Science of Computer Programming 74 (2009) 128–142

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

An incremental algorithm to construct a lattice of set intersections

Derrick G. Kourie [a,*], Sergei Obiedkov [a,b], Bruce W. Watson [a], Dean van der Merwe [a]

[a] Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa
[b] Department of Applied Mathematics, Higher School of Economics, Moscow, Russia

ARTICLE INFO

Article history:
Received 1 November 2005
Received in revised form 1 March 2007
Accepted 1 June 2008
Available online 1 November 2008

Keywords:
Concept lattice
Closure system
Incremental algorithm
Correctness by construction

ABSTRACT

An incremental algorithm to construct a lattice from a collection of sets is derived, refined, analyzed, and related to a similar previously published algorithm for constructing concept lattices. The lattice constructed by the algorithm is the one obtained by closing the collection of sets with respect to set intersection. The analysis explains the empirical efficiency of the related concept lattice construction algorithm that had been observed in previous studies. The derivation highlights the effectiveness of a correctness-by-construction approach to algorithm development.

1. Introduction

Potentially, graphs that represent concept lattices [1–3] can be a rich source of information about the inter-relationship between a set of arbitrary objects that share certain discrete attributes. In recent years, concept lattices have been used extensively as a knowledge representation and management framework in various domains. These include domains such as linguistics [4], social network analysis [5,6], ontology building [7], and information retrieval [8]. Concept lattices also play an important role in some machine learning methods [9] and data mining techniques [10]. Another major area of application is software construction and engineering. (See, for example, [11–16] and a survey in [17]).

However, algorithms to set up these graphs are notoriously inefficient, having exponential worst-case complexity (which is due to the exponential size of the output in the worst case). Here, a correctness-by-construction approach that relies on the guarded command language for notation is used to derive an algorithm that has been shown to perform dramatically better than others when applied to live data.

In Section 2, a number of definitions are provided, and the notation that is used is introduced. This is followed by the derivation of the root algorithm in Section 3. Two variants of this algorithm are derived in Section 4. In Section 5, the worst-case bound of one of the algorithms is derived, using reasoning that is similar to what has been used for other concept lattice construction algorithms [18]. A conclusion follows in Section 6.

2. Preliminaries

2.1. Lattices

First, we recall basic definitions from lattice theory [19,20].

Definition 2.1. A *lattice* is a partially ordered set denoted by $(L, \leq)$ in which, for every pair of elements, there exists the least upper bound (or supremum) and the greatest lower bound (or infimum). A lattice is *complete* if suprema and infima exist

* Corresponding author.
E-mail addresses: dkourie@cs.up.ac.za (D.G. Kourie), sergei.obj@gmail.com (S. Obiedkov), watson@bruce-watson.com (B.W. Watson), dean.van.der.merwe@sap.com (D. van der Merwe).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.015

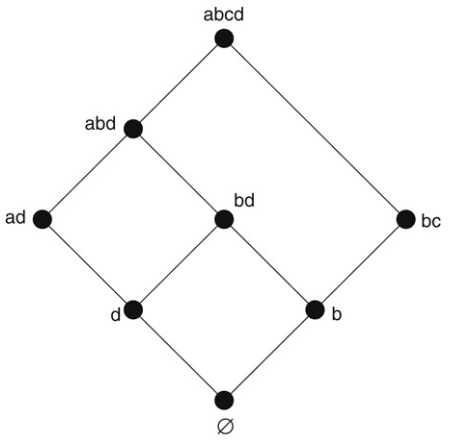

Fig. 1. Example line diagram of a lattice.

for every one of its subsets. Infimum and supremum are often called *meet* and *join*, respectively. $X \wedge Y$ and $X \vee Y$ denote the meet and join of X and Y, respectively.

All (non-empty) finite lattices are complete. In what follows, lattices are always assumed finite.

Where the ordering relation $\leq$ is clear from the context, the lattice is simply indicated by L. The maximum and minimum elements of a non-empty lattice L are denoted by $\top_L$ and $\bot_L$ respectively.

As with any partially ordered set, a lattice L can be represented visually by a line (or Hasse) diagram in which elements are depicted as nodes. Suppose $c, p \in L$ and $c \leq p$. If there is no $r \in L$ such that $r \neq p$, $r \neq c$, and $c \leq r \leq p$, then p is depicted above c and is connected to it by an arc in the associated line diagram. In this case, we say that c is a *child* of p and p is a *parent* of c and denote this relationship by $c \prec p$. This child–parent relationship on a partially ordered set L (which is actually the transitive reflexive reduction of the partial order in question) is called the *cover relation* of L. The graph $(L, \prec)$ is called the *covering graph* of L. In some cases, it will be useful to explicitly indicate the partially ordered set in question; then, we write $c \prec_L p$.

2.2. Set Intersection-closed Lattices

Definition 2.2. A *closure system* $\mathcal{C}$ on some alphabet A is a collection of subsets of A that contains A and that is closed under set intersection—i.e.:

$$\mathcal{C} \text{ is a closure system } \Leftrightarrow (A \in \mathcal{C}) \wedge \left(\forall \mathcal{X} \subseteq \mathcal{C} : \bigcap \mathcal{X} \in \mathcal{C}\right)$$

It is well known that a closure system ordered by set inclusion, $\subseteq$, is a complete lattice; i.e., if $\mathcal{C}$ is a closure system, then $(\mathcal{C}, \subseteq)$ is a complete lattice. We will refer to such a lattice as a *set intersection-closed lattice* (SICL). The meet and join operations in a SICL $(\mathcal{C}, \subseteq)$ are given by the following expressions:

$$X \wedge Y = X \cap Y;$$
$$X \vee Y = \bigcap \{Z \in \mathcal{C} \mid X \cup Y \subseteq Z\}.$$

Note that the join operation does not always coincide with the union, since if $X \in \mathcal{C}$ and $Y \in \mathcal{C}$, it need not necessarily be the case that $X \cup Y \in \mathcal{C}$.

Fig. 1 shows a line diagram of such a SICL with alphabet $\{a, b, c, d\}$. In this figure, as well as in forthcoming examples, a set $\{a, b, c, d\}$ is abbreviated to `abcd`, $\{a, b, c\}$ to `abc`, etc. The SICL is the set of nodes in the diagram, {`abcd`, `abd`, `ad`, `bd`, `bc`, `d`, `b`, Ø}. The line diagram represents the covering graph of this SICL, where a node p is connected by a downward arc to a node c if $c \prec p$. (For example, the fact that `abd` $\prec$ `abcd` means that, in the diagram, `abcd` should be connected by the downward arc to `abd`, etc.).

The top element of the lattice in the figure is `abcd`, the bottom element is Ø and there are various other elements in between. Note that the intersection of every pair of nodes is also a node in the figure. However, the inverse does not have to hold: nodes `abcd`, `abd`, `ad`, `bd`, and `bc` are not produced by the intersection of any other nodes. Such nodes have fewer than 2 parents in the cover relation. The join operation is different from the union: e.g., `bc` $\vee$ `bd` = `abcd`.

We can now formulate the problem to be addressed in the sections that follow:

> Given a SICL $\mathcal{L}$ on alphabet A, its cover relation, and a set $S \subseteq A$, add a minimal number of elements to $\mathcal{L}$ and modify its cover relation accordingly so that $S \in \mathcal{L}$ and $\mathcal{L}$ remains a SICL.

The result of such update is unambiguous: $\mathfrak{L}$ must be enlarged to include S and pairwise intersections of S and every element of $\mathfrak{L}$. This leads us to Definition 2.3.

Definition 2.3. Let $\mathcal{F}$ and $\mathcal{H}$ be two arbitrary families of subsets of some alphabet A and let $X \subseteq A$. Then, $\mathcal{H}$ is said to be an *X-extension* of $\mathcal{F}$, denoted by $\mathcal{F} \sqsubseteq_X \mathcal{H}$, if

- $\mathcal{F} \subseteq \mathcal{H}$ and
- $\forall Y \in \mathcal{H} \setminus \mathcal{F} : ((X = Y) \vee (\exists Z \in \mathcal{F} : Y = (X \cap Z)))$.

This means that each element in $\mathcal{H}$ that is not in $\mathcal{F}$, aside from X itself, can be derived by intersecting some element in $\mathcal{F}$ with X. If $\mathcal{H}$ corresponds to the lattice in Fig. 1 and $\mathcal{F} = \{\mathtt{abcd},\ \mathtt{abd},\ \mathtt{bd},\ \mathtt{bc},\ \mathtt{b},\ \emptyset\}$, then $\mathcal{F} \sqsubseteq_{\mathtt{ad}} \mathcal{H}$. This is because the only element in $\mathcal{H} \setminus \mathcal{F}$ other than $\mathtt{ad}$ itself is $\mathtt{d}$ and this element is the result of intersecting $\mathtt{ad}$ with an element in $\mathcal{F}$, namely $\mathtt{bd}$. On the other hand, if $\mathcal{F}$ consisted of all elements of $\mathcal{H}$ apart from $\mathtt{b}$ and $\mathtt{d}$, then $\mathcal{F} \not\sqsubseteq_{\mathtt{b}} \mathcal{H}$ since $\mathtt{d}$ in $\mathcal{H} \setminus \mathcal{F}$ cannot be derived by intersecting $\mathtt{b}$ with any element in $\mathcal{F}$. Similarly, $\mathcal{F} \not\sqsubseteq_{\mathtt{d}} \mathcal{H}$.

Returning to our problem statement, the problem to be solved by the algorithm can be specified as follows:

Problem 2.4. Given:

- A graph $(\mathfrak{L}_0, \prec_{\mathfrak{L}_0})$, such that $\mathfrak{L}_0$ is a SICL on alphabet A;
- A set $S \subseteq A$;

Find:

- The graph $(\mathfrak{L}, \prec_{\mathfrak{L}})$ such that $\mathfrak{L}$ is a SICL on A, $S \in \mathfrak{L}$ and $\mathfrak{L}_0 \sqsubseteq_S \mathfrak{L}$.

Proposition 1. *In a SICL $\mathfrak{L}$, if $C_1, C_2, P \in \mathfrak{L}$, $C_1 \prec P$, $C_2 \prec P$, and $C_1 \neq C_2$, then $C_1 \not\subseteq C_2$ and $C_2 \not\subseteq C_1$.*

In other words, although children of a node in the line diagram of a SICL may have non-empty intersections with one another, one may not be a subset of the other.

Proposition 2. *If T is some arbitrary element of a SICL $\mathfrak{L}$, it follows that the set $\{S \in \mathfrak{L} \mid S \subseteq T\}$ is also a SICL whose top element is T.*

The set $\{S \in \mathfrak{L} \mid S \subseteq T\}$ is known as a *principal ideal* in lattice theory and is often denoted by $\downarrow T$. A principal ideal of a lattice is always a lattice.

Thus, if $\mathfrak{L}$ is the SICL in Fig. 1, then $\downarrow \mathtt{ad} = \{\mathtt{ad}, \mathtt{d}, \emptyset\}$ is also a SICL. Nodes in $\downarrow T$ might be children, but not parents, of nodes in $\mathfrak{L} \setminus \downarrow T$.

3. The algorithm

Let $\mathcal{F}$ be a collection of sets over A and let $S \in \mathcal{F}$. Assume the following for a graph whose set of nodes is $\mathcal{F}$, and whose set of edges is E, i.e. for the graph $(\mathcal{F}, E)$:

$$
\begin{aligned}
E(S) &= \{T \mid (S, T) \in E\} \text{ is the set of children of } S \text{ in } (\mathcal{F}, E);\\
E^* &= \{(S, T) \mid S = T \text{ or } \exists U \in \mathcal{F} : ((S, U) \in E \text{ and } (U, T) \in E^*)\} \text{ is the reflexive transitive closure of the relation } E;\\
\mathcal{F} \restriction S &= \{T \mid (S, T) \in E^*\} \text{ is the set consisting of the node } S \text{ and all its descendants in } (\mathcal{F}, E);\\
E \restriction S &= E \cap (\mathcal{F} \restriction S \times \mathcal{F} \restriction S);\\
CS(\mathcal{F}) &\text{ is } \textit{true} \text{ iff the collection } \mathcal{F} \text{ is a closure system;}\\
CG(\mathcal{F}, E) &\text{ is true if } \mathcal{F} \text{ is a closure system and } (\mathcal{F}, E) \text{ is its covering graph, i.e., } CS(\mathcal{F}) \wedge (E = \prec_{\mathcal{F}}).
\end{aligned}
$$

An incremental SICL construction algorithm called $insert_0((\mathfrak{L}, E), X)$ is derived. When describing the algorithm, we will use the subscript $(\cdot)_0$ to denote the state of a variable prior to the call of the function where it occurs. For example, $(\mathfrak{L}_0, E_0)$ within the body of the function $insert_0((\mathfrak{L}, E), X)$ denotes the state of the data structure $(\mathfrak{L}, E)$ at the entry point of $insert_0((\mathfrak{L}, E), X)$.

The algorithm $insert_0((\mathfrak{L}, E), X)$ inserts a new set X into $(\mathfrak{L}_0, \prec_{\mathfrak{L}_0})$, the covering graph of a SICL $\mathfrak{L}_0$, to yield $(\mathfrak{L}, \prec_{\mathfrak{L}})$, the covering graph of the $\mathfrak{L}$ that is the (unique) X-extension of $\mathfrak{L}_0$ being a SICL and accommodating X.

3.1. Basic structure

For $\mathfrak{L}$ to remain a closure system after X has been inserted, the resulting $\mathfrak{L}$ must contain all intersections of X with the other nodes in $\mathfrak{L}$. Therefore, a postcondition of adding X to $\mathfrak{L}$ is $\{\forall Y \in \mathfrak{L} : X \cap Y \in \mathfrak{L}\}$. It follows that the algorithm may have to insert nodes other than X into $\mathfrak{L}$. However, since new nodes should be limited to only those that are necessary, $\mathfrak{L}_0 \sqsubseteq_X \mathfrak{L}$ is a conjunct in the postcondition of $insert_0((\mathfrak{L}, E), X)$.

The algorithm assumes that $\mathfrak{L}$ has at least one element, $\top_{\mathfrak{L}}$, and that the element to be inserted into $\mathfrak{L}$ is a proper subset of $\top_{\mathfrak{L}}$. It is trivial to adapt the algorithm for cases where this does not apply. The pre- and postconditions for the algorithm may therefore be stated as follows.

proc $insert_0((\mathfrak{L}, E), X)$
pre : $\{(\mathfrak{L}, E) = (\mathfrak{L}_0, E_0) \wedge CG(\mathfrak{L}, E) \wedge X \subseteq \top_{\mathfrak{L}}\}$
$(\mathfrak{L}, E) : S$
post : $\{\mathfrak{L}_0 \sqsubseteq_X \mathfrak{L} \wedge CG(\mathfrak{L}, E) \wedge X \in \mathfrak{L}\}$

Note that the notation $(\mathfrak{L}, E) : S$ means that the statement S is allowed to modify the variables $\mathfrak{L}$ and E. In particular, in the algorithm above, S inserts pairwise intersections of X, on one hand, and every element of $\mathfrak{L}$, on the other hand, into $\mathfrak{L}$, modifying E accordingly. The following sections describe how this modification can be carried out.

3.2. The main loop

In discussing the expansion of statement S, note that, unless X is already part of $\mathfrak{L}$, it can have only one parent. It is necessary to identify this parent, as well as all children of X. Let C_i, $i = 1, \ldots, n$, be the children of $\top_{\mathfrak{L}}$. Recall that each such child, C_i, is itself the top element of the principal ideal $\downarrow C_i$ (which is a SICL on alphabet C_i), and hence the top element of $(\mathfrak{L} \restriction C_i, E \restriction C_i)$, the covering graph of $\downarrow C_i$. An orderly approach to inserting X into $(\mathfrak{L}, E)$ is to consider the positioning of X in relation to each such subgraph $(\mathfrak{L} \restriction C_i, E \restriction C_i)$ in turn. This implies an iteration over all the original children of $\top_{\mathfrak{L}}$, during which possible children of X will be identified and gathered together into a set called $\mathcal{C}$.

In so doing, we make sure that, after the first $k \leq n$ subgraphs $(\mathfrak{L} \restriction C_1, E \restriction C_1)$, …, $(\mathfrak{L} \restriction C_k, E \restriction C_k)$ have been processed, the set $\mathcal{C}$ contains all maximal subsets of X from these subgraphs. These maximal subsets will be children of X in the updated graph $(\mathfrak{L}, E)$, unless we find more suitable candidates in the yet unexplored parts of the subgraphs $(\mathfrak{L} \restriction C_{k+1}, E \restriction C_{k+1}), \ldots, (\mathfrak{L} \restriction C_n, E \restriction C_n)$. To capture this formally, let $\mathcal{M}_i$ be the set of the first i children of $\top_{\mathfrak{L}}$ and their descendants:

$$\mathcal{M}_i = \begin{cases} \bigcup_{j=1}^{i} \mathfrak{L} \restriction C_j & \text{if } i = 1, \ldots, n \\ \emptyset & \text{if } i = 0. \end{cases}$$

We shall require that changes to $(\mathfrak{L}, E)$ and $\mathcal{C}$ conform to the following invariant as the loop progresses. (Recall that $\mathfrak{L}_0$ denotes the original lattice.)

$$\begin{aligned} &\mathit{Invariant} && \triangleq \mathit{inv}1 \wedge (X \notin \mathfrak{L} \Rightarrow \mathit{inv}2) \\ &\text{where } \mathit{inv}1 && \triangleq CG(\mathfrak{L}, E) \wedge \mathfrak{L}_0 \sqsubseteq_X \mathfrak{L} \\ &\text{and } \mathit{inv}2 && \triangleq \forall C \in \mathcal{C} : C \subset X \wedge \forall Y \in \mathcal{M}_i : (Y \subset X \Rightarrow \exists C \in \mathcal{C} : Y \subseteq C) \end{aligned}$$

Thus, $inv1$ requires that $\mathfrak{L}$ remains a SICL whatever happens and that it remains an X-extension of its original value. This means that it changes minimally, while retaining all nodes that it originally had. Note, however, that $inv1$ does not incorporate the requirement that X already has to be in the lattice $\mathfrak{L}$—it merely requires that the intersection of X with every new node in $\mathfrak{L}$ also has to be in $\mathfrak{L}$.

The second part of the invariant, $inv2$, relates to invariant requirements with respect to $\mathcal{C}$. In particular, it requires that all elements of $\mathcal{C}$ are subsets of X (first conjunct) and that, for any subset of X that is in $\mathcal{M}_i$, it or its superset is always an element of $\mathcal{C}$ (second conjunct). This ensures that, by the time we have processed all nodes C_i, the set $\mathcal{C}$ will contain all children of X in the updated graph. However, $\mathcal{C}$ is only an auxiliary structure, and once $X \in \mathfrak{L}$ becomes true, we no longer have to maintain it. Thus, the second conjunct of the invariant is conditional: $X \notin \mathfrak{L} \Rightarrow inv2$.

In general, the incorporation of X into the graph $(\mathfrak{L}, E)$ is left to a second phase after completing the loop over the children of $\top_{\mathfrak{L}}$. Nevertheless, it will be seen that, in some cases, it is possible to terminate the construction even before all children of $\top_{\mathfrak{L}}$ have been considered, because X can easily and naturally become a part of $\mathfrak{L}$. To indicate this eventuality, we use a flag *inserted* that allows for an earlier termination of the loop when X is already in the lattice. Using this flag, the invariant can be re-stated as $inv1 \wedge (\neg \mathit{inserted} \Rightarrow inv2)$.

The postcondition of the loop is therefore: $\mathit{Invariant} \wedge ((i = n) \vee \mathit{inserted})$, where n is the number of children of $\top_{\mathfrak{L}_0}$ as above. Of course, if $i = n$, then $\mathcal{M}_i$ corresponds to all nodes in the constructed lattice, $\mathfrak{L}$, except $\top_L$.

The invariant is initialized by $i, n, \mathcal{C}, \mathit{inserted} := 0, |E_0(\top_L)|, \emptyset, \mathit{false}$. The following structure for statement S thus suggests itself:

$\{(\mathfrak{L}, E) = (\mathfrak{L}_0, E_0) \wedge CG(\mathfrak{L}, E) \wedge X \subseteq \top_{\mathfrak{L}}\}$
$i, n, \mathcal{C}, \mathit{inserted} := 0, |E(\top_{\mathfrak{L}})|, \emptyset, \mathit{false}$
$\{\mathit{Invariant}\}$
;**do** $(i \neq n \wedge \neg \mathit{inserted}) \rightarrow$

$i := i + 1$
; $(\mathcal{L}, E, \mathcal{C}) : S_0$
$\{Invariant\}$
od
$\{Invariant \wedge ((i = n) \vee inserted)\}$
; $(\mathcal{L}, E, \mathcal{C}) : S_1$
$\{\mathcal{L}_0 \sqsubseteq_X \mathcal{L} \wedge CG(\mathcal{L}, E) \wedge X \in \mathcal{L}\}$

Statement S_0 considers the positioning of X with respect to C_i and finds possible children of X, while S_1 effectively inserts X into $(\mathcal{L}, E)$ and appropriately connects the new node to its children and also to its only parent. In terms of the invariant, by the end of the main loop, $\mathcal{L}$ will contain all intersections between X and original nodes of $\mathcal{L}_0$, while $\mathcal{C}$ will include all nodes to be connected to X as children.

Statement S_0 should find possible children of X in $\mathcal{L} \restriction C_i$ and even generate such children if they are not already there. As a result, the updated $\mathcal{L} \restriction C_i$ should include the *minimum* number of additional elements needed to preserve closedness under set intersection. To do this, determine whether X coincides exactly with C_i, whether it is a superset of C_i, whether it is a subset of C_i, or whether it is none of the foregoing. In each of these cases, statements (designated below as S_{00}, S_{01}, S_{02}, and S_{03}) that preserve the invariant will be executed.

The structure of statement S_0 is therefore as given below.

if $X = C_i \rightarrow \{X = C_i \wedge Invariant\}S_{00}$
[] $C_i \subset X \rightarrow \{C_i \subset X \wedge Invariant\}S_{01}$
[] $X \subset C_i \rightarrow \{X \subset C_i \wedge Invariant\}S_{02}$
[] $\neg((X = C_i) \vee (C_i \subset X) \vee (X \subset C_i)) \rightarrow$
$\{\neg((X = C_i) \vee (C_i \subset X) \vee (X \subset C_i)) \wedge Invariant\}S_{03}$
fi
$\{Invariant\}$

Arguments leading to the elaboration of each of the statements S_{00}, S_{01}, S_{02}, and S_{03} will now be made. It will be evident that the only nodes to be constructed are those necessary to preserve set intersection closedness. Thus, in the interests of conciseness, no further reasoning about the attainment of $\mathcal{L}_0 \sqsubseteq_X \mathcal{L}$ will be given. Even though the corresponding predicates are no longer reflected in the assertions below, it may easily be verified that they continue to hold.

3.3. The first two guards

Clearly, in the case of the first guard, nothing is to be achieved by considering more children of $\top_{\mathcal{L}}$: not only is the invariant preserved by doing nothing, but the postcondition of $insert_0((\mathcal{L}, E), X)$ is also already established. Statement S_{00} thus sets *inserted* to *true* so that the loop will terminate with the *inserted* flag signalling the fact that nothing more needs to be done to achieve the postcondition of $insert_0((\mathcal{L}, E), X)$.

$\{C_i = X \wedge Invariant\}$
$inserted := true$
$\{Invariant \wedge inserted = true\}$

When the second guard holds, every element of $\mathcal{L} \restriction C_i$ is a subset of X. Thus, no intersections have to be generated between X and the elements of $\mathcal{L} \restriction C_i$ to ensure that $\mathcal{L}$ remains a SICL after addition of X.

On the other hand, C_i is a child of $\top_{\mathcal{L}}$ in $\mathcal{L}_0$, but $C_i \subset X \subset \top_{\mathcal{L}}$. This implies that the following holds:

$$\nexists Y \in \mathcal{L} : X \subset Y \subset \top_{\mathcal{L}}$$

affirming that X is a child of $\top_{\mathcal{L}}$ in $\mathcal{L}$ and that the arc between these two nodes should be set up;

$$\nexists Y \in \mathcal{L} : C_i \subset Y \subset X^1$$

affirming that C_i is a child of X in $\mathcal{L}$ and that the corresponding arc should be set up; and

$$\exists Y \in \mathcal{L} : C_i \subset Y \subset \top_{\mathcal{L}}$$

[1] It is easy to check that this statement will also hold after all the children of the top node have been considered, making C_i a child of X in the final lattice.

indicating that C_i is no longer a child of $\top_{\mathfrak{L}}$ in $\mathfrak{L}$ and that the existing arc indicating a parent-child relationship between $\top_{\mathfrak{L}}$ and C_i should later be removed.

However, at this stage, X is not yet in $\mathfrak{L}$, and instead of creating and deleting these arcs, we merely include C_i in $\mathfrak{C}$, the set of possible children of X. The only statement associated with S_{01} is therefore as follows:

$\{C_i \subset X \wedge Invariant\}$
$\mathfrak{C} := \mathfrak{C} \cup \{C_i\}$
$\{Invariant\}$

It is easily verified that this statement preserves the $inv2$ part of the invariant, and since $\mathfrak{L}$ does not change, it also preserves the $inv1$ part.

3.4. The third guard

The third guard implies that the following holds:

$$(\mathfrak{L} \restriction C_i, E \restriction C_i) = (\mathfrak{L}_0 \restriction C_i, E_0 \restriction C_i) \wedge CG(\mathfrak{L} \restriction C_i, E \restriction C_i) \wedge X \subseteq \top_{\mathfrak{L} \restriction C_i}$$

This, of course, is the precondition of $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$. If a call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$ is made, the corresponding postcondition that will be met after the call, is:

$$(\mathfrak{L}_0 \restriction C_i \sqsubseteq_X \mathfrak{L} \restriction C_i) \wedge CG(\mathfrak{L} \restriction C_i, E \restriction C_i) \wedge (X \in \mathfrak{L} \restriction C_i)$$

We argue as follows to show that this call to $insert_0$ preserves $invariant$. Since the call to $insert_0$ establishes $X \in \mathfrak{L} \restriction C_i$, it is also the case that $X \in \mathfrak{L}$ holds. As a result, the second conjunct of $invariant$ is true—there is not need to do anything to establish $inv2$.

From the first conjunct of the above postcondition, it follows directly the last conjunct in $inv1$, namely $\mathfrak{L}_0 \sqsubseteq_X \mathfrak{L}$, is preserved as a result of this call. The question is whether the above postcondition preserves the first conjunct in $inv1$ of the invariant, namely $CG(\mathfrak{L}, E)$. To show this, we need to show that $CS(\mathfrak{L})$ holds, and that $E = \prec_{\mathfrak{L}}$ holds.

First, we show that $CS(\mathfrak{L})$ holds, that is, that $\mathfrak{L}$ contains all intersections involving new nodes. Let Y be an arbitrary newly introduced element of $\mathfrak{L} \restriction C_i$ and let Z be an arbitrary element of $\mathfrak{L}$. We need to show that $Y \cap Z \in \mathfrak{L}$.

Since $Y \subseteq X \subset C_i$, we have $Y \cap Z = Y \cap (Z \cap C_i)$. However, both nodes Y and $Z \cap C_i$ are part of the updated $\mathfrak{L} \restriction C_i$, whose closedness with respect to set intersection is assured by the postcondition of the recursive call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$. Consequently, $Y \cap (Z \cap C_i) \in \mathfrak{L} \restriction C_i$ and therefore also $Y \cap Z \in \mathfrak{L} \restriction C_i$. Thus, $CS(\mathfrak{L})$.

To see that the $E = \prec_{\mathfrak{L}}$ requirement also holds, note that $E \restriction C_i = \prec_{\mathfrak{L} \restriction C_i}$ by the postcondition of the call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$. Hence, we only need to consider arcs connecting nodes in $\mathfrak{L} \restriction C_i$, on the one hand, and nodes outside $\mathfrak{L} \restriction C_i$, on the other hand.

Consider two nodes in $\mathfrak{L}$, say Y and Z, after a call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$, where $Y \notin \mathfrak{L} \restriction C_i$ and $Z \in \mathfrak{L} \restriction C_i$. To affirm that Y and Z are in the right relationship to one another, note that the only possible child of Y in $\mathfrak{L} \restriction C_i$ is $Y \cap C_i$ and consider the following cases:

$Z = Y \cap C_i$: In this case, node Z existed prior to the recursive call. The call does not change the relationship between Y and Z in $\mathfrak{L}$, and the arc between these nodes, whether or not it existed, is not affected.

$Z \neq Y \cap C_i$: Since $E = \prec_{\mathfrak{L}}$ before the call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$, there was no arc between Y and Z at that time. There will not be an arc connecting Y and Z after the call, since the call does not set up arcs outside of $L \restriction C_i$.

Hence, if $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$ is called, then nothing more has to be done to ensure that $E = \prec_{\mathfrak{L}}$ and, consequently, the call preserves the loop's invariant.

Note though, that after this call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$ has executed, X is correctly inserted into $\mathfrak{L}$, i.e., the postcondition of $insert_0((\mathfrak{L}, E), X)$ has been achieved. As in the case of the first guard, there is no point in executing further loop iterations—the loop can therefore just as well be terminated, and this is done by setting $inserted$ to $true$. We thus have the following for S_{02}:

$\{X \subset C_i \wedge Invariant\}$
$\{(\mathfrak{L} \restriction C_i, E \restriction C_i) = (\mathfrak{L}_0 \restriction C_i, E_0 \restriction C_i) \wedge CG(\mathfrak{L} \restriction C_i, E \restriction C_i) \wedge X \subseteq \top_{\mathfrak{L} \restriction C_i} = C_i\}$
$insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$
$\{\mathfrak{L}_0 \restriction C_i \sqsubseteq_X \mathfrak{L} \restriction C_i \wedge CG(\mathfrak{L} \restriction C_i, E \restriction C_i) \wedge X \in \mathfrak{L}\}$
; $inserted := true$
$\{Invariant \wedge inserted\}$

As an aside, note that no harm would be done if the loop were allowed to execute more iterations to check the remaining children of $\top_{\mathfrak{L}}$. It can easily be verified that attempts would then be made, via the fourth guard, to insert subsets of X into $\mathfrak{L}$, but these nodes would already exist in $\mathfrak{L}$, and so $\mathfrak{L}$ would not change.

3.5. The fourth guard

When the fourth guard is true then $X \cap C_i$ is a *proper* subset (possibly empty) of both C_i and X. This means that, when X is inserted into $\mathfrak{L}$, $X \cap C_i$ also has to be inserted into $\mathfrak{L}$ if it is not already there in order to assure set intersection closedness of $\mathfrak{L}$. A call to $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X \cap C_i)$ installs $X \cap C_i$ into $(\mathfrak{L} \restriction C_i, E \restriction C_i)$. Inspection will show that the precondition of $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X \cap C_i)$ justifies this recursive call. Its postcondition guarantees the required properties for the resulting $(\mathfrak{L} \restriction C_i, E \restriction C_i)$.

Arguments similar to those made previously (in respect of the third guard's statements) can be made to prove that these actions result in the set intersection closedness and proper connectedness of $(\mathfrak{L} \restriction C_i, E \restriction C_i)$, leaving $inv1$ intact. To attain $inv2$, set $\mathfrak{C}$ has to be updated with $X \cap C_i$. S_{03} is therefore refined as follows:

$\{\neg((X = C_i) \vee (C_i \subset X) \vee (X \subset C_i)) \wedge Invariant\}$
$insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X \cap C_i)$
$\{inv1\}$
$; \mathfrak{C} := \mathfrak{C} \cup \{X \cap C_i\}$
$\{inv2\}$
$\{Invariant\}$

In the implementation of the algorithm, the call $insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X \cap C_i)$ must return the node corresponding to $X \cap C_i$, either existing in $\mathfrak{L}_0$ or newly created in $\mathfrak{L}$, so that this node could be added to $\mathfrak{C}$.

3.6. Updating arcs

After the main loop terminates, $\mathfrak{L}$ already contains all intersections of X and elements from $\mathfrak{L}_0$ with the possible exception of X itself (if the value of *inserted* is *false*). These elements are even properly connected. All that remains (if anything) is to create node X and connect it to its parent and children.

If the value of *inserted* is *false* and we have to create node X, then neither of the first and third guards has ever been activated. In other words, X is not a subset of any child of the top element. Hence, the top element is the only parent of X.

Children of X in the final lattice, $\mathfrak{L}$, have to be the maximal (with respect to set containment) subsets of X that are in $\mathfrak{L}$. Obviously, every such subset is the intersection of X and a child, C, of the top element in $\mathfrak{L}_0$; of course, this intersection may be equal to C (in which case, C is a child of X). The set $\mathfrak{C}$ consists of all such intersections obtained via the second and fourth guards. It can easily be verified that not every element of $\mathfrak{C}$ need necessarily be a *maximal* subset of X in $\mathfrak{L}$: some may be proper subsets of others. Only the maximal subsets of X in $\mathfrak{C}$ have to be identified and connected to X. Assume that $getMax(\mathfrak{C})$ is a function that returns as a set the maximal sets in $\mathfrak{C}$—i.e., sets in $\mathfrak{C}$ that are not contained in any other sets in $\mathfrak{C}$.

To finish the insertion of X into $(\mathfrak{L}, E)$, we therefore have to create the node X and connect it to maximal subsets of X in $\mathfrak{C}$. If these children of X were linked to $\top_{\mathfrak{L}}$, the corresponding arc must be removed. Statement S_1 can be elaborated as follows.

$\{Invariant\}$
if $\neg inserted \rightarrow$
  $\{\forall Y \in \mathfrak{L} : X \cap Y \in \mathfrak{L}\}$
  $\mathfrak{L} := \mathfrak{L} \cup \{X\}$
  $; E := E \cup \{(\top_{\mathfrak{L}}, X)\}$
  ; **for** $C : getMax(\mathfrak{C}) \rightarrow$
      $E := (E \setminus \{(\top_{\mathfrak{L}}, C)\}) \cup \{(X, C)\}$
  rof
[] $inserted \rightarrow skip$
fi
$\{Invariant \wedge X \in \mathfrak{L}\}$

3.7. The completed algorithm

The resulting incremental algorithm to install X into $\mathfrak{L}$ is given below. To insert node X into $\mathfrak{L}$, the call $insert_0((\mathfrak{L}, E), X)$ has to be made.

Algorithm 1.
proc $insert_0((\mathfrak{L}, E), X)$
pre : $\{(\mathfrak{L}, E) = (\mathfrak{L}_0, E_0) \wedge CG(\mathfrak{L}, E) \wedge X \subseteq \top_{\mathfrak{L}}\}$
$i, n, \mathfrak{C}, inserted := 0, |E(\top_{\mathfrak{L}})|, \emptyset, false$
$\{Invariant\}$
;**do** $(i \neq n \wedge \neg inserted) \rightarrow$
$i := i + 1$
if $X = C_i \rightarrow inserted := true$
[] $C_i \subset X \rightarrow \mathfrak{C} := \mathfrak{C} \cup \{C_i\}$
[] $X \subset C_i \rightarrow insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X)$
; $inserted := true$
[] $\neg((X = C_i) \vee (C_i \subset X) \vee (X \subset C_i)) \rightarrow insert_0((\mathfrak{L} \restriction C_i, E \restriction C_i), X \cap C_i)$
; $\mathfrak{C} := \mathfrak{C} \cup \{X \cap C_i\}$
fi
$\{Invariant\}$
od
$\{Invariant \wedge ((i = n) \vee inserted)\}$
if $\neg inserted \rightarrow$
$\{\forall Y \in \mathfrak{L} : X \cap Y \in \mathfrak{L}\}$
$\mathfrak{L} := \mathfrak{L} \cup \{X\}$
; $E := E \cup \{(\top_{\mathfrak{L}}, X)\}$
; **for** $C : getMax(\mathfrak{C}) \rightarrow$
$E := (E \setminus \{(\top_{\mathfrak{L}}, C)\}) \cup \{(X, C)\}$
rof
[] $inserted \rightarrow$ skip
fi
post : $\{\mathfrak{L}_0 \sqsubseteq_X \mathfrak{L} \wedge CG(\mathfrak{L}, E) \wedge X \in \mathfrak{L}\}$

The algorithm starts with the top node of the lattice and the set X that has to be inserted. The insertion is carried out by exploring all the children of the top node. If one such child, C_i, is equal to X, then nothing more has to be done, as X is already in the lattice. If X is a subset of C_i, then all new nodes including X will appear below C_i; hence, a recursive call is made to insert X under C_i. If, on the contrary, C_i is a subset of X, it must be a descendant of X in the updated graph; therefore, it is added to $\mathfrak{C}$, the set of potential children of X. Lastly, if X and C_i are incomparable, we must make sure that their intersection is in the lattice; a recursive call is made to insert $X \cap C_i$ under C_i and $X \cap C_i$ is added to $\mathfrak{C}$, as $X \cap C_i$ must be a descendant of X in the updated graph. When all children of the top node have been processed in this way, the lattice contains intersections of X with all elements of the initial lattice with the only possible exception of X itself. If X is not yet in the lattice, the algorithm adds the corresponding node connecting it to the top node from above and to the maximal subsets in $\mathfrak{C}$ from below. In so doing, the algorithm also removes edges connecting the top node to the children of X if such edges were present in the old graph.

4. Two variations

The derivation of the above algorithm was based on the intuitively appealing invariant that preserved the $CG(\mathfrak{L}, E)$ property of the evolving graph $(\mathfrak{L}, E)$. It serves as a basis for deriving alternative versions of the algorithm, in particular, by strengthening the invariant. The two versions are examined below. An analysis of the guards in the algorithm provides a starting point for the discussion.

4.1. Guard mutual exclusivity

By Proposition 1, the children of $\top_{\mathfrak{L}}$ cannot be equal to, supersets or subsets of one another. Thus, the first three guards are *mutually exclusive* over the outer loop: if one guard applies to *one* child of $\top_{\mathfrak{L}}$, neither of the other two can apply to *any* of the children of $\top_{\mathfrak{L}}$. The first guard can only be true for at most one child of $\top_{\mathfrak{L}}$, while the second and third can be true for several. (Of course, in some contexts, none of these guards may ever be true.)

Thus, if the first guard is selected in an iteration of the outer loop, then only the fourth guard could have been selected in prior iterations. The recursive call in the body of the fourth guarded command could not have inserted new nodes into the lattice, since all relevant subsets of node X would be present already in the lattice by virtue of the set-intersection closedness requirements. Thus, the work done in all previous iterations would be redundant, merely establishing that some subset of X is already in the lattice.

Similarly, if the third guard is selected sometime after the first iteration of the outer loop, then the fourth guard would have been selected in prior iterations. However, the recursive call in the body of the fourth guarded command would either

recurse to a point where it established that some subset of X is already in the lattice, or it would insert a subset of X into the lattice that would, at any rate, have been installed by the body of the third guard in the later iteration.

One line of optimization is therefore to ensure that the children of $\top_{\mathfrak{L}}$ are offered by the outer loop in an order that guarantees that, where applicable, either the first or the third guard is selected prior to the selection of the fourth guard. This would come at the cost of searching for a child of $\top_{\mathfrak{L}}$, say C, such that $X \subseteq C$. Such a search can be done in $O(mn)$ time, where $O(m)$ is the cost of the set-containment test (i.e., m can be regarded as the size of the alphabet, $m = |A|$) and n is the number of child nodes of $\top_{\mathfrak{L}}$.

4.2. Strengthening the invariant

Another way of implementing a similar optimization is now described. It is possible to remove the first guard from the loop altogether and to do its work ahead of the loop. This amounts to strengthening the invariant to require that $\forall C \in E(\top_{\mathfrak{L}}) : C \neq X$. The main loop would be preceded by a new one that scans the children of $\top_{\mathfrak{L}}$ for equality with X and only falls into the main loop if equality is not found. This would eliminate the redundant work carried out when the fourth guard is selected at the cost of $|E(T)|$ set-equality tests per recursive call, where T is the top element of the current subgraph.

Assuming that $\forall C \in E(\top_{\mathfrak{L}}) : C \neq X$, yet a further optimization is to remove the third guard from the main loop, and to do *its* work ahead of the loop. This means preceding the loop with a focussed search over the children of C and their descendants until a node is reached that is a superset of X, but that does not have a child that is a superset of X. Such a search therefore identifies the uniquely determined node in $\mathfrak{L}$ which is to become the parent of X. Call this node P. The original loop, with the third guard removed, now has a strengthened invariant that incorporates $X \prec P$ as a conjunct—a predicate that asserts that X must be a child of P in the resulting cover relation. Again, the loop iterates over the children of P, installing X as a sibling.

The search for the parent of X proceeds as follows. The variable P is initialized by $\top_{\mathfrak{L}}$, and the children of P are examined. If no child, C, of P is found such that $X \subseteq C$, then the search terminates with P indicated as the parent of X. Otherwise, if such a C is found, then P is reset to C, and the children of the new P are probed in like manner, etc. In the extreme case, the search would end up offering $\bot_{\mathfrak{L}}$ as a parent of X. Clearly, then, the search does not need to probe all nodes in $(\mathfrak{L}, E)$, but only those found on the search path that it follows from parent to children to grandchildren of one (and only one) child, to great-grandchildren of one (and only one) grandchild, etc. Guard mutual exclusivity points to an additional enhancement when checking each child, C of P: the loop could terminate immediately if it is found that $C \subseteq X$, returning the parent of C as the required parent of X. Below, a method called $getP((\mathfrak{L}, E), X)$ that carries out such a search is assumed to be available.

Although complexity issues are more comprehensively treated in Section 5, it is instructive to note that $getP((\mathfrak{L}, E), X)$ effectively follows the same search path that is followed via the recursions through guard three in the original algorithm. Nevertheless, such a strengthening of the invariant and its resulting elimination of guards one and three will lead, on average, to efficiency gains. By locating the parent of X beforehand, the insertion of subsets of X that are initiated by the recursive calls in the fourth guarded command will tend to be more efficient. Moreover, reasoning about the proper links in the graph is simplified.

4.3. A refactored algorithm: Invariant strengthened

The algorithm below, $insert_1((\mathfrak{L}, E), X)$, is a refactoring of $insert_0((\mathfrak{L}, E), X)$ in which the invariant has been strengthened in the manner outlined above. It does not rely on children of $\top_{\mathfrak{L}}$ in sorted order.

The conditional statement of Algorithm 1 is used in which the bodies of the second and fourth guards are left intact, but the first and third guards disappear. Their work is carried out by invoking $P := getP((\mathfrak{L}, E), X)$ before entering the main loop. The postcondition of this statement is that X is determined to either correspond identically to P or X is destined to be installed in $\mathfrak{L}$ as a child of P.

Note that whereas previously C_i was used to denote the children of $\top_{\mathfrak{L}}$, in the algorithm below, C_i refers to children of P.

Algorithm 2.

proc $insert_1((\mathfrak{L}, E), X)$
pre : $\{(\mathfrak{L}, E) = (\mathfrak{L}_0, E_0) \wedge CG(\mathfrak{L}, E) \wedge X \subseteq \top_{\mathfrak{L}}\}$
$P := getP((\mathfrak{L}, E), X)$
$\{(P = X) \vee X \prec P)\}$
; **if** $P = X \rightarrow$ skip
[] $P \neq X \rightarrow$
  $i, n, \mathcal{C} := 0, |E_0(P)|, \emptyset$
  $\{Invariant \wedge X \prec P\}$
  ;**do** $(i \neq n) \rightarrow$
    $i := i + 1$
    if $C_i \subset X \rightarrow \mathcal{C} := \mathcal{C} \cup \{C_i\}$
    [] $\neg(C_i \subset X) \rightarrow insert_1((\mathfrak{L} \restriction C_i, E \restriction C_i), X \cap C_i)$

```
        ; 𝒞 : = 𝒞 ∪ {X ∩ C_i}
      fi
      {Invariant ∧ X ≺ P}
  od
  {Invariant ∧ X ≺ P ∧ (i = n)}
  ℒ : = ℒ ∪ {X}
  ; E : = E ∪ {(P, X)}
  ; for C : getMax(𝒞) →
        E : = (E \ {(P, C)}) ∪ {(X, C)}
  rof
fi
post : {ℒ_0 ⊑_X ℒ ∧ CG(ℒ, E) ∧ X ∈ ℒ}
```

Note that several variants of Algorithms 1 and 2 are possible. It will be convenient to rely on Algorithm 2 to analyze the complexity of this class of algorithms. Before doing so, another variant is derived in the next section. It splits the set of possible children of X, $\mathcal{C}$, into two parts.

4.4. Splitting the children into two groups

A further refinement is to maintain *two* separate sets, say $\mathcal{C}$ and $\mathcal{D}$, of children of X. The first, $\mathcal{C}$, stores nodes that have definitively been established to be children of X. They correspond to children of X identified by the second guard. The second set, $\mathcal{D}$, stores nodes that can only tentatively claim to be children of X. They are indeed subsets of X, but they may also turn out to be subsets of X's children. They correspond to intersections generated by the fourth guard.

Below, methods called $update\mathcal{C}(C)$ and $update\mathcal{D}(C)$ are assumed to be available. The first inserts a child of X, C, into set $\mathcal{C}$ and eliminates from set $\mathcal{D}$ any elements that, as a consequence of this new insertion into set $\mathcal{C}$, can no longer be regarded as children of X. The second inserts C as a potential child of X into set $\mathcal{D}$, provided that C is not a subset of any element in $\mathcal{C}$ or $\mathcal{D}$. $update\mathcal{D}(C)$ also eliminates from set $\mathcal{D}$ any elements that, as a consequence of this new insertion in set $\mathcal{D}$, can no longer be regarded as children of X. Thus, $update\mathcal{C}$ involves a loop that iterates over elements of $\mathcal{D}$, while $update\mathcal{D}$ requires a loop that iterates over elements of $\mathcal{C} \cup \mathcal{D}$. The improvement, as compared with using *getMax* is that we do not have to check if sets in $\mathcal{C}$ are subsets of those in $\mathcal{D}$. This may provide average case improvements, but does not alter the essential worst-case behavior of the initial algorithm.

Since we now have two sets for keeping track of children of X, the corresponding conjuncts in the invariant must be modified:

$$
\begin{aligned}
&Invariant_2 \quad \triangleq\ inv1 \wedge (X \notin \mathcal{L} \Rightarrow inv2_2)\\
&\text{where } inv2_2 \ \triangleq\ \forall Y, Z \in \mathcal{C} \cup \mathcal{D} : ((Y \subset X) \wedge (Y \not\subset Z)) \wedge\\
&\qquad\qquad \forall Y \in \mathcal{M}_i : (Y \subset X \Rightarrow \exists Z \in \mathcal{C} \cup \mathcal{D} : Y \subseteq Z) \wedge\\
&\qquad\qquad CS(\mathcal{M}_i \cup \{X\})
\end{aligned}
$$

4.5. A refactored algorithm: Children split into two groups

The algorithm below, $insert_2((\mathcal{L}, E), X)$, is a refactoring of $insert_1((\mathcal{L}, E), X)$ in which the set $\mathcal{C}$ has been split in two parts in the manner outlined above. After the main loop, the set $\mathcal{C} \cup \mathcal{D}$ contains exactly all children of X in $\mathcal{L}$. Thus, there is no need to use the *getMax* function.

Algorithm 3.

```
proc insert_2((ℒ, E), X)
pre : {(ℒ, E) = (ℒ_0, E_0) ∧ CG(ℒ, E) ∧ X ⊆ ⊤_ℒ}
P : = getP((ℒ, E), X)
{(P = X) ∨ X ≺ P)}
; if P = X → skip
[]  P ≠ X →
    i, n, 𝒞, 𝒟 : = 0, |E_0(P)|, ∅, ∅
    {Invariant_2 ∧ X ≺ P}
    ;do (i ≠ n) →
        i : = i + 1
        if C_i ⊂ X → update𝒞(C_i)
        [] ¬(C_i ⊂ X) → insert_2((ℒ ↑ C_i, E ↑ C_i), X)
```

```
        ; updateD(X ∩ C_i)
      fi
      {Invariant_2 ∧ X ≺ P}
  od
  {Invariant_2 ∧ X ≺ P ∧ (i = n)}
  L : = L ∪ {X}
  ; E : = E ∪ {(P, X)}
  {getMax(C ∪ D) = C ∪ D}
  ; for C : C ∪ D →
        E : = (E \ {(P, C)}) ∪ {(X, C)}
  rof
fi
post : {L_0 ⊑_X L ∧ CG(L, E) ∧ X ∈ L}
```

Other refactorings are of course possible. For example, if $C_i \cap X \subset \bot_{\mathcal{L}}$ then a recursive call to install $C_i \cap X$ is unnecessary—the node may simply be installed below the bottom element. Similar shortcircuiting is possible if $C_i \cap X = \bot_{\mathcal{L}}$. However, details of such refactorings are left for future research.

5. Efficiency

In this section, we derive a theoretical worst case upper bound complexity of the algorithm proposed above and also discuss the algorithm's practical performance. In so doing, we will use formal concept analysis (FCA) [2] notation to align our complexity arguments with those used in deriving complexity estimates for other FCA algorithms [18].

Concept lattices (also referred to as Galois lattices or formal concept lattices) are used in machine learning, data and text mining, and other domains. A recent and comprehensive introduction to concept lattices and, especially, its applications in computer science is provided in [3]. Essentially, a concept lattice is built from a set, M, of attributes, subsets of which characterize each element in a set, G, of objects. Each node in a concept lattice is characterized by two sets: a subset of G, say, A, and a subset of M, say, B, such that A consists of all those objects that possess (at least) all the attributes in B and B is exactly the intersection of objects from A. Such a pair (A, B) is called a concept.

The concept (A, B) is said to have an extent of A and an intent of B. The sets of extents and intents of a concept lattice are closure systems on G and M, respectively. Thus, if two subsets of G (or M), say, A and B, are the extents (respectively, intents) of nodes in a concept lattice, then $A \cap B$ also has to be the extent (intent) of some node in the lattice.

By FCA convention, concepts are assumed to be ordered by set-containment of their *intents*. Thus, for concepts (A, B) and (C, D), we have $(A, B) \leq (C, D)$ iff $B \subseteq D$. (It can easily be shown that, alternatively, $(A, B) \leq (C, D)$ iff $C \subseteq A$.) The relation $\prec$ is defined as above.

One may say that the lattice of concept extents and the lattice of concept intents are actually SICLs. Hence, one way of constructing a graph data-structure for a concept lattice by using one of the algorithms discussed in the paper, making slight adaptations for keeping track of the extent if a SICL based on intents is constructed, or vice-versa if a SICL based on extents is constructed.

For at least the last two decades, many different algorithms for constructing concept lattices have been proposed, e.g., [21–23]. See [18] for a review and comparison. To date, the algorithm reported in [24] appears to have the best theoretical worst-case complexity estimate, namely $O(|L|(|G| + |M|)|G|)$, where $|L|$ is the number of concepts in the resulting lattice. Note that this is the complexity of constructing the lattice from scratch rather than updating an existing lattice by inserting a new object.

We shall now derive the worst-case complexity estimate for constructing the lattice of concept intents and its cover relation using Algorithm 2. Such construction starts by initializing the graph structure with a single node, M, which forms the top node of a singleton lattice, and remains the top node in all subsequently generated lattices. The construction then proceeds by successively calling Algorithm 2 for every object from G, where the set to be inserted into the lattice corresponds to the set of attributes that characterize the particular object, and the lattice generated by each call is used as the starting lattice for the next call. Although our algorithm generates only concept intents, it can be easily adapted to generate extent–intent pairs, i.e., concepts.

Reasoning about complexity is easier if one has in mind both extents and intents. However, when speaking about children of a concept, we assume that concepts are ordered by set-containment of their *intents*: for concepts (A, B) and (C, D), we have $(A, B) \leq (C, D)$ iff $B \subseteq D$ (equivalently, $C \subseteq A$). The relation $\prec$ is defined as above.

Definition 5.1. A *route* in the cover relation of the concept lattice is a sequence of $n > 0$ distinct concepts $(A_1, B_1), \ldots, (A_n, B_n)$, such that $(n > 1) \Rightarrow B_n \subset B_1$, and:

$$\forall j \in (1, n] \exists i \in [1, j) \forall k \in (i, j] : (A_k, B_k) \prec (A_i, B_i)$$

Thus, the immediate predecessor of concept (A_j, B_j) in a route is either its parent in the cover relation, or one of its siblings in the cover relation. In the latter case, the common parent of these two concepts must occur earlier in the route than both these concepts.

The length of a route (i.e., the number of elements in the sequence) cannot be greater than $|G|^2$. To see that this is the case, we consider two types of subsequences of a route: a *chain* and a *segment*.

By a chain of the route, we mean a subsequence of route nodes that are parents of some other route node. Note that the extent of each concept in a chain contains all objects in the extent of the preceding concept in the chain and, at least, one additional object. The length of the chain is therefore bounded by $|G|$. To be more precise, the length of a chain corresponding to the route $(A_1, B_1), \ldots, (A_n, B_n)$ is bounded by $|A_n| - |A_1| + 1$. However, for the present purposes, this more precise bound will not be used.[2]

We now define a segment of a route to be a subsequence of the route whose members have the same parent. Thus, each concept of a chain in a route marks the end of a segment of the route. Concepts within the same segment are children of the last concept of the preceding segment. They are thus siblings of one another. (Note, however, that there may be other siblings that are not included in the route and that are therefore not considered to be part of the segment.) The extent of every child of a concept (A_i, B_i) contains an object from $G \setminus A_i$, and every such object is contained in the extent of at most one child of (A_i, B_i). (If an object from $G \setminus A_i$ was contained in the extent of more than one child of (A_i, B_i), the intersection of the extents of these siblings would be a superset of A_i—which contradicts the properties of a concept lattice.) Therefore, the concept (A_i, B_i) has at most $|G| - |A_i|$ children. The number of concepts in a route is therefore bound by $|G|^2$.

We call a route *optimal* if the intent of only the last concept of each segment is a superset of the intent of the last concept of the route. The size of a segment of an optimal route $(A_1, B_1), \ldots, (A_n, B_n)$ is bounded by $|G| - |A_n| + 1$, as any extent that is a subset of A_n has at most $|G| - |A_n|$ child extents that are not subsets of A_n. Note that $|A_i| < |A_n|$ for every $i < n$. Therefore, segments in an optimal route are generally shorter than in a non-optimal route ($|G| - |A_n| \leq |G| - |A_i|$).

The procedure $getP(X)$ essentially visits nodes along an optimal route. Thus, the number of nodes it visits is bound by $|G|^2$. If the concept with the intent equal to X is already in the lattice, then $getP(X)$ reaches that node starting from $\top_{\mathcal{L}}$ (or, in the case of a recursive call, starting from the top concept of a corresponding sublattice) and following an optimal route. Even if X is not yet in the lattice, $getP(X)$ still follows an optimal route, visiting no more than $|G|^2$ nodes. At each step, $getP(X)$ has to spend at most $O(|M|)$ time to check whether X is a subset of the current intent. Thus, $getP(X)$ has a worst-case upper bound complexity $O(|G|^2|M|)$.

The bound $O(|G|^2|M|)$ is not sharp. In fact, there is no lattice where the intent-based search of a concept would involve examination of $|G|^2$ intents. Indeed, the number of optimal route segments and their lengths are in an inverse relationship: when one of them approaches the number of objects, the other tends to 1. A concept whose extent is C will be found after at most $(|C| + 1)(|G| - |C| + 1)$ steps. Besides, the procedure $getP(X)$ usually starts with a concept quite close to the one to be found, which significantly narrows down the search space. It is also worth noting that $|G|$ in the complexity estimate for the procedure $getP(X)$ refers to the number of objects processed at the time of the procedure call, rather than to the total number of objects to be inserted into the lattice.

The overall complexity of the algorithm depends on the total number of invocations of the $insert_1((\mathcal{L}, E), X)$ function. The same intent can be passed as a parameter of $insert_1((\mathcal{L}, E), X)$ several times, but, clearly, the execution of the algorithm will go past the call to $getP(X)$ at most once for every intent. Thus, for convenience, we may assume that $getP(X)$ is called before the invocation of the $insert_1((\mathcal{L}, E), X)$ function, which, in its turn, is called only if the returned set P is different from X. In this setting, $insert_1((\mathcal{L}, E), X)$ would be called at most once for every intent of the lattice through the insertion of all objects (possible exceptions are object intents, but they are safe to ignore).

Taking into account the above calculations, as well as the fact that the size of the set $\mathcal{C}$ is not greater than $|G|$, we estimate the worst-case time complexity of a single invocation of the Algorithm 2 without the recursive call as $O(|G|^3|M|)$. Hence, we may claim that the complexity of inserting one object with Algorithm 2 is $O(|G|^3|M|(|\mathcal{L}| - |\mathcal{L}_0|)$, where $|\mathcal{L}| - |\mathcal{L}_0|$ is the number of newly generated intents. The complexity of constructing the cover relation for the entire lattice from scratch is then $O(|G|^3|M||\mathcal{L}|)$.

The complexity estimate derived above is somewhat worse than the estimate of other competing algorithms [18], but this is mainly due to the overestimation of the complexity of the procedure $getP(X)$. We now show that, for some especially large lattices of a certain kind, a better estimate can be stated.

Consider the case when the objects from G are all subsets of size $|M|-1$ of the attribute set M, i.e., $G = \{M \setminus \{m\} \mid m \in M\}$. Such data gives rise to a so-called Boolean lattice with $2^{|M|}$ elements: every subset of M is an intersection of some objects from G and, consequently, a part of $\mathcal{L}$, the resulting lattice. In a certain sense, this is when the worst case is realized (for the given $|M|$), since the lattice contains the maximal possible number of nodes.

Fig. 2 shows such a Boolean lattice, namely, the lattice of all subsets of $\Sigma = \{a, b, c, d\}$. By marking arcs in different styles, the figure indicates how successive calls to $insert_1((\mathcal{L}, E), X)$ could grow the lattice from a starting SICL of `abcd`. Inserting `abc` yields a SICL with $2^1 = 2$ nodes; inserting `abd` in next yields a SICL with $2^2 = 4$ nodes; inserting `acd` in next yields a SICL of $2^3 = 8$ nodes; and finally inserting `bcd` yields the full Boolean lattice of $2^4 = 16$ nodes.

[2] Note that a dual argument based on the number of attributes in the intent of a chain's nodes, leads to the conclusion that the length of the chain is also bound by $|M|$, and more precisely, by $|B_1| - |B_n| + 1$. Thus, and even more precise bound on the length of a chain corresponding to route $(A_1, B_1), \ldots, (A_n, B_n)$ is given by $min((|A_n| - |A_1|), (|B_1| - |B_n|)) + 1$.

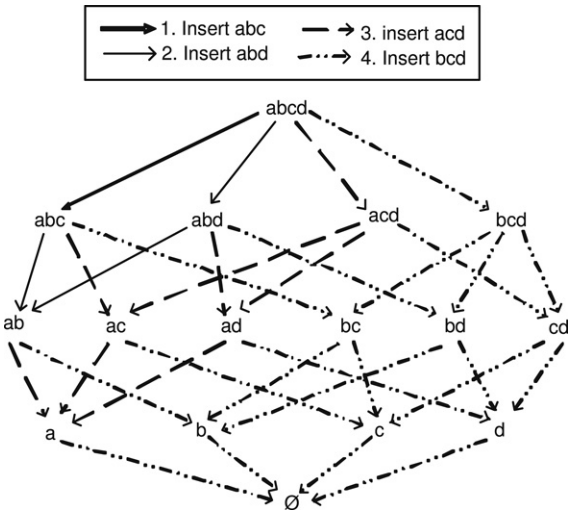

Fig. 2. Boolean lattice.

Thus, inserting each new object $g \in G$ into a partially constructed lattice $\mathfrak{L}_0$ would result in doubling the number of nodes in the lattice: no intent in $\mathfrak{L}_0$ would be a subset of this object g, and no two intents would generate the same intersection with g. Consider how $getP(X)$ works in the context of a call to $insert_1((\mathfrak{L} \restriction C, E \restriction C), X)$.

(1) If this is the first time $insert_1$ is called for the given $(\mathfrak{L} \restriction C, E \restriction C)$, then C has no children that are supersets of X, and $getP(X)$ returns C.
(2) Otherwise, C already has a child X, and it will be returned by $getP(X)$.

In both cases, $getP(X)$ searches only among the children of the top node of the current subgraph never going further down the graph. Thus, the complexity of $getP(X)$ in this case is only $O(|G||M|)$ and the complexity of constructing a Boolean lattice from scratch by Algorithm 2 is $O(|G|^2|M||\mathfrak{L}|)$.

In general, for dense data, i.e., for data with a large number of attributes per object (relative to $|M|$) and, consequently, a large number of concepts (relative to the theoretical maximum of $2^{|M|}$), the $getP(X)$ function works quite fast and makes far fewer than $|G|^2$ steps, since it rarely has to go deep down into the lattice. On sparse data, the $getP(X)$ function can go a long way down (though again never as many as $|G|^2$ steps), but then it means that a large portion of concepts is excluded from further consideration and no intersection is computed between their intents and the new object intent being inserted. This explains good performance of the algorithm on both dense and sparse datasets [25], which is an unusual feature for most known algorithms.

The foregoing emphasizes that the worst-case theoretical complexity is in this case only a rather rough indicator of performance. It should be noted that it is indeed important that the algorithm is linear in terms of $|\mathfrak{L}|$. However, $|\mathfrak{L}|$ totally dominates the other factors ($|G|$ and $|M|$) in the worst-case expressions that have been derived: the latter are in fact exponentially smaller than $|\mathfrak{L}|$ in the worst case. As a result, when evaluating the performance of algorithms in practice, one cannot be strongly guided by the polynomial power of these factors within the worst-case complexity estimate.

Empirically, the experiments in [25] comparing the performance of several most efficient (according to [18]) or most popular lattice construction algorithms: **Norris** [26], **Ganter** (a.k.a. NextClosure) [27], a version of **Bordat** [28] from [18], **Godin** [21], and **Nourine** [24], gave the following picture.[3] It was found that an algorithm called *AddIntent* (first implemented in 1996 in the context of so-called compressed pseudo-lattices [29], first described in [30] and presented in [25] in the form very close to Algorithm 2) generally outperformed the others, except for two scenarios where it was ranked second. Most experiments were based on randomly generated data, or on constructing Boolean lattices of various sizes. However, when four real-world data-sets taken from the UCI repository [31] were used, *AddIntent* dramatically outperformed the other algorithms as seen in Fig. 3. SPECT (Single Proton Emission Computed Tomography) is a real dataset that contains 267 objects and 23 attributes, generating a lattice with 21 550 concepts. The remaining datasets (Breast Cancer,[4] Wisconsin Breast Cancer, and Solar Flare databases) are given in the form of many-valued tables and the QuDA program [32] was used to transform vector representations of objects in these datasets into attribute sets.

[3] Algorithms were implemented in C++ on the same codesbase. Tests were performed on a Pentium 4–2 GHz computer with 1 GB RAM running under Windows XP.

[4] This breast cancer domain was obtained from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data.

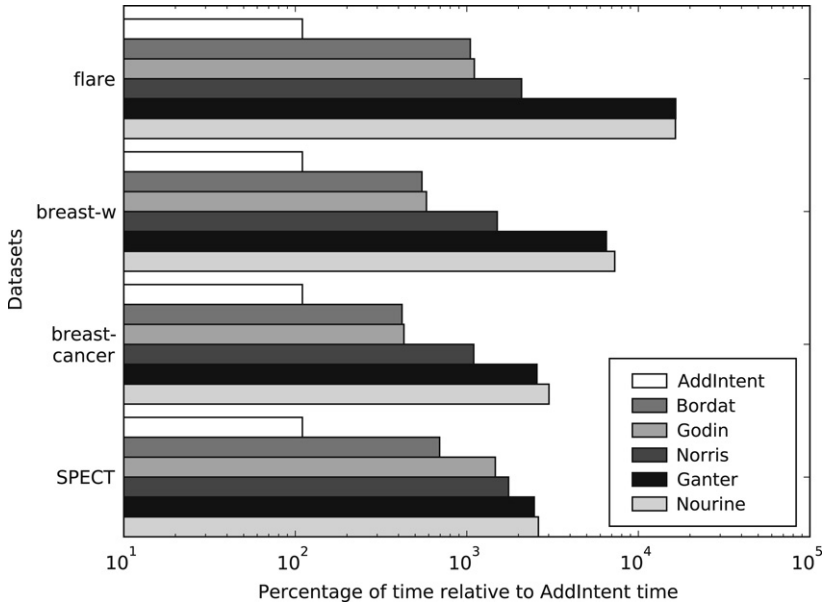

Fig. 3. Performance of *AddIntent* and other algorithms using empirical data (based on data from [25]).

6. Conclusion

Correctness-by-construction argumentation led naturally to the evolution of Algorithm 1, which turned out to be an underlying root in a taxonomy of lattice construction algorithms: Algorithms 2 and 3 result from strengthening the invariant and introducing some other changes. Theoretical complexity of the algorithms has been studied, and empirical evidence has been adduced for the improved efficiency of these algorithms in relation to rivals. Moreover, there would appear to be scope for further refinement of these algorithms. In particular, when inserting an attribute set, the current version of the algorithm can visit an existing node of the lattice more than once, since there may be more than one path to it from the top node. We are going to investigate the possibility to replace the graph corresponding to the cover relation by its spanning tree as in the version of Bordat's algorithm described in [18], thus, avoiding repetitive processing of the same node within a single iteration. This is likely to result in a better theoretical worst-case complexity as well.

To the best of our knowledge, the most closely related lattice construction algorithm among those ever published is the one from [33] (designated Algorithm 5 in the text). It uses a similar *getP*(X)-style methodology, but relies on a stack and tries as utility data structures facilitating lookup of concepts (as opposed to our usage of recursion and the cover relation itself). Since both algorithms follow the same main strategy, one can expect that their behavior is also similar, in particular, as compared to the behavior of other algorithms. A deeper comparison may reveal potential trade-offs between specific issues in which the algorithms differ and may suggest further implementation improvements.

Acknowledgements

The authors would like to thank anonymous reviewers for their valuable comments and suggestions. We are particularly grateful to the reviewer whose suggestions lead to the formulation of Definition 2.3.

References

[1] R. Wille, Restructuring lattice theory: An approach based on hierarchies of concepts., in: R.I. (Ed.), Ordered Sets, Reidel, 1982, pp. 445–470.
[2] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, Springer, Berlin, 1999.
[3] C. Carpineto, G. Romano, Concept Data Analysis: Theory and Applications, John Wiley & Sons, Ltd., 2004.
[4] U. Priss, Linguistic applications of formal concept analysis, in: B. Ganter, G. Stumme, R. Wille (Eds.), Formal Concept Analysis, Foundations and Applications, in: LNAI, vol. 3626, Springer Verlag, 2005, pp. 149–160.
[5] L.C. Freeman, D.R. White, Using Galois lattices to represent network data, Sociological Methodology 23 (1993) 127–146.
[6] C. Roth, S. Obiedkov, D.G. Kourie, Towards concise representation for taxonomies of epistemic communities, in: S. Ben Yahia, E. Mephu Nguifo (Eds.), Proceedings of the 4th International Conference on Concept Lattices and their Applications, Faculté des Sciences de Tunis, Université Centrale, Hammamet, Tunisia, 2006, pp. 205–218.
[7] G. Stumme, A. Mädche, FCA-merge: Bottom-up merging of ontologies, in: B. Nebel (Ed.), Proc. 17th Intl. Conf. on Artificial Intelligence, IJCAI '01, Seattle, WA, USA, 2001, pp. 225–230.
[8] C. Carpineto, G. Romano, A lattice conceptual clustering system and its application to browsing retrieval, Machine Learning 24 (2) (1996) 95–122.
[9] S. Kuznetsov, Machine learning and formal concept analysis, in: P. Eklund (Ed.), Concept Lattices: Proc. of the 2nd Int. Conf. on Formal Concept Analysis, in: LNCS, vol. 2961, Springer-Verlag, 2004, pp. 287–312.
[10] P. Valtchev, R. Missaoui, R. Godin, Formal concept analysis for knowledge discovery and data mining: The new challenges, in: P. Eklund (Ed.), Concept Lattices: Proc. of the 2nd Int. Conf. on Formal Concept Analysis, in: LNCS, vol. 2961, Springer-Verlag, 2004, pp. 352–371.

[11] R. Godin, H. Mili, Building and maintaining analysis-level class hierarchies using galois lattices, in: Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications, 1993, pp. 394–410.
[12] C. Lindig, G. Snelting, Assessing modular structure of legacy code based on mathematical concept analysis, in: Proceedings of the 1997 International Conference on Software Engineering, ICSE'97, Boston, MA, 1997, pp. 349–359.
[13] G. Snelting, F. Tip, Reengineering class hierarchies using concept analysis, SIGSOFT Software Engineering Notes 23 (6) (1998) 99–110.
[14] M. Huchard, H. Dicky, H. Leblanc, Galois lattice as a framework to specify algorithms building class hierarchies, Theoretical Informatics and Applications 34 (2000) 521–548.
[15] U. Dekel, Applications of concept lattices to code inspection and review, Tech. rep., Department of Computer Science, Technion, 2002.
[16] G. Arévalo, Understanding behavioral dependencies in class hierarchies using concept analysis, in: Proceedings of LMO 2003: Langages et Modeles à Objets, Hermes, Paris, 2003, pp. 47–59.
[17] T. Tilley, R. Cole, P. Becker, P. Eklund, A survey of formal concept analysis support for software engineering activities, in: Formal Concept Analysis: Foundations and Applications, in: Lecture Notes in Computer Science, vol. 3626, 2005, pp. 250–271.
[18] S. Kuznetsov, S. Obiedkov, Comparing performance of algorithms for generating concept lattices, Journal of Experimental and Theoretical Artificial Intelligence 14 (2–3) (2002) 189–216.
[19] G. Birkhoff, Lattice Theory, Amer. Math. Soc. Coll. Publ., Providence, R.I, 1973.
[20] B. Davey, H. Priestley, Introduction to Lattices and Order, 2nd ed., Cambridge University Press, Cambridge, 2002.
[21] R. Godin, R. Missaoui, H. Alaoui, Incremental concept formation algorithms based on Galois lattices, Computation Intelligence 11 (2) (1995) 243–250.
[22] P. Valtchev, R. Missaoui, Building concept (Galois) lattices from parts: Generalizing the incremental methods, in: Proc. 9th Int. Conf. on Conceptual Structures, ICCS 2001, in: Lecture Notes in Artificial Intelligence, vol. 2120, Springer, Berlin, 2001, pp. 290–303.
[23] S. Ferré, The use of associative concepts for fast incremental concept formation in sparse contexts, in: B. Ganter, A. de Moor (Eds.), Using Conceptual Structures – Contributions to ICCS 2003, Shaker Verlag, 2003, pp. 171–184.
[24] L. Nourine, O. Raynaud, A fast algorithm for building lattices, Information Processing Letters 71 (1999) 199–204.
[25] F.J. van der Merwe, S. Obiedkov, D.G. Kourie, AddIntent: A new incremental algorithmfor constructing concept lattices, in: P. Eklund (Ed.), Concept Lattices: Proc. of the 2nd Int. Conf. on Formal Concept Analysis, in: LNCS, no. 2961, Springer-Verlag, 2004, p. 411.
[26] E.M. Norris, An algorithm for computing the maximal rectangles in a binary relation, Revue Roumaine de Mathématiques Pures et Appliquées 23 (2) (1978) 243–250.
[27] B. Ganter, Two basic algorithms in concept analysis., fB4-Preprint No. 831, TH Darmstadt, 1984.
[28] J.P. Bordat, Calcul pratique du treillis de Galois d'une correspondance, Mathématiques et Sciences Humaines 23 (2) (1978) 243–250.
[29] F.J. van der Merwe, D.G. Kourie, Compressed pseudo-lattices, Journal of Experimental and Theoretical Artificial Intelligence 14 (2–3) (2002) 229–254.
[30] F.J. van der Merwe, Constructing concept lattices and compressed pseudo-lattices, Master's Thesis, University of Pretoria, 2003.
[31] C.L. Blake, C.J. Merz, UCI repository of machine learning databases, University of California, Irvine, Dept. of Information and Computer Sciences, 1998. URL: http://www.ics.uci.edu/~mlearn/MLRepository.html.
[32] P. Grigoriev, S. Yevtushenko, Elements of an agile discovery environment, in: Proc. 6th Int. Conf. on Discovery Science, DS 2003, in: Lecture Notes in Artificial Intelligence, vol. 2843, Springer, 2003, pp. 309–316.
[33] P. Valtchev, R. Missaoui, R. Godin, M. Meridji, Generating frequent itemsets incrementally: Two novel approaches based on Galois lattice theory, Journal of Experimental and Theoretical Artificial Intelligence 14 (2–3) (2002) 115–142.