ISIMA

Institut Supérieur d'Informatique, de Modélisation et de leurs Applications

1 rue de la Chebarde
Aubières, 63178, France

NATIONAL RESEARCH UNIVERSITY

Higher School of Economics

Kochnovskiy Proyezd, 3
Moscow, 125319, Russia

Master Thesis report:
Data science and 3rd year of computer science engineering

Horn Minimization: an overview of existing algorithms

Author : Simon Vilmin

Academic Supervisor : Sergei A. Obiedkov

Held : June, 26, 2018

Acknowledgement

List of Figures

List of Tables

List of Algorithms

Abstract

Contents

Introduction

Chapter 1

Introduction to implications through closure systems

In this first chapter, we will be involved in presenting our topic of minimization. For this ground to be understandable by as much readers as possible, we will heavily rely on toy examples to illustrate and provide intuition on the various notions we will introduce. To be more precise on the path we are about to follow in this chapter, we are first to expose an informal small example of the task we want to achieve. Then, we shall investigate the history of research on our topic, to act as an exposition of the actual knowledge on the question and to give a context to our study. For the rest of this chapter we will get familiar with mathematical objects called *closure operators* and *closure systems* modelling our problem. As we shall observe, the topic of minimization can be described in several mathematical frameworks. However, even if we describe briefly other objects in next chapters, we will stick to our closure framework in all the report in order to have a leading light among various different terminologies.

1.1 Implications and minimization: first meeting

Let us imagine we are some specialist of flowers and plants in general. As such, we are interested in studying *correlations* between plant characteristics. Some possible traits are: *colourful, bloom, wither, aquatic, seasonal, climbing, scented, flower, perennial* and so forth. Having observed countless plants during our studies, we are able to draw relations among all those *attributes*. For instance, we know that a plant having the attribute *flower* is likely to have traits *scent, bloom, wither* while a plant being *perennial* (i.e: does not need a lot of water to survive, like a cactus) is not likely to be *aquatic*.

Those relations *"if we have some attributes, we get those ones too"* depict correlation between attributes (not cause/consequence!). It is important to stress on the knowledge those relations bring. They just indicate that whenever we have say *flower*, we have also *colourful*. This is very different from saying that *because* some plant is a flower, it will be colourful. We call those correlation relations *implication* and use $flower \longrightarrow colourful$ to denote *"if we have the attribute flower, then we have colourful"*. Now let us give

some implications:

$$(colourful,\ bloom \longrightarrow seasonal),\ (colourful,\ wither \longrightarrow seasonal),\ (bloom \longrightarrow wither)$$

All those implications represent a certain amount of knowledge. While in our example they are not numerous we could imagine having tons of them. Hence we would wonder whether there is a way to reduce the number of implications while keeping all the knowledge they represent. This question is *minimization*. Actually, in our small example we can reduce the number of implications. Take *(colourful, bloom* $\longrightarrow$ *seasonal)*. We can derive this implication relation only with the two other ones. Indeed, because a plant *blooming* is likely to *wither* (3rd implication), we have *(colourful, bloom* $\longrightarrow$ *wither)*, but since we now have *wither* and *colourful* we also have *seasonal* (2nd implication). That is, the implication *(colourful, bloom* $\longrightarrow$ *seasonal)* is useless (or *redundant*) in our context and can be removed. Our set of implications will then be smaller, but pointing out the same relations as before.

To summarize, we have seen that out of a set of *attributes* we can draw several relations called *implications* providing some knowledge. We also realized that sometimes, some implications are not necessary. Consequently, the set of implications we are given can be *minimized* without altering the information it contains. This is the topic we were interested during this master thesis. In the next section, we will trace back the overhaul knowledge on this question.

1.2 Research on implications theories minimization

This section is intended to supply the reader with a general overview of the minimization topic. After a short contextual information, we focus on some relevant results on the question by providing references to algorithms and properties dedicated to our problem. Eventually, we situate our work within this context.

The question of minimization has been discussed and developed through various frameworks, and several computer scientists communities. Notice that in order not to make this synthesis too long, we will stay within the context of minimization and will not trace the field of implication theories in general. For a survey of this domain anyway, the reader should refer to [30]. Also, note that minimality in general terms is not unique. Indeed, one can define several type of minimality among implication systems. For instance, not only we can define minimality with respect to the number of implication within a system (which is our interest) but also with respect to the number of attributes in each implications. The former one is called *canonical* in relational database field, and *hyperarc minimum* within the graph context. Especially in the graph-theoretic and boolean logic settings, one can derive more types of minimality. For general introduction to boolean logic notations, we invite the reader to see [16]. In terms of propositional logic, implications are represented through Horn formulae. Interestingly, the minimization problem we are going to consider is the only one being polynomial time solvable. Other problems are proved to be NP-Complete or NP-Hard. For more discussion on other minimality definitions and their computational complexity, the reader should refer to [14, 7, 8, 6, 30, 12]. In particular for NP-Completeness in the canonical case, one can see [24]. In subsequent explanations, we will refer to minimization with respect to the number of implications.

To the best of our knowledge, the two first fields in which algorithms and properties of minimality arose are Formal Concept Analysis (FCA) (see [22, 21] for an introduction) and Database Theory (DB) (see [25]). Both sides were developed independently in the early 80's. For the first domain, characterization of minimality goes to Duquenne and Guigues [23], in which they describe the so-called *canonical basis* (also called *Duquenne-Guigues basis* after its authors) relying on the notion of pseudo-closed sets. For the database part, study of implications is made by Maier through FD's ([25, 18]). The polynomial time algorithm he gives for minimization heavily relies on a fast subroutine discovered by Beeri and Bernstein in [10], 1979.

From then on, knowledge increased over years and spread out over domains. Another algorithm based on a minimality theorem is given by Shock in 1986 ([26]). Unfortunately, as we shall see and as already discussed by Wild in [29] the algorithm may not be correct in general, even though the underlying theorem is. During the same period, Ausiello and al. brought the problem to graph-theoretic ground, and provided new structure known as *FD-Graph* and algorithm to represent and work on implication systems in [7, 5, 6]. This approach has been seen in graph theory as an extension of the transitive closure in graphs ([1]), but no consideration equivalent to minimization task seems to have been taken beforehand, as far as we know. Still in the 1980 decade, Ganter expressed the canonical basis formalized by Duquenne and Guigues in his paper related to algorithms in FCA, [21] through closure systems, pseudo-closed and quasi-closed sets. Next, Wild ([27, 28, 29]) linked within this set-theoretic framework both the relational databases, formal concept analysis and lattice-theoretic approach. In relating those fields, he describes an algorithm for minimizing a basis, similar to algorithms of Day and, somehow, Shock (resp. [19], [26]). This framework is the one we will use for our study, and can be found in more recent work by Ganter & Obiedkov in [8]. Also, the works of Maier and Duquenne-Guigues have been used in the lattice-theoretic context by Day in [19] to derive an algorithm based on congruence relations. For in-depth knowledge of implication system within lattice terminology, we can see [17] as an introduction and [11] for a survey. Later, Duquenne proposed some variations in Day's work with another algorithm in [20]. More recently, Boròs and al. by working in a boolean logic framework, exhibited a theorem on the size of canonical basis [13, 14]. They also gave a general theoretic approach that algorithm should do one way or another on reduction purpose. Out of these papers, Berczi & al. derived a new minimization procedure based on hypergraphs in [15]. Furthermore, an algorithm for computing the canonical basis starting from any system is given in [8].

Even though the work we are going to cite is not designed to answer this question of minimization, it must also be exposed as the algorithm is intimately related to DG basis and can be used for base reduction. The paper of Angluin and al. in query learning, see [2], provides an algorithm for learning a Horn representation of an unknown initial formula. It has been shown later by Ariàs and Alcazar ([3]) that the output of Angluin algorithm was always the Duquennes-Guigues basis.

Our purpose with this master thesis is to review and implement as much as possible the algorithms we exposed to provide a comparison. This comparison shall act as both theoretical and experimental statement of algorithm efficiency. As we already mentioned we will focus on closure theory framework. The reason for this choice is our starting point. Because we start from the algorithms provided by Wild and

because the closure framework is the one we are the most familiar with, we focus on clearly explain this terminology with examples. However, once we will be comfortable with those definitions, we will relate other frameworks to our main approach in the next chapter, to explain and draw parallels with other algorithms. In the next section we will focus on theoretical definitions we shall need to understand the algorithms we have implemented.

1.3 Implications and minimization: theoretic approach

Here we will dive into mathematical representation of the task we gave in the first section of this chapter. For the recall, our aim here is to get familiar with the representation being closest from closure systems. Most of the notions initially come from [23, 21, 28, 22] but the reader can also find more than sufficient explanations in [8, 30]. Readers with knowledge in relational databases will recognize most of functional dependency notations. The reason is close vicinity between implications and functional dependencies. Talking about our needs, we can consider them as equivalent notations. Actually, the real-life application our set up will be the closest from is FCA ([22]) as we shall see in the last chapter.

1.3.1 Implications and closure systems

The easiest object to project onto mathematical definitions is our attribute set. For all the report, we fix Σ to be a set of *attributes*. Usually, we will denote attributes by small letters: *a, b, c, ...* and subsets of Σ (groups of attributes) will be denoted by capital letters: *A, B, C, ...* We assume the reader to have few background in elementary set-theoretic notations.

Definition 1 (*Implication, implication system*)**.** *An implication over Σ is a pair (A, B) with $A, B \subseteq \Sigma$. It is usually denoted by $A \longrightarrow B$. A set $\mathcal{L}$ of implications is called an implication system, implication theory or implication(al) base(is).*

Note that given as is, this definition seems to lose the semantic relation we depicted earlier. But we should keep in mind that in our set up, we will be given implications more than an attribute set. Hence, implications will make sense on their own, independently from the attribute set they are drawn from. Quickly, remark that implications in logical terms are expressed as *Horn formulae* giving another of its names to implication theories. Also, in $A \longrightarrow B$, A is said to be the *premise* (or *body*) and B the *conclusion* (*head*).

Definition 2 (*Model*)**.** *Let $\mathcal{L}$ be an implication system over Σ, and $M \subseteq \Sigma$. Then:*

(i) M is a model of an implication $A \longrightarrow B$, written $M \models A \longrightarrow B$, if $B \subseteq M$ or $A \nsubseteq M$,

(ii) M is a model of $\mathcal{L}$ if $M \models A \longrightarrow B$ for all $A \longrightarrow B \in \mathcal{L}$.

The notion of model may seem disarming at first sight. But M being a model of $A \longrightarrow B$ simply means that, if A is included in M, then for the implication $A \longrightarrow B$ to hold in M, we must have B in M too. This still suits the intuitive notion of premise/conclusion. Placed in the context of M, $A \longrightarrow B$ says *"whenever we have A, we must also have B"*. Reader with some background in mathematical logic should be familiar with the notation $\models$, denoting semantic entailment, as opposed to $\vdash$ for syntactic deduction (see [16]). By a fortunate twist of fate, semantic entailment is our next step:

Definition 3 (*Semantic entailment*)**.** *We say that an implication $A \longrightarrow B$ semantically follows from $\mathcal{L}$, denoted $\mathcal{L} \models A \longrightarrow B$, if all models M of $\mathcal{L}$ are models of $A \longrightarrow B$.*

Because next definitions are going to be on a slightly different structure, even though closely related to implication systems of course, let us rest for a while and illustrate our definitions with an example.

Example Consider again our plant properties. Let $\Sigma = \{$*colourful, bloom, wither, seasonal, aquatic, perennial, flower, scented*$\}$. An implication could be *flower* $\longrightarrow$ *scented*, or even *(bloom, aquatic)* $\longrightarrow$ *colourful* if we get rid off semantic interpretations. An implication basis $\mathcal{L}$ is for instance:

(colourful, bloom $\longrightarrow$ *seasonal), (colourful, wither* $\longrightarrow$ *seasonal), (bloom* $\longrightarrow$ *wither)*

and $M =$*(colourful, bloom, seasonal)* is a model of *colourful, bloom* $\longrightarrow$ *seasonal* because both the head and the body of the implication belong to M. Also, M is not a model of $\mathcal{L}$ because it is not a model of *bloom* $\longrightarrow$ *wither*. A model of $\mathcal{L}$ could be *(bloom, wither)* or even the empty set $\emptyset$.

Next definitions are about closure operators, and closure systems. We need to ground ourselves in those definitions before returning to implications. 2^Σ is the set of all subsets of Σ, also named the *power set* of Σ.

Definition 4 (*Closure operator*)**.** *Let Σ be a set and $\phi : 2^\Sigma \longrightarrow 2^\Sigma$ an application on the power set of Σ. ϕ is a closure operator if $\forall X, Y \subseteq \Sigma$:*

(i) $X \subseteq \phi(X)$ *(extensive),*

(ii) $X \subseteq Y \longrightarrow \phi(X) \subseteq \phi(Y)$ *(monotone),*

(iii) $\phi(X) = \phi(\phi(X))$ *(idempotent).*

$X \subseteq \Sigma$ is called closed if $X = \phi(X)$.

Definition 5 (*Closure system*)**.** *Let Σ be a set, and $\Sigma^\phi \subseteq 2^\Sigma$. Σ^ϕ is called a closure system if:*

(i) $\Sigma \in \Sigma^\phi$,

(ii) if $\mathcal{S} \subseteq \Sigma^\phi$, *then* $\bigcap \mathcal{S} \in \Sigma^\phi$ *(closed under intersection).*

In the second definition, it is worth stressing on the fact that Σ^ϕ is a set of sets. Also, the notation Σ^ϕ may seem surprising, but it has been chosen purposefully. Indeed, to each closure system Σ^ϕ over Σ, we can associate a closure operator ϕ and vice-versa:

- from ϕ to Σ^ϕ: compute all closed sets of ϕ to obtain Σ^ϕ,
- from Σ^ϕ to ϕ: define $\phi(X)$ as the smallest element of Σ^ϕ (inclusion-wise) containing X. Observe that such a set always exists in Σ^ϕ because $\Sigma \in \Sigma^\phi$.

In any case, this notation used for clear exposition of the link between closure systems and closure operators will be adapted to our context of implication systems as we shall see later on. Notice that one can encounter another object, *closure space*, being a pair (Σ, ϕ) where Σ is a set and ϕ a closure operator over Σ. We are likely to find this notation notably in [27, 28] where a general theory of closure spaces is addressed.

Example Let us imagine we have four people: *Jezabel, Neige, Seraphin* and *Narcisse*. Let us assume they all know each other and then define a relation *"like"* between them. For instance, say *Séraphin likes Jezabel*. this relation is a *binary relation*: it relates pairs of elements. We can represent this relation by a graph where nodes are people and edges represent relations:

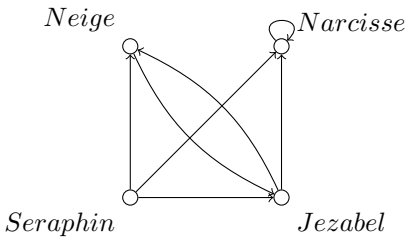


Figure 1.1: Graph of "like" relation

The arrow from *Seraphin* to *Jezabel* stands for *"Seraphin likes Jezabel"* and the arrow from *Narcisse* to itself means equivalently *"Narcisse likes Narcisse"*. With this clear, let us introduce an operation of gathering people. Starting from any group A of persons presented here, let's add to A every person liked by at least one element of A, until we can no more add people. For instance:

- if we start from *Neige*, because *Neige* likes *Jezabel* and *Jezabel* likes *Narcisse* we will add both of them to the group of *Neige*,
- because *Narcisse* only likes himself, we have no people to add in his group.

Now observe that this operation of gathering people is in fact a closure operator:

(i) it is *extensive*: starting from any group of people, we can only add new ones, hence either the group does not change (e.g: *Narcisse*) or it grows,

(ii) it is *monotone*: if we start from a group A containing a group B, it is clear that we will at least gather in A all the people we would add with B,

(iii) *idempotency*: once we added all the people we had to reach, then trying to find new people is useless by definition. Hence the group will remain the same if we apply our operation once more.

We are going to get back to our main implication purpose to illustrate the notion of closure in our context. It turns out that given a basis $\mathcal{L}$ over some set Σ, the set of models of $\mathcal{L}$, $\Sigma^{\mathcal{L}}$, is a closure system. Moreover, the operator $\mathcal{L} : 2^{\Sigma} \longrightarrow 2^{\Sigma}$ associating to a subset X of Σ the smallest model (inclusion wise) containing X is a closure operator. Furthermore, the closure system it defines is $\Sigma^{\mathcal{L}}$. An interesting point is the mathematical computation of $\mathcal{L}(X)$ given $\mathcal{L}$ as a set of implications. We rely on [27, 8] to this end. Let us define a temporary operation $\circ : 2^{\Sigma} \longrightarrow 2^{\Sigma}$ as follows:

$$X^{\circ} = X \cup \bigcup \{B \mid A \longrightarrow B \in \mathcal{L},\ A \subseteq X\}$$

Applying this operator up to stability provides $\mathcal{L}(X)$. In other words $\mathcal{L}(X) = X^{\circ\circ\cdots}$. It is clear that we have a finite amount of iterations since X cannot grow more than Σ. Readers with background in logic (see [14]) or graph theory ([15]) might see this operation as the marking or forward chaining procedure.

Example Let's stick to our vegetable example, but reducing Σ to {*bloom, flower, colourful* } (abbreviated *b, f, c*) for the sake of simplicity. Furthermore, let $\mathcal{L} = \{$*((colourful, bloom)* $\longrightarrow$ *flower), (flower* $\longrightarrow$ *bloom)*$\}$, abbreviated then $cb \longrightarrow f$, $f \longrightarrow b$. For instance, because $f \longrightarrow b \in \mathcal{L}$, the smallest model of $\mathcal{L}$ containing f is bf, and bf is closed. More precisely, the set of closed sets is the following:

$$\Sigma^{\mathcal{L}} = \{\emptyset,\ b,\ c,\ bf,\ bcf\}$$

Having presented the main definitions we shall need, we are to investigate practical computation of closures and more elaborated structures like the canonical basis (or Duquenne-Guigues basis) in the next section.

1.3.2 Canonical Basis, Closure Algorithm

Before giving the definition of canonical basis, we should consider special kind of sets given $\mathcal{L}$ over Σ. Also, we will need to expose particular implications. First of all, let us introduce a property through a proposition we will assume (we redirect the reader to [8] for another proof). When not introduced, we consider a system $\mathcal{L}$ of implications, over some attribute set Σ.

Proposition 1. *Let $A \longrightarrow B$ be an implication. $\mathcal{L} \models A \longrightarrow B$ if and only if $B \subseteq \mathcal{L}(A)$.*

Proof. $\mathcal{L} \models A \longrightarrow B \implies B \subseteq \mathcal{L}(A)$. Every model of $\mathcal{L}$ models $A \longrightarrow B$, hence for each closed set X of $\mathcal{L}$, either $A \subseteq X$ and $B \subseteq X$, or $A \nsubseteq X$. Consider all closed X for which $A \subseteq X$. By definition $\mathcal{L}(A) = \bigcap\{X \in \Sigma^{\mathcal{L}},\ A \subseteq X\}$ and $B \subseteq \mathcal{L}(A)$.

$B \subseteq \mathcal{L}(A) \implies \mathcal{L} \models A \longrightarrow B$. By contraposition suppose $\mathcal{L} \not\models A \longrightarrow B$. Then there must exist at least one model X of $\mathcal{L}$ such that $A \subseteq X$ and $B \nsubseteq X$. Because $A \subseteq X$, $\mathcal{L}(A) \subseteq X$ hence $B \nsubseteq \mathcal{L}(A)$.

□

Definition 6 (*Redundancy*)**.** *An implication $A \longrightarrow B$ of $\mathcal{L}$ is redundant if $\mathcal{L} - \{A \longrightarrow B\} \models A \longrightarrow B$. If $\mathcal{L}$ contains no redundant implications, it is non-redundant.*

Our definition of redundancy models the notion of *"useless"* we were talking about in our toy example: if an implication is true in some $\mathcal{L}$ even if we remove it, it brings no knowledge. In practice, redundancy can be checked as follows: put $\mathcal{L}^-$ as $\mathcal{L}$ without $A \longrightarrow B$ and compute $\mathcal{L}^-(A)$. If $\mathcal{L}^-(A) = \mathcal{L}(A)$ or equivalently, if $B \subseteq \mathcal{L}^-(A)$, then $A \longrightarrow B$ is redundant. Moreover, it is worth commenting that in FCA or DB fields (see [22, 25]), implications (or FD's) are deduced from data presented as contexts or relation schemes. Hence, we usually introduce notions of soundness and completeness ensuring that implications we are working on are meaningful with respect to the knowledge we are dealing with. More precisely, *soundness* ensures that $\mathcal{L}$ does not contain any implication not holding in the dataset. *Completeness* says that all true implications

in the data context are true in $\mathcal{L}$. Because we work directly on implications, $\mathcal{L}$ is by definition sound and complete with respect to the models it defines. Next, we set up minimality.

Definition 7 (Minimality)**.** *$\mathcal{L}$ is minimal if removing one of its implication alters $\Sigma^{\mathcal{L}}$.*

Example We consider our canonical plant example. Take

$$\mathcal{L} = \{((colourful,\ bloom) \longrightarrow seasonal),\ ((colourful,\ wither) \longrightarrow seasonal),\ (bloom \longrightarrow wither)\ \}$$

as we explained in first section, the first implication can be removed. In particular, it is redundant. Hence $\mathcal{L}$ is not minimal. If we get rid of $(colourful,\ bloom) \longrightarrow seasonal)$, $\mathcal{L}$ will be minimal.

Interestingly, depending on the implications we get, non-redundancy is not a sufficient criterion for minimality as we shall see in Maier algorithm. As an example for now, consider $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$ and $\mathcal{L} = \{ab \longrightarrow cde,\ c \longrightarrow a,\ d \longrightarrow b,\ cd \longrightarrow f\}$. $\mathcal{L}$ is not redundant, but is not minimal either. In fact, $\mathcal{L}_m = \{ab \longrightarrow cdef,\ c \longrightarrow a,\ d \longrightarrow b\}$ is equivalent to $\mathcal{L}$ but with one implication less.

For now, we defined what are implication theories, redundancy and minimality. One could expect our next step to be the exposition of some minimal basis. Unfortunately, we need to make a detour to visit some set and order definitions before getting back to our main purpose. Those notions not only deserve to explain minimal basis but also to settle some landmarks for further discussions in the next chapter.

Recall that in our example of closure operator we briefly approached binary relations. To be more formal, let E, F be two sets. A *binary relation* $\mathfrak{R}$ is a set of pairs (e, f) (sometimes denoted $e\mathfrak{R}f$) with $e \in E$, $f \in F$, or equivalently $\mathfrak{R} \subseteq E \times F$. We will assume $\mathfrak{R} \subseteq E^2$. Actually, $\mathfrak{R}$ can present some properties:

(i) *reflexivity*: $\forall x \in E, x\mathfrak{R}x$,

(ii) *irreflexivity*: $\forall x \in E, \neg(x\mathfrak{R}x)$,

(iii) *symmetry*: $\forall x, y \in E, x\mathfrak{R}y \longrightarrow y\mathfrak{R}x$

(iv) *antisymmetry*: $\forall x, y \in E, x\mathfrak{R}y \wedge y\mathfrak{R}x \longrightarrow x = y$,

(v) *asymmetry*: $\forall x, y \in E, x\mathfrak{R}y \longrightarrow \neg(y\mathfrak{R}x)$,

(vi) *transitivity*: $\forall x, y, z \in E, x\mathfrak{R}y \wedge y\mathfrak{R}z \longrightarrow x\mathfrak{R}z$

All possible properties are not given here, see [16] for more. With those properties anyway, we can define several types of relations:

Definition 8. *Let E be a set and $\mathfrak{R}$ a binary relation on E:*

(i) $\mathfrak{R}$ is an equivalence relation (denoted by $=$) if it is reflexive, transitive and symmetric,

(ii) $\mathfrak{R}$ is a (partial) order ($\leq$) if reflexive, transitive and antisymmetric,

(iii) $\mathfrak{R}$ is a strict order ($<$) if irreflexive, transitive and asymmetric.

Example Time has come for some illustrations. First, let us imagine we are looking at some tree in a meadow. Because the season is spring, this tree has branches and leaves. We are interested in the set of all leaves, and we would like to relate them by the branch they are one. Hence define $\mathfrak{R}$ as *"is on the same branch as"*, being a binary relation. It turns out that $\mathfrak{R}$ is an equivalence relation:

- *reflexivity*: every leaf is on the same branch as itself;
- *transitivity*: if a leaf l_1 is on the same branch as a leaf l_2, and l_2 is on the same branch as l_3, then it is clear that l_1 is on the same branch as l_3;
- *symmetry*: l_1 being on the same branch as l_2 clearly implies that l_2 is on the same branch as l_1.

For partial and strict ordering, we will go back to more mathematical examples, in order to slowly go back to our main purpose. Consider the set $\mathbb{N}$ $(= \mathbb{N}_0)$ of positive integers, including 0. The natural relation $\leq$ is an order, and the pair $(\mathbb{N}, \leq)$ is an ordered set. In particular it is a *totally ordered set* or *chain* because every pair of integers can be compared. $<$ is a strict total ordering on $\mathbb{N}$. Another example,let $\Sigma = \{a, b, c\}$ be a set of attributes and consider $\subseteq$ as a binary relation on 2^Σ. Again, $(2^\Sigma, \subseteq)$ is a *partially ordered set* (or *poset* under abbreviation):

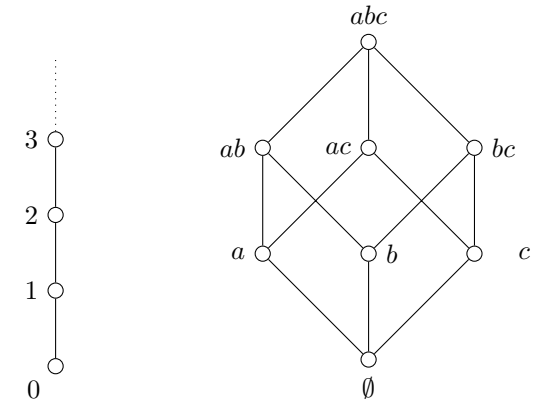
- every subset X of Σ is included in itself, for instance $\{a, b\}$ is a subset or equal to $\{a, b\}$, whence *reflexivity*,
- if $X \subseteq Y$ and $Y \subseteq X$ then necessarily, $X = Y$ (*antisymmetry*),
- if $X \subseteq Y \subseteq Z$, then clearly $X \subseteq Z$ (*transitivity*)

There is a convenient way to represent posets. At least when they are not to heavy. It is sometimes called *Hasse diagram* (see [17]) and relies on the *cover* relation of a partially ordered set. Take any poset $(P, \leq)$ and define the cover relation as $x \prec y$ if $x < y$ and $x \leq z < y \longrightarrow x = z$. In other word, $x \prec y$ says *"there is no element between x and y"*. The example of $\mathbb{N}$ is appealing. For instance, $4 \prec 5$ because there is no integer between 4 and 5, but $4 \not\prec 7$ since we can find 5 and 6 as intermediary elements. Now the Hasse diagram of $(P, \leq)$ is a graph drawn as follows:

1. there is a point for each $x \in P$,
2. if $x \leq y$, then y is placed above x,
3. we draw an arc between x and y if and only if $x \prec y$ in P.

As examples, one can observe the diagrams of $(\mathbb{N}, \leq)$ and $(2^\Sigma, \subseteq)$ described previously in figure 1.2. On the right-hand side we wrote a subset of Σ by a concatenation of its element for readability purpose.

Now equipped with orders and equivalence relation, we can go a bit further in the study of implications and sets. For instance, as exposed in the previous example, we can consider our attribute set Σ (more precisely its power set) equipped with $\subseteq$ as an ordering. Furthermore, recall that $\mathcal{L}$ is a set of implications and hence provide a closure operators. Because every subset of Σ has only one closure in $\mathcal{L}$, we can define an equivalence relation $\equiv_\mathcal{L}$ on 2^Σ as follows:



(a) $(\mathbb{N}, \leq)$ (b) $(2^{\Sigma}, \subseteq)$

Figure 1.2: Hasse diagrams of two ordered sets

$$\forall X, Y \subseteq \Sigma, X \equiv_{\mathcal{L}} Y \text{ if and only if } \mathcal{L}(X) = \mathcal{L}(Y)$$

Let us go one step beyond. With $\equiv_{\mathcal{L}}$, we can set up *equivalence classes* on $(2^{\Sigma}, \equiv_{\mathcal{L}})$. An equivalence class can be defined with respect to an element X as follows:

$$[X]_{\mathcal{L}} = \{Y \subseteq \Sigma \mid X \equiv_{\mathcal{L}} Y\}$$

Those equivalence classes are a partition of 2^{Σ} with respect to the closed sets of $\mathcal{L}$: every model of $\Sigma^{\mathcal{L}}$ defines an equivalence class.

Example Because all this discussion on equivalence class may have been a bit troubling, let us rest for a while before eventually reaching minimal basis definitions. Remind our plant example:

- $\Sigma = \{$*bloom, flower, colourful*$\}$ (abbreviated $\{b, f, c\}$),
- $\mathcal{L} = \{$*((colourful, bloom)* $\longrightarrow$ *flower), (flower* $\longrightarrow$ *bloom)*$\}$ (abbreviated $cb \longrightarrow f$, $f \longrightarrow b$)
- the models (closed sets of $\mathcal{L}$) are: $\Sigma^{\mathcal{L}} = \{\emptyset,\ b,\ c,\ bf,\ bcf\}$

In details, because of the implication $f \longrightarrow b$, we can observe that f and bf belong to the equivalence class defined by bf. The same goes for bc, cf and bcf, describing the class given associated to bcf. In order to make it clear, we give a graphical representation of those classes using orders in figure 1.3.

On the left side of the picture, we drew $(2^{\Sigma}, \subseteq)$. On the right-hand side: $(\Sigma^{\mathcal{L}}, \subseteq)$. Clusters on the left diagram are the equivalence classes, associated (dotted arrows) to their closed representative. If a cluster contains only one element, this element is closed. This drawing shows the relation between a closure operator and its associated system, in particular in implication basis context, where the closure describes models. Finally, one can graphically note that the set of models is indeed closed under intersection. While

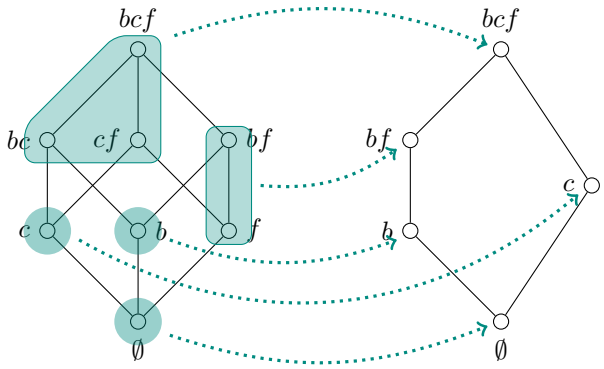


Figure 1.3: Closure operation and equivalence classes

this representation is graphically appealing, it is clearly not tractable for larger attribute set: we have to draw a diagram with an exponential number of elements (one for all $X \in 2^{\Sigma}$). Thus, all Hasse diagrams we are going to draw only aim at providing some intuition of the various notions and not as an efficient representation.

With this detour in order theory made, even though closely related to our topic as we have seen, we can now go back to our main goal: the canonical basis. It relies on some particular sets in the closure systems.

Definition 9 (*Pseudo-closed set*)**.** *Given $\mathcal{L}$ over Σ, we say that $P \subseteq \Sigma$ is pseudo-closed if:*

(i) $P \neq \mathcal{L}(P)$,

(ii) $Q \subset P$ *and* Q *pseudo-closed, implies* $\mathcal{L}(Q) \subseteq P$.

The idea of pseudo-closed sets goes back to Guigues and Duquenne in [23], but the name comes from Ganter in [21]. We can also find explanations in following research [21, 19] and in [8]. It turns out that we can explain pseudo-closure by using so called *quasi-closed sets* (see [27, 21, 23]).

Definition 10 (*Quasi-closed set*)**.** *a set $Q \subseteq \Sigma$ is quasi-closed with respect to $\mathcal{L}$ if:*

(i) $Q \neq \mathcal{L}(Q)$,

(ii) $\forall A \subseteq Q$, $\mathcal{L}(A) \subseteq Q$ *or* $\mathcal{L}(A) = \mathcal{L}(Q)$.

The recursive definition of pseudo-closed sets may seem complicated, and it is somehow since the problem of determining whether a set is pseudo-closed or not has been proven to be NP-Hard (see [8]). Fortunately, quasi-closed sets and equivalence classes give another definition to pseudo-closeness: a set is pseudo-closed if it is quasi-closed and minimal (inclusion-wise) in its equivalence class. Let us illustrate those notions with some diagrams.

Example Let us consider the following case:

- $\Sigma = \{a,\ b,\ c\}$,

- $\mathcal{L} = \{a \longrightarrow b,\ b \longrightarrow a,\ c \longrightarrow ab\}$.

As in 1.3, we will represent the power set of Σ and equivalence classes of $\mathcal{L}$. Two subsets of Σ are in the same class if they have the same closure in $\mathcal{L}$. First, one can observe the effective class representation in figure 1.4. Models of $\mathcal{L}$ are indeed $\emptyset$, ab and abc. For instance $\mathcal{L}(ac) = abc$.

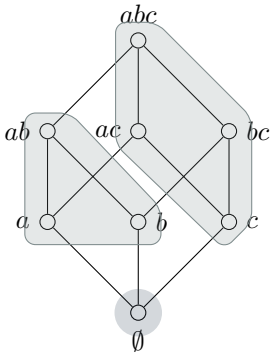


Figure 1.4: Equivalence classes representation of $\mathcal{L} = \{\ a \longleftrightarrow b,\ c \longrightarrow ab\ \}$

Next, we can observe figure 1.5 in which we somehow represented the definition of quasi-closure. We still represent equivalence classes. On the left-hand side figure, we consider the subset c. For c to be quasi-closed, we must look at all of its subsets, and see whether the closure of each subset is either smaller than c or in the same equivalence class as c. The dashed line shows which elements of the diagram we have to consider. In fact it represents what we call in lattice and order theories the *ideal* or *down-set* generated by c:

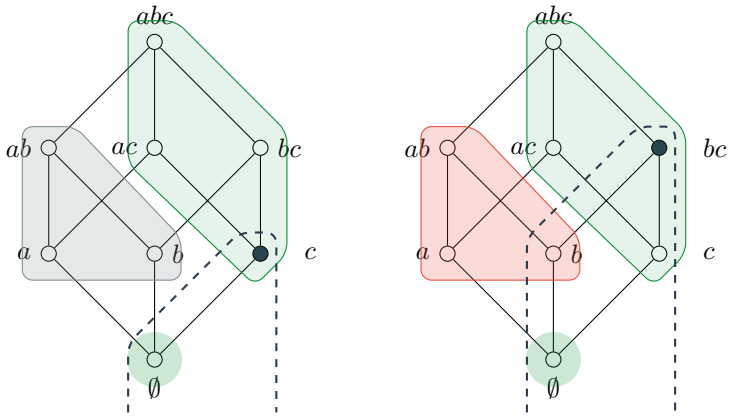
$$\downarrow c = \{X \subseteq \Sigma \,|\, X \subseteq c\}$$

It appears that the only distinct subset of c is $\emptyset$, which is closed. c itself is also not closed. Hence c is indeed quasi-closed.

On the right-hand side we consider the subset bc. As shown in the picture (under the dashed line), there are 3 elements to consider: $\emptyset$, c, b. For the same reason as for c, $\emptyset$ is not a problem. the closure of c is not included in bc, but equals the closure of bc. Hence c is not an issue either. However, b is included in bc, but its closure is ab, neither subset of bc nor equal to abc. Therefore, bc is not quasi-closed. Actually, one could informally say that a set Q is quasi-closed if the following is true:

$$(\forall P \in 2^{\Sigma})\,[(P \in \downarrow Q) \longrightarrow (\mathcal{L}(P) \in \downarrow Q \cup \{\mathcal{L}(Q)\})]$$

where $\downarrow Q$ is the ideal generated by Q (see dashed line in figure 1.5).



(a) Quasi-closed set: c (b) Non quasi-closed set: bc

Figure 1.5: Example of quasi-closeness in small implication system

Example Now let us take

- $\Sigma = \{a,\ b,\ c\}$,
- $\mathcal{L} = \{c \longrightarrow ab,\ b \longrightarrow ab\}$.

Again, we will use equivalence classes and Hasse diagram to represent the closure system of $\mathcal{L}$, see figure 1.6. In this representation we coloured all quasi-closed sets at least in grey. Red (or lighter) nodes are precisely pseudo-closed sets: they are the minimal quasi-closed sets among the equivalence class defined by abc.

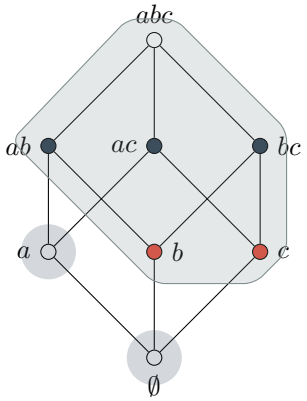


Figure 1.6: Pseudo-closed sets of $\mathcal{L} = \{b \longrightarrow ac,\ c \longrightarrow ab\ \}$

Note that in particular, minimal premises of $\mathcal{L}$ inclusion wise are pseudo-closed. Furthermore, we should be aware that an equivalence class may not contain pseudo-closed set, or more generally, quasi-closed sets. As such, we cannot consider that minimal elements of equivalence classes are quasi-closed. Take for example

$\mathcal{L} = \{\emptyset \longrightarrow a,\ b \longrightarrow a\ \}$. b and ab define a class, but b is not even quasi-closed. With these notions, we can move on and define the canonical basis.

Definition 11 (*Duquenne-Guigues basis*)**.** *The basis $\mathcal{L}$ defined by*

$$\mathcal{L} = \{P \longrightarrow \mathcal{L}(P) \mid P \text{ is pseudo-closed } \}$$

is called the Duquenne-Guigues or canonical basis. It is minimal.

This definition does not say that the canonical basis is the only one being minimal. Actually, it says that every minimal basis should have the same number of implications than this one. We can find a deeper argument in [8] on links between any minimal basis and the canonical one.

So far we discussed several notions: implications, pseudo-closed set, quasi-closed set, canonical basis and so forth. Most of them relies heavily on computing the closure of sets with respect to $\mathcal{L}$. Hence, to have practical efficiency, we must be able to compute closures as fast as possible. Fortunately, several algorithms can be found. Among them, there is a naïve procedure based on the operation $\circ$ we described earlier. Furthermore the algorithm by Beeri and Bernstein in [10] called LinClosure addresses this question. LinClosure as previously mentioned has been widely used, notably in [25, 18, 8, 26, 19]. Before describing those procedures, let us introduce our complexity notations:

- $|\,\Sigma\,|$ will denote the size of the attribute set Σ,
- $|\,\mathcal{B}\,|$ will be the number of implications in $\mathcal{L}$ ($\mathcal{B}$ stands for body),
- $|\,\mathcal{L}\,|$ is the number of symbols used to represent $\mathcal{L}$.

We consider $|\,\mathcal{L}\,|$ to be in reduced form for complexity results. By "reduced" we mean that we do not have distinct implications with same bodies. Indeed, if say, $a \longrightarrow b$ and $a \longrightarrow c$ holds in some $\Sigma^{\mathcal{L}}$ then we can replace those two implications by $a \longrightarrow bc$. Moreover, we shall not explain in details O notation for complexity since we do not need in-depth knowledge within this field. For us, it is enough to say that O is the asymptotically worst case complexity (in time or space). For instance, in the worst case, $|\,\mathcal{L}\,| = |\,\mathcal{B}\,| \times |\,\Sigma\,|$, thus $|\,\mathcal{L}\,| = O(|\,\mathcal{B}\,| \times |\,\Sigma\,|)$. Closure and LinClosure are algorithms 1, 2 (resp.).

As we already mentioned, the algorithm Closure relies on the $\circ$ operation. The principle is to re-roll over the set of implications $\mathcal{L}$ to see whether there exists an implication $A \longrightarrow B$ in $\mathcal{L}$ such that $\mathcal{L}(X) \not\models A \longrightarrow B$ up to stability. Asymptotically, we will need $O(|\,\mathcal{B}\,|^2 \times |\,\Sigma\,|)$ if we remove only one implication per loop. the $|\,\Sigma\,|$ cost comes from the set union.

LinClosure has $O(|\,\mathcal{L}\,|)$ time complexity. The main idea is to use counters. Starting from X, if we reach for a given $A \longrightarrow B$ as many elements as $|A|$, then $A \subseteq \mathcal{L}(X)$ and we must also add B. Because the closure in itself is not the main point of our topic, we will not study LinClosure in depth. Furthermore, there exists other linear time algorithm for computing closure. For more complete theoretical and practical comparisons of closure algorithms, we redirect the reader to [9]. In this paper, LinClosure is shown maybe not to be the most efficient algorithm in practice when used in other algorithms, especially when

Algorithm 1: CLOSURE

Input: A base $\mathcal{L}$, $X \subseteq \Sigma$

Output: The closure $\mathcal{L}(X)$ of X under $\mathcal{L}$

```
closed := ⊥ ;
L(X) := X ;
while ¬closed do
    closed := ⊤ ;
    foreach A ⟶ B ∈ L do
        if A ⊆ L(X) then
            L(X) := L(X) ∪ B ;
            L := L − {A ⟶ B} ;
            closed := ⊥ ;
return L(X);
```

compared with CLOSURE. Anyway, because of its theoretical complexity and use in all algorithms we will review, we will still consider LINCLOSURE, notably because we can separate the initialization step from the computation one in some cases on optimization purpose.

In this last section, we got a step further in building ground for understanding the implication theory structure. We gave definitions of minimality and visual examples of particular sets called pseudo-closed. With the support of those sets, we defined the canonical basis known to be minimal. Finally, algorithms for efficiently computing closures have been presented.

Conclusion In this chapter we first gave a soft introduction to our task with a somehow "physical" example. Then we described briefly advances starting from the first properties found independently in Concept Analysis and Relational Databases fields. We have seen that the question of Horn minimization has been studied in various fields such as graphs, closure spaces, logic (where the name Horn comes from), functional dependencies, lattices. Then, we placed our study within this context. The aim of this study has been exposed as providing a review of some algorithms we talked about, and comparing them under implementation. The last part of the chapter was dedicated to a more formal and theoretical ground necessary for a good understanding of subsequent parts. In the next chapter, we will theoretically discuss in details several algorithms.

Algorithm 2: LinClosure

Input: A base $\mathcal{L}$, $X \subseteq \Sigma$

Output: The closure $\mathcal{L}(X)$ of X under $\mathcal{L}$

```
foreach A ⟶ B ∈ ℒ do
    count[A ⟶ B] := |A| ;
    if |A| = 0 then
        X := X ∪ B ;
    foreach a ∈ A do
        list[a] = list[a] ∪ {A ⟶ B} ;
update := X ;
while update ≠ ∅ do
    choose m ∈ update ;
    update := update − {m} ;
    foreach A ⟶ B ∈ list[m] do
        count[A ⟶ B] := count[A ⟶ B] − 1 ;
        if count[A ⟶ B] = 0 then
            add := B − X ;
            X := X ∪ add ;
            update := update ∪ add ;
return X ;
```

Chapter 2

Minimization algorithms for implication theories

In the first chapter, we settled the context of our study. We introduced our subject of interest and defined the theoretical ground it is built on. In this chapter, we discuss several algorithms and their complexity. Our aim with this review is to provide understanding of the algorithms, study their complexity. We also describe them to prepare their implementation in the next chapter. Our first section within this part will be dedicated to give a general outline of our review. Next, we will effectively dive into in-depth studies of minimization procedure.

2.1 Overview of the study

Here, we draw the main lines of our subsequent explanations. First and foremost, we shall try as much as possible to express all algorithms in our framework of closure systems and orders. Nevertheless, to draw parallels and understand the translation from one framework to another we may proceed to some travels in other terminologies such as query learning or directed graphs.

The study will be divided into three parts. First, we will study algorithms coming from the FCA (Formal Concept Analysis) community. This includes algorithms MinCover from [9, 19, 27] and DuquenneMinimization being a variation of MinCover (more precisely, to the algorithm provided by Day in a lattice-theoretic framework). This second procedure can be found in [20]. Even though ShockMinimization is an algorithm from FCA community, we may include discussion about it in this part because of the theorem it is based on. Secondly, we will dive into DB (Database) and graphs domains by working on MaierMinimization (see [25, 18]) and FD-Graphs, an extension of minimization to graphs provided by Ausiello and al; in [5, 6, 7]. Eventually we get into algorithms coming out of boolean logic and query learning communities with BercziMinimization and AFPMinimization (see [15] and [2, 3] resp.).

Regarding the order in which we will study the algorithms, there is no particular choice but our proximity

with the domains. Because MinCover is the algorithm we are starting from, it seemed logical for us to explain it first.

2.2 Algorithms on closure systems (FCA community)

2.2.1 Minimal Cover

The minimization procedure we will describe in this section, soberly called MinCover is the starting point for this master thesis. It can be found in [8]. As we shall explain, it has roots in Day lattice-based algorithm [19], and more surprisingly, "unknown" ancestor in Shock algorithm [26].

The principle is to perform *right-saturation*, and then *body redundancy* elimination. In fact, not only this is the general idea recently issued in [14], but it is also the main theorem of Shock in [26] and the part of a theorem by Wild ([27, 28]). This procedure has the advantage to be somehow intuitive. Indeed, right-saturation means replacing the conclusion of an implication by the closure of its premise:

$$A \longrightarrow B \text{ becomes } A \longrightarrow \mathcal{L}(A) \text{ (of course } B \subseteq \mathcal{L}(A))$$

hence, it means that we associate to A, all the information we can reach starting from A. Then, we perform body redundancy elimination. That is, for each right-closed implication, we check whether the amount of knowledge represented by $\mathcal{L}(A)$ depends necessarily on A. In other words, we remove $A \longrightarrow \mathcal{L}(A)$ from $\mathcal{L}$, and if starting from A we still get the same amount of information ($\mathcal{L}^{-}_{A \to \mathcal{L}(A)}(A) = \mathcal{L}(A)$), then A is not required to get $\mathcal{L}(A)$: $A \longrightarrow \mathcal{L}(A)$ can be removed. Now that the principle is explained in words, let us introduce the pseudo-code (see algorithm 3).

Algorithm 3: MinCover

Input: $\mathcal{L}$: an implication base
Output: the canonical base of $\mathcal{L}$

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $B := \mathcal{L}(A \cup B)$;
  $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $A := \mathcal{L}(A)$;
  if $A \neq B$ **then**
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

MinCover ends up on the canonical basis. Assuming that closure are computed with LinClosure, the overhaul complexity of the algorithm is $O(|\mathcal{B}||\mathcal{L}|)$. To see correctness of the algorithm, observe that

the resulting $\mathcal{L}_c$ is equivalent to $\mathcal{L}$ at the end of the algorithm. Indeed, at the end of the first loop, we replaced B in every implications $A \longrightarrow B$ of $\mathcal{L}$ by $\mathcal{L}(A)$. But by proposition 1, $\mathcal{L} \models A \longrightarrow B$ if and only if $B \subseteq \mathcal{L}(A)$. This is in particular the case for $B = \mathcal{L}(A)$. In the second loop we remove an implication only if it is redundant, thus the resulting $\mathcal{L}_c$ is indeed equivalent to $\mathcal{L}$. The main question is minimality of $\mathcal{L}_c$. Recall that the DQ-basis, being minimal is based on pseudo-closed sets. Hence, if we can show that we keep an implication in the second loop only if the premise $\mathcal{L}^-(A)$ is pseudo-closed, we are done. This is the purpose of next "hand-made" proposition:

Proposition 2. *Let $\mathcal{L}$ be a right-closed implication theory. Denote $\mathcal{L}^-(A) := (\mathcal{L} - \{A \longrightarrow \mathcal{L}(A)\})(A)$, the following holds for all $A \longrightarrow \mathcal{L}(A) \in \mathcal{L}$:*

(i) if $\mathcal{L}(A) = \mathcal{L}^-(A)$, $A \longrightarrow \mathcal{L}(A)$ is redundant in $\mathcal{L}$,

(ii) if $\mathcal{L}(A) \neq \mathcal{L}^-(A)$, $\mathcal{L}^-(A)$ is pseudo-closed.

Proof. *(i)* is trivial by definition. For *(ii)*, let us show that $\mathcal{L}^-(A)$ is quasi-closed, and then minimal among quasi-closed sets in its equivalence class. Suppose $\mathcal{L}^-(A)$ is not quasi-closed, then there must exist $B \subseteq \mathcal{L}^-(A)$ such that $\mathcal{L}(B) \nsubseteq \mathcal{L}^-(A)$ and $\mathcal{L}(B) \neq \mathcal{L}(\mathcal{L}^-(A)) = \mathcal{L}(A)$. Because $B \subseteq \mathcal{L}^-(A)$, either $\mathcal{L}(B) \subset \mathcal{L}(A)$ or $\mathcal{L}(B) = \mathcal{L}(A)$. If we are in the equality case, we are done. So let $B \subseteq \mathcal{L}^-(A)$ and $\mathcal{L}(B) \subset \mathcal{L}(A)$. By definition of $\mathcal{L}^-$, if there exists such B, either it is closed in $\mathcal{L}$ and we are done, or there exist implications $C_i \longrightarrow \mathcal{L}(C_i)$ such that $C_i \subseteq B$, $\mathcal{L}(C_i) \nsubseteq B$ with $\bigcup \mathcal{L}(C_i) = \mathcal{L}(B)$. But for all such implications, $C_i \subseteq \mathcal{L}^-(A)$, and by construction of $\mathcal{L}^-(A)$, $\mathcal{L}(B) = \bigcup \mathcal{L}(C_i) \subseteq \mathcal{L}^-(A)$. Hence, $\mathcal{L}^-(A)$ is indeed quasi-closed. Now, let us show that it is minimal among quasi-closed sets in its equivalence class. If A is closed in $\mathcal{L}^-$, the result is direct, because for all $C \longrightarrow \mathcal{L}(C)$ in $\mathcal{L}$, either $C \nsubseteq A$ or $(C \subseteq A) \wedge (\mathcal{L}(C) \subseteq \mathcal{L}^-(A) = A)$. Assume the presence of some B such that $A \subseteq B \subseteq \mathcal{L}^-(A)$ with B being quasi-closed. Note that if A is not closed under $\mathcal{L}^-$, it cannot be quasi-closed. If $B \longrightarrow \mathcal{L}(B) = \mathcal{L}(A)$ is in $\mathcal{L}$, then $\mathcal{L}^-(A) = \mathcal{L}(A)$ and we have a contradiction. If $B \longrightarrow \mathcal{L}(B) \notin \mathcal{L}$, then we have $\mathcal{L}^-(A) = B$ because B contains A and will be closed under $\mathcal{L}^-$, which concludes the proof. □

This proposition is sufficient for the algorithm to correctly end up on the canonical basis. Interestingly, the main idea of MinCover is similar to the theorem 2.1 of Shock in [26], but the algorithm in practice is much closer from the procedure given by Day in section 6 of [19]. Before moving to their work, let us settle down an example of trace for MinCover.

Example Let us discuss the following example:

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

We will present a trace of MinCover by a sequence of vectors representing $\mathcal{L}$ after modifications:

$$\begin{pmatrix} ab \longrightarrow cde \\ cd \longrightarrow f \\ c \longrightarrow a \\ d \longrightarrow b \\ abcd \longrightarrow ef \end{pmatrix} \longrightarrow \begin{pmatrix} ab \longrightarrow abcdef \\ cd \longrightarrow abcdef \\ c \longrightarrow ac \\ d \longrightarrow bd \\ abcd \longrightarrow abcdef \end{pmatrix} \longrightarrow \begin{pmatrix} ab \longrightarrow abcdef \\ abcdef \longrightarrow abcdef \\ c \longrightarrow ac \\ d \longrightarrow bd \\ abcdef \longrightarrow abcdef \end{pmatrix} \longrightarrow \begin{pmatrix} ab \longrightarrow abcdef \\ c \longrightarrow ac \\ d \longrightarrow bd \end{pmatrix}$$

The first vector is the initial basis. Then we perform right-saturation. The third vector differs a bit from true execution of MinCover, but it illustrates replacement of A by $\mathcal{L}^-(A)$ in $A \longrightarrow \mathcal{L}(A)$. As we can see, two implications have the same premises and conclusion: they are useless and hence removed in the resulting $\mathcal{L}$, being the last vector.

Now that things should be a bit clearer, let us discuss the two other algorithms previously cited. Remark that we will not explain the procedure given by Wild in [27, 28] because it is strictly MinCover:

1. right-close all implications of $\mathcal{L}$,
2. find a minimal non-redundant subfamily of implications in $\mathcal{L}$ right-closed, i.e redundancy elimination.

Hence, the procedure given by Shock is presented in algorithm 4.

Algorithm 4: ShockMinimization

Input: $\mathcal{L}$: a theory to minimize
Output: a minimum cover for $\mathcal{L}$

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  if $B \not\subseteq \mathcal{L}(A)$ **then**
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow \mathcal{L}(B)\}$;

This routine, co-issued with the theorem we discussed previously is quite different from MinCover. Even though the conditional statement $B \not\subseteq \mathcal{L}(A)$ is equivalent to $\mathcal{L}(A) \neq \mathcal{L}^-(A)$ and replacing $A \longrightarrow B$ by $A \longrightarrow \mathcal{L}^-(B)$ is about right-closing $A \longrightarrow B$, the resulting basis of this algorithm may not be minimal in general:

- if $\mathcal{L} = \{\emptyset \longrightarrow a,\ a \longrightarrow b\}$ (in this order), ShockMinimization will produce $\emptyset \longrightarrow ab$ which is right,
- if $\mathcal{L} = \{a \longrightarrow b,\ \emptyset \longrightarrow b\}$, the result will be $\{a \longrightarrow ab,\ \emptyset \longrightarrow ab\}$ being redundant.

In fact, this error has already been pointed out in 1995 by Wild in [29]. However, the implications given by Shock may have quasi-closed premises and present similarities with the algorithm for generating quasi-closed sets FindCrucialGenerators.In Wild work, we can also find another proof for the theorem of Shock, jointly with minimality of DQ-basis (theorem 5 of [28, 29]). Let us discuss now the algorithm proposed by A. Day in 1992 (see pseudo-code 5: DayMinimization). We express it through our framework of closure systems and implications. Since it would only complicate our explanations, we do not integrate definitions

of lattice theory. However, for the reader with few background in lattice theory, we can mention Day's overhaul framework in some sentences. The main idea is to focus on the partial order $(2^{\Sigma}, \subseteq)$ being a complete join-semilattice (or just complete lattice when we consider finite Σ) where the closure operator $\mathcal{L}(\cdot)$ is in fact a $\bigvee$-morphism from $(2^{\Sigma}, \subseteq)$ to $(\Sigma^{\mathcal{L}}, \subseteq)$. In $(2^{\Sigma}, \subseteq)$, the join operation $(\vee)$ is set union. Consequently, the equivalence classes of $\mathcal{L}$ define a congruence on the powerset of Σ being the kernel of $\mathcal{L}(\cdot)$. The approach developed by A. Day is then to build congruences (hence closure operators) through ordered pairs (or quotients) of the same equivalence class, representing implications.

Algorithm 5: DayMinimization

Input: $\mathcal{L}$: a theory to minimize

Output: canonical basis of $\mathcal{L}$

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $A := \mathcal{L}(A)$;
  $B := \mathcal{L}(A \cup B)$;
  if $A \neq B$ **then**
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

Here, equivalence with MinCover is clear. The only difference is the order in which operations are performed. Even though the 2 algorithms rely on the same computation, it is worth noting a particular case where the algorithm by Day may fail. Consider a system not being reduced, e.g:

$$\mathcal{L} = \{b \longrightarrow ac,\ c \longrightarrow a,\ c \longrightarrow b\}$$

we have:

- MinCover output: $\{b \longrightarrow ac,\ c \longrightarrow abc\}$,
- DayMinimization output: $\{b \longrightarrow ac,\ ac \longrightarrow abc,\ c \longrightarrow abc\}$

In general, performing right-closure before redundancy elimination, as in MinCover avoids this problem.

In this section we reviewed the algorithm acting as our starting point, by studying its complexity and principle. We also linked it to research we made and other algorithms we found. As exposed, MinCover summarizes and corrects all material we covered here, and hence justifies not to implement all of them. The next section defines a slightly different algorithm, being a variation of the work from [19].

2.2.2 Duquenne algorithm

Later, also based on Day lattice theoretic work and its own approach, Duquenne proposed a variation of MinCover based on first computing quasi-closed sets and then using the recursive characterization of pseudo-closed sets to iteratively build the Duquenne-Guigues basis. Recall that pseudo-closed sets are particular open sets that can be defined by two means:

- a set P is pseudo-closed if it is larger than all the closure of any sub pseudo-closed set,
- P is pseudo-closed if it is quasi-closed and minimal among quasi-closed sets of $[P]_{\mathcal{L}}$.

We will call this procedure DuquenneMinimization (see algorithm 6). In fact, It uses the algorithm 2 from [19] to compute quasi-closed sets from premises of $\mathcal{L}$, this is the first loop. Observe that in fact, ShockMinimization does not compute pseudo-closed sets but quasi-closed sets. All pseudo-closed sets are included in the resulting $\mathcal{L}$ after this first step. Then, we use *lectic ordering* to have a $\subseteq$-compatible way to process implications before building $\mathcal{L}_c$.

Algorithm 6: DuquenneMinimization

Input: $\mathcal{L}$ a theory to minimize
Output: $\mathcal{L}_c$ the DQ-basis of $\mathcal{L}$

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} = \mathcal{L} - \{A \longrightarrow B\}$;
  $A := \mathcal{L}(A)$;
  if $B \not\subseteq A$ **then**
    $B = B \cup A$;
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

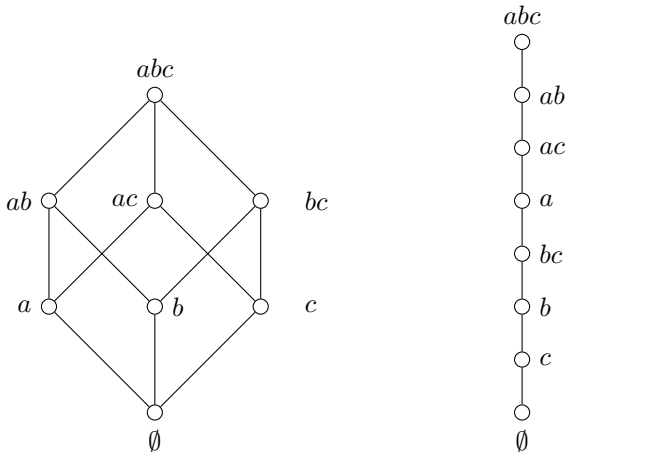
LecticOrder($\mathcal{L}$) ;
$\mathcal{L}_c := \emptyset$;

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  foreach $\alpha \longrightarrow \beta \in \mathcal{L}_c$ **do**
    if $\alpha \subset A \wedge \beta \not\subseteq A$ **then**
      $\mathcal{L} = \mathcal{L} - \{A \longrightarrow B\}$;
      goto next $A \longrightarrow B \in \mathcal{L}$;
  $B = \mathcal{L}(B)$;
  $\mathcal{L}_c := \mathcal{L}_c \cup \{A \longrightarrow B\}$;

return $\mathcal{L}_c$;

Before proving the algorithm, let us define lectic ordering $\leq_\Sigma$. First, we must assume that Σ can be assigned a total order $\leq$. For the recall, an order is total if for all pairs $(x,\ y)$ of Σ, $x \leq y$ or $y \leq x$. Hence, provided Σ is a chain, we can define $\leq_\Sigma$ on 2^Σ as follows: $\forall A, B \subseteq \Sigma\ A \leq_\Sigma B$ if the smallest element in which A and B differ belongs to B. We say that A is *lectically smaller than* B. Note that $\leq_\Sigma$ is *$\subseteq$-compatible*, that is $A \subseteq B \longrightarrow A \leq_\Sigma B$. The opposite direction however does not hold since $\subseteq$ is a partial ordering, while $\leq_\Sigma$ is total.

Example Consider $\Sigma = \{a, b, c\}$ with $a < b < c$. In this setting, $b \leq_\Sigma a$, $b \leq_\Sigma ab$ and $b \leq_\Sigma bc$ for instance. To observe $\subseteq$-compatibility, we can refer to figure 2.1 illustrating the two orderings side-by-side.



(a) Partial order $(2^{\Sigma}, \subseteq)$ (b) Total order $(2^{\Sigma}, \leq_{\Sigma})$

Figure 2.1: Subset and lectic ordering

The advantages of lectic ordering is to allow for easier checking of pseudo-closeness recursive property: if we test sets in lectic order, we are sure not to avoid any pseudo-closed subset of a given set when looking at previously considered ones. Furthermore, because this is a total order, we can use fast logarithmic sorting procedure to order efficiently sets. Next, let us give some elements of proof for this algorithm.

Proposition 3. *The following statements hold for all* $A \longrightarrow B \in \mathcal{L}$, $\mathcal{L}^- = \mathcal{L} - \{A \longrightarrow B\}$*:*

(i) $\mathcal{L}^-(A) = \mathcal{L}(A)$*, then* $A \longrightarrow B$ *is redundant,*

(ii) $\mathcal{L}^-(A) \neq \mathcal{L}(A)$*, then* $\mathcal{L}^-(A)$ *is quasi-closed in* $\mathcal{L}$*.*

Proof. (i). $\mathcal{L}^-(A) = \mathcal{L}(A)$ is equivalent to $B \subseteq \mathcal{L}^-(A)$, that is $\mathcal{L} - \{A \longrightarrow B\} \models A \longrightarrow B$. *(ii).* Assume $\mathcal{L}^-(A) \neq \mathcal{L}(A)$ and $\mathcal{L}^-(A)$ not quasi-closed. Then we must be able to find an implication $\alpha \longrightarrow \beta$ such that $\alpha \subseteq \mathcal{L}^-(A)$ and $\mathcal{L}(\alpha) \nsubseteq \mathcal{L}^-(A)$. We have also $\mathcal{L}(\alpha) \subset \mathcal{L}(A)$. $\mathcal{L}^-$ is a closure operator, hence $\alpha \subseteq \mathcal{L}^-(A) \longrightarrow \mathcal{L}^-(\alpha) \subseteq \mathcal{L}^-(A)$. $\mathcal{L}^-(\alpha) \subseteq \mathcal{L}^-(A)$ and $\mathcal{L}(\alpha) \nsubseteq \mathcal{L}^-(A)$ leads to $\mathcal{L}^-(\alpha) \subset \mathcal{L}(\alpha)$. Because of $\mathcal{L}^-$ computations, this leads to $A \subseteq \mathcal{L}(\alpha)$, hence by monotonicity of $\mathcal{L}$, $\mathcal{L}(A) \subseteq \mathcal{L}(\alpha)$ contradicting $\mathcal{L}(\alpha) \subset \mathcal{L}(A)$.

□

Next, we will assume a lemma given in [20] from where the algorithm comes:

Lemma 1. *For lists* $\mathcal{L}$*,* $\mathcal{H}$ *of implications, with* $\mathcal{H} \subseteq \mathcal{L}$*, the following statements are equivalent:*

(i) $\mathcal{L} \equiv \mathcal{H}$*,*

(ii) $\mathcal{L}$ *has the same canonical basis as* $\mathcal{H}$*,*

(iii) for every pseudo-closed set P *of* $\mathcal{L}$*, there is at least one* $A \longrightarrow B \in \mathcal{H}$ *for which* $A \subseteq P \subset \mathcal{L}(P) = \mathcal{H}(A)$*.*

Because $\mathcal{L}^-$ is a closure operator, if $\mathcal{L}^-(A)$ is not closed under $\mathcal{L}$, it is then the smallest quasi-closed set containing A. Furthermore, both A and $\mathcal{L}^-(A)$ belong to $[A]_{\mathcal{L}}$. With lemma 1 where $\mathcal{H} = \mathcal{L}$, one has that every pseudo-closed sets of $\mathcal{L}$ must be a premise of $\mathcal{L}$ after the first loop.

Proposition 4. *At the end of* DuquenneMinimization $\mathcal{L}_c$ *is the DG-basis of* $\mathcal{L}$.

Proof. We proved that after the first step, all pseudo-closed sets appeared as premise of $\mathcal{L}$. Because we use a $\subseteq$-compatible ordering on premises of $\mathcal{L}$ with lectic order, the nested loop on $\mathcal{L}_c$ checks the recursive property of being pseudo-closed. Hence we only add implications $P \longrightarrow \mathcal{L}(P)$ where P is $\mathcal{L}$-pseudo-closed to $\mathcal{L}_c$. □

Example In order to clarify the algorithm, let us execute it on a small example. Because we may have a lazy imagination, consider the example we used for MinCover:

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

We proceed by steps:

1. *left-saturation*. For all implications $A \longrightarrow B$ of $\mathcal{L}$, we compare $\mathcal{L}(A)$ and $\mathcal{L}^-(A)$ where $\mathcal{L}^-$ is $\mathcal{L}$ from which we removed $A \longrightarrow B$. We should keep in mind that $\mathcal{L}^-$ is different for every $A \longrightarrow B$ then, not only because previous implications are altered or remove, but also since at each step we delete a different implication from $\mathcal{L}$. Let us present results of this step through table 2.1. Observe that at the

$\mathcal{B}(\mathcal{L})$	$\mathcal{L}$ before	$\mathcal{L}(\cdot)$	$\mathcal{L}^-$	$\mathcal{L}^-(\cdot)$	imp ?	$\mathcal{L}$ after
ab	$ab \to cde, cd \to f,$ $c \to a, d \to b,$ $abcd \to ef$	$abcdef$	$cd \to f, c \to a,$ $d \to b,$ $abcd \to ef$	ab	$ab \longrightarrow abcde$	$ab \to abcde,$ $cd \to f, c \to a,$ $d \to b, abcd \to ef$
cd	$ab \to abcde,$ $cd \to f, c \to a,$ $d \to b, abcd \to ef$	$abcdef$	$ab \to abcde,$ $c \to a, d \to b,$ $abcd \to ef$	$abcdef$	*removed*	$ab \to abcde, c \to a,$ $d \to b, abcd \to ef$
c	$ab \to abcde, c \to a,$ $d \to b, abcd \to ef$	ca	$ab \to abcde$ $d \to b, abcd \to ef$	c	$c \longrightarrow ca$	$ab \to abcde,$ $c \to ca, d \to b,$ $abcd \to ef$
d	$ab \to abcde,$ $c \to ca, d \to b,$ $abcd \to ef$	db	$ab \to abcde,$ $c \to ca,$ $abcd \to ef$	d	$d \longrightarrow db$	$ab \to abcde,$ $c \to ca, d \to db,$ $abcd \to ef$
$abcd$	$ab \to abcde,$ $c \to ca, d \to db,$ $abcd \to ef$	$abcdef$	$ab \to abcde,$ $c \to ca, d \to db$	$abcde$	$abcde \longrightarrow abcdef$	$ab \to abcde,$ $c \to ca, d \to db,$ $abcde \to abcdef$

Table 2.1: First step of DuquenneMinimization

second step, when we consider cd, because $\mathcal{L}(cd) = \mathcal{L}^-(cd)$, the implication $cd \longrightarrow f$ is removed from $\mathcal{L}$. The basis we get after this step is: $ab \longrightarrow abcde,\ c \longrightarrow ca,\ d \longrightarrow db,\ abcde \longrightarrow abcdef$.

2. *lectic ordering* Here we will not follow the sorting procedure, but instead, we will illustrate how does lectic ordering work in practice, and how to compute it at hands easily. Say we have the following *total* order in elements of Σ: $a < b < c < d < e < f$. We can imagine represent a subset of Σ by a binary string of size $|\Sigma|$ where the least significant bit (on the right) corresponds to f, and the most significant bit (on the left) matches a. The binary string associated to some $A \subset \Sigma$ will have ones in place of elements it contains. For instance, the subset $acef$ will have 101011 as binary string:

a	b	c	d	e	f
1	0	1	0	1	1

From this point of view, lectic ordering is just binary enumeration. A premise A_1 will be lower than A_2 under lectic order if the binary string associated to A_1 comes before the word related to A_2 when enumerating naturally binary numbers. In our case, we have:

- ab: 110000,
- c: 001000,
- d: 000100,
- $abcde$: 111110.

Ordering those premises by their order of appearance under binary counting, we get $\mathcal{L}$ swapped as follows: $d \longrightarrow db$, $c \longrightarrow ca$, $ab \longrightarrow abcde$, $abcde \longrightarrow abcdef$.

3. *getting pseudo-closed implications* For each implication of $\mathcal{L}$, we want to check whether its premise is pseudo-closed or not. To do this, we will build iteratively our resulting basis $\mathcal{L}_c$, containing left-pseudo-closed and right-closed implications (see table 2.2).

implication	input $\mathcal{L}_c$	pseudo-closed ?	output $\mathcal{L}_c$
$d \to db$	$\emptyset$	$\vee$	$d \to db$
$c \to ca$	$d \to db$	$\vee$	$d \to db, c \to ca$
$ab \to abcde$	$d \to db, c \to ca$	$\vee$	$d \to db, c \to ca$ $ab \to abcdef$
$abcde \to abcdef$	$d \to db, c \to ca$ $ab \to abcdef$	$\times$: $ab \to abcdef$	$d \to db, c \to ca$ $ab \to abcdef$

Table 2.2: DuquenneMinimization third step

As one can see, when we add an implication from $\mathcal{L}$ to $\mathcal{L}_c$ we perform right-closing. This is necessary to check pseudo-closeness. Consequently for the last step, because $ab \longrightarrow \mathcal{L}(abcde) = abcdef \in \mathcal{L}_c$, $abcde$ is not pseudo-closed: $ab \subseteq abcde$ and $\mathcal{L}(ab) = abcdef \nsubseteq abcde$. Thus we do not add it to $\mathcal{L}_c$. At the end of the algorithm we have $\mathcal{L}_c = d \longrightarrow db$, $c \longrightarrow ca$, $ab \longrightarrow abcdef$.

Regarding the complexity of the algorithm, one may note that the first loop has the same complexity as the second step of MinCover, that is $O(|\mathcal{B}||\mathcal{L}|)$ provided we use LinClosure for lowering theoretical

complexity of closure computations. Observe then that the lectic order is a total order, hence we can use a logarithmic sorting function as quick-sort for LecticOrder, resulting in $O(|\Sigma||\mathcal{B}|\log_2(|\mathcal{B}|))$ due to $O(|\Sigma|)$ comparison of two sets. Eventually, for each $A \longrightarrow B$, the nested for each loop may require $O(|\mathcal{L}|)$ since we are performing set operations on at most as much implications as $|\mathcal{B}|$. Then we may perform a closure under $\mathcal{L}$, being $O(|\mathcal{L}|)$ also. Therefore, the overhaul loop should require $O(|\mathcal{B}||\mathcal{L}|)$ time. From the three steps we have, we can conclude that the whole algorithm has complexity $O(|\mathcal{B}||\mathcal{L}|)$ as MinCover. However, the first loop of DuquenneMinimization act as a redundancy elimination which helps to reduce the cost of closure computations in the whole algorithm. This could be an improvement of MinCover to test in practical implementation.

In this section we were interested in studying algorithms built on the work of Wild, Day, Duquenne-Guigues and Ganter mainly relying on left and right-saturation of implications to produce the canonical basis. In the next section we will be involved in studying algorithms based on the work of Maier and later, Ausiello.

2.3 Algorithms based on Maier's database approach

2.3.1 First algorithm: Maier's algorithm on FDs

Here we will consider one of the first algorithm given for the minimization task. It has been proposed by Maier in [25, 18] and rely notably on the algorithm LinClosure issued in [10]. For understandability we will explain this algorithm through implication theories framework while drawing parallel with Maier's notation and definitions. As a soft introduction, we will develop an interesting and simple example in Maier's algorithm context.

Example Let Σ and $\mathcal{L}$ be as follows (in fact, same example as in previous section):

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

Let us try to minimize it "with hands". First, we see $abcd \longrightarrow ef$ to be redundant. Indeed, if we remove it from $\mathcal{L}$, we still have $ab \longrightarrow cde$ and $cd \longrightarrow f$, thus $\mathcal{L}^- := \mathcal{L} - \{abcd \longrightarrow ef\} \models abcd \longrightarrow ef$. $\mathcal{L}^-$ is not redundant any more. Nevertheless, we can still remove an implication. Indeed, not only we can reach ab from cd, but also cd from ab. Consequently, we could remove $cd \longrightarrow f$ from $\mathcal{L}^-$ while adding to the head of $ab \longrightarrow cde$ the element f (the head of $cd \longrightarrow f$) to avoid loss of informations. Hence, we would end up with

- $\mathcal{L} = \{\ c \longrightarrow a,\ d \longrightarrow b,\ ab \longrightarrow cdef\ \}$

Those steps of redundancy elimination and equivalence manipulation are the core manipulations of Maier algorithm. For the recall, Maier worked with functional dependencies, but this makes no difference when it comes as implications.

Redundancy elimination As mentioned in the first chapter, given $\mathcal{L}$, $A \longrightarrow B \in \mathcal{L}$ is redundant if $\mathcal{L} - \{A \longrightarrow B\} \models A \longrightarrow B$ or equivalently if $B \subseteq \mathcal{L}^{-}_{A \to B}(A)$. Thus get rid off redundancy elimination can be done in the following procedure:

Algorithm 7: RedundancyElimination

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}$ without redundant implications

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  if $\mathcal{L} - \{A \longrightarrow B\} \models A \longrightarrow B$ **then**
    remove $A \longrightarrow B$ from $\mathcal{L}$;

Checking for redundancy is done with LinClosure. Because this is done for all implications of $\mathcal{L}$, complexity of redundancy elimination is $O(|\mathcal{B}| \times |\mathcal{L}|)$.

Equivalence classes As briefly described previously, we can identify equivalence classes within $\Sigma^{\mathcal{L}}$. For $X \subseteq \Sigma$, we can set up $[X]_{\mathcal{L}} = \{Y \subseteq \Sigma \mid \mathcal{L}(Y) = \mathcal{L}(X)\}$. Recall that this is the definition of an *equivalence class* we presented in the first chapter. More than this, we can limit those equivalence classes to premises of $\mathcal{L}$, i.e for some $A \subseteq \Sigma$ let

(i) $E_{\mathcal{L}}(A) = \{X \longrightarrow Y \in \mathcal{L} \mid X \in [A]_{\mathcal{L}}\}$

(ii) $e_{\mathcal{L}}(A) = \{X \mid X \in \mathcal{B}(\mathcal{L}) \cap [A]_{\mathcal{L}}\}$

Plus, we say that A *directly determines* B, denoted $A \xrightarrow{\mathrm{d}} B$, if $\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow B$. Now, the minimization process in [25, 18] is the following:

Proposition 5. *Let $\mathcal{L}$ be an irredundant theory. If $A \longrightarrow B$, $C \longrightarrow D \in \mathcal{L}$ (distinct) are such that $C \equiv A$ and $A \xrightarrow{\mathrm{d}} C$, then we can remove $A \longrightarrow B$ from $\mathcal{L}$ and replace $C \longrightarrow D$ by $C \longrightarrow D \cup B$ without altering $\Sigma^{\mathcal{L}}$.*

Proof. Suppose we removed $A \longrightarrow B$ and modified $C \longrightarrow D$ to $C \longrightarrow D \cup B$. Put $\mathcal{L}^{-}$ as the system we obtained. The main point is to show that we still have $\mathcal{L}^{-} \models A \longrightarrow B$. Recall $A \xrightarrow{\mathrm{d}} C$, thus:

$$\begin{aligned} \mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow C &\implies \mathcal{L} - A \longrightarrow B \models A \longrightarrow C \\ &\implies \mathcal{L}^{-} \models A \longrightarrow C \end{aligned}$$

Because we changed $C \longrightarrow D$ to $C \longrightarrow D \cup B$, we have then

$$(\mathcal{L}^{-} \models A \longrightarrow C) \wedge (\mathcal{L}^{-} \models C \longrightarrow D \cup B) \implies (\mathcal{L}^{-} \models A \longrightarrow B)$$

by transitivity. Note that equivalence of A and C is preserved, because in $\mathcal{L}$, by transitivity $C \longrightarrow A \longrightarrow B$. Removing $A \longrightarrow B$ but moving B to $C \longrightarrow D \cup B$ preserves $C \longrightarrow B$. Also, taking equivalent premise is

important in order not to alter the closure system $\Sigma^{\mathcal{L}}$. If A and C were not equivalent, we may have changed the system by adding B to the closure of C even though $\mathcal{L} \not\models C \longrightarrow B$. □

This is worth noting we gave a "light" definition of direct determination more relying on a property proved by Maier than the strict original definition. Also, Wild in [27, 28] drawn a parallel between quasi-closure property and equivalence classes $E_{\mathcal{L}}(\cdot)$. Indeed, a set A will be quasi-closed if and only if $\mathcal{L}(A) = \mathcal{L}^{-}_{E_{\mathcal{L}}(A)}(A)$. Finally, the last proposition gives us an algorithmic test for minimization: given an equivalence class $E_{\mathcal{L}}(X)$, one can run across all its implications and successively remove useless ones. Actually it says that if an non-redundant basis contains direct determinations, it is not minimal. Hence contrapositive yields a condition for minimality ensuring correctness and end of the operation.

The main question is: how to get the equivalence classes efficiently? It turns out this can be done using a modified version of LinClosure. It is sufficient to embed in the function a vector of implied premises. That is, for a given premise X, we provide to LinClosure a bit-vector *implied* of size $|\,\mathcal{B}\,|$. Within the procedure, whenever we reach $count\,[A \longrightarrow B] = 0$ for some $A \longrightarrow B \in \mathcal{L}$, then A is implied by X under $\mathcal{L}$. Hence we set $implied\,[A \longrightarrow B]$ to 1. Doing this operation for all implications in $\mathcal{L}$ provide a matrix M of size $|\,\mathcal{B}\,| \times |\,\mathcal{B}\,|$. Then, to compute equivalence classes, a travel over the matrix is enough. Two implications $A \longrightarrow B$ and $C \longrightarrow D$ of $\mathcal{L}$ belong to the same equivalence class if

$$M\,[A \longrightarrow B,\ C \longrightarrow D] = 1 \quad \text{and} \quad M\,[C \longrightarrow D,\ A \longrightarrow B] = 1$$

Building the matrix requires $|\,\mathcal{B}\,|$ executions of LinClosure in any case, thus has $O(|\,\mathcal{B}\,||\,\mathcal{L}\,|)$ time complexity. Running across M is of course $O(|\,\mathcal{B}\,|^2)$. Hence the whole operation can be done in $O(|\,\mathcal{B}\,||\,\mathcal{L}\,|)$, since $|\,\mathcal{B}\,||\,\mathcal{L}\,| = |\,\mathcal{B}\,|^2|\,\Sigma\,| > |\,\mathcal{B}\,|^2$. We will note rewrite LinClosure altered since the modification is about one line in the algorithm and quite simple to understand as is. We will not write the run over M for conciseness, since principle seems sufficient for understanding. All those steps will be summarized as EquivClasses($\mathcal{L}$) in subsequent algorithm. Finally, the whole Maier minimization process is given in algorithm 8.

A question we could have is about complexity of removing direct determinations. In fact, we can use again a modified version of LinClosure to find direct determination. For each implications $A \longrightarrow B$, we would have to provide LinClosure with a vector of implications with premises equivalent to A. The first one we reach (i.e the first one for which the counter goes to 0) is necessarily an example of direct determination. Moreover, note that equivalence classes in $E_{\mathcal{L}}$ defines a partition of $\mathcal{L}$. That is, we will have to compute at most $|\,\mathcal{B}\,|$ closures to get rid off direct determinations. Whence, the last step of the algorithm requires $O(|\,\mathcal{B}\,||\,\mathcal{L}\,|)$, which is consequently the complexity of MaierMinimization by the previous explanations. It is important to mention that even though the base we obtain is minimal, it is not the canonical basis, as we shall see in the next example, acting as a trace.

Example Let us use the example we presented when studying the first algorithm, but this time, applying formally Maier's algorithm on it. Hence, as a reminder, we had:

Algorithm 8: MaierMinimization

Input: $\mathcal{L}$: a theory to minimize
Output: $\mathcal{L}$ minimized

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  if $\mathcal{L} - \{A \longrightarrow B\} \models A \longrightarrow B$ **then**
    remove $A \longrightarrow B$ from $\mathcal{L}$;

$E_{\mathcal{L}} :=$ EquivClasses($\mathcal{L}$) ;

foreach $E_{\mathcal{L}}(X) \in E_{\mathcal{L}}$ **do**
  foreach $A \longrightarrow B \in E_{\mathcal{L}}(X)$ **do**
    if $\exists C \longrightarrow D \in E_{\mathcal{L}}(X)$ *s.t* $A \xrightarrow{\mathrm{d}} C$ **then**
      remove $A \longrightarrow B$ from $\mathcal{L}$;
      replace $C \longrightarrow D$ by $C \longrightarrow D \cup B$;

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

We will proceed by steps.

1. *redundancy elimination*: in this step, we compute the closure of each premise to see whether there exists $A \longrightarrow B \in \mathcal{L}$ such that $B \subseteq \mathcal{L}^{-}_{A \longrightarrow B}(A)$. It turns out that the only one for which this happens is $abcd \longrightarrow ef$. After this step, we have:

$$\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b\ \}$$

2. *getting equivalence classes*: here is a more interesting step. First, we have to compute the matrix M (see table 2.3). The table does not represent the way computations are done, but is still of interest. On the left-hand side, we described the closure of each premise of $\mathcal{L}$. On the right-hand side, we gave the matrix M. We can see that an element $M(i,j)$ of M equals 1 if the closure of i contains the premise of j.

   Then, we need to derive out of M the different equivalence classes. For all pairs of implications (i,j), if $M(i,j) = M(j,i) = 1$, then they belong to the same equivalence class. In our case, we will partition $\mathcal{L}$ in 3 classes:

   - $E_{\mathcal{L}}(ab) = \{ab \longrightarrow cde,\ cd \longrightarrow f\}$ $(= E_{\mathcal{L}}(cd))$,
   - $E_{\mathcal{L}}(c) = \{c \longrightarrow a\}$,
   - $E_{\mathcal{L}}(d) = \{d \longrightarrow b\}$

3. *removing direct determination*: last step. We have to look in all equivalence classes for distinct implications with direct determination. Because $E_{\mathcal{L}}(c)$ and $E_{\mathcal{L}}(d)$ are of size 1, they cannot be

$\mathcal{B}(\mathcal{L})$	$\mathcal{L}(\cdot)$
ab	$abcdef$
cd	$abcdef$
c	ac
d	bd

(a) closures of $\mathcal{B}(\mathcal{L})$

	$ab \longrightarrow cde$	$cd \longrightarrow f$	$c \longrightarrow a$	$d \longrightarrow b$
$ab \longrightarrow cde$	1	1	1	1
$cd \longrightarrow f$	1	1	1	1
$c \longrightarrow a$	0	0	1	0
$d \longrightarrow b$	0	0	0	1

(b) matrix M

Table 2.3: Computing matrix M of implied premises

reduced. However, $E_{\mathcal{L}}(ab)$ is more interesting. We do not have $ab \xrightarrow{\text{d}} cd$. Indeed, the only way to reach cd from ab is to use $ab \longrightarrow cde$, that is, an element of $E_{\mathcal{L}}(ab)$. Nevertheless, $cd \xrightarrow{\text{d}} ab$ because if we restrict ourselves to $\mathcal{L} - E_{\mathcal{L}}(ab) = \{c \longrightarrow a,\ d \longrightarrow b\}$, $cd \longrightarrow ab$ holds. Consequently, we can apply our modifications: we remove $cd \longrightarrow f$ from $\mathcal{L}$, and $ab \longrightarrow cde$ becomes $ab \longrightarrow cdef$.

After applying this algorithm, we end up with a minimal $\mathcal{L}$ being:

$$\mathcal{L} = \{c \longrightarrow a,\ d \longrightarrow b,\ ab \longrightarrow cdef\}$$

In this section we provided a theoretical study of the algorithm proposed by Maier in [25, 18] for finding a minimal cover of a basis $\mathcal{L}$. Based on his results, we stated that the asymptotic complexity of this algorithm was $O(|\,\mathcal{B}\,||\,\mathcal{L}\,|)$. In the next section, we will develop another procedure coming from the graph theory community.

2.3.2 Graph-theoretic approach to Maier's algorithm

This section is dedicated to a minimization algorithm relying on graphs. It has been set up by Ausiello and al. in [7, 5, 6]. Starting from a directed hypergraph representation of functional dependencies, it builds a special kind of directed graph, called *FD-Graph* with which it reduces the initial hypergraph. In order, we are going to define what is a FD-graph, provide the general idea for the algorithm as explained in [6] and then go into further details and more precise algorithms for such computations as exposed in [5].

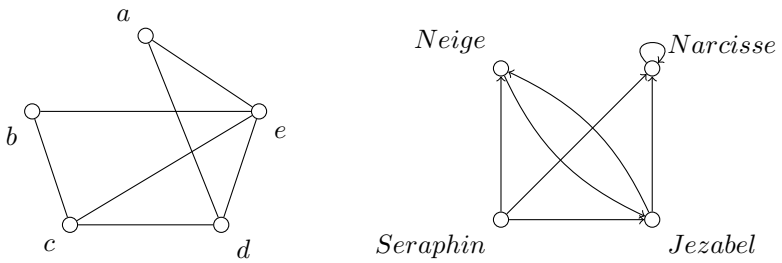
FD-Graphs and minimum covers

As we already mentioned, the graph framework developed by Ausiello and al. in [5, 6] comes from the work of Maier in database theory over functional dependencies (see [25]). Moreover we already discussed the closeness of FD and implications in our context, hence we can still consider the algorithms we are about to study from an implication point of view. This leads to no alteration. Furthermore, the hypergraph representation of some theory $\mathcal{L}$ is no more than worth mentioning for us, since it just presents an attractive graphical description of $\mathcal{L}$. Because the structure presented by Ausiello is a particular kind of directed graph, let us try to keep explanations as simple as possible and stick to this one. It might be first interesting to recall what are graphs (undirected and directed) as an introduction.

Definition 12 (*graph*). *A graph $G = (V, E)$ is a pair of sets where V is a set of nodes or vertices and E is a set of unordered pair called edges or arcs from V^2.*

Definition 13 (*directed graph*). *A graph $G = (V, E)$ where E is a set of ordered pairs from V^2 is called a directed graph.*

Example Let us illustrate those notions with examples. First, let us imagine a graph (not directed) $G_1 = (V_1, E_1)$ where $V_1 = \{ a, b, c, d, e\}$ is a set of cities and E_1 would be railways between them, as (b, c) and (a, d) for example. Because railways are bidirectional, if we can go from one city to another, then the other way around is valid too. One possible "map" is represented on the left side of figure 2.2.



(a) undirected graph G_1 (b) directed graph G_2

Figure 2.2: Representation of graphs G_1, G_2

For an example of directed graph, recall our "like" binary relation from the chapter 1. The associated graph is $G_2 = (V_2, E_2)$ where $V_2 = \{Narcisse, Neige, Jezabel, Seraphin\}$ and for instance, $(Seraphin, Jezabel)$ is an edge of E_2 while $(Jezabel, Seraphin)$ is not. See right-hand side of figure 2.2 for an illustration.

Now that the notion of graph may be clearer, let us introduce a particular kind of directed graph issued in [5, 6] as an improvement of the structure proposed in [4]. It deserves to represent implication theories within a framework simpler than hypergraphs. It has been recently re-issued in [7] being a survey.

Definition 14 (*FD-Graph*). *Given a theory $\mathcal{L}$ over some Σ, the directed graph $G_{\mathcal{L}} = (V,\ E)$ such that:*

- *$V = V_0 \cup V_1$ is the set of nodes where:*
  - *$V_0 = \Sigma$ is the set of simple nodes (a node per attribute in Σ),*
  - *$V_1 = \{X | X \in \mathcal{B}(\mathcal{L})\}$ is the set of compound nodes (a node per distinct body in $\mathcal{L}$),*
- *$E = E_0 \cup E_1$ is the set of arcs where:*
  - *E_0 is the set of full arcs. We have a full arc (X, i) in E_0 if (X, i) is an hyperarc of $\mathcal{L}$,*
  - *E_1 the set of dotted arcs. For each compound node X of V^1, we have a dotted arc $(X,\ i)$ to every attributes i of X,*

is the Functionnal Dependency Graph or FD-Graph associated to $\mathcal{L}$.

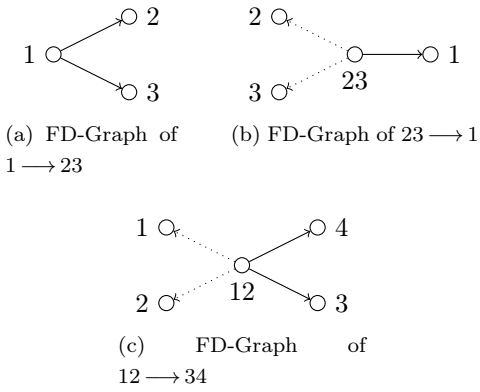
Again, the definition may be quite confusing. Therefore, let us pause our explanations with some toy examples, presented in figure 2.3. Out of those graphs, we can give a way *"with hands"* to build an FD-graph out of some $\mathcal{L}$:

- every single attribute of Σ is a node, as every premise of $\mathcal{L}$,
- For each $A \longrightarrow B$ of $\mathcal{L}$ we draw a *full* arc from the node A to *every attribute* of B,
- For each compound node A, we draw a *dotted* arc from A to *all of its attribute*.

which is indeed what we formally defined previously. Furthermore, for this algorithm, we consider a basis $\mathcal{L}$ over an attribute set Σ, such that:

- there is no $A \longrightarrow B$, $A' \longrightarrow B'$ in $\mathcal{L}$ such that $A = A'$ when $B \neq B'$,
- for all $A \longrightarrow B$ of $\mathcal{L}$, $A \cap B = \emptyset$

this is a *reduced* basis for the recall.



(a) FD-Graph of $1 \longrightarrow 23$

(b) FD-Graph of $23 \longrightarrow 1$

(c) FD-Graph of $12 \longrightarrow 34$

Figure 2.3: Representation of some FD-graph

The next definition is about describing a graph-theoretic way to combine implications to derive new one. This notion is essential for all the following material and is called *FD-paths*.

Definition 15 (*FD-Path*)**.** *Given an FD-Graph $G_{\mathcal{L}} = (V, E)$, an FD-Path $\langle i,\ j\rangle$ is a minimal subgraph $\bar{G}_{\mathcal{L}} = (\bar{V}, \bar{E})$ of $G_{\mathcal{L}}$ such that $i, j \in \bar{V}$ and either $(i, j) \in \bar{E}$ or one of the following holds:*

- *j is a simple node and there exists $k \in \bar{V}$ such that $(k, j) \in \bar{E}$ and there exists a FD-Path $\langle i,\ k\rangle$ included in $\bar{G}$,*
- *$j = \bigcup_{k=1}^{n} j_k$ is a compound node and there exists FD-paths $\langle i,\ j_k\rangle$ included in $\bar{G}$, for all $k = 1,\ \ldots,\ n$.*

Informally, an FD-path from a node i to j describes the implications we use to derive $i \longrightarrow j$. Intuitively, directed paths are FD-paths. But there is also one case in which we can go "backward" in the graph. For better understanding, see examples of figure 2.5 based on the theory described in figure 2.4. To be more precise, $\mathcal{L} = \{ab \longrightarrow f,\ af \longrightarrow g,\ a \longrightarrow c,\ b \longrightarrow d,\ cd \longrightarrow e,\ c \longrightarrow h,\ cd \longrightarrow e\}$.

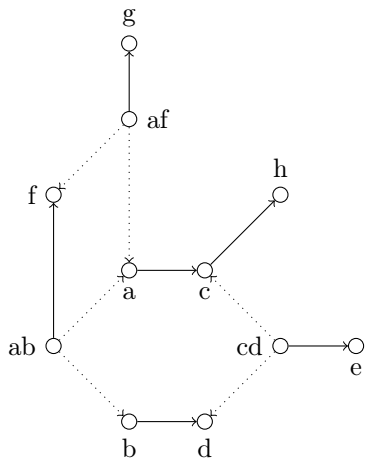
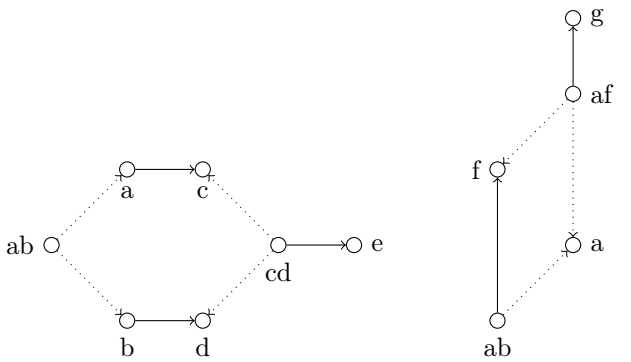


Figure 2.4: FD-Graph of some implicational basis

There are either *dotted* or *full* paths. A path $\langle i, j \rangle$ is dotted if all arcs leaving i are dotted, it is full otherwise.



(a) dotted FD-path for $ab \longrightarrow e$

(b) full FD-path for $ab \longrightarrow g$

Figure 2.5: Representation of some FD-paths

Having explained FD-Graphs, we will now move to explanations of the algorithm developed by Ausiello

and al. The following procedure finds from a given basis its *minimal cover*:

Algorithm 9: AusielloMinimization (Overview, 1983)

Input: $\mathcal{L}$ an implication basis
Output: $\mathcal{L}_c$ a minimal cover for $\mathcal{L}$

Find the *FD-Graph* of $\mathcal{L}$;
Remove *redundant* nodes ;
Remove *superfluous* nodes ;
Remove *redundant* arc ;
Derive $\mathcal{L}_c$ from the new graph ;

As we will see in detailed explanations, those steps are equivalent to Maier's procedure. In fact, the last part, removing redundant arcs, goes beyond the scope of our needs since it deserves to reduce sizes of premises and conclusion, not the number of implications. This has also been studied in Maier's work, but for out of scope reason we did not reviewed it. For the same argument here, we will not focus on it either. To help the reader see where we are heading, one should keep in mind that the two other steps of removing redundant nodes and superfluous nodes will be equivalent to removing redundant implication and direct determination respectively. However, we must first dive into the closure of an FD-Graph (parallel to closure of implications) to be able to perform removal steps.

Closure of an FD-Graph

The closure is based on the following data structures:

- V_0: set of *simple* nodes,
- V_1: set of *compound* nodes,
- D_i $(\forall i \in V)$: nodes from *incoming dotted* arcs $\{j \in V \mid (j,\ i)$ is a dotted arc$\}$,
- L_i^0 $(\forall i \in V)$: nodes from *outgoing full* arcs $\{j \in V \mid (i,\ j)$ is a full arc$\}$,
- L_i^1 $(\forall i \in V)$: nodes from *outgoing dotted* arcs $\{j \in V \mid (i,\ j)$ is a dotted arc$\}$,
- L_i^{0+}, L_i^{1+} $(\forall i \in V)$: the respective closures of L_i^0, L_i^1,
- q_m $(\forall m \in V^1)$: counter of nodes in m belonging to $L_i^{0+} \cup L_i^{1+}$ for some $i \in V$.

To make understanding easier, we first give pseudo-code closer from principle than algorithms. From a general point of view, to determine the closure of a FD-graph, we must compute the closure of all its nodes. The closure of a node is described by its full and dotted outgoing arcs. Because we put a priority on dotted possibilities, they will be computed before. Principle are given in algorithmic/pseudo-code form so that identification between steps of procedures and ideas of principle are easier to see.

First, we introduce the procedure NodeClosure which computes the closure of a node with respect to a type of arc. In other words, to compute the full closure of a node, we must first apply NodeClosure to its dotted arcs, then to its full arcs. The principle and algorithm for Nodeclosure are procedures 10, 11.

Algorithm 10: NodeClosure (Principle)

Input: L_i: set of nodes for which there exists dotted (resp. full) arcs (i, j)
Output: L_i^+: the dotted (resp. full) closure of i

Initialize a list of nodes to treat S_i to L_i ;
while *there is a node j to treat in S_i* **do**
  remove j from S_i ;
  if *j is simple node* **then**
    forall *compound node m except i, j appears in* **do**
      increase q_m by 1 ;
      if *q_m = number of outgoing dotted arcs from m* **then**
        m is reachable from i by *union* ;
        m must be treated, add it to S_i ;
  add j to the closure L_i^+ ;
  forall *nodes k such that there is an arc $(j,\ k)$* **do**
    if *k is not yet in the closure L_i^+ or in the dotted closure L_i^{1+} of i* **then**
      k is reachable from i by *transitivity* ;
      k must be treated, add it to S_i ;
  return L_i^+ ;

We would like to provide some observations on top of their description. Namely on the *union* step and q_m counters. Say $i \longrightarrow m$ where m is a compound node is a valid implication in a FD-graph. Furthermore say $m = \bigcup_i m_i$ where m_i's are simple nodes. The union step models the fact that if we have $i \longrightarrow m_i$ for all m_i in m, then we must have $i \longrightarrow m$ also. The counter q_m ensures that we indeed reached all m_i's in m. Also, the algorithm has access to all the structures we described above (nodes, sets of arcs, and so forth). Parameters are thus lists we are going to modify somewhat. The NodeClosure algorithm runs in time $O(|\mathcal{L}|)$. The first nested loop runs in at most $O(|\Sigma| \times |\mathcal{B}(\mathcal{L})|) = O(|L|)$ because S_i contains at most $|\Sigma|$ elements, and the block referring to the *union* rule runs over compound nodes, that is bodies of $\mathcal{L}$. For the second loop (transitivity) note that we can at most consider all the edges of the FD-graph. In fact, the cost of transitivity operation for all j is $O(\sum_{j=1}^{n} |L_j^0 \cup L_j^1|)$. But by definition, those sets are disjoints, and therefore we cannot treat more than $|E|$ arcs (the total number of arcs in G), that is $|\mathcal{L}|$.

Next, we present the principle and pseudo-code for the closure of an FD-graph 12, 13. Mostly, the principle is the idea we described previously. There is just one observation to make about setting a counter q_m to 1. This variable helps to see whether we can use union rule as we saw in procedure NodeClosure (10, 11). We initialize it in case i is indeed part of some compound node so that we do not omit to count it when dealing with S_i (because S_i does not contain i). In terms of complexity, we are running NodeClosure on all nodes having outgoing edges, that is $|\mathcal{B}(\mathcal{L})|$ nodes (if a compound node is represented, it must have at least one outgoing full arc). Since NodeClosure operates in $O(|\mathcal{L}|)$, the whole closure algorithm must

Algorithm 11: NodeClosure

Input: L_i: set of nodes for which there exists dotted (resp. full) arcs (i, j)
Output: L_i^+: the dotted (resp. full) closure of i

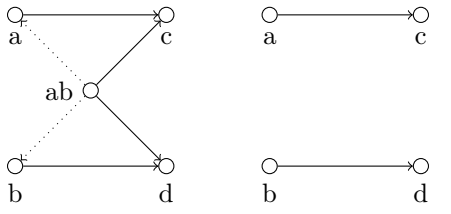
$S_i := L_i$;
while $S_i \neq \emptyset$ **do**
  select j from S_i ;
  if $j \in V^0$ **then**
    forall $m \in D_j - \{i\}$ **do**
      $q_m := q_m + 1$;
      if $q_m = |L_m^1|$ **then**
        $S_i := S_i \cup \{m\}$;
  $S_i^+ := S_i^+ \cup \{j\}$;
  forall $k \in L_j^0 \cup L_j^1$ **do**
    if $k \notin S_i^+ \cup L_i^{1+} \cup \{i\}$ **then**
      $S_i := S_i \cup \{k\}$;
return L_i^+ ;

run in $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$.

Now that algorithms for computing the closure of a FD-graph have been set, we can move to the minimization part.

Removing redundant nodes

The first step is about removing redundant implications. In terms of FD-graphs, we remove redundant nodes. A compound node (only) i is said *redundant* if for each full arc (i, j) leaving i there exists a dotted path $\langle i,\ j\rangle$. We give an example in the figure 2.6.



(a) FD-Graph with redundant node (ab)

(b) FD-Graph with redundant node removed

Figure 2.6: Elimination of redundant nodes

Algorithm 12: GraphClosure (Principle)

Input: V_0, V_1 and $\forall i \in V$ D_i, L_i^0, L_i^1
Output: $\forall i \in V$ L_i^{0+}, L_i^{1+}

forall *node* i *in* V *with outgoing arcs* **do**
  if i *is an attribute of a compound node* m **then**
    set a counter q_m to 1 ;
  initialize the closure of i to $\emptyset$;
  if i *is a compound node* **then**
    determine *dotted* arcs in the closure of i ;
  determine *full* arcs in the closure of i ;

Algorithm 13: GraphClosure

Input: V_0, V_1 and $\forall i \in V$ D_i, L_i^0, L_i^1
Output: $\forall i \in V$ L_i^{0+}, L_i^{1+}

forall $i \in V$ *with* $L_i^0 \cup L_i^1 \neq \emptyset$ **do**
  forall $m \in V^1$ **do**
    if $m \in D_i$ **then**
      $q_m := 1$;
    else
      $q_m := 0$;
  $L_i^{1+} := \emptyset$;
  $L_i^{0+} := \emptyset$;
  if $i \in V^1$ **then**
    $L_i^{1+} :=$ NodeClosure(L_i^1) ;
  $L_i^{0+} :=$ NodeClosure$(L_i^0 - L_i^{1+})$;

In this example, the associated basis is $\mathcal{L} = ab \longrightarrow cd\,;\, a \longrightarrow c\,;\, b \longrightarrow d$. Indeed, in this case, $ab \longrightarrow cd$ is redundant because $\mathcal{L} - ab \longrightarrow cd \models ab \longrightarrow cd$. So removing a redundant node is removing exactly one implication in $\mathcal{L}$ since $\mathcal{L}$ is reduced. It is quite direct to see equivalence between redundancy of a node and redundancy of the implication having this node as a premise. We give a proposition anyway to make everything clear.

Proposition 6. *An implication $A \longrightarrow B$ is redundant in $\mathcal{L}$ if and only if A is a redundant node in the FD-graph $G_{\mathcal{L}}$ associated to $\mathcal{L}$.*

Proof. Assume $A \longrightarrow B$ is redundant in $\mathcal{L}$. Then $A \longrightarrow B$ still holds in $\mathcal{L}^- := \mathcal{L} - A \longrightarrow B$. The FD-graph $G_{\mathcal{L}^-}$ associated to $\mathcal{L}^-$ is in fact $G_{\mathcal{L}}$ where we got rid of node A (being compound) and of all its outgoing arcs, dotted and full. If $\mathcal{L}^- \models A \longrightarrow B$ we must be able to find implications $X_i \longrightarrow Y_i$, $X_i \subseteq A$ such that $\bigcup_i X_i \longrightarrow B$. In particular we could add a compound node $\bigcup_i X_i$ to $G_{\mathcal{L}^-}$ with only dotted arcs to its attribute so that we would have only a dotted FD-path from $\bigcup_i X_i$ to B, hence from A to B.

Suppose A is redundant node in $G_{\mathcal{L}}$. It has full outgoing arcs, and is compound hence corresponds to a premise A of $\mathcal{L}$. Because it is redundant, we can remove all of its full outgoing arcs without any loss of information. Say $A \longrightarrow B$ is the implication represented by the node A and its full outgoing arcs. Removing all is reducing $A \longrightarrow B$ to $A \longrightarrow \emptyset$ that is an implication we can remove. Because there is still FD-path from A to B in this set up we have $A \longrightarrow B$ still holding in $\mathcal{L}$ where $A \longrightarrow B$ has been replaced by $A \longrightarrow \emptyset$ equivalent to $\mathcal{L} - \{A \longrightarrow B\}$.

□

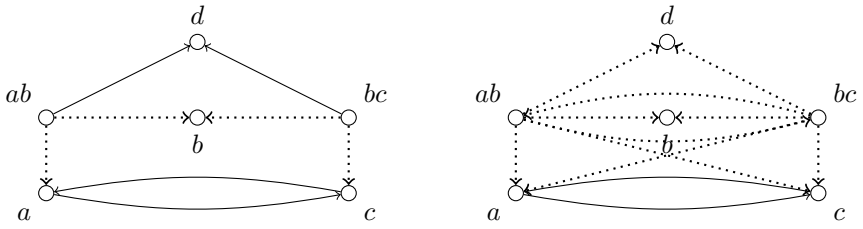
To remove redundant nodes, Ausiello and al. observed that a redundant node will only have dotted arcs in the closure of a FD-Graph. Hence its minimization procedure for some $\mathcal{L}$ and associated $G_{\mathcal{L}}$ suggests to determine the closure of $G_{\mathcal{L}}$ and then to remove all redundant nodes by checking it. However, let us consider the following case:

$$\mathcal{L} = \{ab \longrightarrow d,\ bc \longrightarrow d, a \longrightarrow c, c \longrightarrow a\}$$

with the associated FD-Graph presented in figure 2.7. On the right-hand side of the figure we gave the closure of the FD-graph. We can observe that two nodes are redundant, namely ab and bc. Indeed, we have:

- $\mathcal{L} - \{ab \longrightarrow d\} \models ab \longrightarrow d$
- $\mathcal{L} - \{bc \longrightarrow d\} \models bc \longrightarrow d$

Nevertheless, this does not mean we can remove the two of them. Indeed those two implications are somehow *"mutually redundant"*: if we remove one, the other is not redundant anymore. For instance, consider removing $ab \longrightarrow d$ from $\mathcal{L}$. Then, $\mathcal{L} = \{bc \longrightarrow d, a \longrightarrow c, c \longrightarrow a\}$. In this case $\mathcal{L} - \{bc \longrightarrow d\} \not\models bc \longrightarrow d$ because even though $c \longrightarrow a$, the lack of $ac \longrightarrow d$ prevent redundancy of $bc \longrightarrow d$. Therefore, the idea proposed by Ausiello as we understood it would result in $\mathcal{L} = \{a \longrightarrow c,\ c \longrightarrow a\}$ after redundancy elimination, being not correct. In Maier's term, this would be equivalent to first marking all redundant implications and then removing all of them.



(a) FD-graph of $\mathcal{L}$ (b) Closure FD-graph of $\mathcal{L}$

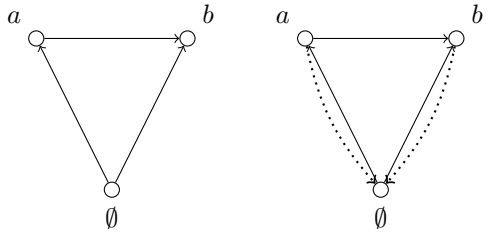
Figure 2.7: Representations of a redundant FD-graph and its closure

FD-Graphs suffer from another drawback: representation of non-closed empty set. To the best of our knowledge, this has not been discussed. While Maier's algorithm is flexible towards open empty set, FD-graphs may not, unless we missed informations. How to represent the basis $\mathcal{L} = \{\emptyset \longrightarrow ab, a \longrightarrow b\}$? We thought of two possible choices:

(i) consider $\emptyset$ as a compound node without dotted arcs, hence like a simple node

(ii) when $\emptyset$ is present, consider all simple nodes as compound, and add a dotted arc for all of them into $\emptyset$

Let us investigate those two representations. As one may have noticed, $\mathcal{L}$ is redundant and we should only keep $\emptyset \longrightarrow ab$. The two ideas are represented in figure 2.8.



(a) Representation (i) (b) Representation (ii)

Figure 2.8: Two possible representations of the empty set in FD-Graphs

In the first representation, we will not remove any implication, since there is no dotted arcs anywhere. On the right-hand side, we would remove simple nodes, namely a, since we have indeed dotted FD-path from a to b. However, we would reduce our attribute set and consequently we would keep only $\emptyset \longrightarrow b$ in $\mathcal{L}$. Therefore, none of those ideas is satisfying. In fact, one possible solution is to add a new element σ to Σ acting as the empty set. Then, for all premises A of $\mathcal{L}$ we add σ as a new element of A. This however brings a solution in pre-processing our implication theory, it does not solve the problem in a graph theoretic manner.

That said, we could argue on two points. First, maybe we should doubt of our interpretation of the

algorithm. Second, let us consider we understood it well, then a possible correction would be to compute both dotted and full closure for all compound nodes of $\mathcal{L}$ to see whether they are redundant or not. If it is the case, we update the graph and its closure(among nodes already computed!) by removing the node. This would be strictly the same operation as Maier's redundancy elimination, plus the cost in time and memory of bidirectional translation from basis to FD-Graph. On top of that, one should consider the cost of removing only one node and all of its outgoing arcs within a graph. While this could be done quite easily in an adjacency matrix representation, the adjacency lists choice (which seems to be the one assumed somehow in Ausiello work) would be more time consuming, namely $O(\mathcal{L})$.

We shall discuss it again later on, but as for now, it seemed to us that Ausiello algorithm requiring bidirectional translation, maybe misleading operations, or equal processing as in the Maier case was not worth implementing.

Removing superfluous nodes

From now on, assume we are given a nonredundant FD-Graph. Let us remove so-called superfluous nodes. A node i is *superfluous* if there is an equivalent node j and a dotted path from i to j. Two nodes i, j are *equivalent* if there are FD-paths $\langle i,\ j\rangle$ and $\langle j,\ i\rangle$. It comes at no surprise that nodes are equivalent exactly when they have the same closure in $\mathcal{L}$. From a theoretical point of view, the minimization algorithm suggests the following operation:

- find a superfluous node i, and an equivalent node j with a dotted path from i to j
- for each full arc ik, we add a full arc jk
- then we remove the node i and all of its outgoing arcs from the graph
- repeat until no more superfluous nodes exist

An example of this procedure is given in the figure 2.9. In this example $\mathcal{L} = ab \longrightarrow e\,;\ a \longrightarrow c\,;\ b \longrightarrow d\,;\ cd \longrightarrow ab$. The node ab is superfluous. Since our basis are reduced, note that removing a superfluous node is removing exactly one implication in $\mathcal{L}$. In this case, the resulting $\mathcal{L}$ will be

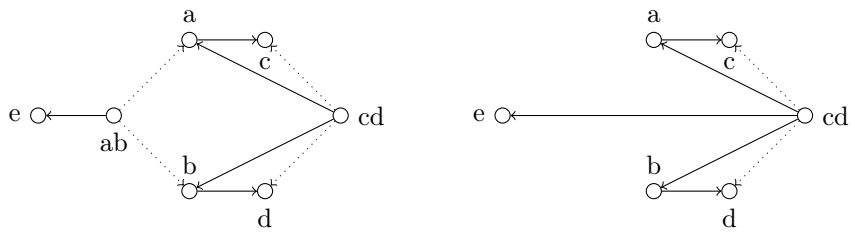
$$\mathcal{L} = a \longrightarrow c, b \longrightarrow d, cd \longrightarrow abe$$

Now we may rewrite this operation in our terms. Let $A \longrightarrow B$ and for instance $C \longrightarrow D$ be part of $\mathcal{L}$ to be general. Then A is superfluous body if

$$\mathcal{L} \models A \longrightarrow C, C \longrightarrow A \wedge \exists X \subset A\ s.t\ \mathcal{L} \models X \longrightarrow C$$

In this case, we apply the following operations

- $C \longrightarrow D$ becomes $C \longrightarrow (D \cup B)$
- we remove $A \longrightarrow B$ from $\mathcal{L}$



(a) FD-Graph with superfluous node (ab)

(b) FD-Graph with superfluous node removed

Figure 2.9: Elimination of superfluous node

In order to prove correctness of this operation, we will show that a node A is superfluous exactly when A directly determines some equivalent B in Maier's terms. Because in both case we are doing same replacement/deletion operation, if two statements are equivalent then the Ausiello algorithm is correct as Maier's one is.

Proposition 7. *The following properties are equivalent (let $A \longrightarrow C$ be the implication of $\mathcal{L}$ with A as body):*

(i) A node A in a FD-graph is superfluous with respect to B,

(ii) $A \equiv B$ and $\mathcal{L} - \{A \longrightarrow C\} \models A \longrightarrow B$.

Proof. (i) $\longrightarrow$ (ii). If A is superfluous, we have a dotted FD-Path $\langle A,\ B\rangle$. Since it is dotted, let us remove this node A and its outgoing arcs. Actually, none of the nodes pointed by dotted arcs of A have been removed, thus we can still find the nodes a_i (attributes of a) used in the dotted path $\langle A,\ B\rangle$ such that $\bigcup_i a_i \models B$. Because $\bigcup_i a_i \subseteq A$, we end up with $\mathcal{L} - \{A \longrightarrow C\} \models A \longrightarrow B$.

(ii) $\longrightarrow$ (i). In the FD-Graph associated to $\mathcal{L} - \{A \longrightarrow C\}$, the node A is not present. But still, $A \longrightarrow B$ holds. This means that we must be able to find a list of proper subsets A_i of A (possibly single attributes) such that $\bigcup_i A_i \models B$. Adding $A \longrightarrow C$ will add the node A and in particular dotted arcs from A to each attributes of $\bigcup_i A_i \subseteq A$. Thus, we will have a dotted path from A to $\bigcup_i A_i$ and consequently, to B. A is indeed superfluous. Because B is equivalent to A by assumption, this property is preserved when adding a node.

□

Proposition 8. *the following statements are equivalent, for A, B bodies of $\mathcal{L}$:*

(i) $A \xrightarrow{\mathrm{d}} B$ and $B \equiv A$,

(ii) the node A is superfluous with respect to B, and there exists a dotted FD-path from A to B not using any outgoing full arcs of nodes equivalent to A.

Proof. (i) $\longrightarrow$ (ii). Using proposition 7 and the equivalence between $A \xrightarrow{\mathrm{d}} B$ and $\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow B$ (from Maier's terminology), showing that there is a direct determination starting from A implies A is a

superfluous node in the FD-Graph is straightforward. If $\mathcal{L} - E_{\mathcal{L}}(A) \subseteq \mathcal{L} - \{A \longrightarrow C\} \models A \longrightarrow B$ so does $\mathcal{L} - \{A \longrightarrow C\}$. This holds in particular if $B \subseteq A$. Moreover, notice that using an outgoing full arc from a node D equivalent to A is exactly using an implication with left hand side equivalent to A. Therefore, if there is not dotted FD-path from A to B not using those arcs, we would contradict direct determination.

(ii) $\longrightarrow$ (i). Suppose A is superfluous and there exists a dotted FD-path from A to B not using any outgoing full arcs from nodes equivalent to A. Those full arcs represent exactly the implications contained in $E_{\mathcal{L}}(A)$. Since we don't use them, the path still holds in $\mathcal{L} - E_{\mathcal{L}}(A)$ (we would remove compound nodes without outgoing full arcs of course, but this would only make the path stops to attributes instead of compound node). Having this path in $\mathcal{L} - E_{\mathcal{L}}(A)$ means that $\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow B$.

□

With propositions 3, 6 and above remarks on operations done in FD-graphs, we can conclude that AusielloMinimization performs from a graph-theoretic point of view the operations made by MaierMinimization in implications framework. Hence both are correct and the resulting FD-graph represents a minimal basis in our terms. As aforementioned, we did not implement this graph-theoretic approach not only because of a possible misunderstanding or mistake in the first step, but also because the operations are the same as in MaierMinimization except that they are moved to a framework where we would need translation procedure (hence overhead in time and memory). However, the superflousness elimination is claimed to be sometimes faster than in Maier's case asymptotically. Unfortunately, we were unable to find a practical representation matching both the expected size of the FD-Graph data structure and the complexity of elementary operations like removing a node. Eventually, this algorithm could be considered as a weak point of our study since for all this reason taking time to investigate, we chose to focus on other algorithms. Nevertheless, as all this section shows, we still provided some theoretical comparison to exhibit the heart of the procedure being a translation from a framework to another. The advantages of FD-graphs are the representation in simple terms of some implication theories, and the difference between dotted and full arcs allowing for some separations in nodes.

discussion about complexity and effective algorithm to delete superfluous nodes.

2.4 Propositionnal logic based approach

2.4.1 Representing implications as Horn clauses

2.4.2 Hypergraphs out of a bounding theorem

Here, we will study an algorithm proposed by Berczi and al. in [15] following a paper by Boros and al. ([14]). Readers having a glance at the paper previously cited will see different notations and framework between our study and the one performed by the authors. This is because they use a graph-theoretic ground equivalent to ours, but as we previously said, in order to stay in a somehow coherent set up all along this report, we will discuss in terms of implications and so forth.

The main idea of the algorithm we should keep in mind, is to build iteratively the DG basis. To describe

briefly the procedure in words, having an initial basis $\mathcal{L}$: we initialize $\mathcal{L}_c = \emptyset$ and then at each step of the algorithm we add a new implication $A \longrightarrow \mathcal{L}(A)$ in $\mathcal{L}_c$ such that A is pseudo-closed in $\mathcal{L}$. By construction then, we will terminate and end up with the DQ basis. Now that the process is defined, let us expose the procedure and discuss it (see algorithm 14)

Algorithm 14: BERCZIMINIMIZATION

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}_c$: the DQ-basis of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
while $\exists B \in \mathcal{B}(\mathcal{L})$ *s.t* $\mathcal{L}_c(B) \neq \mathcal{L}(B)$ **do**
  $P := \min\{\mathcal{L}_c(B),\ B \in \mathcal{B}(\mathcal{L}) \text{ and } \mathcal{L}_c(B) \neq \mathcal{L}(B)\}$;
  $\mathcal{L}_c := \mathcal{L}_c \cup \{P \longrightarrow \mathcal{L}(P)\}$;
return $\mathcal{L}_c$;

In [15, 14], pseudo-closure is not really considered. Instead, an implication of the form $P \longrightarrow \mathcal{L}(P)$ where P is pseudo-closed, is called *left-right-saturated*. To stay close to our definition of the canonical basis, we provide an proposition for the correctness of this algorithm, based on pseudo-closed sets:

Proposition 9. *In algorithm 14, we add an implication $P \longrightarrow \mathcal{L}(P)$ only if P is pseudo-closed.*

Proof. Let us prove this proposition by induction.

Initial Case The initial case is the first implication we add to $\mathcal{L}_c$. Because $\mathcal{L}_c$ is empty, for all $B \in \mathcal{B}(\mathcal{L})$, $\mathcal{L}_c(B) = B$. Thus, we add to I_c an implication $B \longrightarrow \mathcal{L}(B)$ where B is minimal inclusion-wise among bodies of $\mathcal{L}$. Recalling our definition of pseudo-closure, B is compelled to be pseudo-closed then. Note that in fact, this argument will hold for all minimal bodies inclusion-wise. Hence, the proposition is true for the initial case.

Induction Suppose we added only implications with pseudo-closed sets as premises in $\mathcal{L}_c$. We will show that in the next implication $P \longrightarrow \mathcal{L}(P)$ we add, P is pseudo-closed. Take P as mentioned in the algorithm. First observe that taking the minimal non-closed sets of $\mathcal{L}$ closed in $\mathcal{L}_c$ generated by bodies of $\mathcal{L}$ is sufficient to have the minimal such sets in general. Indeed, let X be a closed set of $\mathcal{L}_c$ not closed in $\mathcal{L}$. Then, because bodies are minimal in non-closed sets of $\mathcal{L}$, there must exist implications $\alpha \longrightarrow \beta$ in $\mathcal{L}$ such that $\alpha \subseteq X$. In particular, we must have an implication $\alpha_i \longrightarrow \beta_i$ such that $\alpha_i \subseteq X$ and $\beta_i \nsubseteq X$, because X is not closed. Now by construction of the algorithm, we have the following for all implications $A \longrightarrow B$ of $\mathcal{L}$: either $\mathcal{L}_c(A) \longrightarrow \mathcal{L}(A)$ belongs to $\mathcal{L}_c$, either it does not (here $\mathcal{L}_c(A)$ is the closure of A before adding $\mathcal{L}_c(A) \longrightarrow \mathcal{L}(A)$ to $\mathcal{L}_c$). Because $\beta_i \nsubseteq X = \mathcal{L}_c(X)$ we can conclude that $\mathcal{L}_c(\alpha_i) \longrightarrow \mathcal{L}(\alpha_i) \notin \mathcal{L}_c$. Thus X will not be minimal $\mathcal{L}_c$-closed $\mathcal{L}$-non-closed unless it is the closure of some body of $\mathcal{L}$. Because we are sure to take a minimal $\mathcal{L}_c$-closed $\mathcal{L}$-non-closed set at each step, we are sure to have all possible pseudo-closed sets $P_i \subset P$ when considering P. Furthermore, since we take P to be the minimal close set of $\mathcal{L}_c$, $\mathcal{L}(P_i) \subseteq P$

for all P_i. Hence P is indeed pseudo-closed, which confirms the induction hypothesis and the property in general.

□

This statement saying that if we add an implication, then its premise is pseudo-closed is enough to justify termination of the algorithm on DQ basis. The outer while loop will be executed at most $|\mathcal{B}|$ times since at each step, we take out another body of $\mathcal{L}$. Computing and finding the next pseudo-closed set in this case is done in $O(|\mathcal{B}||\mathcal{L}|)$ operations (an execution of LinClosure for each implication of $\mathcal{L}$), thus resulting in an $O(|\mathcal{B}|^2|\mathcal{L}|)$ asymptotic complexity for the whole algorithm. Even though simple in its form, it is much more time consuming than previous studied algorithms. Before discussing any improvement, we will give an example.

Example To be coherent, let us take again our perpetual example:

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

Let us illustrate the algorithm through a graphical trace (see figure 2.10). In this figure, we represented the 4 steps of BercziMinimization over $\mathcal{L}$ as follows: on the left-hand side of each step, one can find the closures of premises of $\mathcal{L}$ under $\mathcal{L}_c$, denoted $\mathcal{L}_c(\mathcal{B}(\mathcal{L}))$, ordered by inclusion ($\subseteq$). On the right-hand side, we have the closures of premises of $\mathcal{L}$ under $\mathcal{L}$, that is $\mathcal{L}(\mathcal{B}(\mathcal{L}))$, again ordered by inclusion.

In fact, Berczi procedure is a matter of comparing those two orderings. At each step, we should consider all premises B of $\mathcal{L}$ such that $\mathcal{L}(B)$ is not an element of $(\mathcal{L}_c(\mathcal{B}(\mathcal{L})), \subseteq)$. Among those premises, we take one with a minimal $\mathcal{L}_c$-closure. Then, adding $\mathcal{L}_c(B) \longrightarrow \mathcal{L}(B)$ to $\mathcal{L}_c$ ensures in next steps, we will not have to consider elements of $\downarrow \mathcal{L}(B)$ in $(\mathcal{L}(\mathcal{B}(\mathcal{L})), \subseteq)$. In details (a point refers to a step in the figure):

(a) $\mathcal{L}_c = \emptyset$, so for all premises B of $\mathcal{L}$, $\mathcal{L}_c(B) = B$. Hence we take c as a premise with minimal $\mathcal{L}_c$-closure, and append $c \longrightarrow ac$ to $\mathcal{L}_c$.

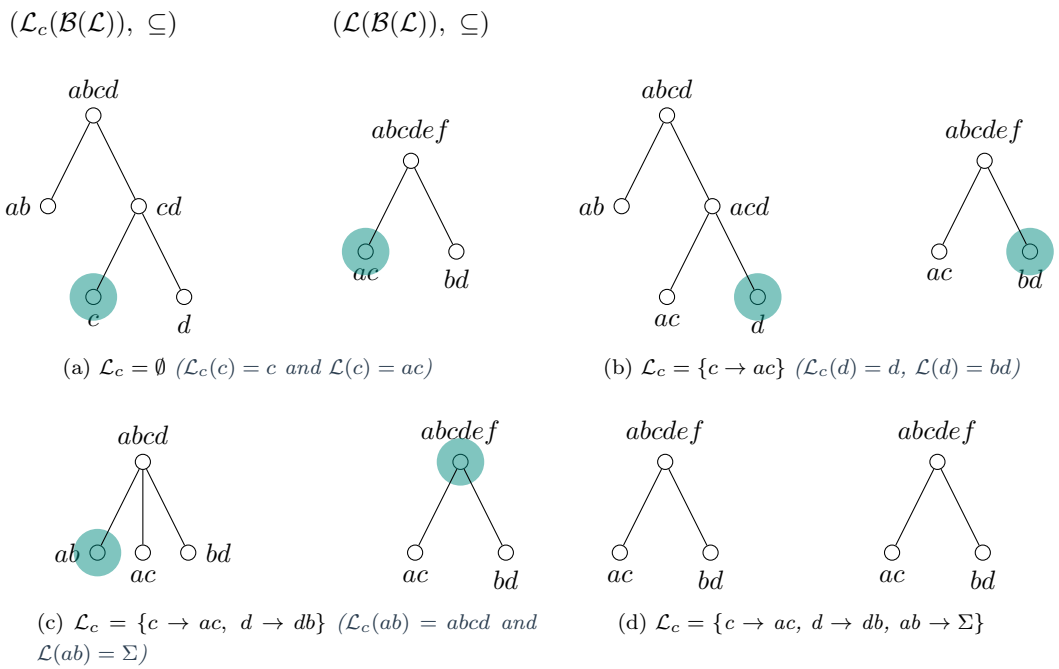
(b) $\mathcal{L}_c = \{c \longrightarrow ac\}$. d is a premise of $\mathcal{L}$ being closed in $\mathcal{L}_c$, hence minimal. Consequently, we add $d \longrightarrow bd$ to $\mathcal{L}_c$.

(c) $\mathcal{L}_c = \{c \longrightarrow ac, d \longrightarrow bd\}$, the closures of c and d, are the same in $\mathcal{L}_c$ and in $\mathcal{L}$. It remains then ab, cd and $abcd$. In $\mathcal{L}_c$, we have:

- $\mathcal{L}_c(ab) = ab$,
- $\mathcal{L}_c(cd) = abcd$,
- $\mathcal{L}_c(abcd) = abcd$,

thus the minimal one is $\mathcal{L}_c(ab) = ab$ and we add $ab \longrightarrow abcdef$ to $\mathcal{L}_c$,

(d) $\mathcal{L}_c = \{c \longrightarrow ac, d \longrightarrow bd, ab \longrightarrow abcdef\}$, for all $B \in \mathcal{B}(\mathcal{L})$, $\mathcal{L}_c(B) = \mathcal{L}(B)$, the two orderings are identical (or *isomorphic*), $\mathcal{L}_c$ is equivalent to $\mathcal{L}$ and canonical whence minimal.



(a) $\mathcal{L}_c = \emptyset$ *($\mathcal{L}_c(c) = c$ and $\mathcal{L}(c) = ac$)*

(b) $\mathcal{L}_c = \{c \to ac\}$ *($\mathcal{L}_c(d) = d$, $\mathcal{L}(d) = bd$)*

(c) $\mathcal{L}_c = \{c \to ac,\ d \to db\}$ *($\mathcal{L}_c(ab) = abcd$ and $\mathcal{L}(ab) = \Sigma$)*

(d) $\mathcal{L}_c = \{c \to ac,\ d \to db,\ ab \to \Sigma\}$

Figure 2.10: Trace of BerczIMinimization execution

Possible improvement As we said, this algorithm is much less efficient than MinCover or even than MaierMinimization. The problem may come from the need to re-compute the closure of bodies in $\mathcal{L}$ under $\mathcal{L}_c$ at each step to find a possible minimum. Thus, an improvement would be to order premises so that the algorithm becomes:

Input: $\mathcal{L}$: a theory to minimize
Output: $\mathcal{L}_c$: canonical basis associated to $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
ORDER($\mathcal{L}$) ;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  if $\mathcal{L}_c(A) \neq \mathcal{L}(A)$ **then**
    $\mathcal{L}_c := \mathcal{L}_c \cup \{\mathcal{L}_c(A) \longrightarrow \mathcal{L}(A)\}$;
return $\mathcal{L}_c$;

where ORDER would be an order on premises of $\mathcal{L}$ in accordance with the partial order defined by pseudo-closed sets. Unfortunately, it seems like neither lectic or premise-size ordering are valid though they are compatible with inclusion partial order.

In this section we are interested in the Angluin algorithm (1992). [2, 3].

2.4.3 Angluin algorithm and AFP: Query Learning based approach

Here, the method for building a minimal base is slightly different. We use so-called *query learning*. The idea is we formulate *queries* to an *oracle* knowing the basis we are trying to learn. The oracle is assumed to provide an answer to our query in constant time. Depending on the query, it might also provide informations on the object we are looking for. For the Angluin algorithm, we need 2 types of queries. Say we want to learn a basis $\mathcal{L}$ over Σ:

1. *membership* query: is $M \subseteq \Sigma$ a model of $\mathcal{L}$? The oracle may answer "yes", or "no".
2. *equivalence* query: is a basis $\mathcal{L}'$ equivalent to $\mathcal{L}$? Again the answers are "yes", or "no". In the second case, the oracle provides a *counterexample* either positive or negative:
   (i) *positive*: a model M of $\mathcal{L}$ which is not a model of $\mathcal{L}'$,
   (ii) *negative*: a non-model M of $\mathcal{L}$ being a model of $\mathcal{L}'$.

To clarify, the terms negative/positive are related to the base $\mathcal{L}$ we want to learn. ANGLUINALGORITHM is the algorithm presented by Angluin, Frazer and Pitts in [2] as HORN1. Initially, it is based on learning logical representation of implication theories: Horn clauses. This learning algorithm has been shown first to terminate on a minimal representation of the basis we want to learn ([2]) and more than that, to end up on the canonical basis by Arias and al. [3]. It uses two operations allowing to reduce implications:

- $refine(A \longrightarrow B, M)$: produces $M \longrightarrow \Sigma$ if $B = \Sigma$, $M \longrightarrow B \cup A - M$ otherwise,
- $reduce(A \longrightarrow B, M)$: produces $A \longrightarrow M - A$ if $B = \Sigma$, $A \longrightarrow B \cap M$ otherwise.

In practice, this procedure is likely to be somehow random because of the oracle. To get rid of non-determinism, one can derive from ANGLUINALGORITHM the algorithm AFPMINIMIZATION.

Algorithm 15: AngluinAlgorithm

Input: $\mathcal{L}$
Output: $\mathcal{L}_c$

$\mathcal{L}_c = \emptyset$;
while *not equivalence*($\mathcal{L}_c$) **do**
  M is the counterexample ;
  if M *is positive* **then**
    foreach $A \longrightarrow B \in \mathcal{L}_c$ *such that* $M \not\models A \longrightarrow B$ **do**
      replace $A \longrightarrow B$ by $reduce(A \longrightarrow B, M)$;
  else
    foreach $A \longrightarrow B \in \mathcal{L}_c$ *such that* $A \cap M \subset A$ **do**
      membership($M \cap A$) ;
    if *Oracle replied "no" for at least one* $A \longrightarrow B$ **then**
      Take the first such $A \longrightarrow B$ in $\mathcal{L}_c$;
      replace $A \longrightarrow B$ by $refine(A \longrightarrow B, A \cap M)$;
    else
      add $M \longrightarrow \Sigma$ to $\mathcal{L}_c$;
return $\mathcal{L}_c$;

Algorithm 16: AFPMinimization

Input: some theory $\mathcal{L}$ over Σ

Output: $\mathcal{L}_c$ the Duquenne-Guigues basis of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
Stack $\mathcal{S}$;
forall $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{S} := [A]$;
  repeat
    $X := \mathcal{L}_c(\text{pop}(\mathcal{S}))$;
    if $X \neq \mathcal{L}(X)$ **then**
      $found := \bot$;
      forall $\alpha \longrightarrow \beta \in \mathcal{L}_c$ **do**
        $C := \alpha \cap X$;
        if $C \neq \alpha$ **then**
          $D := \mathcal{L}(C)$;
          if $C \neq D$ **then**
            $found := \top$;
            change $\alpha \longrightarrow \beta$ by $C \longrightarrow D$ in $\mathcal{L}_c$;
            push($X \cup D$, $\mathcal{S}$) ;
            if $\beta \neq D$ **then**
              push(α, $\mathcal{S}$);
            exit for
      if $found = \bot$ **then**
        $\mathcal{L}_c := \mathcal{L}_c \cup \{X \longrightarrow \mathcal{L}(X)\}$;
  until $\mathcal{S} = \emptyset$;
return $\mathcal{L}_c$;

In this function, questions to and answers from the oracle are replaced by a stack and closure operations. Indeed, membership queries can be done by computing closures under $\mathcal{L}$, $\mathcal{L}_c$. Regarding the stopping criterion equivalence query, observe that premises of $\mathcal{L}$ are a sufficient set of negative counter-example to find $\mathcal{L}_c$: when a premise is no more a negative counter-example, it means that it is neither closed in $\mathcal{L}$ nor in $\mathcal{L}_c$. Furthermore, since implications of $\mathcal{L}_c$ are $\mathcal{L}$-right-closed, every premise A of $\mathcal{L}$ has the same closure in $\mathcal{L}$ and $\mathcal{L}_c$. Hence, when there is no premise left being negative counter-example, $I \equiv \mathcal{L}_c$. Note that this approach does not require positive counter-examples.

Up to know, we are not sure of the correctness of the algorithm, since we do not know whether an implication can be refined twice, hence if the stack can grow more than twice the number of implications in $\mathcal{L}$ (asymptotically). Still, if an implication is refined more than one time, the elements stacked shrink in size. Hence we are sure to have a finite loop. This leads to an approximation of the complexity. Indeed, even though we do not know the number of time one could iterate over the repeat loop, we still have nested iterations over implications of $\mathcal{L}$ and $\mathcal{L}_c$. Furthermore, because the run over implications of $\mathcal{L}_c$ contains computations of elements in $\mathcal{L}$, the overhaul algorithm may be cubic in the number of premises in $\mathcal{L}$. Therefore, this would be the worst time complexity.

Example Let us observe a trace with our toy example:

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

We will proceed by implications in $\mathcal{L}$:

1. $ab \longrightarrow cde$, $\mathcal{L}_c = \emptyset$. We want to build a negative example with ab. The smallest closed set of $\mathcal{L}_c$ containing ab is $\mathcal{L}_c(ab) = ab$. ab is not closed in $\mathcal{L}$, hence it is itself a counter-example. Because $\mathcal{L}_c$ is empty, we can only add $ab \longrightarrow abcdef$ to $\mathcal{L}_c$.

2. $cd \longrightarrow f$, $\mathcal{L}_c = \{ab \longrightarrow abcdef\}$. Again, cd is a negative counter example to $\mathcal{L}$: it is a model of $\mathcal{L}_c$, but not of $\mathcal{L}$. Furthermore, the only smaller example we build is $ab \cap cd = \emptyset$ not being a negative one. Hence, we add $cd \longrightarrow abcdef$ to $\mathcal{L}_c$.

3. $c \longrightarrow a$, $\mathcal{L}_c = \{ab \longrightarrow abcdef, cd \longrightarrow abcdef\}$. c is a negative example: $\mathcal{L}_c(c) = c \neq \mathcal{L}(c)$. For all implications of $\mathcal{L}_c$, we try to compute a new example:

   - $ab \cap c = \emptyset$, which is not a negative counter example,
   - $cd \cap c = c$. It is a negative example.

   For c, we replace $cd \longrightarrow abcdef$ by $c \longrightarrow ca$ to correct $\mathcal{L}_c$. We push ca in the stack since it may generate an counter-example. Furthermore, because we replaced $cd \longrightarrow abcdef$ by $c \longrightarrow ca$ and $abcdef \neq ca$, the closure of cd under $\mathcal{L}_c$ is no more its closure under $\mathcal{L}$, it may also produce a counter-example, we push it into the stack.

   The next attribute set in the stack is cd. The corresponding example is $\mathcal{L}_c(cd) = acd$ not closed in $\mathcal{L}$:

- $acd \cap ab = a$: a is closed both under $\mathcal{L}$ and $\mathcal{L}_c$,
- $acd \cap c = c$: c has the same closure under $\mathcal{L}$ and $\mathcal{L}_c$.

we found no counter example, we add $acd \longrightarrow abcdef$ to $\mathcal{L}_c$.

The last element of the stack is ca. It is closed under $\mathcal{L}$ hence it cannot be a counter example.

4. $d \longrightarrow b$, $\mathcal{L}_c = \{ab \longrightarrow abcdef, c \longrightarrow ca, acd \longrightarrow abcdef\}$. $\mathcal{L}_c(d) = d \neq \mathcal{L}(d)$ is a negative counter example:
   - $ab \cap d = \emptyset$
   - $c \cap d = \emptyset$
   - $acd \cap d = d$, being a negative counter example

   For d, we replace $acd \longrightarrow abcdef$ by $d \longrightarrow bd$. Then we push it then stack bd and because $bd \neq abcdef$, we also push acd.

   The next element in the stack is acd for which the closure under $\mathcal{L}_c$ is the same as the closure under $\mathcal{L}$: $abcdef$. It is not a counter example.

   The last element before the stack goes empty is bd being a model of $\mathcal{L}$.

5. $abcd \longrightarrow ef$, $\mathcal{L}_c = \{ab \longrightarrow abcdef, c \longrightarrow ca, d \longrightarrow bd\}$. $\mathcal{L}_c(abcd) = abcdef = \mathcal{L}(abcd)$. It is not a counter example.

The resulting basis $\mathcal{L}_c = \{ab \longrightarrow abcdef, c \longrightarrow ca, d \longrightarrow bd\}$ is indeed the DG basis of $\mathcal{L}$.

2.5 Theoretical expectations and conclusion

All along previous sections, we discussed in detail several minimization algorithms. To summarize, we studied:

- MinCover with complexity $O(|\mathcal{B}||\mathcal{L}|)$,
- DuquenneMinimization with complexity $O(|\mathcal{B}||\mathcal{L}|)$,
- MaierMinimization, $O(|\mathcal{B}||\mathcal{L}|)$,
- BercziMinimization, $O(|\mathcal{B}|^2|\mathcal{L}|)$,
- AFPMinimization, with an unknown time complexity.

Note that we did not truly listed all the algorithm we have been talking about more than the algorithms we will indeed study in practical terms in the next chapter. As one can observe, at least 3 of the algorithms have the same complexity even though being quite different. It turns out anyway that among the first three algorithms, MinCover, DuquenneMinimization are quite similar. Indeed they both rely on using quasi-closure, right-closure and redundancy elimination. Those three steps first issued in works by Day in [19], Duquenne-Guigues in [20, 23], Wild in [28, 29, 27] have been recently discussed by Boros and al

in [14]. MaierMinimization however relies on quite different structure. The most interesting difference with the two previous algorithms (or in fact all other algorithms but AusielloMinimization being based on MaierMinimization) is that the resulting implication base may not be the Duquenne-Guigues basis. Still, it uses redundancy elimination even though an implication being redundant in MinCover may not be considered as redundant in Maier's terms (because of right-closure in the latter one). Nevertheless, we still have a common point within those three first procedures: we are minimizing a given basis. Quite the opposite, BerczиMinimization and AFPMinimization are building a canonical representation of an input basis. Speaking of theoretical complexity, it seems like those approaches are not the most efficient. Still, we may keep in mind that we based our study of complexity on LinClosure which may not be the fastest closure algorithm in practice (see [9]).

Conclusion In this chapter, we studied several algorithms for minimization task coming from various communities. Starting from algorithms heavily using quasi-closure of FCA community (MinCover, DuquenneMinimization), moving to Maier's database approach and Ausiello graphical representation. Eventually we studied algorithms from boolean logic and query learning with BerczиMinimization and AFPMinimization. After having studied their operations we will in the next chapter implement and compare them under practical computations.

Chapter 3

Implementation

3.1 Test set up

3.1.1 Tools

Here, we understand by tools the general implementation ground we rely on: the languages, some classes, file formatting and so forth. Let us begin by a very large point of view. We use two languages: C++ (11) for implementing the algorithms and testing. For data visualization we use python with libraries `matplotlib`, `numpy`, `csv`. To be more precise about C++, we use MinGW-64 compiler and `-O3` optimization flag. Our tests are timed with release builds. We recorded CPU-time, not wall-clock time which consider also the time spent outside the program. CPU is an Intel I5, 2.4 GHz.

We used the code from `https://github.com/yazevnul/fcai`, used in [9] for experience on LinClosure. It has been made under FCA context. In this framework, sets are represented as `boost::dynamic_bitset` instances (named `BitSet`). Because `boost` was already present in the project and has various built-in tools, we used it for timing and recording purposes with `boost::timer` and `boost::accumulators` respectively.

On top of implementation of the algorithms we wrote few tools to ease I/O and testing. Even though we will not go into details, let us introduce quickly main use of those tools, especially the file format we chose. Because we may need to load or save some implication theories, we have to think of a file format able to represent such basis easily. For this aim, we made `.imp` files, structured as follows:

- the first line contains the number of attributes and the number of implications, separated by a space,
- because of the `Bitset` representation, an implication is written as lists of indices, premise and conclusion, separated by ">". Indices are also space separated.

For instance, consider our running example from the previous chapter: $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$ and $\mathcal{L} = \{ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\}$. Considering a indexed as 0 and f indexed as 5, the corresponding `.imp` file will be:

```
6 5
0 1 > 2 3 4
2 3 > 5
2 > 0
3 > 1
0 1 2 3 > 4 5
```

Using functions `ReadFile` and `WriteFile` in the namespace `ImplicationTools`, one can easily read or write implication basis:

```
theory L, Lc;  // alias for std::vector<FCA::ImplicationInd>
ImplicationTools::ReadFile("input_file.imp", L);
Minimize(L, Lc);  // result of some minimization into Lc
ImplicationTools::WriteFile("output_file.imp", Lc);
```

Note that the meaning of attributes here is somehow lost. So far, this is not an issue and it may be fixed by keeping a trace of correspondence between indices and attributes names. To test minimization, we use a class `GridTester` writing CSV results.

- csv exportation, bitfiles
- quickly describe the class GridTest

3.1.2 Randomly generated data

This paragraph is important for all following experiences. We would like to put the emphasis on the way we randomly generate sets and implications. First, let us focus on set generation. We use `boost::random` and `time` libraries for generating pseudo-random numbers. In particular for a set X, we use discrete uniform random distribution on the interval $[0\ ;\ |\Sigma|]$:

1. determine the size $|X|$ of X by drawing a number out of our distribution,
2. draw again $|X|$ numbers. Because of the interval, we are sure to obtain valid indices of elements to set in X.

Note that an element can be drawn more than once, resulting in effective $|X|$ smaller than the one we got at the beginning. We do not consider this as an issue. Generating theories then is as follows:

1. generate a conclusion randomly,
2. generate a premise. Because we want implications to be informative, we keep as a premise the difference *premise* $-$ *conclusion*.
3. because empty premise is likely to occur several times, resulting in $\emptyset \longrightarrow \Sigma$, we allow for no more than one empty premise. Nevertheless, in order not to loop forever with this condition, we fixed a maximal amount of re-roll with a variable `MAX_ITER`. Passed this number of failure, we accept an empty premise anyway.

Note that this method can be discussed and probably improved. For us, it seems like this method is sufficient to provide theories with informative implications, and uniformly distributed sizes of premises and conclusions thus a "good" representation of the theories space. We did not investigate further ways of generations since the main task was to test algorithms and not to study implication structures in depth. This anyway is an interesting question for further work.

3.1.3 Real data

In this part we will be interested in application to real datasets. The application we used is FCA since the framework we use is dedicated to FCA testing. We will first present briefly FCA and its correlation with our minimization issue, before describing some real datasets we have been using and their characteristics.

Introduction to FCA

Formal Concept Analysis is a technique relying on array-like data and lattices to describe hierarchies in data. It can be used in data mining, text mining or chemistry for instance. Usually we are given a set G of *objects* having some *attributes* of a set M. Between G and M we can define a binary relation I. An object g is related to an attribute m, gIm, if g has the attribute m. The tuple (G, M, I) is called a *context*.

Quite intuitively, we can define an operation $' : 2^G \longrightarrow 2^M$, associating to a set of objects $A \subseteq G$ the set $B \subseteq M$ of attributes shared by all elements of A. Conversely, we can set $' : 2^M \longrightarrow 2^G$ yelling the set A of objects sharing all attributes of some set B. Formally:

$$A' = \{m \in M \mid \forall g \in A,\ gIm\} \quad \forall A \subseteq G$$
$$B' = \{g \in G \mid \forall m \in B,\ gIm\} \quad \forall B \subseteq M$$

When combined together those operators define a closure operator $'' : 2^M \longrightarrow 2^M$. Very surprisingly, this operator is related with attribute implications. Indeed, given a context (G, M, I) we can draw implications between subsets $A \longrightarrow B$ of attributes. Intuitively, an implication $A \longrightarrow B$ will be valid if every time we have all attributes from A, we also have attributes from B: $A' \subseteq B'$ or equivalently, $B \subseteq A''$. A set $\mathcal{L}$ of implications over M will be complete and sound if the operator $''$ from the context coincide with $\mathcal{L}(\cdot)$ for all $A \subseteq M$. For the recall, $\mathcal{L}$ is *complete* with respect to a context $\mathbb{K} = (G, M, I)$ if all valid implications of $\mathbb{K}$ hold in $\mathcal{L}$, in other words $\mathcal{L}$ contains all informations of $\mathbb{K}$. $\mathcal{L}$ is *sound* with respect to $\mathbb{K}$ if all valid implications of $\mathcal{L}$ are also valid in $\mathbb{K}$, that is, $\mathcal{L}$ contains only true informations about $\mathbb{K}$.

Example Pages and chapters ago, we were talking about flowers and vegetables. Let us imagine we have the following:

- plants as G, $G = \{$ *cactus, water lily, apple tree, sea weed, birch* $\}$,
- attributes as M, $M = \{$ *aquatic, perennial, flower, seasonal* $\}$

We can represent a context $\mathbb{K} = (G, M, I)$ as an array, see table 3.1.

	aquatic	perennial	flower	seasonal
cactus		×	×	×
water lily	×		×	×
apple tree			×	×
sea weed	×			
birch				×

Table 3.1: Example of a small context

In this context we have for instance $\{aquatic\}' = \{water\ lily,\ sea\ weed\}$ and $\{cactus,\ apple\ tree,\ water\ lily\}' = \{flower,\ seasonal\}$. $\{perennial\} \longrightarrow \{\ flower,\ seasonal\}$ is an example of valid implication, while $\{aquatic\} \longrightarrow \{\ flower,\ seasonal\}$ is not since *sea weed* is *aquatic* but neither has *flower* nor is *seasonal*.

As shown, contexts contain binary informations. Unfortunately, we may usually be confronted to multi-valued data. If we stick to our green example, we could have an attribute "size" having possibly many values. With such attributes we could get closer from relational databases. Because real datasets we are going to use contains multi-valued attributes, we will need *FCA-scaling*. We will not go into technical details. The idea however is the following:

- for continuous attributes (e.g: size of a plant) we can create disjoint classes based on intervals allowing for new binary attributes
- for discrete attributes (e.g: zone of living) we can create an attribute per existing value.

In our case, considering discrete values is sufficient according to real datasets we will rely on.

Example Let us retake our previous example but adding new attribute: *zone of living*. The context may become table 3.2. Of course, remind that we do not assume any true knowledge about biology, this stands only for appealing examples.

	aquatic	perennial	flower	seasonal	zone of living
cactus		×	×	×	dry area
water lily	×		×	×	water
apple tree			×	×	woods
sea weed	×				water
birch				×	woods

Table 3.2: Example of a (discrete) multi-valued attribute

To be able to work in FCA framework, we may refactor the last attributes into 3 distinct ones, as in table 3.3.

Eventually, we shall discuss various possible basis one can build out of a context. We will use 4 of them:

	aquatic	perennial	flower	seasonal	dry area	water	woods
cactus		×	×	×	×		
water lily	×		×	×		×	
apple tree			×	×			×
sea weed	×					×	
birch				×			×

Table 3.3: Example of FCA-scaling for multi-valued attribute

the canonical basis, the minimal basis resulting from Maier's algorithm, the basis of minimal generators and the proper basis. We already discussed the two first ones. We derive them from the non minimal ones:

(i) minimal generators: a set X is a minimal generator of its closure $\mathcal{L}(X)$ if it is minimal in $[X]_{\mathcal{L}}$,

(ii) proper implications: implications $A \longrightarrow A^{\bullet}$ where $\bullet$ is a saturation operator

Some real datasets

For our tests, we took the datasets used in [9] `https://archive.ics.uci.edu/ml/datasets.html`. They are the following ones:

- *"Zoo"*: it contains animals described by attributes such as "mammals", their number of legs and so forth. Because all attributes are discrete valued, the dataset is scaled as explained previously. For 101 animals, we go from 17 attributes to 28 by scaling.
- *"Flare"*: describes various solar flares by their position, size, flare classification, activity... With 1389 objects, we ended up on 49 attributes by flattening all multi-valued ones.
- *"Breast cancer"*:
- *"Breast Wisconsin"*:
- *"Post-operative"*: this dataset is dedicated to determine in which service should a patient go after its operation. All parameters are indicators on the metabolism of the patient such as body temperature, blood pressure and so forth. For 90 patients and 8 attributes, we scaled to 26 attributes.
- *"SPECT"*:
- *"Vote"*:

To summarize characteristics of all datasets, we may observe table 3.4. For the recall $|\Sigma|$ is the number of attributes, and $|\mathcal{B}|$ the number of implications for each possible dataset.

In all this section we have been interested in describing the tools we got or developed for implementing the algorithms and data we used. Regarding tools and language, we use C++ and MinGW-64 compiler for implementing and testing the algorithms, based on the code provided in [9]. The CPU has 2.4 GHz frequency. Apart from generating random implications out of `boost` and uniform distribution, we took

$\mathcal{L}$		$\lvert\Sigma\rvert$	$\lvert\mathcal{B}\rvert$
Zoo	minimal	28	141
	generators		874
	proper		554
Flare	minimal	49	3382
	generators		39787
	proper		10692
Breast cancer	minimal	43	3352
	generators		16137
	proper		11506
Breast Wisconsin	minimal	91	10640
	generators		51118
	proper		45748
Post operative	minimal	26	625
	generators		3044
	proper		1721
SPECT	minimal	23	2169
	generators		44341
	proper		8358
Vote	minimal	18	849
	generators		8367
	proper		2410

Table 3.4: Summary of real datasets characteristics

various real datasets from the UCI repository. Those real data we used are the same as ones used when testing the code we used on closure algorithms. In the next section we will study implementation of each algorithms, before concluding by a final comparison on the real datasets we evoked.

3.2 Pruning the algorithms

Because we can plug-in various closure procedures (Closure, LinClosure), we are interested in knowing which configuration is the most efficient for each minimization procedure. However, because we may have several possible configurations and tests can be highly time consuming, we will rely on the next assumption based on the result of [9]. Actually, it is exhibited in this paper that LinClosure performs worst in general than Closure. However, as we will see there are some possible optimizations for LinClosure dealing with initialization steps. Consequently, we will first give priority to Closure, and try to replace it by LinClosure whenever we find a possible optimized use of for this closure method. We will keep as the "best version" the most efficient in the tests we will run.

3.2.1 MinCover

Recall that MinCover is a two-steps algorithm: right-closure and redundancy elimination. et $\mathcal{L}$ over Σ be the basis we are trying to minimize. In the first step, we do not remove or add any implications from $\mathcal{L}$. Furthermore, we do not alter its premises. Therefore, when using LinClosure, we may need to initialize counters and list only one time. However, in the second loop, it is question of first removing an implication from $\mathcal{L}$. From our point of view, removing only one implication $A \longrightarrow B$ in counters may be as complex as LinClosure itself:

- if the data structure used for *list* in LinClosure is a chained list, then removing $A \longrightarrow B$ from some $list[a]$, $a \in A$ is $O(|\mathcal{B}|)$. Because this has to be done for all $a \in A$, the overhaul operation should be $O(|\mathcal{L}|)$, like LinClosure and in particular, the initialization step
- if the data structure is an array, there are again two possibilities. Either we store in $list[a]$ directly implications or indices of implications in $\mathcal{L}$, but then finding and removing an implication will be $O(|\mathcal{L}|)$. Or we can use marking procedure to store a boolean value at some index i representing the $i-th$ implication to know whether or not the i-th implication should belong to $list[a]$. Removing an implication then may be $O(|\Sigma|)$. However, we should take care of updating all the boolean values if we remove an implication from $\mathcal{L}$, because the i-th index may not correspond to the implication i anymore. To avoid this update, we can in fact do not remove implications in $\mathcal{L}$ at all and put all redundant ones in some trash-marked state. This would require some more conditional statements in the nested for loop of the second step of LinClosure, but it may offer indeed a slight optimization.

We did not test the last possible optimizations due to lack of time, but also because of the results we shall exhibit hereafter about MinCover, stressing on the bad behaviour of LinClosure in practice. Therefore, the two pseudo-codes we compared for MinCover are algorithms 18, 19.

Algorithm: MinCoverClo

Input: $\mathcal{L}$: an implication base
Output: the canonical base of $\mathcal{L}$

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $B := \text{Closure}(\mathcal{L}, A \cup B)$;
  $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $A := \text{Closure}(\mathcal{L}, A)$;
  if $A \neq B$ **then**
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

Algorithm: MinCoverLin

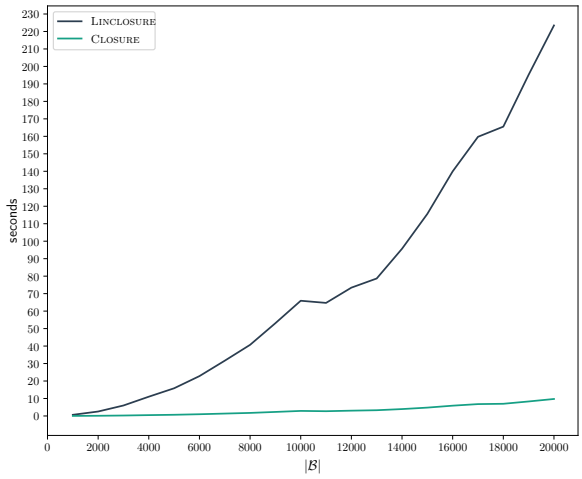
Input: $\mathcal{L}$: an implication base
Output: the canonical base of $\mathcal{L}$

LinClosureInit($\mathcal{L}$) ;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $B := \text{LinClosure}(\mathcal{L}, A \cup B)$;
  $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

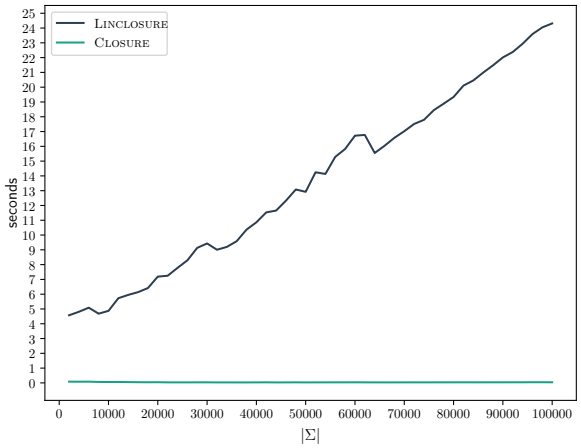
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$;
  $A := \text{Closure}(\mathcal{L}, A)$;
  if $A \neq B$ **then**
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

In MinCoverLin we used a function called LinClosureInit. In fact, this function is the initialization step of LinClosure. It sets up the containers *list* and *count*. Then, when we call LinClosure, we just consider pure computation of the closure for a given set. In both versions, redundancy elimination is done the same way.

The main idea to test efficiency of each version of the algorithm is to perform two series of tests corresponding to fixing one of the two parameters of its complexity : $|\Sigma|$ or $|\mathcal{B}|$. Each series consists in fixing one of those parameters and testing a range of value for the other. We can observe figure 3.1 as results of those experiments.



(a) Average time (in s), $|\Sigma| = 100$



(b) Average time (in s), $|\mathcal{B}| = 1000$

$\lvert\mathcal{B}\rvert$	LinClosure	Closure
5000	15.846	0.688
10000	65.930	2.724
15000	115.627	4.784
20000	223.453	9.721

(c) Some landmarks times, $|\Sigma| = 100$

$\lvert\Sigma\rvert$	LinClosure	Closure
10000	4.873	0.058
20000	7.187	0.045
30000	9.432	0.038
40000	10.862	0.034
50000	12.923	0.032
60000	16.721	0.040
70000	17.023	0.036
80000	19.337	0.038
90000	22.013	0.040
100000	24.312	0.042

(d) Some landmarks times

Figure 3.1: Comparison of closure operators for MinCover

The top left figure represents the time spent in MinCover when $|\Sigma|$ is fixed to 100, and $|\mathcal{B}|$ goes from 1000 to 20000 by steps of 1000. The top right one, is $|\mathcal{B}| = 100$ and $|\Sigma|$ running from 1000 to 100000 by step of 2000. In both cases We associated an array (right under) with some particular values, being useful to see evolution of Closure, flattened by LinClosure. The reason for this limitation is time spent in test relatively to the information it brings. As one can see, there is a huge difference of speed between MinCoverLin and MinCoverClo even if they differ only in the first loop. For each test case, we iterate the algorithm over 100 randomly generated basis so as to explore somehow the space of implication theories. Hence, we see for instance that when the number of implications $|\mathcal{B}|$ reaches 20000, with $|\Sigma| = 100$, MinCover with LinClosure needs roughly 225 seconds to run on average for one execution. Assume we would like to go up to $|\mathcal{B}| = 100000$. Then, we would need at least $225 \times 100 \times 80 = 1800000s$ being approximately 3 weeks of test. Of course, we could change the range function to reduce the number of step, but still in our case we want to compare efficiency of Closure against LinClosure. According to the results, we admitted this range was sufficient to see the gap between those two closure operator. Interestingly, LinClosure permits to see practical complexity of MinCover: while the evolution of $|\mathcal{B}|$ draws a somehow quadratic curve of time, $|\Sigma|$ matches a linear growth in spite of noisy points. This matches the theoretical complexity $O(|\mathcal{B}||\mathcal{L}|)$ required by MinCover. One could argue that the algorithms we tested both used Closure in the second loop, altering the overhaul complexity. Indeed, but as we said, LinClosure performs much worse than Closure so that the practical complexity is driven by LinClosure. For MinCover we can conclude that Closure is a better choice.

3.2.2 DuquenneMinimization

3.2.3 MaierMinimization

3.2.4 BercziMinimization

The algorithm issued by Berczi and al in [15] (algoritm 14) can be fine-tuned at first sight without considering closure operators. Indeed, recall that we compute only closure of premises under $\mathcal{L}$ being our input basis, and $\mathcal{L}_c$ our output one. Furthermore, the algorithm suggests to compute the closure of some premises under $\mathcal{L}$ several times, which is extensive and redundant. Furthermore, because we build $\mathcal{L}_c$ only by adding implications, all the closure previously computed can only grow. Therefore we can improve BerczиMinimization by the next means:

- compute all $\mathcal{L}(A), A \in \mathcal{B}(\mathcal{L})$, and store them in a list: $C_{\mathcal{L}}$,
- keep a list of growing $\mathcal{L}_c(A), A \in \mathcal{B}(\mathcal{L})$: $C_{\mathcal{L}_c}$.

To illustrate, we can observe the pseudo-code BerczиImp. This algorithm does not strictly reflect its implementation of course, but it presents the two ideas we were talking about previously. Observe that the closures under $\mathcal{L}$ are the most complicated to compute (because $|\mathcal{B}(\mathcal{L}_c)| \leq |\mathcal{B}(\mathcal{L})|$ even though we repeatedly add implications to $\mathcal{L}_c$) whence the interest of avoiding useless computations especially for $\mathcal{L}$.

When getting the minimum, note that we get both the minimum $\mathcal{L}_c$-closed set not closed in $\mathcal{L}$ and its associated premise (or equivalently, the associated $\mathcal{L}$-closure). Because notations may be a bit heavy, let us illustrate the meaning of $C_{\mathcal{L}}$ and $C_{\mathcal{L}_c}$ in an example.

Algorithm 17: BERCZIIMP

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}_c$: the DQ-basis of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
$C_{\mathcal{L}} := \emptyset$, $C_{\mathcal{L}_c} := \emptyset$;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $C_{\mathcal{L}}[A] = \mathcal{L}(A)$;
  $C_{\mathcal{L}_c}[A] = A$;

while $\exists A \in \mathcal{B}(\mathcal{L})$ *s.t* $C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]$ **do**
  foreach $A \longrightarrow B \in \mathcal{L}$ **do**
    if $C_{\mathcal{L}}[A] \neq C_{\mathcal{L}_c}[A]$ **then**
      $C_{\mathcal{L}_c}[A] = \mathcal{L}_c(C_{\mathcal{L}_c}[A])$;
  $A_P, P := \min\{A, C_{\mathcal{L}_c}[A] : C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]\}$;
  $\mathcal{L}_c := \mathcal{L}_c \cup \{P \longrightarrow C_{\mathcal{L}}[A_P]\}$

return $\mathcal{L}_c$;

Example As usual, let us retake our small example:

- $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$,
- $\mathcal{L} = \{\, ab \longrightarrow cde,\ cd \longrightarrow f,\ c \longrightarrow a,\ d \longrightarrow b,\ abcd \longrightarrow ef\ \}$

Table 3.5 is a trace of the algorithm, using $C_{\mathcal{L}}$ and $C_{\mathcal{L}_c}$. The first column contains premises of $\mathcal{L}$, the second one elements of $C_{\mathcal{L}}$ (that is closures of premises of $\mathcal{L}$ under $\mathcal{L}$) and the third one, $C_{\mathcal{L}_c}$ (closures of $\mathcal{B}(\mathcal{L})$ under $\mathcal{L}_c$). The last column says whether $C_{\mathcal{L}}[A] = C_{\mathcal{L}_c}[A]$.

At each step, the highlighted row is the set of premises and closure satisfying $\min\{C_{\mathcal{L}_c}[A] : C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]\}$, or this row contains the minimal (inclusion-wise) $\mathcal{L}_c$-closed set not closed in $\mathcal{L}$. As wished, the only some closures of $\mathcal{L}_c$ are updated at each step, instead of re-computing all of them every time.

More than fine-tuning the principle, we can optimize the use of LINCLOSURE on two aspects:

(i) in the loop where we initialize lists, we can use the same pruning as in the right-closing step in MINCOVER,

(ii) when computing closures of $\mathcal{L}_c$, because $\mathcal{L}_c$ is only increasing in implications, updating list and counters can be done in $O(|\,\Sigma\,|)$, hence better than the initialization step of $O(|\,\mathcal{L}\,|)$.

Consequently, we will again compare for this algorithm two versions, given as BERCZIIMPCLO and BERCZIIMPLIN (21, 22).

$\mathcal{B}(\mathcal{L})$	$C_{\mathcal{L}}$	$C_{\mathcal{L}_c}$	$=$
ab	$abcdef$	ab	$\times$
cd	$abcdef$	cd	$\times$
c	ca	c	$\times$
d	db	d	$\times$
$abcd$	$abcdef$	$abcd$	$\times$

$\mathcal{L}_c$ before
$\emptyset$

$\mathcal{L}_c$ after
$c \longrightarrow ca$

(a) after initialization step and first loop

$\mathcal{B}(\mathcal{L})$	$C_{\mathcal{L}}$	$C_{\mathcal{L}_c}$	$=$
ab	$abcdef$	ab	$\times$
cd	$abcdef$	acd	$\times$
c	ca	ca	$\vee$
d	db	d	$\times$
$abcd$	$abcdef$	$abcd$	$\times$

$\mathcal{L}_c$ before
$c \longrightarrow ca$

$\mathcal{L}_c$ after
$c \longrightarrow ca,\ d \longrightarrow bd$

(b) second loop

$\mathcal{B}(\mathcal{L})$	$C_{\mathcal{L}}$	$C_{\mathcal{L}_c}$	$=$
ab	$abcdef$	ab	$\times$
cd	$abcdef$	$abcd$	$\times$
c	ca	ca	$\vee$
d	db	db	$\vee$
$abcd$	$abcdef$	$abcd$	$\times$

$\mathcal{L}_c$ before
$c \longrightarrow ca,\ d \longrightarrow bd$

$\mathcal{L}_c$ after
$c \longrightarrow ca,\ d \longrightarrow bd,\ ab \longrightarrow abcdef$

(c) third loop

$\mathcal{B}(\mathcal{L})$	$C_{\mathcal{L}}$	$C_{\mathcal{L}_c}$	$=$
ab	$abcdef$	$abcdef$	$\vee$
cd	$abcdef$	$abcdef$	$\vee$
c	ca	ca	$\vee$
d	db	db	$\vee$
$abcd$	$abcdef$	$abcdef$	$\vee$

Resulting $\mathcal{L}_c$
$c \longrightarrow ca,\ d \longrightarrow bd,\ ab \longrightarrow abcdef$

(d) final loop

Table 3.5: Example of execution of BerczIImp using $C_{\mathcal{L}}$ and $C_{\mathcal{L}_c}$

Algorithm: BerczIImpClo

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}_c$: the DQ-basis of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
$C_{\mathcal{L}} := \emptyset,\ C_{\mathcal{L}_c} := \emptyset$;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $C_{\mathcal{L}}[A] = \textsc{Closure}(\mathcal{L}, A)$;
  $C_{\mathcal{L}_c}[A] = A$;

while $\exists A \in \mathcal{B}(\mathcal{L})\ s.t\ C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]$ **do**
  foreach $A \longrightarrow B \in \mathcal{L}$ **do**
    if $C_{\mathcal{L}}[A] \neq C_{\mathcal{L}_c}[A]$ **then**
      $C_{\mathcal{L}_c}[A] = \textsc{Closure}(\mathcal{L}_c, C_{\mathcal{L}_c}[A])$;
  $A_P, P := \min\{A, C_{\mathcal{L}_c}[A] :\ C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]\}$;
  $\mathcal{L}_c := \mathcal{L}_c \cup \{P \longrightarrow C_{\mathcal{L}}[A_P]\}$;

return $\mathcal{L}_c$;

Algorithm: BerczIImpLin

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}_c$: the DQ-basis of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
$C_{\mathcal{L}} := \emptyset,\ C_{\mathcal{L}_c} := \emptyset$;
LinClosureInit($\mathcal{L}$) ;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  $C_{\mathcal{L}}[A] = \textsc{LinClosure}(\mathcal{L}, A)$;
  $C_{\mathcal{L}_c}[A] = A$;

while $\exists A \in \mathcal{B}(\mathcal{L})\ s.t\ C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]$ **do**
  foreach $A \longrightarrow B \in \mathcal{L}$ **do**
    if $C_{\mathcal{L}}[A] \neq C_{\mathcal{L}_c}[A]$ **then**
      $C_{\mathcal{L}_c}[A] = \textsc{LinClosure}(\mathcal{L}_c, C_{\mathcal{L}_c}[A])$;
  $A_P, P := \min\{A, C_{\mathcal{L}_c}[A] :\ C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]\}$;
  $\mathcal{L}_c := \mathcal{L}_c \cup \{P \longrightarrow C_{\mathcal{L}}[A_P]\}$;
  LinClosureAddImp($\mathcal{L}_c, P \longrightarrow C_{\mathcal{L}}[A_P]$);

return $\mathcal{L}_c$;

In fact, BerczIImpClo does differ from BerczIImp only by the closure notations. However, when

dealing with LinClosure in BerzziImpLin, we use first LinClosureInit to instanciate *list* and *count* for LinClosure in $\mathcal{L}$. Hence a call to this procedure performs only the effective closure computations. Then, LinClosureAddImp just updates *list* and *count* for $\mathcal{L}_c$ by adding a new implication in them. Note that before the first call to this subroutine, *list* and *count* for I_c are empty, hence LinClosure($\mathcal{L}_c$, X) returns X.

3.2.5 AFPMinimization

During all this section we have been testing our minimization procedures with two closure algorithms: Closure and LinClosure. From all of our tests it seemed like LinClosure performs much worse than the other operator. This is in accordance with conclusions and hypothesis drawn in [9] about efficiency of LinClosure. However, those tests allowed us to have a glimpse of the correspondence between theoretical and practical complexity. Still, we may be able to find various more combinations of closure operations to test to see which one is the best. For us, in the meantime of the master thesis and regarding the performance of LinClosure it seemed that our tests were sufficient. Now that we found out which version of each algorithm was the fastest, we can move forward to further tests and joint comparison.

3.3 Joint comparison

In this section we consider the algorithms pruned previously and we compare them with the real datasets we exposed in the first section of this chapter (see table 3.4). For the recall they are datasets taken from the UCI repository. For all of those 28 basis, we ran the 5 minimization procedure we reviewed and implemented. The value recorded is the execution time in seconds for minimizing the given basis. See the results in 3.6.

First, one can observe the wide range of performances. On the one hand, MinCover and Duquenne-Minimization do not exceed 100 seconds of execution time, while BerzziMinimization and AFPMinimization can require up to several hours. The most striking case is for Breast Wisconsin case, with proper and generators basis. This is probably because of the number of implications in those cases. Anyway, while the two first algorithms require about 30 to 90 seconds for minimization, while BerzziMinimization needs roughly one hour and AFP up to ten or eleven hours. This stresses on the intractability of AFP in practice, and possibly the Angluin algorithm when we are prevented from the speed of an oracle.

Regarding the difference between MinCover and DuquenneMinimization, observe that in general, the algorithm by Duquenne is faster on non-minimal basis and slightly slower on already minimal ones. In both cases, one hypothesis to explain those gaps could be the second loop of DuquenneMinimization allowing for a limitation of closure computations. As we already discussed, the number of closure computations in MinCover is likely to be constant whatever the case is, even though those operations become lighter with the reduction of the input basis. In DuquenneMinimization however, apart from first left-saturation and redundancy elimination allowing for some savings in closure computations, we stop an iteration of the second loop whenever some quasi-closed premise is not pseudo-closed. This break point permits to omit closure computations and in practice, thanks to lectic ordering, we may not need to go over all implications

of the output basis to check whether an implication is redundant or not. The only case when the second loop should require more computations is when the base is already minimal, because for each step of the second loop, we have to go over all implications of the growing output basis. As remarked in most of execution times, when the basis is already minimal DuquenneMinimization performs somehow slighter worse than MinCover, which could be explained by the previous hypothesis.

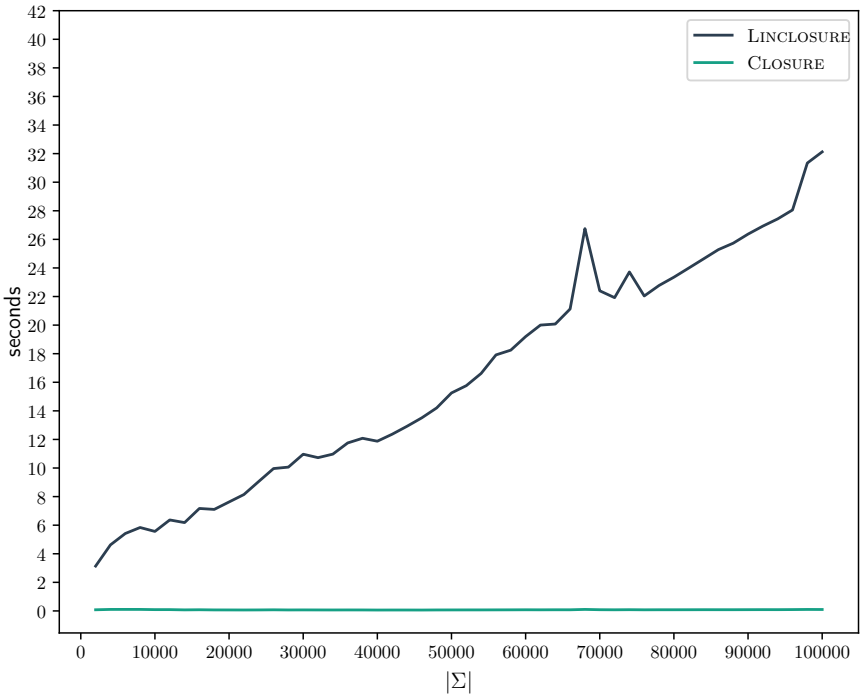
$\mathcal{L}$		MinCover	Duquenne	Maier	Berczi	AFP
Zoo	DQ	< 0.001	< 0.001	< 0.001	< 0.001	0.016
	min	< 0.001	< 0.001	0.015	< 0.001	0.016
	proper	0.007	< 0.001	< 0.001	0.016	0.063
	mingen	0.009	0.016	< 0.001	0.047	0.094
Flare	DQ	0.110	0.203	0.328	27.922	115.656
	min	0.162	0.219	0.406	27.750	116.594
	proper	2.166	0.953	1.250	88.375	524.031
	mingen	19.382	9.906	11.344	160.328	2810.620
Breast Cancer	DQ	0.120	0.140	0.234	33.047	90.031
	min	0.147	0.156	0.297	26.578	89.516
	proper	2.712	1.031	1.219	93.266	429.844
	mingen	2.850	1.578	1.703	102.562	598.172
Breast Wisconsin	DQ	1.333	1.703	2.8125	1005.750	3109.920
	min	1.801	2.016	3.531	949.953	3140.940
	proper	88.188	33.688	46.312	3675.910	40521.0
	mingen	51.378	32.859	33.843	2772.980	38310.200
Operative	DQ	0.004	< 0.001	0.015	0.219	0.734
	min	0.005	< 0.001	0.031	0.219	0.719
	proper	0.048	0.016	0.016	0.594	2.422
	mingen	0.085	0.063	0.109	0.813	4.063
SPECT	DQ	0.040	0.078	0.141	10.328	23.609
	min	0.055	0.094	0.203	8.156	22.906
	proper	1.270	0.484	0.531	51.063	118.531
	mingen	32.580	13.375	13.609	194.875	930.578
Vote	DQ	0.007	0.016	0.032	0.484	1.579
	min	0.009	0.016	0.047	0.469	1.516
	proper	0.090	0.047	0.078	1.625	6.203
	mingen	0.682	0.313	0.485	4.109	22.875

Table 3.6: Comparison of the algorithms on real datasets (execution in s)

One can also denote that MaierMinimization seems to be placed in between DuquenneMinimization and MinCover except in the minimal cases. Our hypothesis is the same as for the difference between DuquenneMinimization and MinCover: in fact, the first loop of Maier's algorithm is to get rid of

redundant implications, therefore when a basis is highly redundant, first removing them allow to spare numerous closure computations and reduces closure cost. This is not the case in MinCover where we compute right closure of all implications. Still, the rest of the algorithm seems to be slower than the second loop of DuquenneMinimization. This may be because to determine equivalence classes and removing direct determinations we need to use closure operators twice for each implications, while with the algorithm by Duquenne, we only compute closure when it is needed after an increasing number of set operations though. The interesting point which seems to go in the direction of our explanation is the behaviour under already minimal basis: Maier's algorithm requires an asymptotically higher number of closure computations than both DuquenneMinimization and MinCover making it the heaviest procedure when it comes at minimizing a basis already minimal.

We would like to remind that this explanation holds within the scope of our tests. We cannot assume it in general and it should be tested in further work on other datasets. Let us say this is a valid hypothesis. In this case, one should be first worried about removing as much redundant implications as possible to lighten the burden of subsequent closure computations, whether it is for getting the canonical basis or not.

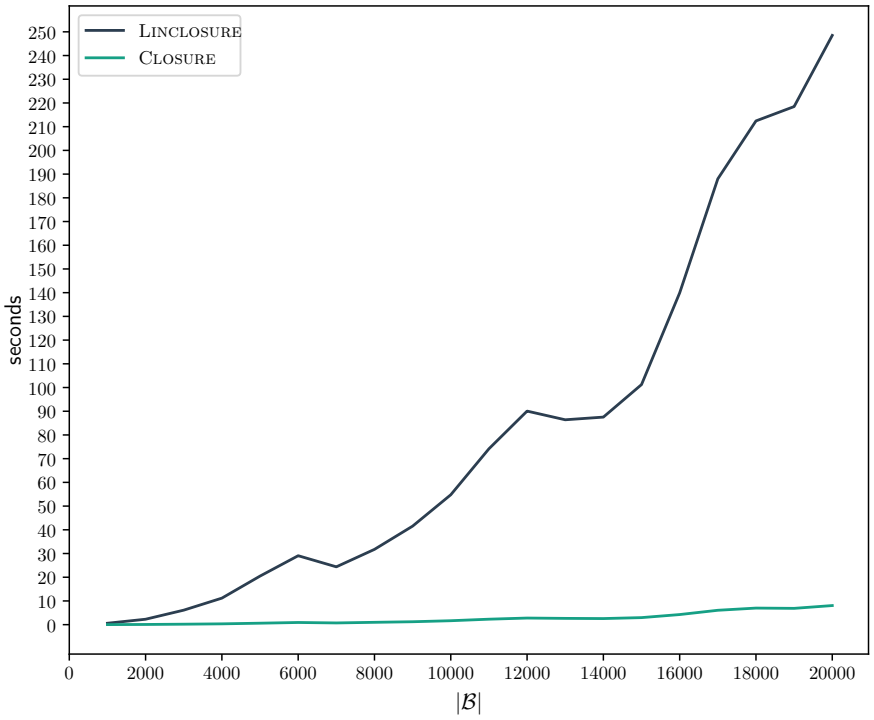


(a) Graphical representation of average time (in s)

$\lvert\Sigma\rvert$	LinClosure	Closure
10000	4.101	0.045
20000	5.632	0.030
30000	11.293	0.043
40000	11.280	0.035
50000	13.870	0.035
60000	16.271	0.040
70000	17.023	0.036
80000	19.337	0.038
90000	22.013	0.040
100000	24.312	0.042

(b) Some landmarks times

Figure 3.2: Average execution time of MaierMinimization when $|\mathcal{B}| = 1000$

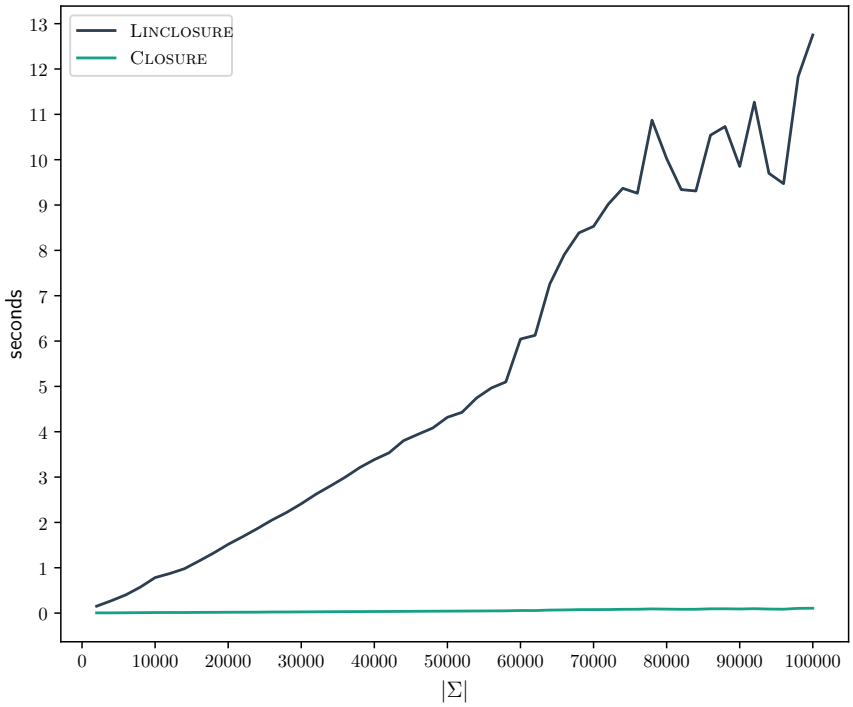


(a) Graphical representation of average time (in s)

$\|\mathcal{B}\|$	LinClosure	Closure
5000	15.846	0.688
10000	65.930	2.724
15000	115.627	4.784
20000	223.453	9.721

(b) Some landmarks times

Figure 3.3: Average execution time of BercziMinimization when $|\Sigma| = 100$



(a) Graphical representation of average time (in s)

$\|\Sigma\|$	LinClosure	Closure
10000	4.101	0.045
20000	5.632	0.030
30000	11.293	0.043
40000	11.280	0.035
50000	13.870	0.035
60000	16.271	0.040
70000	17.023	0.036
80000	19.337	0.038
90000	22.013	0.040
100000	24.312	0.042

(b) Some landmarks times

Figure 3.4: Average execution time of Berczi when $|\mathcal{B}| = 100$

Conclusion

Bibliography

[1] Aho, A. V., Garey, M. R., and Ullman, J. D. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing* (July 2006).

[2] Angluin, D., Frazier, M., and Pitt, L. Learning conjunctions of Horn clauses. *Machine Learning 9*, 2 (July 1992), 147–164.

[3] Arias, M., and Balcázar, J. L. Canonical Horn Representations and Query Learning. In *Algorithmic Learning Theory* (Berlin, Heidelberg, 2009), R. Gavaldà, G. Lugosi, T. Zeugmann, and S. Zilles, Eds., Springer Berlin Heidelberg, pp. 156–170.

[4] Ausiello, G., D'Atri, A., and Sacca', D. Graph algorithms for the synthesis and manipulation of data base schemes. In *Graphtheoretic Concepts in Computer Science* (June 1980), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 212–233.

[5] Ausiello, G., D'Atri, A., and Saccà, D. Graph Algorithms for Functional Dependency Manipulation. *J. ACM 30*, 4 (Oct. 1983), 752–766.

[6] Ausiello, G., D'Atri, A., and Saccá, D. Minimal Representation of Directed Hypergraphs. *SIAM J. Comput. 15*, 2 (May 1986), 418–431.

[7] Ausiello, G., and Laura, L. Directed hypergraphs: Introduction and fundamental algorithms—A survey. *Theoretical Computer Science 658*, Part B (2017), 293 – 306.

[8] B. Ganter, S. O. *Conceptual Exploration.* Springer, 2016.

[9] Bazhanov, K., and Obiedkov, S. Optimizations in computing the Duquenne–Guigues basis of implications. *Annals of Mathematics and Artificial Intelligence 70*, 1-2 (Feb. 2014), 5–24.

[10] Beeri, C., and Bernstein, P. A. Computational Problems Related to the Design of Normal Form Relational Schemas. *ACM Trans. Database Syst. 4*, 1 (Mar. 1979), 30–59.

[11] Bertet, K., Demko, C., Viaud, J.-F., and Guérin, C. Lattices, closures systems and implication bases: A survey of structural aspects and algorithms. *Theoretical Computer Science* (Nov. 2016).

[12] Boros, E., Čepek, O., and Kogan, A. Horn minimization by iterative decomposition. *Annals of Mathematics and Artificial Intelligence 23*, 3-4 (Nov. 1998), 321–343.

[13] Boros, E., Čepek, O., Kogan, A., and Kučera, P. Exclusive and essential sets of implicates of Boolean functions. *Discrete Applied Mathematics 158*, 2 (2010), 81 – 96.

[14] Boros, E., Čepek, O., and Makino, K. Strong Duality in Horn Minimization. In *Fundamentals of Computation Theory* (Berlin, Heidelberg, 2017), R. Klasing and M. Zeitoun, Eds., Springer Berlin Heidelberg, pp. 123–135.

[15] Bérczi, K., and Bérczi-Kovács, E. R. Directed hypergraphs and Horn minimization. *Information Processing Letters 128* (2017), 32 – 37.

[16] Cori, R., and Lascar, D. *Mathematical Logic: Part 1: Propositional Calculus, Boolean Algebras, Predicate Calculus, Completeness Theorems*. OUP Oxford, Sept. 2000. Google-Books-ID: Cle6_dOLt2IC.

[17] Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[18] David, M. Minimum Covers in Relational Database Model. *J. ACM 27*, 4 (1980), 664 – 674.

[19] Day, A. The Lattice Theory of Functional Dependencies and Normal Decompositions. *International Journal of Algebra and Computation* (1992).

[20] Duquenne, V. Some variations on Alan Day's algorithm for calculating canonical basis of implications. In *Concept Lattices and their Applications (CLA)* (Montpellier, France, 2007), pp. 17–25.

[21] Ganter, B. Two Basic Algorithms in Concept Analysis. In *Formal Concept Analysis* (Mar. 2010), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 312–340.

[22] Ganter, B., and Wille, R. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin Heidelberg, 1999.

[23] Guigues J.L, D. V. Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Mathématiques et Sciences Humaines 95* (1986), 5–18.

[24] Hammer, P. L., and Kogan, A. Optimal compression of propositional Horn knowledge bases: complexity and approximation. *Artificial Intelligence 64*, 1 (Nov. 1993), 131–145.

[25] Maier, D. *Theory of Relational Databases*. Computer Science Pr, 1983.

[26] Shock, R. C. Computing the minimum cover of functional dependencies. *Information Processing Letters 22*, 3 (Mar. 1986), 157–159.

[27] Wild, M. Implicational bases for finite closure systems. *Informatik-Bericht 89/3, Institut fuer Informatik* (Jan. 1989).

[28] Wild, M. A Theory of Finite Closure Spaces Based on Implications. *Advances in Mathematics 108*, 1 (Sept. 1994), 118–139.

[29] Wild, M. Computations with finite closure systems and implications. In *Computing and Combinatorics* (Aug. 1995), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 111–120.

[30] Wild, M. The joy of implications, aka pure Horn formulas: Mainly a survey. *Theoretical Computer Science 658* (2017), 264 – 292.