ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук

Вильмэн Симон Лео

ФИО

Минимизация формул Хорна

Название ВКР

Группа М ФКН мНоД16_ИССА

Выпускная квалификационная работа - МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению 01.04.02 «Прикладная математика и информатика»
Программа: Науки о данных

Научный руководитель
доцент, профессор

Объедков Сергей Александрович,
доцент

Москва 2018

FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION FOR HIGHER PROFESSIONAL EDUCATION NATIONAL RESEARCH UNIVERSITY
«HIGHER SCHOOL OF ECONOMICS»
Faculty of Computer Science

Vilmin , Simon Léo

Name, Surname

Horn Minimization

Title

Group M ФКН мНоД16_ИССА

Qualification paper – Master of Science Dissertation
Field of study 01.04.02 «Applied Mathematics and Computer Science»
Program: Data Science

Supervisor
Obiedkov, Sergei Alexandrovich

Moscow 2018

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

Author : Simon Vilmin

Academic Supervisor : Sergei A. Obiedkov

Restitution: May, 21, 2018
Held: June, 6, 2018

Acknowledgement

Before any discussion, I would like to thank people having helped me within this master thesis. First of all, I am deeply grateful to my academic supervisor Mr. Sergei A. Obiedkov for his availability and precious advices, very interesting discussions and above all his patience with me relentlessly knocking at his door.

I am also thankful toward people having given me feedback on this report.

Eventually, since this master thesis is for me a representative of a year studying in Russia, I am thankful to Ekaterina Pavlova and Sergei Kusnetsov for their support, and my friends here, Betty and Irina for memories they contributed to build.

List of Figures

List of Tables

List of Algorithms

Abstract

This document is a report of a master thesis made at HSE computer science faculty (Moscow). It has been conducted in the context of a double-diploma program with the ISIMA (France). The topic was implication theories, or Horn, minimization, used in database applications for instance. More precisely the aim was to provide a review of existing algorithms for performing minimization and implement them to see how do they behave under practical test cases.

By the end of the period, we found several algorithms and reviewed them within the context of closure systems. On top of providing explanations on their operation and complexity analysis, we implemented those algorithms using C++. Using randomly generated systems and real data, we determined which closure operator matches the best each algorithm and which algorithm or steps are likely to be used in practice. Those results being valid within the context of our experiments, suggest further research and experiments to lead in future work.

Keywords: theoretical computer science, implications, closure systems, minimization, canonical basis, algorithmic.

Contents

Introduction

This document reports a master thesis conducted at the faculty of Computer Science of the Higher School of Economics in Moscow, department of Data Analysis and Artificial Intelligence. It took place within the context of double-diploma ISIMA (France)/HSE in the Data Science master programme, second year.

The topic during this master thesis was to review and implement several algorithms for implication theories minimization, also known as Horn minimization. Such systems model knowledge deduction through correlation relations known as implications. The aim of algorithms we studied is to minimize the size of those implicational basis while keeping all the information they provide. Such relations arise in various applications such as formal concept analysis, relational databases, conceptual exploration linguistics or learning spaces for instance. The present document is relative to a theoretical study, hence applications will not be extensively discussed.

To expose our work, we will stick to an outline divided in three chapters. First, we will review the overhaul definition of our problem and mathematical elements we shall need to understand subsequent algorithms. The second part will be dedicated to the review of algorithms we effectively implemented. In the last chapter, we will focus on implementation of procedures (in C++) and their experimental results in terms of execution time.

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 many 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 here, 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 *implications* and use *flower* $\longrightarrow$ *colourful* to denote *"if we have the attribute flower, then we have colourful"* or equivalently *"flower*

implies 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 [32]. Also, note that minimality in general terms is not unique. Indeed, one can define several types of minimality among implication systems. For instance, not only we can define minimality with respect to the number of implications 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 [17]. 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 [15, 8, 22, 7, 32, 13, 25]. 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 [23, 21] for an introduction) and Database Theory (DB) (see [27]). Both sides were developed independently in the early 80's. For the first domain, characterization of minimality goes to Duquenne and Guigues [24], in which they describe the so-called *canonical basis* (also called *Duquenne-Guigues basis* after its authors, abbreviated DG all along this report) relying on the notion of *pseudo-closed sets*. For the database part, study of implications is made by Maier through functional dependencies ([27, 26]). The polynomial time algorithm he gives for minimization heavily relies on a fast subroutine discovered by Beeri and Bernstein in [11], 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 ([28]). Unfortunately, as we shall see and as already discussed by Wild in [31] the algorithm may not be correct in general, even though the underlying theorem is. During the same period, Ausiello et 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 [8, 6, 7]. 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. 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 [18] as an introduction and [12] for a survey. Next, Wild ([29, 30, 31]) 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], [28]). This framework is the one we will use for our study, and can be found in more recent work by Ganter & Obiedkov in [22]. Later, Duquenne proposed some variations of Day's work with another algorithm in [20]. More recently, Boròs et al. by working in a boolean logic framework, exhibited a theorem on the size of canonical basis [14, 15]. 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 [16]. Furthermore, an algorithm for computing the canonical basis starting from any system is given in [22]. This last algorithm was our starting point for this study.

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 et 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 explaining 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 [24, 21, 30, 23] but the reader can also find more than sufficient explanations in [22, 32]. 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 (Formal Concept Analysis, [23]) 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: $a, b, c, \ldots$ and subsets of Σ (groups of attributes) will be denoted by capital letters: $A, B, C, \ldots$ We assume the reader to have some background in elementary set-theoretic and logical 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 [17]). 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 $A \longrightarrow B$. Given two bases $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_1 \models \mathcal{L}_2$ if $\mathcal{L}_1 \models A \longrightarrow B$ for all implications $A \longrightarrow B$ of $\mathcal{L}_2$. Furthermore, $\mathcal{L}_1$ and $\mathcal{L}_2$ are equivalent if $\mathcal{L}_1 \models \mathcal{L}_2$ and $\mathcal{L}_2 \models \mathcal{L}_1$.*

In fact, we can also say that $A \longrightarrow B$ *holds* in $\mathcal{L}$. Because future 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 do not care anymore about semantic interpretations. An implication basis $\mathcal{L}$ is for instance:

$$(\textit{colourful, bloom} \longrightarrow \textit{seasonal}),\ (\textit{colourful, wither} \longrightarrow \textit{seasonal}),\ (\textit{bloom} \longrightarrow \textit{wither})$$

and $M =$*(colourful, bloom, seasonal)* is a model of *colourful, bloom* $\longrightarrow$ *seasonal* because both head and 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 event, 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 [29, 30] 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 *Seraphin 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 (see 1.1). Informally, a graph (see chapter 2) is a pair of sets: nodes and edges. While a node is just an object somehow, an edge relates two nodes.

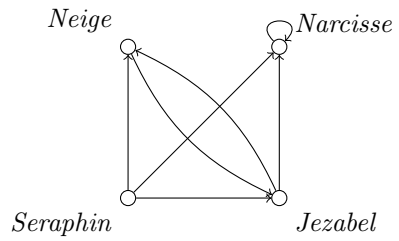


Figure 1.1: Graph of *"like"* relation

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 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 adding new people is not possible any more. For example:

- 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* is related to himself only, 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 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. Formally, this can be written as

$$\mathcal{L}(X) = \bigcap \{A \in \Sigma^{\mathcal{L}},\ X \subseteq A\}$$

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 [29, 22] 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 [15]) or graph theory ([16]) might see this operation as the marking or forward chaining procedure.

Example Let us stick to our vegetable example, but bounded to Σ as {*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 on the link between an implication and the closure of its premise (we redirect the reader to [22] for another proof). When not introduced, we consider a system $\mathcal{L}$ of implications, over some attribute set Σ.

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

Proof. $(\mathcal{L} \models A \longrightarrow B) \longrightarrow B \subseteq \mathcal{L}(A)$. 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 $B \subseteq \mathcal{L}(A)$.

$(B \subseteq \mathcal{L}(A)) \longrightarrow (\mathcal{L} \models A \longrightarrow B)$. 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 $A \subseteq X$, $\mathcal{L}(A) \subseteq X$ and $B \nsubseteq \mathcal{L}(A)$.

□

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 [23, 27]), implications (or functional dependencies) 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 minimum if there is no equivalent system $\mathcal{L}'$ with fewer implications.*

Example We consider our canonical plant example. Take

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

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

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 minimum 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 minimum 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 minimum 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*: $\forall x, y, z \in E, x\mathfrak{R}y \wedge y\mathfrak{R}z \longrightarrow x\mathfrak{R}z$

As a reminder for some statements a and b, $\neg a$ denotes logical negation ($\neg a$ is true when a is not), $a \wedge b$ is the conjunction (the statement $a \wedge b$ is true if both a and b are) and $a \vee b$ is the disjunction ($a \vee b$ is true if at least one of a, b is). $a \longrightarrow b$ is a logical implication, if a is true, so is b. All possible properties are not given here, see [17] for more. With those properties anyway, we can define several types of relations:

Definition 8 (*Equivalence, order*)**.** *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 an (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 on. 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.

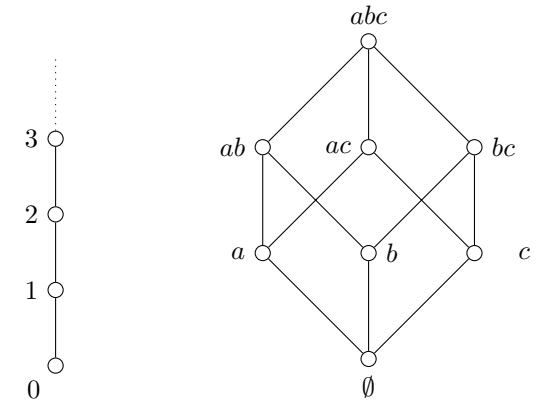
For partial and strict ordering, we will go back to more mathematical examples, in order to slowly go back to our main purpose. Consider the set $\mathbb{N}$ ($= \mathbb{N}_0$) of positive integers, including 0. The natural relation $\leq$ is an order, and the pair $(\mathbb{N}, \leq)$ is an ordered 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 too heavy. It is sometimes called *Hasse diagram* (see [18]) 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.



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

Figure 1.2: Hasse diagrams of two ordered sets

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 operator. 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:

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

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.

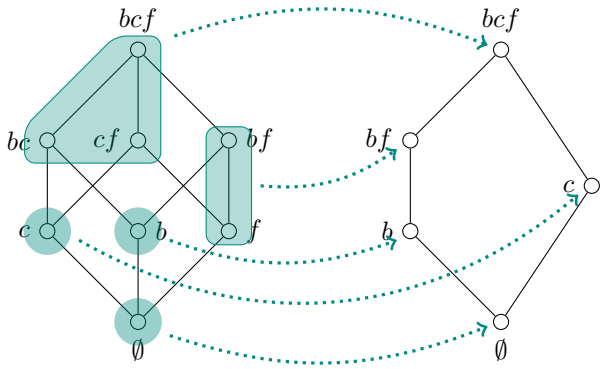


Figure 1.3: Closure operator and equivalence classes

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 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 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 [24], but the name comes from Ganter in [21]. We can also find explanations in following research [21, 19] and in [22]. It turns out that we can explain pseudo-closure by using so called *quasi-closed sets* (see [29, 21, 24]).

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

$$\forall A \subseteq Q, \mathcal{L}(A) \subseteq Q \text{ or } \mathcal{L}(A) = \mathcal{L}(Q)$$

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 [9]). Fortunately, quasi-closed sets and equivalence classes give another definition to pseudo-closedness: a set is pseudo-closed if it is non-closed, quasi-closed, and minimal (inclusion-wise) among quasi-closed sets in its equivalence class. Noticing that a closed set is quasi-closed, a minimal quasi-closed set of a class being closed is not pseudo-closed. Let us illustrate those notions with some diagrams.

Example Let us consider the following case:

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

We could replace $a \longrightarrow b$ and $b \longrightarrow a$ by $a \longleftrightarrow b$ since it means that a and b are equivalent attributes. As in figure 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 $\mathcal{L}(ac) = abc$.

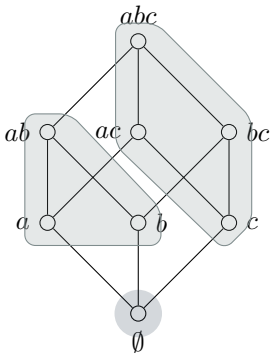


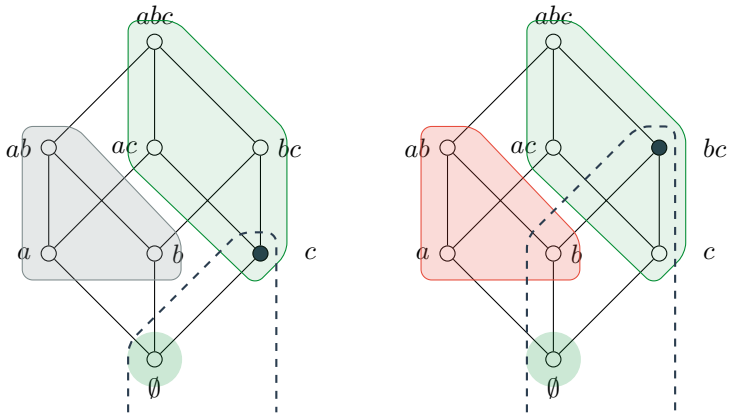
Figure 1.4: Equivalence classes representation of $\mathcal{L} = \{\ a \longleftrightarrow b,\ c \longrightarrow ab\ \}$

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.



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

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

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 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).

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 non-closed 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.

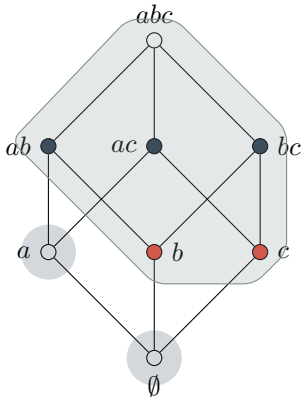


Figure 1.6: Pseudo-closed sets of $\mathcal{L} = \{ b \longrightarrow ac,\ c \longrightarrow ab \ \}$

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 sets, 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*

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

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

This definition does not say that the canonical basis is the only one being minimum. For a links between other possible minimum systems and Duquenne-Guigues basis, see [22] (chapter III, proposition 17).

So far we discussed several notions: implications, pseudo-closed set, quasi-closed set, canonical basis and so forth. Most of them rely 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 [11] called LinClosure addresses this question. LinClosure as previously mentioned has been widely used, notably in [27, 26, 22, 28, 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.).

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);
```

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. $\bot$ stands for *false* and $\top$ for *true*.

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 algorithm for computing closure given by Wild in [31]. It is derived from LinClosure, but we did not consider it because it brings no improvement in complexity. For more complete theoretical and practical comparisons of closure algorithms, we redirect the reader to [10]. In this paper, LinClosure is shown maybe not to be the most efficient algorithm in practice when used in other algorithms, especially when compared with Closure. Anyway, because of its theoretical complexity and use in all algorithms we will review, we will still consider LinClosure.

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 minimum. Finally, algorithms for efficiently computing closures have been presented.

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 ;
```

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 implication systems 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.

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 [10, 19, 29] and DuquenneMinimization being a variation of MinCover (more precisely, of the algorithm provided by Day in a lattice-theoretic framework). This second procedure can be found in [20]. Even though ShockMinimization is not 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 [27, 26]) and FD-Graphs, an extension of minimization to graphs provided by Ausiello and al; in [6, 7, 8]. Eventually we get into algorithms coming out of boolean logic and query learning communities with BercziMinimization and AFPMinimization (see [16] 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. Furthermore, we try as much as possible to provide hands made elements of proof so that the study would be self-sufficient. Still, we may find in all the papers we cite more technical definitions and proofs extending the knowledge we summarize here.

2.2 Algorithms on closure systems

In this section we will be interested in algorithms arising from Formal Concept Analysis. More precisely we will rely on the work provided by Ganter, Day, Wild, Duquenne-Guigues in [23, 21, 19, 29, 31, 24, 20]. In this framework we will heavily rely on pseudo-closed and quasi-closed sets.

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 [22]. As we shall explain, it has roots in Day lattice-based algorithm [19], and more surprisingly, *"unknown"* ancestor in Shock algorithm [28].

The principle is to perform *right-saturation*, and then *body redundancy* elimination. In fact, not only this is the general idea recently issued in [15], but it is also the main theorem of Shock in [28] and the part of a theorem by Wild ([29, 30]). 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 ($\mathcal{L}^-(A) = \mathcal{L}(A)$), then A is not required to get $\mathcal{L}(A)$: $A \longrightarrow \mathcal{L}(A)$ can be removed. The second loop can also be named *left-saturation* since we are maximizing a premise up to its implication. As we will see, because implications are right-closed, this is left-saturation is pseudo-closing or closing a premise. Now that the principle is explained in words, let us introduce the pseudo-code (see algorithm 3).

MinCover ends up on the canonical basis. Assuming that closures are computed with LinClosure, the overall 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 DG basis, being minimum 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:

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**
    $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;

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. □

In fact, the second loop of MinCover is iterating somehow the following operation: $A \cup \bigcup\{\mathcal{L}(B) \mid B \subset A\}$ up to $\mathcal{L}^-(A) :=$, related to *proper implications* we shall encounter in the third chapter. Still, 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 [28], 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 [29, 30] 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 ⟶ B ∈ L do
    L := L − {A ⟶ B} ;
    if B ⊈ L(A) then
        L := L ∪ {A ⟶ L(B)} ;
```

This routine, co-issued with the theorem we discussed previously is quite different from MinCover. Even though the conditional statement $B \nsubseteq \mathcal{L}(A)$ 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 minimum in general:

- if $\mathcal{L} = \{\emptyset \longrightarrow a,\ a \longrightarrow b\}$ (in this order), ShockMinimization will produce $\emptyset \longrightarrow ab$ which is right,
- if $\mathcal{L} = \{a \longrightarrow b,\ \emptyset \longrightarrow a\}$, 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 [31]. However, the implications given by Shock may have quasi-closed premises and present similarities with the algorithm for generating quasi-closed sets FindCrucialGenerators of Day ([19]). In Wild work, we can also find another proof for the theorem of Shock, jointly with minimality of DG basis (theorem 5 of [30, 31]). 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 framework in some sentences. At the best of our understanding, 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.

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 contains the closure of every its pseudo-closed proper subset,
- 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. 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($\mathcal{L}$) ;
$\mathcal{L}_c := \emptyset$;

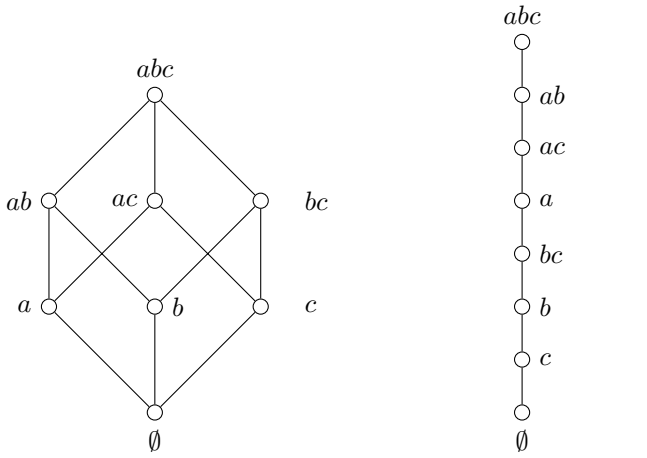
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  foreach $\alpha \longrightarrow \beta \in \mathcal{L}_c$ **do**
    if $\alpha \subset A \wedge \beta \not\subseteq A$ **then**
      $\mathcal{L} = \mathcal{L} - \{A \longrightarrow B\}$;
      goto next $A \longrightarrow B \in \mathcal{L}$;
 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

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.

The advantage of lectic ordering is to allow for easier checking of pseudo-closedness 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 sorting procedure



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

Figure 2.1: Subset and lectic ordering

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.

$\mathcal{B}(\mathcal{L})$	$\mathcal{L}$ 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

Observe that at the 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 $\mathcal{L}$ shuffled 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 overall 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

Here, we deal with algorithms based on Functional Dependencies (FD's). In database theory, functional dependencies are also implications between sets of attributes. The difference lies in the value of those attributes. up to now we have been considering *"binary"* attributes and implications: either we have x or we do not have it. In DB, an attribute can be multi-valued (imagine an attribute *"age"* for our plant example). Therefore, the notion of implication is stricter: if we are given two attributes x and y, $x \longrightarrow y$ will be a valid functional dependency if the *value* (not the presence) of x determines the value of y. For more precision, see [27]. Actually, when it comes at minimization this semantic difference does not matter. Therefore, the algorithms we are about to study suit also our framework. For this reason we will explain algorithms within our usual context.

2.3.1 First algorithm: Maier's algorithm on FDs

Here we will consider one of the first algorithm given for minimization purpose. It has been proposed by Maier in [27, 26] and relies notably on the algorithm LinClosure issued in [11]. 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 information. 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 $\mathcal{L}^- := \mathcal{L} - \{A \longrightarrow B\} \models A \longrightarrow B$ or equivalently if $B \subseteq \mathcal{L}^-(A)$. Thus get rid of 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 [27, 26] 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 $C \equiv_{\mathcal{L}} A$ 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:

$$(\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow C) \longrightarrow \mathcal{L} - A \longrightarrow B \models A \longrightarrow C$$
$$\longrightarrow \mathcal{L}^{-} \models A \longrightarrow C$$

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

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

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 [29, 30] drawn a parallel between quasi-closedness 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 a non-redundant basis contains direct determinations, it is not minimum. 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 not 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 implication $A \longrightarrow B$, we would have to provide LinClosure with a vector of implications with premises equivalent to A. The first one we

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 $C \longrightarrow D \cup B$;

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 $E_{\mathcal{L}}$ define a partition of $\mathcal{L}$. That is, we will have to compute at most $|\mathcal{B}|$ closures to get rid off direct determinations. So, the last step of the algorithm requires $O(|\mathcal{B}||\mathcal{L}|)$, which is consequently the complexity of MaierMinimization by previous explanations. It is important to mention that even though the base we obtain is minimum, 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 explicitly Maier's algorithm on it. As a reminder, we had:

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

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 $B \subseteq \mathcal{L}^{-}(A)$. 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

$\mathcal{B}(\mathcal{L})$	$\mathcal{L}(\cdot)$
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

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 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 minimum $\mathcal{L}$ 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 [27, 26] for finding a minimum 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 et al. in [8, 6, 7]. 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 [7] and then go into further details and more precise algorithms for such computations as exposed in [6].

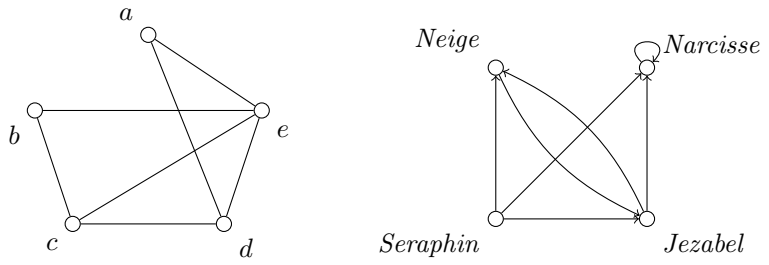
FD-Graphs and minimum covers

As we already mentioned, the graph framework developed by Ausiello et al. in [6, 7] comes from the work of Maier in database theory over functional dependencies (see [27]). 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 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 [6, 7] as an improvement of the structure proposed in [5]. It deserves to represent implication theories within a framework simpler than hypergraphs. It has been recently re-issued in [8] 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 $\mathcal{L}$*.*

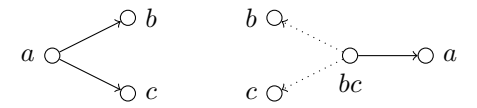
Again, the definition may be quite confusing. $\mathcal{B}(\mathcal{L})$ is still the set of premises of $\mathcal{L}$. Therefore, let us pause our explanations with some toy examples, presented in figure 2.3. From those graphs, we can give *"handy"* way to build an FD-graph out of some theory $\mathcal{L}$:

- 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*.

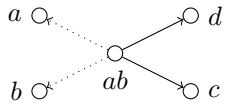
This is indeed what we formally defined previously. Furthermore, for this algorithm, we consider a basis $\mathcal{L}$ over an attribute set Σ, such that:

- 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 $\mathcal{L}$, $A \cap B = \emptyset$

A theory in this form is said *reduced*.



(a) FD-Graph of $a \longrightarrow bc$ (b) FD-Graph of $bc \longrightarrow a$



(c) FD-Graph of $ab \longrightarrow cd$

Figure 2.3: Representation of some FD-graph

The next definition is about describing a graph-theoretic way to combine implications to derive new ones. 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, $\mathcal{L} = \{ab \longrightarrow f,\ af \longrightarrow g,\ a \longrightarrow c,\ b \longrightarrow d,\ cd \longrightarrow e,\ c \longrightarrow h,\ cd \longrightarrow e\}$.

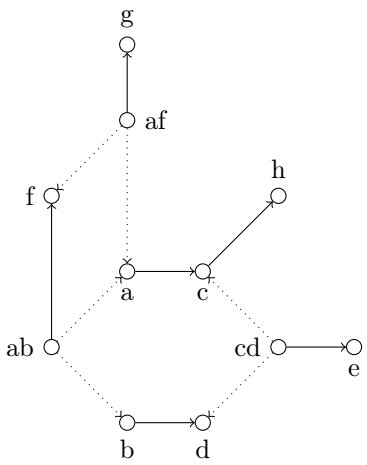


Figure 2.4: FD-Graph of some implicational basis

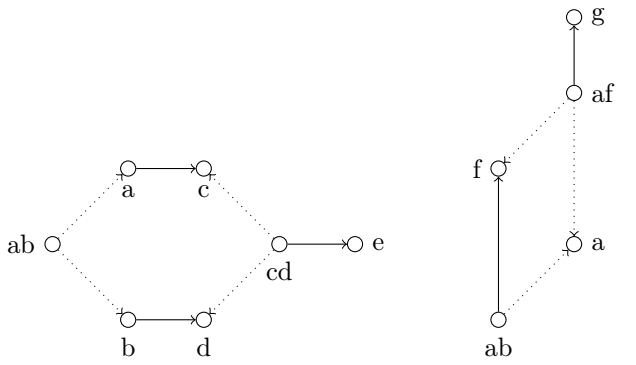
There are either *dotted* or *full* paths. A path $\langle i, j\rangle$ is dotted if all arcs leaving i are dotted, it is full otherwise.

Having explained FD-Graphs, we will now move to explanations of the algorithm developed by Ausiello et al. The procedure finds from a given basis its minimum representation in our terms (see alorithm 9).

Algorithm 9: AusielloMinimization (Overview, 1983)

Input: $\mathcal{L}$ an implication basis
Output: $\mathcal{L}_c$ a minimum 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 ;



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

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

Figure 2.5: Representation of some FD-paths

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 review 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 $O(|\Sigma| \times |\mathcal{B}|) = O(|\mathcal{L}|)$ 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 $|\mathcal{L}|$. It is important to note that we can reach this complexity only if we consider the closure of a node to be a matrix in which

accessing an element is $O(1)$. Hence, if we consider the FD-Graph to be an adjacency list so as to ensure $|G_{\mathcal{L}}| = O(|\mathcal{L}|)$ as mentioned in [6, 7], the closure is represented through an adjacency matrix and adding element to the closure may be understood as *"setting the value to 1 in the vector corresponding to the closure of i"*. The size of the graph representing the closure is then $O((|\mathcal{B}| + |\Sigma|)^2)$ since every node appears in the closure.

Algorithm 11: NodeClosure

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

```
S_i := L_i ;
S_i^+ := ∅ ;
while S_i ≠ ∅ do
    select j from S_i ;
    if j ∈ V^0 then
        forall m ∈ D_j − {i} do
            q_m := q_m + 1 ;
            if q_m = |L_m^1| then
                S_i := S_i ∪ {m} ;
    S_i^+ := S_i^+ ∪ {j} ;
    forall k ∈ L_j^0 ∪ L_j^1 do
        if k ∉ S_i^+ ∪ L_i^{1+} ∪ {i} then
            S_i := S_i ∪ {k} ;
return S_i^+ ;
```

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

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

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 ;

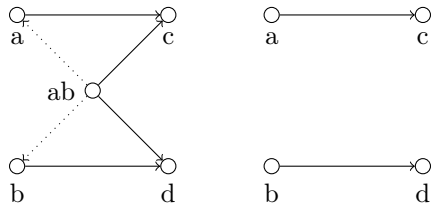
Algorithm 13: GraphClosure

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$(L_i^0 - L_i^{1+})$;

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

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 $\mathcal{L} - \{A \longrightarrow B\}$.

□

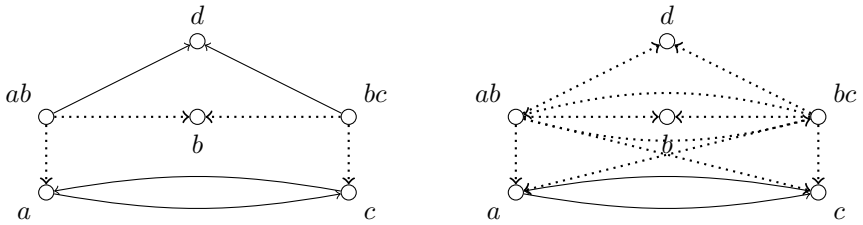
To remove redundant nodes, Ausiello et al. observed that a redundant node will only have dotted arcs in the closure of a FD-Graph. Hence its minimization procedure for some $\mathcal{L}$ and associated $G_{\mathcal{L}}$ suggests to determine the closure of $G_{\mathcal{L}}$ and then to remove all redundant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

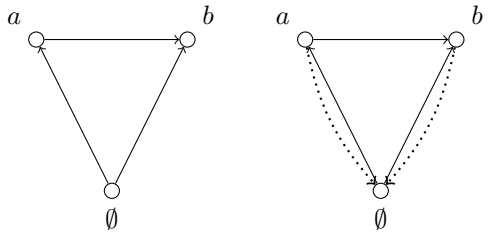
FD-Graphs suffer from another drawback: representation of non-closed empty set. To the best of our knowledge, this has not been discussed. While Maier's algorithm is flexible towards open empty set, FD-graphs may not, unless we missed informations. How to represent the basis $\mathcal{L} = \{\emptyset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

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



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

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

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's 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.

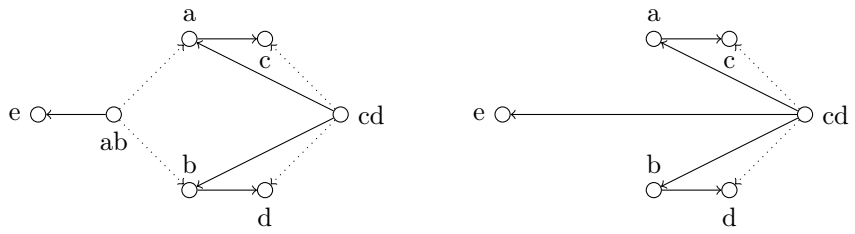
Removing superfluous nodes

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 bases 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

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



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

(b) FD-Graph with superfluous node removed

Figure 2.9: Elimination of superfluous node

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 $\mathcal{L}$

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

(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 Maier-Minimization in implications framework. Hence both are correct and the resulting FD-graph represents a minimum basis in our terms.

In fact, the manipulations we described previously to remove superfluous nodes do not represent the true algorithm given by Ausiello and al for this purpose. To be more precise, the papers [6, 7] suggests to perform the computations of algorithm 14.

For all compound nodes i, we check in its dotted closure L_i^{1+} whether we can find an equivalent node j. This can be done in $O(|\mathcal{B}| + |\Sigma|)$. If we found such j, then i is superfluous and we move to the full closure of j all nodes k such that (i, k) is full and (j, k) dotted. This corresponds to the union operation we depicted earlier. When we remove an implication with a superfluous premise, we append to the consequence of the *"target"* implication, the consequence of the superfluous one. This is also $O(|\mathcal{B}| + |\Sigma|)$. Then, removing i from the closure does also require a run over all nodes of the closure: $O((|\mathcal{B}| + |\Sigma|))$. Therefore, the first part of this algorithm is $O(|\mathcal{B}|(|\mathcal{B}| + |\Sigma|))$. The second part is about removing nodes and moving arcs. If run over L from the end, we may be able to find the final destination of full arcs with only one

Algorithm 14: SuperfluousnessElimination

Input: $G_{\mathcal{L}}$: the FD-Graph of some non-redundant basis $\mathcal{L}$
Output: $G_{\mathcal{L}}$: the associated minimum FD-Graph

forall $i \in V^1$ **do**
  find an equivalent node j ;
  if j *exists* **then**
    $L_j^{0+} := L_j^{0+} \cup (L_i^{0+} \cap L_j^{1+})$;
    $L_j^{1+} := L_j^{1+} - (L_i^{0+} \cap L_j^{1+})$;
  remove i from the closure ;
  add (i, j) to a list L ;
remove superfluous nodes ;
move arcs to their final destination ;

run. Then, we can run over the graph, and for each arc, whether it points out to a superfluous node and we remove it, the node itself is redundant and the arc is full and we move it to its destination, or we do nothing. All these operations are $O(1)$ and the overall operation requires to visit all the edges, resulting in $O(|\mathcal{L}|)$ complexity. Therefore, the cost of removing superfluous node provided we have the closure of the FD-Graph we work on seems to be $O(|\mathcal{B}|(|\mathcal{B}| + |\Sigma|))$, in accordance with [6, 7] being better than the removal of direct determination in Maier's algorithm.

What we can conclude of the procedure issued by Ausiello is that it performs the same work as Maier's algorithm but using advantage of the structure of FD-Graph and its distinction between dotted/full arcs to enhance removing superfluous nodes / direct determination. However, as aforementioned, we did not implement this graph-theoretic approach. The main reason is some difficulties in clearly understanding the ideas of Ausiello, notably for the redundancy step, empty set representation, or the data structures one should use to match theoretical complexity. Furthermore, as we exposed, the operations performed are somehow very similar to the work done by Maier even though FD-Graphs use their dotted-full arcs to slightly improve computations. Nevertheless, the whole algorithm may still require as much theoretical time as Maier since most of the properties of AusielloMinimization use the closure of an FD-Graph needing $O(|\mathcal{B}||\Sigma|)$ to be computed. On top of that, to get from a basis to a graph, we may need (linear) translation time, as much as a large amount of memory (especially for the closure of a graph). 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. Plus, because of the similarities with MaierMinimization, limiting ourselves to the Maier's algorithm permits to have a first insight on the efficiency of core operations. Nevertheless, as all this section shows, we still provided some theoretical comparison to exhibit the heart of AusielloMinimization being a translation from a framework to another. Hence, with those explanations we leave the door open to a future implementation. The interest of truly implementing this graph algorithm will depend on the efficiency of MaierMinimization in practice.

In this section we studied MaierMinimization and its graph-theoretic version AusielloMinimization, two algorithms designed for functional dependencies minimization. It remains now to focus on algorithms coming from logical and query learning communities.

2.4 Propositional logic based approach

The last domain we will explore is boolean logic. Actually, boolean logic is just a background as previous domains and we will focus to the expression of the algorithms within our framework. The main studies we will focus on can be found in [15, 16, 2, 3]. Anyway, as an informal introduction, let us see the parallel between logic and our set-theoretic representation. This introduction is absolutely not mandatory for the understanding of next material, but it can be interesting for the reader with some knowledge in propositional logic.

2.4.1 From sets to boolean logic

We will need the usual logical notations $\neg, \vee, \wedge, \longrightarrow$ (negation, disjunction, conjunction, implication for the recall) we defined in the first chapter. The reader can read [17] to have in-depth explanations. For a moment, let Σ be a set of *boolean variables*: variables being either *true* ($\top$, 1) or *false* ($\bot$, 0). From this point of view, an implication is in fact a so-called *clause*. A clause is a disjunction of literals. For instance:

- $(x_1 \vee x_2 \vee \neg x_3)$ is a clause,
- $((x_1 \vee x_2) \wedge x_3)$ is not (in our context).

And in particular, a clause in which there is *at most* one positive literal is called a *Horn clause*. When there is exactly one positive variable, it is a *definite* such clause. In our context, we only have definite Horn clauses:

- $(x_1 \vee \neg x_2 \vee \neg x_3)$ is a definite Horn clause,
- $(\neg x_1 \vee \neg x_2)$ is a Horn clause not definite.

Because $x_1 \longrightarrow x_2$ is logically equivalent to $x_2 \vee \neg x_1$ and thanks to de Morgan's law on negation (the negation of a disjunction is the conjunction of the negations and vice-versa), we may rewrite a clause $(x_1 \vee \neg x_2 \vee \neg x_3)$ as an implication $(x_2 \wedge x_3) \longrightarrow x_1$. Now, we can associate several clauses in a conjunction:

$$((x_2 \wedge x_3) \longrightarrow x_1) \wedge ((x_2 \wedge x_3 \wedge x_4) \longrightarrow x_5) \wedge (x_2 \wedge x_3) \longrightarrow x_4)$$

Such a conjunction is called a *Horn formula*. When written under its clause form, it is also a formula under *Conjunctive Normal Form* (CNF): a conjunction of disjunctive clauses. To get back up on our feet with sets, we only lack one step. Given a subset A of an attribute set Σ, we can use the *characteristic vector* ψ_A of A. It is a binary vector of size $|\Sigma|$ such that the i-th bit of ψ_A is a boolean variable saying whether the i-th element of Σ is in A or not. This supposes Σ to be totally ordered, which is never an issue. For instance, let $\Sigma = \{a, b, c, d, e\}$ be an attribute set (not boolean variables), and $A = abd$, then $\psi_A = 11010$.

Now, we can associate propositional variables x_a, x_b, x_c, x_d, x_e to a, b, c, d, e respectively. Finally, let us take an implication say $ab \longrightarrow de$. This attribute implication can be transformed into logical one:

$$x_a \wedge x_b \longrightarrow x_d \wedge x_e = (x_a \wedge x_b \longrightarrow x_d) \wedge (x_a \wedge x_b \longrightarrow x_e)$$

We use conjunction because ab contains a *and* b. This very short paragraph concludes our quick parallel with logical background, we will now go back to our usual framework to study algorithms. With this procedure explained, it remains now to focus on query learning and Angluin algorithm.

2.4.2 Iterative building of the canonical basis

Here, we will study an algorithm proposed by Berczi et al. in [16] following a paper by Boros et al. ([15]). 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 to express logical process, 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 system $\mathcal{L}$: 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 DG basis. Now that the process is defined, let us expose the procedure and discuss it (see algorithm 15)

Algorithm 15: BerczIMinimization

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}_c$: the DG 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 [16, 15], pseudo-closure is not explicitly considered. Instead, an implication of the form $P \longrightarrow \mathcal{L}(P)$ where P is pseudo-closed, is called *left-right-saturated*. In fact, left-saturation stands for quasi-closure somehow. 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 15, 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 DG 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. Let us study 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 BerczиMinimization 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 $\mathcal{L}_c$.

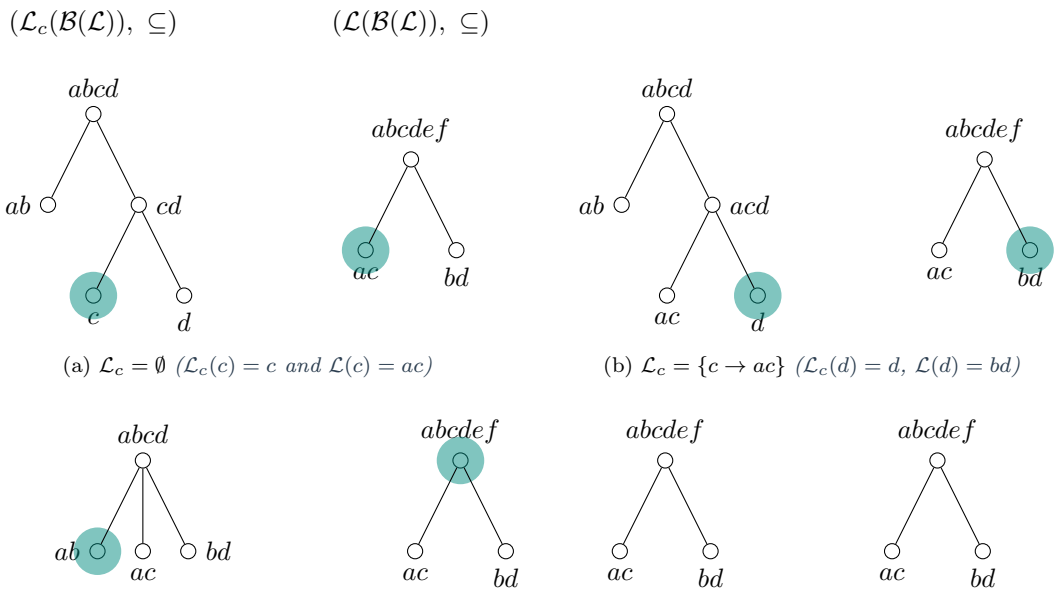
(b) $\mathcal{L}_c = \{c \longrightarrow ac\}$. d is a premise of $\mathcal{L}$ being closed in $\mathcal{L}_c$, hence minimal. Consequently, we add $d \longrightarrow bd$ to $\mathcal{L}_c$.

(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 minimum.



(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

As we said, this algorithm is much less efficient in theory 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 (see chapter 3).

2.4.3 Angluin algorithm and AFP: Query Learning based approach

Here, the method for building a minimum 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 information on the object we are looking for. For 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 (16) 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 minimum representation of the basis we want to learn ([2]) and more than that, to end up on the canonical one by Arias et 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.

The main idea is to ask the oracle whether the theory we are building ($\mathcal{L}_c$) is equivalent to $\mathcal{L}$ until it answers *"yes"*. If it says *"no"* then it provides an example on which $\mathcal{L}_c$ and $\mathcal{L}$ differ. If the example is a model of $\mathcal{L}$, then we track implications in $\mathcal{L}_c$ falsified by this example and correct them. If the example however is not a model of $\mathcal{L}$ we look for the first possible smaller example out of the one we got. The main idea is to say that if we correct a smaller example, we are likely to correct a larger one too. If we do not find any smaller example to correct, we add an implication in $\mathcal{L}_c$ addressing the problem. 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 (17).

Algorithm 16: AngluinAlgorithm

Input: $\mathcal{L}$ a theory to learn and an *Oracle* with *membership, equivalence* queries
Output: $\mathcal{L}_c$ the canonical representation of $\mathcal{L}$

$\mathcal{L}_c = \emptyset$;
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 $\mathcal{L}_c$;

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. The nested for all loop over $\mathcal{L}_c$ is similar to the operations we make in case of negative counter-example in AngluinAlgorithm: whenever we find a smaller counter-example out of X and an implication of $\mathcal{L}_c$ we refine this implication and move to the next possible example. If we do not find any implication in $\mathcal{L}_c$ to refine (i.e the oracle replied "no" for all implications in $\mathcal{L}_c$ in other Angluin's algorithm), then we add $X \longrightarrow \mathcal{L}(X)$ to $\mathcal{L}_c$.

up to now, we are not sure of the running time 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 $\mathcal{L}$. Still, if an implication is refined more than one time, the elements stacked shrink in size. Hence we are likely to have a loop requiring in the worst case $O(|\mathcal{L}|)$ to terminate. Because we are doubtful about this assumption, we do not integrate it in complexity estimation. This leads to an approximation of the overall time complexity. Indeed, even though we do not know the number of times 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

Algorithm 17: 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 $\mathcal{L}_c$;

over implications of $\mathcal{L}_c$ contains computations of elements in $\mathcal{L}$, the overall algorithm may be cubic in the number of premises in $\mathcal{L}$ (that is $O(|\mathcal{B}|^2|\mathcal{L}|)$). Therefore, this would be the worst time complexity.

Even though we do not have a proof for this algorithm, we can leave for further research some ideas helping in approaching the correctness of AFP. Nevertheless, because they are somehow *"abstract"* ideas, they may suffer from lack of precisions, or omit details:

- a set A in $\mathcal{S}$ is always comprised between a premise of $\mathcal{L}$ and its closure $\mathcal{L}(A)$ in terms of inclusion,
- the principle of AFPMinimization has echoes in BerczıMinimization. In fact, one could rewrite the procedure given by Berczi as follows: while we can find a premise in $\mathcal{L}$ generating a $\mathcal{L}_c$-closed $\mathcal{L}$-non-closed set (that is, a negative counter-example), take the minimal such example and add its full implication to $\mathcal{L}_c$. The minimality argument permits to limit the algorithm to appending full implications. Hence here is a possible idea for further work: it is likely that because in AFP we do not use minimality, we may need to update several implications. This processing relies on the main loop iterating over all implications and not being an equivalence query (as opposed to BerczıMinimization and AngluinAlgorithm).
- one should be interested in describing an invariant for the procedure. For instance, we may observe that at each step the remaining set of premises of $\mathcal{L}$ are a sufficient set of counter-examples for $\mathcal{L}_c$ with respect to $\mathcal{L}$. More than this, this may also be the case for $\mathcal{S} \cup \mathcal{B}_i$ where B_i denotes the remaining set of premises of $\mathcal{L}$ at step i. If a premise is a counter example, it is likely to be pushed in $\mathcal{S}$ so that $mathcalS \cup \mathcal{B}_i$ does not change overall. Knowing that an element of the stack cannot be larger than its closure, and that if an implication can be refine more than one time, then in the worst case all elements of the stack may shrink in size because $|\Sigma|$ is finite. Therefore, termination of the repeat-until loop should be guaranteed, allowing for reducing the size of sufficient set of premises by 1 at each step of the main loop.

In spite of the lack of proof or counter-example, we can still provide a trace to see how does AFP work for real.

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

Unlike AusielloMinimization, we implemented AFPMinimization even though we cannot prove its correctness. The choice comes out of the following reasons. Actually, AFPMinimization has a principle very different from all other algorithms, which is not the case of AusielloAlgorithm being somehow an optimization (at best) of MaierMinimization. Furthermore, because AFPMinimization can be expressed within closure systems, it does not require memory overhead and can be implemented quickly, especially in our case where it was one of the last algorithms to be studied. Since we do not have a concrete idea of its complexity, implementing AFPMinimization allows to have a glimpse of its efficiency, as much as an idea of the tractability of AngluinAlgorithm when we get rid of the randomness of an oracle.

We discussed here algorithms coming from logical and query learning. Interestingly those two procedures have another approach to minimization: they try to build a minimum base against an input theory. Unfortunately, it seems like this approach may not be more efficient than minimizing the input base itself, at least from a theoretical point of view. Before concluding this chapter, we will quickly discuss some expectations we can derive from our study.

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 a complexity supposed to be greater than $|\mathcal{B}|^2|\mathcal{L}|$.

Note that we did not truly listed all the algorithms 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 and right-closure. Those steps first issued in works by Day in [19], Duquenne-Guigues in [20, 24], Wild in [30, 31, 29] have been recently discussed by Boros and al in [15]. On the other hand MaierMinimization and DuquenneMinimization share redundancy elimination as a first step, being the last part of MinCover. The most striking difference of MaierMinimization 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. Furthermore, where the two first functions need at most two closure computations per implication, we may need three for Maier's algorithm. This could suggest that MaierMinimization is worse in practice than DuquenneMinimization and MinCover. However, this argument is opposed to redundancy elimination of Duquenne's and Maier's functions. Removing several implications lighten subsequent loops and closure computations. Because MinCover does not perform redundancy elimination beforehand, there is also a chance that it is more expensive than the two others procedures, at least with non-minimum base. 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 theory. Quite the opposite, BerczMinimization and AFPMinimization are building a canonical representation of an input basis. For DuquenneMinimization however, even if we are indeed minimizing $\mathcal{L}$ by removing its redundant implications, we are also building our result. Speaking of theoretical complexity, it seems like this building approach is 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 [10]).

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

Chapter 3

Implementation of algorithms

In previous chapters we were interested in exposing our problem of minimization and reviewing several algorithms dedicated to this task. In the present chapter we will study the implementation of those algorithms and their concrete effective speed. First, we will set up our ground for testing and the data we generated or used. Then we will observe the efficiency of each algorithm using different closure operators. Eventually we compare algorithms on real datasets, and discuss perspectives of our work.

3.1 Test set up

The presentation of our step is three-parted. First we talk about the tools we used such as programming language, file format and so forth. Then we move to data we used, either randomly generated or real.

3.1.1 Tools

Here, we understand by tools the general implementation ground we rely on: languages, 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 taking into account the time spent outside the program. Our CPU is an Intel I5, 2.4 GHz.

We used the code from `https://github.com/yazevnul/fcai`, used in [10] 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
```

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-formatted results. The code we wrote is available at `https://github.com/SimVil/Horn-Minimization-Code`. However, we must mention that as of today it lacks some documentation and is likely to suffer from some signed/unsigned and unused variables related warnings at compilation time (under `-Wall -Wextra` at least).

3.1.2 Randomly generated data

This paragraph is important for all following experiment. 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.

Later, we will tests randomly generated basis on two parameters, the number of implications and the number of attributes. More precisely, we will set one of them, and let the other go over a range. When $|\mathcal{B}|$ is fixed, and $|\Sigma|$ grows, especially when $|\Sigma| \gg |\mathcal{B}|$, we may encounter the problem of generating only non-redundant basis, because we are more likely to generate disjoints sets (the attribute set being very large, for few implications) and hence non-redundant implications only. To bypass this problem, we introduce another randomisation based on Armstrong rules [4, 22]. Given some implications, those rules are an inference system to derive new implications out of others. Some of them are:

- $\mathcal{L} \models A \longrightarrow B$, then $\mathcal{L} \models A \cup C \longrightarrow B$ (*augmentation*)
- $\mathcal{L} \models A \longrightarrow B$ and $\mathcal{L} \models C \longrightarrow D$, then $\mathcal{L} \models A \cup C \longrightarrow B \cup D$ (*union*)
- $\mathcal{L} \models A \longrightarrow B$ and $\mathcal{L} \models B \cup C \longrightarrow D$, then $\mathcal{L} \models A \cup C \longrightarrow D$ (*pseudo-transitivity*)

Therefore, when we generate theories with an increasing attribute sets, given a number of implications $|\mathcal{B}|$, we only generate randomly a part of (by tenth for simplicity) those implications and we use randomly Armstrong rules to generate the remainder. This creates then a certain amount of redundancy even if $|\Sigma| \gg |\mathcal{B}|$. Nevertheless, for the other case around, redundancy is very likely to be built in, since there is much more implications than attributes. In this case we just generate the required number of implications. Ideally, we would like to supply a certain amount of non-redundancy for which we did not do enough research to find a solution.

Note that this method can be discussed and probably improved. For us, it seems like this is sufficient for supplying us with theories having informative implications, and various sizes of premises and conclusions. 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 real application we used is FCA (Formal Concept Analysis) since the framework we use is dedicated to this field. We will first present it briefly 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 $'' : 2^M \longrightarrow 2^M$. Pairs (A_G, A_M) of closed subsets from $G \times M$ with $A'_G = A_M$ and $A'_M = A_G$ are called *concepts*. A_G is the *concept extent* while A_M is called *concept intent*. As one could expect, 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.

	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

Eventually, we shall discuss various possible bases one can build out of a context. We will use 5 of them: the canonical basis, the minimum basis resulting from Maier's algorithm (twice), the basis of minimal generators and the proper basis. We already discussed the two first ones. We derive them from the non minimum:

(i) minimal generators: a set X is a minimal generator of its closure $\mathcal{L}(X)$ if it is minimal in $[X]_{\mathcal{L}}$. The basis will contain right-closed implications $X \longrightarrow \mathcal{L}(X)$ where X is a minimal generator,

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

$$X^{\bullet} = \mathcal{L}(X) - X \cup \bigcup \{\mathcal{L}(B) \mid B \subset X\}$$

in words, $X^\bullet$ is the set of attributes in the closure of X to which no proper subset of X can lead. X is the *"minimal"* set required to get those attributes in $\mathcal{L}(X)$. We redirect the reader to [22] for more details. If $X^\bullet = \mathcal{L}(X) - X$, $X^\bullet$ is pseudo-closed. In fact, we already presented this operation when discussing left-saturation in MinCover. It is also used in [20] to prove DuquenneMinimization.

We use the theory produced by MaierMinimization on both proper and minimal generator basis since the results may differ. algorithms to compute those basis from a context are included in the code we got from `https://github.com/yazevnul/fcai`. We give pseudo-code of the algorithms here, but we will not review them in details. MininalGeneratorBasis (algorithm 18) generates minimal generators. recall that $>_M$ is lectic ordering in M. Given a set we popped from $\mathcal{Q}$, we test all sets with only 1 more elements and lectically greater. Because we begin by the empty set, we are sure to go over all possible subsets whenever they are to be investigated. Using lectic ordering avoids testing a subset twice. Note that implications of $\mathcal{L}$ are indeed right-close. Moreover, the conditional statement for adding an implication in $\mathcal{L}$ is useful since an equivalence class can contain only one set. This (closed) set will be by definition minimal, even though the corresponding implication is useless, whence the test.

Algorithm 18: MinimalGeneratorBasis

Input: A context (G, M, I) and a closure operator $''$
Output: $\mathcal{L}$ the basis of minimal generators

queue $\mathcal{Q}$;
$\mathcal{L} := \emptyset$;
push($\mathcal{Q}$, $\emptyset \longrightarrow \emptyset''$) ;
while $\mathcal{Q} \neq \emptyset$ **do**
  $A \longrightarrow B := \text{pop}(\mathcal{Q})$;
  if $A \neq B$ **then** $\mathcal{L} := \mathcal{L} \cup \{A \longrightarrow B\}$;
  foreach $C \subseteq M$: $(|C| = |A| + 1) \wedge (C >_M A)$ **do**
    if $(C - c)'' \neq C''$, $\forall c \in C$ **then**
      push($\mathcal{Q}$, $C \longrightarrow C''$) ;
return $\mathcal{L}$;

Algorithm ProperBasis (19) computes proper implications of a given context. Again, starting from $\emptyset \longrightarrow \emptyset''$ we cover all possibilities. For all implications of $\mathcal{L}$ we try to build new sets out of its premises and a given attribute. Observe that $D := C'' - C$ is the *"first part"* of saturating C. If C is not closed, then we may refine D by removing elements so that $D = C^\bullet$ after the $D \neq \emptyset$ conditional statement. Especially the second for loop, which removes from D all elements of C'' we would get with another implication than $C \longrightarrow D$.

Having reviewed the application to FCA as much as the main definitions we shall need from this domain, we can move to the description of real datasets we used for our tests.

Algorithm 19: PROPERBASIS

Input: A context (G, M, I) and a closure operator $''$

Output: $\mathcal{L}$ the basis of proper implications

$\mathcal{L} := \emptyset \longrightarrow \emptyset''$;
foreach $m \in M$ **do**
  $\mathcal{L}_t = \mathcal{L}$;
  foreach $A \longrightarrow B \in \mathcal{L}$ **do**
    $C := A \cup \{m\}$;
    $D := C'' - C$;
    if $D \neq \emptyset$ **then**
      foreach $\alpha \longrightarrow \beta \in \mathcal{L}_t$ **do**
        if $(A \subset \alpha) \wedge (\alpha - A \subseteq D)$ **then** remove $\alpha \longrightarrow \beta$ from $\mathcal{L}_t$;
      foreach $\alpha \longrightarrow \beta \in \mathcal{L}$ **do**
        if $\alpha \subset C$ **then** $D := D - \beta$;
    $\mathcal{L} := \mathcal{L} \cup \{C \longrightarrow D\}$;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  if $(A \cup B \neq \emptyset) \wedge (A \neq B)$ **then** $B := B - A$;
return $\mathcal{L}$;

Some real datasets

For our tests, we took the datasets used in [10] from the UCI Machine Learning repository `https://archive.ics.uci.edu/ml/datasets.html`. Therefore, all scaling credits go to the authors of [10]: Sergei Obiedkov and Konstantin Bazhanov. 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"*: diagnostic of breast cancer. For 286 patients, we have 9 multi-valued attributes (nominal or ranged). Getting rid of the first attribute, and removing values never assigned, we end up on 43 attributes by scaling.

- *"Breast Wisconsin"*: the purpose is still diagnostic of breast cancer. Based on 9 attributes scaled from 1 to 10 being various measures and a binary classification attribute, scaling resulted in 91 attributes on 699 records.

- *"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, scaling leads to 26 attributes.

- *"SPECT"*: cardiac data took from tomography. Out of patterns, 23 binary attributes help to provide a diagnosis. Because attributes are binary we do not need scaling. There are 267 objects.

- *"Vote"*: records from a congressional voting in USA (1984). For 435 records, there are 18 binary attributes being resolutions to accept or reject.

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.

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

Table 3.4: Summary of real datasets characteristics

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 [10]. The CPU has 2.4 GHz frequency. Apart from generating random implications out of `boost` and uniform distribution, we took various real datasets from the UCI repository. Those real data we used are the same as ones used when

testing the code we use on closure algorithms. In the next section we will study improvements 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 [10]. Actually, it is exhibited in this paper that LinClosure performs worst in general than Closure. Nevertheless, 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.

Usually we will compare two versions of each algorithm. Moreover, improving the procedures will rely on generating random bases with fixing either the number of attributes ($|\Sigma|$) or the number of implications ($|\mathcal{B}|$). Recall from the paragraph dedicated to random generation that to avoid constant non-redundancy when the number of attributes grows, we generate only a random part of a given number of implications and then, use Armstrong rules randomly to increase the size of the basis up to our need. When $|\mathcal{B}|$ evolves however, we just generate implications since redundancy may occur often.

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 overall 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 MinCover only using Closure and algorithm 20 where we used an improvement for LinClosure.

Algorithm: MinCoverLin

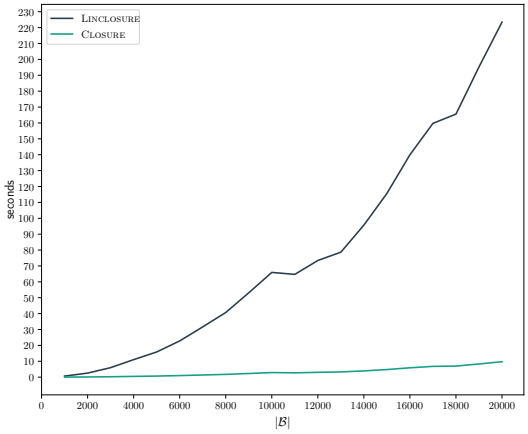
Input: $\mathcal{L}$: an implication base
Output: the canonical base of $\mathcal{L}$

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

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

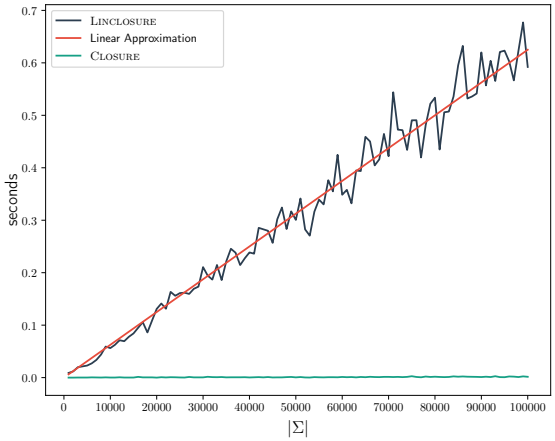
The main idea to test efficiency of each version of the algorithm is to perform two series of tests corresponding to fixing one of the two parameters of its complexity : $|\Sigma|$ or $|\mathcal{B}|$. Each series consists in fixing one of those parameters and testing a range of value for the other. We can observe figure 3.1 as results of those experiments. One may denote in all following tests (for all algorithms) a much more noisy curve when we fix $|\mathcal{B}|$. This has 2 possible explanations. The most important is the way we generate implications as explained previously, it is plausible that we generate a less general sets of basis when $|\Sigma|$ is fixed therefore resulting in less noisy results. Second, as one will observe, times recorded when $|\mathcal{B}|$ is fixed are much smaller and more close from one point to another, increasing differences. In the case where $|\Sigma|$ is fixed however, there are less points usually and because of the impact of $|\mathcal{B}|$ in the cost of computations, we may find less cases where a measure is lower than its ancestors. Consequently, whenever we deal with fixed $|\mathcal{B}|$ we also try to add to graphical representations a *handmade* (always linear) approximation. In every cases, this approximation does *only* deserve to help us see the underlying behaviour of the curve. To plot it, we compute a slope out of our points. We should keep this explanation in mind for all upcoming results, for all algorithms.



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

$|\mathcal{B}|$	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, $|\Sigma| = 100$



(c) Average time (in s), $|\mathcal{B}| = 100$

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. Below is $|\mathcal{B}| = 100$ and $|\Sigma|$ running from 1000 to 100000. For the first one we associated an array (on the right) with some particular values, being useful to see evolution of Closure, flattened by LinClosure. In the case of $|\mathcal{B}| = 100$, MinCover using Closure does not exceed $1.e^{-4}$ seconds. 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 the two versions of MinCover 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. For this reason though, we may reduce the range of testing to 10000 with $|\mathcal{B}|$ in subsequent experiments. According to the results, we admitted this range was sufficient to see the gap between those two closure operators. 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 corresponds to theoretical complexity $O(|\mathcal{B}||\mathcal{L}|)$ required by MinCover. In fact, when we fix $|\mathcal{B}|$, the performance of LinClosure can be explained by denoting that, when computing a closure it first runs over attributes (*list*), and then implications. Therefore, when $|\Sigma| \gg |\mathcal{B}|$ a run of LinClosure becomes extremely expensive. When $|\Sigma|$ is fixed however, we can see that the gap between the two versions of MinCover reduces. An idea is: because we have few attributes and several implications, an implication is likely to appear several time among the implication lists, therefore increasing the number of elements to proceed. Plus, it may appear for the same reason that no more than 2 run over $\mathcal{L}$ are sufficient for Closure to compute closures. Still, because benchmarking Closure and LinClosure on their own is not our objective, we must leave these hypothesis for further experiment.

One could argue that the algorithms we tested both used Closure in the second loop, altering the overall 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

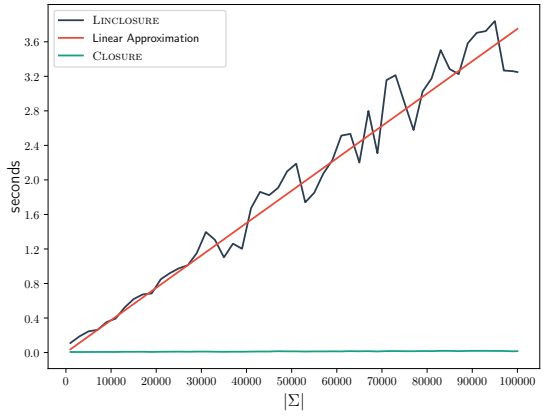
In the algorithm provided by Duquenne, the first step consists in redundancy elimination. For this step, let us use Closure anyway since as we explained in previous paragraphs, removing an implications in counters and lists may be as time consuming as a run of LinClosure itself. However, in the second loop, we could use an improvement of LinClosure provided we do not remove any implication from $\mathcal{L}$. In this algorithm, removal occurs when a premise is not pseudo-closed, but the closure system defined by $\mathcal{L}$ does not change. Hence, we could instead keep such useless implication in $\mathcal{L}$ and use previously computed lists and counters to speed up closure computations. Here again, we test two versions of the algorithm: either only Closure, or using an improvement of LinClosure as following pseudo-code shows.

Algorithm: DuquenneMinimization (LinClosure), last step

Input: $\mathcal{L}$ a theory to minimize (non-redundant, lectically ordered)
Output: $\mathcal{L}_c$ the DQ-basis of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$;
LinClosureInitCounters($\mathcal{L}$) ;
foreach $A \longrightarrow B \in \mathcal{L}$ **do**
  foreach $\alpha \longrightarrow \beta \in \mathcal{L}_c$ **do**
    if $\alpha \subset A \wedge \beta \nsubseteq A$ **goto next** $A \longrightarrow B \in \mathcal{L}$;
  $B = \text{LinClosure}(\mathcal{L}, B)$;
  $\mathcal{L}_c := \mathcal{L}_c \cup \{A \longrightarrow B\}$;
return $\mathcal{L}_c$;

First, let us focus on the behaviour of those two algorithms when we fix the number of implications to 500, and Σ runs from 1000 to 100000 attributes (see figure 3.2). As previously supposed, and also exposed in [10], keeping track of lists and counters in LinClosure is extremely expensive. Hence on this point, Closure is a clear winner.



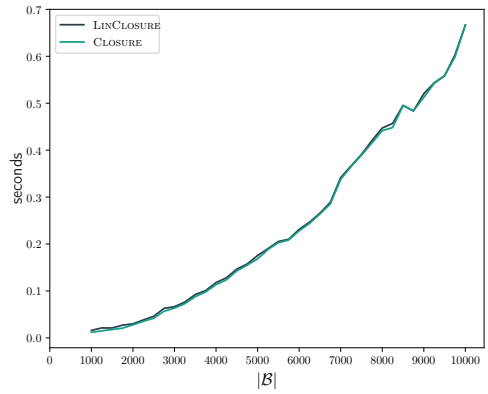
(a) Graphical representation of average time (in s)

$\lvert\Sigma\rvert$	Closure
1000	0.007
5000	0.007
25000	0.011
45000	0.012
65000	0.015
85000	0.019
100000	0.016

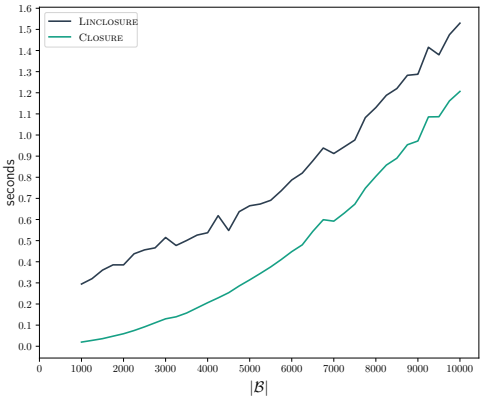
(b) Some landmarks times

Figure 3.2: Average execution time over 100 bases of DuquenneMinimization when $|\mathcal{B}| = 500$

However, let us observe what happens when we fix $|\Sigma|$. We try Σ with 100 and 500 attributes (see figure 3.3). Here seeing which one is the most efficient is much more complicated. Not only Closure and LinClosure have the same efficiency when $|\Sigma| = 100$, but also the gap between them shrinks when we set $|\Sigma| = 500$. In both cases $|\mathcal{B}|$ goes from 1000 to 10000.



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



(b) Average time (in s), $|\Sigma| = 500$

$\lvert\mathcal{B}\rvert$	LinClosure	Closure
1000	0.016	0.012
4000	0.118	0.113
7000	0.341	0.337
10000	0.667	0.666

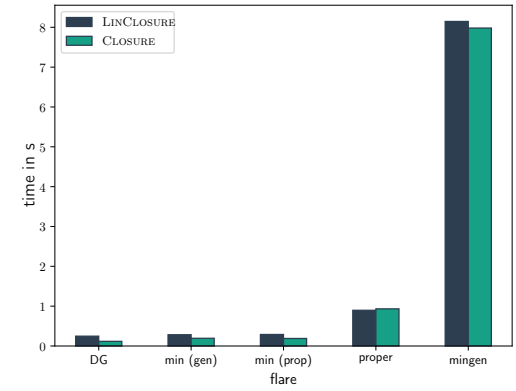
(c) Some landmarks times, $|\Sigma| = 100$

$\lvert\mathcal{B}\rvert$	LinClosure	Closure
1000	0.295	0.020
4000	0.537	0.206
7000	0.912	0.592
10000	1.530	1.206

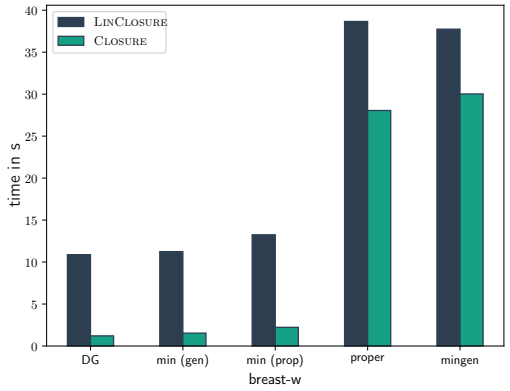
(d) Some landmarks times, $|\Sigma| = 500$

Figure 3.3: Comparison of closure operators for DuquenneMinimization

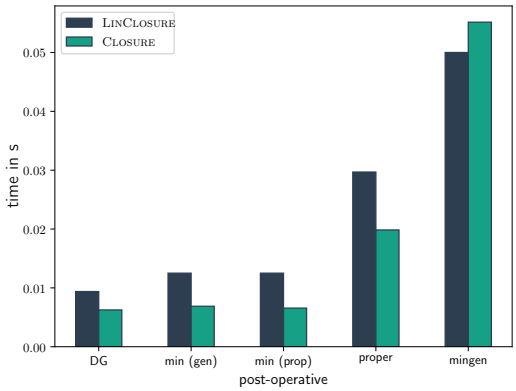
These remarks suggests that when we increase the number of implications and not the size of Σ, LinClosure is asymptotically better than Closure even in practice. This hypothesis corroborate observations of [10]. Thus, to see which versions will be the most efficient when confronted to real datasets, we also tested them on our real data. Because we are more interested in determining which one is the most efficient, we present results through bar plots because they emphasize differences. Report to figure 3.4



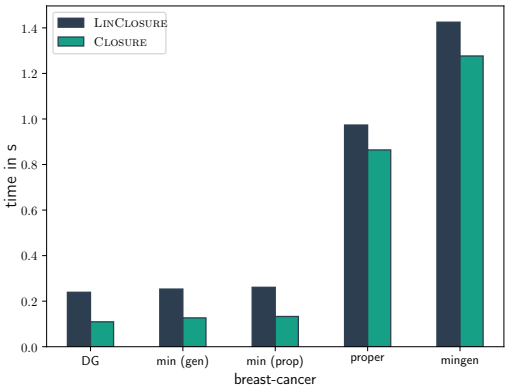
(a) Flare dataset: $|\Sigma| = 49$



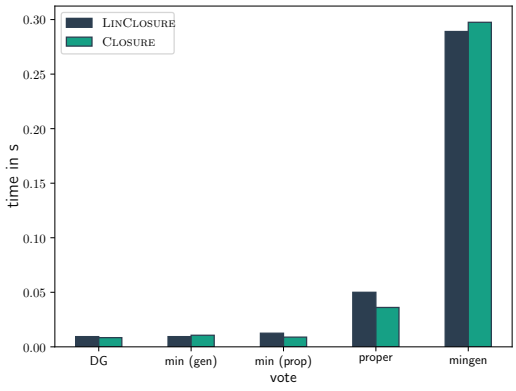
(b) Breast Wisconsin: $|\Sigma| = 91$



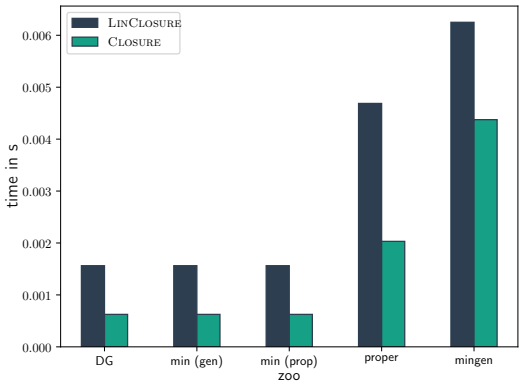
(c) Post operative: $|\Sigma| = 26$



(d) Breast Cancer: $|\Sigma| = 43$



(e) Vote: $|\Sigma| = 18$



(f) Zoo: $|\Sigma| = 28$

Figure 3.4: DuquenneMinimization tested with Closure, LinClosure on real datasets

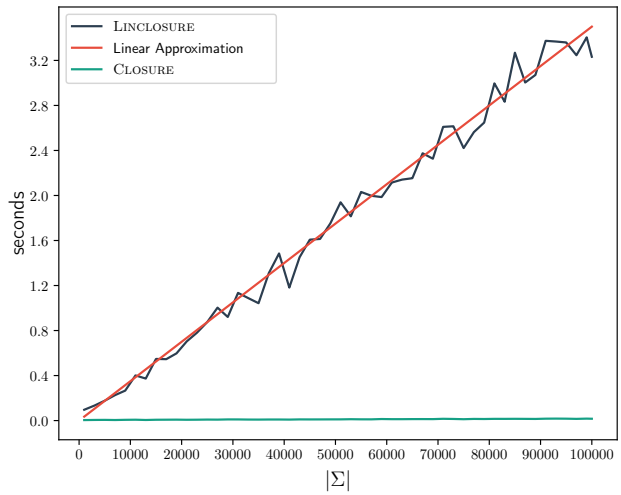
We represented in those bar plots the time required by both studied versions of DuquenneMinimization to minimize the basis we derive from our bag of real datasets. We do not represent the *SPECT* dataset because most of the values are near 0 except for minimal generators resulting in a non that attractive view point. Furthermore, it does not bring more informations. We can see that in general Closure still performs better than LinClosure even though they are very close for several results. Nevertheless LinClosure performs slightly better than Closure on some minimal generators basis having usually the most implications, strengthening our hypothesis on the asymptotic behaviour of LinClosure. However, this is clearly *"Breast Wisconsin"* dataset having the most attributes and implications and still, this is one of the dataset where the gap between LinClosure and Closure is the widest. This may be because $|\Sigma|$ is somehow large also, as we have seen in MinCover. Another possible explanation is the structure of randomly generated basis as compared to real data. This is left for further research though. Still, we can conclude here that in spite of our observations, Closure remains a better choice for real data at least.

3.2.3 MaierMinimization

Recall that the procedure given by Maier relies on three steps (on some theory $\mathcal{L}$): removing redundant implications, finding equivalence classes among premises of $\mathcal{L}$, removing direct determinations. Because we remove implications in the first and last steps, we applied Closure in all cases for those parts. However, when it comes at determining equivalence classes, one can use only one initialization of counters and list for LinClosure. For this reason we tested two versions of MaierMinimization: one only using Closure, the other using LinClosure for the second step. Again, we have two experiments:

1. $|\mathcal{B}| = 500$, $|\Sigma|$ from 1000 to 100000,
2. $|\Sigma| \in \{100, 500\}$, $|\mathcal{B}|$ from 1000 to 10000

the results of first case are given in figure 3.5. As in previous cases, Closure performs better than LinClosure because it does not run over attributes. However, even though they do not have been tested on same theories, times recorded are similar to times of DuquenneMinimization for the same parameters. Furthermore they are the lower bounds of times we record over all algorithms as we will see in next sections. This similarity is not limited to one parameter, and we can also have a look at the second test case in figure 3.6.

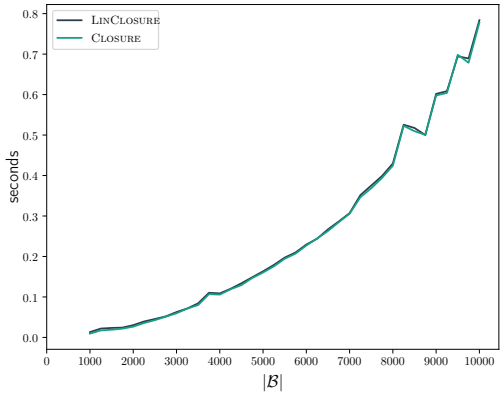


(a) Graphical representation of average time (in s)

$\lvert\Sigma\rvert$	Closure
1000	0.004
5000	0.006
25000	0.009
45000	0.010
65000	0.013
85000	0.016
100000	0.016

(b) Some landmarks times

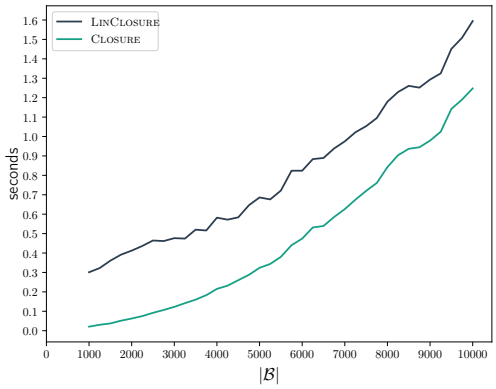
Figure 3.5: Average execution time of MaierMinimization when $|\mathcal{B}| = 500$



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

$\lvert\mathcal{B}\rvert$	LinClosure	Closure
1000	0.013	0.009
4000	0.109	0.106
7000	0.307	0.306
10000	0.783	0.778

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



(b) Average time (in s), $|\Sigma| = 500$

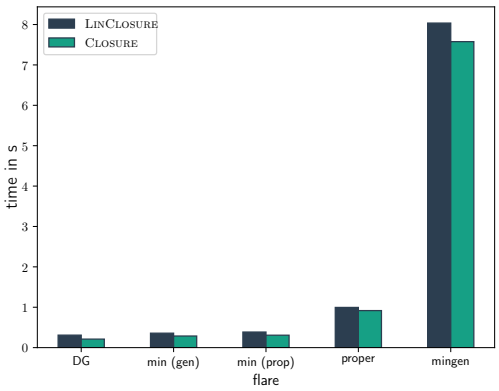
$\lvert\mathcal{B}\rvert$	LinClosure	Closure
1000	0.301	0.021
4000	0.582	0.215
7000	0.975	0.626
10000	1.594	1.248

(d) Some landmarks times, $|\Sigma| = 500$

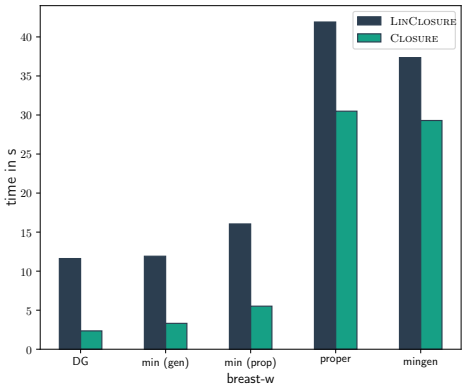
Figure 3.6: Comparison of closure operators for MaierMinimization

In this figure too, minimization time is close from DuquenneMinimization. We will discuss this proximity also in joint comparison, but let us observe that they do in fact the same redundancy elimination. This step may also explain their efficiency in practice as compared to other algorithms, especially MinCover expected to have the same complexity. Because theories in practice are likely to be redundant, removing them at first sight provides a speed up in following closure computations and loops over the basis we minimize. Furthermore, because we keep the same number of implications in both algorithms (due to minimality on termination) we are likely to perform a similar number of operations in both cases. In DuquenneMinimization, apart from ordering, we first check whether an implication has pseudo-closed premise being $|\mathcal{L}|$ price. If it is the case, we require one more closure computation. If it is not we discard the implication and spare closure operations as much as some inclusion tests in $|\mathcal{L}|$. In MaierMinimization, we are sure to figure out at least one closure for each implication. However we may avoid the second one whenever we find direct determination. In both algorithms we have this principle of avoiding as much operations as possible whenever we declare redundant an implication. And also we are sure to perform all of them on exactly the same number of implications.

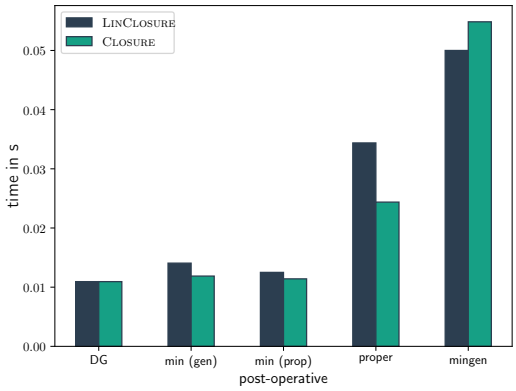
Since the behaviour of MaierMinimization is somehow similar to DuquenneMinimization we also ran tests on our real data to see which closure operators suits well real life (see figure 3.7). In this figure we omitted *SPECT* because of lack of visibility. However, cases where LinClosure is better are more frequent than for DuquenneMinimization. This is in particular the case for the whole *Vote* dataset. Still, Closure is overall more efficient and we will keep the associated version as the fastest one.



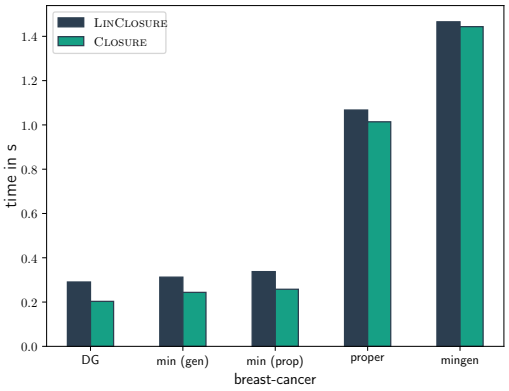
(a) Flare dataset: $|\Sigma| = 49$



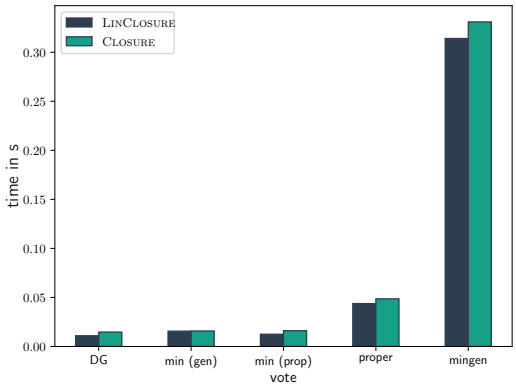
(b) Breast Wisconsin: $|\Sigma| = 91$



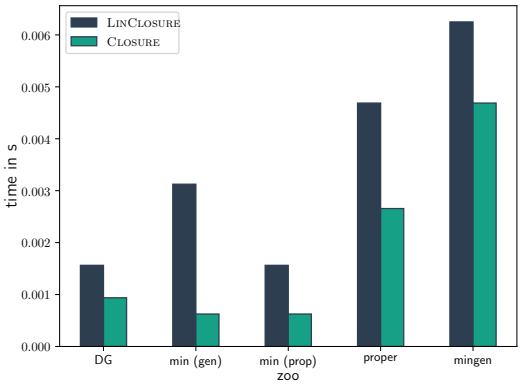
(c) Post operative: $|\Sigma| = 26$



(d) Breast Cancer: $|\Sigma| = 43$



(e) Vote: $|\Sigma| = 18$



(f) Zoo: $|\Sigma| = 28$

Figure 3.7: MaierMinimization tested with Closure, LinClosure on real datasets

3.2.4 BerczIMinimization

The algorithm issued by Berczi and al in [16] (algoritm 15) 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. Moreover, because we build $\mathcal{L}_c$ only by adding implications, all the closure previously computed can only grow. Therefore we can improve BerczIMinimization by:

- computing all $\mathcal{L}(A), A \in \mathcal{B}(\mathcal{L})$, and store them in a list: $C_{\mathcal{L}}$,
- keeping 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 BerczIImp. 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 $\mathcal{L}$.

Algorithm 20: 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 $\mathcal{L}_c$;

When getting the minimum, note that we get both the minimal $\mathcal{L}_c$-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.

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 $C_{\mathcal{L}}[A] = C_{\mathcal{L}_c}[A]$.

$\mathcal{B}(\mathcal{L})$	$C_{\mathcal{L}}$	$C_{\mathcal{L}_c}$	$=$
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 $C_{\mathcal{L}_c}$

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, 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, using only Closure and one using improvements of LinClosure (see 23).

Algorithm: BercziImpLin

Input: $\mathcal{L}$: an implication theory
Output: $\mathcal{L}_c$: the DG 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**
  $C_{\mathcal{L}}[A] =$ LinClosure($\mathcal{L}, A$) ;
  $C_{\mathcal{L}_c}[A] = A$;

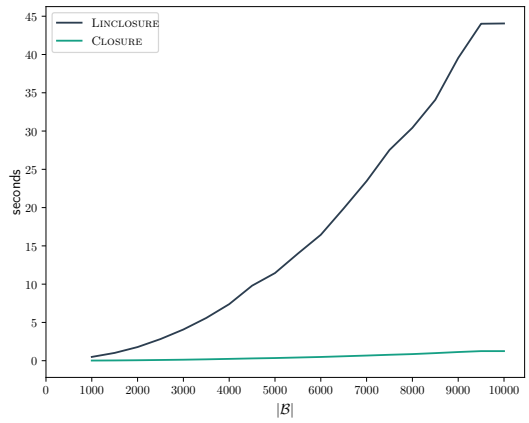
while $\exists A \in \mathcal{B}(\mathcal{L})$ *s.t* $C_{\mathcal{L}_c}[A] \neq C_{\mathcal{L}}[A]$ **do**
  foreach $A \longrightarrow B \in \mathcal{L}$ **do**
    if $C_{\mathcal{L}}[A] \neq C_{\mathcal{L}_c}[A]$ **then**
      $C_{\mathcal{L}_c}[A] =$ LinClosure($\mathcal{L}_c, C_{\mathcal{L}_c}[A]$)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

return $\mathcal{L}_c$;

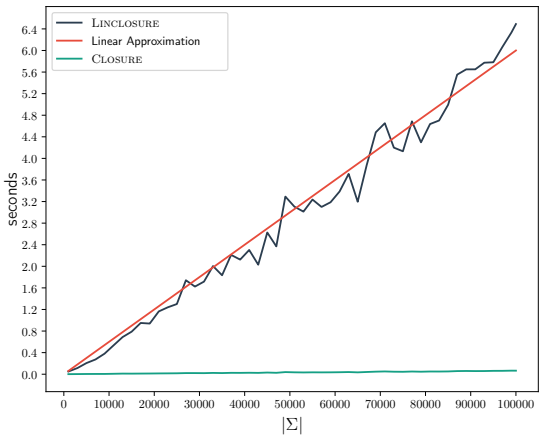
When dealing with LinClosure in BercziImpLin, we use first LinClosureInit to instanciate *list* and *count* for LinClosure in $\mathcal{L}$. Hence a call to this procedure performs only the effective closure computations. Then, LinClosureAddImp just updates *list* and *count* for $\mathcal{L}_c$ by adding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

We may observe tests of BercziMinimization (or more precisely, BercziImp) in graphs and tables of figure 3.8. As in previous cases we represent two range-based test on one parameter, when the other is fixed. On the left part we have a graph and a table of tests with fixed $|\Sigma| = 100$. On the other side, we fixed $|\mathcal{B}|$ to 100 also (and not 500 as in previous cases!). We lowered the boundary $|\mathcal{B}|$ because of the complexity of the algorithm. As one can observe if we recall previous tests, even if $|\mathcal{B}| = 100$, times are still higher than previous tests where $|\mathcal{B}| = 500$. Or course we should consider this quick comparison as no more but a quick glance to previous results since tests are not ran on the same bases (because of randomness). In any event, in spite of the noisy behaviour when $|\Sigma|$ tends to 100000, we can observe a linear growth of the execution time in accordance with the theoretical complexity of BercziMinimization.

Regarding the evolution of time when $|\Sigma|$ is fixed we may perceive a polynomial growth of the time



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



(b) Average time (in s), $|\mathcal{B}| = 100$

$\lvert\mathcal{B}\rvert$	LinClosure	Closure
1000	0.494	0.019
4000	7.387	0.237
7000	23.460	0.673
10000	44.054	1.252

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

$\lvert\Sigma\rvert$	Closure
1000	0.001
5000	0.004
25000	0.017
45000	0.031
65000	0.034
85000	0.052
100000	0.065

(d) Some landmarks times, $|\mathcal{B}| = 100$

Figure 3.8: Comparison of closure operators for BercziMinimization

required by BercziMinimization when used with LinClosure matching theoretical expectations. Again, the difference between Closure and LinClosure is clearly visible and we can take Closure as our fastest operator.

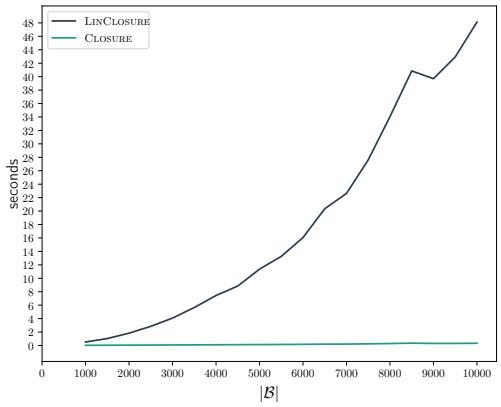
3.2.5 AFPMinimization

Regarding AFPMinimization, we can consider using an optimization as previously mentioned on $\mathcal{L}$, our input basis. Because $\mathcal{L}$ is not altered, we can consider initializing counters only one time for this theory. For $\mathcal{L}_c$, when we refine an implication we may need not only to update the counter of the implication, but also to go over every list to potentially remove the refined implication, being as complex as LinClosure itself, unless we perform the same operation as mentioned for deleting an implication in MinCover which we did not test due to lack of time and already observed difference between LinClosure and Closure.

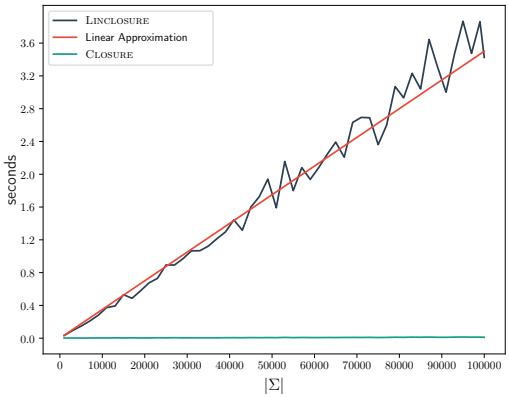
Hence we test here two versions: 1 - only using Closure, 2 - using LinClosure for $\mathcal{L}$ with InitCoun-

TERS jointly with CLOSURE for $\mathcal{L}_c$. We do not write the two procedures here since it may be too heavy (regarding the structure of AFPMINIMIZATION) for operations we already developed. Results of our tests are represented in figure 3.9. As in previous cases we tested both $|\Sigma|$ and $|\mathcal{B}|$. While it appears as in most of the algorithm we studied that even under evolution of $|\mathcal{B}|$, CLOSURE is a clear winner, the overall behaviour informs us on the complexity of AFPMINIMIZATION. In particular it appears that $|\Sigma|$ has only a linear cost while $|\mathcal{B}|$ is polynomial. This tells us that the repeat-until loop dealing with the stack in AFP does not require a time proportional to $|\Sigma|$. However it may be dependant of $|\mathcal{B}|$ which means that we stack a number of implications similar to $|\mathcal{B}|$ at most. In other words, this goes in the direction of the hypothesis where an implication is refine only once. Nevertheless, this acts just as an observation or an hint in this direction and still we are dealing here with practical complexity. We have seen on almost all algorithms that in spite of its expensive theoretical complexity, CLOSURE was much more appealing than LINCLOSURE. Consequently, we should not take this experiment as a general proof of complexity.

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 worse than the other operator. Potentially, it becomes faster when we deal with a very large amount of implications as shown with DUQUENNEMINIMIZATION and MAIERMINIMIZATION. However when tested under real data, it seems like CLOSURE remains, within our set of parameters the best operator. This is in accordance with conclusions and hypothesis drawn in [10] about efficiency of LINCLOSURE as compared to the effective structure of real data and randomly generated systems. Furthermore, those tests allowed us to have a glimpse of the correspondence between theoretical and practical complexity. Also they began to draw argument of practical interest with redundancy elimination. 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.



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



(b) Average time (in s), $|\mathcal{B}| = 100$

$\lvert\mathcal{B}\rvert$	LinClosure	Closure
1000	0.519	0.028
4000	7.437	0.107
7000	22.627	0.208
10000	48.116	0.333

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

$\lvert\mathcal{B}\rvert$	Closure
1000	0.002
5000	0.001
25000	0.004
45000	0.007
65000	0.008
85000	0.012
100000	0.012

(d) Some landmarks times, $|\mathcal{B}| = 100$

Figure 3.9: Comparison of closure operators for AFPMinimization

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 35 bases, 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 table 3.6. For clarity, recall that all of the algorithms use only Closure.

First, one can observe the wide range of performances. On the one hand, MinCover, DuquenneMinimization and MaierMinimization do not exceed 100 seconds of execution time, while BercziMinimization and AFPMinimization can require up to several hours. The most striking case is for *Breast Wisconsin*, 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 70 seconds for minimization, BercziMinimization 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.

$\mathcal{L}$		MinCover	Duquenne	Maier	Berczi	AFP
Zoo	DG	< 0.001	< 0.001	< 0.001	< 0.001	0.016
	min (mingen)	< 0.001	< 0.001	< 0.001	< 0.001	0.016
	min (proper)	< 0.001	< 0.001	< 0.001	< 0.001	0.016
	proper	0.005	0.002	0.003	0.016	0.063
	mingen	0.007	0.004	0.005	0.047	0.094
Flare	DG	0.097	0.117	0.211	27.922	115.656
	min (mingen)	0.134	0.194	0.288	27.750	116.594
	min (proper)	0.200	0.190	0.308	30.063	108.812
	proper	1.684	0.933	0.917	88.375	524.031
	mingen	16.047	7.981	7.576	160.328	2810.620
Breast Cancer	DG	0.089	0.109	0.203	33.047	90.031
	min (mingen)	0.108	0.126	0.243	26.578	89.516
	min (proper)	0.153	0.132	0.258	29.438	105.141
	proper	1.920	0.864	1.014	93.266	429.844
	mingen	2.006	1.277	1.444	102.562	598.172
Breast Wisconsin	DG	0.940	1.208	2.348	1005.750	3109.920
	min (mingen)	1.277	1.533	3.314	949.953	3140.940
	min (proper)	3.121	2.225	5.514	1682.03	5408.98
	proper	67.147	28.053	30.488	3675.910	40521.0
	mingen	35.488	30.023	29.295	2772.980	38310.200
Operative	DG	0.005	0.006	0.011	0.219	0.734
	min (mingen)	0.007	0.007	0.012	0.219	0.719
	min (proper)	0.006	0.007	0.011	0.234	0.640
	proper	0.039	0.020	0.024	0.594	2.422
	mingen	0.078	0.055	0.055	0.813	4.063
SPECT	DG	0.045	0.066	0.108	10.328	23.609
	min (mingen)	0.061	0.080	0.134	8.156	22.906
	min (proper)	0.078	0.070	0.150	8.250	23.031
	proper	0.930	0.394	0.451	51.063	118.531
	mingen	24.077	10.206	10.858	194.875	930.578
Vote	DG	0.006	0.008	0.015	0.484	1.579
	min (mingen)	0.007	0.011	0.016	0.469	1.516
	min (proper)	0.008	0.009	0.016	0.469	1.593
	proper	0.076	0.036	0.049	1.625	6.203
	mingen	0.584	0.298	0.331	4.109	22.875

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

Regarding the difference between MinCover and DuquenneMinimization, observe that in general, the algorithm by Duquenne is faster on non-minimum bases and slightly slower on already minimum ones (but the minimum (proper) case we will consider after). In both cases, one hypothesis to explain those gaps could be both redundancy elimination and limitation of closure computations in the second loop of DuquenneMinimization. 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 several 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 base to check whether an implication is redundant or not. The only case where the second loop should require more computations is when the base is already minimum, because for each step of the second loop, we have to go over all implications of the growing output system. As remarked in most of execution times, when the basis is already minimum DuquenneMinimization performs somehow slightly worse than MinCover, which could be explained by the previous hypothesis. However, as we mentioned at the beginning of this paragraph, when it comes at minimizing the minimum base we obtained from the proper one, DuquenneMinimization tends to be a bit better. In fact, implications of the proper basis are not right-closed, therefore there is a possibility in the first loop of MinCover (because we use Closure) to run over all implications twice: one loop is sufficient to get the closure, but a second is required by the algorithm to exhibit no updates. And again, because of non-redundancy elimination at this step, DuquenneMinimization may get the point by first removing numerous redundant implications reducing the time spent in closures.

One can also denote that MaierMinimization seems to be as efficient as DuquenneMinimization except in the minimum cases where it performs worse (compared to DuquenneMinimization and MinCover). 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. Once the first step is done in both DuquenneMinimization and MaierMinimization they may perform a similar amount of operations explaining the small gap between them for non-minimum bases:

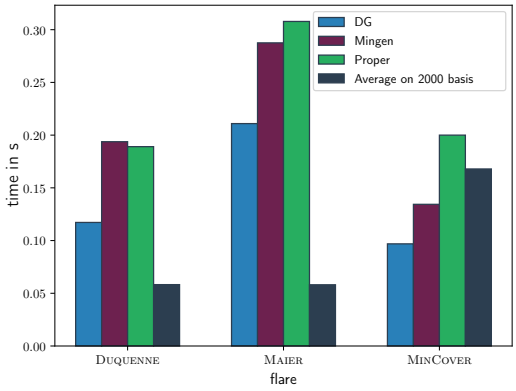
- DuquenneMinimization performs ordering and then, for all implications, either we need less than $|\mathcal{B}||\Sigma|$ time (because we break the for loop whenever we observe a set is not pseudo-closed) or we will require $|\mathcal{B}||\Sigma|$ + the time for closing since we need to run across $\mathcal{L}_c$ and computing a closure,
- For MaierMinimization, we are sure to perform at least one closure for all implications, to determine equivalence classes. Nevertheless, notice that when removing direct determination, we perform another full closure only if the implication must be kept in the base, otherwise we stop once we found direct determination.

In both case we are likely to perform the same number of set operations if an implication has not to be removed. Still, the difference can rely on which of lectic ordering or determining equivalence classes (which is also a run over a $|\mathcal{B}| \times |\mathcal{B}|$ matrix!). So far, we did not find the true explanations. One possible test to

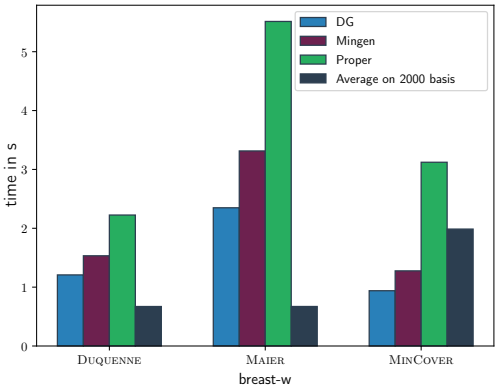
run in further work is using a profiler to see whether the small difference is explained by performance of the computer, or by the structure of the basis we are minimizing.

However, our discussion yields another idea on the worst case of minimization. In fact, the case where we would require more time given fixed parameters $|\Sigma|$ and $|\mathcal{B}|$, is when there is no implication to remove. From our point of view, having no implication to remove maximizes the computations, because all operations of removing redundant implications, figuring out closures and so forth do not reduce the basis. To see whether our idea is interesting or not, we ran few more tests: we took the first three algorithms (because they are faster) and characteristics of the minimum bases. Then, we took the average minimization time over 2000 randomly generated theories of corresponding size. We omit *"zoo"* and *"Breast cancer"* datasets since the first one has very low results and the characteristics of the second one are almost the same as *"Flare"*. Results are presented in figure 3.10.

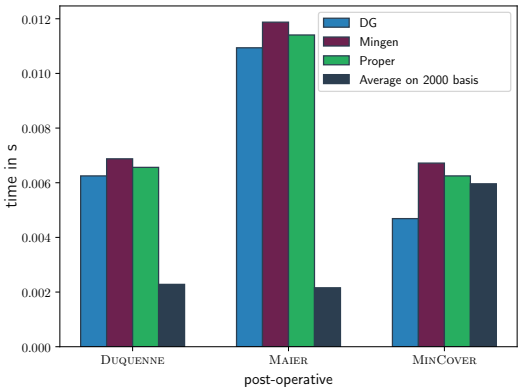
Each graph corresponds to a dataset. Inside all of them, we have a series of bar plots per algorithm. In order, one may find the time (in seconds) required for the Duquenne-Guigues basis, the minimum basis we had with the minimal generators, with proper implications and eventually the average time we obtained for minimization over 2000 randomly generated bases. For DuquenneMinimization and MaierMinimization, it seems like our hypothesis holds: on average, minimizing a redundant system is less time consuming than a base being minimum, for fixed $|\Sigma|$ and $|\mathcal{B}|$. However, this does not hold for MinCover for which the worst case is often the minimum bases with non right-closed implications. Moreover reducing a redundant theory is more expensive on average than minimizing right-closed bases. Our idea is that right-closed results in fast closure computations, because for each implication the information we may get is maximal. In the case of non-closed implications, we are likely to need several runs over $\mathcal{L}$ for this task. Then, depending on the order in which we consider implications in the second loop, the execution time may vary. If all redundant implications are handled before others, we will spare several operations. On the contrary, if they are all placed at the end, we will maximize the cost.



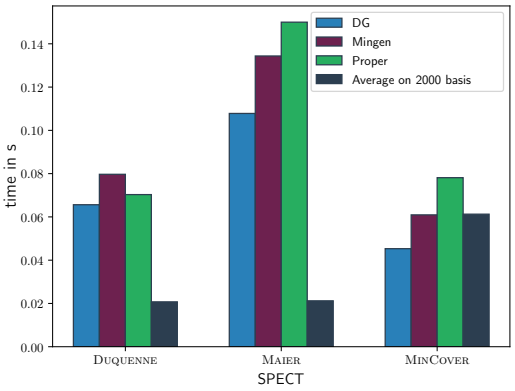
(a) Flare dataset: $|\Sigma| = 49$, $|\mathcal{B}| = 3382$



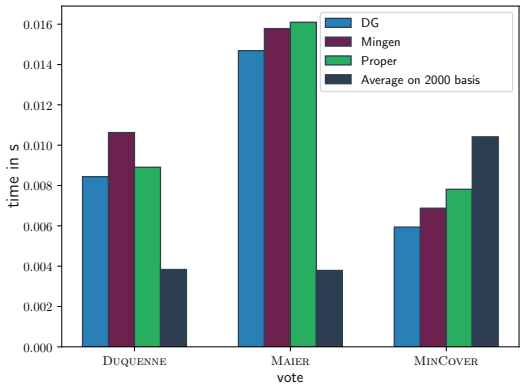
(b) Breast Wisconsin: $|\Sigma| = 91$, $|\mathcal{B}| = 10640$



(c) Post operative: $|\Sigma| = 26$, $|\mathcal{B}| = 625$



(d) SPECT: $|\Sigma| = 23$, $|\mathcal{B}| = 2169$



(e) Vote: $|\Sigma| = 18$, $|\mathcal{B}| = 849$

Figure 3.10: Average minimization against minimal basis

If we were to summarize all our results, we observed that as expected in [10], LinClosure is much less efficient than Closure in practice. Nevertheless, it usually helped us to observe the correlation between theoretical and practical complexity. Regarding the algorithms, we have shown that BerczıMinimization and AFPMinimization were not efficient, especially the second one for which we still lack the proof. Among MinCover, DuquenneMinimization and MaierMinimization we observed that the two last ones were more efficient than MinCover when dealing with redundant bases. This case is very likely to occur in real life applications and if one is to use or design an algorithm for minimization, we should either use one of DuquenneMinimization, MaierMinimization or consider removing redundant implications first to lighten subsequent computations. Noting that MinCover was fast in minimal right-closed case rises another idea: trying to use right-close implications to maximize the information we get. Of course, this presumes a closure-based approach and in fact closure operations lead the complexity of the algorithms. Another idea (as mentioned in [20]) would be to find a way to avoid closures if possible.We would like to remind that those results are valid within the scope of our tests and can be somehow related to our way of generating random data for instance. We cannot assume it in general and it should be tested in further work on other datasets. Plus, one could enquiry about more statistical measures to sharpen the understanding of those procedures. This relativity allows for perspective in testing, trying to increase the number of real datasets, testing more combinations of closure operators or even extending range-based tests provided some more power or more time.

3.4 Perspectives and difficulties

So far we have developed and explained our work during this master thesis. However, there are two points we should mention at last being in a sense out of the scope of the algorithms, namely difficulties we encountered and perspectives our study leave for future work.

First, let us discuss difficulties. Apart from having to deal with various mathematical frameworks we concentrated to closure systems, some algorithms or articles appeared to be very demanding. This the case for instance with the work of Ausiello and al. which took us about one month of work before concluding to a potential mistake. This is also the case for AFP's correctness for which we may still have few hints for further research. Often, those technical difficulties to understand some articles as much as knowing whether to study or not some frameworks found solutions in discussions with the academic supervisor. Tests have also been a problem since most of them are time-consuming (up to several days) relatively to the time we had to work on this thesis. Even though tests were performed in parallel with other work, having only one computer slows down tests because of side tasks. Still, we got some results allowing for hypothesis on the behaviour of several algorithms leading to hints and tracks to follow for future work.

Speaking about future work, let us talk about perspectives. We provided both theoretical and practical review. The two approaches benefit from possible improvement or in-depth look. First on a theoretical point of view, the algorithm AFPMinimization derived from query learning still requires a proof of correctness, even if we observed a dreadful efficiency in practice. Furthermore, we may still investigate Ausiello to sharpen our insight of redundancy elimination and removing superfluous nodes. An interesting question

drawing a link between theoretical and practical study would be to get more understanding of redundancy so as to regulate its generation under testing and improving algorithms by pointing out special kinds of redundancies. Talking about testing, the main perspective on a practical side would be first to run tests on new real datasets to see whether our hypothesis hold or not. One may also be interested in testing more combinations of closure algorithms, finding more optimizations in each minimization procedure or even conducting experiment on the structure of randomly generated systems.

Conclusion After a theoretical study of some minimization algorithms, we finally got to implementation in C++. After having explained our experiment framework through randomization, short introduction to FCA and real datasets, we were first interested in finding which closure operator has better behaviour. Within the scope of our tests, Closure has been better even though we denoted a possibility for LinClosure to overcome its cousin when increasing implication basis size. Then we moved to joint comparison of the procedure on real data. Out of this comparison we observed that in practice, the algorithm from Berczi and AFP procedure were inefficient. MinCover however has good results with minimum bases, especially when they are right-closed. Furthermore we saw that Maier's and Duquenne's functions where the most efficient with non-minimum systems probably thanks to their redundancy elimination step. Hence, we concluded that in practice one should be interested in this elimination to decrease execution time.

Conclusion

In this master thesis, we were interested in the problem of implication theories minimization which is: having a system of implications, trying to remove as much elements as possible without altering the knowledge described by the system. More precisely, we had to review existing algorithms and implement them to see how do they perform under practical computations.

To answer this problematic, we first tried to find in literature existing algorithms matching our minimization task. Then we brought to the field of closure spaces and implications some algorithms to study them. In this review, we gave elements of proof and complexity analysis. Because of some difficulties, errors or similarities in algorithms we found, we can still improve our insight on some of them as a perspective of our study.

Consequently to this review, and using a code previously developed for FCA and closure computing purposes, we implemented the algorithms we worked on in C++. Next, after setting up a testing framework, we ran tests on implemented procedure. Experiments gave results on the behaviour of algorithms and their efficiency in practice. We exhibited possible improvements in the implementation of each procedure and accordingly, difference of speed for closure operators. We also depicted some practical conclusions on which algorithm to use depending on the case or at least, some step to consider when trying to think of another way to minimize systems with a closure-based approach.

Finally, our work opens some possibilities for future research and experiments. One could for instance be interested in studying a better way to handle the structure (notably in terms of redundancy) of randomly generated data as compared to real life datasets. More generally, we could be interested in extending our tests to larger datasets and a broader range of improvements for closure operators to strengthen our hypothesis.

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] Armstrong, W. W. Dependency Structures of Data Base Relationships. In *IFIP Congress* (1974), pp. 580–583.

[5] 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.

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

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

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

[9] Babin, M. A., and Kuznetsov, S. O. Computing premises of a minimal cover of functional dependencies is intractable. *Discrete Applied Mathematics 161*, 6 (Apr. 2013), 742–749.

[10] 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.

[11] 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.

[12] 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).

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[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 Obiedkov, S. *Conceptual Exploration*. Springer, 2016.

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

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

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

[26] Maier, D. Minimum Covers in Relational Database Model. *J. ACM 27*, 4 (1980), 664 – 674.

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

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

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

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

[31] 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.

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