle$ if any one of the following conditions are satisfied:

1. $\|E\| = 1$ and $F = E'$
2. $\|F\| = 1$ and $E = F'$
3. $E = \varnothing$ and $F = A$
4. $E = O$ and $F = \varnothing$
5. $F = E'$ and $E = F'$

The set of all EA-formal concepts from $\mathcal{P}(O) \times \mathcal{P}(A)$ in a formal context $C = \langle O, A, I\rangle$, is called an *EA-formal concept lattice*[2] (or simply an *EA-lattice*) with respect to the partial ordering relation $\leq_{EA}$[3] defined as:

$\langle E_1, F_1\rangle \leq_{EA} \langle E_2, F_2\rangle$ iff $E_1 \subseteq E_2$ or $F_1 \supseteq F_2$ for two concepts $c_1 = \langle E_1, F_1\rangle$ and $c_2 = \langle E_2, F_2\rangle$

In such a lattice, L, the *zero concept*, denoted by 0_L, corresponds to the EA-formal concept $\langle \varnothing, A\rangle$ (condition 3), whereas the *unit concept*, denoted by 1_L, corresponds to the EA-formal concept $\langle O, \varnothing\rangle$ (condition 4). *Attributes* are in the form $\langle F', F\rangle$ where F is a set containing only one element of A (condition 2). Objects are in the form $\langle E, E'\rangle$ where E is a set containing only one element of O (condition 1).

The lattice below is the corresponding EA-lattice of the Living Context applying the definition of an EA-lattice (for comparative purposes 0_L and 1_L are shown but will be excluded from further lattice diagrams). The same concept numbering of corresponding concepts in figure 2.2 is used to enable comparisons. Concepts that are exactly the same (in terms of their intents and extents) to that of the formal concept lattice are shaded.

[2] Note that some authors use the term 'lattice' interchangeably with 'formal concept lattices'. Here we distinguish between the order-theoretic term 'lattice' and 'sublattice' and the FCA terms 'formal concept lattice' and 'concept lattice' both of which are special 'order-theoretic lattices'. An EA-lattice is an order-theoretic lattice but not necessarily a formal concept lattice.

[3] It should be noted that $\leq_{EA}$ is a partial ordering relation only on EA-formal concepts and not on all possible concepts (this is also the case with $\leq_C$ and formal concepts).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

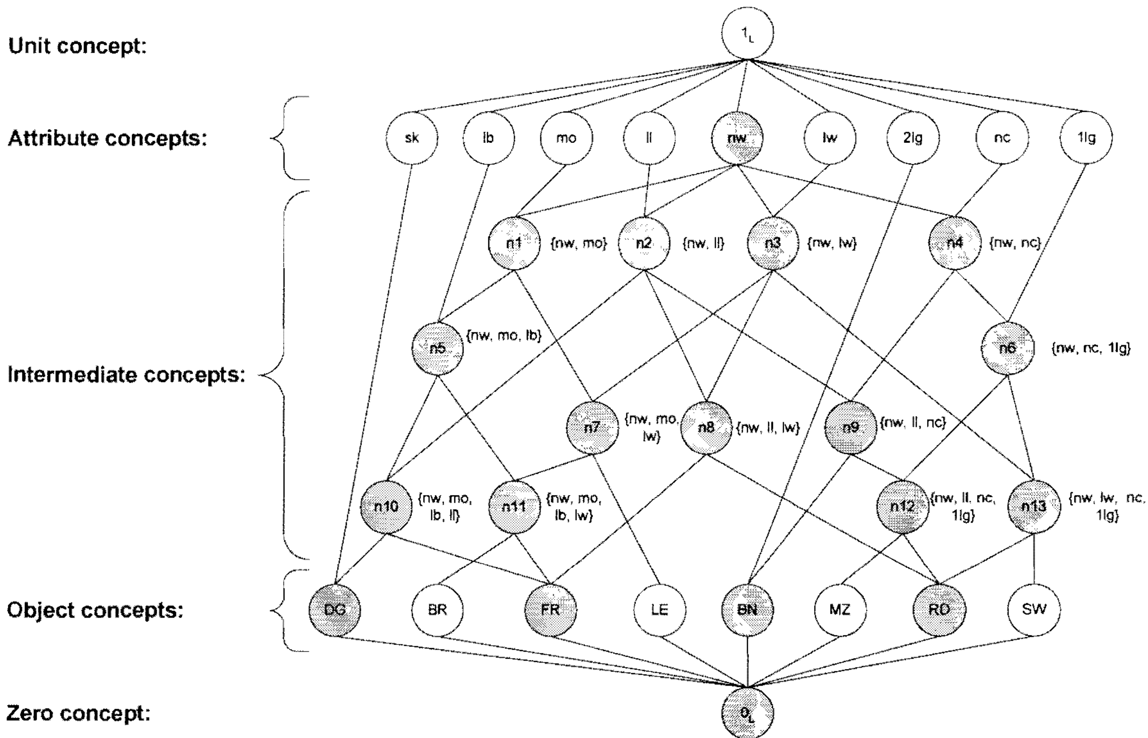


Figure 2.3: EA-lattice of the Living Context showing the partitioning of the concepts (compare to figure 2.2)

Note that an EA-lattice of a context differs only from the formal concept lattice if the concepts of conditions 1-4 are not generated by condition 5. This happens for example in a given context if an attribute f, is such that $F'' \neq F$, $F = \{f\}$. In such a case, $\langle F', F\rangle$ will be included in the context's EA-lattice, but not in the context's concept lattice in which case the concept $\langle F', F''\rangle$ will be labelled with the attribute f. Consider for example concepts MZ or sk in the EA-lattice in figure 2.3 and compare them to the formal concept lattice of the same context of figure 2.2. The concepts in the EA-lattice in figure 2.3 that correspond to the formal concepts in figure 2.2 are shaded and are identified by the fact that they have only one parent or child concept. In this example the zero concept is the same for the EA-lattice and formal concept lattice but not the unit concept. The EA-lattice is therefore essentially a generalisation of a formal concept lattice in the sense that it possibly contains a number of additional concepts. These additional concepts correspond **only** to objects (condition 1), attributes (condition 2), the unit concept (condition 3) or the zero concept (condition 4).

An *intermediate concept* of an EA-lattice is defined as a concept with more than one object in its extent and more than one attribute in its intent. Applying the same definition of intermediate concepts to formal concept lattices it can be seen from the definitions of both kinds of lattices of that the sets of intermediate concepts of the two types of lattices are identical.

It is important to note that an EA-lattice cannot use ordering relation $\leq_c$ since it is not antisymmetric for the given set of concepts. In the example above the antisymmetric property $(x \leq y \wedge y \leq x \Rightarrow x = y)$ does not hold for $\leq_c$ in regard to the concept $nc = \langle\{BN, MZ, SW, RD\}, \{nc\}\rangle$ and $n_4 = \langle\{BN, MZ, SW, RD\}, \{nw, nc\}\rangle$. The slight modification of the partial ordering relation (from $\leq_c$ to $\leq_{EA}$) is therefore due to the fact that, in an EA-lattice, the intent or the extent of some attribute concepts and object concepts may be the same as some intermediate concepts. The supremum and infimum also need to be recomputed in terms of the revised partial ordering and may thus differ from those of a formal concept lattice. (The EA-lattice may for example have a different unit element compared to its

corresponding formal concept lattice, as in the Living Context example). The EA-lattice is however for all practical purposes the same as a formal concept lattice (e.g. in lattice construction algorithms where only slight modifications are required).

From the definition it is easy to show that an EA-lattice has the following properties:

- There is one concept associated with each object e_i in the form $\langle E, E' \rangle$ where $E = \{e_i\}$. Similarly there is one associated concept for each attribute a_j in the form $\langle F', F \rangle$ where $F = \{a_j\}$. These concepts are called the *associated object concept* and the *associated attribute concept* respectively.
- Zero concept, $0_L = \langle \varnothing, A \rangle$ and unit concept, $1_L = \langle O, \varnothing \rangle$ are distinct from the object and attribute concepts and always have the same structure (i.e. empty extent and intent respectively).
- The EA-lattice, excluding the unit and zero concepts, can be partitioned into three sets corresponding to objects, attributes and intermediate concepts. The three sets correspond to the top, bottom and middle sections of the line diagram of a lattice (refer to figure 2.3).
- All EA-formal concepts in the formal concept lattice of a corresponding EA-lattice are contained in the EA-lattice. An EA-lattice thus has at least the same number of concepts as a formal concept lattice. Since the number of concepts in a large lattice is dominated by the intermediate concepts, the size of an formal concept and EA-lattice differs very little for large lattices. The difference in algorithmic complexity for their respective construction is also negligible (refer to chapter 5).
- The EA-lattice may contain a number of concepts in addition to those of the corresponding formal concept lattice. These additional concepts will correspond to the objects, attributes, unit concept or zero concepts, but there are no additional intermediate concepts.
- Each intermediate concept in an EA-lattice has at least two parent- and two children concepts. (Note that this is also true for non-atom and non-coatom concepts in boolean lattices but is not true in general for all FCA lattices such as the lattice in figure 2.2). Attributes concepts have only one parent (1_L) and only the attribute concepts generated by condition 5 have at least two child concepts. Other attribute concepts have one child concept. Similar dual observations can be made for object concepts.
- Each attribute has no parent concepts other than 1_L.
- Each object concept has no children concepts other than 0_L.
- Attribute concepts in the EA-lattice that do not correspond to a concept in the formal concept lattice have only one child concept. (Note that the corresponding concept in the formal concept lattice has only one parent.)
- Object concepts in the EA-lattice that do not correspond to a concept in the formal concept lattice have only one parent concept. (Note that the corresponding concept in the formal concept lattice has only one child concept.)
- In an EA-lattice the atoms and coatoms have a one-to-one relationship to the object- and attribute concepts of that lattice respectively.

In drawing the EA-lattice a number of conventions will be followed:

- The 1_L and 0_L concepts and associated cover relationships will not be shown so that the lattice is bounded from above by the attributes of the lattice and bounded from below by the objects of the lattice.
- The attribute and intermediate concepts are labelled with their intent only whilst object concepts are labelled with the object identifier itself.
- When convenient or to simplify the line diagram of a lattice, the labels of the concepts (especially the intermediate concepts) will be substituted for a concept number. From time-to-time a concept number and attribute list may also be used in conjunction especially for labelling intermediate concepts.

These conventions do not impact on the generality of the discussions or line diagrams since the complete line diagram and complete labels for concepts can be easily determined by inspecting the lattice using the definition of an EA-lattice (e.g. the intent of a concept is all the attributes contained in its upward closure).

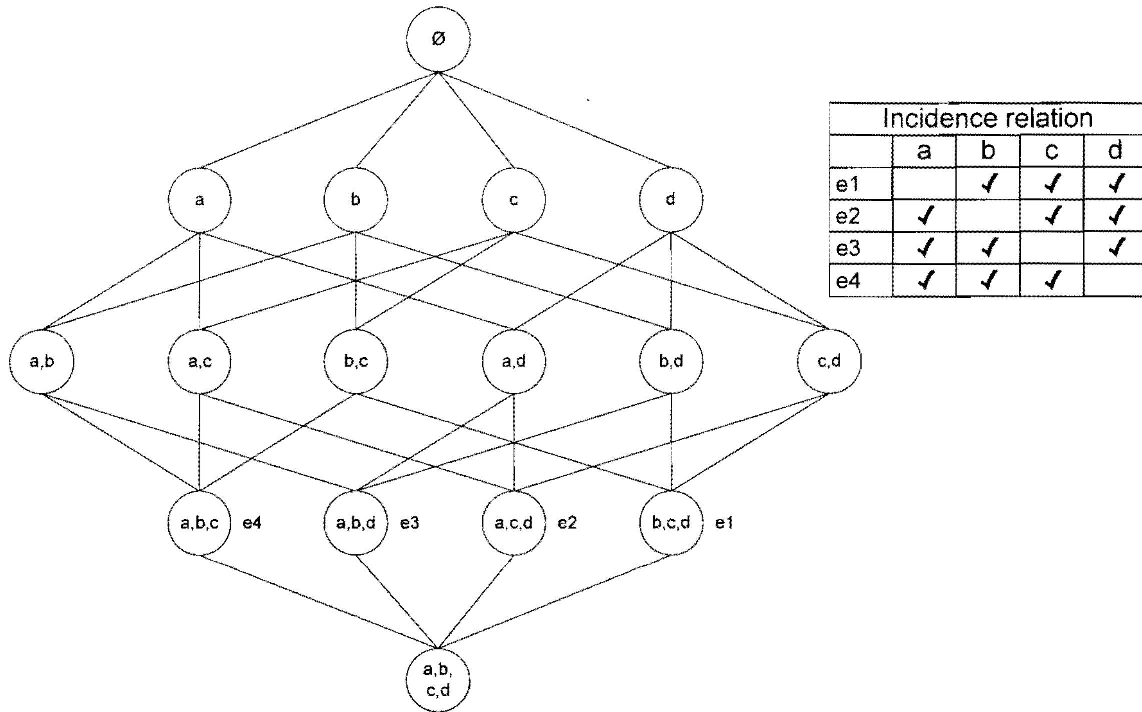
As can be expected from the similarity in the definitions of the formal concept- and EA-lattices there is a direct mapping from the one to the other.

2.6 BOOLEAN LATTICES

A power-set lattice (Ganter and Wille (1999)) of a set of attributes A is the lattice $\langle \mathcal{P}(A), \subseteq \rangle$. A *Boolean lattice* is a lattice that is isomorphic to some power-set lattice. The figure below is the Boolean formal concept lattice of $A = \{a, b, c, d\}$. The corresponding incidence relation shows that a Boolean lattice is formed when there are as many objects as attributes and each object differs from each of the other objects by only one attribute. Since a Boolean lattice contains all concepts from $\mathcal{P}(O) \times \mathcal{P}(A)$ it follows that a Boolean lattice is both a formal concept- and an EA-lattice. The converse is however not true in general. Formal concept- and EA-lattices are not in general Boolean lattices.

Since a Boolean lattice contains all the possible concepts that can be formed with a given set of attributes, it also forms the theoretical upper limit of the size of a lattice with the given set of attributes. A Boolean lattice has exactly the same number of concepts as the elements of $\mathcal{P}(A)$. Since $\mathcal{P}(A)$ is exponential in terms of $\|A\|$ it follows that the number of concepts in a Boolean lattice is exponential in terms of $\|A\|$ and has $2^{\|A\|}$ elements (refer to chapter 5 for more formulas related to the various size aspects of a Boolean lattice).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



Incidence relation				
	a	b	c	d
e1		✓	✓	✓
e2	✓		✓	✓
e3	✓	✓		✓
e4	✓	✓	✓	

Figure 2.4: Boolean lattice with four attributes and objects

2.7 AUGMENTED LATTICES

In applications such as machine learning and data mining that use formal concept lattices and EA-lattices it is common to augment or label the concepts of a lattice with additional meta-information that can be used in the application. These lattices are called augmented lattices (Kourie and Oosthuizen (1998)).

Examples of such meta-information could be:

- The support of a concept.
- A reference to a external database for object concepts.
- A name or descriptor for attribute and object concepts.
- Pre-computed upward and downward closures.

2.8 DIFFERENT VIEWS OF A LATTICE

In describing lattices, their properties and their construction, there are two different but essentially equal "views" of a lattice. One view of a lattice is to describe the lattice from a set theoretic point of view and the other from a graph theoretic point of view.

2.8.1 Set-theoretic view of lattices

The set-theoretic view of lattices emphasises the fact that a lattice consists of a set of concepts. Using the convention that we refer to a concept only by its intent (and assuming that there are no objects with the same intent), a formal concept- or an EA-lattice is

essentially a set of sets (i.e. a set of concepts in which each concept is a set of attributes). In Kourie & Oosthuizen (1998), for example this view is taken.

Since concepts are merely sets, set operations such as union and intersection can be performed directly on (the intents of) concepts. The supremum of two concepts with intents A and B in a Boolean lattice is for example the concept with the intent $A \cap B$. The cover relationship between concepts is defined by set containment. One of the key aspects of the set-theoretic view is that the lattice is described in terms of its attributes (sometimes the objects) and since there are far fewer attributes than other (intermediate) concepts most computations are very efficient.

The disadvantage of the set-theoretic view is that we emphasise the set related properties of lattice concepts in favour of the graph-related properties especially the child-parent relationship of nodes in a graph. The advantage is however that lattice properties, operations and theorems are more easily proved because the lattice is essentially a construct defined in terms of sets, which have more easily manipulated mathematical relationships.

2.8.2 Graph-theoretic view of lattices

The graph-theoretic view of lattices emphasises the lattice as a collection of nodes (as opposed to sets of attributes) with the specific partial ordering relationship between them depicted by the arcs in the graph. The lattice is thus described in terminology such as “parent”, “child”, “upward closure”, “downward link”, “top”, “bottom” etc. This greatly increases the understanding of the lattice concepts by newcomers to the subject since it refers to a graphic representation of the lattice namely the line diagram rather than the more abstract concept of a set of partially ordered sets of sets. In this view the intent and extent of concepts as well as the context can be inferred by closure operations. Although this view describes exactly the same lattice, different aspects of the lattice (in this case the line diagram representation) are promoted.

To distinguish between the two views we will follow the convention of referring to “concepts” when using the set-theoretic view and to “nodes” when using the graph theoretic view. Graph terminology such as “arcs”, “parent node” and “child node” is also freely used when taking a graph theoretic view.

Often the graph properties of the lattice are emphasised in that nodes are numbered and referred to by number in-stead of their intent, extent or both. The intent and extent of the node is often not explicitly shown and must then be derived by the graph properties using closure operations.

The disadvantage of the graph-theoretic view is that the mathematical properties of the lattice may be obscured and compared to the set-theoretic view the proving of theorems is not straightforward (if one would be restricted to graph terminology only). An example of this is to be seen in Oosthuizen (1994b) where both the construction and application of lattices to machine learning from a graph-theoretic view are described. The graph-theoretic view is essentially a description of the lattice as a data structure consisting of nodes that have a number of node properties such as its extent, intent, child nodes and parent nodes.

In the set-theoretic view, a set of concepts can be proven to be a lattice by verifying that a unique infimum and supremum exist for any set of nodes. To do the same in the graph-theoretic view, the line diagram of the lattice may be inspected and the arcs leading upward or downward from a set of nodes are followed until they meet. If, for example, the arcs leading upward join at two or more nodes that are unrelated (i.e. the one is not a parent of the other), there does not exist a unique smallest upper bound and therefore the

lattice property does not hold. A similar inspection should be done to verify the uniqueness of the greatest lower bound of each set of nodes. In figure 2.5 below both n_3 and n_5 are upper bounds for $\{n_7, n_8\}$ as can be seen by following the arcs in bold. However, n_3 and n_5 are unrelated. (In this example, it is of course assumed that there is in fact no top element – i.e. the assumption previously mentioned that the top is implicit has been lifted.) There is therefore not a unique least upper bound for $\{n7, n8\}$ and the graph therefore does not represent a lattice. Oosthuizen (1991) refers to the subgraph consisting of n_8, n_5, n_7, n_3 and the connecting arcs as a *quad* and describe the problem of algorithmically constructing a lattice as one in which an acyclic graph is constructed to connect all objects and attributes without any quads (i.e. every pair of nodes has a join and meet and that these should necessarily be unique). Note that any number of intermediate nodes can exist between the four "corner" nodes of a quad.

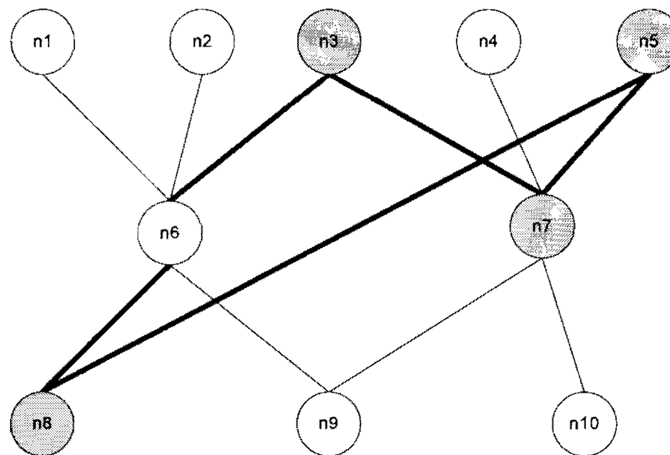


Figure 2.5: Example of a poset with non-unique infima and suprema of concepts n_3, n_5, n_7, n_8 (also called a quad)

Although the two viewpoints are essentially the same and there are many ways to combine the two views they represent the two main ways in which authors have chosen to describe lattices and especially the construction and application thereof. Authors such as Godin (1991) and Carpineto and Romano (1996b) tend to emphasise the set-theoretic view whilst Oosthuizen (1991) has favoured the graph-theoretic view in the description of their lattice construction algorithms.

Further on in this text it will be argued that the key to the AddAtom lattice construction algorithm is was developed using a more graph-theoretic view. The graph-theoretic view emphasises the use of the information that is contained in the lattice – the partial ordering of nodes and therefore the characteristics of the context from which the lattice was constructed.

In order to gain the advantages of both these views we will use both in various sections of this text. The set-theoretic view will be used to formally describe the lattice and its construction whilst the graph theoretic view will be used to informally describe the lattice construction concepts.

2.9 INTENT AND EXTENT OPERATIONS

In this section we define the *approximate intent representative* set (AIR) as well as the *exact intent representative* set (EIR) of a set of attributes with regards to a concept lattice or complete concept sublattice. These operations define in some sense a 'second-order

i16542654
b15954390

meet operation' or 'second-order infimum' for concept lattices. The intent- and extent representative operations were originally defined in the context of compressed pseudo-lattices or concept sublattices where the infimum and supremum of some concepts in L have been removed (such lattices are under consideration in chapter 6). Since the removal of concepts from a lattice is the opposite of lattice construction there is a relationship between the two concepts. This relationship is explored in section chapter 6.

Let $Q \subseteq A$ be a set of attributes and H be the set of associated attribute concepts of Q in an EA-lattice L. The meet of the set of attribute concepts H, Meet(L, H) is a concept. If Intent(Meet(L, H)) = Q then the Meet(L, H) is called an *exact meet* (or *exact infimum*) of the attributes in Q since it spans only the attribute concepts H and contains only elements of Q in its intent. If the meet contains additional attributes in its intent or alternatively spans attribute concepts other than H, then it is called an *approximate meet* (or *approximate infimum*) of the attributes. Similarly an *exact join* (or *exact supremum*) and *approximate join* (or *approximate supremum*) can be defined using a set G of object concepts associated with a set R of objects in stead of the set H of attribute concepts associated with the set Q.

Since the elements of the set of attributes, A, are not elements of an EA-lattice or concept lattice, L, the meet or infimum of a subset of A in L is not defined. Furthermore, since lattices of which concepts have been removed will be considered it is useful to define Inf'(L, Q), $Q \subseteq A$ as the maximal elements[4] of the set $\{x : L|\ \text{Intent}(L, x) \supseteq Q\}$ (i.e. the set of greatest concepts that contains at least the attributes of Q in their intents). This function is closely related to the infimum or meet of a set of associated attribute concepts. In an EA-lattice Inf'(L, Q) = Inf(L, H) where H is the set of attribute concepts associated with Q, but this function is also defined on lattices of which concepts (e.g. the associated attribute concepts themselves) have been removed and in which case Inf'(L, Q) may not be a single concept. Similarly Sup'(L, R) is the minimal elements of $\{x : L|\ \text{Extent}(L, x) \supseteq R\}$ where R is a set of objects.

Consider a concept lattice or -sublattice, L. Let Q be a set of attributes, $Q \subseteq A$. Let S be the set of all elements of Inf'(L, F), $F \in \mathcal{P}(Q)$ (i.e. $S = \{x : L \mid \exists\ F \in \mathcal{P}(Q),\ x \in \text{Inf'}(L, F)\}$). Exclude the zero concept from S. The *set of approximate intent representatives* of Q in L, denoted by AIR(L, Q), is the set of minimal concepts in S.

Now let T be that subset of S whose elements have intents that are not subsets of Q. The *set of exact intent representatives* of Q with respect to L, denoted by EIR(L, Q), is the set of minimal elements in S – T. If $T = \varnothing$ then clearly EIR(L, Q) = AIR(L, Q).

From the definition it follows that the results of both AIR(L, Q) and EIR(L, Q) are anti-chains. They are said to be *infimum-dense*[5] and are therefore a concise way of representing Q. In the case of the exact intent representative there is a close relationship between Q and EIR(L, Q) since the intents of elements of EIR(L, Q) not only span the associated attribute concepts of Q, they constitute in fact, the minimal set of meets that do so. There is not necessarily such a direct mapping between Q and AIR(L, Q) in the sense that concepts in the intents of elements of AIR(L, Q) possibly contain attributes in addition to those in Q.

Dual extent operations for AIR(L, Q) and EIR(L, Q) can be defined as follows. R can be seen as a set of objects (instead of Q, the set of attributes) and Inf' and the maximal operations can be substituted by Sup' and minimal operations in the above definitions

---

[4] $x \in S$ is minimal, iff $\nexists y \in S,\ y \neq x$; such that $y \leq_{EA} x$. Similarly $x \in S$ is maximal, iff $\nexists y \in S,\ y \neq x$; such that $x \leq_{EA} y$.

[5] A set $X \subseteq Y$ is called infimum-dense in Y if every element from X can be represented as the infimum of a subset of Y.

respectively. The zero concept is replaced by the unit concept. This defines the set of *approximate extent representatives*, AER(L, R) and the set of *exact extent representatives*, EER(L, R). In an EA-lattice if Sup'(L, R) is non-trivial (i.e. if Sup'(L, R) is not 1_L), Sup'(L, R) = AER(L, R) = EER(L, R).

It is useful to define a further related set of operations, namely EIR(L, Q, c), $c \in L$. This is the set of *exact intent representatives of* Q *not less than* c. It corresponds identically to EIR(L, Q), except that in determining the minimal elements of S above, the downward closure of a designated concept, c, is specifically excluded from consideration. As a result, if c is in EIR(L, Q), then EIR(L, Q, c) contains no concepts that are less than or equal to c. In particular, if c = Meet(L, Q), then EIR(L, Q, c) is the set of concepts covering c in the sublattice L. The set of *exact extent representatives of* R *not greater than* c, EER(L, R, c) is defined similarly. It is easy to see that EIR(L, Q) = EIR(L, Q, 0_L). Similarly EER(L, R) = EER(L, R, 1_L). In the same way, the set of *approximate intent representatives of* Q *not less than* c, AIR(L, Q, c) is defined. The set of *approximate extent representatives of* R *not less than* c is similarly defined.

The operations defined here are collectively referred to as the *intent- or extent operations* of a concept lattice (or sublattice).

For example in figure 2.3 calculating AIR(L, {nw, nc, 1lg, 2lg}), S = {nw, 2lg, nc, 1lg, n4, n6, BN}. {BN, n_6} is the set of minimal elements of S and therefore AIR(L, {nw, nc, 1lg, 2lg}) = {BN, n_6}. In calculating EIR(L, {nw, nc, 1lg, 2lg}) we see that T = { BN } since BN also spans ll in addition. The minimal elements of S – T = { nw, 2lg, nc, 1lg, n_4, n_6 } is { n_6, 2lg} and therefore EIR(L, {nw, nc, 1lg, 2lg}) = { n_6 }. Chapter 6 provides more examples.

2.10 SUMMARY

In this chapter the basic building blocks and definitions that are key to formal concept analysis and that will be used in the rest of this text have been defined. This includes the notion of a lattice, sublattice, formal concept lattice and EA-lattice. A number of different operations have also been defined on these structures, this include the intent- and extent representative operations.

Chapter 3: Lattice construction

This chapter discuss the considerations when algorithmically constructing a concept lattice or rather the line diagram of the concept lattice. This is done through the formulation of an ineffective lattice construction algorithm.

3.1 ALGORITHMIC LATTICE CONSTRUCTION

Lattice construction algorithms use A, O and I of a context C = ⟨O, A, I⟩ as input (or a labelled version of cross table as shown in the incidence relation of the Living Context in figure 2.2 in chapter 2). All the concepts are discovered and connecting arcs representing the cover relationship are constructed between appropriate pairs of concepts. The basic output of such an algorithm is thus a set containing the concepts of the lattice as well as a set of arcs connecting these concepts.

3.2 INCREMENTAL VS. BATCH LATTICE CONSTRUCTION ALGORITHMNS

There are two basic strategies for algorithmically constructing a concept lattice form I. The first is to consider the whole context and construct the lattice in a non-incremental or batch way (i.e. if more objects are added to the context, the whole lattice must be reconstructed from the start). (Bordat (1986), Chein (1969), Ganter (1984), Kuznetsov (1993), Lindig (1999, 2000), Zabezhailo et al. (1987)) have followed this strategy. The second strategy is to incrementally build the lattice, adding objects to the lattice until the lattice of the whole context is constructed. In each invocation of an incremental algorithm an existing lattice L_i is used as input. The new object is added to the lattice along with any new concepts and attributes that may be required to create a new lattice L_{i+1}. Therefore unlike the non-incremental strategy, a valid lattice exists after each iteration of the algorithm. The incremental algorithm will also modify the arcs of L_i to create L_{i+1}. Godin (1991), Carpineto and Romano (1993, 1996b), Oosthuizen (1991), Dowling (1993) and Norris (1978) have followed an incremental lattice construction strategy (refer to section 5.1 for references to more algorithms).

The main advantage of incremental lattice construction algorithms is that new objects can efficiently be added to the lattice without rebuilding the whole structure and therefore the incremental construction algorithms are often used. This may however be at some expense since the algorithm cannot optimise across all objects in the context.

3.3 CONSTRUCTING THE LINE DIAGRAM

A number of published non-incremental lattice construction algorithms only generate the set of concepts of the lattice and do not generate the line- or Hasse diagram that represents the cover relationships between lattice elements. Although there are applications such as the generation of all implication rules of a context in which only the set of concepts is used, the majority of lattice-based applications do explicitly use the

ordering of concepts in terms of generalisation and specialisation. This ordering is after all one of the primary benefits of lattice-based applications.

For the purposes of comparing the algorithmic performance of various lattice construction algorithms a common framework is required and therefore it is argued that only construction algorithms that do produce the line diagram of the lattice should be considered since they have a more general application. Kuznetsov and Obiedkov (2002) have adapted a number of the non-incremental algorithms that do not generate the line diagram to generate it.

3.4 AN INEFFICIENT BATCH LATTICE CONSTRUCTION ALGORITHM

A simple way to demonstrate the basic considerations and challenges in concept lattice construction algorithms is to consider a very inefficient construction algorithm which uses the basic definition of an EA-lattice to construct the lattice data structure. (Note that the algorithm can be easily adapted to formal concept lattices by changing the conditions for testing.)

The BruteForceEAConstruct algorithm below, "blindly" applies the definition of the EA-lattice in a very inefficient way. From the definition of EA-concepts it is clear that all the concepts of an EA-lattice can be discovered by enumerating and inspecting all possible combinations of $E \subseteq O$ and $F \subseteq A$ (i.e. $\mathcal{P}(O) \times \mathcal{P}(A)$) and then inspecting each couple $\langle E, F \rangle$ to test whether it is EA-formal. Once all the lattice concepts have been discovered, the definition of the cover relationship between concepts is used to test each pair of concepts to determine whether they cover each other. As one might expect this algorithm is very inefficient since it inspects all possible combinations of attributes and objects without having any strategy to prune the search space.

```
//===================================================================
Function BruteForceEAConstruct (anObjSet, anAttrSet,
                anIncidenceRelation) Return aLattice
//===================================================================
CreateNewLattice(L)
L.Concepts = ∅
For ∀ E ∈ P(anObjSet)
   For ∀ F ∈ P(anAttrSet)
      // Determine E'
      E' = ∅
      For ∀ o ∈ E
         For ∀ a ∈ anAttrSet
            If oIa with I = anIncidenceRelation then E' = E' ∪ {a}
         Rof
      Rof
      // Determine F'
      F' = ∅
      For ∀ a ∈ F
         For ∀ o ∈ anEntSet
            If oIa with I = anIncidenceRelation then F' = F' ∪ {o}
         Rof
      Rof
      // Test against EA-formal concept definition
      If ||E|| = 1 and F=O' then L.Concepts = L.Concepts ∪ {⟨E, F⟩}
      If ||F|| = 1 and E=A' then L.Concepts = L.Concepts ∪ {⟨E, F⟩}
      If E = ∅ and F = anAttrSet then L.Concepts =L.Concepts ∪ {⟨E, F⟩}
      If F = ∅ and E = anObjSet then L.Concepts = L.Concepts ∪ {⟨E, F⟩}
      If F'= E and E'= F then L.Concepts = L.Concepts ∪ {⟨E, F⟩}
   Rof
Rof
// Discover the cover relationships
L.Cover = ∅    // L.Cover is a set of concepts in the form ⟨b, c⟩ this
               // assumes a unique symbol is associated with each
For ∀ x ∈ L.Concepts // x and y are concepts in the form ⟨E, F⟩
   For ∀ x ∈ L.Concepts, x ≠ y, x ≤EA y
      CoverFlag = True
      For ∀ z ∈ L.Concepts, z ≠ x, z ≠ y
         If x ≤EA z ≤EA y then CoverFlag = False
      Rof
      If CoverFlag then L.Cover = L.Cover ∪ {⟨x, y⟩}
   Rof
Rof
Retrun L
End BruteForceEAConstruct
//===================================================================
```

Because of the non-specific and unfocussed way in which the algorithm creates concepts and arcs it soon becomes hopelessly inefficient for all but the smallest of contexts.

This algorithm does however address the basic functions of a concept lattice construction algorithm:

- Generating all the concepts of the lattice and representing them in a data structure.

- Discovering the cover relationship between the concepts and representing it in a data structure.

There is however a number of other issues that need to be addressed in order to be more efficient:

- Avoiding the generation of duplicate concepts or at least determining whether a concept is generated for the first time.
- Efficiently testing and/or generating the cover relationships.
- Efficiently searching for concepts in the set of concepts that has been generated up to that point.
- Avoiding the generation of cover relationships that might be deleted later in the algorithm.

One important property of the algorithm is that it is non-incremental in that we cannot incrementally construct the lattice by starting with a lattice and adding a new object to create a new lattice – the whole lattice needs to be constructed anew. With an incremental algorithm we can create a lattice L_{n+1} with $n + 1$ objects by taking the lattice of n objects, L_n and add the $n + 1$'th object. The key to the incremental algorithm is the observation that the generated lattice (L_{n+1}) always contains all the concepts of the original lattice (L_n) but concepts were either added, modified or left unchanged. Such an algorithm will discover and new concepts and arcs needed to create L_{n+1} as well as deleting arcs where the newly created concepts redefine the cover relationship between specific concepts. This is the strategy followed by the AddAtom algorithm defined in the next chapter.

Another important property of the algorithm is that it constructs the set of all concepts of the lattice as well as the line diagram of the lattice. In applications such as those using association rules where only the set of concepts are required, the last part of the algorithm can be skipped. Section 5.1 lists a number of published lattice construction algorithms and compare them with regards to their key characteristics.

Although this algorithm can be improved upon in a number of ways, we will not pursue this further but will instead define a new algorithm using more elegant strategy in the next chapter.

Chapter 4: The AddAtom lattice construction algorithm

In this chapter we describe and define a concept lattice construction algorithm called AddAtom. This is done in two parts. In the first, in section 4.1, we give an informal description of the strategies used in the AddAtom lattice construction algorithm using a graph theoretic view. Then, in section 4.2, we describe the relation of the intent- and extent representative operations defined in chapter 2 to lattice construction and show that these operations have a direct relationship to the structural properties of a lattice. In section 4.3 the AddAtom algorithm is formally defined in pseudo code using a set theoretic point of view. This is followed by an example of the execution of the algorithm (section 4.4). Since the algorithm defined in section 4.3 is very inefficient as stated, section 4.6 considers efficient implementations of the algorithm derived from an efficient algorithm for determining the intent- and extent representative operations (section 4.5). The chapter concludes with a general discussion of the algorithm (section 4.7).

4.1 INFORMAL DESCRIPTION

This section is an informal discussion of the AddAtom concept lattice construction algorithm. The description incrementally builds an understanding of the algorithm by describing the various strategies used in the algorithm. This approach is taken to give the reader an intuitive understanding of lattice construction without trying to decipher the concepts of a more formal description. In the next section a formal description of the algorithm is given. Readers familiar with lattice construction may wish to skip this section.

As a starting point, an observation that can be made about the inefficient algorithm defined in the previous chapter (BruteForceEAConstruct) is that it ignores the information already contained in the lattice L_n. The algorithm computes all concepts and consider each as possibilities regardless of whether there is a likelihood of finding any new EA-formal concept or not. However by inspecting the nodes and arcs in L, the creation of a number of concepts could have been avoided (e.g. generating only combinations of attributes that actually occur in I). The process of creating arcs could also be significantly improved by using the "information already contained in the lattice". This idea of using the information already contained in the lattice is the key to the AddAtom algorithm. It is therefore worthwhile to take a closer look at the lattice before defining the algorithm in order to see how the lattice itself can be used to more efficiently construct L_{n+1}.

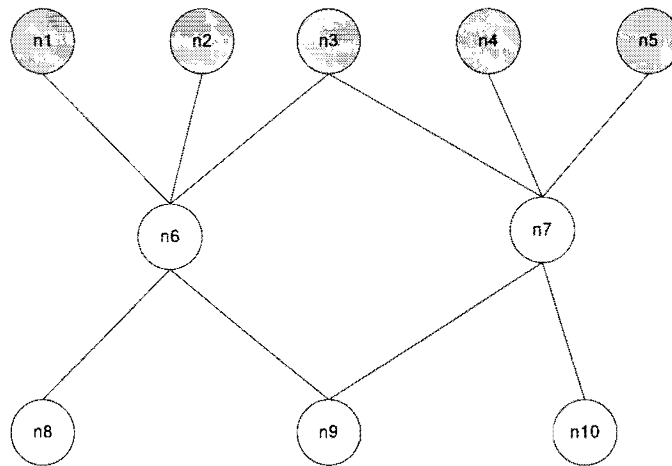


Figure 4.1: Nodes in a lattice are connected to the meet of subsets of their intents

In general, any node is always connected to the meet of some subset of the attributes in its intent. For example node n_9 in figure 4.1 has an intent of $\{n_1, n_2, n_3, n_4, n_5\}$. In this case n_9 is connected to n_6 and n_7, the meets of $\{n_1, n_2, n_3\}$ and $\{n_3, n_4, n_5\}$ respectively. Since the node itself is the meet of all the attributes in its intent it seems that if we want to insert a new object node e into the lattice, we must find the meets $m_1 \ldots m_j$ of all subsets of n's intent and connect the new node to some of these meets. However if such a meet is spanned by another meet (not the unit concept), lower down in the lattice, it must be ignored. Only the lowest, or minimal, meets should be taken.

In the figure 4.2 object node e with intent $A = \{a, b, c, d\}$ was inserted into a lattice (in the following, the intent attributes of all objects to be inserted are shaded in grey). The set of the meets of all the subsets of A is $\{a, b, c, d, m_1, m_2, m_3\}$. Since a, b, c, d, m_1 are covered by either m_2 or m_3 they can be ignored and e only connected to m_2 and m_3. By inspection, it can be verified that the resulting line diagram is indeed a lattice in that the supremum and infimum of any pair of concepts are unique (keep in mind that the unit and zero nodes were omitted in the figure but are implied).

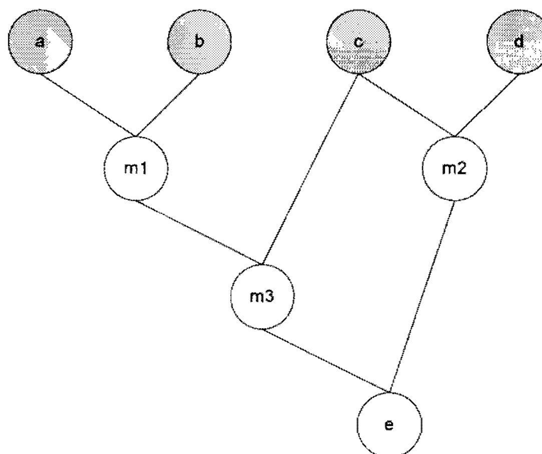


Figure 4.2: node e with intent $A = \{a, b, c, d\}$ was inserted into a lattice by connecting it to m_2 and m_3

This observation suggests a possible lattice construction algorithm. The approach is to find the minimal meets of Intent(o) (i.e. all the meets of all possible subsets of Intent(o), excluding the unit node, not spanned by another meet). This set of nodes can be found by computing the set of meets of all possible subsets of Intent(o) and then removing the zero

node and any other node that is spanned by another node lower down in the lattice. Note that this corresponds to the definition of the approximate intent representatives of Intent(o) or AIR(L, Intent(o)) defined in chapter 2.

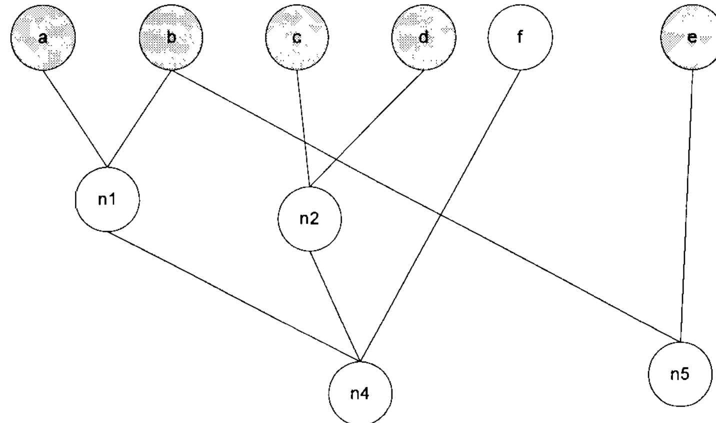


Figure 4.3: Lattice before inserting node m with intent Intent(m) = {a, b, c, d, e} to create lattice in figure 4.4

This approach to a lattice construction algorithm does however not always function correctly. Consider the lattice in figure 4.3 and suppose that the node m with intent Intent(m) = {a, b, c, d, e} is inserted into the lattice. Using this approach it creates the lattice in figure 4.4, i.e. because the set of approximate intent representatives of {a, b, c, d, e} in figure 4.3 is $\{n_4, n_5\}$, m is connected to both n_4 and n_5 as shown in figure 4.4. (To aid the readability of the figures, the newly inserted nodes are shown in black.) On closer inspection, we see that m has now gained an extra attribute in its intent namely f via node n_4 (i.e. instead of m's intent being {a, b, c, d, e} as was intended, it is in fact {a, b, c, d, e, f} in figure 4.4). It thus seems as if this approach only works when the intent representative concepts span only attributes in Intent(m) (i.e. when they are exact). If not, then the intent of the new node could unintentionally be extended.

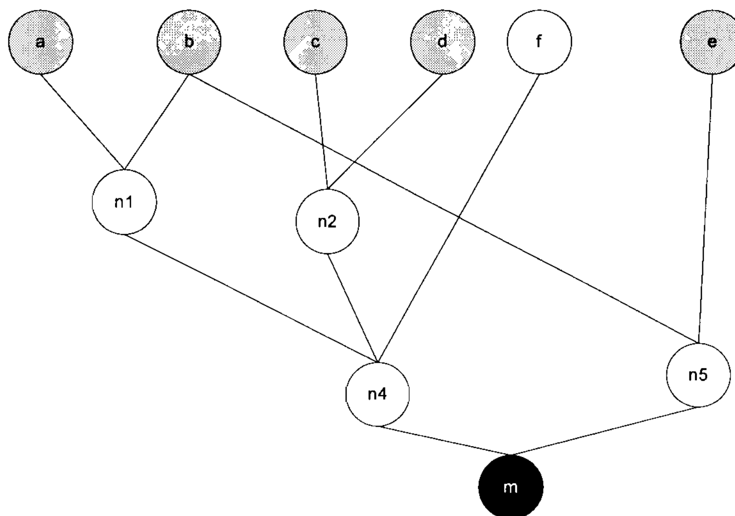


Figure 4.4: Lattice after inserting node m with Intent(m) = {a, b, c, d, e}, but showing that m now has f in its intent in addition

Since n_4 is not an *exact meet* of Intent(m) in figure 4.3, it cannot be connected directly to the new node. We might be tempted to connect m to n_1, n_2 and n_5, leaving us with the

graph in figure 4.5 below. But as indicated using thick arcs, both m and n_4 are lower bounds of $\{n_1, n_2\}$. Since the greatest lower bound of $\{n_1, n_2\}$ is non-unique, the lattice property does not hold and this approach is therefore also not correct.

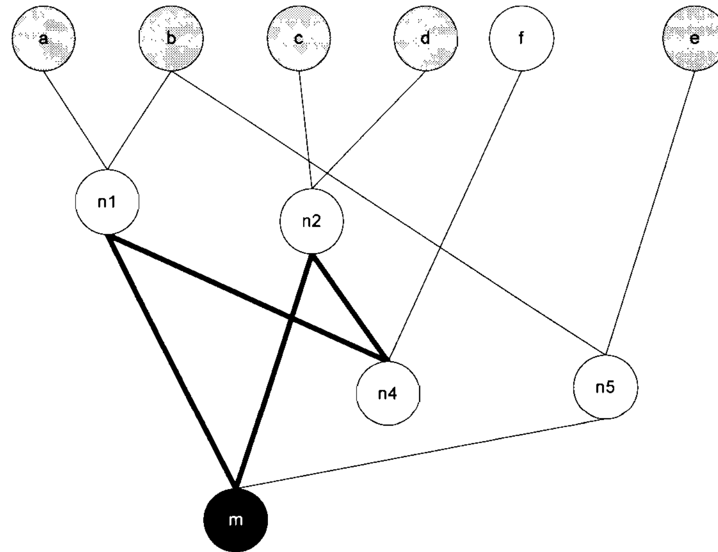


Figure 4.5: Connecting m to n_1 and n_2 creates multiple greater lower bounds of $\{n_1, n_2\}$

The solution to the problem lies in the creation of a new intermediate node n_3 spanning n_1 and n_2 and connecting n_4 and m to n_3 as in figure 4.6. In doing so arcs $\langle n_4, n_1\rangle$ and $\langle n_4, n_2\rangle$ had to be removed and the new arcs $\langle n_3, n_1\rangle$, $\langle n_3, n_2\rangle$, $\langle n_4, n_3\rangle$ and $\langle m, n_3\rangle$ had to be created.

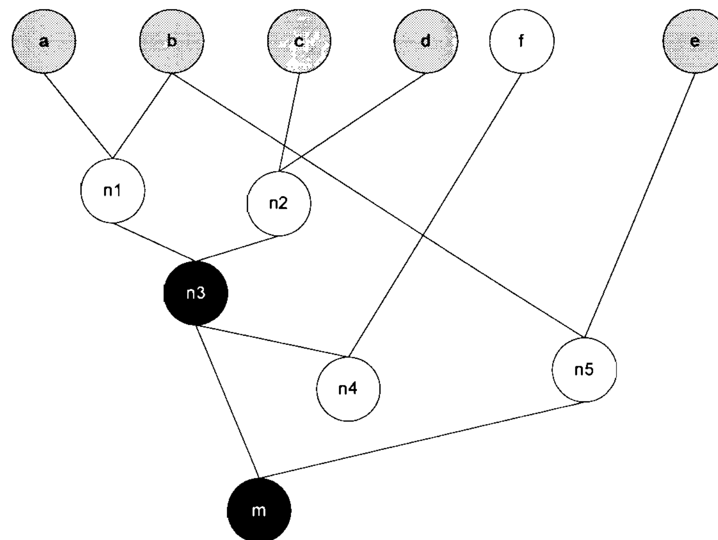


Figure 4.6: To insert m into the lattice a new node n_3 needs to be created

Although we define the AddAtom algorithm in a more formal way in the next section, the key to the algorithm is that new nodes can be directly linked to their exact intent representatives. Additional nodes must be inserted when the intent representatives are approximate. By doing this, we are in effect creating the exact meets of the intent when they do not already exist in the lattice.

The algorithm we are now informally defining needs one extra part: the process of creating the exact meets must be recursively applied. This is demonstrated in the

following example where a new node m with intent {a, b, d, e, f, h, g} must be inserted into the lattice in figure 4.7.

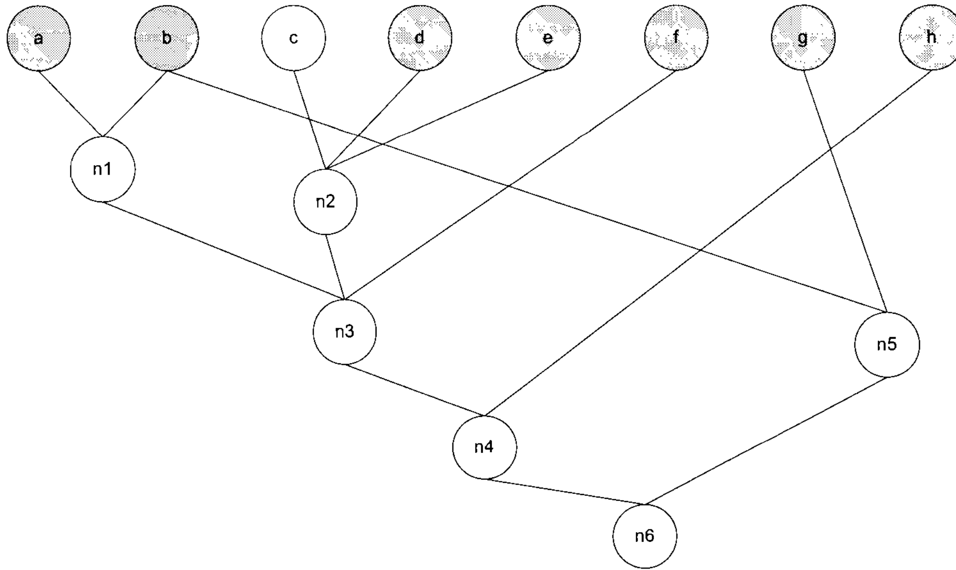


Figure 4.7: A lattice where a new node m with intent {a, b, d, e, f, h, g} must be inserted

The meet of {a, b, d, e, f, h, g} in the lattice in figure 4.7 is $\{n_6\}$. Since n_6 is approximate (it spans c in addition to {a, b, d, e, f, h, g}), a new node (n_{10}) with intent {a, b, d, e, f, g, h} must be created above n_6. This node creates an exact meet to which m can be connect to. However the same reasoning needs to be applied to n_{10} the insertion of itself – it should also be connected the minimal meets of {a, b, d, e, f, g, h} and these meets should be exact. However, when calculating the minimal meets of {a, b, d, e, f, g, h}, n_6 and nodes below it needs to be excluded from consideration. The set of minimal meets of {a, b, d, e, f, g, h} excluding n_6 is therefore $\{n_4, n_5\}$. Node n_5 is an exact meet of {a, b, d, e, f, g, h} and n_{10} can be directly connected to it. Node n_4 is however not exact and an additional node needs to be created above n_4 in the same way n_{10} was created above n_6 (refer to figure 4.8). The insertion of n_{10} can also be viewed as the insertion of an object with the intent of {a, b, d, e, f, h, g} into the sublattice of which n_6 is the zero node. In this context (i.e. n_6 is considered to be the zero node of a sublattice) the set of approximate intent representatives of {a, b, d, e, f, h, g} is $\{n_4, n_5\}$.

Recursively continuing with this process we see that n_3 and n_2 are also approximate meets. Each time such approximate meets are encountered a node is created above the approximate meet. The intent of the new node is that subset of the intent of the approximate meet where only those attributes that are in the intent of the original object (m) are kept. The nodes n_9, n_8 and n_7 are therefore created above the approximate meets n_4, n_3 and n_2 respectively resulting in the lattice in figure 4.8.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

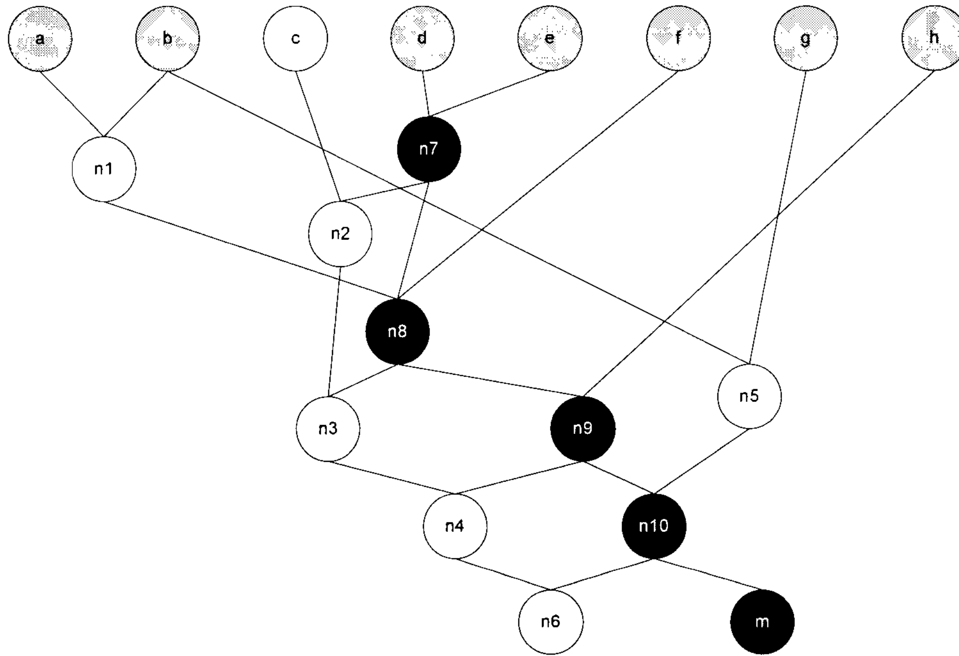


Figure 4.8: The lattice of figure 4.7 after inserting node m *with intent* $\{a, b, d, e, f, h, g\}$

4.2 INTENT- AND EXTENT REPRESENTATIVE OPERATIONS AND LATTICE CONSTRUCTION

At a high level of abstraction, lattice construction algorithms may be thought of as searching the space of all concepts (i.e. $\mathcal{P}(O) \times \mathcal{P}(A)$) to find all formal or EA-formal concepts. This can for example be done by intersecting the intents of the concepts and searching for sets of attributes each of which are not already present as the intent of some other concept. At a somewhat lower level of abstraction, an incremental lattice construction algorithm that inserts a new object o into a lattice L_i to create a new lattice L_{i+1} may be described (Valtchev and Missaoui 2001) as a search for three sets of concepts in L_i: *generator* concepts, $G(o)$, that give rise to new concepts; modified concepts, $M(o)$, whose arcs must be modified in order to integrate o into their extents; and old concepts, $U(o)$, that remain entirely unchanged. In addition, a set of new concepts $N(o)$ to be inserted into L_i to give L_{i+1} must also be constructed.

The discussion below will indicate that the intent representative operations may be deployed to identify generator, modified, old concepts and new concepts, and may consequently be used to construct concept lattices.

The intent representative operations reflect some of the properties of a lattice and its line diagram. For any concept c in a lattice L (potentially a concept sublattice), $EIR(L, Intent(c), c)$ **is** the set of parent concepts of c and therefore defines the cover relationships of c. This property is due to EIR being the minimal meets, not spanning c, that span only contains subsets of $Intent(c)$ in their intents. Similarly $EER(L, Extent(c), c)$ is the set of child concepts of c.

However, this property only holds for concepts that already belong to an existing lattice, L_i. When inserting a *new* object, o, into L_i to create L_{i+1}, it will not necessarily be true that the set $EIR(L_i, Intent(o), Inf'(Intent(o)))$ represents *all* the parent concepts of o in L_{i+1}. Indeed, an incremental lattice construction algorithm will invariably have to construct (or 'spawn') additional intermediate concepts that are not yet part of L_i. This is in order to achieve the objective that $EIR(L_{i+1}, Intent(o), o)$ is the set of parent concepts of c in L_{i+1}.

Furthermore, these additional intermediate concepts and their associated cover relationships in the new structure also have to comply with the lattice property in that any pair of concepts must have a unique infimum and supremum. Therefore in addition to creating the parent concepts of o, other concepts could be created recursively and connected higher up in the lattice in order for this uniqueness property to hold.

It can be shown that an incremental lattice construction algorithm that inserts an object o into a lattice L_i to give L_{i+1}, merely needs to intersect the intent of o with the intent of current concepts in L_i to determine the intent of concepts of L_{i+1}. Any intents of derived in this way that are not the intents of concepts in L_i are that of new concepts that must be added to L_i to derive L_{i+1}. Put differently, the intent of each of these new concepts corresponds to the intersection of Intent(o) with one of the concepts in G(o), the generator concepts for o. In fact, this property is precisely what determines a generator concept for o – that its intersection of its intent with Intent(o) gives the intent of a new concept. This is however a computationally inefficient way to construct lattices and hence the search for efficient construction algorithms.

For simplicity, we will not consider contexts and their corresponding lattices in which the intent of an object is a subset of the intent of some other object (i.e. it is assumed that objects are not comparable). Also assume that the extent of an attribute is not a subset of the extent of any other attribute (i.e. it is assumed that attributes are not comparable). In other words only contexts where the attributes and objects are the co-atoms and atoms respectively of the FCA lattice, and where the FCA lattice is therefore isomorphic to the EA-lattice are considered. This will not detract from validity of the discussion but will prevent the discussion from being cluttered by having to consider some exceptions associated with such contexts.

Consider inserting an object o into L_i. The trivial case is when there are no generator concepts except for the zero concept, 0_L. In this case all concepts in AIR(L_i, Intent(o), 0_L) are exact meets. The object should be inserted, as an atom, above 0_L and connected to its parent concepts as given by EIR(L_i, Intent(o), 0_L). 0_L is the object's only child concept. (Note that this is by virtue of the simplification of the context as described in the previous paragraph.) The extent of each concept in M(o) also needs to be updated as a result of the insertion of o.

If EIR(L_i, Intent(o), 0_L) $\neq$ AIR(L_i, Intent(o), 0_L) then there is at least one concept in L_i that is the meet of a subset of Intent(o) that spans attributes other than those in Intent(o) (i.e. the meet is not exact). All non-exact meets are elements of the set T in the definition of EIR(L_i, Intent(o), o) (refer to section 2.9). For each such non-exact meet, a new concept must be created whose intent corresponds to the intent of the generator concept less the additional attributes. Each such meet is a concept in G(o). Therefore, if EIR(L_i, Intent(o), o) $\neq$ AIR(L_i, Intent(o), o), generator concepts of o do exist in L. Indeed the concepts in the set AIR(L_i, Intent(o), o) - EIR(L_i, Intent(o), o) are all generator concepts, the intersection of the intent of each of these generator concepts with Intent(o) does not represent the intent of any concept already contained in L. (If it did, then that concept would be an element of EIR or AIR.) (Note that these are not the *only* generator concepts as explained below.) An incremental concept lattice construction algorithm can thus compute the minimal (but not all) concepts in G(o) if it can compute the intent representative operations. For the purposes of this discussion, it will be assumed that efficient algorithms to calculate AIR and EIR are indeed available.

The next 'level' of the elements of G(o) can be found by using the intents of the minimal concepts in G(o) restricted to Intent(o) as a generating set and then calculating their respective EIR and AIR sets (i.e. using Intent(g) $\cap$ Intent(o), g $\in$ G(o) for calculating EIR and AIR). This strategy can be recursively applied to calculate all elements of G(o).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

If all concepts in $G(o)$ are known, then the new concepts to be inserted can be determined as follows. Each element g of $G(o)$ gives rise to a new concept $n \in N(o)$ ($N(o)$ being the set of new concepts inserted in L_i to yield L_{i+1}) with $Intent(n) = Intent(g) \cap Intent(o)$ and $Extent(n) = Extent(g) \cup \{o\}$. Some of the parent concepts of n could be newly created concepts of $N(o)$ in L_{i+1} higher up in the lattice whilst the others are elements of $EIR(L_i, Intent(o), g)$. Before connecting n, all elements of $N(o)$ must be generated since n might be connected to one of them. The child concepts of n are given by $EER(L_{i+1}, Extent(n), n)$. g will be one of the child concepts but it could have additional child concepts. Each of these child concepts will be in $N(o)$ corresponding to another generator concept lower down in the lattice in a similar way as the parent concepts.

From this description it thus follows that the set of concepts in $G(o)$ is partially ordered. The concepts in $M(o)$ are all the exact meets of subsets of $Intent(o)$ in L_0. Elements of $G(o)$ are all approximate meets of $Intent(o)$. The elements of $U(o)$ are those concepts not in either $G(o)$ or $M(o)$.

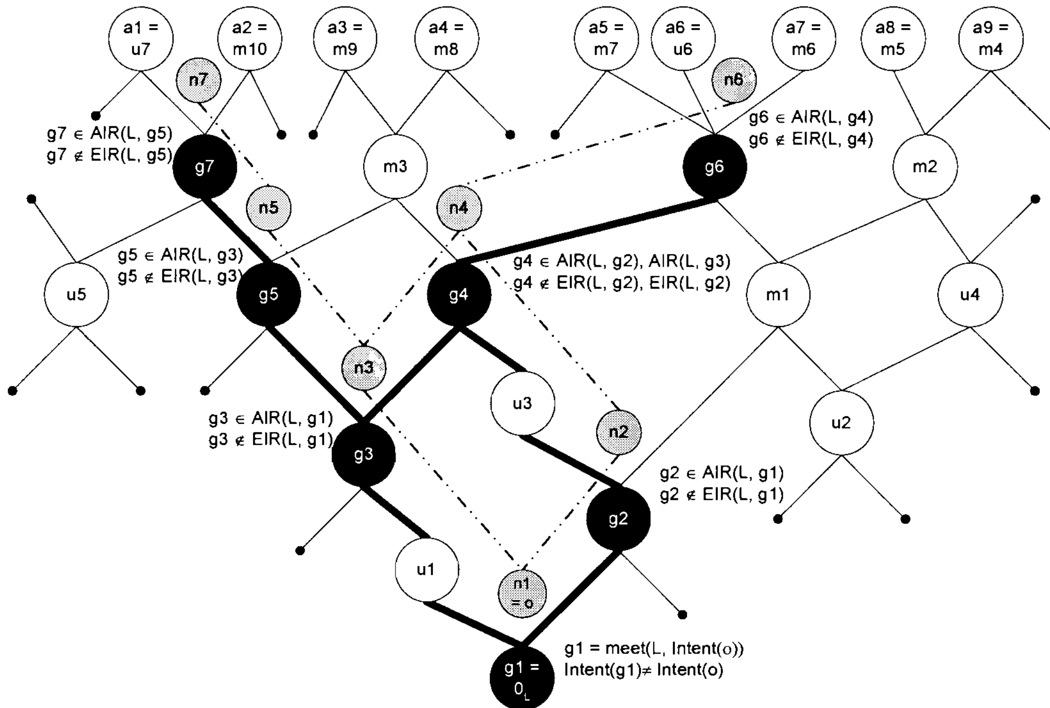


Figure 4.9: The relationship between $G(o)$, $M(o)$, $U(o)$ and $N(o)$ when inserting o with $Intent(o) = \{a_2, a_3, a_4, a_5, a_7, a_8, a_9\}$ into the lattice

Figure 4.9 shows the lattice concepts of L_i as larger circles. They comprise of $U(o)$, $M(o)$ and $G(o)$. Membership of a particular set is indicated by the prefix u, m and g in the concept labels respectively and attributes are prefixed by a. In the example, object o with $Intent(o) = \{a_2, a_3, a_4, a_5, a_7, a_8, a_9\}$ is to be inserted. The elements of $G(o)$ form a partial order, indicated by the thick arcs, with g_1 as zero concept and 1_L as unit concept. The rest of the lattice concepts are not shown and are indicated by thin arcs that do not end/start in concepts. These concepts are members of $U(o)$ and will remain unchanged. The elements of $M(o)$ are all located *above* the largest concepts of $G(o)$. The elements of $N(o)$ are superimposed on the concepts in L_i and shown as smaller, grey shaded, concepts connected by dotted arcs. Each element of $N(o)$ is shown above its respective generator concept. Note that the concepts of $N(o)$ are not yet properly connected into L_i to form L_{i+1}. As explained, $g_1 \in G(o)$ and is in fact 0_L and $n_1 \in N(o)$ is in fact the object o.

These ideas are made more explicit in the formulation of the AddAtom lattice algorithm defined in the next section.

4.3 DEFINITION OF THE ADDATOM ALGORITHM

In this section the algorithm hinted at in the two previous sections is formally defined using pseudo code. For the purpose of reference we call the algorithm AddAtom since it inserts an atom concept (i.e. an object) above the zero concept into the lattice. As defined the algorithm is conceptually simple but very inefficient. Efficient versions of the algorithm are discussed and defined later on in the text. Once again we only consider contexts that have objects that are unrelated to other objects and attributes that are completely unrelated to other attributes as explained earlier. In the corresponding lattice all the objects are thus atoms and the all attributes, coatoms.

The algorithm involves the recursive application of the ideas presented in the previous section. The algorithm is initiated by a set of attributes representing the intent of the object to be inserted (i.e. as a new atom) as well as the zero concept as the first generator concept. Each recursive AddAtom call creates aNewConcept with Intent(aNewConcept) = anAttributeSet. After each recursive call of the algorithm a new concept has been inserted into the lattice above the generator concept. This newly inserted concept has also been properly connected to its parent concepts (possibly involving further recursive AddAtom calls to create the necessary concepts). The called function returns this newly created concept and the calling function inserts this concept into the upper cover of its respective aNewConcept. Thus the recursive calls construct the additional concepts required for the insertion of the object. In this way there is no need to separately compute the covers of the newly inserted and modified concepts since the nature of the intent representative sets as traversed by the recursive calls already indicate these relationships (as depicted in the structure of $G(o)$ in figure 4.9).

Using parameter names to imply types the AddAtom algorithm is defined as follows:

```
//===================================================================
Function AddAtom (L, anAttributeSet, aGeneratorConcept)
      Return aNewConcept
//===================================================================
//Pre-condition:
// L is a partial order such that:
// 1) anAttributeSet is a set of attributes
// 2) UpwardClosure(L, aGeneratorConcept) is a complete sublattice
// 3) Meet(L, anAttributeSet) = aGeneratorConcept
// 4) aGeneratorConcept is a generator concept for
//     anAttributeset and an approximate meet of anAttributeSet
//===================================================================
//Post-condition: L is a minimally updated in such a way
// to ensure that:
// 1) UpwardClosure(L, aGeneratorConcept) remains a sublattice
// 2) Meet(L, anAttributeSet)= aNewConcept (an exact meet, AIR=EIR)
// 3) aNewConcept covers only aGeneratorConcept and nothing else
// 4) All generator concepts above aGeneratorConcept
//    have been visited and the corresponding
//    new concept has been created an appropriately
//    linked into L
//===================================================================
ApproxMeets =
   AIR(L, anAttributeSet, aGeneratorConcept) -
   EIR(L, anAttributeSet, aGeneratorConcept)
// Remove all elements of EIR from AIR
// Pre-condition 2 guarantees that the meets are unique
// Next, generate N(o)
Do While (ApproxMeets ≠ Ø)
   Select and mark any x ∈ ApproxMeets
   SubAttr = anAttributeSet ∩ x.Intent
   bNewConcept = AddAtom(L, SubAttr, x)
   Recompute ApproxMeets
   Remove all marked concepts from ApproxMeets
Od
// Post-condition 4 achieved and AIR = EIR
aNewConcept = CreateConcept(L)
aNewConcept.Extent = aGeneratorConcept
aNewConcept.Intent = anAttributeSet
// Next, connect elements of N(o) to aNewConcept
For ∀ x ∈ EIR(L, anAttributeSet, aGeneratorConcept)
   CreateArc(L, aNewConcept, x)
   //Assume no effect if arc already exists
   DeleteArc(L, aGeneratorConcept, x)
   //Assume no effect if arc does not exist
Rof
// Next update the extents of N(o) and M(o)
If aGeneratorConcept = 0_L then
   For ∀ x ∈ UpwardClosure(L, aNewConcept)
      x.Extent = x.Extent ∪ {aNewConcept}
   Rof
Fi
// Post-condition 1 & 2 achieved
CreateArc(L, aGeneratorConcept, aNewConcept)
// Post-condition 3 & 5 achieved
Return aNewConcept
End AddAtom
//===================================================================
```

Thus, to incrementally insert a new object o into a lattice for the context $\langle A, O, I\rangle$ the function call $\mathrm{AddAtom}(L, \mathrm{Intent}(o), 0_L)$ would be used. Note that L is passed as an in/out parameter. It is assumed that the individual attributes of the object o are already present in the lattice (i.e. as coatoms).

As indicated a list of marked concepts needs to be kept in order that such concepts are not revisited in the Do While…Od loop.

To operate on arbitrary contexts the AddAtom algorithm should be slightly extended to consider the following cases:

- The object is the first to be inserted into an empty L_0.
- The object to be inserted into the lattice is in fact not an atom in L_i (i.e. $\mathrm{Intent}(o)$ is a subset of some other object's intent).
- The object has same intent as another object in the context.
- The attributes of the object are not all coatoms.
- More than one attribute may correspond to a single concept in L_i (i.e. the extent of two attributes is the same).
- Some of the attributes in $\mathrm{Intent}(o)$ do not already exist in L_i.
- Some modifications are required for FCA lattices (since all objects are not atoms and all attributes not coatoms).

In addition to these the AddAtom algorithm can be modified to operate on compressed pseudo-lattices (refer to chapter 6) in that it respects the virtual arcs and compressed pseudo-lattice properties and does not assume the existence of all formal concepts in L_i.

4.4 ADDATOM EXAMPLE

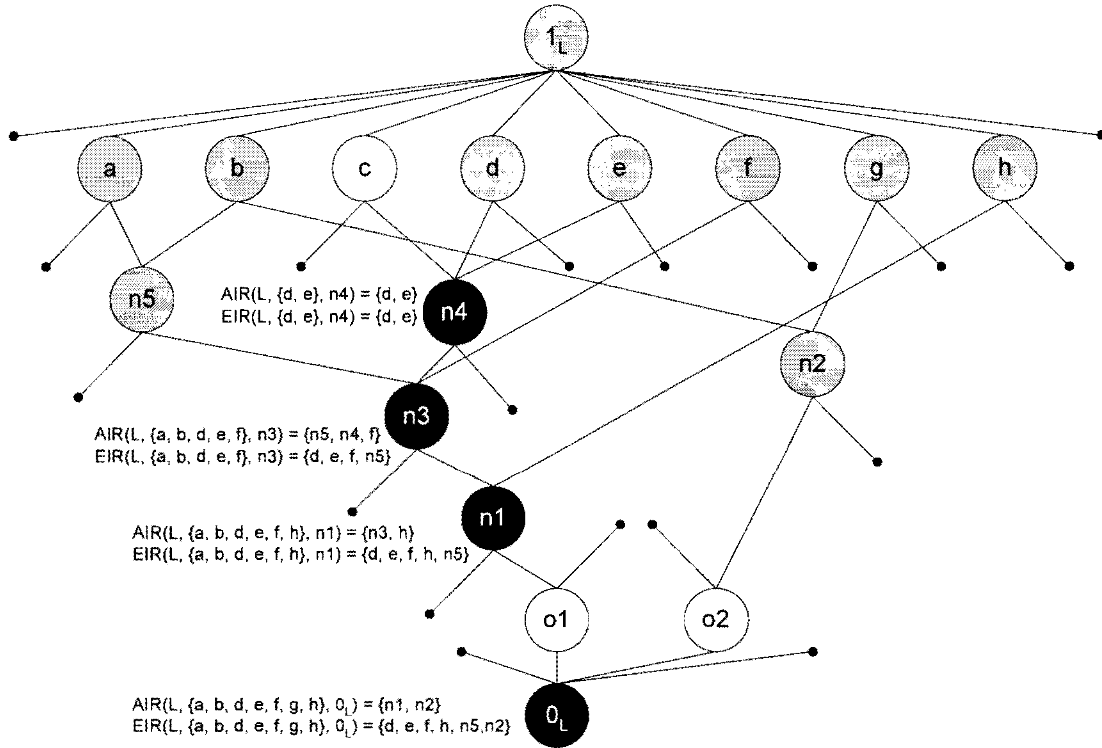


Figure 4.10: A lattice before inserting o_3 with $Intent(o_3) = \{a, b, d, e, f, g, h\}$ indicating $G(o)$ as well as the AIR and EIR sets of elements of $G(o)$

As an example consider inserting object o_3 with $\text{Intent}(o_3) = \{a, b, d, e, f, g, h)$ into the lattice, L, in figure 4.10. Since the algorithm does not consider and visit irrelevant concepts, only the relevant part of L is shown – the relationships to the rest of L are shown by arcs that do not terminate in concepts.

L is an in/out parameter in the algorithm. Thus, throughout the algorithm, operations use L as it exists at that point in the computation - not in its state when that given level of recursion was invoked with L as a parameter.

The algorithm begins with the function call AddAtom(L, $\{a, b, d, e, f, g, h\}$, 0_L). ApproxMeets has to be computed, and this requires that both AIR(L, $\{a, b, d, e, f, g, h\}$, 0_L) and EIR(L, $\{a, b, d, e, f, g, h\}$, 0_L) have to be computed. In this case $S = \{1_L, a, b, d, e, f, g, h, n_5, n_4, n_3, n_2, n_1\}$ (refer to section 2.9 for the definition of AIR and EIR). These concepts are shown in black or grey in figure 4.10. Other concepts are in white. The concepts in black are generator concepts as will become clear later. n_1 and n_2 are the two minimal concepts in S, therefore AIR(L, $\{a, b, d, e, f, g, h\}$, 0_L) = $\{n_1, n_2\}$.

In order to find EIR(L, $\{a, b, d, e, f, g, h\}$, 0_L), we see that $T = \{n_4, n_3, n_1\}$ and therefore $S - T = \{1_L, a, b, d, e, f, g, h, n_5, n_2\}$. Thus, EIR(L, $\{a, b, d, e, f, g, h\}$, 0_L), (the set of minimal concepts, excluding 0_L, in $S - T$) is $\{d, e, f, h, n_5, n_2\}$. As a result ApproxMeets (AIR – EIR) is $\{n_1\}$. n_1 is therefore a generator concept.

The first loop of the algorithm is thus executed, where $x = n_1$. Intent(L, n_1) = $\{a, b, c, d, e, f, h\}$ and SubAttr = $\{a, b, d, e, f, h\}$. Thus, AddAtom(L, $\{a, b, d, e, f, h\}$, n_1) is recursively called. Note that n_1 is an approximate meet of $\{a, b, d, e, f, g, h\}$ since it also spans the attribute c. To create an exact meet that does not span the additional attribute, c. The algorithm searches for any additional approximate meets above n_1 and creates additional concepts that will form exact meets.

In tracing the function call AddAtom(L, {a, b, d, e, f, h}, n_1) we see that AIR(L, {a, b, d, e, f, h}, n_1) = {n_3, h} and EIR(L, {a, b, d, e, f, g, h}, n_1) = {d, e, f, h, n_5, n_2} so that ApproxMeets = {n_3}. SubAttr = {a, b, d, e, f} with n_3 being an approximate meet of SubAttr, again spanning c in addition. n_3 is thus a generator concept. AddAtom(L, {a, b, d, e, f}, n_3) is therefore recursively called to create an exact meet above n_3.

AddAtom(L, {a, b, d, e, f}, n_3) calculates AIR(L, {a, b, d, e, f}, n_3) = {f, n_5, n_4} and EIR(L, {a, b, d, e, f}, n_3) = {d, e, f, n_5}, so that ApproxMeets = {n_4}.

Once again n_4 is a generator node and AddAtom(L, {d, e}, n_4) is called recursively. Since AIR(L, {d, e}, n_4) = EIR(L, {d, e}, n_4) = {d, e}, ApproxMeets = ∅ and the algorithm progress past the while loop to create n_6 whose intent is to become {d, e} (figure 4.11). Moving to the next loop of AddAtom EIR(L, {d, e}, n_4) = {d, e} and therefore arcs are created between n_6 and d and n_6 and e. n_4 disconnected from both d and e. Finally after completion of the for loop an arc is created between n_4 and n_6 and AddAtom(L, {d, e}, n_4) terminates with n_6 as the result which is passed back to AddAtom(L, {a, b, d, e, f}, n_3).

AddAtom(L, {a, b, d, e, f}, n_3) now creates n_7 and calculates EIR(L, {a, b, d, e, f}, n_3) = {n_6, n_5, f} (n_6 being the newly created exact meet). It then creates arcs from n_7 to n_5, n_6 and f. The arcs from n_3 to n_5 and f are deleted. An arc is created between n_3 and n_7 and the function returns n_7 as the result.

AddAtom(L, {a, b, d, e, f, g, h}, n_1) creates n_8 and since EIR(L, {a, b, d, e, f, g, h}, n_1) = {h, n_7} arcs from each to n_8 are created. The arc between n_1 and h is deleted. An arc between n_1 and n_8 is created and AddAtom(L, {a, b, d, e, f, g, h}, n_1) terminates with n_8 as result.

Finally AddAtom(L, {a, b, d, e, f, g, h}, 0_L) creates o_3 and create arcs between o_3 and n_2 and n_8. Since o_3 is a newly inserted object it is added to the extent of all the concepts above it. 0_L is connected to o_3. This concludes the recursive AddAtom calls and AddAtom returns the inserted object o_3 to the calling function. Since L was an in/out parameter, it now refers to the newly created lattice.

The resulting EA-lattice is shown in figure 4.11 with the newly created concepts shown in grey and their corresponding generator concepts in black. The AddAtom function calls are also shown next to the respective generator concepts.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

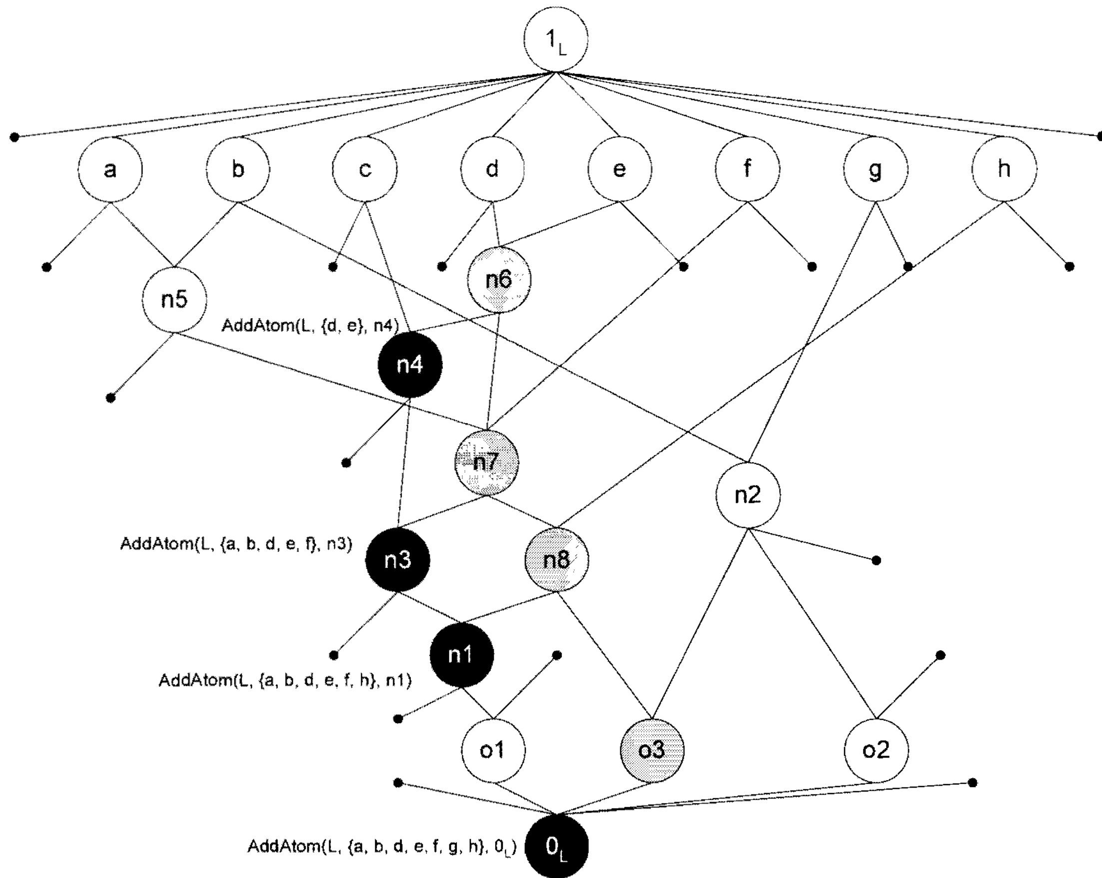


Figure 4.11: The AddAtom example after inserting o_3 with $Intent(o_3) = \{a, b, d, e, f, g, h\}$, $G(o)$ and $N(o)$ as well as the recursive AddAtom calls are indicated

AddAtom thus starts at the bottom of the lattice at the zero concept and traverses the lattice upward, creating new concepts associated with 'approximate' meets. The new concepts form exact meets of the intent of the object. The recursion terminates when AddAtom encounters only 'exact' meets (i.e. elements of $M(o)$) to which the newly created concepts are connected. In this way the recursive calls efficiently search the lattice for generator concepts and, whilst doing so, use the inherent structure of L_0 to search for, create and connect the concepts of L_1.

The example also shows how the structure and ordering of concepts in L_0 can be used to efficiently eliminate many concepts in the lattice from consideration by using the AIR and EIR operations. Some incremental lattice construction algorithms resort, in a sense, to a more brute force approach in considering a much larger set of concepts in order to test for generation concepts or in order to intersect the intent of the object with these concepts.

4.5 AN ALGORITHM FOR AIR AND EIR

It might be argued that the AddAtom algorithm is merely a restatement of an incremental lattice construction algorithm in terms of AIR and EIR but that the calculation of AIR and EIR is computationally inefficient. This research indicates that there are indeed efficient algorithms for calculating AIR and EIR but these rely on the explicit representation of the line diagram or cover relationship as a data structure.

One way of efficiently calculating AIR and EIR is to use the concept of marker propagation in which so-called "markers" are propagated downward along *all* paths leading from each

of the attributes of the object o. Afterwards the number of markers that have accumulated on each of the concepts is counted. The number of markers thus indicates how many attributes of o a concept has in its intent. Concepts with zero markers therefore need not be considered as candidates for being minimal meets in AIR or EIR. Concepts with a higher number of markers are lower down in the lattice than those with a lower number of markers. Furthermore, there will be many concepts that have the same number of markers. The number of markers increases as one moves down in the lattice.

There are three key observations to finding AIR (and EIR) using marker propagation. The first key observation is that any concept that has somewhere below it in the lattice another concept with more markers than itself is not a candidate for AIR, since it can not be minimal. The second key observation is that a concept is only a candidate if it does not have a parent concept which has the same number of markers as itself (i.e. if it is the highest concept with that number of markers and has no parent with the same number of markers). This is because if any concept has a parent concept above it with the same number of markers, it cannot be a greatest (i.e. highest) lower bound of a subset of Intent(o). The third observation is that when searching for candidate concepts by starting with those with the highest number of markers and eliminating all concepts above and below them from consideration, all candidate concepts will be found.

Using markers one thus has to search for all concepts that have the largest number of markers accumulated upon them but that have no concept below them with more markers. All such concepts are candidate concepts, but only those that have no concept above them with the same number of markers are elements of AIR.

Figure 4.12 is part of a lattice before inserting object o into it. Suppose Q is the set of attributes associated with o and markers are propagated down from each attribute. The concepts are labelled by the number of markers accumulated on them (i.e. the number of attributes of Q it spans). Arcs to the rest of the lattice are shown as lines ending in small circles without concept numbers. Those arcs ending in filled/solid circles indicate arcs to attributes in Q and those to unfilled circles indicate arcs to unique attributes not in Q. The marker count is therefore the number of filled small circles above each concept.

To search for AIR(Q) the set of concepts with the highest number of markers (5 markers) is considered. In this case the set is $\{n_{21}, n_{24}, n_{25}, n_{26}\}$. n_{25} and n_{26} have a concept above them with the same number of markers so they can be discarded from the set, leaving $\{n_{21}, n_{24}\}$. Next we eliminate all the concepts in n_{21} and n_{24}'s upward and downward closure from consideration and continue searching for concepts with the highest number of markers. In the remaining concepts, n_{27} has the highest number of markers with 4. After eliminating its upward and downward closures from consideration the only concepts with more than zero markers that remain are n_{12}, n_{17}, n_{18} and n_{23} with three markers each. Since n_{17}, n_{18} and n_{23} have concepts above them with the same number of markers, n_{12} is the last remaining element of AIR(Q). Therefore AIR(Q) = $\{n_{12}, n_{21}, n_{24}, n_{27}\}$. These concepts are shown in black. They are all generator concepts of o but are not the only generator concepts of o (the other generator concepts are n_4, n_6, n_7, n_8, n_{10}, n_{14} and n_{19}).

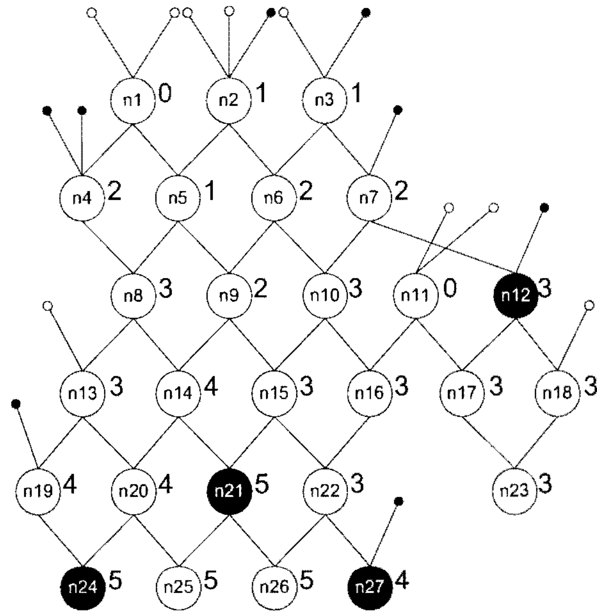


Figure 4.12: Part of a lattice before inserting object o into it showing $AIR(o)$ in black. Each concept (n_1 to n_{27}) is labeled with the number of markers / attributes of o that has accumulated on it.

This process formalised in the following algorithm:

```
//=================================================================
Function AIR(L, anAttributeSet) Return aConceptSet
//=================================================================
//Pre-condition:
// L is a concept lattice with anAttributeSet a non-empty subset of
// L's attributes
//=================================================================
//Post-condition:
// aConceptSet contains the minimal (possibly approximate) meets
// of anAttributeSet or AIR(L, anAttributeSet)
//=================================================================
NotVisited = ∅
MaxAttr = 0
Let attrCount[c] = 0 for all c ∈ L
For ∀ a ∈ anAttributeSet
   For ∀ b ∈ DownwardClosure(L, a)
      attrCount[b] = attrCount[b] + 1
      NotVisited = NotVisited ∪ {b}
      If attrCount [b] > MaxAttr then
         MaxAttr = attrCount[b]
      Fi
   Rof
Rof
Candidates = ∅
Do While (NotVisited ≠ ∅ and MaxAttr > 0)
   Let d be any c ∈ NotVisited with attrCount[d] = MaxAttr
   If such a c does not exist then
      MaxAttr = MaxAttr - 1
   Else
      // If d has concepts above it with the same number of markers
      // find the one that is the greatest
      Found = False
      Do While Not Found
         Found = True
         For ∀ p ∈ Parents(d)
            If attrCount[p] = attrCount[d] then
               d = p
               Found = False
               Exit For
            Fi
         Rof
      Od
      Candidates = Candidates ∪ {d}
      // Remove the upward closure of d from further
      // consideration - its elements can not be minimal meets
      UCD = UpwardClosure(L, d)
      NotVisited = NotVisited - UCD
      // Remove any candidates that are greater
      // than d - they can not be minimal
      Candidates = Candidates - UCD
      // Remove all concepts below d since they have MaxAttr
      // markers or have been considered
      DCD = DownwardClosure(L, d)
      NotVisited = NotVisited - DCD
   Fi
Od
Return Candidates
End AIR
//=================================================================
```

The calculation of EIR can be done in a similar way but only concepts that are exact must be considered as candidates. This process can be fast-tracked by eliminating the union of the downward closure of all attributes not in Intent(o) from consideration before propagating the markers. The calculation of AER and EER can be done using the same strategy, but this time propagating markers in the opposite direction and appropriately changing the direction of the relevant operators in the algorithm.

The algorithms of the intent- and extent operations were defined in terms of the closure and set operations. When representing sets as strings of bits in memory, these operations can be very efficiently performed on modern architectures using 32 or 64 bit words. The calculation of AIR and EIR is therefore very efficient.

Since the intents and extents of the concepts in the lattice can be derived from the upward- and downward closures of the concepts in the line diagram, these need not be calculated explicitly.

It is also possible to have attrCount pre-computed when the AIR etc. will be computed for a subset of A. This optimisation is considered in the efficient AddAtom algorithm defined in the next section.

4.6 EFFICIENT ADDATOM ALGORITHM

The AddAtom algorithm as described in section 4.3 is not optimal in terms of efficiency. A number of basic performance improvements can be made on the algorithm. Examples include the possible avoidance of recalculation of ApproxMeet and the processing of the generator concepts in the order of the size of their intent. The calculation of both the exact and approximate intent representative sets can also be computationally inefficient and may duplicate many operations due to the similarity between the two sets. The following algorithm is an efficient version of the AddAtom algorithm of section 4.3. It builds on the ideas of the calculation of AIR and avoids the repeated and calculations of AIR and EIR.

```
//=====================================================================
Function OptimisedAddAtom(aContext) Return aLattice
//=====================================================================
L = CreateEmptyLattice()
1L = NewConcept(L)
0L = NewConcept(L)
0L.Intent = aContext.Attr
For ∀ a ∈ aContext.Attr
   anAttributeConcept = NewConcept(L)
   anAttributeConcept.Intent = {a}
   CreateArc(L, 0L, anAttributeConcept)
   CreateArc(L, anAttributeConcept, 1L)
Rof
For ∀ o ∈ aContext.Obj
   // Calculate attrCount[x], the number of attributes in o.Intent
   // that occur in x.Intent
   Let attrCount[x] = 0 for all x ∈ L
   For ∀ x ∈ L
      attrCount[x] = ||x.Intent ∩ o.Intent||
   Rof
   NewObject = AddAtom(L, o.Intent, 0L, attrCount)
   For ∀ x ∈ UpdwardClosure(NewObject)
      x.Extent = x.Extent ∪ {o}
   Rof
Rof
Return L
End OptimisedAddAtom

//=====================================================================
Function GetMeet(L, target, aConcept, attrCount)
                                 Return returnConcept
//=====================================================================
//Pre-condition:
// L is a concept lattice, attrCount[aConcept] = target
//=====================================================================
//Post-condition:
// returnConcept is the greatest upper bound/concept in L with
// attrCount[returnConcept] = target
//=====================================================================
returnConcept = aConcept
ParentIsMeet = True
Do While ParentIsMeet
   ParentIsMeet = False
   For ∀ Parent ∈ ConceptParents(L, aConcept)
      If attrCount[Parent] = target then
         returnConcept = Parent
         ParentIsMeet = True
         Exit For
      Fi
   Rof
Od
Return returnConcept
End GetMeet

//=====================================================================
Function AddAtom(L, anIntent, GeneratorConcept, attrCount)
                         Return aConcept
//=====================================================================
//Pre-condition:
// 1) UpwardClosure(L, GeneratorConcept) is a complete sublattice
```

```
// 2) GeneratorConcept is the meet of anIntent and is approximate
// 3) attrCount[c] = Intent(c)∩Intent(newObject)
//=====================================================================
//Post-condition:
// returnConcept is the greatest upper bound/concept in L with
// attrCount[returnConcept] = target
//=====================================================================
CandidateParents = ConceptParents(L, GeneratorConcept)
NewConceptParents = ∅
For ∀ Candidate ∈ CandidateParents
   newIntent = Candidate.Intent ∩ anIntent
   If newIntent ≠ ∅
      If Candidate.Intent ≠ newIntent then
         aMeet = GetMeet(L, ||newIntent||, Candidate, attrCount)
         If aMeet.Intent ≠ newIntent
            // If aMeet is approximate it is a generator concept and an
            // exact meet needs to be created
            aMeet = AddAtom(L, newIntent, aMeet, attrCount)
         Fi
      Else
         aMeet = Candidate
      Fi
      addMeet = True
      For ∀ g ∈ NewConceptParents
         If aMeet.Intent ⊆ g.Intent
            addMeet = False
            Exit For
         Else If g.Intent ⊂ aMeet.Intent then
            NewConceptParents = NewConceptParents - {g}
         Fi
      Rof
      If addMeet then
         NewConceptParents = NewConceptParents ∪ {aMeet}
      Fi
   Fi
Rof
NewConcept = CreateNewConcept(L)
NewConcept.Extent = GeneratorConcept.Extent
NewConcept.Intent = anIntent
attrCount[NewConcept] = attrCount[GeneratorConcept]
For ∀ g ∈ NewConceptParents
   DeleteArc(L, GeneratorConcept, g)
   CreateArc(L, NewConcept, g)
Rof
CreateArc(L, GeneratorConcept, NewConcept)
Return NewConcept
End AddAtom
//=====================================================================
```

Some optimisations are still possible, but these do not change the basic structure of the algorithm as stated above. Appendix A contains the pseudo code for one such optimised version of AddAtom that amongst other strategies considers concept parents in descending order of their attrCount value. This allows for the removal of many additional concepts from consideration.

4.7 DISCUSSION

Initially some of the meets of subsets of Intent(o) are approximate meets (i.e. generator concepts). After each completion of a recursive call, additional concepts have been created that would now form the exact meets of those subsets of Intent(o) and replace the approximate meets. The algorithm terminates when all meets of all subsets of Intent(o) are exact with regards to Intent(o). Initially, L is a lattice but as new concepts are generated that are not yet fully integrated to the lattice structure, some parts of L may violate the lattice properties up until the completion of all levels of the recursion. When terminating, the AddAtom algorithm ensures that all concepts in UpwardClosure(L, aGeneratorConcept) form a lattice. Since the first AddAtom call uses 0_L as the generator concept, L will be a lattice when that AddAtom call terminates.

The AddAtom algorithm generates the new concepts and cover relationships in one step and therefore seems to be more focussed than incremental lattice construction algorithms that first generate the concepts and then search and generate the upper covers of concepts using a separate function such as Godin et al. (1991) and Carpineto and Romano (1993, 1996b). Experiments to date (discussed in chapter 5) also suggest AddAtom is more efficient.

The algorithm exploits the relationships between concepts already represented in the lattice to efficiently search for the generator concepts using the intent representative operations. To this extent the algorithm makes explicit use of the line diagram that represents the original lattice structure when searching for G(o) by means of the ordering relationship and the intent representative operations rather than considering all concepts at once in a more brute force search. Indeed, the intent representative operations themselves imply a ordering of the generator and new concepts in L_1.

A very important property of the algorithm is that it can operate on sublattices where the formal concept lattice of a context is not used as input. This is due to the fact that the algorithm is entirely general in not requiring the lattice to have a specific set of atoms or coatoms (i.e. those representing the attributes and objects) but not necessarily that of the formal concept lattice or EA-lattice (similar to those concept sublattices created in compressed pseudo-lattices). Such lattices are not closed with respect to the intersection of intents or the union of extents. The only requirement is that anAttributeSet consists only of coatoms (and not necessarily attribute concepts of the context). The operations used are therefore based on closure operations rather than intersections of intents. Such lattices are for example under consideration in compressed pseudo-lattices where the lattice is not closed with regards to the intersection of intents. Not all lattice construction algorithms are suitable for applications using sub-lattices in this kind of way.

An optimised, object-oriented version of the algorithm was implemented and tested in C++ (chapter 7). In addition, the implementation also implements the concept of a compressed pseudo-lattice (chapter 6). The algorithm therefore takes the existence of virtual- and lattice arcs into consideration during its operation.

Since the direction of the operations can be reversed (e.g. meet replaced by join, EIR by EER, atom by coatom, etc.) a dual for the AddAtom algorithm namely AddCoatom can be defined. In the implementation this was achieved by adding an additional parameter named aDirection to all lattice operations to indicate the direction in which the operation should operate.

The next chapter (chapter 5) analyses the algorithmic performance of the algorithm by comparing the performance of AddAtom to that of other lattice construction algorithms both theoretically and experimentally.

Chapter 5: AddAtom algorithmic performance

The AddAtom algorithmic performance was studied both theoretically (worst case behaviour) and empirically. This chapter starts with a short survey of published concept lattice construction algorithms (section 5.1) before deriving a theoretical upper bound to the complexity of AddAtom (section 5.2). A theoretical worst-case performance bound of AddAtom is $O(||L||.||O||^2.\max(||O'||))$. This performance upper bound is however a higher of magnitude as the current best performer namely that of Nourine and Raynaud (1999, 2002) with an upper bound of $O((||O||+||A||).||O||.||L||)$. Despite being cubic in nature relative to the lattice size, it is argued that this bound of AddAtom is not a very sharp upper bound and that the terms in the complexity expression are in practice, much more of an overestimate than the $||O||$ and $||A||$ terms that appear in the upper bound estimates of other construction algorithms. The performance is thus best confirmed via experimental comparisons.

For the purposes of experimental comparison, a two-step approach was taken. Firstly, a pilot study comparing the original AddAtom implementation in C++ (described in chapter 7) to implementations of two published construction algorithms, namely that of Godin (1991) and Carpineto and Romano (1993, 1996b). The pilot comparison showed that there is *prima facie* evidence that the algorithm performs very well and that wider study is justified. The second study, involving a wider set of experimental comparisons across a larger number of lattice construction algorithms, was conducted in collaboration with another researcher. For the sake of reference, the first, smaller experimental comparison will be referred to as the "pilot study" (section 5.3) and the second as the "wide performance study" (section 5.4).

The results of both experimental comparisons indicate that the algorithmic performance of AddAtom is very good, and often the best of the test bed of 11 concept lattice construction algorithms. AddAtom performs especially well compared to other algorithms with "natural" data sets (i.e. non-random generated context). When the density of the cross table of the context is either very high (i.e. every object possesses almost all attributes) or very low (every object possesses only very few attributes) there are other concept lattice construction algorithms that do outperform AddAtom. AddAtom is however still the next-to-best performer in these circumstances and therefore a worthy candidate for a general-use algorithm. AddAtom was the fastest incremental lattice construction algorithm in the study. The experimental comparison results are therefore consistent with the argument that the theoretical complexity bound of AddAtom derived here is not a very sharp upper bound for AddAtom.

Note that the discussion of both the theoretical and empirical performance, is with reference to the optimised version of the AddAtom algorithm (see section 4.6). In the comparisons, performance issues are related to constructing both the set of all concepts as well as the cover relationships (i.e. the line diagram).

5.1 A SURVEY OF CONCEPT LATTICE CONSTRUCTION ALGORITHMS

It is not the objective of this dissertation to analyse and describe other construction algorithms. Readers are referred to recent comparative studies by Kuznetsov and Obiedkov (2001, 2002) for a broad discussion and pseudo code of other algorithms. A number of optimisations of these algorithms as well as adaptations to generate the line diagram of concept lattices where the algorithm does not generate it already are also described by Kuznetsov and Obiedkov. Unless otherwise stated, references to the complexity or experimental performance of these algorithms refer to the improvements and adaptations propose by Kuznetsov and Obiedkov. Although not exhaustive, the following table lists a number of the published concept lattice construction algorithms and briefly describes all the algorithms referred to in this chapter (adapted from Kuznetsov and Obiedkov (2001, 2002)). The theoretical and experimental comparisons will be made to a subset of these algorithms.

In each case an algorithm is classified as either incremental or batch (non-incremental) and also whether it generates only the set of all concepts or the line diagram of the lattice.

Algorithm	Incremental / Batch	Notes
Chein	Batch	Chein (1969) Concepts are represented as extent-intent pairs and each new concept is generated as the intersection of the intents of two existent concepts. Similar to AI-tree. Modifications were suggested by Kuznetsov and Obiedkov (2002). Generates the set of all concepts of the lattice.
Ganter, NextClosure	Batch	Ganter (1984) Batch algorithm adding one object to earlier generated extent and calculating closure. Generate concepts in topological order using lexical order for concept lookup and comparison. Modifications were suggested by Kuznetsov and Obiedkov (2002). Generates the set of all concepts or the line diagram of the lattice.
Bordat	Batch	Bordat (1986) Batch algorithm intersecting the intent of concepts with intents of objects that don't belong to concept. Generate concepts in depth-first order using a tree for concept lookup and comparison. Modifications were suggested by Kuznetsov and Obiedkov (2002). Generates the set of all concepts or the line diagram of the lattice.
AI-tree	Batch	Zabezhailo et al. (1987) A top-down batch algorithm that searches for concepts in the set of concepts generated thus far. Similar to Chein. Generates the set of all concepts of the lattice.

Algorithm	Incremental / Batch	Notes
CbO, Close by One	Batch	Kuznetsov (1993) Batch algorithm (similar to NextClosure) adding one object to earlier generated extent and then calculating the closure. Generate concepts in depth first order using lexical order for concept lookup and comparison. Also use an intermediate structure for concept searches and the generation of the line diagram. Generates the set of all concepts or the line diagram of the lattice.
Lindig	Batch	Lindig (1999, 2000) Bottom-up batch algorithm adding one attribute at a time to the intent of generated concepts and then calculating its closure. Generate concepts in a depth-first order using tree for concept searches. Generates the diagram of the lattice.
Titanic	Batch	Stumme et al. (2000)
Yevtushenko	Batch	Yevtushenko (2002)
Norris	Incremental	Norris (1978) Incremental algorithm intersecting the new object intent with that of concepts generated earlier. Keep list of added objects, checking whether new concepts can be generated using intersection of objects added earlier. This has been described as being an incremental version of the CbO algorithm. Modifications were suggested by Kuznetsov and Obiedkov (2002). Generates the set of all concepts or the line diagram of the lattice.
Godin, GodinEx	Incremental	Godin et al. (1991, 1995b) Incremental algorithm intersecting new object intent with that of concepts generated earlier. Use a heuristic hash function to sort the concepts when generating and searching concepts. There are two versions of the algorithm: GodinEx refers to the version that uses the size of the extents and Godin the size of the intents. Modifications were suggested by Kuznetsov and Obiedkov (2002). Generates the set of all concepts or the line diagram of the lattice.

Algorithm	Incremental / Batch	Notes
Grand	Incremental	Oosthuizen (1991) Incremental algorithm using a graph theoretic approach to insert an object into a lattice. Grand connects objects attribute by attribute, to an increasing subset of its intent until the object is connected to all the attributes in its intent. During the process a function called transform ensures that the uniqueness of suprema and infima is maintained through the manipulation, addition and deletion of concepts and arcs. Constructs EA-lattices. Generates the line diagram of the lattice.
Carpineto	Incremental	Carpineto and Romano (1993, 1996b)
Nourine	Incremental	Nourine and Raynaud (1999, 2002) Incremental algorithm intersecting new object intent with that of concepts generated earlier. Use a lexical tree for concept lookup and comparison. Generates the line diagram of the lattice.
Valtchev, Divide and conquer	N/A	Valtchev et al. (2000) Algorithm based on the combination of two concept sub-lattices that are combined to construct the full lattice. The context of each sub-lattice is obtained by splitting the cross table of the original context either by objects or by attributes. Generates the line diagram of the lattice.
AddAtom	Incremental	Described in chapter 4. Incremental algorithm using the approximate and exact intent representative (minimal meets) of the object intent to find generator concepts and recursively generate new concepts above these. AddAtom use the lattice itself for concept lookup, comparison and avoiding duplicate generation of concepts. Generates the line diagram of the lattice.

5.2 A THEORETICAL PERFORMANCE BOUND FOR ADDATOM

In this section the notation used for the description of the theoretical complexity (section 5.2.1) and a number of lattice size related formulae are given (section 5.2.2 and 5.2.3). This will be used to derive an upper bound for the theoretical complexity of AddAtom (5.2.4)

5.2.1 Notation

The following notation is used for the theoretical performance of constructing a formal concept lattice of the context $C = \langle O, A, I \rangle$:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Notation	Description
$\|O\|$	The number of objects in the context.
$\|A\|$	The number of distinct attributes in the data set or context itself, not the theoretical limit of the domain from which the context was taken. As the number of objects increase, $\|A\|$ typically approaches the theoretical limit (e.g. in the case of randomly generated contexts).
$\|I\|$	The number of “crosses” in the cross table of the context. It is therefore the number of attribute-object pairs in the incidence relation. The maximum number of crosses in the cross table is $\|O\|.\|A\|$.
$\|O'\|$	The average number of attributes per object in the context, i.e. the average intent size of atoms in the EA-lattice of the context. $\|O'\| = \|I\| / \|O\|$. For contexts with a varying number of attributes per object, the maximum number of attributes per object the notation $\max(\|O'\|)$ is used to indicate the maximum number of objects per attribute.
$\|A'\|$	The average number of objects per attribute in the context, i.e. the average extent size of co-atoms in the EA-lattice of the context. $\|A'\| = \|I\| / \|A\|$. For contexts with a varying number the maximum number of attributes per object the notation $\max(\|A'\|)$ is used to indicate the maximum number of objects per attribute.
$\|L\|$	The number of concepts in the lattice of the context including the unit and zero concepts. L_j indicates the lattice after the insertion of the j’th object into the lattice.
$\|<\|$	The number of arcs in the line diagram of the lattice L. $\|<_j\|$ indicates the number of arcs in lattice L_j.
$\|O'\| / \|A\|$	This is referred to as the “density” of the cross table and is defined as the proportion of crosses in the cross table relative to the total number of possible crosses in the cross table (i.e. $\|I\| / (\|O\|.\|A\|) = \|O'\| / \|A\| = \|A'\| / \|O\|$). It can, of course, be specified as a percentage and is useful as a normalised metric to compare contexts with a different number of attributes.

5.2.2 Concept lattice size formulae

In this section a number of formulae and equations on concept lattice aspects related to size are derived. These will be used in deriving complexity bounds for AddAtom. Before deriving the actual lattice formulae, a number of generic equivalences are given. These equivalences will be used to refine the lattice formulae.

The following two generic equivalences, found in many texts on algebra and combinatorics (e.g. Cameron (1996)), can be proved by induction. The formulas are derived from the Binominal theorem.

$$\sum_{k=0 \text{ to } n} \binom{n}{k} = 2^n \tag{5.1}$$

$$\sum_{k=0 \text{ to } n} k \binom{n}{k} = n.2^{n-1} \tag{5.2}$$

The next two equivalences, can be found in texts on mathematical analysis (e.g. Clark(1931)), and are also proved via induction.

$$\sum_{k=0 \text{ to } n} a^k = \frac{1-a^{n+1}}{1-a} \; ; a \neq 1 \tag{5.3}$$

$$\sum_{k=1 \text{ to } n} k.a^k = \frac{a}{(1-a)^2}\left(n.a^{n+1} - (n+1)a^n + 1\right); a \neq 1 \tag{5.4}$$

Figures 5.1 and 5.2 show the Boolean lattices of contexts with 3 and 4 attributes respectively. The discussion below refers to Boolean lattices L_j from a context $C = \langle O, A, I \rangle$. These figures are included here to serve as an aid in explaining the derivation of the lattice size equations 5.5 to 5.8.

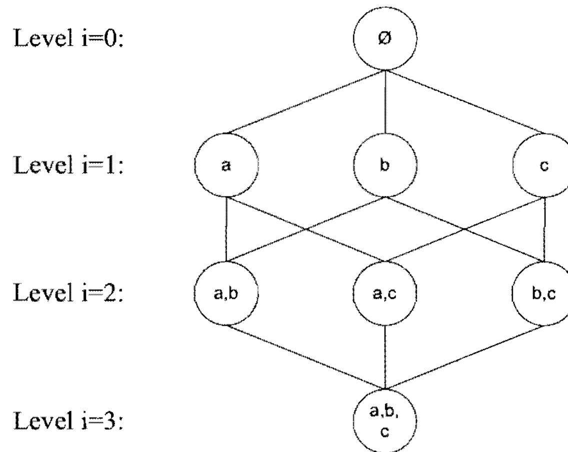


Figure 5.1: A Boolean lattice with 3 attributes (only concept intents are shown; the level of the concept is also shown)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

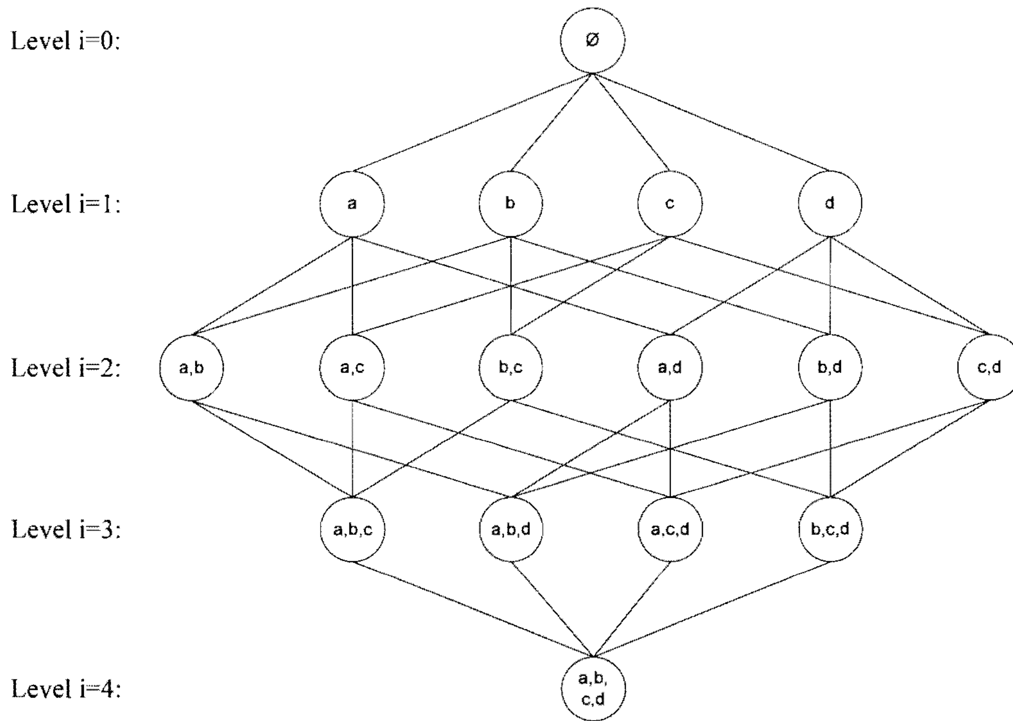


Figure 5.2: A Boolean lattice with 4 attributes (only concept intents are shown; the level of the concept is also shown)

For the purpose of discussion, the concepts in the Boolean lattice will be divided into a number of levels, where the number of attributes in the intent of the concept indicates its level. The variable i will indicate the level. Where multiple, successive lattices are under discussion, the variable j will indicate the j'th lattice in the sequence of lattices (i.e. after the insertion of the j'th object). The equations in the table below characterise important aspects of Boolean lattices related to size. Note that for theoretical purposes an initial lattice consisting only of a single concept with an empty intent and -extent called L_0 with $\|L_0\| = 1$ and $\|\prec\| = 0$ is included in the equations. For the sake of simplicity only $\|A\|$ is used since $\|O\| = \|A\|$ for Boolean lattices. The remarks indicate how these equations have been derived. These equations hold for Boolean lattices. It is assumed that $\|O\| > 0$ and $\|A\| > 0$.

Equation	Remark	Nr
$\|L\| = \sum_{i=0 \text{ to } \|A\|} \binom{\|A\|}{i} = 2^{\|A\|}$	The total number of concepts on each level i of a Boolean lattice is the number of distinct combinations of subsets of A of size i. The final result follows from equation 5.1.	(5.5)
$\|\prec\| = \sum_{i=0 \text{ to } \|A\|} i \binom{\|A\|}{i} = \|A\| \, 2^{\|A\|-1} = \frac{1}{2} . \|A\| . \|L\|$	Inspecting figures 5.1 and 5.2 it can be seen that each concept on any level i, has i arcs leading to it's i parents. Once again the number of concepts on level i is the number of distinct subsets of A of size i. After counting the	(5.6)

total number of arcs, equation 5.2 is used to simplify the result.

$$\sum_{j=0 \text{ to } \|A\|} \| L_j \| = \sum_{j=0 \text{ to } \|A\|} 2^j = 2^{\|A\|+1} - 1 = 2.\| L \| - 1$$

The total number of concepts in all lattices L_j, $j = 0$ to $\|A\|$ follows from combining equation 5.5 and 5.3. (5.7)

$$\sum_{j=0 \text{ to } \|A\|} \|\prec_j\| = \sum_{j=0 \text{ to } \|A\|} j.2^{j-1} = \sum_{j=0 \text{ to } \|A\|-1} j.2^j + \sum_{j=0 \text{ to } \|A\|-1} 2^j = (\| A \| -1).2^{\|A\|} + 1 = (\| A \| -1).\| L \| + 1$$

The total number of arcs in all lattices L_j, $j = 0$ to $\|A\|$ can be derived by combining equations 5.6, 5.3 and 5.4. (5.8)

Non-Boolean concept lattices

Boolean FCA lattices contain the maximum number of possible concepts (i.e. unique combinations of intent and extent) for a given number of arcs and therefore contexts that do not give rise to Boolean lattices have fewer concepts in their lattices. The size of arbitrary lattices is therefore bound by the minimum of the unique number of extents or intents possible, i.e. $2^{\min(\|A\|, \|O\|)}$.

The number of outbound arcs is bound by the unique combinations of attributes in the intents of its parents and/or the unique combinations of objects in the extents of its parents. In a Boolean lattice, the number of possible unique intents of the parents of a concept c is $\|\text{Intent}(c)\|$ - 1, but non-Boolean lattices may potentially have more (up to $\binom{\text{Intent}(c)}{\text{Intent}(c)\div 2}$). (Using the extents of the parent concepts provides a sharper bound to the number of outward arcs. The parent concepts $p_1 \ldots p_n$, of a concept c must be unique and therefore have at least one object in their extents in addition to that of c, i.e. $\text{Extent}(p_i) \supset \text{Extent}(c)$. Furthermore, for any two parent concepts, p_i and p_j, $p_i \neq p_j$ of c, $\text{Extent}(p_i) \cap \text{Extent}(p_j) = \text{Extent}(c)$. Parent concepts can therefore have no object in common with the extent of any other parent concept except that of c. The extent of any concept must also be unique in the lattice. Given these constraints, the maximum number of parent concepts of any c is therefore $\|O\|$ - $\|\text{Extent}(c)\|$ since each parent concept will have one at least additional concept of O in addition to $\text{Extent}(c)$. A bound for the maximum number of outbound (upward) arcs of any concept in a lattice is therefore $\|O\|$. In practice however the maximum number of outbound arcs may be fewer.

Using a similar argument, but based on the intent of any concept $\|A\|$ is the maximum number of inbound (downward) arcs into any concept in a lattice.

Using these bounds as a base it is clear that for non-Boolean lattices of any substantial size the number of outbound arcs $\|\prec_{\text{Outbound}}\| \leq \|O\|.\|L\|$. Using a similar argument $\|\prec_{\text{Inbound}}\| \leq \|A\|.\|L\|$. Since the number of outbound- and inbound arcs in any lattice should be equal to the total number of arcs, $\|\prec\| \leq \min(\|A\|, \|O\|).\|L\|$.

Most contexts used in practical applications have $\|A\| < \|O\|$. It is assumed that $\|O\| > 0$ and $\|A\| > 0$. The following inequalities hold in such cases:

Equation	Nr

$$\|L\| \leq 2^{\min(\|A\|,\|O\|)} \leq 2^{\|A\|} \tag{5.9}$$

$$\|\prec\| \leq \min(\|A\|, \|O\|).\|L\| \tag{5.10}$$

$$\sum_{j=0 \text{ to } \|O\|} \|L_j\| \leq \|O\|.\|L\|+1 \tag{5.11}$$

$$\sum_{j=0 \text{ to } \|O\|} \|\prec_j\| \leq \min(\|A\|,\|O\|).\|O\|.\|L\|+1 \leq \|A\|.\|O\|.\|L\|+1 \tag{5.12}$$

From equation 5.6 it can be seen that a Boolean lattice contains, on average ½.||A|| outbound arcs per concept and also on average ½.||A|| inbound arcs per concept since the total number of outbound- and inbound arcs in a lattice are equal. It is therefore clear that the above equations do not always provide very sharp upper bounds. Where ||L|| is exponential in terms of ||A|| or ||O|| it may be better to use equations 5.5 to 5.8 and substitute $\|L\| = 2^{\|A\|}$.

5.2.3 Complexity of set operations

For the purposes of calculating complexity upper bounds, it is assumed that sets are implemented as ordered lists defined using fixed length arrays. A linear order relationship is assumed to be defined on all possible elements of the set (i.e. set is completely ordered as opposed to partially ordered). This does not affect the result of the algorithms but will avoid unnecessary iterations and searches through the unordered elements of a set. A typical strategy is to number all concepts and implement sets as bit strings with set membership in the set indicated by the bit that correspond to the concept number. This takes advantage of modern CPU architectures with 32 and 64 bit, bitwise operations to improve the efficiency of set operations. Effectively this means that the following complexity bounds will be used on sets:

Operation	Complexity
Set operations: union, copy/assignment, set cardinality	$O(\|Set_1\| + \|Set_2\|)$
Set operations: test for subset and proper subset ($\subset$ and $\subseteq$), test for set equality, set intersection ($\cap$)	$O(\max(\|Set_1\|, \|Set_2\|))$
Single element insertions	$O(1)$
Test for set membership for single element	$O(1)$
Set initialisation	$O(\|Set_1\|)$
Set cardinality	$O(\|Set_1\|)$

For the union, copy/assignment, set cardinality operations on concept intents the bound $O(\|A\|)$ is used whilst the bound $O(\|O\|)$ is used for set operations on concept extents. For subset and proper subset testing, test for set equality and set intersection operations on concept intents $O(\max(\|O'\|))$ is used, whilst $O(\max(\|A'\|))$ is used for concept extents.

These bounds are however not very sharp since in implementation a single CPU operation would for example perform 32 or 64 comparisons on set elements.

5.2.4 AddAtom theoretical performance

The theoretical (worst case) performance of lattice construction algorithms is expressed using the input and output sizes of the algorithms. This is done in two ways: firstly, as the time complexity associated with the construction of the complete lattice of the context. Since the output size is exponential, a second complexity metric called the delay is also used. An algorithm for listing a family of combinatorial structures is said to have *polynomial delay* (Johnson et al. 1988) if it executes at most polynomially many computational steps before either outputting each next structure or terminating. An algorithm is said to have a *cumulative delay* d (Goldberg 1993) if at any point in any execution of the algorithm with any input p the total number of computational steps that have been executed is at most $d(p)$ plus $K.d(p)$ where K is the number of structures that have been output so far. If $d(p)$ can be bounded by a polynomial of p, the algorithm is said to have a *polynomial cumulative delay*.

The number of concepts of the lattice is exponential in the worst case (i.e. a Boolean lattice). Furthermore, the problem of determining the number of concepts in the lattice is NP-complete (Kuznetsov 1989, 2001). In this sense, any lattice construction algorithm unavoidably has intractable (i.e. exponential) worst case behaviour, both in time (since each node has to be generated) and in space (since each node has to be stored). Lattice construction algorithms are therefore differentiated in terms of their time delay characteristics. An algorithm can therefore be considered efficient if it generates the lattice with a polynomial time delay and space linear in the number of all concepts in the lattice. Although "dense" contexts that approach this limit may not be used very often in practice, the theoretical complexity of an algorithm nevertheless expresses an aspect of its performance and is therefore relevant.

A bound for the theoretical worst-case time complexity of AddAtom will be shown below to be $O(\|L\|.\|O\|^2.\max(\|O'\|))$. (The discussion will be based on the optimised form of this construction algorithm, as described in section 4.6.)

As an aid to the discussion, appendix B contains an outline of the algorithm, highlighting its main loops and instructions that add to its complexity characteristics, assist in the analysis of the complexity.

One approach to estimating an upper time bound for constructing the lattice, L, from scratch, is to consider $AddAtom_o_j$ as the upper bound for inserting a single object, o_j, into the lattice (including all the time required for all the recursive calls to AddAtom and all the calls to GetMeet). Let $Housekeeping_o_j$ be the upper bound for doing the housekeeping in preparation for inserting o_j into the existing lattice but excluding the calls to AddAtom. The upper time bound for constructing L would then be:

$$O(\Sigma_{j=1 \text{ to } \|O\|} AddAtom_o_j + \Sigma_{j=1 \text{ to } \|O\|} Housekeeping_o_j)$$

However, instead of attempting to derive upper bounds on each $AddAtom_o_j$., another more global line of reasoning route will be followed.

To this end, let $AddAtom_Total$ be the upper time bound on executing all instructions relating to all calls to AddAtom, in order to insert all objects into L including the calls to GetMeet. Let $Housekeeping_Total$ be the upper bound for the total amount of time taken for the housekeeping and preparation for the construction of the complete lattice. The complexity of the algorithm would then be bounded by:

$$O(AddAtom_Total + Housekeeping_Total)$$

It will be shown below that an upper bound on AddAtom_Total is $O(||L||.||O||^2.max(||O'||))$, and that an upper bound on Housekeeping_Total is $O(||L||.||O||.||A||)$. Under these assumptions, an upper bound on the algorithm to construct the lattice is then:

$$O(||L||.(||O||^2.max(||O'||)+ ||O||.||A||))$$

Since we are interested in order of magnitude estimates of the time for constructing a lattice, L, the lesser term may be left out since it will be dominated by the greater when constructing large lattices. A resulting upper bound (i.e. worst case) estimate for constructing L is thus $O(||L||.||O||^2.max(||O'||))$.

The following three subsections deal with the complexity of each of the three parts of the algorithm.

AddAtom complexity

Looking at the functioning of AddAtom and its parameters, it is clear that there is only one recursive call made to AddAtom for each concept in the lattice. This is since concepts are only created within AddAtom and there are no concepts that are deleted or duplicated. The maximum number of generator concepts for all the lattices L_j is in fact the total number of concepts in the lattice (i.e. $||L||$). Furthermore, for each generator concept that is used as parameter to AddAtom, the outer for loop (using candidate as variable) is executed for each of its parent concepts (a maximum of $||O||$ times for each generator concept). The maximum number of iterations of the outer for loop across all invocations of AddAtom would therefore coincide with $O(||L||.||O||)$.

Within the first and outer for loop of AddAtom, the maximum number of algorithmic steps is determined by the maximum number of steps taken by GetMeet or the inner for loop (using g as variable), whichever is biggest. NewConcept contain only concepts that are prospective parents for the new concept and this list is reduced during each iteration. NewConcept's number of elements is bound by the maximum number of parents of any concept i.e. $O(||O||)$. Within the inner for loop a number of set operations on sets of concept intents are executed. The most complex of these operations is the subset and proper subset tests which is bound by $O(max(||O'||))$. Therefore the number of steps taken by the inner for loop during each iteration of the outer for loop is bound by $O(||O||.max(||O'||))$.

The complexity of the last for loop is dominated by the others and therefore it is not considered in the complexity bound.

Below it will be argued that the complexity of a single call to GetMeet is bound by $O(||O||.max(||O'||))$. The number of algorithmic steps taken by all invocations of AddAtom to inset all objects into the lattice is therefore bound by $O(||L||.||O||.(||O||.max(||O'||) + ||O||.max(||O'||))) = O(||L||.||O||^2.max(||O'||))$.

GetMeet complexity

GetMeet traces a path between the parent of a generator concept and a meet of a subset of Intent(o) somewhere above it. The maximum number of iterations of the outer while loop is bounded by the number of attributes in the intent of generator (i.e. $O(max(||O'||))$. The maximum number of times the for loop can be executed is bounded by the maximum number of parents of a concept (i.e. $||O||$) since each parent has at least one attribute less in its intent. Since the instructions within the while loop is of $O(1)$ complexity, the complexity of a single call to GetMeet is $O(||O||.max(||O'||))$.

Housekeeping_Total complexity

The complexity bound of Housekeeping_Total is determined by the second and outer for loop (with o as variable). Within it the two inner for loops are executed $O(||L_j||)$ times per

object – i.e. $O(\sum_{j=1 \text{ to } \|O\|}\|L_j\|) \approx O(\|L\|.\|O\|)$ times for inserting *all* objects. Within these for loops the number of algorithmic steps of set operations executed are bounded by $O(\|A\|)$ and $O(1)$ for the first and second for loops respectively. The complexity of Housekeeping_Total is therefore bounded by $O(\|L\|.\|O\|.\|A\|)$.

Theoretical complexity comparison

The following table summarises the algorithmic complexity for other construction algorithms[6]

Algorithm	Incremental / Batch	Complexity
Bordat	Batch	Time complexity = $O(\|O\|.\|A\|^2.\|L\|)$ Polynomial delay = $O(\|\|O\|.\|A\|^2)$
CbO, Kuznetsov	Batch	Time complexity = $O(\|O\|^2.\|A\|.\|L\|)$ Polynomial delay = $O(\|O\|^3.\|A\|)$
Chein	Batch	Time complexity = $O(\|O\|^3.\|A\|.\|L\|)$ Polynomial delay = $O(\|\|O\|^3.\|A\|)$
Dowling	Incremental	Time complexity = $O(\|O\|^2.\|A\|.\|L\|)$
Godin	Incremental	Time complexity = $O(\|L\|^3)$
Lindig	Batch	Time complexity = $O(\|O\|^2.\|A\|.\|L\|)$ Polynomial delay = $O(\|O\|^2.\|A\|)$
NextClosure, Ganter	Batch	Time complexity = $O(\|O\|^2.\|A\|.\|L\|)$ Polynomial delay = $O(\|O\|^2.\|A\|)$
Norris	Incremental	Time complexity = $O(\|O\|^2.\|A\|.\|L\|)$
Nourine	Incremental	Time complexity = $O((\|O\| + \|A\|).\|O\|.\|L\|)$
Valtchev	N/A	The complexity of the procedure assembling lattices L_1 and L_2 into the global lattice L is $O((\|O\| + \|A\|)(\|L_1\|.\|L_1\| + \|L\|.\|A\|))$ L_1 and L_2 can however be built in parallel.
AddAtom	Incremental	Time complexity = $O(\|L\|.\|O\|^2.\max(\|O'\|))$

For the purpose of direct comparison and since $\|O'\| < \|A\|$, $\|O'\|$ can be substituted with $\|A\|$. A slightly less sharp complexity bound for AddAtom is therefore $O(\|L\|.\|O\|^2.\|A\|)$.

The AddAtom complexity estimate is therefore cubic in nature relative to the lattice size. This is a feature that it shares with most other algorithms. Since this estimate is not quadratic relative to the number of concepts, as is the Nourine algorithm, it might seem that AddAtom does not offer very much in terms of theoretical performance overall.

The complexity bound as stated is however not very sharp. One area where the theoretical complexity is overstated is within GetMeet. The maximum length of a path in GetMeet is stated as $\|O'\|$ but in general no path would stretch from 0_L to 1_L (implied by a

[6] Where these algorithms have been improved as discussed in Kuznetsov and Obiedkov (2002), the complexity of the improved algorithm is given.

path length of $||O'||$). It is interesting to note that if it could be proved that GetMeet return each of the respective meet concepts above a particular generator concept only once, the total combined length of all paths traced in calls to GetMeet to insert a single object would not exceed the total number of concepts in the lattice. This is because none of such paths can cross each other except at the meet of a subset of Intent(o). Under this assumption the maximum number of iterations of inner for loop for each concept on the path is the number of parents of that concept. The total number of iterations of the for loop across all invocations for the insertion of one object is therefore the total number of arcs in the lattice. Therefore $O(\sum_{j=1 \text{ to } ||O||} ||\prec j||) \leq O(||A||.||O||.||L||)$ (or $O(\frac{1}{2}||A||.||L||)$ in the case of a Boolean lattice) would be an upper bound on the complexity of all calls to GetMeet across all the recursive calls to AddAtom to insert all objects of the context (for Boolean lattices that is). For the algorithm as stated in section 4.6, used in the wide comparison study in section 5.4 this is not the case, but the version of the algorithm in appendix A makes use of this optimisation. The complexity bound derived here is however still an upper bound for this algorithm.

Another area where the theoretical complexity bound is not very sharp is in the AddAtom part of the algorithm. The theoretical complexity bound assumes that the number of iterations of the outer for loop is bounded by the number of arcs in the lattice. In the algorithm itself however, only concepts with at least some attribute in common with the to-be inserted object will be visited and therefore not all arcs will be "followed" during the iterations of the for loop. For non-Boolean lattices with $||O'|| << ||A||$[7] this will be a very significant portion of the concepts in the lattice that will not be visited by the for loop. To quantify this further, consider a Boolean lattice and an object intent o'. There are in general $2^{||A||-||o'||}$ concepts in the lattice that have no attribute in common with o'. Clearly for non-Boolean lattices this number will be significantly less, but for many contexts this is still very significant, indicating an overestimation of the overall complexity.

The use of $||O||$ as the upper bound to the number of parents of a lattice leads to an overestimate of the total number of arcs in a lattice. A case in point is the fact that Boolean lattices have on average $\frac{1}{2}.||A||$ inbound or outbound arcs per concept – far fewer than the upper bound $||O||$ used here.

The AddAtom algorithm can be easily adapted to be symmetrical and insert attributes into the lattice and link them to their extents instead of inserting objects into the lattice and linking them to their intents. Using the same reasoning AddCoatom, the dual incremental concept lattice construction algorithm would have a complexity bound of $O(||L||.||A||^2.\max(||A'||))$ which may include smaller terms than that of AddAtom.

The best way to obtain clarity on this and other issues is via empirical studies. The next two sections present the results of the pilot and wider empirical studies. The results of the empirical studies support the claims on the over estimation of the theoretical complexity of AddAtom and indicate that it does indeed perform very well and is often the best performer of the algorithms surveyed.

5.3 EMPIRICAL PERFORMANCE: PILOT STUDY

The pilot study was conducted to establish the relative performance of AddAtom using the code described in chapter 7 to seek *prima facie* evidence that would justify a wider study. The basic strategy of the pilot study was to implement the incremental lattice construction

---

[7] The notation a << b indicates that A is significantly smaller than b by some measure.

algorithms of Godin et al. (1995b)[8] and Carpineto and Romano (1993) using the same base code and data structures as AddAtom (described in chapter 7). The pseudo code of the implemented algorithm can be found in appendix A (note that there are differences to the algorithm in section 4.6). This would serve as a good indication of the relative performance of the algorithm and clearly indicate if the time performance was worse (or not) than that of the Godin or Carpineto algorithms, justifying the effort of a wider study.

Note that for the pilot study EA-lattices were generated and the Godin and Carpineto algorithms were modified to generate EA-lattices.

In addition to the Godin and Carpineto algorithms, the Grand algorithm (Oosthuizen (1991)) was also available for comparison but due to it using different data structures and utility functions as well as being implemented in a different programming language (refer to chapter 7 for further discussion), it was not included in the study since it would not make a apples-with-apples comparison possible. The performance of Grand is however worse than AddAtom in all types of contexts by a significant margin.

The pilot study comparison showed that AddAtom is indeed faster than the Godin and Carpineto algorithms and this suggested that a more thorough study of the algorithm's performance would be worth while. However, it also exposed the fact that the code base and data structures were inefficient and that a wider study would require a revised strategy towards the data structures and utility functions (also refer to chapter 7).

For the pilot study, care was taken to ensure a valid comparison. To this end, the algorithms were implemented on the same base-code and performance tests run under the same platform. However, any inefficiency in the particular implementation approach and data structures could have negatively penalised the relative performance of the Godin and Carpineto algorithms. This is because the data structures used could have conceivably suited AddAtom better and could have given it an unfair advantage under the experimental comparison. To avoid this situation from influencing the outcome, a number of additional performance metrics, other than time, were collected. These metrics tracked basic lattice operations such as lattice closures and set operations and did confirm the trend of the time based results.

A number of artificial and "natural" data sets were used as contexts for the experimental comparisons. The artificial data sets were randomly generated whilst the natural data sets were taken from the well-known UCI Machine Learning Repository (Blake and Merz 1998).

The following table provides an overview of the data sets and describes the notation used to identify the data sets.

Data set	Description
Rnd-100-YY-XXX	A random data set of XXX objects. Each object possesses exactly YY attributes, randomly chosen from 100 possible attributes. When referring to the data set as a whole, the notation Rnd-100-YY is used.
Bool-XX	A data set of XX objects. The data set has XX attributes. Every object has XX – 1 attributes and differs from each of the other objects in only one attribute. The resulting lattice of this arrangement forms a Boolean lattice. When referring to the data set as a whole, the notation Bool is used.

[8] The implementation follows the description in Godin et al. (1995b) and not the improvements suggested by Kuzetnov and Obiedkov (2002).

Data set	Description
SPECT	A natural data set taken from the UCI repository called the Single Proton Emission Computed Tomography (SPECT) set. The dataset has 22 binary feature patterns and one overall diagnosis attribute. When referring to the data set as a whole, the notation SPECT is used.
BCW-XXX	The Breast-Cancer-Wisconsin natural data set taken from the UCI repository. XXX indicates the number of objects in the context. Objects were randomly selected from the data set. The set of discrete attributes was used unaltered. The total data set consists of 698 objects, each object has 10 attributes, whilst each of the 10 attributes could assume any one of 10 values. Some objects do not possess a value for a specific attribute (the value is unknown). Such objects were still included in the set and the unknown value was treated as an eleventh value of that specific attribute. Each value of each attribute was treated as a separate attribute in the experimental results. Theoretically there were thus 10 $\times$ 11 = 110 attributes, but in practice the data set contained only 86 attributes since all attribute values were not observed. When referring to the data set as a whole, the notation BCW is used.

The key metrics describing the data sets that were used are as follows:

Set name	$\|O\|$	$\|A\|$	$\|I\|$	$\|L\|$	$\|<\|$	$\|O'\| / \|A\|$
Rnd-100-10-40	40	98	379	312	871	10%
Rnd-100-10-45	45	100	434	351	990	10%
Rnd-100-10-50	50	98	477	413	1198	10%
Rnd-100-10-75	75	100	725	697	2197	10%
Rnd-100-10-100	100	100	975	1058	3425	10%
Rnd-100-10-150	150	100	1433	1957	6567	10%
Rnd-100-10-200	200	100	1915	3031	10423	10%
Rnd-100-30-15	15	100	392	426	1206	26%
Rnd-100-30-20	20	100	520	799	2588	26%
Rnd-100-30-25	25	100	643	1313	4589	26%
Rnd-100-30-30	30	100	779	2183	7962	26%
Rnd-100-30-35	35	100	914	3329	12623	26%
Rnd-100-30-40	40	100	1039	4288	16652	26%
Bool-07	7	7	42	128	448	86%
Bool-08	8	8	56	256	1024	88%

Set name	$\|O\|$	$\|A\|$	$\|I\|$	$\|L\|$	$\|<\|$	$\|O'\| / \|A\|$
Bool-09	9	9	72	512	2304	89%
Bool-10	10	10	90	1024	5120	90%
Bool-11	11	11	110	2048	11264	91%
Bool-12	12	12	132	4096	24576	92%
BCW-030	30	69	300	240	564	14%
BCW-035	35	71	350	317	795	14%
BCW-040	40	75	400	312	751	13%
BCW-045	45	77	450	323	783	13%
BCW-050	50	84	500	499	1349	12%
BCW-075	75	84	750	701	1948	12%
BCW-100	100	84	1000	1091	3331	12%
BCW-200	200	86	2000	1704	5455	12%

Tests for the pilot study were performed on an Intel 110 mhz Pentium processor based platform with 256 megabytes of memory under the Windows 2000 Professional operating system. Note that EA-lattices were generated for the pilot study.

The following graphs summarise the results.

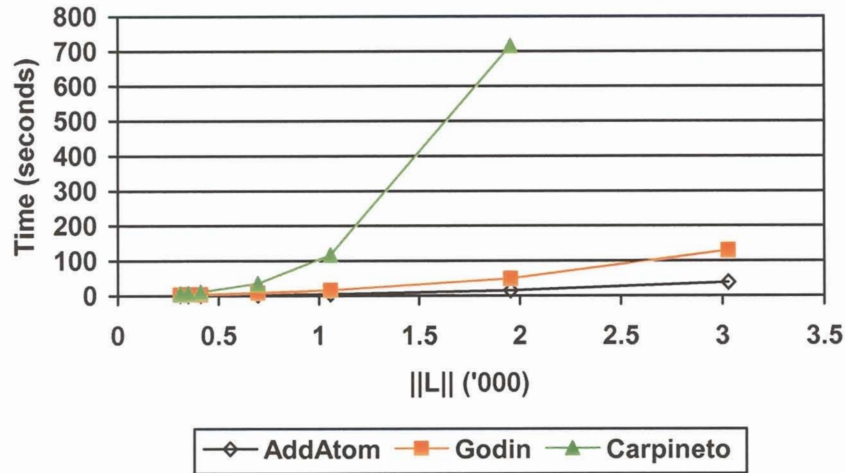


Figure5.3: Pilot study performance test results for the Rnd-100-10 data set plotting time performance against lattice size ($||O'||/||A||$ = 10%)

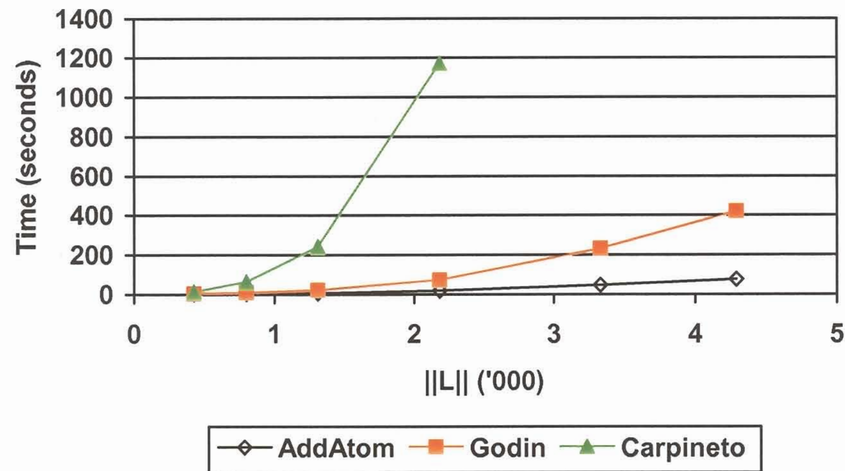


Figure 5.4: Pilot study performance test results for the Rnd-100-30 data set plotting time performance against lattice size ($||O'||/||A||$ = 26%)

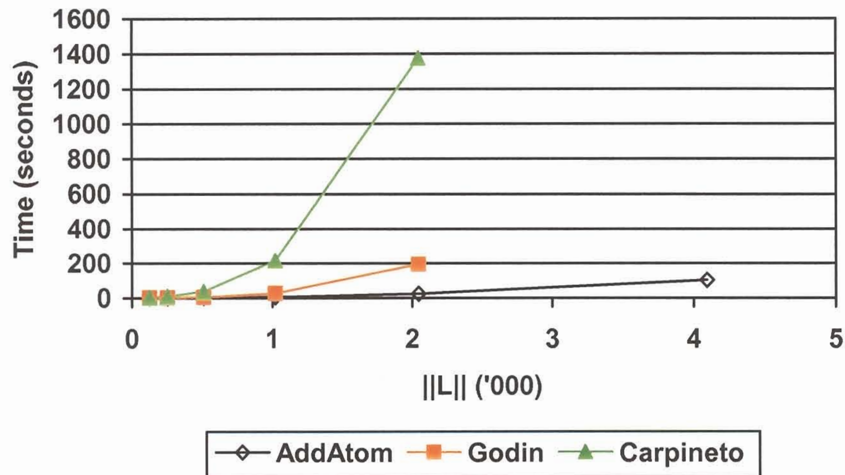


Figure 5.5: Pilot study performance test results for the Bool data set plotting time performance against lattice size (||O'||/||A|| = 86-92%)

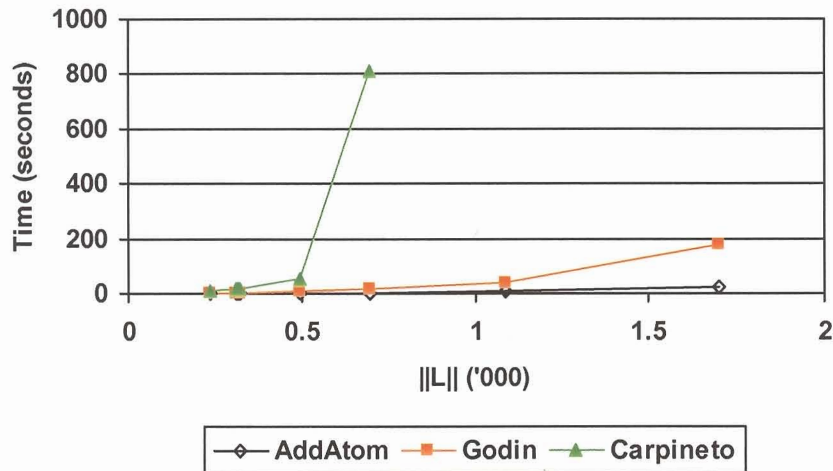


Figure 5.6: Pilot study performance test results for the BCW data set plotting time performance against lattice size (||O'||/||A|| = 14-12%)

The following table summarise the relative algorithmic performance on the largest contexts in the experimental comparison (the construction time is expressed as a percentage of the time of the fastest algorithm for a particular context):

	Performance index (fastest algorithm = 100%)						
	Rnd-100-10		**Rnd-100-30**		**Bool**	**BCW**	
‖L‖	1955	3029	2183	4288	2046	699	1702
Context density	10%	10%	26%	26%	86-92%	14-12%	14-12%
AddAtom	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
Godin	320%	337%	360%	540%	804%	400%	738%
Carpineto	4753%		5840%		5717%	27067%	

Discussion: The results clearly indicate that the Carpineto algorithm is not very efficient in constructing lattices of any type of context. The AddAtom algorithm performance was consistently better than that of the Godin algorithm. The relative performance was however not the same across the different contexts. The Godin algorithm performed worst, compared to AddAtom in the Boolean and BCW contexts.

Set operations are operations such as the union, intersection, subtraction etc. The graph below compares the number of set operations used by each of the algorithms in the construction of the lattices. (Similar results were obtained with all data sets.)

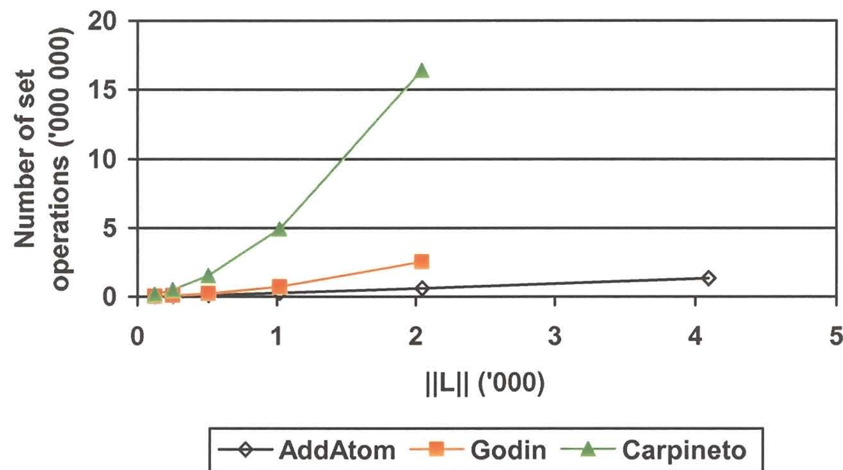


Figure 5.7: Pilot study performance test results for the Bool data set plotting the number of set operations against lattice size ($||O'||/||A||$ = 86-92%)

Node references are instructions that require the reference to a member variable of a concept object (e.g. its intent, extent or testing whether it is a concept that still exists in L). The graph below compares the number of node operations used by each of the algorithms in the construction of the lattices.

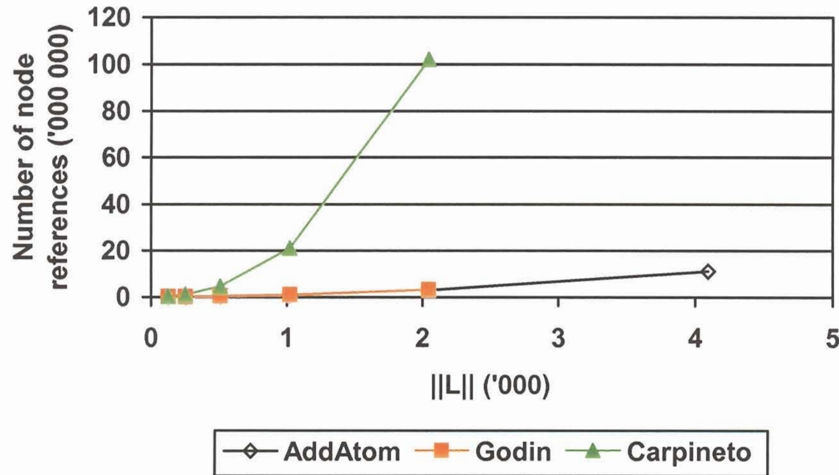


Figure 5.8: Pilot study performance test results for the Bool data set plotting the number of node operations against lattice size (||O'||/||A|| = 86-92%)

The graph below compares the number of closure operations used by each of the algorithms in the construction of the lattices.

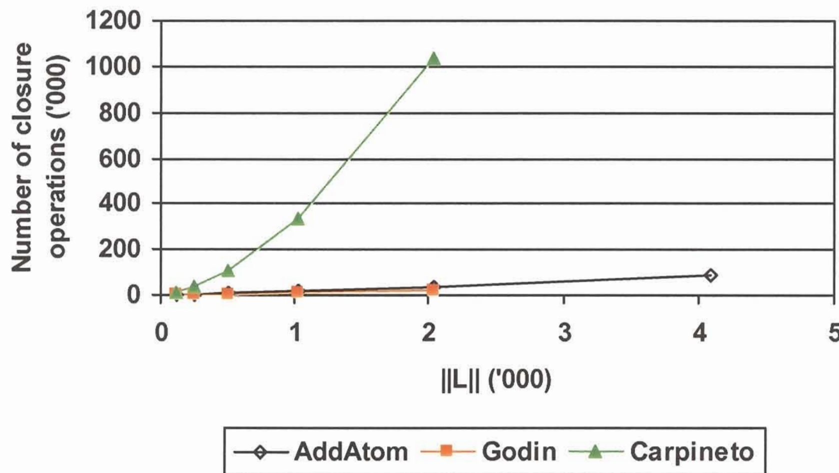


Figure 5.9: Pilot study performance test results for the BCW data set plotting the number of closure operations against lattice size (||O'||/||A|| = 86-92%)

For the sake of brevity, all performance test results are not included here. It is interesting to note that the biggest performance gap was in the natural data set (BCW). The results show that the number of operations in general follows the trend of the time-based results, except that the Godin algorithm uses less closure operations. The use of less closure operations is a feature of the Godin algorithm which is defined in terms of set operations on intents and extents rather than closure operations and is therefore not inconsistent with the results.

The results of the pilot experimental performance tests indicate that AddAtom indeed performs very well compared to the Godin and Carpineto implementations. The relative performance was good both from a time and number of basic operations perspective. This result is in line with the relative theoretical performance of AddAtom.

However the time taken to construct the lattices was well outside that reported by other authors such as Kuznetsov and Obiedkov (2001, 2002) when compensating for the differences in CPU size. This issue is further discussed chapter 7 and is mainly due to structural inefficiencies in the data structures and utility functions used. In addition, the tests did not include comparisons with the current best incremental lattice construction algorithms (e.g. that of Nourine with the best theoretical complexity). The performance tests were therefore not conclusive and lead to a further set of wider experimental comparisons.

In the wider performance comparison discussed in the next section it can be seen that the relative performance of the Godin implementation is in line with the results obtained in the pilot study and that AddAtom indeed perform better than other algorithms in most contexts.

5.4 EMPIRICAL PERFORMANCE: WIDE STUDY

In the past number of years a number of lattice construction algorithms with improved performance have been published. Meanwhile the computing power and memory capacity of even personal computers are now sufficient for building very large lattices. The result of these two factors is that the relative importance of fast and efficient construction algorithms has decreased whilst making more advanced applications using larger concept lattices possible. For practical purposes it is however still relevant to make use of the best known algorithms wherever possible since computing resources are not infinite.

The first comparative study of several algorithms was Guénoche (1990) whilst more recently Kuznetsov and Obiedkov (2001, 2002) have studied a large number of the most well-known algorithms. Kuznetsov and Obiedkov (2002) is currently the most comprehensive study of algorithmic performance and therefore provide a useful benchmark with regards to the algorithmic performance of lattice construction algorithms. The author has collaborated with S.A. Obiedkov (2003) and a version of the AddAtom algorithm adapted to Obiedkov's base code and data structures was implemented and included in Obiedkov's on-going research. The results[9] show that AddAtom performs very well in constructing lattices from most data sets compared to other incremental and non-algorithms and is often the best performer.

The key metrics describing the data sets that were used are as follows:

Data set used	$\|O\|$	$\|A\|$	$\|I\|$	$\|L\|$	$\|<\|$	$\|O'\| / \|A\|$
Rnd-100-4-100	100	100	400	2222	4936	4%
Rnd-100-4-200	200	100	800	4256	9939	4%
Rnd-100-4-300	300	100	1200	6522	15496	4%
Rnd-100-4-400	400	100	1600	9153	22050	4%

[9] The author acknowledge contribution of S.A. Obiedkov in conducting the performance tests and making the data available for inclusion in section 5.4.

<table>
<tr><th>Data set used</th><th>||O||</th><th>||A||</th><th>||I||</th><th>||L||</th><th>||< ||</th><th>||O'|| / ||A||</th></tr>
<tr><td>Rnd-100-4-500</td><td>500</td><td>100</td><td>2000</td><td>12032</td><td>29598</td><td>4%</td></tr>
<tr><td>Rnd-100-4-600</td><td>600</td><td>100</td><td>2400</td><td>15093</td><td>37943</td><td>4%</td></tr>
<tr><td>Rnd-100-4-700</td><td>700</td><td>100</td><td>2800</td><td>18213</td><td>46792</td><td>4%</td></tr>
<tr><td>Rnd-100-4-800</td><td>800</td><td>100</td><td>3200</td><td>21511</td><td>56601</td><td>4%</td></tr>
<tr><td>Rnd-100-4-900</td><td>900</td><td>100</td><td>3600</td><td>24816</td><td>66522</td><td>4%</td></tr>
<tr><td>Rnd-100-25-010</td><td>10</td><td>100</td><td>250</td><td>1286</td><td>3568</td><td>25%</td></tr>
<tr><td>Rnd-100-25-020</td><td>20</td><td>100</td><td>500</td><td>7057</td><td>23568</td><td>25%</td></tr>
<tr><td>Rnd-100-25-030</td><td>30</td><td>100</td><td>750</td><td>18962</td><td>69122</td><td>25%</td></tr>
<tr><td>Rnd-100-25-040</td><td>40</td><td>100</td><td>1000</td><td>37296</td><td>143136</td><td>25%</td></tr>
<tr><td>Rnd-100-25-050</td><td>50</td><td>100</td><td>1250</td><td>63118</td><td>251416</td><td>25%</td></tr>
<tr><td>Rnd-100-25-060</td><td>60</td><td>100</td><td>1500</td><td>95619</td><td>391827</td><td>25%</td></tr>
<tr><td>Rnd-100-25-070</td><td>70</td><td>100</td><td>1750</td><td>135618</td><td>569250</td><td>25%</td></tr>
<tr><td>Rnd-100-25-080</td><td>80</td><td>100</td><td>2000</td><td>181877</td><td>778487</td><td>25%</td></tr>
<tr><td>Rnd-100-25-090</td><td>90</td><td>100</td><td>2250</td><td>236729</td><td>1031820</td><td>25%</td></tr>
<tr><td>Rnd-100-25-100</td><td>100</td><td>100</td><td>2500</td><td>300257</td><td>1329793</td><td>25%</td></tr>
<tr><td>Rnd-100-50-10</td><td>10</td><td>100</td><td>500</td><td>5537</td><td>22152</td><td>50%</td></tr>
<tr><td>Rnd-100-50-20</td><td>20</td><td>100</td><td>1000</td><td>128748</td><td>644901</td><td>50%</td></tr>
<tr><td>Rnd-100-50-30</td><td>30</td><td>100</td><td>1500</td><td>752491</td><td>4112730</td><td>50%</td></tr>
<tr><td>Rnd-100-50-40</td><td>40</td><td>100</td><td>2000</td><td>2493490</td><td>14388396</td><td>50%</td></tr>
<tr><td>Bool-10</td><td>10</td><td>10</td><td>90</td><td>1024</td><td>5120</td><td>90%</td></tr>
<tr><td>Bool-11</td><td>11</td><td>11</td><td>110</td><td>2048</td><td>11264</td><td>91%</td></tr>
<tr><td>Bool-12</td><td>12</td><td>12</td><td>132</td><td>4096</td><td>24576</td><td>92%</td></tr>
<tr><td>Bool-13</td><td>13</td><td>13</td><td>156</td><td>8192</td><td>53248</td><td>92%</td></tr>
<tr><td>Bool-14</td><td>14</td><td>14</td><td>182</td><td>16384</td><td>114688</td><td>93%</td></tr>
<tr><td>Bool-15</td><td>15</td><td>15</td><td>210</td><td>32768</td><td>245760</td><td>93%</td></tr>
<tr><td>Bool-16</td><td>16</td><td>16</td><td>240</td><td>65536</td><td>524288</td><td>94%</td></tr>
<tr><td>Bool-17</td><td>17</td><td>17</td><td>272</td><td>131072</td><td>1114112</td><td>94%</td></tr>
<tr><td>Bool-18</td><td>18</td><td>18</td><td>306</td><td>262144</td><td>2359296</td><td>94%</td></tr>
<tr><td>SPECT</td><td>267</td><td>23</td><td>2042</td><td>21549</td><td>110589</td><td>33%</td></tr>
</table>

The following set of algorithms that generate the line diagram of a (CFA) lattice were used for the comparison. Note that these algorithms incorporate the improvements and changes described in Kuznetsov and Obiedkov (2002) alluded to between brackets.

- CbO.
- Ganter (use binary search to find the canonical generation of a concept).
- Norris (use a tree).
- Bordat (not using a tree).
- Godin (use heuristic based on the size of concept extents).
- Lindig.
- Nourine.
- Valtchev (using horizontal splitting of the object set).
- AddAtom (interpretation of Obiedkov (2003) similar to the algorithm in section 4.6 but using a different version of GetMeet that has a greater complexity bound namely $O(\max(||O'||^2.||O||))$.

Note that for the wide study, FCA-lattices rather than EA-lattices were generated.

Tests were performed on a Pentium 4–2GHz with 1 Gigabyte RAM. Also note that a much faster processor was used than that used in the pilot study.

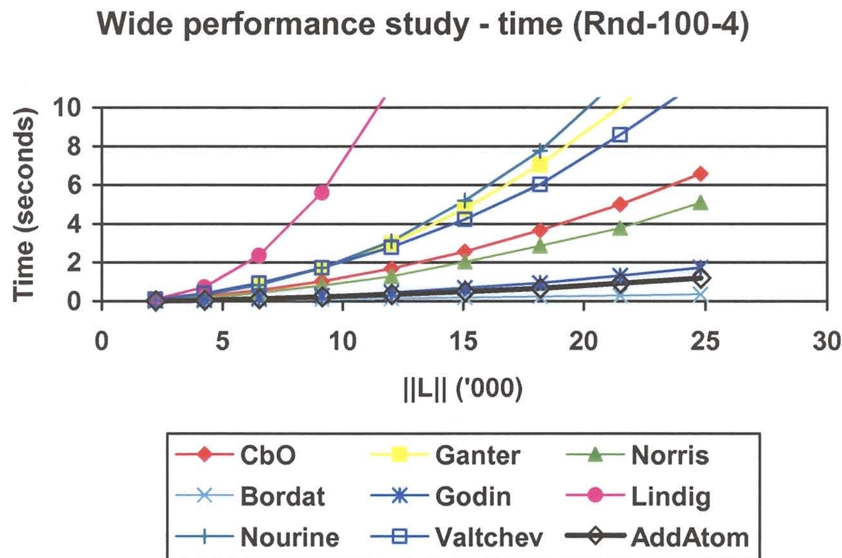


Figure 5.10: Wide performance study test results for the Rnd-100-4 data set plotting the time performance against lattice size ($||O'||/||A||$ = 4%)

Analysis: In random contexts with a very low density (4%) the Bordat algorithm performs the best with AddAtom coming in second.

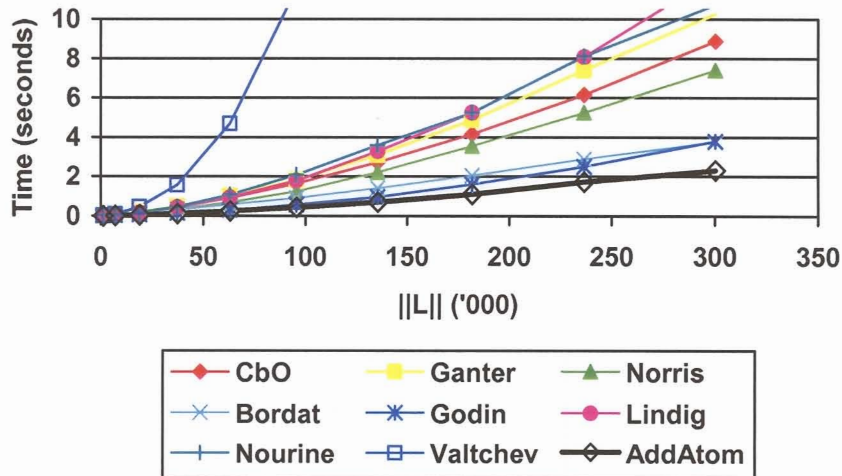


Figure 5.11: Wide performance study test results for the Rnd-100-25 data set plotting the time performance against lattice size (||O'||/||A|| = 25%)

Analysis: In random contexts with a relatively modest density (25%) the AddAtom algorithm performs the best with Godin coming second.

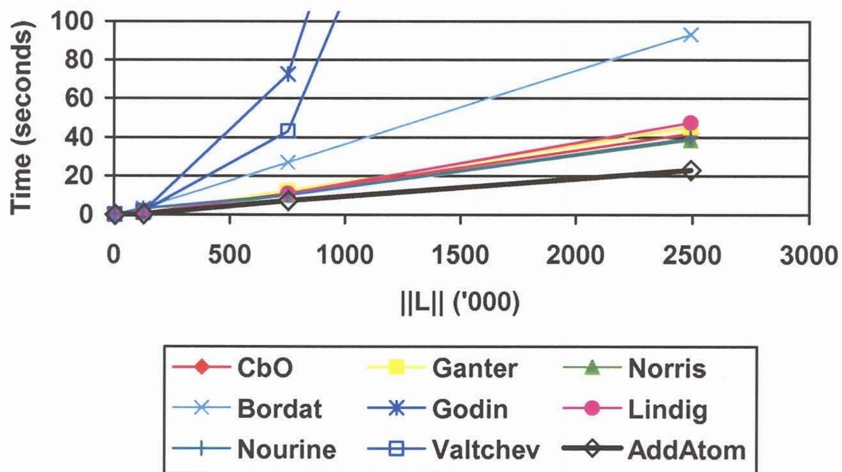


Figure 5.12: Wide performance study test results for the Rnd-100-50 data set plotting the time performance against lattice size (||O'||/||A|| = 50%)

Analysis: In random contexts with a relatively high density (50%) AddAtom is still the best performer with Norris second and Nourine a close third.

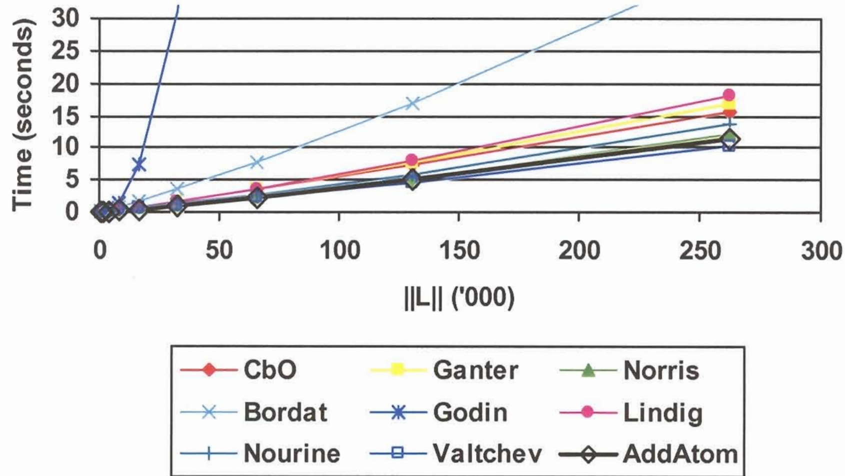


Figure 5.13: Wide performance study test results for the Bool data set plotting the time performance against lattice size (||O'||/||A|| = 90-94%)

Analysis: In artificial contexts that create Boolean lattices (very high density 94%), Valtchev performs the best with AddAtom second and Norris a close third. These contexts can be described as the theoretical limit of random contexts in the sense that the attributes in these contexts are functionally completely independent in that no association rules can be formulated upon any of the attributes of these contexts whereas random contexts of a similar density generated using random functions that are not exceedingly large will always have at least some association rules on some of the attributes.

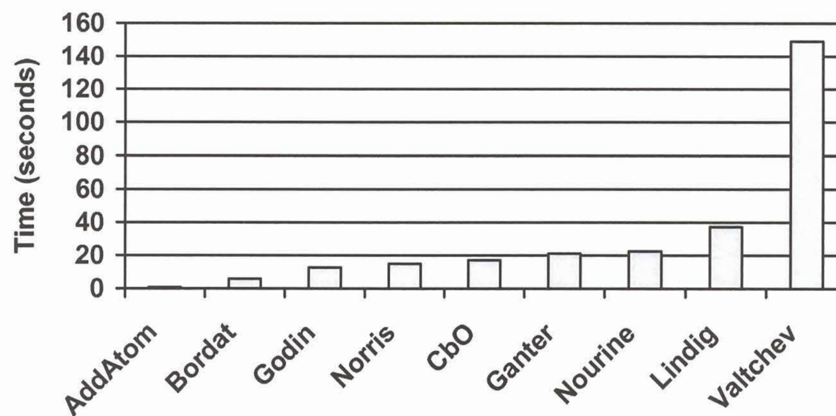


Figure 5.14: Wide performance study test results for the SPECT data set plotting the time performance for different algorithms (||O'||/||A|| = 33%)

Analysis: In a context taken from natural or "real life" data the performance gap between AddAtom and the nearest best performers (in this case Bordat and Godin) even bigger than with artificial contexts. This observation is however based on only one data point.

The table below summarise the performance of the algorithms relative to the best performing algorithm in the largest of each type of context. The letters "B" and "I" are used to indicate which of the algorithms are incremental and batch algorithms.

	Performance index (fastest algorithm = 100%)				
	Rnd-100-4	**Rnd-100-25**	**Rnd-100-50**	**Bool**	**SPECT**
$\|\|L\|\|$	2222	300257	2493490	2493490	21549
Context density	4%	25%	50%	90-94%	33%
CBo (B)	1884%	385%	182%	151%	1973%
Ganter (B)	3960%	445%	196%	164%	2462%
Norris (I)	1461%	320%	166%	119%	1743%
Bordat (B)	**100%**	162%	404%	374%	684%
Godin (I)	493%	164%	3057%	19431%	1457%
Lindig (B)	24636%	529%	206%	177%	4330%
Nourine (I)	4564%	465%	172%	133%	2606%
Valtchev (-)	3261%	5411%	2326%	**100%**	17304%
AddAtom (I)	341%	**100%**	**100%**	113%	**100%**

From the above table it is clear that the various algorithms perform differently across the various context densities. Only AddAtom perform very well (albeit not always the best) over the range of context densities. The graph below depicts this graphically but the data points used are only that of the largest contexts. It shows the average time taken per inserted concept for the various artificial contexts and plots it against the density of the context. It clearly shows that both Bordat and Valtchev perform well at opposite ends of the spectrum, but that each (especially Valtchev) performs relatively poorly at the other end. In the graph Boolean contexts are seen as the extreme case of random contexts since there is no implication rule on any of the attributes as would be expected in a infinitely large random context with $||A'|| = ||O'|| = ||A|| - 1 = ||O|| - 1$.

Based on the results and compared to the other algorithms AddAtom is more robust in having less variation in its performance across the range of random contexts in that the standard deviation of the time taken per concept to construct the lattice for these various context were lower. Therefore should a heuristic be used to select the algorithm in a multi-algorithm strategy and the incorrect algorithm be chosen, AddAtom could behave in a more predictable and narrower range of performance that others.

The relative closeness of the AddAtom and Nourine performance confirms that the theoretical performance upper bound of AddAtom as derived is overstated.

Wide performance study - time (largest artificial contexts)

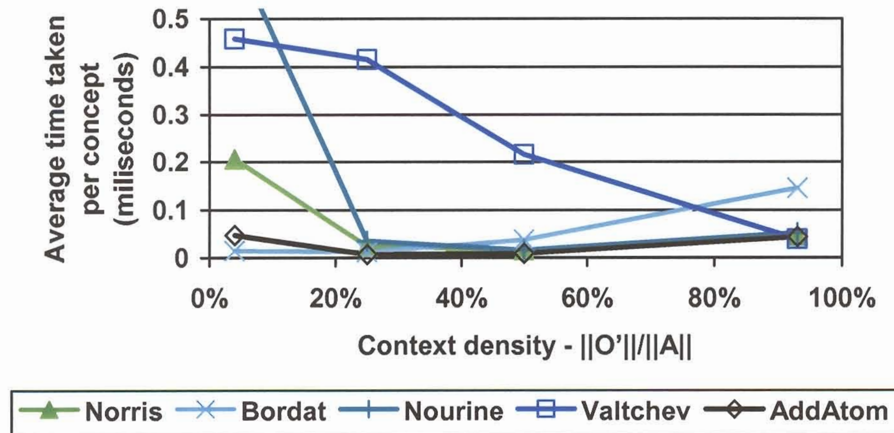


Figure 5.15: Wide performance study test results for the largest of the artificial contexts comparing the time taken per concept to the context density

It should be noted that the graph is constructed on very few data points on contexts of different sizes and the picture it presents is therefore not conclusive. It does however lead to a hypothesis: "for random or artificial contexts, the density of the context is a good predictor of algorithm with the best algorithmic performance".

When looking at the natural context, there is however a significant difference between the time taken to build the lattice for the SPECT dataset compared to any of the random contexts with similar sized lattices. Once again the data is insufficient to support any conclusions, but the observation lead to a number of hypotheses such as "the algorithmic performance of the various algorithms is significantly different between random and 'natural' contexts" and "context density is not a good predictor of algorithmic performance for 'natural' contexts". Investigating these hypotheses is an area for further study and will be useful for a lattice construction strategy of using a combination of algorithms that depend on the context and event the object to be inserted. Such a strategy is proposed by Kuznetsov and Obiedkov (2002).

As was discussed in section 5.2.4, the theoretical complexity bound of AddAtom derived here is not very sharp, especially for non-Boolean lattices and the actual performance of the algorithm may be significantly better than what is suggested by the cubic nature of the theoretical complexity. The experimental results support this claim and also indicate the extent to which the performance is dependent on the nature of the context itself. It is also interesting that the three best performing algorithms in this study namely Bordat, Valtchev and AddAtom are not of the same type. Bordat is a batch algorithm, AddAtom incremental whilst the approach of Valtchev can not be classified as either. AddAtom is however the best performing incremental algorithm for the contexts included in the study.

The good performance of AddAtom can be attributed to the focussed way it traverses the L_{n-1} lattice to find generator concepts. Instead of visiting a substantial number of the concepts in L_{n-1}, it focuses only on concepts that have at least some attributes in common with the object to be inserted. At each generator concept, the search for additional generator concepts higher up in the lattice is more and more focussed resulting in an efficient algorithm.

Lastly, it should be noted that the AddAtom algorithm and not AddCoatom was in the tests. As was indicated in section 5.2, AddCoatom has a smaller theoretical performance

bound. In preliminary tests an approximately 10% performance improvement (in terms of the number of set and lattice operations) in all random contexts was noted on the code used for the pilot performance study when using AddCoatom. Although this might reflect inefficiencies in the implementation; it is likely that some improvement can be gained in certain contexts by using AddAtom in stead of AddCoatom. Other algorithms may also benefit from this approach. This is another area of further study.

5.5 CONCLUSIONS

From the results of the performance tests it is clear that AddAtom is a very good lattice construction algorithm and compares well with other algorithms from a theoretic but especially from an experimental point of view. It does however not perform the best across all types of contexts. In contexts with either very low or very high densities other algorithms (Bordat in low density contexts and Valtchev in high density contexts) perform better than AddAtom. In these contexts AddAtom is still the second best performer. AddAtom is however the best performing incremental lattice construction algorithm. Further study is however required to better quantify the size and nature of the AddAtom performance relative to other algorithms in especially non-random contexts.

Kuznetsov and Obiedkov (2001, 2002) have suggested that the choice of algorithm should be based on the context. The study shows that the context density may be a good predictor for such a test in the case of random contexts but due to the small number of data points as well as the completely different performance on natural data sets it is clear that this is a topics for further study. The density of a context can be pre-computed with relatively little effort before building the concept lattice of a context and could be a very useful tool in selecting an appropriate lattice construction algorithm for building a lattice. This strategy may also benefit from using algorithms adapted to insert attributes (i.e. construct the lattice column by column from the cross table) as well as objects (i.e. constructs the lattice row by row from the cross table) into concept lattices.

On natural / “real world” contexts AddAtom performs particularly well compared to random contexts. Although the performance tests in this study do not fully explore this, AddAtom seems to be a prime contender, for the best performing algorithm for natural data contexts, especially since its performance range was relatively limited compared to other algorithms.

Chapter 6: Compressed pseudo-lattices

'Everything should be made as simple as possible, but not simpler.'

Albert Einstein

In this chapter we define the notion of a compressed pseudo-lattice. A compressed pseudo-lattice essentially consists of a sublattice embedded in a bipartite graph. This allows for the reduction of the size of the lattice but allows control over the amount of information that is lost in the process. The aim of this is to simplify the lattice but still retain the essence of the context it represents.

The discussion starts by introducing and developing an IR (Information Retrieval) problem and develops the problem domain into one where it is argued that a compressed pseudo-lattice plays a significant role. The properties and use of compressed pseudo-lattices are discussed as well as their interpretation. It is argued that this approach may have significant advantages over approaches using the complete lattice in particular areas.

6.1 A BIPARTITE DATABASE AND QUERY OPERATION

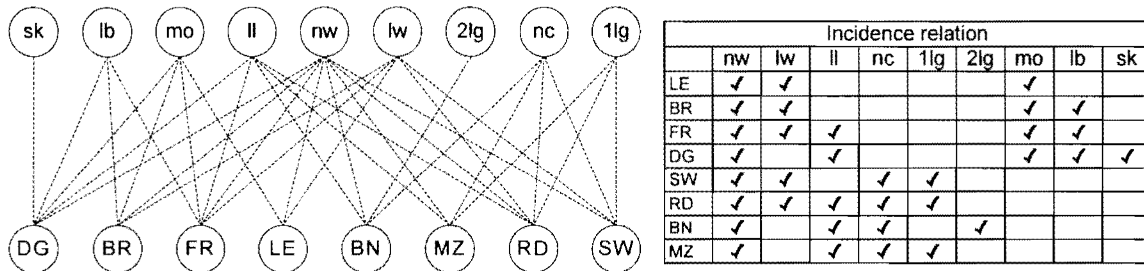
We now sketch an IR problem domain and do not consider lattices until the next section. For this problem domain we define a database $D = \langle S, \leqslant \rangle$ related to a context $C = \langle O, A \rangle$ as consisting of a set, S, of concepts which are partially ordered by the relation $\leqslant$ (this set need not be a lattice although it is one of the possibilities). An incidence relation I can be derived from the partial order that describes which objects possesses which attributes. The database is restricted in that the maximal elements are the attribute concepts (representing A) in C and the minimal elements are the object concepts (representing O) in C. In addition D may contain any number of intermediate concepts M (i.e. S = attribute concepts $\cup$ intermediate concepts $\cup$ object concepts). The *upward closure* of any concept c, denoted by $\text{UpwardClosure}(D, c)$, is the set of all concepts greater than or equal to c in terms of the partial order, $\leqslant$. The *downward closure* is the set of concepts that are less than or equal to c, and is denoted by $\text{DownwardClosure}(D, c)$. The *extent* of a concept is defined as the set of objects in its downward closure. Similarly the *intent* of a concept is the set of attributes in its *upward closure*.

We consider the problem of retrieving a set of objects relevant (in some abstract and as yet undefined way) to a specific query. The query is formulated as a set of attributes in the form $Q = \{a_1, a_2, ..., a_m\}$ (i.e. $Q \subseteq A$). Different query operations, taking Q as a parameter, can be defined on D. The result of a specific query operation O based on database D with respect to query Q is denoted by $O(D, Q)$. A query operation may return any number of concepts from D, the objective being the identification of concepts relative to the query Q.

Notwithstanding the foregoing, we choose to interpret the final results in terms of objects, namely those objects that are in the union of the extents of the concepts returned by the query operation. As a consequence the results of a query can be interpreted as clusters of objects represented or referenced by the concepts returned by the query operation.

Different query operations may be evaluated in terms of the well-known information retrieval (IR) metrics, precision (the proportion of objects returned by $O(D, Q)$ that are

relevant in relation to all objects returned by O(D, Q)) and recall (the proportion of relevant objects returned by O(D, Q) in relation to all relevant objects in the database, D) (Salton 1989). Conversely using the same query operation, different databases of the same context can be evaluated against each other in terms of these metrics. Clearly only concepts in a given database can be returned. Therefore it can be expected, for a given query, that a database containing more 'meaningful' concepts will return concepts that result in higher precision and recall values. We argue that a compressed pseudo-lattice (as defined later on in this chapter) is a versatile data structure to represent various databases of this type and could prove useful in researching information retrieval and machine learning strategies.



Incidence relation									
	nw	lw	ll	nc	1lg	2lg	mo	lb	sk
LE	✓	✓					✓		
BR	✓	✓					✓	✓	
FR	✓	✓	✓				✓	✓	
DG	✓		✓				✓	✓	✓
SW	✓	✓		✓	✓				
RD	✓	✓	✓	✓	✓				
BN	✓		✓	✓		✓			
MZ	✓		✓	✓	✓				

Figure 6.1: The Living Context and its associated bipartite graph

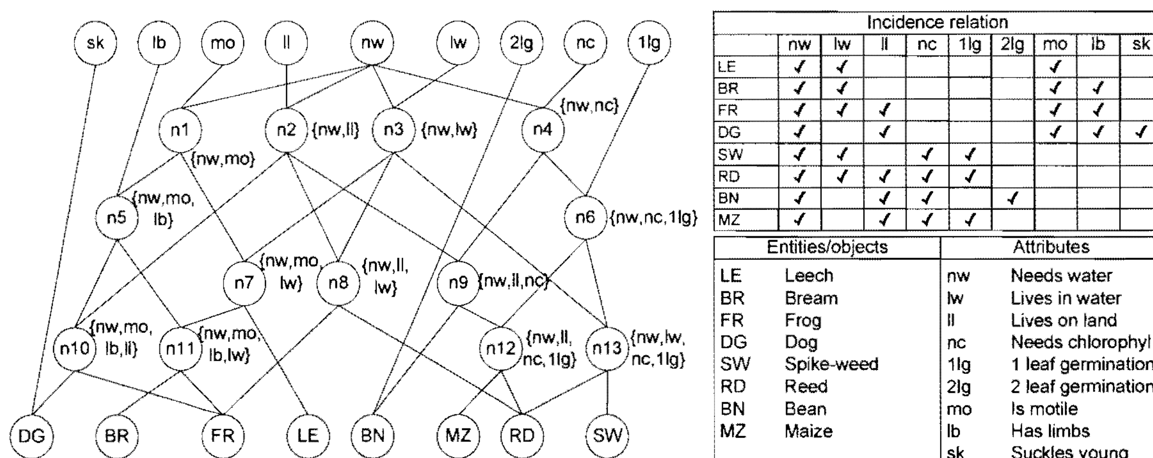
The simplest example of such a database is that of a bipartite graph (essentially representing the incidence relation of the context) with objects at the bottom, attributes at the top and arcs from each object to all the attributes it possesses in a specific context, as illustrated in figure 6.1.

Consider $O_{BP}(D, Q)$, a query operation on the bipartite database, D in figure 6.1. For a query Q we define $O_{BP}(D, Q) = Q$. As a trivial example we see that for Q = {mo, lw} (where the key of figure 6.1 indicates that mo means 'motile' and lw means 'lives in water', etc.) the query $O_{BP}(D, Q)$ would return {mo, lw}, effectively referencing the set of objects {LE, BR, FR, DG, SW, RD} (i.e. the extent of the concepts mo and lw). This can be verified by inspecting the line diagram in figure 6.1 of the database for DownwardClosure(D, mo) = {LE, BR, FR, DG} and DownwardClosure(D, lw) = {LE, BR, FR, SW, RD}.

A shortcoming of the bipartite database and of O_{BP} is the fact that the query operation returns a very general set of objects, each of which has any, but not all, of the attributes specified in the query Q. (Thus leech, bream, frogs, spike-weed and seeds are either motile and/or live in water.) In IR terms the query operation has low precision but high recall. One way of improving the precision is to introduce a new intermediate concept called 'mo_lw' that groups all objects that possesses both mo and lw into the database. This concept would be connected via upward arcs to mo and lw and all objects possessing mo and lw would have upward arcs to the new concept. In this way, a query operation might be able to use the new concept in arriving at the results of the query presumably yielding better results.

Continuing this line of thought and introducing new 'useful' or 'meaningful' intermediate concepts, the other end of the spectrum would be to use a formal concept lattice or EA-lattice as the database. Databases of this type are discussed in the next section.

6.2 A CONCEPT LATTICE DATABASE AND QUERY OPERATION



Incidence relation									
	nw	lw	ll	nc	1lg	2lg	mo	lb	sk
LE	✓	✓					✓		
BR	✓	✓					✓	✓	
FR	✓	✓	✓				✓	✓	
DG	✓		✓				✓	✓	✓
SW	✓	✓		✓	✓				
RD	✓	✓	✓	✓	✓				
BN	✓		✓	✓		✓			
MZ	✓		✓	✓	✓				

Entities/objects		Attributes	
LE	Leech	nw	Needs water
BR	Bream	lw	Lives in water
FR	Frog	ll	Lives on land
DG	Dog	nc	Needs chlorophyl
SW	Spike-weed	1lg	1 leaf germination
RD	Reed	2lg	2 leaf germination
BN	Bean	mo	Is motile
MZ	Maize	lb	Has limbs
		sk	Suckles young

Figure 6.2: The EA-lattice of the Living Context (the unit- and zero concepts are not shown)

Figure 6.2 shows an EA-lattice for the Living Context. It has clearly partitioned sets of attributes (top row of concepts), objects (bottom row) and intermediate concepts. The unit- and zero concepts of the lattice have been excluded from the database. (They are, however always implied.) The intents of some of the EA-formal concepts are shown as a guide.

$O_{Meet}(D, Q)$ is a query operation on a lattice database and is defined as the meet or infimum of the attributes of Q in the lattice (i.e. the concept ⟨Q', Q''⟩ corresponding to all the objects that have the attributes Q in common and all the attributes this set of objects have in common). For the query Q = {mo, lw} the resulting objects are therefore {BR, FR, LE} since the meet of {mo, lw} is concept n_7 which has an extent of {BR, FR, LE} (i.e. objects possessing all of the attributes in Q are returned). Note that it is now possible to obtain a result with a higher precision due to the fact that concepts lower down in the database discern between objects in a more granular way. (Thus leeches, bream and frogs are all both motile and live in water.)

6.3 AN ADAPTED SUBLATTICE DATABASE AND QUERY OPERATION

Assuming that we were looking for all the fish objects in the Living Context (in this case only BR qualifies) with query Q = {mo, lw} (i.e. all the living objects that can move and live in water). The lattice-based query operation O_{Meet} has a higher precision and recall than O_{BP}. O_{Meet} has however the disadvantage that it is not tolerant of errors or ambiguity in either the context or formulation of the query terms as indicated in the following example.

Assume, for example, that we are looking for all edible plants in the context using the query Q = {nw, nc, 1lg, 2lg}. (The key of figure 6.1 indicates that nw means 'needs water', nc means 'needs chlorophyll', 1lg means 'one leaf generation' and 2lg means 'two leaf generation', all attributes being related to edible plants.) $O_{Meet}(D, Q)$ would not return any relevant objects since the meet of Q is not in the database – the meet is in fact the zero concept, 0_L, in the lattice. In this case the query was too specific and the query operation was unable to find a concept corresponding to exactly Meet(D, Q). This is clearly not ideal since the database did contain objects relevant to Q and the query operation should ideally have coped with the situation.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

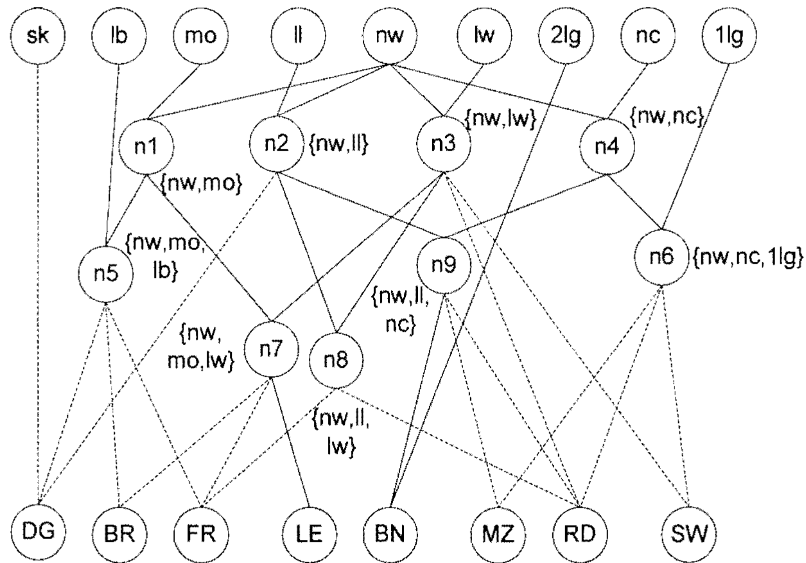


Figure 6.3: A database (compressed pseudo-lattice) with intermediate concepts with an intent that has more than three attributes removed

One strategy that could help in this case and increase the tolerance for errors is to specify a query operation for a context of n attributes that will return the minimal concepts that have at most $k \leq n$ attributes, all of which are in Q. Since the domain of discourse as defined requires that queries only be formulated in terms of concepts already contained in the database we can adopt one of two strategies. The first is to redefine the query operation to examine all concepts and return the appropriate minimal concepts. In this case, the database D is kept the same. A second option is to modify the query operation and the database. For reasons that will become clear, we will pursue the latter option.

Removing all intermediate concepts in the lattice in figure 6.2 that have more than $k = 3$ attributes creates the new database in figure 6.3. Where the original EA-lattice concepts have been removed, dashed arcs indicate successors defined by the partial ordering relation. Note that a subset, L, of the database namely all the concepts except DG, BR, FR, MZ, RD, SW and BN still forms a sublattice when the unit- and zero concepts are appropriately inserted (i.e. it forms a poset of which the supremum and infimum of all pairs of concepts exist and therefore a sublattice of the EA-lattice in figure 6.2 - this may be verified by inspection). This lattice (identified by all the concepts connected with solid arcs as well as all attribute concepts plus the implied unit- and zero concepts) does not correspond to either the formal concept lattice or EA-lattice lattice for the given context but is a sublattice of the EA-lattice of the Living Context. A query operation on the database in figure 6.3 now cannot discern between objects that have more than k attributes in common and would therefore presumably still return objects even when the query is too specific.

6.4 REMOVING CONCEPTS FROM A LATTICE

An EA-lattice as defined in chapter 2 has a fixed set of concepts for a fixed context. It is however possible to remove some of the concepts from the EA-lattice and still have a complete sublattice (albeit not a formal concept lattice or EA-lattice) that is based on the original partial ordering relationship[10]. For example, an EA-lattice can clearly be reduced

[10] Throughout this discussion, it is assumed that the same ordering relationship is used. Thus the ordering $x \leq_{EA} y$ either holds in all lattices in which concepts x and y appear, or it holds in none of them.

to a formal concept lattice by removing EA-formal concepts defined by EA-formal conditions 1 to 4 (section 2.5) if they are not generated by condition 5, but retaining EA-formal concepts defined by condition 5. Appropriately removing even more concepts will result in a set of concepts that constitute a lattice (i.e. a complete sublattice), but not a formal concept lattice (as per example in section 6.3).

In order to ensure the retention of lattice properties, when removing concepts from a lattice, steps must be taken to ensure that both the supremum and infimum of any two remaining concepts exist and are unique (as required by the definition of a lattice). Should this not be the case, the resulting set of concepts will not be a sublattice. Removing a concept from a lattice thus involves not only the removal of the concept from the lattice's set of concepts, but also the revision of the order relationship so that the parents and children of the removed concept are partially ordered as per the original $\leq_{EA}$ ordering relationship. Removing atoms or coatoms from an EA-lattice is a particular case where this is possible as formulated in the next theorem.

Theorem 6.1: Removing an atom or coatom from an EA-formal concept lattice or sublattice results in a set of remaining concepts that is a complete sublattice with respect to the partial ordering relation $\leq_{EA}$.

> **Proof:** Let L_0 be the concept lattice and L_1 the set of concepts remaining after removing an atom or coatom concept. To prove that L_1 is a lattice we need to prove that two arbitrary concepts $x, y \in L_1$ have a unique supremum and infimum in relation to $\leq_{EA}$.
>
> We first prove that x and y have a unique infimum. By implication $x, y \in L_0$. Let $q = \mathrm{Inf}(L_0, \{x, y\})$. Two alternatives exist depending on whether q is in L_1 (i.e. whether it was possibly an atom removed from L_0 or not). If q is indeed in L_1 then q is a lower bound of x and y because $q \leq_{EA} x$ and $q \leq_{EA} y$ in L_1 since q was the infimum of x and y in L_0 and the partial ordering relationship of L_1 was redefined to preserve all the transitive ordering relationships between the concepts of L_0 in L_1. Since q was also the unique greatest lower bound of x and y in L_0 and no other concepts have been added to L_1, q is therefore also the unique greatest lower bound of x and y in L_1. Thus, the infimum of x and y in L_1 exists and is unique.
>
> If q is not in L_1 then q must be an atom that was removed from L_0. (q can not be a coatom since then $x = q$ or $y = q$ in which case either x or y will not be part of L_1.) Furthermore, any possible lower bound of x and y in L_1 must be a concept that is smaller than q in L_0, since q was the unique greatest of all lower bounds of x and y in L_0. But q is an atom in L_0 and therefore it has only one child concept in L_0 namely 0_L. The 0_L is a lower bound of all concepts in L_1 and L_0 including x and y. Since q do not exits in L_1, 0_L also is the only lower bound of x and y. The infimum of x and y in L_1 is therefore unique.
>
> A similar argument can be used to prove that the supremum of x and y in L_1 is unique with q possibly being a coatom. Since the supremum and infimum of x and y in L_1 exist and are unique, L_1 is a sublattice.

Using this theorem it is therefore clear that atom or coatom concepts can be removed from a lattice (or sublattice) without violating the lattice property. Since this theorem is generic, it can be extended to apply to formal concept lattices. Removing any concept does however mean that the resulting lattice is not the EA-formal concept lattice (or formal concept lattice) of the specific context since all EA-formal concepts are not present. We follow the convention of calling these derived lattices, *sublattices* and only refer to a lattice as an EA-lattice (or formal concept lattice) when it contains all the EA-formal concepts (or

formal concepts) of the context. Although some of the attributes or objects concepts may have been removed from the lattice (but not from the context – the context remains unchanged), the atoms and coatoms (i.e. concepts covering the zero concept or covered by the unit concept, respectively) of the resulting sublattice may effectively be regarded as its new objects and attributes respectively.

Theorem 6.1 can be generalised to removing whole areas of the lattice by progressively removing one atom or coatom at a time whether it involves object concepts, attribute concepts or intermediate concepts. The atoms and coatoms of the sublattice define it sufficiently since the resulting lattice still only contains EA-formal concepts. The set of atoms and coatoms of a sublattice is called a *cut* on the EA-lattice and consists of two (possibly overlapping) sets called the *atomic cut* and the *coatomic cut.* These latter sets correspond to the set of atoms or of coatoms of the sublattice respectively and both sets are therefore anti-chains. An EA-lattice can be reduced to a given sublattice by removing all concepts that are not comparable to elements of the atomic cut or the coatomic cut as discussed below. That is, if $y \in$ atomic cut of the sublattice and $x \in$ coatomic cut of the sublattice, then the following EA-formal concepts are retained in the sublattice: all EA-formal concepts c such that $y \leq_{EA} c \leq_{EA} x$.

There are five conditions under which EA-formal concepts (excluding 1_L and 0_L) are removed to create a sublattice defined by its cut on an EA-lattice:

- A concept that is smaller than some concept in the atomic cut.
- A concept that is larger than some concepts in the coatomic cut.
- A concept that is smaller than some concept in the sublattice's coatomic cut but not comparable to any concept in the atomic cut.
- A concept that is larger than some concept in the sublattice's atomic cut but not comparable to any concept in the coatomic cut.
- A concept that is not comparable to any concept in either the sublattice's atomic cut or the sublattice's coatomic cut.

Note that the zero- and unit concepts are always part of the sublattice since only atoms and coatoms may be removed. Due to the definition of an EA-lattice, the definitions of the zero- and unit concepts however remain constant and are not dependent on the elements of the sublattice. (They are however not shown in the compressed pseudo-lattice figures below).

All concepts not in the resulting sublattice (identified by the above conditions) can be progressively removed from the original EA-lattice. All these concepts and their relations with the remaining concepts are effectively 'compressed' into either the unit concept or zero concept. In figure 6.4, the relationship of the original EA-lattice to the sublattice's cut (formed by the atoms and coatoms of the sublattice) is schematically depicted.

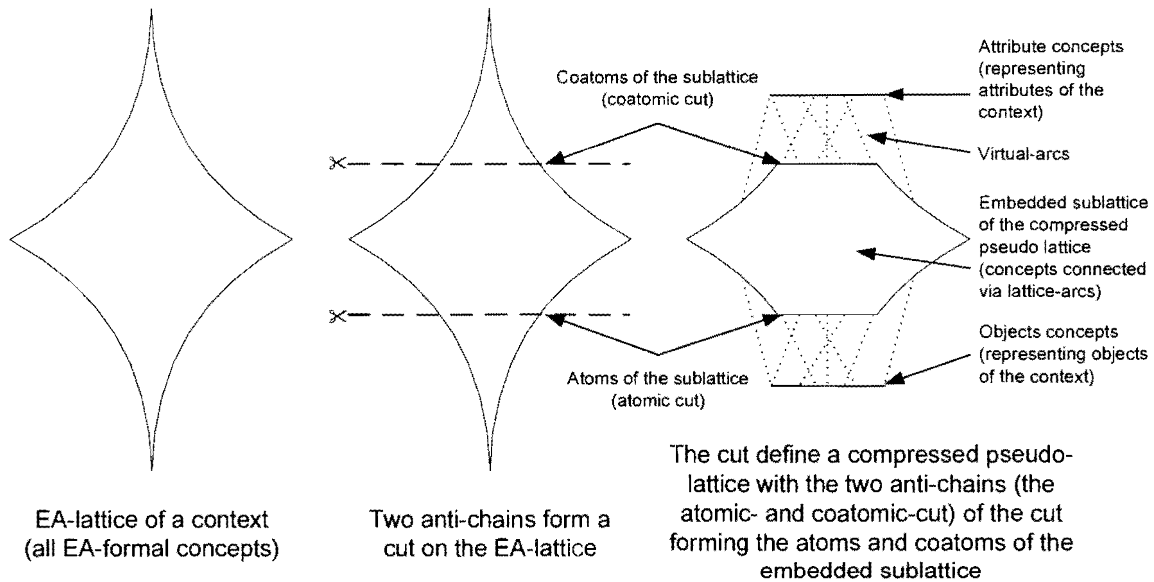


Figure 6.4: A cut on an EA-lattice defines a compressed pseudo-lattice (with an embedded sublattice) that can be created by the removal of atoms and coatoms of the EA-lattice

In the resulting compressed pseudo-lattice data structure we choose to keep the attribute and object concepts, but indicate their relationship to concepts that were previously in their upward or downward closure by using *virtual arcs.* The exact nature of these virtual arcs will be defined later.

It is important to note that, as a result of the five different conditions under which concepts can be removed from the original EA-lattice with respect to the cut, the nature of the removed concepts may be much more complex than depicted in figure 6.4. For example, instead of compressing the EA-lattice to a certain level (i.e. removing concepts with the same cardinality of the extent or intent), cuts may be defined to effectively remove entire sets of attributes, objects or areas from the EA-lattice. It is also worth noting that, in general, an arbitrary sublattice cannot be generated via compress lattice operations since removing non-atom concepts or non-coatom concepts can in certain circumstances also yield sublattices – there are thus limitations to this approach of generating sublattices.

6.5 THE USE OF THE INTENT- AND EXTENT REPRESENTATIVE OPERATIONS

Removing concepts from an EA-lattice has the effect of removing what was previously the infima and suprema (meets and joins) of certain subsets of concepts of the EA-lattice, these effectively moving to the zero- and unit concepts respectively.

Repeating the query operation O_{Meet} for $Q = \{nw, nc, 1lg, 2lg\}$ in figure 6.3, we find that there is no meet in the database (i.e. the meet is the zero concept in the sublattice – a 'trivial' meet). The intent- and extent representative operations defined in chapter 2 provides a logical solution for the problem and a revised query operation using these are therefore considered. This operation will define a 'second-order meet' in the case of a trivial meet.

Suppose a query operation, $O_{AIR}(D, Q)$, returns the approximate intent representatives (AIR) of Q in the sublattice embedded in D in figure 6.3, i.e. $O_{AIR}(D, Q) = AIR(D_{Embed}, Q)$ where D_{Embed} is the sublattice embedded in D (all concepts joined with normal arcs except those with dashed arcs such as DG, BR, FR, MZ, RD and SW). If $Q = \{nw, nc, 1lg, 2lg\}$ then S = {nw, 2lg, nc, 1lg, n_4, n_6, BN} and {n_6, BN} is the set of minimal elements of S. Thus

$O_{AIR}(D, Q)$ returns $AIR(D_{Embed}, Q) = \{n_6, BN\}$, assuming D_{Embed} is the sublattice in figure 6.3. $O_{AIR}(D, Q)$ thus references the objects {BN, MZ, RD, SW} and therefore solves the problem of a too specific query.

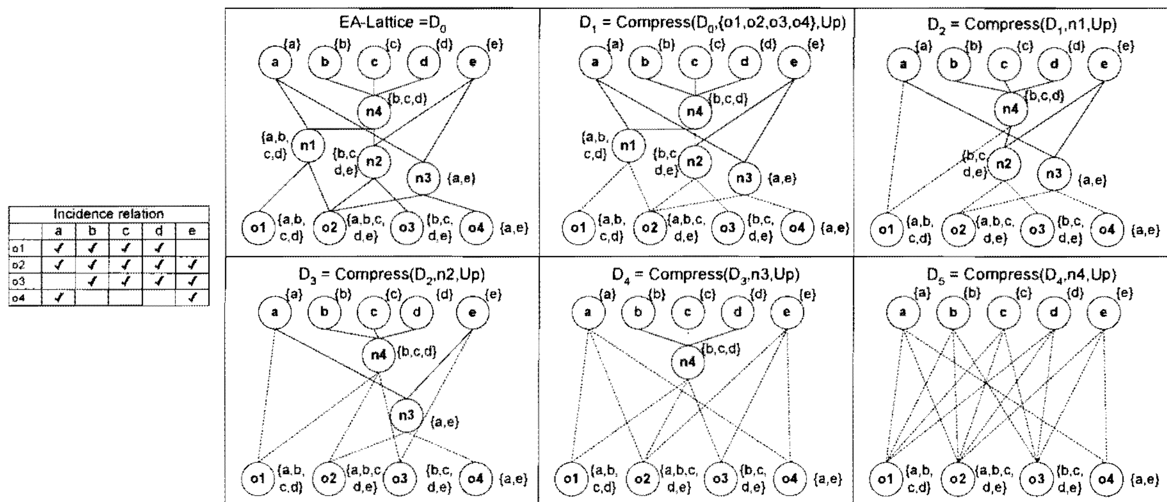
Inspecting the intents of the concepts in AIR(D, Q) we see that BN, for example, has the attribute 1l in its intent that is not in Q. If we wish to restrict a query operation to find only concepts possessing attributes in Q (i.e. exactly representing Q), then we need to use the exact intent representative operation (EIR).

Let $O_{EIR}(D, Q) = EIR(D_{Embed}, Q)$. For Q = {nw, nc, 1lg, 2lg} we saw that $AIR(L, Q) = \{n_6, BN\}$ whilst $EIR(L, Q) = \{2lg, n_6\}$ since T = {BN} in the calculation of EIR. $O_{EIR}(D, Q) = EIR(D_{Embed}, Q) = \{2lg, n_6\}$. Thus, in the present example, $O_{EIR}(D, Q)$ references the same set of objects as before, namely {BN, MZ, RD, SW}.

The O_{EIR} and O_{AIR} operations can however be applied to the database in figure 6.1, in which the embedded sublattice is reduced to only the set of attributes. In that case, $O_{BP}(D, Q) = O_{EIR}(D, Q) = O_{AIR}(D, Q)$. If D is the sublattice in figure 6.2, then $O_{Meet}(D, Q) = O_{EIR}(D, Q)$ for a non-trivial Meet(D, Q).

The point is that both the O_{AIR} and O_{EIR} operations are defined in terms of a sublattice and should the sublattice be changed (keeping the same context) as in the examples for figures 6.1 to 6.3, the representative sets also change. When Q has a non-trivial meet (i.e. not the zero concept) in the lattice or sublattice then EIR(D, Q) = AIR(D, Q) = Meet(D, Q). The representative sets of Q were defined to deal with situations when Q has a trivial meet (as is often the case when working with sublattices) and yield better results. The operations may be seen as extensions of the meet or join operations.

6.6 THE COMPRESSLATTICE OPERATION



	a	b	c	d	e
o1	✓	✓	✓	✓	
o2	✓	✓	✓	✓	✓
o3		✓	✓	✓	✓
o4	✓				✓

Figure 6.5: A CompressLattice example compressing EA-lattice D_0 to a bipartite graph, D_5

The CompressLattice operation removes an atom or coatom concept y from the sublattice embedded in the database and replaces the concept with virtual arcs (indicated as dashed arcs). The virtual arcs interconnect all the parent- with all the child concepts of y. Figure 6.5 shows an example of a compressed pseudo-lattice structure where all the intermediate concepts have been removed by successively using CompressLattice operations. Similarly, figure 6.3 can be verified to be the result of successive

CompressLattice operations on the EA-lattice in figure 6.2, removing the concepts DG, BR, FR, MZ, RD, SW, n_{10}, n_{11}, n_{12} and finally n_{13}.

It is important to note that the CompressLattice operation works from a *particular direction*. In the examples, the lattice was compressed in the *upward* direction, but the operation is equally valid when compressing the lattice from the top downward (or any combination of the two). Essentially, compression in the upward direction involves the removal of atoms, while compression in the downward direction involves the removal of coatoms from a sublattice.

The CompressLattice operation creates a data structure that is not an EA-lattice but one that does contain an embedded sublattice. This data structure is called a *compressed pseudo-lattice*. A compressed pseudo-lattice consists of EA-formal concepts interconnected by virtual- and lattice arcs. The concepts connected by lattice arcs (with 0_L and 1_L implied) define a sublattice called the *embedded sublattice* (of the compressed pseudo-lattice). The virtual arcs represent the relationship between the sublattice and the context. Using parameter names to imply types, the CompressLattice operation is defined as follows in terms of its pre- and post-conditions:

```
//=========================================================
CompressLattice(aCompressedLattice, aConcept, aDirection)
                  Return outCompressedLattice
//=========================================================
//Pre-condition: aConcept is an atom or coatom in the embedded
//sublattice in aCompressedLattice, it has at least one lattice arc in
//aDirection and no lattice arcs in the opposite direction
//(except to the unit- or zero concept).
//Post-condition: outCompressedlattice retains all the concepts
//(except possibly aConcept) and arcs of aCompressedLattice, except in
//the following respects. If aConcept is an attribute or object
//concept, then lattice arcs connecting it to other concepts in
//aCompressedLattice are replaced by virtual arcs in
//outCompressedLattice. Otherwise aConcept and its arcs are not in
//outCompressedLattice. Instead, virtual arcs link each of aConcept's
//parents to each of aConcept's children.
//=========================================================
```

Note that the definition above is in functional terms and the definition changes slightly when defined in an object-oriented fashion as discussed in chapter 7 where the references to aCompressedLattice and outCompressedLattice fall away.

6.7 DEFINITION AND PROPERTIES OF COMPRESSED PSEUDO-LATTICES

A compressed pseudo-lattice essentially represents a sublattice of an EA-lattice from which a number of atoms and/or coatoms have been removed. Additionally the relation of the sublattice to the context from which it was derived is preserved. As a data structure it represents a particular context $C = \langle O, A, I \rangle$. The data structure consists of a number of EA-formal concepts that are connected by one of two types of directed arcs: lattice arcs and virtual arcs. Lattice arcs preserve the existence suprema and infima across the concepts they interconnect; virtual arcs do not necessarily. The concepts are partitioned into three sets: the attribute concepts (of the context), the object concepts (of the context) and a number of intermediate concepts. A compressed pseudo-lattice contains an embedded sublattice. The embedded sublattice is the set of all concepts complying with one of the following:

- The concept is an attribute concept with no incoming virtual- or lattice arcs.
- The concept has at least one lattice arc connecting into or out of it.
- The unit and zero concepts.

This corresponds to the set of concepts remaining after reducing an EA-lattice to a sublattice by successive removal of atom and coatom concepts, as discussed previously. The properties of the compressed pseudo-lattice are thus implied by the CompressLattice operation. It is important to note that, for a given set of concepts, there may be more than one compressed pseudo-lattices that can be defined upon that set of concepts using different cuts on that lattice. This is due to the fact that concepts can be interconnected by either virtual- or lattice arcs. Both the concepts and arcs thus uniquely define the embedded sublattice. The context and the atoms and coatoms (i.e. the cut) of an embedded sublattice (i.e. the atomic and coatomic cut) also uniquely define a compressed pseudo-lattice.

The following are the *compressed pseudo-lattice properties*. They define sufficient conditions for a data structure to be a valid compressed pseudo-lattice. Note that the conditions listed are not disjoint – they may be related to or imply one another.

- ***EA-formal concepts:*** All concepts are EA-formal concepts as defined in chapter 2.
- ***Poset:*** Concepts in a compressed pseudo-lattice form a partially ordered set with respect to the partial ordering relation ($\leq_{EA}$) specified by the directed arcs (lattice or virtual).
- ***Object and attribute concepts:*** All objects, o_i, in the context have a corresponding unique associated object concept in the form $\langle E, E' \rangle$, $E = \{o_i\}$. Similarly, all attributes, a_i, in the context have a corresponding unique associated attribute concept in the form $\langle F', F \rangle$, $F = \{a_i\}$. All object- and attribute concepts are not necessarily in the embedded sublattice. If some are, they form the atoms and coatoms of the embedded-lattice. If they are not, they have only virtual arcs from or to other concepts.
- ***Context preservation:*** An object contains in its upward closure (following lattice or virtual arcs) all the corresponding attribute concepts specified in the incidence relation of the context, and no other attribute concepts. Similarly an attribute contains in its downward closure all its corresponding object concepts specified in the incidence realtion of the context, and no other object concepts.
- ***Unconnected object- and attribute concepts:*** An attribute concept cannot have any outgoing arcs to concepts other than the unit concept and similarly, an object concept cannot have any incoming arcs from concepts other than the zero concept. Object- and attribute concepts are therefore not represented as generalisations or specialisations of each other.
- ***Unique intermediate concepts:*** No two intermediate concepts may have the same extent or the same intent. This property (as well as the above property) implies that any intermediate concept has at least two upward and two downward arcs (virtual or lattice). This does not preclude attribute- and object concepts from having the same extent or intent respectively or sharing the same extent or intent of an intermediate concept (in such cases one of the concepts will have only one parent or child concept). Such attribute or object concepts are represented as distinct concepts in a compressed pseudo-lattice.

- ***Non-empty intent:*** No concept (other than 1_L and 0_L) may have an empty intent or extent (i.e. all objects must possess at least one attribute but some attributes may not have any object possessing the attribute). This limits the contexts for which a valid compressed pseudo-lattice may be constructed. Although the property is not strictly required, the practical benefits of contexts that do not conform to this requirement are not immediately clear. Attribute concepts may have an empty extent.

- ***Embedded sublattice:*** The set of all concepts in the embedded sublattice together with the partial ordering implied by the lattice arcs used in the embedded sublattice, constitute a sublattice when appropriately connected to the implied unit- and the zero concepts (i.e. the supremum and infimum of any pair of concets exits and are unique).

- ***Supremum and infimum:*** Any set, S, of concepts in the embedded sublattice has a supremum in the embedded sublattice itself. Similarly S has an infimum in the embedded sublattice.

- ***Intermediate virtual arcs:*** Intermediate concepts may not be connected to one another via virtual arcs. Their virtual arcs must end in an attribute concept or start at an object concept. This property is implied by the fact that only atoms and coatoms of a sublattice are removed.

- ***Exact representative connection:*** Virtual arcs in a compressed pseudo-lattice are not to arbitrary intermediate concepts and respect the EIR and EER operations. An object concept, o, is only connected via virtual or lattice arcs to EIR(L, Intent(o), o) (where L is the embedded sublattice) and no other concepts. A similar property holds for any attribute a and EER(L, {a}, a). These dual properties are critical in ensuring that the closure operations function as expected (e.g. that the downward closure of a concept contains its extent either via lattice- or virtual arcs).

- ***Arc duplication:*** A concept may only have one arc (either lattice or virtual) to any concept that covers it.

- ***Cover:*** A concept may not have an arc to any other concept to which it is indirectly linked[11].

The compresses pseudo-lattice definition and properties show that a compressed pseudo-lattice is essentially a bipartite graph (virtual arcs) that contains an embedded sublattice (lattice arcs). Furthermore, the compressed pseudo-lattice properties ensure a well-defined and unique structure for a given context and a given sequence of CompressedLattice operations. Various operations can be defined on a compressed pseudo-lattice, but the most important are:

- CompressLattice and ExpandLattice (described in the next section).

- Closure and LatticeClosure, where LatticeClosure follows only lattice arcs when discovering concepts whilst Closure follows any type of arc.

- AddAtom, i.e. insert a new object into the context and embedded sublattice by using a modified incremental lattice construction algorithm that operate under compressed pseudo-lattices.

- InsertVirtualObject, an alternative to AddAtom that does not use a computationally expensive lattice construction algorithm to update the embedded sublattice (refer

[11] Concept x is indirectly linked to concept y iff x has a path to y via one or more intermediate concepts.

to section 6.11). The object is inserted into the compressed pseudo-lattice by simply creating virtual arcs to its exact intent representatives.

6.8 THE EXPANDLATTICE OPERATION

A complementary operation to CompressLattice, namely ExpandLattice, can be defined to enlarge the embedded sublattice of a compressed pseudo-lattice by the insertion of new atoms or coatoms into the embedded sublattice. ExpandLattice essentially recreates concepts removed by CompressLattice. The operation works in a particular direction, starting with a concept that is incident to at least one virtual arc. In general, a concept may be incident to zero, one or more than one virtual arcs. Some virtual arcs may connect the concept to objects while others may connect it to attributes. When invoking the ExpandLattice operation, a 'direction' has to be specified. If the concept is an atom of the embedded sublattice and it has virtual arcs connecting to it, then the direction is designated 'downwards' and the operation will create new atoms in the lattice below the concept. Alternatively, if the concept is a coatom of the embedded sublattice and it has virtual arcs connecting to it, then the direction is designated 'upwards' and will create coatoms above the concept. If the concept is both an atom and coatom, then the direction may be specified as either downwards or upwards.

In the upward direction, starting with concept c, the ExpandLattice operation determines the minimal number of coatoms that must be inserted into the embedded sublattice to replace the virtual arcs from c with lattice arcs to these inserted concepts. Concept c is directly connected to these concepts by lattice arcs replacing c's virtual arcs. To comply with compressed pseudo-lattice properties further generation of concepts and/or creation or removal of arcs may be necessary. Similar remarks apply *pari passu* when expanding a given concept in the downward direction.

Note that the CompressLattice and ExpandLattice operations *are not symmetric* in that the one does not reverse the other. In most instances ExpandLattice does not recreate the concepts removed via a single CompressLattice operation. It is however always possible to completely compress an EA-lattice into a bipartite graph or to use ExpandLattice operations to completely rebuild the EA-lattice from a bipartite graph. Our implementation of this latter series of operations indicates that it is computationally more expensive than using a 'traditional' incremental lattice construction algorithm to construct a lattice but it still indicates the versatility of a compressed pseudo-lattice. The context preservation and exact representative connection properties of a compressed pseudo-lattice play important roles in the ability to rebuild the EA-lattice from a compressed pseudo-lattice.

ExpandLattice is defined below in terms of its pre- and post conditions. Again, parameter names imply their corresponding types.

```
//=====================================================================
Function ExpandLattice(aCompressedLattice, aConcept, aDirection)
            Return outCompressedLattice
//=====================================================================
//Pre-condition: aConcept is a concept in aCompressedLattice that has
//virtual arcs in aDirection.
//Post-condition: outCompressedlattice retains all the concepts and
//arcs of aCompressedLattice, except in the following respects. If
//aDirection is down (up), then the minimal number of new atom
//(coatom) concepts are inserted into outCompressedLattice's embedded
//sublattice to cover aConcept and replace the its virtual arcs with
//lattice arcs. Additional concepts are created and arcs are created,
//removed or relabelled if and only if necessary to maintain
//compressed lattice properties. If appropriate, object (attribute)
//concepts are reconnected to the embedded sublattice via lattice arcs.
//=====================================================================
```

As an ExpandLattice example, consider figure 6.5 with the compressed pseudo-lattices D_0 to D_5. When starting with D_5, i.e. the bipartite graph, the following order of ExpandLattice operations will reconstruct D_0: D_4 = ExpandLattice(D_5, c, Downward); D_2 = ExpandLattice(D_4, e, Downward); D_1 = ExpandLattice(D_2, a, Downward) and finally D_0 is the result of successive ExpandLattice calls that expand the concepts n_1, n_2 and n_3 in a downward direction. Note that ExpandLattice(D_4, e, Downward) does not produce D_3 because D_3 does not contain *all* the atoms created by ExpandLattice needed to replace the virtual arcs to e with lattice arcs (this is a example of CompressLattice and ExpandLattice not being symmetrical).

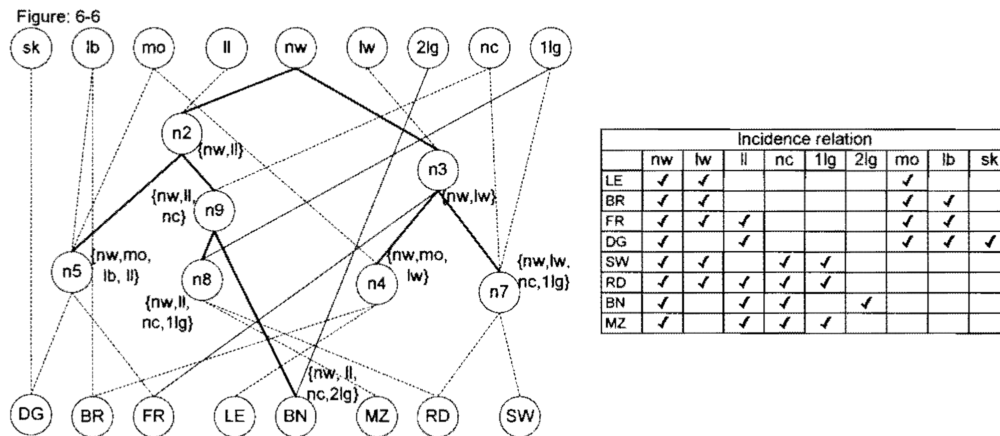
6.9 INTERPRETATION OF COMPRESSED PSEUDO-LATTICES

Since the embedded sublattice of a compressed pseudo-lattice is indeed a sublattice, the interpretation of the concepts in a compressed pseudo-lattice is analogous to that of concepts in a concept lattice. In a concept lattice, concepts are partially ordered in terms of generalisation and specialisation of their intents and extents. A parent concept, p, of a concept, c, is (in a concept lattice) the smallest concept that is more general than c and therefore moving upwards in a lattice involves the smallest increments of generalisation supported by the 'evidence' in the context. In a compressed pseudo-lattice this continues to be the case, except for the fact that concepts that were removed are not 'discovered' or visited due to being 'uninteresting', not useful or insignificant in the application context in which the compressed pseudo-lattice is being used. Algorithms based on compressed pseudo-lattices are therefore not able to discern the relationships between objects that are part of clusters referenced by removed or compressed concepts.

One may argue that this can be detrimental to such algorithms but it should be remembered that classification algorithms based on other structures such as hierarchies (for example Quinlan's (1986) ID3 decision trees and Fisher's (1987) COBWEB) do precisely this: they minimise the clusters or concepts used to describe a context often with greater classification accuracy than using more concepts. This is because at a certain point the additional resolution obtained by using more concepts is used to approximate and describe the noise inherent in the data rather than depicting the abstractions that hold in the larger population from where the data was taken. These approaches have proved successful in many areas of research, particularly in KDD and machine learning. The compressed pseudo-lattice gives the researcher the ability to apply the ideas used in other areas of research on concept lattices whilst still maintaining the benefits of the lattice properties. In essence the removal of concepts restricts the vocabulary of concepts available to KDD or machine learning algorithms in a controlled way without the loss of many desirable features of concept lattices.

One way of viewing the removal of atoms or coatoms from a concept lattice is to see the removal thereof as the change of the attributes and objects of a context. The context is redefined in terms of the new attributes and objects defined by the atoms and coatoms of the embedded sublattice. An embedded sublattice in which attributes are removed can therefore be seen as an abstraction of the context where attributes are more specialised by conjoining some of the original attributes. A removed object is 'replaced' by a more general object, this being the next more general concept in the EA-lattice. The virtual arcs in the compressed pseudo-lattice in these cases indicate and preserve the relationship between the original context and the new implied context with its specialised attributes and generalised objects.

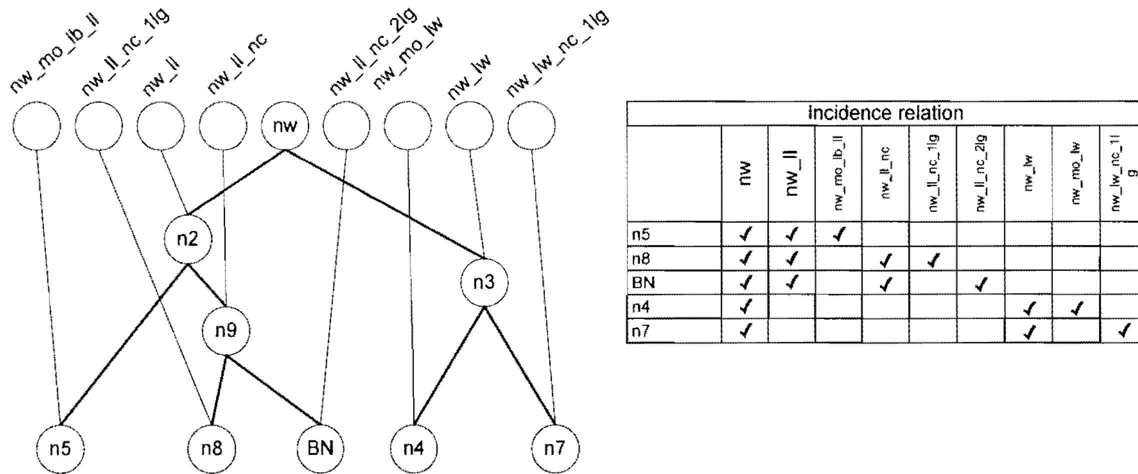
Figure 6.6 is an example in which the EA-lattice of the Living Context has been compressed (or reduced) to a hierarchy (or in ID3 terms a decision tree) that describes the context as a set of objects all of which need water. These objects are then divided into those that live on land (ll) and those that live in water (lw) and so forth. This description or classification of the context is not incorrect - it is just not the *only* description or classification of the Living Context. The use of a compressed pseudo-lattice also has the benefit that even though the embedded sublattice is essentially representing a hierarchy, the fact that there are objects that belong to more than one branch of the hierarchy is easily represented (e.g. concepts FR, RD).



Incidence relation									
	nw	lw	ll	nc	1lg	2lg	mo	lb	sk
LE	✓	✓					✓		
BR	✓	✓					✓	✓	
FR	✓	✓	✓				✓	✓	
DG	✓		✓				✓	✓	✓
SW	✓	✓		✓	✓				
RD	✓	✓	✓	✓	✓				
BN	✓		✓	✓		✓			
MZ	✓		✓	✓	✓				

Figure 6.6: A compressed pseudo-lattice structured to contain a hierarchy that implies the context and EA-lattice in figure 6.7

The embedded sublattice of a compressed pseudo-lattice implies a new context with more specialised attributes and generalised objects. Figure 6.7 shows the incidence relation of the new (implied) context as well as the EA-lattice. Note that the attributes of this new context are now conjunctions of the attributes of the previous context.



Incidence relation									
	nw	nw_ll	nw_mo_lb_ll	nw_ll_nc	nw_ll_nc_1lg	nw_ll_nc_2lg	nw_lw	nw_mo_lw	nw_lw_nc_1lg
n5	✓	✓	✓						
n8	✓	✓		✓	✓				
BN	✓	✓		✓		✓			
n4	✓						✓	✓	
n7	✓						✓		✓

Figure 6.7: The EA-lattice and context implied by the compressed pseudo-lattice in figure 6.6

6.10 WHY COMPRESSED PSEUDO-LATTICES?

'A little knowledge that acts is worth infinitely more than much knowledge that is idle'

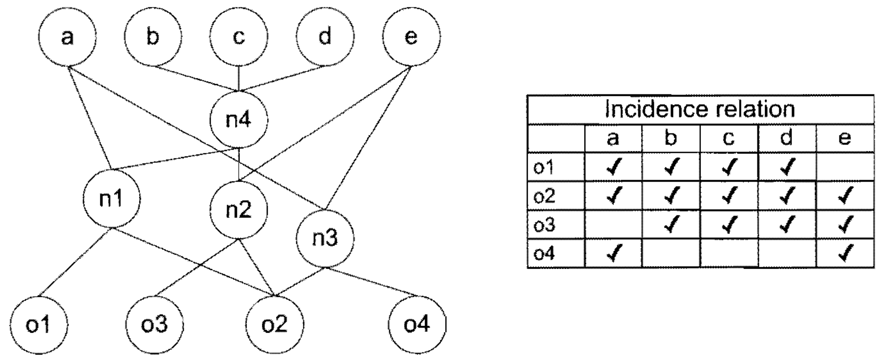
Kahlil Gibran

Key questions surrounding the use of compressed pseudo-lattices are: Why would we remove formal concepts from a formal concept lattice in the first place? Is it not better to work with all formal concepts? The answer to the question lies in the nature of the formal concept lattice: a formal concept lattice contains a concept for all possible clusters of objects supported by the incidence relation (i.e. every possible grouping of objects that have some attributes in common are represented by a formal concept). This results in a data structure that is very large and, in the worst case, exponential in size. In addition, the interpretation and use of this data may be obscured by the large amount of detail (often caused by noise in the data). Authors such as Duquenne et al. (2001) have expressed the difficulty in working with large concept lattices and have called for useful approximations of lattices. Hereth and Stumme (2001) generate Iceberg Concept Lattices in which they have purposefully removed concepts to reduce the lattice size. Iceberg Lattices are a specialisation of compressed pseudo-lattices in the sense that only atoms are removed. Mephu Nguifo (2001) also do not use the whole concept lattice in the context of machine learning. Compressed pseudo-lattices allow one to retain the benefits of a lattice, but allow for the selective and discretionary removal of concepts, thereby reducing the size of the lattice. Should it be required, the EA-lattice can be re-created using the ExpandLattice operation.

Another observation regarding the nature of a concept lattice is that some of the concepts may not, in some sense, represent 'meaningful' or 'useful' clusters of objects. An example is when attributes in a context do not imply each other[12]. For example in the Living Context, the occurrence of the attribute ll always implies the attribute nw.

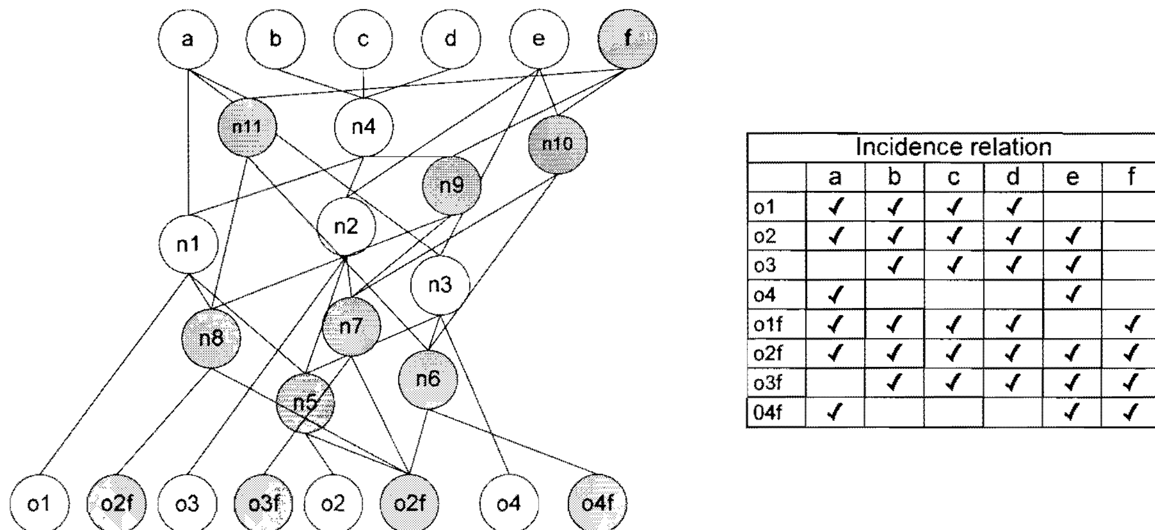
[12] An implication rule is a rule in the form $B \rightarrow C$ where B and C are sets of attributes. The support of a rule is the number of objects in a context for which this rule holds whilst the confidence of the rule is the number of times the rule holds in all objects that have B in their intent. A rule with a confidence of 100% indicates an implication rule.

As another example, consider the concept lattice for the context in figure 6.8 showing a simple context and its EA-lattice. The incidence relation of figure 6.9 is an extension of that of figure 6.8 in that the objects of figure 6.8 have been duplicated and an attribute f, which is not implied by any other attribute(s), introduced in the intent of the duplicated objects. The additional attribute, f, was added to the intent of each of the new duplicated objects so that the new context has pairs of related objects that differ only in respect of one attribute (e.g. o1f has the same intent as o1 except for the additional attribute f). Figure 6.9 shows the EA-lattice of this context. In this context, the attribute f is not implied by any other attribute since any combination of the attributes, a to e, that occurs with f in some set of objects in the context, also occurs without f in some other set of objects in the context.



Incidence relation					
	a	b	c	d	e
o1	✓	✓	✓	✓	
o2	✓	✓	✓	✓	✓
o3		✓	✓	✓	✓
o4	✓				✓

Figure 6.8: The EA-lattice of a simple context



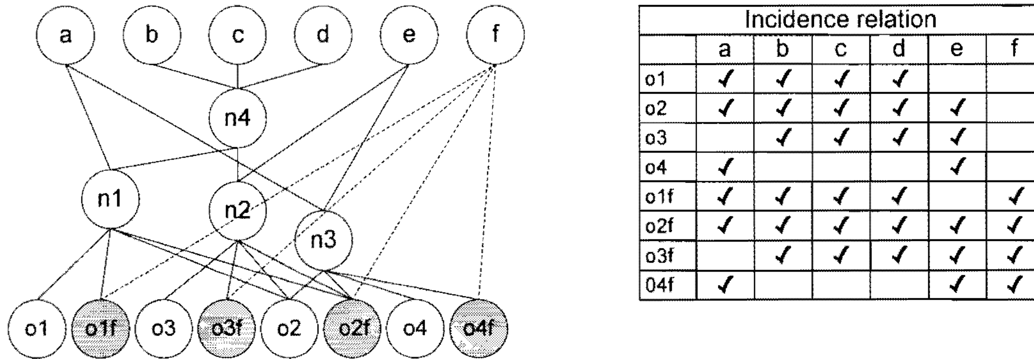
Incidence relation						
	a	b	c	d	e	f
o1	✓	✓	✓	✓		
o2	✓	✓	✓	✓	✓	
o3		✓	✓	✓	✓	
o4	✓				✓	
o1f	✓	✓	✓	✓		✓
o2f	✓	✓	✓	✓	✓	✓
o3f		✓	✓	✓	✓	✓
04f	✓				✓	✓

Figure 6.9: An attribute, f, not implied by any other attribute(s) introduced into the context of figure 6.8 creates a large number of additional concepts

The attribute f is thus not implied by any combination of the other attributes. With regard to deriving implication rules from the lattice, no implication rule based on f is possible and yet the lattice explicitly represent all possible combinations of f and the other attributes in the newly created concepts (newly created concepts are shaded). The effect of adding such an attribute to a context can clearly be seen to significantly increase the number of concepts in the lattice. This is in fact the worst-case example, where the addition of each new attribute (or object) doubles the number of concepts in the lattice. We argue that such

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

structures and attributes are not very useful in machine learning and KDD and are best not represented in the lattices used in these applications.



Incidence relation						
	a	b	c	d	e	f
o1	✓	✓	✓	✓		
o2	✓	✓	✓	✓	✓	
o3		✓	✓	✓	✓	
o4	✓				✓	
o1f	✓	✓	✓	✓		✓
o2f	✓	✓	✓	✓	✓	✓
o3f		✓	✓	✓	✓	✓
04f	✓				✓	✓

Figure 6.10: A compressed pseudo-lattice of the context in figure 6.8 in which the embedded sublattice corresponds to that of figure 6.8

Concept n_6 in the EA-lattice (figure 6.9) has no implication rule associated with it and has a very little support. It is thus doubtful whether it is useful in any KDD or machine learning effort and would therefore worth removing by first using CompressLattice to remove the objects underneath it and then the concept itself. The compressed pseudo-lattice in figure 6.10 shows how the creation of additional concepts can be avoided by using the CompressLattice operation to remove the concepts involving f (i.e. all shaded concepts in figure 6.9) – these relations are not lost and are still indicated by the virtual arcs. The embedded sublattice of this compressed pseudo-lattice corresponds to the EA-lattice in figure 6.8. The objects in the compressed pseudo-lattice therefore still have f in their upward closure. The advantage of a compressed pseudo-lattice is that even tough most of the information is not lost in figure 6.10, should it be required, the ExpandLattice operation can be used to regenerate the lattice in figure 6.9.

This example may seem artificial but in KDD and machine learning, a large number of objects are usually used to construct a lattice (e.g. a training set). If the sample is large enough, statistically most combinations of attributes that are not implied by other attribute(s) will occur in the set of objects. As a result, large number of concepts will be created and parts of the lattice will resemble a Boolean lattice. Even if there are very clear implication rules, if *only one* object does not conform to the rule (i.e. the confidence is not exactly 100% due to noise or errors in the data) its insertion into the lattice will cause the creation of all these additional concepts despite the low support for them. Put differently, only one exception to the rule will cause an otherwise implied set of attributes to loose this property, even though, statistically speaking, there is a high dependence correlation.

A typical approach in KDD and machine learning is to use the number of objects in the extent of each concept as a measure of support for the implication rules on its attributes. It is thus important to keep the objects in the data structure to calculate this measure. Since this is exactly what happens in a compressed pseudo-lattice it is ideal for this purpose.

In the field of KDD and machine learning, the ability to remove concepts from large lattices may prove beneficial in a number of respects. The reduced size of the lattice will improve the efficiency of algorithms whilst the removal of erroneous or noise-induced concepts with a small support may improve the results of such algorithms. A compressed pseudo-lattice based on a suitable compression strategy, offers researchers the ability to reduce the concepts in a lattice to those clusters that most accurately represent the context. This is done by removing clusters closer to the top of the lattice that are too general to allow meaningful classifications whilst also removing concepts that are too specific closer to the bottom of the lattice.

Previously the meet and join operation on a lattice was used to find the concept that represents objects that are relevant to a set of attributes. The introduction of the intent representative sets now allows these to be used as approximate or 'rough' meets in algorithms. This enhances the ability of algorithms to cope with noise in both the data as well as in the query itself (as indicated in the IR examples earlier in this chapter). Clearly, however, the results are dependent on the structure of the database. The purpose of KDD and machine learning in such situations is then to search for or construct a database (sublattice) that best typifies the inherent clusters, rules and implications of a context instead of generating, by brute force, all possible rules and implications that can be derived from a context. To paraphrase Einstein, the lattice should be as simple as possible but not simpler. The benefit of using intent and extent representative sets are that they are defined in terms of discrete operations and behave in a predictable fashion, whereas operations using 'fuzzy' and other approximations often result in a number of anomalies due to their inherently non-discrete nature.

It should be noted that the approaches described above are in general not appropriate for all areas where concept lattices have been used. Some authors (e.g. Wille (2001)) have described numerous case studies where the line diagrams of concept lattices aid human understanding of a context. Clearly compressed pseudo-lattices are less appropriate in these areas of application.

Compressed pseudo-lattices are however an alternate method of supporting *conceptual views* (Wille 2001, 2002). Conceptual views are formed when focussing on a particular part of the context. In this case a certain number of columns (i.e. attributes) in the cross table of the context are selected and the conceptual view is then defined as the lattice formed by only those columns. Each concept in the conceptual view is annotated with the number of objects that are in the extent of the concept, in this way a human is able to focus on a specific domain or view within the context which may assist in finding relevant information. It is easy to see that the lattice for a particular conceptual view can be easily extended using virtual arcs to form the compressed pseudo-lattice for which the conceptual view is the embedded lattice. The advantage of this approach is that it is then easier (from a data rather than a human perspective) relate the conceptual view back to the original concept. In addition, compressed pseudo-lattices are not restricted to sublattices defined by the columns of the cross table. Compressed pseudo-lattices therefore provide a more flexible but still formal way of defining conceptual views.

6.11 COMPRESSION STRATEGIES AND CRITERIA

The start of this chapter defined a very specific domain of discourse of which there are three components: the context, the database and the query operation. A compressed pseudo-lattice may serve as such a database. The separation of the database and query operation creates an interesting deviation from some traditional information retrieval approaches: the organisation of the database co-determines the outcome of the of the query operation. Given a context and a query, the result of the query depends on the compressed pseudo-lattice used to represent the context. The question that arises is thus: 'Are there databases derived from compressed pseudo-lattice databases that, on average, result in better retrieval for the same context and query operations?'

Although a full exploration of this question is beyond the scope of this text, limited experimental results to date suggest that there are indeed better methods of organisation. Specifically, it appears that a database consisting of the EA-lattice or formal concept lattice of a given context need not, in general, be the best database. In many instances significantly compressed EA-lattices performed equally or better, hinting at an amount of redundancy embedded in concept lattices. Further experimentation is required in order to

explore compression strategies and concept pruning criteria that are likely to lead to optimal performance in various contexts.

In general, there are a number of possible compression strategies that seem to deserve such exploration. The most obvious is the one stated above where a lattice is compressed up to a specific level of above a support threshold.

It is useful to have a compression strategy combined with a threshold on the embedded sublattice size. The embedded sublattice is then repeatedly compressed until the embedded sublattice size is below the threshold. This can be combined with an adapted incremental lattice construction algorithm where the pruning mechanism is invoked after each individual object or batch of objects has been inserted into the compressed pseudo-lattice. This has the added advantage of limiting the size of the lattice and therefore the time taken to build a compressed pseudo-lattice.

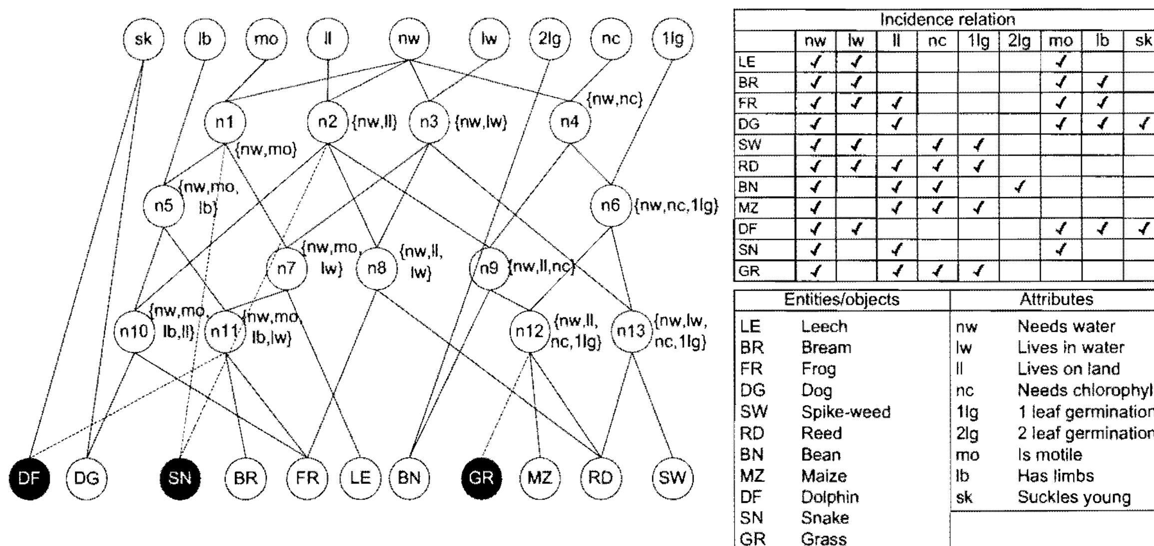
Compression strategies that have been preliminarily tested use a combination of the following:

- Compress concepts with an extent of size smaller than t and larger than u. This is a more general approach than just excluding concepts at the bottom of the lattice. This approach is taken by Hereth and Stumme (2001). They call the resulting structures iceberg lattices.
- Compression based on the number of arcs to child or parent concepts in the lattice.
- Compression based on $EP(c)$, an estimate of prior probability of the concept c. $EP(c)$ is the number of objects in the extent of c divided by the total number of objects in the context. Refer to Oosthuizen (1994b) for a discussion and examples.
- Compress, based on the difference between an estimate of the expected probability $ExP(c)$[13] and $EP(c)$. This compression strategy performed the best in most preliminary test results.

A useful variation on the insertion of an object into a lattice, that does not require the search for and insertion of the necessary new concepts into a lattice, can be defined as follows. Instead of the creation of concepts, the new object is simply connected to $EIR(L, Intent(o))$ by means of virtual arcs to create a compressed pseudo-lattice. The example in figure 6.11 shows such a compressed pseudo-lattice after the living has been extended with the objects DF, SN and GR. Even though the context and example is relatively simple, it does show that a number of operations were already avoided (e.g. the creation of an intermediate EA-formal concept $\langle \{SN\}, \{mo, nw, lw\} \rangle$). When large lattices are involved, this method of insertion saves much processing. This function is called InsertVirtualObject.

---

[13] $ExP(c) = EP(a_1) \times EP(a_2) \times \ldots \times EP(a_n)$, where a_i is an attribute in the intent of c. This estimate assumes that the attributes are independent.



Incidence relation									
	nw	lw	ll	nc	1lg	2lg	mo	lb	sk
LE	✓	✓					✓		
BR	✓	✓					✓	✓	
FR	✓	✓	✓				✓	✓	
DG	✓		✓				✓	✓	✓
SW	✓	✓		✓	✓				
RD	✓	✓	✓	✓	✓				
BN	✓		✓	✓		✓			
MZ	✓		✓	✓	✓				
DF	✓	✓					✓	✓	✓
SN	✓		✓				✓		
GR	✓		✓	✓	✓				

Entities/objects		Attributes	
LE	Leech	nw	Needs water
BR	Bream	lw	Lives in water
FR	Frog	ll	Lives on land
DG	Dog	nc	Needs chlorophyl
SW	Spike-weed	1lg	1 leaf germination
RD	Reed	2lg	2 leaf germination
BN	Bean	mo	Is motile
MZ	Maize	lb	Has limbs
DF	Dolphin	sk	Suckles young
SN	Snake		
GR	Grass		

Figure 6.11: A compressed pseudo-lattice after the living has been extended with the objects DF, SN and GR by connecting the new objects via virtual arcs to $EIR(L, Intent(o))$ using the $InsertVirtualObject$ function

In this manner large numbers of objects can be inserted into a concept sublattice for the purposes of KDD or machine learning without incurring the potentially exponential time complexity of the creation of the EA-lattice. Since the support of each concept in both the EA-lattice and the compressed pseudo-lattice is exactly the same, algorithms using these statistical metrics will operate correctly on these lattices.

A related strategy is to insert an object into the lattice by the compressed pseudo-lattice adapted AddAtom algorithm. Upon reaching a threshold on the lattice size, the lattice is compressed to a size below this threshold using a suitable compression strategy and the CompressLattice function. This procedure is repeated for all inserted objects. Since the size of the lattice can be controlled, a compressed pseudo-lattice for a large number of objects can be efficiently constructed. For an appropriate compression strategy this could potentially prevent the degradation of the performance of KDD or machine learning algorithms if it is assumed that 'uninteresting' or 'unnecessary' concepts are not represented in the resulting lattice by being compressed. Note that in the process new concepts would continuously be created by AddAtom. CompressLattice would remove concepts but not necessarily the recently created ones if the compression strategy rates the latter more important or 'meaningful'. In this manner the lattice size remains under control. Experiments have shown that given sufficient data and an appropriate compression strategy, the structure of the embedded lattice stabilise. Different data sets from the same universe with objects presented in different sequences also resulted in substantially similar embedded lattices hinting to the fact that this approach avoids the problems created by hill-climbing searches that are very sensitive to the order in which training sets are presented to these algorithms. This is a topic for further research.

6.12 IMPLEMENTATION AND DISCUSSION OF PRELIMINARY RESULTS

The potential gain in computational efficiency of having a compressed pseudo-lattice should be weighed against the advantages of having a larger and more complete set of concepts available in a particular domain. Preliminary results however suggest that a compressed pseudo-lattice may be a useful generic data structure for various IR (information retrieval) and machine learning problem domains.

Chapter 7: AddAtom implementation

The AddAtom algorithm evolved over a number of years and eventually led to the notion of a compressed pseudo-lattice. The author was responsible for modifying and maintaining the implementation environment of the Grand algorithm of Oosthuizen (1991). Originally, most of the utility data structures and supporting code for both AddAtom and compressed pseudo-lattice implementation was derived from the Grand algorithm's implementation environment but over time this was completely redeveloped since Grand and AddAtom use completely different strategies. The code was initially developed in C but was later completely re-written in an object-oriented fashion in C++. The most important C++ classes used in the implementation are the set and lattice classes respectively (sections 7.2 and 7.3). In addition to the basic functionality of lattice construction, a number of other functionalities, such as the caching of the lattice structure to disk, were also implemented.

This chapter outlines the class functions; the tradeoffs made when implementing the algorithm; as well as important features of the code such as memory optimisation, caching of results, etc. It ends with the discussion of several other implementation issues that were addressed.

Note that the code used for the wider performance comparisons in chapter 5 is not discussed here. The AddAtom implementation for that particular exercise relied on the pre-existing code base of S. Obiedkov that had been used for prior comparative studies of lattice construction algorithms (Kuznetsov and Obiedkov 2001 and 2002). Also note that the algorithms for the pilot and wide experimental studies in chapter 5 were different. The pilot study implementation used the additional optimisations of appendix A whereas the wide study used the basic algorithm of section 4.6.

An object-oriented notation is used for functions and not the functional notation used in previous chapters.

7.1 EVOLUTION OF CODE

The code from which the most recent implemented version of AddAtom emerged, has evolved over a long period of time, from 1995 to 2001. The coded was used by various researchers in areas of related to lattice construction, machine learning and machine translation. Various additions and extensions were implemented during this period of time. Three major versions of the code were produced:

Version 1: The original Grand algorithm was described in Oostuizen (1991). This algorithm used heuristics to construct a lattice. The algorithm itself does not have much in common with the AddAtom since the Grand algorithm used a number of heuristics and consisted of a procedure that connected an object concept to each of the attribute concepts in its intent, one attribute at a time, whereas AddAtom considers all attributes of the object at once. A function called transform was used to ensure the integrity of the lattice in having unique suprema and infima. Grand was implemented in C. This code was maintained by the author and adapted for use in machine learning and other experiments.

Version 2: The AddAtom algorithm was implemented in C using the Grand data structures, but with all the Grand lattice construction code completely replaced by the

AddAtom algorithm. The primary reasons for this was to modularise and clean up the version 1 code. In addition, the compressed pseudo-lattice data structures and functions such as virtual arcs, CompressLattice and ExpandLattice were also implemented. This all extensions to the code (relative to version 1) were completely written the author.

Version 3: The object-oriented version of the version 2 code. All data structures were completely rewritten and the option of optimising whether on time or space complexity depending on the application. In order to provide the maximum flexibility a significant amount of modularisation and encapsulation was used in the class design. Version 3 was implemented in C++ by the author. A number of different compressed pseudo-lattice operations such as ExpandLattice were also improved. The primary reason for this rewrite was to obtain flexibility to use the code in many research projects.

7.2 SET CLASS

The most important class of the implementation (other than the lattice class itself) is the set class. It contains implementations set operations such as union, intersection, difference and set complement on sets of integers. These are the basic operations in the implementation of AddAtom and other lattice related algorithms.

Each concept in the lattice is identified by a unique integer. A set may contain any number of concepts. In addition, representation of infinite sets is also supported. (An infinite set would be required to represent, for example, the complement of an empty set.)

The set class represents the set as a string of bits and set operations such as union (or) and intersection (and) can therefore be very efficiently calculated using normal bitwise processor operations that operate on 32 or 64 bits at a time.

The most important methods of the set class are (with parameter names implying types):

- Initialise(anInitValue)
- And(aSet) Return aSet
- Or(aSet) Return aSet
- Not() Return aSet
- FirstElement() Return anElement
- NextElement(anElement) Return anElement
- AddElement(anElement)
- RemoveElement(anElement)
- TestIfSetContains(anElement) Return aBoolean

7.3 LATTICE CLASS

In the lattice class, the lattice is represented by a list of nodes. Each node is numbered and this number serves as the index to that node. Each node has a number of attributes such as its name and support (i.e. the number of concepts in its extent). In addition, each node has two sets that contain references to its parent and child nodes. This represents

the arcs associated with the node. In the case of compressed pseudo-lattices each node has two additional sets that contain its virtual parent and child nodes.

In principle, a lattice can be represented as a set of sets (i.e. a set of nodes where each node is a set of attributes representing the associated concept's intent) without explicitly representing the cover relationship (parents and children). Pragmatically, however, the explicit representation of the cover relationship is irnportant for the efficiency of the AddAtom lattice construction algorithm. Without it, cover relationships have to be rediscovered/inferred each time the lattice is traversed.

Additionally, and also for efficiency reasons, the intent and extent sets of a node are stored explicitly with each node, despite the fact that these sets can be derived from its upward closure. The decision to explicitly store these sets represents a trade-off between space and time efficiency: to save on time, the sets are precomputed and stored, costing space. This developed into a lattice operation cache. The purpose of the cache was to keep the results of lattice closure operations in memory or on disk and avoid. This cache of pre-computed values significantly improved the performance of the lattice construction algorithms as well as the browsing and traversing thereof.

The upward- and downward closure operations are important lattice operations. They return all the nodes above or below a node, respectively, and include the node itself. In addition to these standard closure operations, a number of variations were also implemented. These involve, for example, returning nodes encountered in a given direction (upwards or downwards) when:

- Following a maximum number of successive arcs.
- Following only lattice arcs in a compressed pseudo-lattice.
- Following only virtual arcs in a compressed pseudo-lattice.

The AddAtom function in the lattice class inserts a new object into the lattice. It implements the AddAtom lattice construction algorithm described in chapter 4. It inserts the object as a new node, creating and connecting the new intermediate nodes that are required to retain lattice properties.

Due to the symmetry of lattices, all functions have duals that operate in the opposite direction (e.g. UpwardClosure and DownwardClosure; AddAtom and AddCoAtom). Rather than having separate functions for each of these, an additional parameter (aDirection) was added to each of the methods. This parameter identifies the direction of the operation (i.e. either upwards or downwards).

A number of the functions have been overloaded so that the functions can be called either with a single concept as parameter or with a set of concepts (e.g. instead of Closure returning only the upward closure of a single node, it can return the union of the upward closures of a set of nodes).

The following Lattice class methods were implemented:

- Closure(aNode, aMaximumLevel, aDirection) Returns aSet
- UpwardClosure(aNode) Returns aSet
- DownwardClosure(aNode) Returns aSet
- EIR(anAttrSet, aNode, aDirection) Return aSet
- AIR(anAttrSet, aNode, aDirection) Return aSet

- AddAtom(aAttSet, anObject, RootNode, aDirection) Return aNode
- AddCoatom(aObjSet, anAttr, RootNode, aDirection) Return aNode
- CreateNewNode(aNodeType) Return aNode
- Link(aNode1, aNode2, aDirection)
- DeLink(aNode1, aNode2, aDirection)
- GetNodeIntent(aNode1, aDirection) Return aSet
- GetNodeExtent(aNode1, aDirection) Return aSet
- GetNodeType(aNode1, aDirection) Return aNodeType (e.g. returns all attributes is aDirection = upward)
- GetAllAttributes(aDirection) Return aSet
- GetAllObjects(aDirection) Return aSet
- Join(Attrs, aDirection) Return aNode
- Meet(Attrs, aDirection) Return aNode
- GetMinimalConcepts(aSet, aDirection) Return aSet

The PersistantLattice class was used to implement functionality such as persistency and the caching of concepts and closures. Over the period of time the code was developed a number of different lattice and other classes with various functionality and optimisations were developed. Although this approach was beneficial for experimentation, it did negatively affect the performance of the implementation. This was due to the increased number of function calls which required a significant amount of parameter passing and range checking.

Various additional utility methods were also implemented to perform a number of commonly used basic functions. An exhaustive enumeration of these methods is not appropriate here.

7.4 COMPRESSED PSEUDO-LATTICE IMPLEMENTATION

Due to the similarity between compressed pseudo-lattices and EA-lattices, a single class was used for the implementation of both. As was mentioned, this required the modification of methods such as AddAtom, Closure, EIR, Link etc. to be able to cope with a data structure in which both lattice- and virtual arcs can occur. In many of the lattice class methods mentioned above, this required yet another parameter to indicate whether the relevant operation (for example a closure operation) should follow lattice-, virtual- or both types of arcs.

The following compressed pseudo-lattice related methods were added to the lattice class:

- Closure(aNode, aMaximumLevel, FollowVirtualArcs, aDirection) Returns aSet
- CompressLattice(aNode, aDirection) Return aSuccessIndicator
- ExpandLattice(aNode, aDirection) Return aSuccessIndicator
- InsertVirtualObject(anAttrSet, anObject, aDirection)

- Link(aNode1, aNode2, CreateVirtualArc, aDirection)
- DeLink(aNode1, aNode2, CreateVirtualArc, aDirection)

The extension of the AddAtom algorithm to operate on compressed pseudo-lattices greatly increased its complexity since a large number of exceptions that would not occur in EA-lattices had to be tested and catered for (e.g. the removal of virtual arcs when adding concepts at the top or bottom of the embedded lattice and maintenance of these arcs).

As was mentioned earlier, the ExpandLattice algorithm can be used as a non-incremental lattice construction algorithm that is an alternative to the AddAtom algorithm for constructing lattices. The basic approach would be to start with a fully compressed pseudo-lattice such as the bipartite graph in figure 6.1 and using successive ExpandLattice calls, construct the complete lattice. However, this strategy proved to be very inefficient and was ruled out as a useful lattice construction algorithm. Nevertheless, ExpandLattice is a useful operation in manipulating and forming compressed pseudo-lattices.

7.5 IMPLEMENTATION ISSUES

The most important hurdles encountered during the implementation and testing of the code stem from the exponential nature of a lattice as a data structure. This creates problems both in terms of the time and memory size needed to build and represent the lattice. A number of trade-offs thus presented themselves and had to be dealt with. Less efficient implementations used too many resources or took a long time to execute and test.

7.5.1 Time

To build a data structure that has an exponential number of elements, obviously and unavoidably takes an exponential amount time to complete. However, even in this context, inefficient coding can lead to even worse time inefficiencies than is necessary.

A number of different implementation options were evaluated by implementing each as a different class and comparing the classes using test data sets. These options relate to the use of different internal data structures such as hash tables combined with lists instead of unordered sets. Different optimisations of the AddAtom algorithm as well as the calculation of meets and generator concepts were also considered. A number of ways to prevent the unnecessary consideration of concepts were also investigated. The algorithm in appendix A documents the algorithm with the best time performance. The strategy of caching closures of nodes was also implemented and did also improve the performance of the algorithms.

7.5.2 Space / memory

The amount of memory available for data structures is an important limitation and will always remain potentially problematic, since the worst-case number of lattice nodes is exponential in the number of attributes. The problem is thus no less acute now than when the first implementation of the AddAtom algorithm was developed (i.e. in 1996) even though the average PC of today could probably handle data sets were problematic at the time. To ameliorate the inherent problem of exponential space requirements, inefficient memory use should therefore be avoided wherever possible.

Since the algorithm is generic, any number of objects or attributes can, in principle, be added at any time. Thus, the maximum size of arrays and sets needed for a particular lattice cannot be reliably calculated beforehand. The maximum size of many of the sets is determined by the number of intermediate nodes, which is in turn determined by the data. It is a catch-22 situation in that to determine the limits, the lattice must first be built.

Two alternative approaches were taken to address this problem. The first approach declared a fixed amount of memory for each node and set. This strategy is very memory-inefficient since the maximum size required is only needed in a limited number of instances. The second approach was to use variable length sets and lists. The first approach had the benefit of not requiring the reorganisation and additional testing required for variable length sets and lists used in the second approach and was therefore very time efficient. Using this approach with a data structure that is already exponential in size resulted in reaching the practical limit of memory very quickly on the smaller PC based platforms.

The second approach is beneficial in terms of memory usage, but the increase in computing overheads to manage the data structure slowed the performance down.

In the light of this time-space trade-off it was opted to retain both of the approaches as alternative lattice classes. The appropriate class were then chosen based on the availability of memory and computing time and power in the particular application. Typically the fastest approach possible was used when building lattices (usually on UNIX servers with large memories), whilst browsing lattices on smaller systems used a more memory efficient approach.

Since many of the larger lattices built for testing could not be used in PCs running under Windows a persistent data structure and cache was implemented. This has the capability of keeping only a number of nodes in a memory cache and swapping them from disk as needed. This slowed down the performance considerably, but had the benefit that very large lattices could be traversed in a machine with a relatively small amount of memory. This cache was further extended to cache the upward and downward closures of concepts as they were calculated. This (pre-computed) closure cache improved the time efficiency of the AddAtom algorithm.

7.5.3 Object-oriented implementation

A significant decrease in performance was noted when changing form C to object-oriented and more modular C++ code (estimated at more than 30%). This was due to the increased modularisation and looser coupling of different data structures and objects as well as to the significantly increased number of function calls and parameter range checking. This lead to an increase in the number of function calls to access variables and data structures via formalised interfaces. It did have the benefit of making the experimentation with different memory, caching and optimisation strategies much easier since only small parts of the code needed to be changed. This was however at the expense of efficiency (also refer to section 5.3).

7.5.4 UNIX and Windows

Testing and implementation was conducted on both Windows-based operating systems as well as various flavours of UNIX. In order to minimise the problems of porting code between operating systems, the basic program used a text-based interface. A small number of operating system and hardware specific issues were dealt with using conditional defines in the code. This enabled the code to be compiled under both

operating systems with no modification and allowed development under Windows whilst large test runs were performed under UNIX.

As output, the program created lattice files (with a *.lat extension). These files were created in an operating system agnostic fashion which enabled the transferral between operating systems.

For the purpose of constructing GUI-based interfaces, a Windows DLL library was created that exposed all the lattice object interfaces to any Windows-based application.

7.5.5 Testing

The AddAtom algorithm in its present form was not discovered immediately and came from an iterative process during which a number of refinements and optimisations have been made. During this process an intensive testing procedure was used to determine whether each resulting data structure was indeed a lattice, and to test the various compressed lattice properties.

Some of the tests used were:

- Validating the lattice property by brute force (i.e. testing that any two nodes have only one supremum and infimum).
- Concepts labelled as attributes have no parents and object concepts must have no children whilst intermediate concepts must have both (the unit and empty concepts are not stored as part of the data structure).
- Objects are connected to the correct attributes after the building of the lattice and have not gained or lost any attributes in their intent.
- All node support values are correct.
- No intermediate node has only one parent/child node (EA-lattice property).
- Lattices built on the same context, but with objects in a different order resulted in the equivalent lattices (i.e. the two resulting lattices were isomorphic).
- No redundant links to nodes exist (i.e. there is not an direct and indirect arc between two nodes).
- All intermediate concepts have at least two parents and two children (EA-lattice property).
- The attributes and objects of a concept is the same as all the attributes in its upward and downward closures.
- Any concept c was connected only to concepts in EIR(L, Intent(c), c) and EER(L, Intent(c), c).

Using this intensive testing the correctness of the AddAtom algorithm and its variants were proved empirically whilst problems were also identified early.

In addition, the pseudo code of the (inefficient) AddAtom algorithm as described in section 4.6 was re-implemented from scratch to verify that no mistakes had been made in its formulation.

7.5.6 User interface

Due to the requirement of operating under both Windows, MS DOS and UNIX operating systems, a text based user interface was developed that provided basic functionality.

Some of the user interface functions are:

- Building lattices from a given incidence relation provided as a text-based file.
- Saving and restoring lattices to binary *.lat files.
- Performing single lattice operations such as EIR, AIR, AddAtom, CompressLattice and ExpandLattice.
- Performing specific validity and consistency tests such as the comparison of different lattices.
- Obtaining performance metrics such as the time taken for an operation, the number of certain basic lattice operations such as closures, concept references and set operations performed to construct a lattice.
- Interactively interrogating the lattice structure for debugging and other purposes.
- Producing text based files suitable for human reading that describe the lattice concepts and cover relationships (this was used for debugging purposes).

In addition the user interface allowed text-based script files to be executed that automate the execution of a number of the above functions. This was used for testing, debugging and performance measurement purposes.

7.5.7 Continued advances in hardware

As indicated, the majority of the code was written during 1995-1996 at a time when the average server and personal computer hardware available was significantly slower and had significantly less memory than at present, especially on Windows/Intel based platforms. This factor was therefore a primary driver in the development of more advanced code such as persistency, caching, variable length data structures etc. to cope with the restrictions.

7.6 COMPARISON WITH OTHER LATTICE CONSTRUCTION ALGORITHMS

For comparative purposes, both Godin et al. (1995b) and Carpineto and Romano (1993) algorithms were implemented using the object oriented version 3 data structures, utility functions and virtual classes. However, to ensure unbiased comparison with the newly derived algorithm, certain changes in the data structures were necessary. This was because both of these algorithms made extensive use of a node's intent which was not, at the time, explicitly stored as an attribute of a node. The data structures and utility operations were changed to be fair to both the algorithms before any time complexity comparisons were made. The results of the comparisons mentioned in chapter 5 are therefore on an "apples-with-apples" basis.

As was indicated in chapter 5 the results of the pilot study were confirmed by the wider study which used completely different and separate implementations.

Chapter 8: Summary and future work

In this dissertation efficiency problems associated with lattice-based approaches have been ameliorated by two strategies: developing faster algorithms and using less concepts in a lattice. The AddAtom algorithm was defined and shown to be a fast lattice construction algorithm whereas the compressed pseudo-lattice data structure that was introduced, support lattices with fewer concepts. The two approaches to more efficient deployment of lattices are complementary.

The AddAtom algorithm efficiently constructs lattices using a tightly focussed search for generator concepts. This search is performed through the use of intent- and extent representative operations. The algorithmic performance of AddAtom is very good both from a theoretical- and an experimental point of view. A worst-case performance bound is $O(\|L\|.\|O\|^2.\max(\|O'\|))$. In experimental comparisons on artificial contexts AddAtom was the best performer in all contexts except those with very high densities or very low densities in which cases it was the second best performer. It was also the best performing incremental algorithm. This indicates that the theoretical complexity bound as stated is not very sharp. In natural contexts the performance advantage of AddAtom was even more pronounced. Initial results suggests that AddAtom has the added advantage of having a relatively tight performance range over contexts of different densities whereas the performance other algorithms that offer good performance differ more significantly over different density contexts.

The compressed pseudo-lattice data structure that was defined is closely related to the line diagram of a lattice and its use as a computational tool in applications such as machine learning, information retrieval and knowledge discovery in databases is discussed. The data structure, essentially a bipartite graph that incorporates an embedded sublattice, combines desirable features of concept lattices in a structure that allows for a flexible mechanism of scaling the size of the embedded sublattice. The scaling is done using defined operations that compress and expand it by removing or adding atoms and coatoms. A compressed pseudo-lattice essentially represents a complete sublattice from which a number of atoms and/or coatoms have been removed. Additionally the relation of the sublattice to the context from which it was derived is preserved. An application-dependent compression strategy or criterion is required to guide this process. It was argued that the removal of concepts from a concept lattice may hold advantages over traditional approaches. Compressed pseudo-lattice shows promise in many field of research due to its close resemblance to that of a formal concept lattice.

The intent- and extent representative operations of a lattice were defined as substitutes for the infimum and supremum operations in compresses pseudo-lattices since the removal of concepts leads to trivial infima and suprema. In both of these areas the notion of the intent- and extent representative operations were shown to be defining in nature. AddAtom uses it to search for generator concepts and, in essence, it repeatedly insert concepts into the lattice in order that $AIR = EIR$.

8.1 POSITIONING AND RELATED RESEARCH

The theoretical experimental comparison in chapter 5 included many of the well-known lattice construction algorithms. The theoretical complexity of the Nourine and Raynaud

(2002) algorithm has the best theoretical complexity of $O((||O|| + ||A||).||O||.||L||)$. Although the cubic nature of the AddAtom theoretical complexity is higher than the quadratic nature of that of Nourine, it is argued that the AddAtom theoretical performance bound is not very sharp – this is confirmed by the experimental results.

The work of Kuznetsov and Obiedkov (2002) indicates that there is no single best algorithm for constructing lattices. This is supported by the findings of chapter 5. A hybrid approach that uses a number of criteria, such a context density, to select an algorithm that would be best to construct the concept lattice is proposed.

Because of the very large data structures associated with lattices the interpretation and use of this data may be obscured by the large amount of detail (often caused by noise in the data). Authors such as Duquenne et al. (2001) have expressed the difficulty in working with large concept lattices and have called for useful approximations of lattices. The approach taken with compressed pseudo-lattices is however not the only approach. A number of alternative approaches for dealing with large lattices have also been proposed:

- Wille (2002) proposes conceptual views that are built using human assistance. Each view represents a small part of the lattice. Since these views are defined by a subset of attributes from the context, they can easily be structured as lattices themselves. This approach supports the idea of browsing a larger lattice where a user can select a conceptual view which is “zoomed” into.
- Hereth and Stumme (2001) generate Iceberg Concept Lattices in which they have purposefully removed nodes to reduce the lattice size. Iceberg Lattices are a specialisation of compressed pseudo-lattices in the sense that a particular compressions strategy is used.
- Pernelle et al. (2002) uses a partial order called nesting. A nested concept lattice is obtained by reducing (through projections) the original lattice. As a consequence it makes the equivalence relation defined on the extents and intents of concepts coarser.
- Godin and Missaoui (1994), proposed ways of reducing concepts in a lattice called a pruned concept lattice. In general, a compressed pseudo-lattice is not directly comparable to a pruned concept lattice.
- Mephu Nguifo (2001) use flexible concept lattices that also do not use the whole concept lattice
- Alternative ideas of reducing concepts are also discussed in (Oosthuizen 1994b).

In general, the first three of these approaches can be supported by compressed pseudo-lattices since they rely on sublattices.

Other approaches focus on reducing or filtering the input context, either in terms of attributes, objects or both such as commonly used in knowledge discovery in databases and information retrieval (e.g. controlled document indexing in Salton (1989), explanation based learning in Oosthuizen (1994b) and Oosthuizen and Avenant (1992)) may also be used to avoid contexts that contain irrelevant attributes and/or erroneous objects which may lead to less effective concept lattice based approaches.

In most instances, FCA-based approaches to problem solving, such as those mentioned in chapter 1, have competing non-lattice based techniques which do not suffer to the same extent from the complexity and size issues as FCA approaches. The future success of FCA-based approaches will thus depend on either having superior predictive or classification performance that outweighs possible time performance issues. Alternatively, approaches resulting in reduced lattice sizes may result in superior time performance.

8.2 FURTHER WORK

Many of the concepts put forward in this dissertation, especially those cantered on the use of compressed pseudo-lattices require further investigation. Below are listed some possible areas for further study related to both AddAtom and compressed pseudo-lattices.

Further work related to AddAtom:

- Further experimental comparison of the AddAtom performance with that of other algorithms not included in this study should be conducted.
- AddAtom can easily be extended to operate on sub-lattices such as those in compressed pseudo-lattices. As was stated in chapter 7, a version of AddAtom with this capability was implementation, but the pseudo-code of the implementation has not been fully documented.
- The experimental comparison in chapter 5 suggests that the performance gap of AddAtom in relation to other algorithms may be the most significant in natural data sets. A wider study is required to support and generalise this observation. Specifically, the extent to which the algorithmic performance of most algorithms running natural data may differ from their performance in terms of artificial data set should be investigated.
- Hybrid approaches combining construction algorithms should be explored whereby various criteria are used to predict a construction algorithm that is most likely to be the best performer. This may even involve a per-object based decision, relying on various incremental lattice construction algorithms to insert objects. Optimisations such as the use of AddCoatom may also be considered.
- As indicated in chapter 7, the developed code is not as efficient as it might be and introduces too many overheads. As a result, there is the need to re-implement and fine-tune the code.

Further work related to compressed pseudo-lattices:

Since the compressed pseudo-lattice is a generic data structure that in essence still uses a lattice (albeit a sublattice), it lends itself to most approaches that rely on FCA. There are, however, a number of areas of research and key research questions that seems most promising. These are listed below.

- In what areas of application are compressed pseudo-lattices beneficial? Specifically, how do compressed pseudo-lattices (and the intent- and extent representative operations) perform in comparison with a formal concept lattice (with the meet and join operations) in areas where the latter has proven successful?
- What compression strategies and criteria should be used and in which areas of application? Specifically, is there a universal compression strategy applicable to many areas of application or are useful compression strategies domain specific?
- What is the relationship of a compressed pseudo-lattice and associated operations to other fields of research in databases, rough sets, etc., given its apparent ability to deal with ambiguity?
- How do various supervised machine learning algorithms perform using compressed pseudo-lattices based on various compression strategies? Here the approach and classifiers proposed in Xie et al. (2002) may be a useful start.

- What is the relationship between the performance gained when fewer concepts are processed, the predictive accuracy of algorithms and the size of the compressed pseudo-lattice?
- To what extent and in what ways may compressed pseudo-lattices be used to support information retrieval?
- A further exploration of the theoretical aspects associated with sublattices would seem to be required.
- A more complete comparison is required of compressed pseudo-lattices with other methods that use sublattices and lattices with a reduced number of concepts.
- There is a need to investigate how compressed pseudo-lattices may be combined with other techniques and approaches.

The research into there and other related issues are on-going.

--------------------oOo--------------------

References

Barbut, M., and Monjardet, B., 1970, Ordre et Classification. *Algèbre et Combinatoire, Tome II.* Hachette, Paris.

Birkhoff , B., Lattice Theory, volume 25. *American Mathematical Society Colloquium Publ.*, Providence, revised edition, 1973.

Blake, C.L., and Merz, C.J., 1998, UCI Repository of machine learning databases [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

Bordat, J.P., 1986, Calcul pratique du treillis de Galois d'une correspondance. *Math. Sci. Hum.*, **96**, pp. 31–47.

Cameron, P.J., 1996, *Combinatorics: topics, techniques, algorithms.* University of Cambridge Press.

Carpineto, C., and Romano, G., 1993, GALOIS: an order-theoretic approach to conceptual clustering. *Machine Learning, proceedings of the tenth International conference, Amherst, MA*, Morgan Kaufmann Publishers, pp. 33-40.

Carpineto, C., and Romano, G., 1996, Information retrieval through hybrid navigation of lattice representations. *International Journal of Human-Computer Studies*, **45**, pp. 553-578.

Carpineto, C., and Romano, G., 1996, A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning Journal*, **24**, pp. 95-122.

Chein, M., 1969, Algorithme de recherche des sous-matrices premières d'une matrice. *Bull. Math. Soc. Sci. Math.*, R.S. Roumanie, **13**, pp. 21–25.

Clark, C.W., 1931, *Elementary mathematical analysis, second edition*, Brooks/Cole Publishing Company.

Cole, R., and Eklund, P., 2001, Browsing semi-structured web texts using formal concept analysis. *Proceedings of ICCS-2001 International workshop on concept lattices-based theory, methods, and tools for knowledge discovery in databases (CLKDD'01),* Stanford University, Palo Alto, Springer, pp. 319-332.

Cole, R., and Stumme, G., 2000, CEM: A Conceptual Email Manager. *Proceedings of the 8th International Conference on Conceptual Structures, ICCS'2000, Springer Verlag, 2000,* LNAI **1867**, pp. 438-452.

Dowling, C.E., 1993, On the irredundant generation of knowledge spaces. *J. Math. Psych.*, **37**(1), pp. 49-62.

Duquenne, V., 1999, Lattical structures in data structures in data analysis. *Theoretical Computer Science,* **217**(2), pp. 407-436.

Duquenne, V., Chabert, C., Cherfouh, A., Delebar, J.M., Doyen, A., and Pickering, D., 2001, Structuration of Phenotypes / Genotypes through Galois Lattices and Implications. *Proceeding of the ICCS-2001 International workshop on concept lattices-*

based theory, methods, and tools for knowledge discovery in databases (CLKDD'01), Stanford University, Palo Alto, Springer, pp. 21-32.

Fisher, D., 1987, Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, **2**, pp. 139-172.

Ganter, B., 1984, Two Basic Algorithms in Concept Analysis, FB4-Preprint No. 831, TH Darmstadt and supplemented with construction of the cover relationship via binary search.

Ganter, B., Attribute exploration with background knowledge. *Theoretical Computer Science*, **217** (1999), 215-233.

Ganter, B., and Wille, R., 1999, *Formal Concept Analysis, Mathematical Foundations,* Berlin, Springer-Verlag.

Ganter, B., Rindfrey, K., and Skorsky, M., 1986, Software for formal concept analysis. *In Classification as a tool of research,* Elsevier Science.

Godin, R., and Missaoui, R., 1994. An Incremental Concept Formation Approach for Learning from Databases. *Theoretical Computer Science, Special Issue on Formal Methods in Databases and Software Engineering*, **133**(2), pp. 387-419.

Godin, R., Missaoui, R., and Alaoui, H., 1991, Learning Algorithms using a Galois Lattice Structure. *In Proceedings of International Conference on Tools with Artificial Intelligence (ICTAI), San Jose, CA, November*, pp. 22-29.

Godin, R., Mineau, G. W., and Missaoui, R., 1995, Incremental structuring of knowledge bases. *In Proceedings of the first International Symposium on Knowledge Retrieval, Use and Storage for Efficiency (KRUSE'95), Santa Cruz, CA, USA*, pp. 179-198.

Godin, R., Missaoui, R., and Alaoui, H., 1995, Incremental concept formation algorithms based on Galois lattices. *Computation Intelligence*, **11**(2), pp. 246-267.

Goldberg, L.A., 1993, *Efficient Algorithms for Listing Combinatorial Structures.* Cambridge University Press.

Gratzer, G., 1971, *Lattice Theory: First Concepts and Distributive Lattices,* W.H. Freeman & Co.

Guénoche, A., 1990, Construction du treillis de Galois d'une relation binaire. *Math. Inf. Sci. Hum.*, **109**, pp 41-53.

Hereth, J., and Stumme, G., 2001, Reverse pivoting in conceptual information systems. *Proceedings of the ICCS-2001 International workshop on concept lattices-based theory, methods, and tools for knowledge discovery in databases (CLKDD'01), Stanford University, Palo Alto*, Springer, pp. 202-215.

Johnson, D.S., Yannakakis, M., and Papadimitriou, C.H., 1988, On generating all maximal independent sets. *Inf. Proc. Let.*, **27**, pp 119-123.

Kourie, D.G., and Oosthuizen, G.D., 1998. Lattices in machine learning: complexity issues. *Acta Informatica,* **35**, pp. 269-292.

Kuznetsov, S.O., 1989, Interpretation on Graphs and Complexity Characteristics of a Search for Specific Patterns, *Nauch. Tekh. Inf.*, Ser. **2**, no. **1**, pp. 23–28.

Kuznetsov, S.O., 1993, A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Automatic Documentation and Mathematical Linguistics*, **27**(5), pp. 11–21.

Kuznetsov, S.O., 2001, On computing the size of a lattice and related decision problems. *Order*, **18**(4), pp. 13-21.

Kuznetsov, S.O, and Obiedkov, S.A., 2001, Comparing Performance of Algorithms for Generating Concept Lattices. *Proceedings of ICCS-2001 International workshop on concept lattices-based theory, methods, and tools for knowledge discovery in databases (CLKDD'01), Stanford University, Palo Alto*, Springer Verlag, pp. 35-47.

Kuznetsov, S.O., and Obiedkov, S.A., 2002, Comparing performance of algorithms for generating concept lattices. *In Journal of Experimental & Theoretical Artificial Intelligence, Special issue on Concept Lattice-based theory, methods and tools for knowledge Discovery in Databases*, Volume **14,** number **2/3** April-September 2002, pp. 189-216.

Lindig, C., Algorithmen zur begriffsanalyse und ihre anwendung bei softwarebibliotheken, (Dr.-Ing.) Dissertation, Techn. Univ. Braunschweig.

Lindig, C., Fast Concept Analysis. In Gerhard Stumme, editors, *Working with Conceptual Structures - Contributions to ICCS 2000*, Shaker Verlag, Aachen, Germany, 2000. (http://www.eecs.harvard.edu/~lindig/papers/)

Mephu Nguifo, E., and Nijiwoua, P., 2001, IGLUE: a lattice-based constructive induction system. *Intelligent Data Analysis*, **5**(1), pp. 73-91.

Norris, E.M., 1978, An algorithm for computing the maximal rectangles in a binary relation. *Revue Roumaine de Mathématiques Pures et Appliquées,* **23**(2), pp. 243–250.

Nourine, L., and Raynaud, O., 1999, A fast algorithm for building lattices. *Information Processing Letters*, **71**, pp. 199-204.

Nourine, L., and Raynaud, O., A fast incremental algorithm for building lattices. *In Journal of Experimental & Theoretical Artificial Intelligence, Special issue on Concept Lattice-based theory, methods and tools for knowledge Discovery in Databases*, Volume **14,** number **2/3** April-September 2002, pp. 217-228.

Obiedkov, S.A., Personal communications 2001-2003. Russian State University for the Humanities, Moscow, Russia.

Oosthuizen, G.D., 1991, Lattice-based Knowledge Discovery. *In Proceedings of AAAI-91 Knowledge Discovery in Databases Workshop, Anaheim*, pp. 221-235.

Oosthuizen, G.D., 1994, A Dynamic Indexing Mechanism for Memory-based Reasoning. *Proceedings of the international AMSE conference on 'intelligent systems',* SMSE Press, pp. 127-136.

Oosthuizen, G.D., 1994, The application of concept lattices to machine learning. Technical Report CSTR 94/01 Department of Computer Science University of Pretoria.

Oosthuizen, G.D., and Avenant, C., 1992, Integrating Similarity- based Learning and Explanation-based learning. *South African Computer Journal,* **6**, 1992, pp. 72-78.

Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L., 1999, Discovering frequent closed itemsets for association rules. *In Proceedings of 7th International Conference on Database Theory (ICDT), Jerusalem, Israel*, January, pp. 398-416.

Pernelle, N., Rousset, M.-C., Soldano, H., and V. Ventos, 2002. *In Journal of Experimental & Theoretical Artificial Intelligence, Special issue on Concept Lattice-based theory, methods and tools for knowledge Discovery in Databases*, Volume **14,** number **2/3** April-September 2002, pp. 157-188.

Quinlan, J.R., 1986, Induction of decision trees. *Machine Learning,* **1**(1): pp. 81-106.

Salton, G., 1989, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Reading, MA: Addison-Wesley.

Stumme, G., Wille, R., and Wille, U., 1998, Conceptual Knowledge Discovery in Databases Using Formal Concept Analysis Methods. *In Proc. 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98), Nantes, France,* pp. 450-458.

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., and Lakhal, L., 2000, Fast computation of concept lattices using data mining techniques. *In proceesings of 7th International Workshop on Knowledge Representation meets Databases (KRDB 2000), Berlin, Germany,* August 21-22, pp. 129-139.

Valtchev, P., Missaoui, R., and Lebrun, P., 2000, A Partition-Based Approach Towards Building Galois (Concept) Lattices, Rapport de recherche no. 2000-08, Département d'Informatique, UQAM, Montréal, Canada.

Valtchev, P., and Missaoui, R., 2001, Building concept (Galois) lattices from parts: generalizing the incremental methods. *Conceptual structures: broadening the base, 9th International conference on conceptual structures, ICCS 2001, Stanford,* Springer, pp. 290-303.

Vogt, F., and Wille, R., TOSCANA - a graphical tool for analyzing and exploring data. *In: R. Tamassia, I. G. Tollis (eds.): Graph Drawing.* Springer, Berlin-Heidelberg-New York 1995, pp. 193-205.

Wille, R., 1982, Restructuring lattice theory: an approach based on hierarchies of concepts. *In I. Rival (ed.) Ordered sets (Dordrecht & Boston: Reidel),* pp. 445-470.

Wille, R., 2001, Why Can Concept Lattices Support Knowledge Discovery in Databases?. *In proceedings of ICCS-2001 International workshop on concept lattices-based theory, methods, and tools for knowledge discovery in databases (CLKDD'01), Stanford University, Palo Alto,* Springer, pp. 7-20.

Wille, R., 2002, Why can concept lattices support knowledge discovery in databases? *In Journal of Experimental & Theoretical Artificial Intelligence, Special issue on Concept Lattice-based theory, methods and tools for knowledge Discovery in Databases,* Volume **14**, number **2/3** April-September 2002, pp. 81-92.

Xie, Z., Hsu, W., Liu, Z., and Li Lee, M. 2002, Concept lattice based composite classifiers for high predictability. *In Journal of Experimental & Theoretical Artificial Intelligence, Special issue on Concept Lattice-based theory, methods and tools for knowledge Discovery in Databases,* Volume **14**, number **2/3** April-September 2002, pp. 143-156.

Yevtushenko, S., 2002, BDD-based Algorithms for the construction of the Set of All Concepts. *In Contributions to ICCS 2002, Borovets, Bulgaria,* pp. 61-73.

Zabezhailo, M.I., Ivashko, V.G., Kuznetsov, S.O., Mikheenkova, M.A., Khazanovskii, K.P., and Anshakov, O.M., Algorithms and Programs of the JSM-Method of Automatic Hypothesis Generation, *Nauch. Tekh. Inf.*, Ser. **2**, 1987, no. **10**, pp. 1–14.

Appendix A: Further optimised AddAtom algorithm

This appendix documents the variation of the AddAtom algorithm with the best time performance. Other variations of the algorithm can be found in sections 4.3 and 4.6.

Note that for the sake of simplicity of the pseudo code, it is assumed that the context has no comparable objects and attributes and the objects and attributes of the context form the atoms and coatoms of the lattice. This is the case when FCA-lattices are equal to EA-lattices. Slight modifications and additional tests are necessary depending on which type of lattice is to be built.

```
Input context ⟨O, A, I⟩
1L = NewConcept(L)
0L = NewConcept(L)
0L.Intent = A
For ∀ a ∈ A
   aAttributeConcept = NewConcept(L)
   aAttributeConcept.Intent = {a}
   CreateArc(L, 0L, aAttributeConcept)
   CreateArc(L, aAttributeConcept, 1L)
Rof
For ∀ o ∈ O
   FastAddAtom (L, Intent(o), o, 0L)
Rof

//=================================================================
Function GetMeet(anIntent, aConcept, attrCount) Return aConcept
//=================================================================
// Determine the meet of anIntent by starting at Concept
parentIsMeet = True
Do While parentIsMeet
   parentIsMeet = False
   Parents = ConceptParents(L, aConcept)
   For ∀ Parent ∈ Parents
      If attrCount[Parent] = || anIntent || then
         aConcept = Parent
         parentIsMeet = True
         Exit For
      Fi
   Rof
Od
Return aConcept
End GetMeet

//=================================================================
Function AddAtomRecurse(L, anIntent, GeneratorConcept, attrCount,
                     ExactConcepts, DirtyConcepts, IgnoreConcepts)
                      Return aConcept
//=================================================================
CadidateParents = ConceptParents(L, GeneratorConcept)
ConceptParents = ∅
UCConceptParents = ∅
DCConceptParents = ∅
Exit = False
j = 0
// Concepts in CandidateParents that have the highest number of
// attributes of anIntent in their intents should be considered first
// and therefore sorted in decending order of the number of attrCount
For ∀ Candidate ∈ CandidateParents
   SortArray [j] = Candidate
   j = j + 1
Rof
Sort SortArray in decending order of attrCount[SortArray[j]]
For ∀ k = 0 to j - 1
   // Get candidate with next highest number of markers
   Candidate = SortArray[k]
   If (Candidate ∉ IgnoreConcepts) and (Candidate ∉ UCConceptParents)
         and (Candidate ∉ DCConceptParents) and Not Exit
      //Only Candidates with at least one attribute of anIntent should
      // be considered
```

```
      newIntent = Candidate.Intent ∩ anIntent
      Generator = GetMeet(L, newIntent, Candidate, attrCount,
                  ExactConcepts, DirtyConcepts, IgnoreConcepts)
      If Generator ∉ ExactConcepts then
         Generator = AddAtomRecurse(L, newIntent, Generator,
                     attrCount, ExactConcepts, DirtyConcepts,
                     IgnoreConcepts)
      Fi
      // At this point Generator is now an exact meet of anIntent
      If newIntent = anIntent then
         Exit = True
      Fi
      If Generator ∉ UCConceptParents and not Exit then
         ConceptParents = ConceptParents ∪ {Generator}
         If attrCount[Generator] > 1 then
            // If the Generator is not an attribute we can remove the
            // concepts below and above it from consideration - this
            // is possible because all concepts that will be considered
            // hereafter will have a smaller attrCount
            UCGenerator = UpwardClosure(L, Generator)
            ConceptParents = ConceptParents - UCGenerator
            // Do not consider ConceptParents that are spanned by
            // Generator
            UCConceptParents = UCConceptParents ∪ UCGenerator
            //Concepts above need not be considered
            DCGenerator = DownwardClosure(L, Generator)
            DCConceptParents = DCConceptParents ∪ DCGenerator
            //Concepts below it will not be considered
         Fi
      Fi
   Fi
Rof
NewConcept = CreateNewConcept(L)
NewConcept.Extent = GeneratorConcept.Extent
NewConcept.Intent = anIntent
attrCount [NewConcept] = ||anIntent||
ExactConcepts = ExactConcepts ∪ {NewConcept}
For ∀ ConceptMeet ∈ ConceptParents
   If ConceptMeet in CandidateParents then
      DeleteArc(GeneratorConcept, ConceptMeet)
   Fi
   CreateArc(NewConcept , ConceptMeet)
Rof
DeleteArc(GeneratorConcept , NewConcept)
Return NewConcept
End AddAtomRecurse
//=================================================================

//=================================================================
Function FastAddAtom(L, anIntent, o, GeneratorConcept)
//=================================================================
DirtyAttrs = GetAttributes(L) - anIntent
DirtyConcepts = ∅
// DirtyConcepts: contains intents with attributes other than Intent
// and therefore all the approximate meets of anIntent
For ∀ attr ∈ DirtyAttrs
   DirtyConcepts = DirtyConcepts ∪ DownwardClosure(L, attr)
   //It is also possible to calculate DirtyConcepts using attrCount
Rof
CandidateConcepts = ∅
```

```
For ∀ attr ∈ anIntent
   CandidateConcepts = CandidateConcepts ∪ DownwardClosure(L, attr)
Rof
ExactConcepts = CandidateConcepts - DirtyConcepts
// ExactConcepts have only attributes of anIntent in their intents
// and form exact meets of subsets of anIntent
IgnoreConcepts = DirtyConcepts - CandidateConcepts
// IgnoreConcepts have no attributes of anIntent in their intents
// and can be ignored when searching for GeneratorConcepts

// Calculate markers: attrCount[Concept] is the number
// of attributes of anIntent for that concept (i.e. markers
// accumulated)
Let attrCount [x] = 0 for all x ∈ L
For ∀ Concept ∈ CandidateConcepts
   attrCount [Concept] = ||Concept.intent ∩ anIntent ||
Rof
NewConcept = AddAtom2(L, anIntent, EmptyConcept(L), GeneratorConcept,
               attrCount, ExactConcepts, DirtyConcepts, IgnoreConcepts)
For ∀ Concept ∈ UpwardClosure(L, NewConcept)
   Concept.Extent = Concept.Extent ∪ {g}
Rof
End FastAddAtom
//==================================================================
```

Appendix B: AddAtom algorithmic complexity bounds

The algorithm outline below show complexity bounds of the steps or group of steps for various parts of the AddAtom algorithm (documented in section 4.6). The complexity column indicates the complexity bound or alternatively the maximum number of iterations in the case of loops.

```
Step                                        Complexity

Function OptimisedAddAtom
   .                                        O(||A||)
   For a                                    O(||A||) times
      .                                     O(max(||O||, ||A||))
   Rof
   For o                                    O(||O||) times
      .                                     O(||Lj||)
      For x                                 O(||Lj||) times
         .                                  O(max(||A||))
      Rof
      AddAtom()
      For x                                 O(||Lj||) times
         .                                  O(1)
      Rof
   Rof
End OptimisedAddAtom

Function GetMeet()                          O(max(||O'||).||O||)
   Do While ParentIsMeet                    O(max(||O'||)) times
      For Parent ...                        O(||O||) times
         If ...                             O(1)
            .                               O(1)
         Fi
      Rof
   Od
End GetMeet

Function AddAtom()
   .                                        O(||O||)
   For Candidate ...                        O(||O||) times
      .                                     O(||A||)
      If ...                                O(||A||)
         GetMeet()                          O(max(||O'||).||O||)
         If
            AddAtom()
         Fi
      Else
         .                                  O(1)
      Fi
      For g ...                             O(||O||) times
         .                                  O(max(||O'||))
      Rof
   Rof
   .                                        O(max(||A'||, ||O'||))
   For g ...                                O(||O||) times
      .                                     O(1)
   Rof
   .                                        O(1)
End AddAtom
```