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

1 rue de la Chebarde
Aubières, 63178, France

NATIONAL RESEARCH UNIVERSITY

Higher School of Economics

Kochnovskiy Proyezd, 3
Moscow, 125319, Russia

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

Horn Minimization: an overview of existing algorithms

Author : Simon Vilmin

Academic Supervisor : Sergei A. Obiedkov

Held : June, 26, 2018

Acknowledgement

List of Figures

List of Algorithms

Abstract

Contents

Introduction

Chapter 1

Definitions, Framework

pseudo-closed set pouet pouet. closure hus hus. beignet and pouf.

In this chapter, we will focus on giving definitions of the mathematical objects we may need later. As we shall see, each section giving definitions is a different way to represent the same idea. Then, we will introduce the Horn Minimization task without any algorithm, to provide the reader with some intuition and simple examples of our problem. A more detailed description of the minimization task and existing solutions or studies will be given in chapter 2.

To be more precise, we develop first the main framework we may use to reason: closure systems over attribute sets. We will often try to think of various algorithms and proofs within this framework to have a red line to follow. Then, we approach propositional logic and few elements of graph theory. To emphasize the links between various aspects, we will try to use the same notations on equivalent notions.

1.1 Introduction to implications through closure systems

In this section, we will establish basic definitions and provide some examples to have a sufficient (and strong enough) background to understand the topic of Horn minimization. We assume some knowledge of set theory. For more complete and detailed introduction to our topic, the reader may refer to [6, 12]. Some of the most practical applications of the following material dwell into artificial intelligence, with *Formal Concept Analysis* and *Attribute Exploration*. We can find other applications within database field (see [15]). For now, let us begin with an example to illustrate implications.

Example Let us imagine we are provided with a set of music genres: *shoegaze, electronic, coldwave, pop, rock, dream-pop, jazz, experimental, atmospheric, contemporary jazz* Assume we can attach various genre to a given music. Our aim is to draw inference of styles with other ones. In other words, say we have a music with tags *rock, pop*, can we deduce other tag? Remind this example aims to illustrate, not to settle any musical knowledge. Let's try to give some ideas:

- a music being *jazz, experimental* can also be categorized as *contemporary jazz*,

- a music called *coldwave, rock* is likely to be tagged *shoegaze*,
- song with *atmospheric* will probably be said to be *experimental, electronic*,
- a last one, *shoegaze, pop* will lead to *dream-pop*.

Here we drew what we will call *implications* because we can summarize our sentences by *if a tag is present, then this one is too*, and denote them:

- *jazz, experimental* $\longrightarrow$ *contemporary jazz*,
- *coldwave, rock* $\longrightarrow$ *shoegaze*,
- *atmospheric* $\longrightarrow$ *experimental, electronic*,
- *shoegaze, pop* $\longrightarrow$ *dream-pop*.

In a sense, those implications describe some possible knowledge of our genre set or *attributes* set.

From our point of view, this example is sufficient to say that an *attribute set* is simply a set. We call its elements *attributes* to stick with the literature terminology. If not specified in the context, we will call Σ such a set, and $a,\ b,\ c, \dots$ its elements. Subsets of Σ we will be denoted by capital letters $A,\ B, \dots$. With the example and Σ, we can go over some more definitions

Definition 1 (Implication basis)**.** *Let Σ be an attribute set. An implication basis $\mathcal{L}$ over Σ is a set of implications where implications are pairs $(A,\ B)$, denoted $A \longrightarrow B$, with $A, B \subseteq \Sigma$.*

Usually, given an implication $A \longrightarrow B$, A is called *body* or *premise* while B is called *head* or *consequence*.

Examples Let $\Sigma = \{a,\ b,\ c,\ d,\ e\}$. Some possible implication basis are (for the sake of readability, a subset of Σ we will be written as a concatenation of its elements):

- $\{ab \longrightarrow de,\ a \longrightarrow c, ce \longrightarrow b\}$
- $\emptyset$
- $\{abc \longrightarrow ab,\ d \longrightarrow abcde\}$

Back to the musical example, the implication basis we described is of course { *jazz, experimental* $\longrightarrow$ *contemporary jazz*, *coldwave, rock* $\longrightarrow$ *shoegaze*, *atmospheric* $\longrightarrow$ *experimental, electronic*, *shoegaze, pop* $\longrightarrow$ *dream-pop* }.

Each implication basis describe (possibly different!) knowledge from Σ. Say we have $\mathcal{L}$ over Σ, at least two questions arise:

(i) is $A \subseteq \Sigma$ *"coherent"* with respect to $\mathcal{L}$?

(ii) What can we deduce from $A \subseteq \Sigma$?

Example Consider again musical example. The first question would be "given our basis, can we imagine having a music with some tags only ?". For instance, consider a music with tags *shoegaze, pop*. In our basis, we cannot have a music with these tags *only* because the implication *shoegaze, pop* $\longrightarrow$ *dream-pop* states that if a music has *shoegaze, pop* tags, it *must* also have *dream-pop*. On the other hand, taking only the tag *shoegaze* is possible.

The second question would be "given a set of tags, which tags will we obtain in order to stay coherent with our implications?". Take *coldwave, pop, rock*. Using our implications we will reach two other attributes: *shoegaze, dream-pop*.

In this last paragraph we described some of the *most important* basic notions of our problem: *model* and *closure*.

Definition 2 (Model of an implication). *Let Σ be an attribute set, and $A \longrightarrow B$ an implication over Σ. A subset $M \subseteq \Sigma$ is a* model *of $A \longrightarrow B$, written $M \models A \longrightarrow B$ if $A \nsubseteq M$ or $B \subseteq M$.*

Definition 3 (Model of an implication basis). *Let $\mathcal{L}$ be a basis over Σ. A subset $M \subseteq \Sigma$ is a* model *of $\mathcal{L}$, $M \models \mathcal{L}$, if $M \models A \longrightarrow B$ for each $A \longrightarrow B \in \mathcal{L}$.*

In other words, a subset M of Σ will be a model of an implication if when it contains the body it contains also the head, or the body is not in M. An implication $A \in B$ *follows* from a basis $\mathcal{L}$, denoted $\mathcal{L} \models A \longrightarrow B$ if all models of $\mathcal{L}$ are models of $A \longrightarrow B$. In the example of musics we talked about, *shoegaze* is a model when *shoegaze, pop* is not. For completeness, we will define closure operators and closure systems in general before applying it to our context.

Definition 4 (Closure operator, closure system). *Let Σ be a set, and define $\phi : \Sigma \longrightarrow \Sigma$ an application. ϕ is a* closure operator *if it has the three following properties for all $X, Y \subseteq \Sigma$:*

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

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

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

The pair (Σ, ϕ) is called a closure system.

Definition 5 (Closed set). *Let (Σ, ϕ) be a closure system. A subset X of Σ is a* closed set *(with respect to ϕ) if $\phi(X) = X$. We will denote by Σ_ϕ the set of all closed sets of (Σ, ϕ), that is:*

$$\Sigma_\phi = \{X \subseteq \Sigma \mid \phi(X) = X\}$$

and Σ_ϕ has the following properties:

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

(ii) if $X, Y \in \Sigma_\phi$, so does $X \cap Y$ (Σ_ϕ is closed under intersection).

Note that a closure system can be characterized either by its closure operator, or by its set of closed sets. In other words, we can derive Σ_ϕ from ϕ, as ϕ from Σ_ϕ. The closed set associated to X is the smallest closed set containing X, i.e:

$$\phi(X) = \bigcap\{Y \in \Sigma_\phi \mid X \subseteq Y\}$$

Since Σ_ϕ is closed under intersection, the resultant of the intersection is also a closed set.

Example Let $\Sigma = [\![1 \; ; \; 4]\!]$ and $\phi(X) = X \cup \{4\}$. The pair (Σ, ϕ) is a closure system whose closed sets are all the subsets containing 4. Another interesting definition of Σ_ϕ is:

$$\Sigma_\phi = \{\phi(X) \mid X \subseteq \Sigma\}$$

Then, in our case:

$$\Sigma_\phi = \{\{4\}, \{1,4\}, \{2,4\}, \{3,4\}, \{1,2,4\}, \{1,3,4\}, \{2,3,4\}, \{1,2,3,4\}\}$$

Now we have defined closure systems, let's get back to our implicational context. Let $\mathcal{L}$ be a basis over an attribute set Σ. Let $M \subseteq \Sigma$ and define $\mathcal{L}(M)$ as the smallest model of $\mathcal{L}$ containing M. In this sens, $\mathcal{L}$ defines a *closure system* over Σ for which the closed sets are exactly the models of $\mathcal{L}$. The point is: how to define $\mathcal{L}(M)$ by computations? We rely on [6] for this purpose. Let

$$M^{\mathcal{L}} = M \cup \{B \mid A \longrightarrow B \in \mathcal{L},\ A \subset M \wedge B \nsubseteq M\}$$

Then $\mathcal{L}(M)$ is obtained by repeated application of the operator $M^{\mathcal{L}}$, that is $\mathcal{L}(M) = M^{\mathcal{L}\,\mathcal{L}\ldots\,\mathcal{L}}$ until the computed M is unchanged. For readers with background in SAT-solving or graph theory, this is equivalent to *marking procedure, forward chaining.* In words, if we can find an implication $A \longrightarrow B$ with the body included in M, but not the head B, we add B to M. Let us give an example to set things clear.

Example As usual, let's stick to our musical example. For the recall, our set Σ is: *shoegaze, electronic, coldwave, pop, rock, dream-pop, jazz, experimental, atmospheric, contemporary jazz* and the basis we are working on $\mathcal{L}$ being:

jazz, experimental $\longrightarrow$ *contemporary jazz,*

coldwave, rock $\longrightarrow$ *shoegaze,*

atmospheric $\longrightarrow$ *experimental, electronic,*

shoegaze, pop $\longrightarrow$ *dream-pop.*

In previous examples, we were talking about two subsets of Σ:

- $A =$*coldwave, pop, rock*
- $B =$*shoegaze*

Let us try to compute their closure with respect to $\mathcal{L}$. A is not a model of $\mathcal{L}$ because of the implication *coldwave, rock* $\longrightarrow$ *shoegaze.* Indeed *coldwave, rock* is included in A but not *shoegaze* so we add it, thus $A =$*coldwave, pop, rock, shoegaze.* The new A is still not a model of $\mathcal{L}$, see the implication *shoegaze, pop* $\longrightarrow$ *dream-pop.* We must also add *dream-pop.* Finally, $\mathcal{L}(A) =$*coldwave, pop, rock, shoegaze, dream-pop* will be the smallest model (closed set) containing A.

On the other hand, B does respect all the implications so that it is already a model of $\mathcal{L}$, hence $\mathcal{L}(B) = B$. One interesting point about closure of a subset with respect to an implicational basis can be computed in linear time (in the size of the basis). Of course there exists various algorithms for computing the closure, but since it is not the object of our study we give only the procedure we will use. To have more details on other algorithms to compute the closure of a set with relation to an implicational basis, the reader may refer to [7, 6].

Algorithm 1: LinClosure

```
Input: A basis $\mathcal{L}$, $X \subseteq \Sigma$
Output: The closure $\mathcal{L}(X)$ of $X$ in $\mathcal{L}$

foreach $A \longrightarrow B \in \mathcal{L}$ do
    $count[A \longrightarrow B] := |A|$ ;
    if $|A| = 0$ then
        $X := X \cup B$ ;
    foreach $a \in A$ do
        $list[a] += A \longrightarrow B$ ;
$update := X$ ;
while $update \neq \emptyset$ do
    choose $m \in update$ ;
    $update := update \ \{m\}$ ;
    foreach $A \longrightarrow B \in list[m]$ do
        $count[A \longrightarrow B] -= 1$ ;
        if $count[A \longrightarrow B] = 0$ then
            $add := B \ X$ ;
            $X := X \cup add$ ;
            $update := update \cup add$ ;
return $X(\mathcal{L})$ ;
```

Definition 6 (Closure of an implicational basis)**.** *Given $\mathcal{L}$, the closure $\mathcal{L}^+$ if $\mathcal{L}$ is the of all implications holding in $\mathcal{L}$.*

Definition 7 (Equivalence of implicational basis)**.** *Two implicational basis are equivalent if they have the same closure.*

Also, two basis are equivalent if they have the same models. The question would be: how to determine the closure of an implicational basis? For this purpose, one could use *Armstrong rules* (see [6, 15]).

Before concluding this section, we would like to introduce an useful result as much as some remarks on the use of closure systems and implications. First a proposition (admitted here):

Proposition 1. *Let $\mathcal{L}$ be an implication basis over an attribute set Σ. An implication $A \longrightarrow B$ follows from $\mathcal{L}$ if and only if $B \subseteq \mathcal{L}(A)$.*

Proof. if part. Suppose $B \subseteq \mathcal{L}(A)$. $\mathcal{L}(A)$ is the smallest model of $\mathcal{L}$ containing A. Therefore, for all models M of $\mathcal{L}$ such that $A \subseteq M$ we also have $\mathcal{L}(A) \subseteq \mathcal{L}(M) = M$ (because closed sets are models) and $B \subseteq M$ by extension. Therefore for all models M of $\mathcal{L}$, $A \nsubseteq M$ or $B \subseteq M$ holds. That is $\mathcal{L} \models A \longrightarrow B$.

only if part. Suppose $\mathcal{L} \models A \longrightarrow B$. By definition, all models M of $\mathcal{L}$ satisfy $A \nsubseteq M$ or $B \subseteq M$. In particular, it must hold for the smallest model (inclusion wise) containing A being $\mathcal{L}(A)$ which yields $A \nsubseteq \mathcal{L}(A) \vee B \subseteq \mathcal{L}(A)$. Because this formula holds also for $\mathcal{L}(A)$ (as it is a model) and A $\nsubseteq \mathcal{L}(A)$ being a contradiction with respect to the definition of a closure operator, we conclude that necessarily $B \subseteq \mathcal{L}(A)$ must be true.

□

To conclude this short introduction on closure and implications terminology, we would like to say that these objects model an intuitive way of representing knowledge and inference (see music example). More than this, they are mainly used because they can be checked easily, namely in linear time in the size of our implication basis (see algorithm linclosure, [6, 13, 15]).

In this section we went over general definitions of closure systems, and implication basis. The reader may find a more exhaustive presentation in [6, 12]. Note that these are not all the definitions from closure terminology we shall use. Nevertheless, we prefer to give them when required as they are more closely related to specific problems, while the aim of this section is to be general. The next part is dedicated to a review of propositional logic in accordance to our needs.

1.2 Propositional logic and Horn formulas

This section is dedicated to the introduction of some propositional logic notations and notions. Again, we assume the reader has some background in propositional logic anyway (we are not going to cover the meaning of disjunction, conjunction and so forth). The reader can refer to [11] for an introduction out of our scope.

Before going into definitions, we set up some notations. Let us denote by Σ the set of propositional variables. Disjunction is written with $\vee$, conjunction with $\wedge$, and negation $\neg$. Truth values are 0 (resp. $\bot$) and 1 (resp. $\top$).

Our aim is to build so called Horn formulas. To this purpose we must first introduce what we call clauses, and Horn clauses.

Definition 8 (clause)**.** *Let x_1, dots, x_n, $n \in \mathbb{N}$ be variables of Σ. A clause $\mathcal{C}$ over x_i's is a disjunction of literals p_i:*

$$\mathcal{C} = \bigvee_{i=1}^{n} p_i$$

where $p_i \in \{x_i, \neg x_i\}$.

Definition 9 ((pure) Horn clause)**.** *A clause C over variables of Σ is said Horn (resp. pure Horn) if it contains at most (resp. exactly one) positive literal.*

Example To clarify our idea let us give a simple example. Let $x_1, x_2, \dots, x_n$ be boolean variables. We have the following:

- $\emptyset$ is a clause (true),
- $(x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4)$ is a clause,
- $(\neg x_1 \vee \neg x_2)$ is a Horn clause,
- $(\neg x_1 \vee \neg x_2 \vee x_3)$ is pure Horn.

To begin to draw a link with the previous section, one can note that we can write Horn clauses with disjunction, or with logical implication $\longrightarrow$. Indeed remind that $(x_1 \vee \neg x_2)$ can be equivalently rewritten as $x2 \longrightarrow x_1$. Hence, the Horn clauses defined in the previous examples become:

- $(x_1 \wedge x_2) \longrightarrow \bot$,
- $(x_1 \wedge x_2) \longrightarrow x_3$.

We can use terms body, head the same way as with implicational basis. Before going any further, remind that any boolean formula h can be rewritten in terms of *Conjunctive Normal Form* (or CNF).

Definition 10 ((pure) Horn CNF)**.** *A Horn CNF (resp. pure Horn CNF) $\mathcal{L}$ over Σ is a conjunction of Horn clauses (resp. pure Horn clauses) $\mathcal{C}_i$, $i \in \mathbb{N}$:*

$$\mathcal{L} = \bigwedge_{i} \mathcal{C}_i$$

Definition 11 ((pure) Horn formula)**.** *A boolean function h over Σ is a Horn formula (resp. pure Horn formula) if it can be represented with a Horn CNF $\mathcal{L}$ (resp. pure Horn CNF).*

Here is the link with our implication basis. A Horn CNF $\mathcal{L}$ can be seen as an implicational basis. This relies on some notes:

- for P, Q, R propositional formulas, $(P \longrightarrow Q) \wedge (P \longrightarrow R)$ is equivalent to $P \longrightarrow (Q \wedge R)$,
- representing sets with their *characteristic vectors* (or bitmaps), translating $\cup$ by $\vee$ and $\cap$ by $\wedge$ give almost a one-to-one correspondence between sets and boolean formulas and their models.

Example To clarify let's think of short example taken from the previous section: $\Sigma = \{a, b, c, d, e\}$ and

$$\mathcal{L} = \{ab \longrightarrow de,\ a \longrightarrow c,\ ce \longrightarrow b\}$$

If we think of a, b, c, d, e as boolean variables indicating their presence into a set, we can derive the following translation of $\mathcal{L}$:

$$((a \wedge b) \longrightarrow (d \wedge e)) \wedge (a \longrightarrow c) \wedge ((c \wedge e) \longrightarrow b)$$

which can be represented by a (pure) Horn CNF, say $\mathcal{L}_h$:

$$\begin{aligned} \mathcal{L}_h &= ((a \wedge b) \longrightarrow d) \wedge ((a \wedge b) \longrightarrow e) \wedge (a \longrightarrow c) \wedge ((c \wedge e) \longrightarrow b) \\ &\equiv (d \vee \neg a \vee \neg b) \wedge (e \vee \neg a \vee \neg b) \wedge (c \vee \neg a) \wedge (b \vee \neg e \vee \neg c) \end{aligned}$$

We could also have gone from $\mathcal{L}_h$ to $\mathcal{L}$.

For this reason, we can also represent a Horn CNF as a set of clauses and clauses as pairs (body, head).

Definition 12 (semantic entailment)**.** *A formula Q semantically follows from a formula F, denoted $F \models Q$ if every model of F satisfies Q.*

Since the meaning of semantic entailment is the same as encountered with sets, we can use the $\models$ the same way for both. This concludes our overview of logical needs. We have seen the strong link between sets and logic which will allow us to go from one representation to another without any requirements. The next section is dedicated to a short presentation of hypergraphs.

1.3 Directed graphs and Hypergraphs

Definition 13 (*Hypergraph*)**.** *An hypergraph is a pair $H = (V, E)$ where V is a set of vertices (as in a graph) and E a set of subsets of V describing hyperarcs.*

In fact, hypergraphs are an extension of graphs.

Definition 14 (*Directed Hypergraph*)**.** *A directed hypergraph $H = (V, E)$ is a pair with V a set of vertices (nodes) and E a set of elements of $2^V \times 2^V$, denoting edges going from a subset of V to another subset of V.*

Directed hypergraphs are useful to graphically represent implication basis.

Example Let us consider the following implication basis $\mathcal{L}$:

$$\mathcal{L} = \{1 \longrightarrow 2, 2 \longrightarrow 34, 3 \longrightarrow 12, 41 \longrightarrow 3\}$$

Then we can define an hyper graph $L = (V, E)$ where

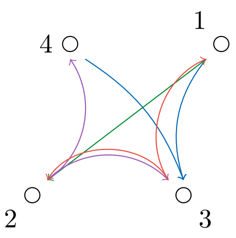


Figure 1.1: Example of Directed Hypergraph

- $V = \{1, 2, 3, 4\}$
- $E = \{(\{1\}, \{2\}), (\{2\}, \{3, 4\}), (\{3\}, \{1, 2\}), (\{1, 4\}, \{3\})\}$

To be clearer, we can see the graphical representation of this figure in 1.1

1.4 Horn Minimization task

Previously we settled down mathematical tools we shall use. In the next paragraphs we will focus on describing "with hands" the problem of Horn minimization in our sense. Here we suppose $\mathcal{L}$ is in so-called *reduced form*, that is for all distinct implications $A \longrightarrow B, C \longrightarrow D$ of $\mathcal{L}$, $A \neq C$. That is we do not have distinct implications with same bodies. This said, we state one of the *most important* of our problem.

Definition 15 (Minimality of $\mathcal{L}$)**.** *Let $\mathcal{L}$ be a set of implications over Σ, and (Σ, ϕ) be a closure system. $\mathcal{L}$ is:*

(i) sound if each implications of $\mathcal{L}$ holds in Σ_ϕ

(ii) complete if each implications holding in Σ_ϕ follows from $\mathcal{L}$

(iii) nonredundant if no implication in $\mathcal{L}$ follows from other implications of $\mathcal{L}$.

An implicational basis with such properties is called minimal.

Note that here, minimality is defined with relation to a closure system. In our case we will just be given Σ and $\mathcal{L}$, without an associated (Σ, ϕ). Therefore, our basis will automatically be sound and complete because the closure system we will consider is the set of models of $\mathcal{L}$. Hence, we can provide a simpler definition of minimality (which is in fact nonredundancy):

Definition 16 (Mimimality)**.** *A reduced implicational basis $\mathcal{L}$ is said minimal if we cannot remove any implication without altering its closure.*

As we shall see later, there exists several definition of minimality. Our point of view could also be called *body minimality*. More intuitively, our aim is to concentrate all the knowledge in a minimal number of implications.

Example Let us recall our music example. We had music styles: *shoegaze, electronic, coldwave, pop, rock, dream-pop, jazz, experimental, atmospheric, contemporary jazz* and some implications:

jazz, experimental $\longrightarrow$ *contemporary jazz,*

coldwave, rock $\longrightarrow$ *shoegaze,*

atmospheric $\longrightarrow$ *experimental, electronic,*

shoegaze, pop $\longrightarrow$ *dream-pop.*

Unfortunately, because all the bodies are disjoint, this basis is already minimal. But let's imagine we have the following implications instead:

coldwave $\longrightarrow$ *rock,*

shoegaze $\longrightarrow$ *rock, coldwave, dream-pop,*

shoegaze, dream-pop $\longrightarrow$ *rock, coldwave,*

rock, dream-pop $\longrightarrow$ *atmospheric, shoegaze, coldwave.*

Here, the third implication can be removed. Indeed, from tags *shoegaze, dreampop* following the implications we can reach *rock, coldwave, atmospheric.* But say we remove the third implication, and thus keep:

coldwave $\longrightarrow$ *rock,*

shoegaze $\longrightarrow$ *rock, coldwave, dream-pop,*

rock, dream-pop $\longrightarrow$ *atmospheric, shoegaze, coldwave.*

What can we reach starting from *shoegaze, dreampop*? Using the second implication we get *rock, coldwave*, and because we have *rock, dreampop*, we also get *atmospheric.* That is, we can get the same knowledge than in the previous case but with one less implication.

Let us present a less practical but more visual example. Suppose $\Sigma = \{a,\ s,\ c,\ r,\ d\}$ and $\mathcal{L}$ as follows:

$$c \longrightarrow r,\ s \longrightarrow rcd,\ rd \longrightarrow asc,\ sd \longrightarrow rc$$

This basis is not minimal. Indeed, let us remove the fourth implication and call $\mathcal{L}^-$ the new basis. We want to know whether the implication we removed still holds in $\mathcal{L}^-$ so that its closure is kept (and therefore its "knowledge"). To show that $\mathcal{L}^- \models sd \longrightarrow rc$, it is enough to show that $rc \subseteq \mathcal{L}^-(sd)$ (see proposition 1). Because we have $s \longrightarrow rcd$, then $rd \longrightarrow asc$ we conclude that $\mathcal{L}^-(sd) = acdrs$. Thus $\mathcal{L}^- \models sd \longrightarrow rc$ and $\mathcal{L}^-$ is smaller than $\mathcal{L}$, with $\mathcal{L}^- \equiv \mathcal{L}$. Also, $\mathcal{L}^-$ is minimal.

Note that the latter example is in fact the same as the musical one. Take the first letter of each style and we end up with the basis described in the second paragraph.

To go in details, we will introduce some elements and objects related to body minimality, namely *Pseudo-closed sets* and *Duquenne-Guigues basis*.

Definition 17 (Pseudo-closed set)**.** *Let (Σ, ϕ) be a closure system. A subset $M \subseteq \Sigma$ is pseudo-closed if and only if*

- *$\phi(M) \neq M$ (M is not closed),*
- *if $Q \subset M$ is pseudo-closed, then $\phi(Q) \subseteq M$.*

Note that the empty set $\emptyset$ is pseudo-closed if and only if it is not closed. Also, one can note that if P is pseudo-closed and P' is covered (inclusion wise, in terms of partial ordering) then P' cannot be pseudo-closed. For the next definition, one can refer to [6, 14].

Definition 18 (Duquenne-Guigues Basis)**.** *Let (Σ, ϕ) be a closure system. The Duquenne-Guigues basis or canonical basis $\mathcal{L}$ is:*

$$\mathcal{L} = \{M \longrightarrow \phi(M) \mid M \subseteq \Sigma,\ M \text{ pseudo-closed}\}$$

and $\mathcal{L}$ is complete, sound and nonredundant.

Thus, the Duquenne-Guigues basis is body-minimal. In this section we presented the minimization problem we will worked on during this thesis. Another remark, as we shall see in the next chapter, this problem is solvable in polynomial time.

1.2em

To conclude this first chapter, we introduced in general our problem of minimizing implicational basis as our way of working. We developed an overview of the mathematical tools we needed to

Chapter 2

Review of existing algorithms

In the first chapter we introduced the aim of our study, our working procedure and the main tools for us to understand the next explanations. In this part, we will focus on reviewing the existing algorithms for the Horn minimization task.

2.1 Minimization by reduction

2.1.1 Obiedkov and Ganter algorithm: use of saturation

Compute the canonical (or Duquenne-Guigues) basis $\mathcal{L}_c$ given $\mathcal{L}$. [14, 6]. This algorithm performs so called operations of *left-saturation*, *right-saturation* and *redundancy elimination*.

Algorithm 2: Minimal Cover

```
Input: $\mathcal{L}$ an implication basis
Output: $\mathcal{L}_c$ the canonical basis derived from $\mathcal{L}$

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 $A \longrightarrow B$ do
    $\mathcal{L} = \mathcal{L} - \{A \longrightarrow B\}$ ;
    $A = \mathcal{L}(A)$ ;
    if $A \neq B$ then
        $\mathcal{L} = \mathcal{L} \cup \{A \longrightarrow B\}$ ;
```

The first loop maximizes each heads, that is $A \longrightarrow B$ becomes $A \longrightarrow \mathcal{L}(A)$. We summarize the knowledge. The second loop maximizes left side of implications with $\mathcal{L}^{-}(A)$. The second step kills redundancy.

Notes: few tips to see that the resulting basis is equivalent to the input. By the end of the first loop, since we replaced $A \longrightarrow B$ by $A \longrightarrow \mathcal{L}(A)$, and by definition of $A \longrightarrow B$, $A \longrightarrow B$ still holds for all B ($B \subseteq \mathcal{L}(A)$). For the second loop, observe that we omit only redundant implications from the basis, because if $\mathcal{L}^{-}(A) = \mathcal{L}(A)$ it means that all the implications $A \longrightarrow B$ true in $\mathcal{L}$ are true in $\mathcal{L}^{-}$.

Complexity the two loops have the same complexity: $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$ because we run LinClosure for all implications of $\mathcal{L}$. Thus, the complexity of the algorithm is $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$.

2.1.2 Maier Algorithm: using equivalence classes

This part relies mainly on [13], [15] in which an algorithm for minimizing set of functional dependencies (FD). Because FDs behave as implications in implicational basis, we will adopt the latter terminology.

Theoretical set up and definitions

We are going to work in the FD framework, that is first notations we defined. Before diving into the algorithm, we may need some definitions. Indeed, the minimization procedure in this case relies on a criterion for minimality using specific objects.

Definition 19 (Equivalence classes)**.** *Let $X \subseteq \Sigma$. The set*

$$E_{\mathcal{L}}(X) = \{A \longrightarrow B | A \longrightarrow B \in \mathcal{L}, A \equiv X\}$$

is the equivalence class of X under $\mathcal{L}$. In fact, it is the set of implications of $\mathcal{L}$ with body equivalent to X. The set of all non-empty such classes is denoted $\bar{E}_{\mathcal{L}}$.

One should note in order to get $\bar{E}_{\mathcal{L}}$, there is no need to check all the subsets of Σ. Because $\bar{E}_{\mathcal{L}}$ defines a partition of $\mathcal{L}$ based on implication bodies, it is sufficient to go over implications of $\mathcal{L}$.

The second tool used by Maier in his algorithm is *direct determination*. This notion is extensively discussed in [13, 15]. Thus, the definition we are going to define is more the result of a very useful necessary and sufficient condition than the definition as given at the beginning. Our aim is not to (re)build all the material, but to explain the algorithm.

Definition 20 (Direct Determination)**.** *Given a basis $\mathcal{L}$, we say that A directly determines B under $\mathcal{L}$, denoted $A \xrightarrow{\text{d}} B$, if $\mathcal{L} - \bar{E}_{\mathcal{L}} \models A \longrightarrow B$.*

More intuitively, we say that A directly determines B if we can reach B without using any implications with body equivalent to A (including A). Those definitions are sufficient to understand the algorithm.

Algorithm

In this section we will investigate the Maier Algorithm to minimize a basis $\mathcal{L}$. The principle is short enough to describe it as follows:

1. remove redundant implications

2. remove direct determinations in this sens: if $A \xrightarrow{\text{d}} B$ with $A \longrightarrow C$ and $B \longrightarrow D$ implications of $\mathcal{L}$, remove $A \longrightarrow C$ and replace by $B \longrightarrow D$ by $B \longrightarrow D \cup C$.

Of course, proof of correctness has been given in cite [13, 15]. Nevertheless, we shall give later some different elements for proving the algorithm ends with the right result. We do not give them here, because those elements also provide a proof for an another algorithm (namely Ausiello and al. algorithm) through equivalence properties. For now, we send the reader to cited papers. The procedure is presented in algorithm 3.

Algorithm 3: Maier minimization algorithm

```
Input:
Output:

// redundancy elimination
foreach $A \longrightarrow B \in \mathcal{L}$ do
    $\mathcal{L}^- := \mathcal{L} - \{A \longrightarrow B\}$ ;
    if $\mathcal{L}^-(A) = \mathcal{L}(A)$ then
        $\mathcal{L} := \mathcal{L} - \{A \longrightarrow B\}$ ;
$\bar{E}_{\mathcal{L}} := EquivClasses(\mathcal{L})$ ;
// direct determination elimination
foreach $E_{\mathcal{L}} \in \bar{E}_{\mathcal{L}}$ do
    foreach $A \longrightarrow C \in E_{\mathcal{L}}$ do
        if $\exists B \longrightarrow D \in E_{\mathcal{L}}$ such that $A \xrightarrow{\text{d}} B$ then
            remove $A \longrightarrow B$ from $\mathcal{L}$ ;
            replace $B \longrightarrow D$ by $B \longrightarrow C \cup D$ ;
```

Observations There are some observations and questions to answer about procedure 3:

- How do we compute $\bar{E}_{\mathcal{L}}$?
- How do we check for direct determination?
- Does the operation of remove and replace alter an equivalence class more than removing one of its implications?

Let us take those questions in order. To find $\bar{E}_{\mathcal{L}}$, we will consider applying a variation of LinClosure to each implication of $\mathcal{L}$. Thus for each $A \longrightarrow B$, instead of returning closure of A under $\mathcal{L}$, we will a bit vector of size $|\mathcal{B}(\mathcal{L})|$ with the i-th entry being 1 if the body of this implication is in the closure of A, and 0 otherwise. the i-th bit of the vector is set if $count[i\text{-th implication}] = 0$ in the procedure. Running this modified version of LinClosure over all implications produces a $|\mathcal{B}(\mathcal{L})| \times |\mathcal{B}(\mathcal{L})|$ matrix such that the entry

$[A \longrightarrow C, B \longrightarrow D]$ is 1 if $\mathcal{L} \models A \longrightarrow B$. Consequently, finding equivalence classes shall take no more than a run over this matrix to be done (after having computed closures).

Direct determination can be considered as well with a modification of LinClosure on $\mathcal{L} - E_{\mathcal{L}}(A)$. In this version, for a given $A \longrightarrow C$, we also have to be provided with $E_{\mathcal{L}}(A)$. Then, direct determination will be exposed the first time we reach $\mathit{count}[B \longrightarrow D] = 0$ in the computation, for $B \longrightarrow D$ different from $A \longrightarrow C$ and belonging to $E_{\mathcal{L}}(A)$. Because we stop to the first time this will ensure that we did not reach any other body of this equivalence class, otherwise we would have stopped before. Because equivalence classes partition the implications of $\mathcal{L}$ we will proceed to this operation at most once for each implication.

Finally, we consider the question of whether equivalence classes are altered more than removal during computations. Of course we remove direct determination is found. Does replacing $B \longrightarrow D$ by $B \longrightarrow C \cup D$ has any chances to add other implication to $E_{\mathcal{L}}(A)$? Fortunately the answer is no. Indeed, before removing $A \longrightarrow C$ from $E_{\mathcal{L}}(A)$, anything we could reach from A could be reached from B (because of their equivalence) and thus would have implied C from A and D from B. Thus anything reachable from $C \cup D$ would already be in $E_{\mathcal{L}}(A)$.

Complexity Again, the complexity of this algorithm has been studied and discussed in [13, 15]. It has been challenged (somewhat) in [3, 4]. We provide a complexity analysis anyway to adapt previous material to our notations (and ease comparison with other algorithms) and to have this report to be self-contained as much as possible. Let us recall few elements to analyse complexity:

- $\mathcal{B}(\mathcal{L})$ is the number of distinct bodies in $\mathcal{L}$ (therefore the total number of implications for us), and $|\mathcal{B}(\mathcal{L})|$ its size,
- $|\Sigma|$ is the number of attributes in the universe Σ,
- $|\mathcal{L}|$ is the size of $\mathcal{L}$ in memory, that is roughly $|\mathcal{B}(\mathcal{L})| \times |\Sigma|$.

Let us proceed by steps. First, non-redundancy elimination. Because the inner **foreach** loop goes for all implications of $\mathcal{L}$ requiring $O(|\mathcal{B}(\mathcal{L})|)$ times looping. Then, computing closure of A under $\mathcal{L}$ and $\mathcal{L}^{-}$ requires $O(|\mathcal{L}|)$ operations thanks to LinClosure. Hence, redundancy elimination can be done in $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|) = O(|\mathcal{B}(\mathcal{L})|^2 \times |\Sigma|)$ time. Secondly, finding equivalence classes as mentioned previously can be done by using modified version of LinClosure performing same time as the original one. Since we have to go over (at most) all implications to find all implications with bodies implied by a given one, this result in $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$ complexity. Then, having the $|\mathcal{B}(\mathcal{L})| \times |\mathcal{B}(\mathcal{L})|$ matrix, building equivalence classes need a run over the matrix, that is $O(|\mathcal{B}(\mathcal{L})|^2)$ operations. Finally, removing direct determination can be done at most for all implications, and checking for this property is again made through LinClosure. Thus performing the last step of the algorithm is $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|) = O(|\mathcal{B}(\mathcal{L})|^2 \times |\Sigma|)$ time. As a conclusion, the whole algorithm works in $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$.

2.1.3 Ausiello Algorithm: minimality through directed graphs

This section is dedicated to a minimization algorithm relying on hypergraphs. It has been set up by Ausiello and al. in [5, 3, 4]. Starting from a directed hypergraph representation, it builds a special kind of DAG, 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 [4] and then go into further details and more precise algorithms for such computations as exposed in [3].

FD-Graphs and minimum covers

In this part, we assume that our basis $\mathcal{L}$ is represented through the framework of hypergraphs (see chapter 1). Again, all the definitions we are going to state here are exposed in [3, 4]. First, let us define the central object we will work on, FD-Graphs.

Definition 21 (FD-Graph)**.** *Given a hypergraph $\mathcal{L} = (\Sigma, C)$ representing an implication basis, 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 $\mathcal{L}$.

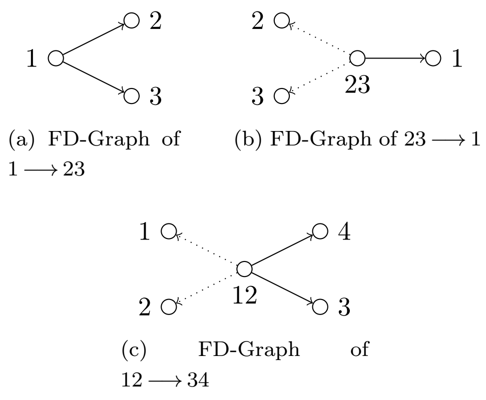
In fact, hypergraph representation is not mandatory to represent a FD-Graph, since as we saw, hypergraphs in our context rely on "usual" implicational basis. To enlighten the definition of FD-Graph, simple examples are shown in 2.1.

We can see in those example that drawing an FD-Graph goes this way:

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

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

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



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

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

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

Figure 2.1: Representation of some FD-graph

An important notion we may need for all the following is *FD-paths*.

Definition 22 (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 $k = 1, \ldots, n$.*

As is, the definition may not seems clear. Informally, a FD-path from a node i to j describes the implications we use to derive $i \longrightarrow j$ (with Armstrong rules, especially transitivity and union). 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 2.3 based on the basis described in figure 2.2.

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 and al.

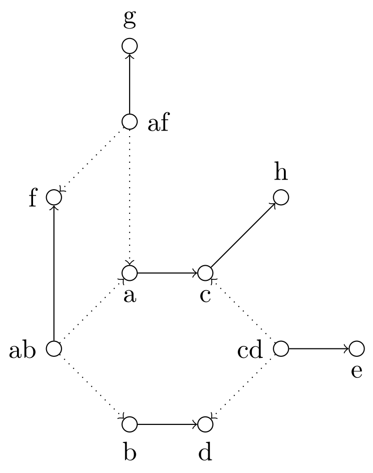


Figure 2.2: FD-Graph of some implicational basis

Ausiello algorithm: general point of view

The following procedure (4) finds from a given basis its *minimal cover*.

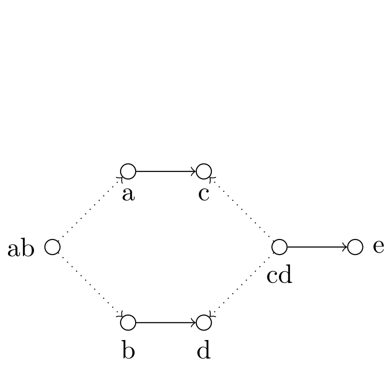
```
Algorithm 4: Ausiello algorithm (1986)
    Input: $\mathcal{L}$ an implication basis
    Output: $\mathcal{L}_c$ a minimal cover for $\mathcal{L}$

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

Let us emphasize on the meaning of those removal steps. The algorithm studied in the sources, aims to minimize an implicational basis in terms of bodies. As with the Duquenne-Guigues basis. It uses 3 steps. Two to remove redundancy, the third one aims to lighten bodies and heads of remaining implications. We will try to express each of these steps in terms of sets, closure, and so forth.

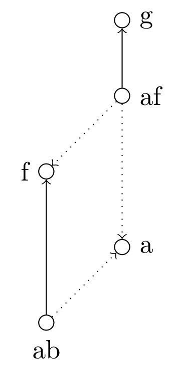
Removing redundant nodes

The first step is about removing redundant implications (without right maximization). In terms of FD-graphs, we remove *redundant* nodes. A compound node (only) i is said redundant if for each full arc ij leaving i there exists a dotted path (i, j). We give an example in the figure 2.4.

In this example, the basis 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. In details, let $A \longrightarrow B$ be an implication of $\mathcal{L}$ with $A = a_1 a_2 \dots a_n$ and $B = b_1 b_2 \dots b_m$. $A \longrightarrow B$ will be redundant in terms of FD-Graph if for each b_i there exists $X_i \subset A$

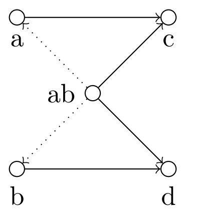


(a) dotted FD-path for $ab \longrightarrow e$

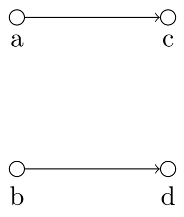


(b) full FD-path for $ab \longrightarrow g$

Figure 2.3: Representation of some FD-paths



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



(b) FD-Graph with redundant node removed

Figure 2.4: Elimination of redundant nodes

such that $X_i \longrightarrow b_i$. That is:

$$\bigcup_i X_i \subseteq A \longrightarrow B$$

Which may be rewritten in terms of $\mathcal{L}$ closure as

$$\bigcup_i \mathcal{L}(X_i) = \mathcal{L}(A)$$

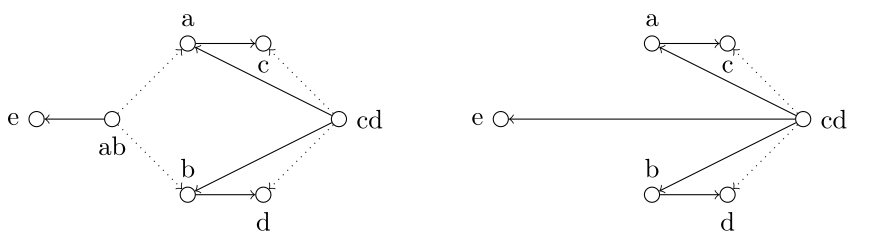
Thinking of opposite direction, $A \longrightarrow B$ will be nonredundant if there exists $b \in B$ such that $((b \in \mathcal{L}(X)) \land (X \subseteq A)) \longrightarrow (X = A)$.

To sum up: *the first step is about considering each $A \longrightarrow B$ where $|A| > 1$, and removing it of $\mathcal{L}$ if $\mathcal{L}^{-}(A) = \mathcal{L}(A)$*. Note: $\mathcal{L}^{-} = \mathcal{L} - A \longrightarrow B$.

Removing superfluous nodes

Next, we remove from the nonredundant FD-Graph *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 paths (i, j) and (j, i).

In terms of sets and closure, two attribute sets $A, B \subseteq \Sigma$ are equivalent in $\mathcal{L}$ if $\mathcal{L} \models A \longrightarrow B, B \longrightarrow A$, that is, if $\mathcal{L}(A) = \mathcal{L}(B)$.



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

Figure 2.5: Elimination of superfluous node

The algorithm suggests the following:

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

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

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

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

In this case, we apply the following operations

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

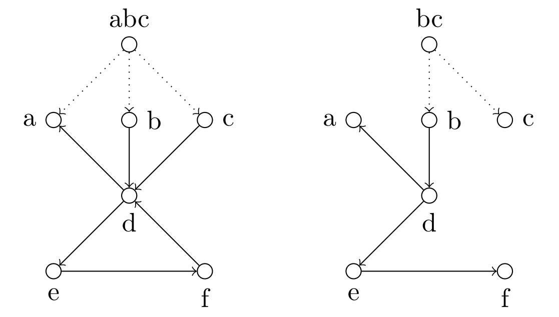
We exhibit some arguments to convince ourselves that this is a valid operation. Let us call temporarily $\mathcal{L}^-$ the basis we obtain after the previous operations. We would like to show that $\mathcal{L}^- \models A \longrightarrow B$, i.e that $\mathcal{L}^- \equiv \mathcal{L}$. We removed $A \longrightarrow B$ but we still have $\mathcal{L}^- \models X \longrightarrow C$ and then $X \longrightarrow B$ because $C \longrightarrow D \cup B$. That is, $B \subset \mathcal{L}^-(X)$. Because $X \subset A$, we have $B \subset \mathcal{L}^-(A)$ which is what we wanted.

Removing redundant arcs

Finally, we remove from a minimum nonredundant FD-Graph, *redundant arcs*:

- dotted case: a dotted arc ij is redundant if there is a dotted path (i, j) not using ij,
- full case: a full arc ij is redundant if there is a dotted/full path (i, j) not using ij.

we can think of eliminating redundant arcs as explicit transitivity elimination. If we remove a full arc in $A \longrightarrow B$ then we are minimizing B. If we remove a dotted arc, we are minimizing A. We have examples in 2.6.



(a) FD-Graph with redundant arcs ($abc \longrightarrow a$, $f \longrightarrow d$)

(b) FD-Graph with redundant arcs removed

Figure 2.6: Elimination of redundant arcs

In terms of sets, consider an implication $A \longrightarrow B$. Removing a dotted arc is saying that given $a \in A$, and substituting $A \longrightarrow B$ by $A - a \longrightarrow B$ in $\mathcal{L}$ preserves $\mathcal{L} \models A \longrightarrow a_i$. On the other side, removing a full arc is, given $b \in B$ and replacing by $A \longrightarrow B - b$, we preserve $\mathcal{L} \models A \longrightarrow b$. It appears that this steps goes beyond our scope, because it aims to minimize implication itself, not the number of implications. So, in our minimization context, we can stick to the first two operations to obtain a minimal representation in our terms. The algorithm we have to follow becomes the procedure 5

Algorithm 5: Ausiello algorithm (1986, reduced)

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

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

In fact, this algorithm stands for a high-level principle of what has to be done to minimize a given basis. The question we are going to investigate in the next paragraphs is how to do such computations.

Ausiello algorithm: effective procedure

In this part we will focus on how the algorithm proposed by Ausiello in [4] performs redundancy and superfluousness elimination. Actually, those operations are detailed in [3]. But in order to stick to our subject (reviewing the existing algorithms) we study it here. Also, note that we will consider that the mapping step from $\mathcal{L}$ to $G_{\mathcal{L}}$ is already done. Also, as mentioned in the related articles, going from one representation to another can be done in linear time.

The algorithm relies on closure of a FD-Graph. Of course the idea of closure is similar to the one we encountered before. Nevertheless, Ausiello and al. introduce a notion of priority in the way they determine the closure. Because we cannot represent to different arcs with same nodes (two arcs $(i,\ j)$), me must provide priority between dotted and full arcs. The authors decided to stress on dotted arcs since they are more likely to be a criterion for removal during minimization.

Closure of a 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 $\{j \in V \mid (j,\ i) \text{ is a dotted arc}\}$,
- L_i^0 $(\forall i \in V)$: nodes from *outgoing full* arcs $\{j \in V \mid (i,\ j) \text{ is a full arc}\}$,
- L_i^1 $(\forall i \in V)$: nodes from *outgoing dotted* arcs $\{j \in V \mid (i,\ j) \text{ is a dotted arc}\}$,
- 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 6, 7.

Algorithm 6: NodeClosure (Principle)

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

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

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

Algorithm 7: NodeClosure

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

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

Next, we present the principle and pseudo-code for the closure of an FD-graph 8, 9. 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 (6, 7). We initialize it in case i is indeed part of some compound node so that we do not omit to count it when dealing with S_i (because S_i does not contain i). In terms of complexity, we are running NodeClosure on all nodes having outgoing edges, that is $|\mathcal{B}(\mathcal{L})|$ nodes (if a compound node is represented, it must have at least one outgoing full arc). Since NodeClosure operates in $O(|\mathcal{L}|)$, the whole closure algorithm must run in $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$.

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

Minimization algorithm for FD-Graphs We have two steps in this algorithm. First, we need to remove redundant nodes, then superfluous nodes. To delete redundant nodes, the claim of [3, 4] is that removing nodes with dotted arcs only in the closure of an FD-Graph is sufficient. Indeed, if a node i has only dotted paths in the closure of an FD-graph, it means that for every $i \longrightarrow j$ holding, we can find a subset of k of i such that $k \longrightarrow j$. In our terms, it is saying that $\mathcal{L}(i) = \mathcal{L}^-$ where $\mathcal{L}^-$ denotes $\mathcal{L}$ from which we removed the implication having i as a body.

Let us try to investigate the complexity of this algorithm. According to the article and previous study, Closure is achieved in $O(|\mathcal{B}(\mathcal{L})| \times |\mathcal{L}|)$. If the graph is represented as adjacency lists, deleting a node

Algorithm 8: Closure (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 9: Closure

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

Algorithm 10: Ausiello Minimization Algorithm

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

$G_{\mathcal{L}}^{+} = Closure(G_{\mathcal{L}})$ ;
// Redundancy Elimination
forall $i \in V^1$ do
    if $L_i^{1+} = \emptyset$ then
        remove $i$ and its outgoing arcs from $G_{\mathcal{L}}$ ;
// Superfluousness Elimination
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 ;
```

and its outgoing arc can be $O(1)$ (it consists in freeing the node and its associated two lists). Running over all compound nodes is $O(|\mathcal{B}(\mathcal{L})|)$. In the case we would have to remove arcs one by one, note that we would have at most $O(|\,\Sigma\,|)$ arcs to delete. This would yield a $O(|\mathcal{B}(\mathcal{L})| \cdot |\,\Sigma\,|) = O(|\,\mathcal{L}\,|)$ complexity. *How to remove compound nodes that have disappeared from the closure of other nodes ? It is time consuming*. It is $O(|\mathcal{B}(\mathcal{L})| \cdot |\,\mathcal{L}^+\,|)$. We don't count for intern data structure operation. We can consider that adding and removing can be done in $O(1)$. The first loop of superfluousness elimination goes over all compounds nodes, and finding an equivalent node is $O(V)$ Moving elements in lists associated to j is also $O(V)$ thus the first loop is $O(V^1 \cdot V)$. The second loop may need to go over all the full arcs of the FD-graph, which account for a $O(|\,\mathcal{L}\,|)$ complexity. As a consequence the whole algorithm may run in $O(|\mathcal{B}(\mathcal{L})| \cdot |\,\mathcal{L}\,|)$ (because of the closure computation).

The remaining part of this section will be devoted to propose ways to implement efficiently the principles we just presented. By "implementing efficiently" we mean to give algorithms, using adjacency list structure, to perform the operations with required amount of time. In our terms, time complexity for finding and removing superfluousness must not exceed $O(|B|^2 + |\,\mathcal{L}\,|)$. We divide those computations in 2 parts:

1. removing superfluous nodes within the closure (see algorithm 11)
2. removing superfluous nodes in the initial graph (see algorithm 12)

Let us focus first on superfluousness elimination within the closure of an FD-Graph. Let us consider G^+ to be the closure of some FD-graph G. First step is to find superfluous nodes. For the recall, this property can hold only for compound bodies of the implication system G models. At first sight, one could think of finding superfluousness as follows:

```
foreach $i \in V_c$ do
    foreach $j \in L_i^{1+}$ do
        foreach $k \in L_j^{0+} \cup L_j^{0+}$ do
            if $k = i$ then
                $i$ superflous w.r.t $j$ ;
```

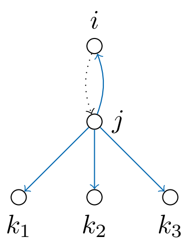


Figure 2.7: Finding superfluous nodes in FD-Graphs

The issue being the required amount of time to perform such computations. Indeed, Such nested loops will account for $O(|V_c||V|^2) = O(|B|(|B| + |\Sigma|)^2) = O(|B|^3 + |B||\mathcal{L}| + |\mathcal{L}||\Sigma|)$ operations.

Algorithm 11: SuperfluousnessClosureElimination

```
Input: $G^+_{\mathcal{L}}$: the closure of some FD-Graph $G_{\mathcal{L}}$
Output: $L$: list of pair of nodes to be altered in the initial graph

$L = \emptyset$;
foreach $i \in V_c$ do
    foreach $j \in V$ do
        $color(j) := 0$ ;
    foreach $j \in L_i^{1+}$ $s.t$ $color(j) = 0$ do
        foreach $k \in L_j^{0+} \cup L_j^{1+}$ $s.t$ $color(k) = 0$ do
            $color(k) := 1$ ;
            if $k = i$ then
                $i$ superfluous ;
                keep $j$ ;
    if $i$ superfluous (w.r.t $j$) then
        foreach $k \in L_i^{0+}$ do
            $color(k) := 2$ ;
        foreach $k \in L_j^{1+}$ do
            if $color(k) = 2$ then
                move $k$ to $L_j^{0+}$ ;
        $L_i^{0+} = \emptyset$ ;
        $L_i^{1+} = \emptyset$ ;
        add $(i, j)$ to $L$ ;
foreach $i \in V$ $s.t$ $i$ not superfluous do
    foreach $j \in L_i^{0+} \cup L_i^{1+}$ do
        if $j$ superfluous then
            remove $j$ from $L_i^{0+} \cup L_i^{1+}$ ;
```

2.1.4 Elements of proofs

This section needs to be adjusted and corrected. In fact, statements lacks some precision, but conclusion is unchanged: both algorithm (Maier and Ausiello) performs computations on same objects, but according to the following statement

Proposition 2. *We have equivalence between the following:*

Algorithm 12: SuperfluousnessElimination

```
Input: $L$: a list of pair of vertices $(i, j)$, $i$ superfluous w.r.t $j$

$dest$ array of size $|V|$;
foreach $v \in V$ do
    $dest[v] := v$ ;
    $color(v) := 0$ ;

foreach $(i, j) \in L$ in reverse order do
    $dest[i] = dest[j]$ ;

foreach $i \in V$, $i$ superfluous do
    foreach $j \in L^0_{dest[i]} \cup L^1_{dest[j]}$ do
        $color(j) := 1$ ;
    foreach $j \in L^0_i$, s.t $color(j) = 0$ do
        move $j$ to $L^0_{dest[i]}$ ;
    Reset all colors to 0 ;

foreach $i \in V$, $i$ is simple do
    foreach $j \in D_i$, $j$ superfluous do
        remove $j$ from $D_i$ ;

Remove superfluous nodes from $V$ ;
```

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

(ii) A *is superfluous (in particular, "minimal" superfluous with respect to* B*),*

(iii) third part of proposition 7. (to be adapted)

In this section we write various claims which help us prove correctness and equivalence of Maier/Ausiello's algorithms. Knowledge about FD-graphs is assumed. Equivalence of redundancy elimination is not shown (1st step of the algorithm). We place ourselves in a non-redundant context to match definitions of direct determination. Non-redundancy does not affect superfluous definition.

Proposition 3. *The following properties are equivalent (let* $A \longrightarrow C$ *be the implication of* $\mathcal{L}$ *with* A *as body):*

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

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

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

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

□

Proposition 4. *The following statements are equivalent:*

(i) $A \xrightarrow{\text{d}} B$*,*

(ii) $\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow B$*.*

Proof. Translation of notions. $A \xrightarrow{\text{d}} B$ means that if we remove all implications with left sides equivalent to A we can still derive $A \longrightarrow B$ using remaining implication, that is $\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow B$. □

Proposition 5. *the following statements are equivalent, for* A, B *bodies of* $\mathcal{L}$*:*

(i) $A \xrightarrow{\text{d}} B$ *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 propositions 4, 3, 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 $\mathcal{L} - E_{\mathcal{L}}(A) \models A \longrightarrow B$.

□

Proposition 6. *The following two statement are equivalent, given the FD-graph of $\mathcal{L}$:*

(i) A is a superfluous node,

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

Proof. (ii) $\longrightarrow$ (i) is trivial. let's focus on (i) $\longrightarrow$ (ii). If A is superfluous, then there exists B such that $A \equiv B$ and there has a dotted path from A to B.

If $B \subseteq A$, the dotted path is straight forward. If it is not the case, path from A to B uses outgoing full arcs from nodes equivalent to A, or it does not. If it does not we are done. Now Suppose this path uses an outgoing full arc from a node C equivalent to A. This means, that $A \equiv B \equiv C$ by definition and therefore, there exists a dotted path from A to C (because we need to reach C, so to derive it, to use its outgoing arcs). We can reiterate this reasoning until reaching A. Getting to A or one of its subset would contradict our assumptions meaning that we stopped finding used equivalent arcs earlier.

□

From the previous claims, we can yield the following one

Proposition 7. *The following statements are equivalent:*

(i) there exists $B \equiv A$ such that $A \xrightarrow{\text{d}} B$,

(ii) A is superfluous with respect to B,

(iii) there exists C such that $A \equiv C$ and $\mathcal{L} - \{A \longrightarrow D\} \models A \longrightarrow C$. (possibly, $C = B$).

Proof. (i) $\longleftrightarrow$ (ii) comes from propositions 5 and 6. (ii) $\longleftrightarrow$ (iii) comes from proposition 3.

□

Those claims help to see the relation between operations of the algorithms. Indeed, the last claim states that finding a direct determination is the same as finding a superfluous node (under equivalence constraint), therefore the two algorithms are looking for the same structures in different terminologies. This emphasizes the fact they work on the same computations. Remark: Maier algorithm do it in a special order (some kinds of minimal paths in terms of FD-graph. Can we remove them in any order? It seems to since this is what the Ausiello algorithm does, and the previous proposition states that direct determination is equivalent to the existence of superfluous nodes.

The next claim provides an argument in this way. It also explains in what extent the operation of removal in those algorithm is exact.

Proposition 8. *Let $A \longrightarrow C, B \longrightarrow D$ be implications of $\mathcal{L}$. If $A \equiv B$ and $\mathcal{L} - \{A \longrightarrow C\} \models A \longrightarrow B$, then $\mathcal{L}$ is not minimum.*

Proof. Let $\mathcal{L}$ be an implication basis with implications $A \longrightarrow C, B \longrightarrow D$, $A \equiv B$, and such that $\mathcal{L} - \{A \longrightarrow C\} \models A \longrightarrow B$. Let us build $\mathcal{L}_c$ by removing $A \longrightarrow C$ from $\mathcal{L}$, and replacing $B \longrightarrow D$ by $B \longrightarrow C \cup D$. Obviously, $\mathcal{L}_c \models A \longrightarrow B$. Moreover, $A \longrightarrow C$ still holds since $A \longrightarrow B \longrightarrow C \cup D$. □

This proposition can be stated as its contraposition. That is, if a basis is minimum then we cannot find equivalent bodies with one to be removed. This states that the second step of Ausiello and Maier algorithms end up with a body minimal basis, with help of equivalences proved in proposition 7.

2.2 Minimization by construction

2.2.1 An algorithm relying on a bounding theorem

In this section we are going to expose an algorithm using the so-called Boros theorem (see [10, 9, 8]). Depending on our needs we will state some definitions and some propositions of the sources cited. The algorithm exposed in this part can be found in [10].

Theoretical setting

Again, all the terms we are going to define here can be found in [10, 9, 8]. The articles from Boros use the logical terminology. The context is pure Horn functions, and by Horn function, Horn CNF, we will mean pure.

Definition 23 (Implicates)**.** *Let $\mathcal{L}$ be a Horn CNF. A clause $\mathcal{C}$ is an implicate of $\mathcal{L}$ if $\mathcal{L} \models \mathcal{C}$. $\mathcal{I}(\mathcal{L})$ is the set of implicates of $\mathcal{L}$.*

In fact, $\mathcal{I}(\mathcal{L})$ can be seen as the closure of $\mathcal{L}$, the set of all implications following from $\mathcal{L}$.

Definition 24 (Essential sets)**.** *Let $\mathcal{L}$ be a Horn CNF, and $X \subseteq \Sigma$. The essential set associated to X, $\mathcal{E}_X$ is the set of clauses of $\mathcal{I}(\mathcal{L})$ for which X is a false point:*

$$\mathcal{E}_X = \{\mathcal{C} \in \mathcal{I}(\mathcal{L}) \mid \mathcal{B}(\mathcal{C}) \subseteq X, \mathcal{H}(\mathcal{C}) \notin X\}$$

A few remark on this definition. It could be extended in terms of implicational basis with the following:

$$\mathcal{E}_{\mathcal{X}} = \{\mathcal{A} \longrightarrow \mathcal{B} | \mathcal{A} \subseteq \mathcal{X}, \mathcal{B} \nsubseteq \mathcal{X}\}$$

with a detail to write down anyway. Remind that a clause $\mathcal{C}$ can be rewritten as $A \longrightarrow b$. If $A \longrightarrow b_1$ and $A \longrightarrow b_2$ are elements of $\mathcal{E}_{\mathcal{X}}$ then obviously $A \longrightarrow (b_1 \wedge b_2)$ will be. On the other side, $A \longrightarrow B$ being part of $\mathcal{E}_{\mathcal{X}}$ implies that there is *at least* one $b \in B$ such that $A \longrightarrow b$ is in $\mathcal{E}_{\mathcal{X}}$ also. It does not necessarily concern all the elements of B. Actually, this detail does not cause any problem for the next discussion because existence of such $b \in B$ is sufficient.

state MinMax Theorem.

Algorithm

Pseudo-code and principle The algorithm relies on hypergraph representation. The principle is to construct a minimal basis from a pure Horn CNF, by adding successivly implications with pairwise body-disjoint essential sets. Thanks to the theorem of Boros, this can be done only a minimal number of times, leading to body minimal representation. The pseudo-code is

Algorithm 13: BodyMinimal (Hypergraphs)

```
Input: $\mathcal{L}$ an implicational basis
Output: $\mathcal{L}_c$ a body-minimal representation of $\mathcal{L}$
```

Actually, the graph representation is not mandatory. Indeed the authors use hypergraphs because of a forward chaining procedure in linear time, and also due to their proofs relying on graphs. We could rewrite this algorithm as in 14

Algorithm 14: BodyMinimal)

```
Input: $\mathcal{L}$ an implicational basis
Output: $\mathcal{L}_c$ a body-minimal representation of $\mathcal{L}$

$\mathcal{L}_c := \emptyset$ ;
while $\exists X \in \mathcal{B}(\mathcal{L}) : \mathcal{L}(X) \neq \mathcal{L}_c(X)$ do
    $A := \min(\mathcal{L}_c(X) : X \in \mathcal{B}(\mathcal{L}) \wedge \mathcal{L}(X) \neq \mathcal{L}_c(X))$ ;
    $B := \mathcal{L}(X)$ ;
    $\mathcal{L}_c := \mathcal{L}_c \cup \{A \longrightarrow B - A\}$ ;
return $\mathcal{L}_c$ ;
```

Elements of proof of correctness

Proposition 9. *Let P_1, P_2 be pseudo-closed with respect to $\mathcal{L}$ over Σ. If their intersection is a false point of $\mathcal{L}$, then it is pseudo-closed.*

Proof. Let P_1, P_2 be two pseudo-closed sets of $\mathcal{L}$ over Σ. If $P_1 \subset P_2$ (or vice-versa) the result is straightforward. Let consider then $P_1 \nsubseteq P_2$, $P_2 \nsubseteq P_1$. If $P_1 \cap P_2 = \emptyset$, $\emptyset$ being a false point of $\mathcal{L}$, by definition $\emptyset$ will be pseudo-closed.

Suppose $P_1 \cap P_2 \neq \emptyset$ is a false point of $\mathcal{L}$. Then it must exist $A \subset P_1 \cap P_2$ pseudo-closed. Thus

- $A \subset P_1 \cap P_2 \longrightarrow A \subset P_1$
- $A \subset P_1 \cap P_2 \longrightarrow A \subset P_2$

Because A, P_1, P_2 are pseudo-closed we have

- $A \subset P_1 \longrightarrow \mathcal{L}(A) \subseteq P_1$
- $A \subset P_2 \longrightarrow \mathcal{L}(A) \subseteq P_2$

Therefore $\mathcal{L}(A) \subseteq P_1 \cap P_2$. Since this reasoning holds for any pseudo-closed $A \subset P_1 \cap P_2$, we end-up with $P_1 \cap P_2$ being pseudo-closed.

□

Remark The previous proposition does *NOT* state that pseudo-closed sets are a closure system. Indeed, the intersection of two pseudo-closed sets can also be a true point of $\mathcal{L}$. Also note that the minimal (inclusionwise) false point of an implication basis are pseudo-closed because all their subsets are true points. Since false points represent a partial order under inclusion, every elements of this poset contains at least one pseudo-closed set.

Proposition 10. *If two distinct sets P_1, P_2 are pseudo-closed, their essential sets are body disjoint.*

Proof. We are going to consider various cases. First suppose $P_1 \subset P_2$. Denote $\mathcal{E}_{P_1}, \mathcal{E}_{P_2}$ their essential sets. Suppose there exists $A \longrightarrow b_1$ in $\mathcal{E}_{P_1}$, and $A \longrightarrow b_2 \in \mathcal{E}_{P_2}$. If $b_2 \neq b_1$, we would have $b_2 \notin P_2$ and $b_2 \in P_1$ because $A \longrightarrow b_2 \notin \mathcal{E}_{P_1}$ contradicting $P_1 \subset P_2$. Then, let $b = b_1 = b_2$. We have $A \longrightarrow b \in \mathcal{E}_{P_1}$ and $A \longrightarrow b \in \mathcal{E}_{P_2}$. Hence:

- $b \notin P_1$ and $b \notin P_2$
- $A \longrightarrow b \in \mathcal{E}_{P_1}$ implies $\mathcal{L} \models A \longrightarrow b$. Because $A \subseteq P_1$, $\mathcal{L} \models P_1 \longrightarrow b$ holds too. That is $b \in \mathcal{L}(P_1)$

Recall that $P_1 \subset P_2$ and P_1, P_2 are pseudo-closed. Thus, $\mathcal{L}(P_1) \subseteq P_2$. But $b \in \mathcal{L}(P_1)$ and $b \notin P_2$ which is a contradiction. So, if $P_1 \subset P_2$ their essential sets are indeed body disjoint. Note that this case holds also if $P_1 = \emptyset$.

Suppose now $P_1 \cap P_2 = \emptyset$. If one of them is empty, the paragraph shows the result. Otherwise, their essential sets can obviously not contain implication with same bodies, contradicting the empty intersection.

Finally, consider the case $P_1 \cap P_2 \neq \emptyset$. If there is no false point in their intersection, then there is np $A \longrightarrow b$ with $b \notin A$, $A \subset P_1 \cap P_2$ holding. Thus their essential sets cannot share any implications with same body. Hence, let $A \subseteq P_1 \cap P_2$ be the greatest false point of $\mathcal{L}$ within $P_1 \cap P_2$:

(i) if $A \subset P_1 \cap P_2$, then $\mathcal{L}(P_1 \cap P_2) = P_1 \cap P_2$.

(ii) if $A = P_1 \cap P_2$, then by proposition 9, $P_1 \cap P_2$ must be pseudo-closed.

case (i). Suppose $A \longrightarrow b_1 \in \mathcal{E}_{P_1}$ and $A \longrightarrow b_2 \in \mathcal{E}_{P_2}$. Again either $b = b_1 = b_2$ or $b_1 \neq b_2$. If $b_1 = b_2 = b$ then $b \notin P_1 \cap P_2$ but $b \in \mathcal{L}(A) \subseteq \mathcal{L}(P_1 \cap P_2) = P_1 \cap P_2$ which is a contradiction. If $b_1 \neq b_2$ we have $b_1 \notin P_1$ but $b_1 \in \mathcal{L}(A) \subseteq \mathcal{L}(P_1 \cap P_2) = P_1 \cap P_2 \subseteq P_1$ which is a contradiction. The same goes for b_2. In fact we implicitly used the notion of separation used in [10].

case (ii). P_1, P_2 and $P_1 \cap P_2$ being pseudo-closed leads to:

- $P_1 \cap P_2 \subset \mathcal{L}(P_1 \cap P_2) \subset P_1$,
- $P_1 \cap P_2 \subset \mathcal{L}(P_1 \cap P_2) \subset P_2$

which can be used the same way as in case (i), concluding the proof.

□

Several elements of proofs have been given in [10] for the algorithm 14. Here we shall exhibit another kind of proof, using pseudo-closed sets. The claim is that the algorithm end up with the Duquenne-Guigues basis. Therefore we will show that in the procedure, if we add an implication, its premise is pseudo-closed.

Proposition 11. *In the algorithm BodyMinimal, we add an implication $\mathcal{L}_c(X) \longrightarrow \mathcal{L}(X) - \mathcal{L}_c(X)$ only if $\mathcal{L}_c(X)$ is pseudo-closed.*

Proof. Let us prove this proposition by induction. Our hypothesis that for all $0 \leq n \leq |\mathcal{B}(\mathcal{L})|$, at step n, taking $A = \min(\mathcal{L}_c(X) : \ X \in \mathcal{B}(\mathcal{L}) \wedge \mathcal{L}(X) \neq \mathcal{L}_c(X))$ implies that A is pseudo-closed.

Initial case At $n = 0$, taking A in $min(I_c(X))$ among bodies of $\mathcal{L}$ is taking A in $min(\mathcal{B}(\mathcal{L}))$ since $I_c = \emptyset$. Then, because minimal bodies of $\mathcal{L}$ are minimal false points of $\mathcal{L}$, $A = min(\mathcal{B}(\mathcal{L}))$ will be pseudo-closed.

Induction Suppose that the hypothesis is valid up to step n. Let us prove it for the step $n+1$. Consider taking A as in the algorithm. We have some cases:

(i) $\mathcal{L}_c(X) = X$

(ii) $X \subset \mathcal{L}_c(X)$

case (i). For all implications $E \longrightarrow F$ of $\mathcal{L}_c$ we have two possibilities, either $E \subseteq A$ or $E \nsubseteq A$. We cannot have $A \subset E$, because we would contradict the minimality constraint over closures of bodies. If for all implications $E \nsubseteq A$ we are in the initial case, because there is no subset of A being a false point of $\mathcal{L}$, then A is pseudo-closed. Now let $E \longrightarrow F$ be an implication such of the basis we are building such that $E \subset A$. Recall that E is a pseudo-closed set of $\mathcal{L}$ by induction hypothesis. Then since $A = \mathcal{L}_c(X) = X$, we conclude that $\mathcal{L}_c(E) = \mathcal{L}(E) \subseteq A$. In other word, E is pseudo-closed and its closure is included in A. Also, note that thanks to minimality constraint we must have taken all pseudo-closed sets included in A before the $n+1$-th iteration of the algorithm. Because this reasoning holds for all such implications, we can conclude that A is indeed pseudo-closed.

case (ii) Suppose $X \subset \mathcal{L}_c(X)$. Then there exists at least one pseudo-closed set E such that $E \longrightarrow \mathcal{L}(E)$ is in $\mathcal{L}_c$. Therefore $\mathcal{L}(E) \subset \mathcal{L}_c(X)$. Hence thanks to minimality constraint and induction hypothesis, all the pseudo-closed included in $\mathcal{L}_c(E)$ are in $\mathcal{L}_c$. $\mathcal{L}_c(X)$ being a false point of $\mathcal{L}$, we conclude that $\mathcal{L}_c(X) = A$ is indeed pseudo-closed.

Since the induction hypothesis is true for the initial step and for a general step n, we conclude that it is true in general, which proves the property.

$\square$

Proposition 11 shows that the algorithm produces indeed a body minimal basis from $\mathcal{L}$ and in particular, the Duquenne-Guigues basis.

Complexity analysis

2.2.2 Angluin Algorithm: Query Learning

In this section we are interested in the Angluin algorithm (1992). [1, 2].

Query Learning

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

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

To clarify, the terms negative/positive are related to the base $\mathcal{L}$ we want to learn.

Algorithm

The algorithm has been proved to end up with the Duquenne-Guigue Basis, again we can refer to [1, 2] for further explanations. The first algorithm provided by Angluin et al. relies on clauses with unitary heads. 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.

Algorithm 15: Angluin Algorithm

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

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

Some Observations

- Proved to be polynomial ([1])
- Proved to end up on the DG basis ([2])
- We can see that (apart from previous remark) that we end up on implications of the form $A \longrightarrow \mathcal{L}(A)$ (induction)
- .

Chapter 3

Implementation

3.1 Code we are starting from

The code uses `boost` library for c++. Especially bitsets. The file `config.h` defines the underlying structure we rely on (either `boost` or `bitmagic`). Sets are represented as bitmaps. The corresponding class is `FCA::BitSet` (we use the namespace `FCA`). Since the code is not documented, we will provide few remarks to ease understanding.

Here is a list with short description of file construction:

- `config.h`: defines which library to use for bitmaps,
- `biset.h/.cpp`: set representation as bitmaps
- `definition.h`: two operations:
  1. `IsPrefixIdentical`: check equality of sets up to $n-th$ element,
  2. `Convert`: convert a bitset to its string representation (provided an initial attributes names set)
- `implications.h/.cpp`: defines implications. One `Implication` class and one structure `ImplicationInd`. `Implication` is string base, `ImplicationInd` bitset based. Add some conversion methods (bitset to string, change attribute set).
- `datastructure.h`: gathers implemented datastructure to lighten inclusions (act as a package).
- `Closure.h/.cpp`: closure is a class (with only one method). In the `apply` function, the `if (false) return false` bloc after modification of NewClosure (|=) stands for testing whether the given set (`current` is closed or not). The method has thus two uses: computing the closure AND determining whether a set is closed or not.
- `LinClosure.h/.cpp`: two methods `Apply` (overloading). In the first one, the `bool` vector `use` aims to model $update \cup add$ because they are not the same structure in code. Thus it ensures that we do not add an element already treated.
- `LinClosure_Improved.h/.cpp`: Wild's Closure.

3.2 Measuring performance

This section is dedicated to explain the way we measure and compare algorithms implementation. We give the tools we use, and the way we use them. Of course, the main point is to compare them on time to see how do they react on various cases, against theoretical complexity.

3.2.1 Timer, profiler

We use two main tools, timers and profilers. The first one aims to precisely time execution of some functions, as much as comparing wall-clock time and CPU-time spent on a program. The profiler depicts a recursive analysis of time spent in each function, resulting in a call-graph, a tree-like structure. Profiling a program helps to understand bottlenecks. In our case, we can use it to see where does an algorithm is the most time-consuming, and possibly why.

Boost timer

Since the structure of the code relies on `boost` library. We use the `boost_timer` library to time programs. Few details though. The `boost_timer` dependency is not header-only, which means it must be compiled to be used and linked at compilation time. see boost doc.

`gperftools` CPU-profiler

`gperftools` is originally provided by Google but has been left to community since 2012. It contains tools such as heap-profiler, faster implementation of `malloc` and a CPU-profiler, which is our interest. It uses sampling method to determine time spent in functions scopes. Probably because of the CPU frequency (2.50 GHz), we cannot set a sampling frequency higher than 250 samples per second, that is 1 sample each 4 millisecond. Thus, the time results of the profiler are not more precise than 10e-3. This is the reason why we also use timer to get precise execution time. But as far as we are concerned, having this precision for profiling is enough for our purpose of examining bottlenecks.

`gperftools` is quite easy to use, and does not lead to runtime overhead contrary to `callgrind/valgrind` even though the latter one counting assembly instructions is more precise.

3.3 Ausiello Algorithm: Implementation

3.3.1 Data Structure

Recall the theoretical structure

In this paragraph, we will recall essential elements of an FD-Graphs, namely the possible arcs, the structure we may need to compute closure and so forth.

Code definition

The algorithms are given in terms of the size of an FD-graph. In [4], one can read that this size is assumed to be the size of a graph under adjacency list representation. Using a matrix based structure would take a quadratic amount of memory, and would be less efficient for adding/removing nodes, unmatching theoretical complexity. In the next paragraphs, we will explain more precisely the structure we chosen.

An FD-Graph can have two types of nodes (simple, compound), and up to 5 type of arcs (full, dotted, closed full, closed dotted, D). Furthermore, for closure and marking purposes, each vertex must have a counter embedded. Finally, we are supposed to build a graph out of an implication system based on `BitSet`. Hence, to ease conversion and computations, we label each vertex by its `BitSet` representation. Eventually, a vertex will be defined as in the following snippet:

```
typedef struct vertex vertex_t;

struct vertex{
  std::map<std::string, std::list<elt_t *>> edges;
  unsigned int counter;
};

typedef std::pair<FCA::BitSet, vertex_t> elt_t;
```

Where `elt_t` will be elements stored in the overhaul graph structure, being only a list of `elt_t`:

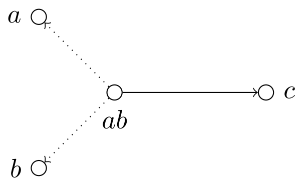
```
typedef std::list<elt_t> graph;
```

Note that we use pointer on adjacent nodes to represent edges (`std::list<elt_t *>`). This avoids duplication of `BitSet`, ease use of counters and allow for fast access to one node to a given adjacent one. The `map` structure deserves to represent all type of edges a vertex can have. We prefered `map` instead of say `list` or `vector` for the trade-off between access and code readability. Having string keys makes the code clearer, while keeping a logarithmic access time to a given list. Because the number of list is constant, the excess of computations with respect to a `vector` is negligible.

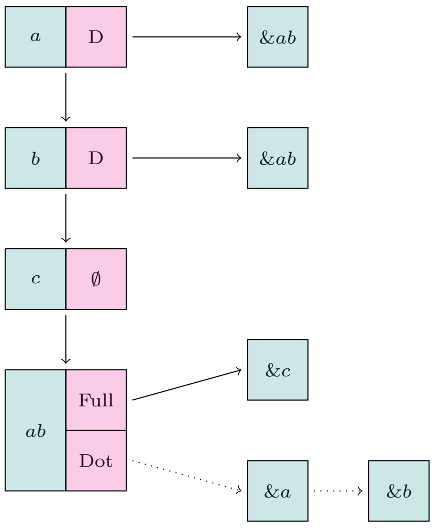
In order to illustrate, we give an example of a simple basis $ab \longrightarrow c$ represented through FD-Graph both graphically and with our data structure in figure 3.1. In this figure "&" stands for the *address* of following vertex. Also, for sake of clarity we did not drawn the closure of the graph nor the counters. Only initial arcs are represented: dotted, full and D ones.

Due to lack of time, this is the only structure we provided. Here are some ideas and points that one should read in regard to implement another structure for representing FD-Graphs. First recall that the graph should have a space complexity in $O(|\mathcal{L}|)$ according to Ausiello and al. (see [4]). In this paper, implication basis are provided under hypergraphs framework and the size $|\mathcal{L}|$ of $\mathcal{L}$ is the size of the structure in terms of adjacency lists. Fortunately, whether we consider hypergraphs or usual set notations, $|I|$ is still $O(|B||\Sigma|)$ since we have $|B|$ hyperarcs, going to at most $|\Sigma|$ single attributes (under reduced form). The size of the associated FD-Graph should be linear in the size $\mathcal{L}$.

We are going to investigate our structure. Because we made a structure supposed to store the closure of



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



(b) FD-Graph of $ab \longrightarrow c$ in memory

Figure 3.1: Representations of the implication $ab \longrightarrow c$ through FD-Graph

a graph also, we will study here the space used only for representation of the FD-Graph (simple/compound nodes, and dotted/full arcs) to be fair in comparison:

- we have $O(|\Sigma| + |B|)$ nodes,
- we will have full arcs to single attributes for bodies only, that is, $O(|B||\Sigma|) = O(|\mathcal{L}|)$ space complexity,
- we will have dotted arcs only from compound bodies of $\mathcal{L}$ to single attributes, also resulting in $O(|\mathcal{L}|)$ space complexity,
- for each dotted arc, we will have a D-arc, leading also to $O(|\mathcal{L}|)$ space complexity

Summing up, we will have at most $O(|\Sigma| + |B|)$ vertices and $O(3|\mathcal{L}|)$ edges into the whole graph, resulting in $O(|\Sigma| + |B| + 3|\mathcal{L}|) = O(|\mathcal{L}|)$ space complexity.

Note that the closure has a space complexity of $O((|B| + |\Sigma|)^2) = O(|\mathcal{L}|^2)$ because the closure can be a complete graph. This also matches theoretic expectations. Because algorithms sometimes need to run over compound nodes only, single attributes only or bodies only, we could have split the lists of vertices into both single and compound nodes. Here are few arguments explaining why we did not make so:

- single attribute can be bodies of $\mathcal{L}$, meaning that tearing apart single attributes on the one hand, and compound ones on the other would have bodies of $\mathcal{L}$ (nodes with full arcs) distributed over two distinct structures, breaking the interest of splitting;
- whereas the split would have ease for loops in algorithms, limiting them to a minimal number of steps, because structure may implicitly refers to each other (through edges), we would have lost code readability;

- the gain obtained in previously mentioned loops is also negligible in a sense. Provided we run over all vertices of the graph, selecting a vertex or not according to its properties is $O(1)$. Consequently, computations for vertices not matching wanted properties are reduced to this $O(1)$ checking, giving a small practical computation overhead and same theoretical complexity as divided nodes structure.

Also, to keep this splitting, we could have duplicated some nodes in order not to jump implicitly from one structure to another. But this would have obsoletely increased allocated space, and may have caused problem or required more operations for proper removal of vertices and edges. Indeed, we would have been compelled to check that if a node is deleted, there is no duplicate left in other lists.

A quick overview of complexity

In this part, we will describe a complexity analysis of routines not being algorithms of minimization. Namely, conversion from an implication basis to its graph representation and vice-versa. Because we use lists, adding an edge to a given node is constant time. Put another vertex in the graph without edges is also constant time for the same reason. Let us consider conversion from implications to graph first. We present the theoretical algorithm in 16. The procedure uses lists of vertices we previously describe $(L, D, \dots)$ instead of theoretical notations such as E for edges. Actually, this is just a matter of notations, and whenever we modify an L list, this can be seen as adding an edge to E. We chose this representation because the algorithm shows creation of the structure (notably in memory), and as such, this way of writing demonstrates it better.

Algorithm 16: Implications to FD-Graph

```
Input:
    $\mathcal{L}$: an implication system
    $\Sigma$: attribute set
Output: FD-Graph represention of $\mathcal{L}$ ($G_{\mathcal{L}}$)

foreach $x \in \Sigma$ do
    $V = V \cup \{x\}$ ;

foreach $A \longrightarrow B \in \mathcal{L}$ do
    if $A \notin V$ then
        $V = V \cup \{A\}$ ;
        foreach $a \in A$ do
            $L^1_A = L^1_A \cup \{a\}$ ;
            $D_a = D_a \cup \{A\}$ ;
    foreach $b \in B$ do
        $L^0_A = L^0_A \cup \{b\}$ ;
```

Provided access to a vertex can be done in $O(1)$, this algorithm runs in linear time in the size of $\mathcal{L}$. Because access to an element in a list requires linear time complexity in the size of the list, we used a marking trick to access vertices quickly. Access to an element is required to get D. Here is the idea: we use a temporary array (`std::vector`) of size $|\Sigma|$. Whenever we add a new attribute to the graph, we add its address to the array at its index within the attribute set. Because the articles assume reduced form, there is no need to test whether a node is already present when adding a new one. Hence, the algorithm runs in $O(|\mathcal{L}|)$. In practice however, to widen the application field of the algorithm, we added the possibility for $\mathcal{L}$ to have two distinct implications with same bodies. Ensuring that a same body is added only once requires $O(|B| + |\Sigma|)$ time (accessing an element in a list) resulting in an $O(|B|^2 + |\mathcal{L}|)$ conversion algorithm.

The conversion from graph to implications consists in running over all nodes of the graph with non-empty list of full arcs ($O(|B|)$) and adding each vertex of outgoing full arcs to the conclusion of such nodes ($O(|\Sigma|)$). Hence, backward conversion is $O(|\mathcal{L}|)$ time consuming.

One would be curious about removing a vertex from the graph. Since we use lists we must run across all nodes to see whether there is outgoing arcs to the node we wan to remove. Because the closure of a graph has quadratic size with respect to $\mathcal{L}$, removing a vertex may be quadratic as well. However, as we have seen in preceding chapter, removing a vertex is highly contextual, and with sufficient analysis we can achieve better complexity using the context of execution.

Conclusion

Bibliography

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

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

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

Glossary

beignet gauffrette. 2

closure operator or system, see definitions what.. 2

pouf pouif. 2

pseudo-closed set Minimal quasi-closed set (among its equivalence class). 2