UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

Data Mining using Constraint Programming

Dissertation presented by
Armand BOSQUILLON DE JENLIS

for obtaining the Master's degree in
Computer Science and Engineering
Option(s): Artificial Intelligence, Computing and applied mathematics

Supervisor(s)
Pierre SCHAUS, Aoga JOHN

Reader(s)
Pierre DUPONT, Hélène VERHAEGHE

Academic year 2015-2016

Abstract

The tasks of finding all frequent or closed frequent itemsets is a widely studied subject in the field of data mining. Recently, there have been some attempts to tackle those issues using the constraint programming (CP) paradigm. These attempts proved to be useful as CP is a declarative paradigm which enables to add any combination of constraints. This method contrasts with the traditional approaches taken to solve the itemset mining problems with specialized algorithms. These specialized algorithms show less flexibility as they must be modified in order to add a new combination of constraints on the itemsets found. For some specific tasks, the CP approach showed significant improvements over the existing algorithms, but its performances are worse on standard tasks. This thesis focuses on closing the gap between the performances of the traditional approach and the CP approach. The experiments conducted in this work highlight significant performance improvements over previous CP attempts. It also provides new insights regarding the use of ideas coming from state-of-the-art itemset mining systems in order to enhance CP approaches and vice versa. This thesis gives new theoretical bounds for the CP approach. It also shows that the operations executed to find the frequent (resp. closed frequent itemsets) are very similar to the one executed by the state-of-the-art Eclat (resp. LCM algorithms). It proves that these problems can be handled very efficiently by the use of a CP approach as demonstrated by LCM victory at the FIMI04 competition.

Contents

0.1 Introduction

Mining frequent itemsets is one of the most widely studied fields of data mining as it is an essential step in many data mining applications. The frequent itemset mining problem (FIMP) aims at finding all frequent itemsets in a large database of item transactions. A frequent itemset is an itemset present in at least a certain number of transactions given by the user.

The FIMP plays a key role in multiple dataming tasks as the discovery of association rules [2], correlations [7]and many other important data mining problems [16].

The set of all closed frequent itemsets $\mathcal{C}$ is a lossless compression of the set of all frequent itemsets $\mathcal{F}$. The closed itemsets are itemsets for which no supersets are present in less transactions.

A lot of specialized algorithms have been created to solve the frequent itemset mining problem (FIMP) and the closed frequent itemset mining problems (CFIMP) [25, 28, 27, 13, 4, 5]. These algorithms are often used as a first step in finding relevant patterns. For example, after solving the FIMP, one can search over $\mathcal{F}$ to find the frequent itemsets satisfying some relevant constraints.

Those could be the minimal cost and maximal size required for a frequent itemset to be considered as relevant.

As the set of frequent patterns satisfying the constraints of interest $\mathcal{F}_c$ is generally much smaller than $\mathcal{F}$, it is more interesting to create algorithms able to find directly $\mathcal{F}_c$. Adding additional constraints directly can be useful in order to reduce the large search space required to find $\mathcal{F}$ and to avoid the second step that creates $\mathcal{F}_c$ based on the very large set $\mathcal{F}$.
The existing specialized algorithm are now able to compute very efficiently $\mathcal{F}$ and $\mathcal{C}$. However, computing $\mathcal{F}_c$ or $\mathcal{C}_c$ from these sets is often computationally heavy, this is why multiple declinations of the specialized algorithms exist for different constraints or combinations of constraints.

In this thesis, the set of solutions $\mathcal{S}$ will refer interchangeably to $\mathcal{F}$ or $\mathcal{C}$, and $\mathcal{S}_c$ will refer interchangeably to $\mathcal{F}_c$ or $\mathcal{C}_c$.
Issues arise with these procedural approaches: if c_1 and c_2 are two different constraints, we usually have to make non-trivial modifications to these algorithms in order to compute $\mathcal{S}_{c_1}$, non trivial modifications in order to compute $\mathcal{S}_{c_2}$ and once again non trivial modifications in order to compute $\mathcal{S}_{c_1 \wedge c_2}$. However, in CP, we can easily create a so-called specialized propagator for each different constraints $c_1, c_2, .., c_n$ (n being any integer), and directly combine these propagators by means of the declarative modeling CP uses to directly find $\mathcal{S}_{any\ desired\ combination\ of\ c_1,c_2,..,c_n}$.
Even if the constraint programming approach has shown performance improvements over the specialized algorithms for some problems and their capacity to easily model more complex problems [14, 19, 21], they require much longer computation time when it comes to finding $\mathcal{S}_c$ with c being a combination of frequently used constraints like the minimum size or the minimum average cost required for an itemset to be returned as a solution.

The first focus of this thesis is to close the performance gap between the CP and the procedural approach. The experiments show that the algorithms presented here provide significant improvements over the previous approaches. It is especially true for large and sparse datasets. This is an expected result of using the Reversible Sparse Bit-Set data structure [10]. All my implementation use the state-of-the-art OscaR solver [22].

Solving Data Mining problems through the CP paradigm is nowadays an active research area [19, 14, 3, 8, 18, 15]. Part of the interest of this research area is to see how the declarative CP approach compares to the procedural approach used by the algorithms in Data Mining.

The second focus of this thesis is to give new insight about how these two approaches compare to one another.
To achieve it, I give insights about how one can transform a procedural algorithm based on the principles of Eclat [27] in a propagator and vice versa. I will also show that the operations executed by my algorithms to find $\mathcal{F}$ (resp. $\mathcal{C}$) are very similar to the one executed by Eclat [27] (resp. LCM [25, 26]). At the end of my analysis, I will give new theoretical bounds on the computation time needed to find $\mathcal{S}$ by the mean of CP. These bounds are unsurprisingly the same as the bounds of Eclat and LCM. This study is interesting as it shows how one can use ideas coming from the CP approach (like the frequency success rule developed here) to enhance existing procedural approaches and vice versa. It proves that these problems can be handled very efficiently by the use of a CP approach as demonstrated by LCM victory at the FIMI04 competition.
My theoretical analysis was also useful to show that the filtering rule introduced in [19] to produce C in a backtrack-free manner is time inefficient.

This thesis is organize as follows. Chapter 1 recalls preliminaries. It first introduces the necessary notions to understand and solve the itemset mining problems. Then it gives an introduction to the main principles of CP. Finally it explains the functioning of some useful data structure, the most important one being the Reversible Sparse Bit-Set [10]. Chapter 2 starts by providing two different kind of CP models to find $\mathcal{S}$. This thesis and the article A global

constraint for closed itemset mining (GCFCIM) [19] use the first kind of model while in CP4IM [8, 14] they use the second one. Then it introduces the different propagators needed by the first kind of models. Chapter 3 makes the theoretical comparison between traditional itemset miners and my algorithms. It also make an analysis of the time and space complexities. The last chapter gives some experimental results and concludes.

Chapter 1

Background

1.1 Itemset Mining

In this section I am going to present three itemset mining problems, *frequent*, *closed* and *maximal* itemset mining.
The problem of maximal itemset mining will be explained more briefly as I will not present the code I wrote to solve it.

Inuput:

vertical itemset database $\mathcal{V}$

Tid	A	B	C	D
1	1	0	1	0
2	0	1	0	0
3	1	0	1	1
4	1	0	1	1

minimum support threshold

$\theta = 2$

Output:

frequent itemsets $\mathcal{F}$

itemset	frequency
{A,C,D}	**2**
{A,D}	2
{C,D}	2
{D}	2
{A,C}	**3**
{A}	3
{C}	3
$\emptyset$	**4**

closed itemsets $\mathcal{C}$

itemset	frequency
{A,C,D}	**2**
{A,C}	3
$\emptyset$	4

maximal itemsets $\mathcal{M}$

itemset	frequency
{A,C,D}	2

Table 1.1: Examples of the frequent, closed frequent and maximal frequent itemset mining problems.

1.1.1 Frequent Itemset Mining

The frequent itemset mining problem was introduced in 1993 by Agrawal et al. [2]. Let the set $\mathcal{I} = \{1, ..., nItems\}$ be the set of every item identifiers. The set $\mathcal{T} = \{1, ..., nTrans\}$ be the set of every transaction identifiers. The **inputs** of the three itemset mining problems are first the **itemset database** $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{I}$ and second $\theta \in \mathbb{N}$ the **minimum frequency threshold**.
The itemset database can be represented in three equivalent ways:

binary representation $\mathcal{B}$

Tid	A	B	C	D
1	1	0	1	0
2	0	1	0	0
3	1	0	1	1
4	1	0	1	1

horizontal representation $\mathcal{H}$

Tid	itemset
1	{A,C}
2	{B}
3	{A,C,D}
4	{A,C,D}

vertical representation $\mathcal{V}$

item	tid-set
A	{1,3,4}
B	{2}
C	{1,3,4}
D	{3,4}

Table 1.2: Example of the three equivalent representations of the same itemset database $\mathcal{D}$. (left) binary matrix representation $\mathcal{B}$, (center) horizontal representation $\mathcal{H}$, (right) vertical representation $\mathcal{V}$. The set of all items is $\mathcal{I} = \{1, 2, 3, 4\}$, 1 being the identifier of the item A, 2 the identifier of item B etc. The set of all transactions $\mathcal{T} = \{1, 2, 3, 4\}$.

three itemset database representations:

- *binary matrix representation* $\mathcal{B}$: the itemset database is represented as a binary matrix $\mathcal{B}$ of size n×m. The matrix is such that $\mathcal{B}_{ti} = 1$ if the transaction t contains the item i ((t,i) $\in$ $\mathcal{D}$), $\mathcal{B}_{ti} = 0$ otherwise.
- *horizontal representation* $\mathcal{H}$: the itemset database is represented as a bag of n itemsets $\mathcal{H}_t$, each of them represent a transaction. $\forall\ t\ \in\ \mathcal{T}:\ \mathcal{H}_t = \{i\ \in\ \mathcal{I} \mid (t, i)\ \in \mathcal{D}\}$.
- *vertical representation* $\mathcal{V}$: the itemset database is represented as a bag of m tid-sets (or tid-lists) $\mathcal{V}_i$, each of them represent the set of transactions that contain the item i. $\forall\ i\ \in\ \mathcal{I}:\ \mathcal{V}_i = \{t\ \in\ \mathcal{T} \mid (t, i)\ \in \mathcal{D}\}$.

An example of the three equivalent representations of the same itemset database can be found in Table 1.2.
To understand the problem definition we need to understand the concepts of *coverage* and *frequency*.

Definition 1 (***coverage***)**.** *The coverage of an itemset I,* $\mathcal{T}_\mathcal{D}(I)$*, is the set of transactions that contain I:*

$$\mathcal{T}_\mathcal{D}(I) = \{t\ \in\ \mathcal{T} \mid \forall i\ \in\ I,\ (t, i)\ \in\ \mathcal{D}\} = \bigcap_{i\ \in\ I} \mathcal{V}_i$$

Definition 2 (***frequency***)**.** *The frequency of an itemset I,* $Freq_\mathcal{D}(I)$ *is the size of its coverage:*

$$Freq_\mathcal{D}(I) = |\mathcal{T}_\mathcal{D}(I)|$$

For example in the database of Table 1.2 we have $\mathcal{T}_\mathcal{D}(\{A, D\}) = \mathcal{V}_A \cap \mathcal{V}_D = \{3, 4\}$ *and* $Freq_\mathcal{D}(\{A, D\}) = |\{3, 4\}| = 2$.
Given a *minimum frequency threshold* θ, an itemset is called **frequent** if $Freq_\mathcal{D}(I) \geq \theta$, otherwise it is called **infrequent**. Now that we have introduced the necessary notions we can define what is the Frequent Itemset Mining problem FIMP.

Definition 3 (***Frequent Itemset Mining Problem FIMP***)**.** *Given an itemset database* $\mathcal{D}$ *and a minimum frequency threshold* θ*, return all the frequent itemsets i.e. compute the set* $\mathcal{F}$ *with:*

$$\mathcal{F} = \{I \mid I \subseteq \mathcal{I},\ Freq_\mathcal{D}(I) \geq \theta\}.$$

To simplifie the notations I will omit to put the reference to the itemset database $\mathcal{D}$ in indice, for example I will note $Freq(I)$ instead of $Freq_\mathcal{D}(I)$.
An important principle which is used to solve the itemset mining problem more efficiently is the Apriori principle.

Property 1 (***Apriori principle***)**.** *If an itemset is infrequent then any superset of it is also infrequent i.e.* $\forall I_1,\ I_2\ \subseteq \mathcal{I},\ if\ I_2 \supseteq I_1\ then:\ Freq(I_1) < \theta \Rightarrow Freq(I_2) < \theta$

This principle is due to the fact that $\forall I_1,\ I_2\ \subseteq \mathcal{I},\ if\ I_2 \supseteq I_1$ then any transaction containing I_2 contains I_1 so $\mathcal{T}(I_1) \supseteq \mathcal{T}(I_2)$ thus $Freq(I_1) \geq Freq(I_2)$. The fact that the frequency of any superset of an itemset is lower or equal than the frequency of the itemset is known as the **anti-monotone** property of the frequency.

Property 2 (***anti-monotone property of the coverage***)**.**

$$\forall I_1,\ I_2\ \subseteq \mathcal{I},\ I_2 \supseteq\ I_1 \Rightarrow \mathcal{T}(I_2) \subseteq \mathcal{T}(I_1)$$

Property 3 (***anti-monotone property of the frequency***)**.**

$$\forall I_1,\ I_2\ \subseteq \mathcal{I},\ I_2 \supseteq\ I_1 \Rightarrow Freq(I_2) \leq Freq(I_1)$$

Example and Application

An instance of the FIMP, along with its solution is shown in table 1.1. we have first the two inputs, the itemset database and the minimum frequency threshold, the solution is the set of itemsets that appear in the table **Frequent** itemset.

It is interesting to notice that the number of frequent itemset grows when θ decreases. This problem takes longer to solve for small frequency threshold as we have to find more patterns.

A typical application of this problem is for a supermarket to find items that are frequently bought together. A supermarket database can easily be seen as an itemset database, each recorded transactions can be seen as a line in the horizontal representation of the itemset database. For example if I have a supermarket database containing the three following transactions $t_1 = \{$bier, chips, goblet$\}$, $t_2 = \{$bier, chips, rice$\}$, $t_3 = \{$tea, water$\}$, by applying the Frequent Itemset Mining problem with $\theta = 2$, I can find the interesting pattern {bier,chips}. Then I can for example use this information in order to enhance the organization of the supermarket by putting the biers and the chips next to each other.

1.1.2 Closed Itemset Mining

Computing the set of closed frequent itemsets $\mathcal{C}$ is interesting as $\mathcal{C}$ is a lossless compression of $\mathcal{F}$ which can be exponentially smaller than $\mathcal{F}$. It avoids the presence of redundant itemsets which are itemsets necessarily present if other itemsets are present. We can also use the maximal frequent itemsets $\mathcal{M}$ as a compression of $\mathcal{F}$. The advantage of $\mathcal{M}$ is that it usually is much smaller than $\mathcal{C}$ but when we generate $\mathcal{F}$ from $\mathcal{M}$ we do not recover the information about the frequency of the elements of $\mathcal{F}$. To summarize we have $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$ and only $\mathcal{C}$ is a lossless compression of $\mathcal{F}$ as we do not lose the information about the frequency.
For remainder of this thesis I will talk about the **Itemset Mining Problem IMP** when talking interchangeably about the Frequent, Frequent Closed and Frequent Maximal Itemset Mining Problems.

Definition 4 (***closure***)**.** *Let* $\mathcal{I}_\mathcal{D}(T)$ *be a function that takes as argument a set of transactions* T *and returns the set of items common to those transactions:*

$$\mathcal{I}_\mathcal{D}(T) = \{i\ \in\ \mathcal{I}\ |\ \forall t\ \in\ T,\ (t,i)\ \in\ \mathcal{D}\}$$

Given this function, the ***closure*** *of an itemset* I $Closure(I)$ *is simply* $\mathcal{I}_\mathcal{D}(\mathcal{T}_\mathcal{D}(I))$*. We say that an itemset is* ***closed*** *if* $I = Closure(I)$*, otherwise we say that it is* ***unclosed****.*

Using the horizontal representation for the itemset database, the closure can be computed as follows:

$$Closure(I) = \mathcal{I}_{\mathcal{D}}(\mathcal{T}_{\mathcal{D}}(I)) = \bigcap_{t \in \mathcal{T}(I)} \mathcal{H}_t$$

For example using the horizontal database of Table 1.2 we can see that the closure of the itemset $\{A, D\}$ is $\{A, C, D\}$ as $\mathcal{T}(\{A, D\}) = \{3, 4\}$ and $\mathcal{H}_3 \cap \mathcal{H}_4 = \{A, C, D\}$.

The following property is used to recognize quickly if an item in the closure of an itemset and can also be used to compute the closure of an itemset.

Property 4 (***frequency equality***)**.** *if an item is in the closure of an itemset I then adding this item to the itemset will not change the value of the frequency i.e.*

$$\forall\ I\ \subseteq\ \mathcal{I},\ i\ \in\ \mathcal{I}:\ i\ \in\ Closure(I) \Leftrightarrow Freq(I) = Freq(I \cup i)$$

proof of the frequency equality property:

- $\Rightarrow$ Thanks to the anti-monotone property of the frequency we know that $Freq(I) \geq Freq(I \cup i)$.
  As $i\ \in\ \mathcal{I}(\mathcal{T}(I))$ then $\forall\ t\ \in\ \mathcal{T}(I),\ i\ \in\ t\ thus\ I \cup i\ \in\ t\ too \Rightarrow Freq(I) \leq Freq(I \cup i)$ thus $i\ \in\ Closure(I) \Rightarrow Freq(I) = Freq(I \cup i)$.
- $\Leftarrow$ I prove it by showing that the contraposition is true ($i\ \notin\ Closure(I) \Rightarrow Freq(I) \neq Freq(I \cup i)$). Using the fact that in any case $\mathcal{T}(I \cup i) \subseteq \mathcal{T}(I)$, I know that if $i\ \notin\ \mathcal{I}(\mathcal{T}(I))$ then $\exists\ t\ \in\ \mathcal{I}(I)\ s.t.\ i\ \notin\ t\ thus\ I \cup i\ \notin\ t\ but\ I\ \in\ t$ so $\mathcal{T}(I \cup i) \subset \mathcal{T}(I)$ which gives $Freq(I) \neq Freq(I \cup i)$.

□

Example of application: as $Freq(\{A, D\}) = Freq(\{A, D\} \cup \{C\})$ we know that C is in the closure of $\{A, D\}$. We can find the closure of an itemset by adding to this itemset all the elements in its closure. Here, as C is the only item i such that $Freq(\{A, D\}) = Freq(\{A, D\} \cup \{i\})$ we know that the closure of $\{A, D\}$ is $\{A, C, D\}$.
I presented the two strategies to compute the closure of an itemset, the first uses the horizontal representation of the itemset database and the second uses the frequency equality property.
Here are the condition usually used by the algorithms to check whether an itemset is closed or not.

Definition 5 (***closed itemset***)**.** *We say that $I \subseteq \mathcal{I}$ is closed iff:*

$$Closure(I) = I \Leftrightarrow \bigcap_{t \in \mathcal{T}(I)} \mathcal{H}_t = I \Leftrightarrow \nexists I' \subseteq \mathcal{I}\ s.t.\ I' \supset I\ and\ Freq(I') = Freq(I)$$

$$\Leftrightarrow \nexists i\ \in\ \mathcal{I} \setminus I\ s.t.\ Freq(I \cup i) = Freq(I)$$

Now that we have introduced the necessary notions we can define what is the Closed Frequent Itemset Mining problem CFIMP.

Definition 6 (***Closed Frequent Itemset Mining Problem CFIMP***)**.** *Given an itemset database $\mathcal{D}$ and a minimum support threshold θ, return all the closed frequent itemsets i.e. compute the set $\mathcal{C}$ with:*

$$\mathcal{C} = \{I \mid I \subseteq \mathcal{I},\ Freq_{\mathcal{D}}(I) \geq \theta,\ Closure_{\mathcal{D}}(I) = I\}.$$

An instance of the CFIMP, along with its solution is shown on table 1.1.

1.1.3 Maximal Itemset Mining

Definition 7 (***Frequent Maximal Itemset Mining Problem MFIMP***)**.** *Given an itemset database $\mathcal{D}$ and a minimum support threshold θ, return all the maximal frequent itemsets i.e. compute the set $\mathcal{M}$ with:*

$$\mathcal{M} = \{I \mid I \subseteq \mathcal{I},\ Freq_{\mathcal{D}}(I) \geq \theta,\ \forall\ I_2 \subseteq \mathcal{I},\ I_2 \subset I \Rightarrow Freq(I_2) < \theta\}.$$

An example of an instance of the MFIMP, along with its solution is shown on table 1.1.

1.2 Constraint Programming CP

Constraint Programming is a declarative programming paradigm used to solve Constraint Satisfaction Problems (CSP).

Definition 8 (***CSP***)**.** *A CSP is a triplet (V, D, C) where:*

- *V is a finite set of **variables** $\{v_1, ..., v_n\}$.*
- *D is called the **domain**, it maps every variable $v \in V$ to a set of possible values $D(v)$ that it can take ($D(v)$ is the domain of the variable v). $D = \{D(v_1), ..., D(v_n)\}$.*
- *C is a finite set of **constraints**. The constraints $c(x_1, ..., x_m) \in C$ are any boolean function defined on a subset of the variables. The set of variables $\{x_1, ..., x_m\} \subseteq V$ a constraint is defined on is called the scope of the constraint noted scp(c). A constraint can also be seen as a subset of the Cartesian product of the domains of the variables in its scope.*

We say that a variable v is **bound** if $|D(v)| = 1$ (v is assigned to a value in its initial domain). We say that the domain is **fixed** if all variables are bound.

Definition 9 (***Solution***)**.** *A solution to a CSP is a fixed domain that satisfies all the constraints.*

An example of CSP could be: V = $\{v_1, v_2\}$, D = $\{D(v_1) = \{3, 4\}, D(v_2) = \{3, 4\}\}$, C = $\{c_1\}$ with c_1 being the boolean function $v_1 \neq v_2$. The solutions of this CSP are $\{D(v_1) = \{3\}, D(v_2) = \{4\}\}$ and $\{D(v_1) = \{4\}, D(v_2) = \{3\}\}$.

1.2.1 Outlines of CP

CP can be defined by the equation **CP = Model + Search**. The model is a declarative description of the CSP to solve written by the user. The search is done by the system. I am now going to explain intuitively the search and the concepts we need to understand it, then I will define it more formally.

The basic idea of the search is to assign variables to values in their domain in a depth first search way. Then to output a solution each time we attain a leaf of the obtained search tree that satisfies all the constraints. As each leafs correspond to a fixed domain, this search simply enumerates all the possible fixed domains and outputs as solution the fixed domains that satisfies all the constraints. Clearly this approach is too expansive, in order to enhance it, we use a process called the **propagation** which is used to prune the search tree.

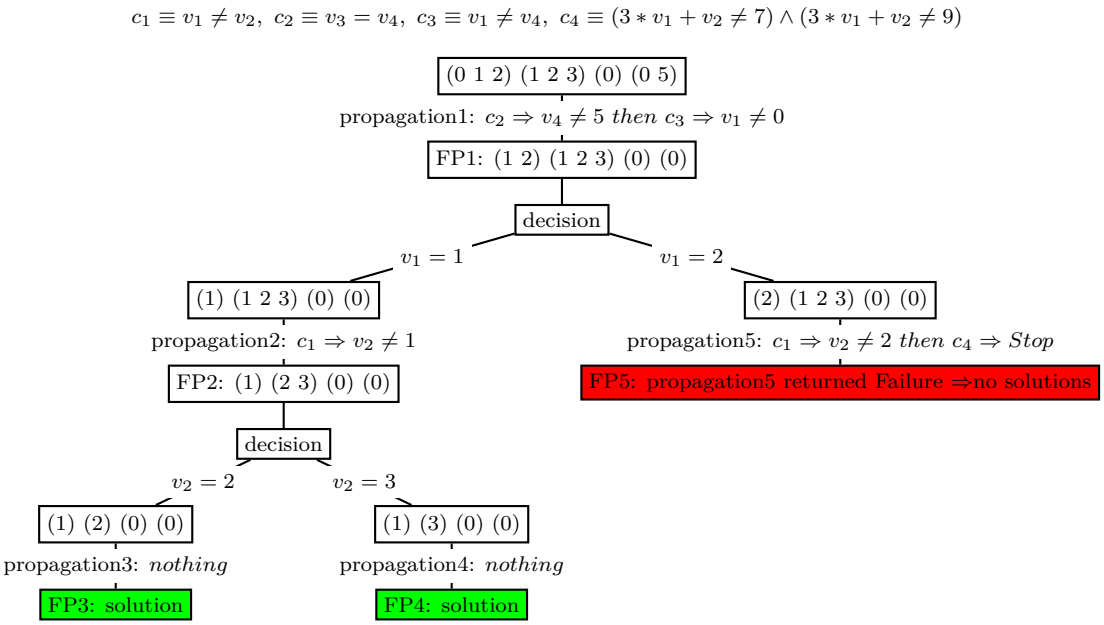


Figure 1.1: Example of CP search.

The constraints are implemented by **propagators**. The propagators are used during the propagation. A propagator can **stop the search** under the current node if it notices that the constraint it implements will no longer be satisfied. It can also **remove a value x from a variable v** if it notices that the constraint it implements will never be satisfied when v = x. It is called each time the domain of one or multiple variables in its scope have been modified. The search mainly consists of two steps:

- **propagation step:** this step uses the propagators to reduce the search space.
- **decision step:** this step is used to explore the search space. It takes a variable and tries every possible assignment for it.

1.2.2 Example of CSP solved by CP

An example of solving a CSP with CP can be found in figure 1.1. the CSP solved is: $V = \{v_1, v_2, v_3, v_4\},\ D = \{D(v_1) = \{0, 1, 2\},\ D(v_2) = \{1, 2, 3\},\ D(v_3) = \{0\},\ D(v_4) = \{0, 5\}\},\ C = \{c_1, c_2, c_3, c_4\}$. with the constraints being the following boolean functions: $c_1 \equiv v_1 \neq v_2,\ c_2 \equiv v_3 = v_4,\ c_3 \equiv v_1 \neq v_4,\ c_4 \equiv (3 * v_1 + v_2 \neq 7) \wedge (3 * v_1 + v_2 \neq 9)$. In order to have lighter notations, I will write the domain in a simpler way, for example if $D = \{D(v_1) = \{0, 2\}, D(v_2) = \{2, 3\}\}$, I will note it (0 2) (2 3).
The notation **FP** means **fixed point**, which means that no propagator will modify the domain anymore thus after a fixed point is reached, a decision is taken.

1.2.3 CP In More Details

I will now reexplain more formally what has just been said. A domain D' is called **stronger** than a domain D if $\forall\ v\ \in\ V,\ D'(v) \subseteq D(v)$. When using the notation **D'** what I mean is that D' is a domain stronger than the domain D currently considered.

Definition 10 (***propagator***)**.** *A propagator implements a constraint c by implementing failure,*

filtering and success rules. It is called each time the domain of a variable in scp(c) is reduced. The propagator takes the domain D as input and:

1. ***failure rule: Stops the search under the current node*** *by returning* ***Failure*** *if it notices that c can no longer be satisfied. More formally a failure rule is correct if it allows to return failure only if:*
   $\forall\ D',\ c(D'(x_1), ..., D'(x_m)) = false.$

2. ***filtering rule: Removes values from domains*** *that will never allow c to be true. More formally, a filtering rule is correct if it removes a value val from the domain of a variable var only if:*
   $\forall\ D'\ s.t.\ D'(var) = val,\ c(D'(x_1), ..., D'(x_m)) = false.$

3. ***success rule:*** *Tells the propagation process that c will always be satisfied under the current node by returning* ***Success****. It* ***avoids redundant calls*** *of the propagator*[1]*. More formally a success rule is correct if it allows the propagator to return Success only if:*
   $\forall\ D',\ c(D'(x_1), ..., D'(x_m)) = true.$

In OscaR [22], a propagator must implement the setup() and propagate() methods. The ***setup()*** *method is where we initialize the data structures used by the propagator and where we perform the first pruning. The* ***propagate()*** *method is used to enforce the rules of the propagator. For example propagation1 of figure 1.1 could have followed the following execution:* $c_1.setup()$, $c_3.setup()$, $c_4.setup(), c_2.setup()$ *removes 5 from* $D(v_4)$, *as* $v_4 \in scp(c_3)$ *call* $c_3.propagate()$ *which removes 0 from* $D(v_1)$, *as* $v_1 \in scp(c_1)$ *and* $scp(c_4)$ *call* $c_1.propagate()$ *and* $c_4.propagate()$.

Algorithm 1 gives the outlines of the search. This is not a totally general formulation how the CP search works as for example this algorithm uses only variables with binary domains but it is useful to write it like this to make the rest of this thesis easier to understand.
The main concepts represented in Algorithm1 along with their corresponding lines in this algorithm are the following.

- **propagation step** line 1: During this step the propagators are used to reduce the search space. We have to call a propagator P each time the domain of a variable in its scope has been modified in comparison to the domain at the end of the last call of P. If a propagator returns failure then propagate() stops and returns **failure**. when there is no more propagator to call we reach a **fixed point FP**.

- **variable heuristic** line 5: It gives the order over which we should try to bind the variables. To solve the IMP, I have one variable I_i for each item $i \in \mathcal{I}$ and the variable heuristic f I use is $f(I_i) = Freq(i)$. The idea of using a variable heuristic is to reduce the search tree by taking first the variables which are more likely to fail[2]. When I will note $i \leq j$, for any $i, j \in \mathcal{I}$ what it means is that $f(I_i) \leq f(I_j)$. In my algorithms when if I say itemset $I = \{1, 2, 3\}$, what it means is that I contains the first, the second and the third items occuring in the total order R created by f.

- **decision step** line 4 to 7: It takes the first variable in the order created by the variable heuristic and tries every possible assignments for it.

[1] When the propagator returns Success, it is never called again for the propagations happening under the current node.

[2] A variable fail when there is a constraint defined on this variable that can not be respected for any values of this variable

- **backtracking** line 3 and 9: This process is what restores the previous state where we have to make our next decision. The main structures for which backtracking restores the state are the domain of the variables. but other structures used by the propagator need be restored to their previous state during backtracking, I will indicate these structures by the prefix **r**. For example: rint, rboolean, etc...
  Two main strategies exist to restore the information:

  1. **Copying:** In this strategy, we copy the states before taking a decision. The backtracking simply use the copy to restore the states. This is the strategy used by Gecode [11] which is the solver used by CP4IM [14].
  2. **Trailing:** In this strategy, we only record the modifications. The backtracking undo those modifications. This is the strategy used by or-tool [12], OscaR [22], and Choco [23].

  An example of using the trailing strategy would be to use a rset S whose invariant is that it contains the number of each propagations that have been launched. My example is based on figure 1.1. At the end of the first propagation we have $S = \{1\}$. Then during propagation2 we add 2 to S and we record that when backtracking we should remove 2. Same process during propagation 3. When we reach FP3 we have $S = \{1, 2, 3\}$. As we find a solution, we backtrack to the the state of the next decision to take which is to bind v_2 to 3. During backtracking we undo the modification of propagation3 by removing 3 from S. When reaching FP4 we have $S = \{1, 2, 4\}$. As it is a solution we backtrack to the state of the next decision to take which is $v_1 = 2$. Doing So we undo the modification done in propagation4 and done in propagation 2 by removing 4 and then 2 from S.

Algorithm 1 CP-Search(D)

```
1: D = propagate(D)
2: if D == Failure then
3:     backtrack
4: if |unbound| ≠ ∅ then $ unbound = {v ∈ V | |D(v)| > 1}
5:     var = argmin_{v∈unbound} f(v)
6:     CP-Search(D_0) $ D_0 = D except that D(var) = 0
7:     CP-Search(D_1) $ D_1 = D except that D(var) = 1
8: else
9:     Output D as solution, backtrack
```

1.2.4 Proof of Correctness for a Propagator

Let SolCSP be the set of solutions of a CSP, and SolCP the set of fixed domains that its CP implementation output as a solution.
A propagator for the constraint c is correct iff:

1. **necessary condition:** When returning failure or removing values from domains, it does not remove any possible solutions. This implies that $D' \in SolCSP \Rightarrow D' \in SolCP$ thus $SolCSP \subseteq SolCP$.

2. **sufficient condition:** It does not allow any fixed domain D' stronger than the initial domain (the domain defined in the model before the search started) to be outputted as a solution if D' is such that $c(D'(x_1), ..., D'(x_m)) = false$. This implies that $D' \in SolCP \Rightarrow D' \in SolCSP$ thus $SolCP \subseteq SolCSP$.

Thus during the next sections, I will
prove that a propagator is correct as follows:

1. **necessary condition:** Prove that all the rules it implements are correct.
2. **sufficient condition:** Prove that it enforces the sufficient condition.

1.2.5 General Arc Consistency

I will now explain the concept of General Arc Consistency. This concept will be useful to prove the complexities of my algorithms. It is also interesting prove that a propagator is General Arc Consistent (i.e. it maintains the GAC property for the constraint it implements) because it proves that it can not perform a better filtering without removing some solutions.

Definition 11 (***General Arc Consistency***)**.** *A constraint c is General Arc Consistent iff all the values of the variables in its scope are* ***supported****.*

Definition 12 (***support***)**.** *A value* $val_i \in D(var_i)$ *is supported w.r.t. a constraint c iff for each other variables* var_j *in the scope of c, there exists a value* $val_j \in D(var_j)$ *s.t.* $c(x_1 = a_1, ..., x_i = a_i, ..., x_m = a_m) = true$

When a constraint is General Arc Consistent we can also say that it is domain consistent. When the search is performed with a GAC propagator, every leaf of the search tree is a solution. This is due to the fact that the propagator will never return failure[3]. Thus if nSol is the number of solutions of the CSP, the search will perform $nSol * 2 - 1$ calls to the propagator.

1.3 Reversible Structures

The Reversible structures are data structures for which we restore the previous state when backtracking. I will use them a lot in my propagators, especially the **Reversible Sparse Set** [9] as almost all the loops in my propagators are over the elements of a Reversible Sparse Set and the **Reversible Sparse Bit-Sets** [10] which will most of the time be used to represent the coverage of my itemsets.

1.3.1 Reversible Sparse Set

A Reversible Sparse Set rset represents a set S by the mean of three fields. The first field is an array **element** of size **p** containing a permutation of the elements possibly in S and the second is a rint **limit** used to delimit which elements of element are in S. The last field of the rset is the array **position** for which the eth entry represents the position of e in element The invariants of a rset are the following:

- limit = $|S| - 1$ and $e \in S \Leftrightarrow$ position[e]$\leq$ *limit* ($\Rightarrow$ element[0..limit] = S).
- element is a permutation of 0..p-1 and element[position[e]] = e

The **removal** of an element is done efficiently in $\mathcal{O}(1)$. To remove an element e, we swap it with the last element e' at position limit and decrements limit. For example if | marks the limit then

[3] As all values of every variables are supported after the fixed point is reached and the decision step binds one variable at a time, the propagation induced by the decision step will not return failure.

removing e from **(..e...e'|...)** (I represent only the table element) is done by swapping **(..e'...e|...)** then decrementing the limit **(..e'...|e...)**. For this **swapping** we do the following operations: posE = position[e], e' = element[limit] then swap the elements in element, element[posE] = e' and element[limit] = e, finally update the array position, position[e] = limit and position[e'] = posE.
The **reversiblility** is very efficiently implemented by just restoring the previous state of the rint limit. At each fixed point FPnew we store the previous value of limit if it has changed compared to the last fixed point FPold and when backtracking from FPnew to FPold we restore the value of limit. Thus to continue the example, if we backtrack to the state before e was removed we obtain **(..e'...e|...)**. The rset [9]are used in OscaR [22] implement the domains of the variables.

1.3.2 Reversible Sparse Bit-Sets

This subsection describes the class RSparBitSet which is the data structure that I use for fast coverage, closure and frequency computations. The class RSparBitSet (Reversible Sparse Bit-Set) is given in Algorithm 2.

RSparBitSet is similar to Reversible Sparse Set. It represents a set **T** thanks to an array called **words** of **p** 64-bits words. These 64-bit words are actually reversible long integers. We represents the elements of T by setting bits in words. When all the elements of a word in **words** are removed, as in rset we swap this word with the word just before the limit and we decrement limit. Doing so only the non 0 words are kept before limit while the 0 words are considered removed as they have an index greater than limit. When backtracking we restore the value of the rint limit and of the rlongs in words.

More formally, A RSparBitSet has three important fields, **words**, **limit** and **index**. The first invariant is:

1. ith bit of words[j] is set $\Leftrightarrow (j \times 64 + i) \in T$

Let's give an example of this invariant. First for the ease of understanding I will use 4-bit words in my examples, second I will use the notation $\leftrightarrow$ which means "corresponds to" third I will show the position of limit by |.

- RSparBitSet: (words: 1100 1000, index: **0 1**|, limit: **1**) $\leftrightarrow T = \{1, 2, 5\}$.

In RSparBitSet we have the second bit (i=2) of words[0] (j=0) set thus $(0 \times 4 + 2) = 2 \in T$. the second and third invariant are:

2. index is a permutation of [0,...,p-1]
3. words[index[i]] $= 0^{64} \Leftrightarrow i > limit, \forall\ i \in 0..p-1$ ($\Rightarrow index[0..limit]$ contains the index of the non zero words over which we loop during the different computations.)

For example if we remove 1 and 2 from T we obtain

- RSparBitSet': (words: 0000 1000, index: **1**| **0**, limit: **0**) $\leftrightarrow T' = \{5\}$.

by swapping index[0] with index[limit] and decrementing limit. When backtracking we restore the value 1100 of words[0] and restore limit to 1.
The implementation of RSparBitSet uses the BitSet data structure that contains an array words of p 64-bits words.

In Algorithm2, I present only the useful methods to understand the pseudo-codes of my propagators. Each of these methods take the BitSet **set** as argument. I added three methods to the RSparBitSet class, the first, isIncludedIn returns true iff $T \subseteq set$, the second and third compute $|T|$ and $|T \cap set|$ by the mean of the bitCount operation which counts the number of bit set in a Long. When calling Long.bitCount() in java, if the computer executing the code has the fast processor popcnt instruction for counting the number of bits set, the popcnt instruction is used.
The intersect method which returns true iff T and set have common elements *i.e.* $T \cap set \neq \emptyset$ was an already existing method. intersectionWith and remove methods are slight modifications of two existing methods. They modify the state of the RSparBitSet by computing the new state $words = T \cap set$ and $words = T \setminus set$. These methods return true iff words has been changed.

Algorithm 2 Class RSparBitSet

```
1: words: array of rlong $ words.length = p, it represents the set T
2: index: array of int $ index.length = p
3: limit: rint
4: procedure INTERSECT(set: BitSet): Boolean $ return true iff T ∩ set ≠ ∅
5:     for i from limit downto 0 do
6:         offset = index[i]
7:         if words[offset] & set.words[offset] ≠ 0^64 then $ bitwise And
8:             return true
9:     return false
10: procedure INTERSECTIONWITH(set: BitSet): Boolean $ words = T ∩ set
11:     changed = false
12:     for i from limit downto 0 do
13:         offset = index[i]
14:         w = words[offset] & set.words[offset] $ bitwise AND
15:         if w ≠ words[offset] then
16:             changed = true
17:             words[offset] = w
18:             if w == 0^64 then
19:                 index[i] = index[limit]
20:                 index[limit] = offset
21:                 limit = limit - 1
22:     return changed
23: procedure REMOVE(set: BitSet): Boolean $ words = T \ set
24:     changed = false
25:     for i from limit downto 0 do
26:         offset = index[i]
27:         if  words[offset] & set.words[offset] ≠ 0^64 then
28:             w = words[offset] & ¬set.words[offset] $ ¬ = bitwise NOT
29:             changed = true
30:             words[offset] = w
31:             if w == 0^64 then
32:                 index[i] = index[limit]
33:                 index[limit] = offset
34:                 limit = limit - 1
35:     return changed
36: procedure ISINCLUDEDIN(set: BitSet): Boolean $ return true iff T ⊆ set
37:     for i from limit downto 0 do
38:         index[limit] = offset
39:         if  words[offset] & ¬set.words[offset] ≠ 0^64 then
40:             return false
41:     return true
42: procedure CARDINALITY: Int $ return |T|
43:     sum = 0
44:     for i from limit downto 0 do $ bitCount gives the number of bit set.
45:         sum = sum + bitCount(words[index[i]])
46:     return sum
47: procedure CARDINALITYINTERSECTIONWITH(set: BitSet): Int $ return |T ∩ set|
48:     sum = 0
49:     for i from limit downto 0 do
50:         offset = index[i]
51:         sum = sum + bitCount(words[offset] & set.words[offset])
52:     return sum
```

Chapter 2

Models and Propagators

In the following sections I will explain the different propagators that I designed for the three different problems. As a reminder, in order to **prove that a propagator is correct** I will:

1. **necessary condition:** Prove that all the rules it implements are correct.
2. **sufficient condition:** Prove that it enforces the sufficient condition.

When using the notation D' what I mean is that D' is a fixed domain stronger than D. The different kinds of rules for a constraint c are correct under the following conditions:

1. **failure rule:** is correct if it allows to return failure only if:
   $\forall\ D',\ c(D'(x_1), ..., D'(x_m)) = false.$
2. **filtering rule:** is correct if it removes a value val from the domain of a variable var only when: $\forall\ D'\ s.t.\ D'(var) = val,\ c(D'(x_1), ..., D'(x_m)) = false.$
3. **success rule:** is correct if it allows the propagator to return Success only if $\forall\ D',\ c(D'(x_1), ..., D'(x_m)) = true.$

2.1 Models

In this section, I will first present the The models used in this thesis in order to solve the different IMP. These models are simpler to understand and based on the idea to use only one global constraint (i.e. a constraint with a variable number of arguments). This idea first appeared in [17]. Then I will explain the first CP model used to solve the different IMP. This CP model was proposed in [8], it is also explained in [14] and [21].

2.1.1 Models with a Global Constraint

The models for the IMP presented below use the same variables and initial domains but they differ on the constraints they use.
The triplet variables, domains, constraints is the following:

Variables: For each item i, I associate a variable I_i.

Domains: At the beginning I have $\forall\ i\ \in\ \mathcal{I}:\ D(I_i) = \mathbf{\{0,1\}}$.

When $D(I_i) == \{0,1\}$ I say that I_i is **unbound** and I write $I_i == U$. When $D(I_i) == \{0\}$ (resp. $D(I_i) == \{1\}$) I say that I_i is **bound** to 0 (resp. 1) and I write $I_i == 0$ (resp. $I_i == 1$), $(I_i == 0) \Leftrightarrow \neg(I_i == 1)$. When I write $I_i = 0$ (resp. $I_i = 1$) it means that that we bind I_i to 0 (resp. 1) Each itemset $X \subseteq \mathcal{I}$ is equivalent to an assignment of all the variables to values in their domain such that: the variables bound to 1 are the variables In the itemset and the variables bound to 0 are out of the itemset. I will often use the notation $\boldsymbol{I(D)}$ to refer to the set $\{i \in \mathcal{I} \mid I_i = 1\}$.

constraints: Let's $I = (I_1, ..., I_{nItems})$, the three constraints used to model the three IMP are defined as follows, $\forall\ I_2 \subseteq \mathcal{I}$:

1. (frequency constraint) $\boldsymbol{Frequent(I)} \Leftrightarrow Freq(I(D)) \geq \theta$.
2. (closedness constraint) $\boldsymbol{Closed(I)} \Leftrightarrow Closure(I(D)) = I(D)$.
3. (maximality constraint) $\boldsymbol{Maximal(I)} \Leftrightarrow if\ I_2 \supset I(D)\ then\ Freq(I_2) < \theta$.

As these constraints can take a variable number of arguments (depending on $nItems$), they are called global constraints. The idea to express a pattern mining problem with a global constraint first came for a sequential pattern mining task [17].Here is how they are used to model the different IMP:

1. *Frequent itemset mining problem:* $Frequent(I)$
2. *Frequent closed itemset mining problem:* $Frequent(I) \wedge Closed(I)$
3. *Frequent maximal itemset mining problem:* $Frequent(I) \wedge Maximal(I)$

We could also have defined the constraints in extension i.e. as a subset of the Cartesian product of the variables it is defined on. For example we can see the Frequency constraint as the set of domains D' s.t. $Freq(I(D')) \geq \theta$.

An item **i** is called **frequent** if the itemset $I(D) \cup i$ is frequent.

2.1.2 Models with Reified Constraints

This model uses the same decision variables $\boldsymbol{I} = (I_1, ..., I_{nItems})$ as the previous to represent an itemset. It also uses the auxiliary variables $\boldsymbol{T} = (T_1, ..., T_{nTrans})$ to represent the coverage of $I(D)$. So if we define $T(D)$ as being the set $\{t \in \mathcal{T} \mid T_t = 1\}$ we have $T(D) = \mathcal{T}(I(D))$. This model represents every given itemset $X \subseteq \mathcal{I}$ as follows: $i \in X \Leftrightarrow I_i == 1$ and $X \subseteq \mathcal{H}_t \Leftrightarrow T_t == 1$.

The coverage constraint: $\forall\ t \in \mathcal{T} : T_t == 1 \leftrightarrow \sum_{i \in \mathcal{I}} I_i(1 - \mathcal{B}_{ti}) = 0$ is used to maintain the relation $T(D) = \mathcal{T}(I(D))$ between T and I. Whenever an element of I is bound this constraint will:

0. bind any T_t for which $\exists\ i \in I(D)\ s.t.\ i \notin \mathcal{H}_t$ to 0.
1. bind any T_t for which $(I(D) \cup unbound) \subseteq \mathcal{H}_t$ to 1. With unbound $= \{i \in \mathcal{I} \mid I_i == U\}$.

The frequency constraint can be expressed in two different ways. The simplest is $\sum_{t \in \mathcal{T}} T_t \geq \theta$, it returns failure if $|T(D) \cup \{t \in \mathcal{T} \mid T_t == U\}| < \theta$. An equivalent way to express this constraint is: $\forall\ i \in \mathcal{I} :\ I_i == 1 \rightarrow \sum_{t \in \mathcal{T}} T_t \mathcal{B}_{ti} \geq \theta$. This constraint binds I_i to 0 if i is infrequent.
The closedness constraint: $\forall\ i \in \mathcal{I} :\ I_i == 1 \leftrightarrow \sum_{t \in \mathcal{T}} T_t(1 - \mathcal{B}_{ti}) = 0$ is used when solving the CFIMP. This constraint binds to 1 every $I_i\ s.t.\ I_i == U\ and\ i \in \bigcap_{t \in T(D)} \mathcal{H}_t = Closure(I(D))$ so that $I(D) = Closure(I(D))$. It returns Failure whenever $\exists\ i \in Closure(I(D))\ s.t.\ I_i == 0$.

2.2 Frequent Itemset Mining

In this section I explain the different propagators I designed to enforce the frequency constraint. Each of the following propagators take as input the vector of variables I and must enforce the constraint $Frequent(I)$.

2.2.1 Basic Frequency Propagator

In this subsection I am going to describe my first implementation of the frequency constraint. First let's show the only rule it enforces

Rule 1 (***failure of frequency***)**.**

$$Freq(I(D)) \geq \theta = false \Rightarrow return\ Failure$$

Let's now prove that this failure rule is correct:

correctness of the failure of frequency rule. If for some D, $Freq(I(D)) \geq \theta = false$, then thanks to the Apriori principle and the fact that for any fixed D' stronger than D, $I(D') \supseteq I(D)$, we can say that $\forall D' :\ Freq(I(D')) \geq \theta = false$. □

During the proof of the failure of frequency rule, I used the following trivial property:

Property 5 (***monotone property of I(D)***)**.**

$$\forall\ D'\ stronger\ than\ D,\ we\ have\ I(D) \subseteq I(D')$$

.

and proved the following property.

Property 6 (***failure of frequency***)**.** *for any domain D we have:*

$$Freq(I(D)) \geq \theta = false \Rightarrow \forall\ D'\ Freq(I(D')) \geq \theta = false$$

.

Each time I will define a propagator I will give a reference to its pseudo-code and the rules it enforces in the order in which it enforces them.

propagator 1 (***basic frequency propagator Algorithm 3***)**.** *Enforces the following rules:*

1. *failure of frequency rule.*

Let's now prove that my propagator is correct:

correctness of the basic frequency propagator.

1. **necessary condition:** already proved
2. **sufficient condition:** For any infrequent itemset, its representation with a fixed domain D is such that $Freq(I(D)) \geq \theta = false$ so no infrequent itemset can be outputed as a solution.

□

As all the following propagators of this section will first enforce the failure of frequency rule, I will not have to prove their sufficient condition. As I will prove the correctness of all the rules that my propagators use, they will all be trivially correct.
The basic frequency propagator presented in Algorithm 3 uses the following data structures.

coverage a RSparBitSet such that $coverage = \mathcal{T}$ when all the variables are unbound and $coverage = \mathcal{T}(I(D))$ otherwise. Thus coverage.cardinality = $Freq(I(D))$

unbound ReversibleSparseSet initialized to $\mathcal{I}$ that contains all unbound variables.

All my propagators will use the coverage data structure for fast coverage and frequency computation and a RSparBitSet (here it is the set unbound) containing all the variables usefull to enforce their different rules.

Setup And Cover Changed

In each propagator I implemented, the setup method and the propagate method enforce all the rules of the propagator, but the propagate method does not always try to enforce the rules when it is redundant to do so.
When the coverage didn't changed since the last time we called the propagator, we know that some of the rules are still enforced because they have already been enforced by the propagator for the same coverage. This is why in the pseudo codes I have a boolean variable **coverChanged** that avoids to try enforcing some already enforced rules if the cover did not changed.

When I will give the pseudo code of my propagators I will just give the pseudo code of their propagate method. Algorithm 3 is the pseudo code of the basic frequency propagator. As the following propagators receive the same inputs as my basic frequency propagator, I will not write them in their pseudo code.

Algorithm 3 BasicFrequency

```
1: Input: 𝒱 : vertical database; θ: minimum frequency threshold
2: Input: I = {I_1...I_nItems}: boolean item variables
3: coverChanged = false
4: for i ← unbound do
5:     if I_i == 0 then $ if I_i is bound to 0
6:         unbound.remove(i)
7:     else if I_i == 1 then
8:         coverChanged = coverage.intersectionWith(𝒱_i), unbound.remove(i)
9: if coverChanged then $ If the cover did not changed, it is redundant to enforce the failure
   of frequency rule.
10:    frequency = coverage.cardinality()
11:    if frequency < θ then $ failure of frequency rule
12:        return Failure
```

2.2.2 Search Trees

Figure 2.1 represent the application of my propagators for the frequency constraint applied to the FIMP of Table 1.1. At the beginning of the search all the variables are unbound this is represented by a U. When a variable is bound to 0 we represent it by a 0 and when it is bound to 1 we write its corresponding item.

For each of these search trees, The first vertical branch corresponds to the execution of the **setup** method. The red nodes represent the domains such that when the propagator is called with these domains as input, it returns failure. The non failed nodes contain the state of the domains when the fixed point is reached. The beginning of the edges correspond to a decision of the decision step. The green nodes represent the solutions. The **black branches** correspond to the branches where the propagator **doesn't compute** any intersection. The red branches correspond to the branches where the propagator has to compute at least one intersection. These are the heavy branches in terms of computation time.

If the surrounding of a node is green it means that the propagator returned success and thus don't have to enforce any constraint anymore (this is why I colored the end of the tree in light gray). The success check is performed in the green branches. section 2.2.6 will explain what a success check is.

2.2.3 Different Sets of Variables

To clarify my algorithms and demonstrations, I use the three of sets of variables represented in figure 2.6. Each parent set is the union of the two children sets. Each pair of children have no element in common. When talking about the elements of these sets, I can interchangeably talk about the variables they contain or about the items ids associated to these variables.

Definitions Of The Sets

The sets are defined as follows:

allVariables $= \{I_i \mid i \in \mathcal{I}\}$

unbound $= \{I_i \in allVariables \mid I_i = U\}$

bound $= \{I_i \in allVariables \mid I_i \neq U\}$

unboundNotInClosure $= \{I_i \in unbound \mid i \notin Closure(I(D))\}$

unboundInClosure $= \{I_i \in unbound \mid i \in Closure(I(D))\}$. By the frequency equality property we have:

$$\forall\, i \in unboundInClosure : \; Freq(I(D) \cup i) = Freq(I(D)).$$

boundTo0 $= \{I_i \in allVariables \mid I_i = O\}$

I(D) $= \{I_i \in allVariables \mid I_i = 1\}$

boundTo0Infrequent $= \{I_i \in boundTo0 \mid i \; is \; infrequent\}$

boundTo0Frequent $= \{I_i \in boundTo0 \mid i \; is \; frequent\}$

2.2.4 Reified Frequency Propagator

This propagator allows a better pruning than the first propagator but costs more computation at each call. I called it reified frequency propagator because it makes the same pruning as the reified frequency constraint of CP4IM. It works as follows:

propagator 2 (***reified frequency propagator Algorithm 4***)**.** *Enforces the following rules:*

1. *failure of frequency rule.*

Figure 2.1: **Search Trees of the different propagators solving the FIMP of Table 1.1**

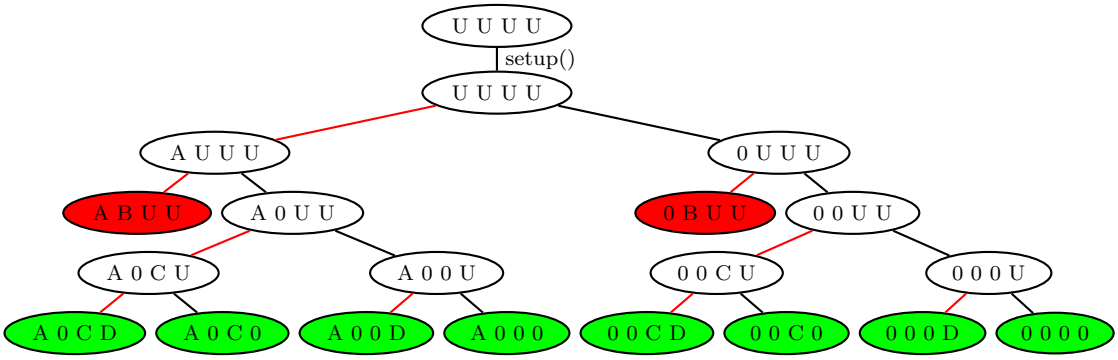


Figure 2.2: **basic frequency propagator**.

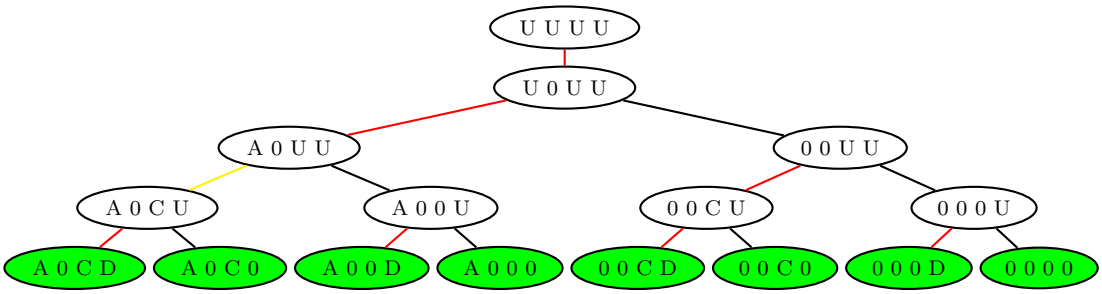


Figure 2.3: **reified frequency and reified frequency with closure propagators**: As we can see, these propagators notice earlier the items that we can not add to $I(D)$. During the first propagation, the infrequent variable I_B is bound to 0 by these propagators. Each time the decision step adds an item i to $I(D)$, we know that $I(D) \cup i$ will be frequent so these propagators never return failure, this is why they are GAC. The yellow edge represents the fact that when called on domain A 0 C U the reified frequency with closure propagator doesn't compute any intersection while the reified frequency propagator does. This is because the closure version already detected that C is in the closure of A and removed it from the unboundNotInClosure set. The reified frequency propagator has to compute the intersection at line 7 of Algorithm4 but as the cover does not change, it does not compute lines 9 to 15.

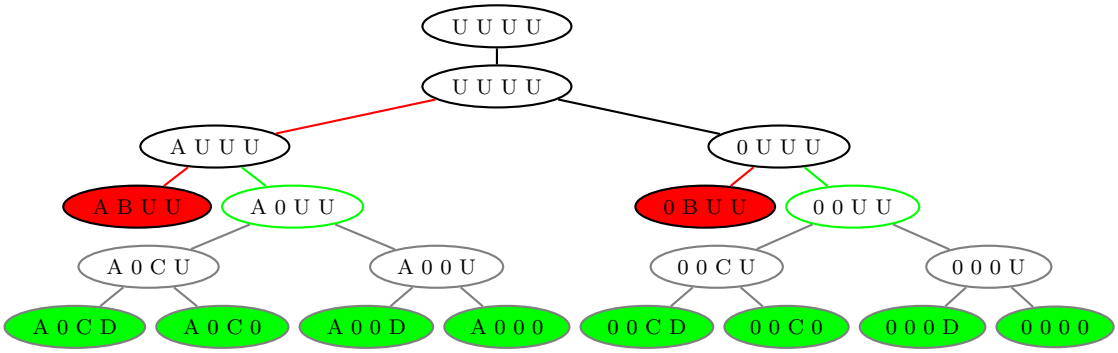


Figure 2.4: **basic frequency with success propagator:** As $Freq(\{B,\ C,\ D\}) < 2$, we have $nBound < trySuccess$ for the three top branches of the search tree thus the propagator doesn't perform the success check on them.

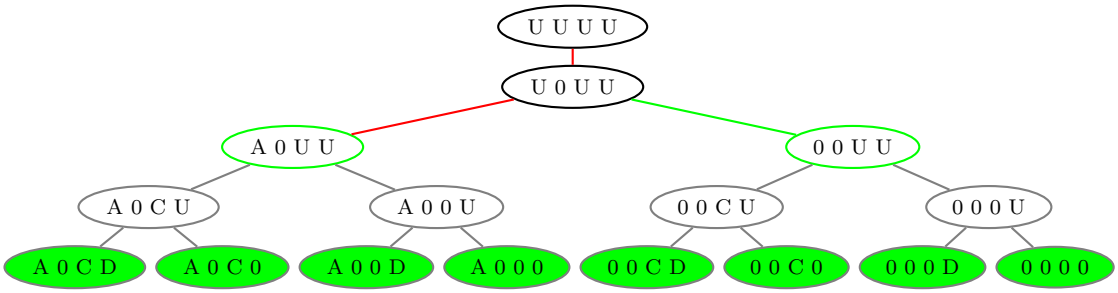


Figure 2.5: **reified frequency with success propagator**: A success check is performed on domains A 0 U U and 0 0 U U

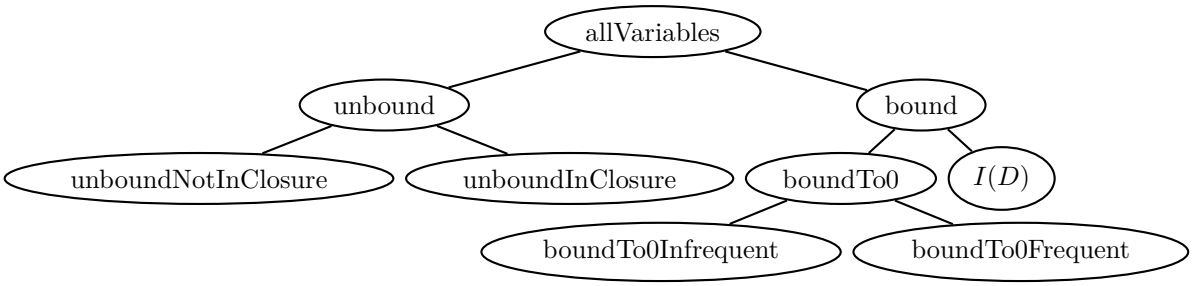


Figure 2.6: This tree represents the different sets of variables. The parent nodes are the union of the two children nodes and the intersection of these childrens is empty.

2. reified frequency filtering rule.

The reified frequency filtering rule is the following:

Rule 2 (***reified frequency filtering rule***)**.** *Bind to 0 any unbound variable I_i that corresponds to an infrequent item i, i.e.*

$$\forall\ i\ \in\ unbound:\ (Freq(I(D) \cup i) < \theta) \Rightarrow I_i = 0$$

Let's now prove that this filtering rule is correct:

correctness of the reified frequency filtering rule. When the reified frequency filtering rule removes the value 1 from the domain of a variable v, we now that $Freq(I(D) \cup i) < \theta$ thus by the failure of frequency property, we now that $\forall\ D' s.t.\ D'(v) = 1,\ Freq(I(D')) \geq \theta = false$. □

Algorithm 4 ReifiedFrequency(WithClosure)

```
1: $ The ReifiedFrequencyWithClosure algorithm is obtained by replacing every occurence of
   the set unbound by the set unboundNotInClosure and by using lines 18 and 19.
2: coverChanged = false
3: for i ← unbound do
4:     if I_i == 0 then $ if I_i is bound to 0
5:         unbound.remove(i)
6:     else if I_i == 1 then
7:         coverChanged = coverage.intersectionWith(V_i), unbound.remove(i)
8: if coverChanged then
9:     frequency = coverage.cardinality()
10:    if frequency < θ then $ failure of frequency rule
11:        return Failure
12:    for j ← unbound do $ reified frequency filtering rule
13:        freqWithIt = coverage.cardinalityIntersectionWith(V_j)
14:        if freqWithIt < θ then
15:            unbound.remove(i), I_j = 0 $Binds I_j to 0
16:        $ if freqWithIt == frequency
17:        $     unboundNotInClosure.remove(i)
```

General Arc Consistency

I will show that when enforcing the reified frequency filtering rule, $I(D)$ will always stay frequent. Let's show this by induction on the size of $I(D)$. At the beginning $I(D) = \emptyset$ thus, by definition

it is frequent. Now let's show that if the property is true when the cardinality of $I(D)$ is n then it is maintained true when the cardinality is n+1. By contradiction if it was false then the search could bind an infrequent[1] variable I_i to 1. But as we reached a fixed point before binding I_i to 1 and before reaching the fixed point I_i was an unbound variable such that $Freq(I(D) \cup i) < \theta$), I_i must be bound to 0. This contradicts the fact that I_i could be bound to 1 by the search.

As $I(D)$ is always kept frequent, we can bind every unbound variable to 0 and obtain a solution, thus we have the following property.

Property 7 (***support existence***)**.** *The values of the bound variables and the 0 values in the unbound variable are supported. The support is obtained by binding every unbound variable to 0.*

The only values that could not be supported are the 1 values in the domains of the unbound variable. The only way for these value to not be supported is to have an infrequent associated item. As my reified frequency filtering rule detects every infrequent items i and binds the corresponding variable I_i to 0, it is GAC.

2.2.5 Reified Frequency Propagator with Closure

This propagator is very similar to the reified frequency propagator but more efficient as it removes some useless operations.

propagator 3 (***reified frequency propagator with closure Algorithm 4***)**.** *Enforces the following rules:*

1. *failure of frequency rule.*
2. *reified frequency with closure filtering rule.*

The reified frequency with closure filtering rule is the following:

Rule 3 (***reified frequency with closure filtering rule***)**.** *Bind to 0 any unbound variable I_i among those that are not in the closure of $I(D)$ and that correspond to an infrequent item i, i.e.*

$$\forall\ i\ \in\ unboundNotInClosure:\ (Freq(I(D) \cup i) < \theta) \Rightarrow I_i = 0$$

Let's now prove that this filtering rule is correct:

correctness the reified frequency with closure filtering rule. To prove the correctness of this rule I am going to show that enforcing it is equivalent to enforcing the reified frequency filtering rule.

- $\Leftarrow$ This way is clear because unbound $\supseteq$ unboundNotInClosure.
- $\Rightarrow$ The only way to have the reified frequency with closure filtering rule enforced while not having the reified frequency filtering rule enforced is to be in a case where: $\exists\ i\ \in\ unBound\mathbf{In}Closure\ s.t.\ Freq(I(D) \cup i) < \theta$, which is impossible because if it is in the closure of $I(D)$ then $Freq(I(D) \cup i) = Freq(I(D))$ and as the failure of frequency rule is enforced first, we have $Freq(I(D)) \geq \theta$.

□

[1] So that $Freq(I(D) \cup i) < \theta$

2.2.6 Frequency Success Rule

The frequency success rule is a success rule that I use to enhance all my propagators. Using it leads to a faster computation of $\mathcal{F}$ when the variable heuristic f I use is $f(I_i) = Freq(i)$. Otherwise it slightly slows down the computation of finding $\mathcal{F}$. The idea is to have a fast way to compute the predicate $Freq(I(D) \cup unbound) \geq \theta$ and to return Success when it is true. To the best of my knowledge no one used it before. It helps removing some useless operations.

As for some constraints c like the minimum average cost constraint we need to use another heuristic than $f(I_i) = Freq(i)$ in order to compute $\mathcal{F}_c$ efficiently, this rule is not always interesting to add, I will thus discuss it in the section C.1.

2.3 Closed Itemset Mining

In this section I explain the propagators I designed to enforce the frequency and the closedness constraints together.
Each of the following propagators take as input the vector of variables I, the frequency threshold θ, the transactional database $\mathcal{D}$ and enforce the constraint $Frequent(I) \wedge Closed(I)$.

As a reminder we say that $I \subseteq \mathcal{I}$ is closed iff: $\nexists\ i\ \in\ \mathcal{I} \setminus I\ s.t.\ Freq(I \cup i) = Freq(I)$. We thus have the following property.

Property 8 (***failure of closedness***)**.** *For a given domain D, if* $\exists\ I_i\ \in\ boundTo0\ s.t.\ Freq(I(D) \cup i) = Freq(I(D))\ \ (\Leftrightarrow i\ \in\ Closure(I(D)))$*, we have* $\forall\ D'\ closed(I(D')) = false$.

This is due to the fact that we cannot bind i to 1 once it has been bound to 0. Thanks to the failure of closedness property we can easily see that the following failure rule is correct.

Rule 4 (***simple failure of closedness***)**.** *Return Failure each time there is a variable* I_i *bound to 0 s.t.* $i\ \in\ Closure(D)$.

$$\forall i\ \in\ boundTo0\ :\ Freq(I(D) \cup i) = Freq(I(D)) \Rightarrow return\ Failure$$

As when using the failure of frequency rule, no infrequent itemset can be outputed as a solution, and when using the failure of closedness rule no unclosed itemset can be outputed as a solution, for D to be a solution, we must have $Frequent(I(D))$ and $Closed(I(D))$. It implies that any propagator that enforces the failure of frequency rule and the failure of closedness rule, enforces the sufficient condition. So to show the correctness of the propagators to come I just have to show the correctness of their rules. For every definitions that follow, I assume that I first try to enforce the failure of frequency and failure of closedness rules, then I try to enforce the filtering rules.

The search trees associated to the different propagators that will be explained in this section can be found in figure 2.7.

2.3.1 LCMColumn Propagator

I called the propagator presented here LCMColumn propagator because, as it will be explained in the next chapter, I found that this propagator along with the search executes very similar operations to the LCM algorithm operations.

Rule 5 (***closure filtering rule***)**.** *Bind to 1 any unbound variable* $I_i\ s.t.\ i\ \in\ Closure(I(D))$

$$\forall\ i\ \in\ unbound:\ Freq(I(D) \cup i) = Freq(I(D)) \Rightarrow I_i = 1$$

Figure 2.7: **Search Trees of the different propagators solving the CFIMP of Table 1.1**

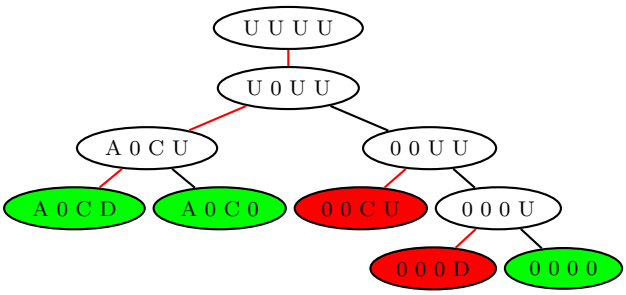


Figure 2.8: **LCMColumn and LCMLine propagators**: When going from U 0 U U to A 0 C U, the propagator binds I_C to 1 because, before it binds it to 1, we have: $I_C = U$ and $C \in Closure(\{A\})$ thus the **closure filtering rule** has to bind I_C to 1. The 0 0 C U and 0 0 0 D nodes correspond to domains for which the **failure of closedness rule** returns failure. The failure of closedness rule returned failure for the 0 0 C U node because $I_A = 0$ and $A \in Closure(\{C\})$.

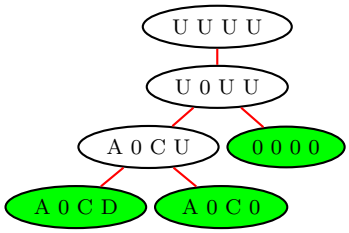


Figure 2.9: **GAC propagator**: As if we bind I_C or I_D to 1 we have $A \in Closure(I(D))$, when going from U 0 U U to 0 0 0 0, the **GAC filtering rule** binds I_C and I_D to 0.

to see the correctness of this rule we just have to see that the failure of closedness property applies to every D' s.t. D'$(I_i) = 0$.
Let's now reformulate the simple failure of closedness rule in order to avoid redundant work. As I assume that I always start by checking if $I(D)$ is frequent then the following property applies.

Property 9 (***non closure inclusion of infrequent items***)**.** *no infrequent item is in the closure of a frequent itemset.*

this property is a simple application of the frequency equality property. So I can rewrite the simple failure of closendess rule as follows

Rule 6 (***failure of closedness***)**.** $\exists\, I_i \in boundTo0Frequent$ *s.t.* $Freq(I(D) \cup i) = Freq(I(D)) \Rightarrow$ *return* $Failure$

The reified frequency filtering rule is still correct as the infrequent items it removes will never be part of the closure. Thus the following propagator is correct

propagator 4 (***LCMColumn propagator Algorithm 5***)**.** *Enforces the following rules:*

1. *failure of frequency rule.*
2. *failure of closedness rule.*
3. *reified frequency filtering rule.*
4. *closure filtering rule.*

The pseudo-code of this propagator is presented in Algorithm5. The set unboundOrBoundTo0Frequent used in this algorithm is defined as follows: **unboundOrBoundTo0Frequent = unbound ∪ boundTo0Frequent**

Algorithm 5 LCMColumnPropagator

```
1: for i ← unboundOrBoundTo0Frequent do
2:     if I_i = 1 then
3:         coverChanged=coverage.intersectionWith(V_i)
4:         unboundOrBoundTo0Frequent.remove(i)
5: if coverChanged then
6:     frequency = coverage.cardinality()
7:     if frequency < θ then $ failure of frequency rule
8:         return Failure
9:     for i ← unboundOrBoundTo0Frequent do $ failure of closedness rule
10:        if I_i = 0 then
11:            freqWithIt = coverage.cardinalityIntersectionWith(V_i)
12:            if freqWithIt == frequency then
13:                return Failure
14:            else if freqWithIt < θ then
15:                unboundOrBoundTo0Frequent.remove(i)
16:    for i ← unboundOrBoundTo0Frequent do
17:        if I_i == U then
18:            freqWithIt = coverage.cardinalityIntersectionWith(V_i)
19:            if freqWithIt < θ then $ reified frequency filtering rule
20:                unboundOrBoundTo0Frequent.remove(i), I_i = 0
21:            if freqWithIt == frequency then $ closure filtering rule
22:                unboundOrBoundTo0Frequent.remove(i), I_i = 1
```

2.3.2 LCMLine Propagator

The LCMLine propagator has the same effect as the LCMColumn propagator on the vector of variables it receives as input but this effect is reached through a different approach much closer to the one of the LCM algorithm. To see intuitively the difference between the LCMColumn and the LCMLine propagators, the first one uses intersections over the columns of $\mathcal{B}$ to know if an item is in the closure of $I(D)$ and the second uses intersections over the lines of $\mathcal{B}$.
More rigorously In the LCMColumn propagator, in order to know if an item is in the closure of $I(D)$, we used the frequency equality property[2] whereas in the LCMLine propagator, we start by computing directly the closure of $I(D)$ by computing $\bigcap_{t \in \mathcal{T}(I)} \mathcal{H}_t$ (line 11 of Algorithm6). This computation is performed by using a RSparBitSet.

2.3.3 GAC propagator

The GAC propagator is a general arc consistent propagator for the closure and frequency constraints. The rules used by this propagator are similar to the one used in the ClosedPattern-DC propagator presented in [19] . The difference between these propagators is that the rules used by the GAC propagator avoid some redundant operations. A more detailed explanation of the GAC propagator and the rules it uses can be found in the section C.2.

[2] by checking if $Freq(I(D)) = Freq(I(D) \cup i)$

Algorithm 6 LCMLinePropagator

```
1: coverChanged = false
2: for i ← unbound do
3:     if I_i == 0 then $ if I_i is bound to 0
4:         unbound.remove(i)
5:     else if I_i == 1 then
6:         coverChanged = coverage.intersectionWith(V_i), unbound.remove(i)
7: if coverChanged then
8:     frequency = coverage.cardinality()
9:     if frequency < θ then $ failure of frequency rule
10:        return Failure
11:    Q = ⋂_{t ∈ coverage.toSet()} H_t $ Q = Closure(I(D))
12:    for j ← Q do
13:        if I_j == 0 then $ simple failure of closedness rule
14:            return failure
15:        if I_j == U then $ closure filtering rule
16:            bind I_j to 1, unbound.remove(j)
17:    for j ← unbound do $ reified frequency filtering rule
18:        freqWithIt = coverage.cardinalityIntersectionWith(V_j)
19:        if freqWithIt < θ then
20:            unbound.remove(j), I_j = 0
```

Chapter 3

Comparison with Traditional Itemset Miners

In this section I will show how my CP-based algorithms compare with the traditional algorithms. There is two family of exact algorithms, the first explores the search space with a breadth-first search strategy [4, 5] and the second uses the depth-first search strategy [25, 28, 27, 13]. The depth-first search strategy is the new approach used to solve the itemset mining problems. It is usually faster and consumes less memory space. This approach is very similar to the one I use. I am going to show that all my algorithms can be transformed to an equivalent exact DFS algorithm computing the same number of intersections and coverage cardinalities. The reverse is also true, at least for the family of exact algorithms based on Eclat [27] algorithm, thus showing that my model can be solved with the same theoretical complexities as the exact DFS algorithms.

BFS: The BFS strategy was the first used strategy. The most well known BFS algorithm is APRIORI [4]. APRIORI solves the FIMP by first finding every frequent itemsets of size 1 then every frequent itemsets of size 2 and so on. To do this it uses two passes:

Generate: this pass generates a list G_i of itemsets of size i that might be frequent. To do this it uses the list F_{i-1} of frequent itemset of size i-1 that has been found by the pruning pass on G_{i-1}.

Prune: this pass uses the list G_i and outputs the list of every frequent itemsets F_i of size i by removing the infrequent elements of G_i.

Thus the algorithms starts as follows: generate a list containing all itmsets of size 1 then remove all the infrequent itemsets of this list we thus obtain $F_1 = \{i \in \mathcal{I} | Freq(\{i\}) \geq \theta\}$. Then the list G_2 contains every subset of size 2 of F_1 thus we do not have to prune over all the set of subsets of size 2 of $\mathcal{I}$ to find F_2.

3.1 Comparison with DFS Algorithms

I will study the DFS approach in greater details as it is very similar to my approach and usually more efficient than the BFS strategy. As I said I will show that my algorithms can be expressed as DFS algorithm and vice versa (at least for the family of algorithms based on the ideas of the ECLAT algorithm [27, 25, 28]). Meaning that a lot of what has been found in the research around DFS algorithms for solving IM problems can be reused to solve my model and what will

be found to solve my model can be reused by the traditional algorithms. The main concept used by the DFS algorithms is the concept of projected database.

projected database: The idea is that that the nodes of the search tree uses a database which is the same as the initial database but without useless information for the computation going on in the subtree behind it. We use the following observations to build the projected database

1. The information about the items that will not be used under the current node are removed. For example if I use a vertical database and I know that an item is infrequent I can remove its corresponding column from $\mathcal{V}$. An example of this in my algorithms is to keep only the set of items that might lead to the failure of a failure rule or to the filtering of a filtering rule. This set of items is for example the set unboundNotInClosure in the ReifiedFrequencyWithClosure propagator or the set unboundOrBoundTo0Frequent in the LCMColumn propagator.
2. The transactions that do not contain the itemset currently considered can be removed from the database. An example of this in my algorithms is to stop considering 0 words in the RSparBitSet data structure coverage.

To each node of the search tree corresponds a unique frequent itemset I, for which the corresponding projected database $\mathcal{D}_I$ is built. The outline of the DFS Algorithms for FIM is given in Algorithm7.
In order to easily see the similarity between my algorithms and the DFS algorithms I created the two following definitions:

1propagation: Is the propagation that occur because the decision step bound a variable to 1. A **0propagation** is the same but due to a bind to 0.

1node: noted $\boldsymbol{n(I,i)}$ Is the child node of a branch where a 1propagation occured due to the bind of variable I_i to 1. At this node we have $I(D) = I \cup i$. This 1propagation is noted $\boldsymbol{p(I,i)}$. A **0node** is the child node of a branch where a 0propagation occured. I consider the root of the search tree as a 1node noted n($\emptyset, \emptyset$). The child 1nodes of the 1node $n(I,i)$ are the 1nodes n($I \cup i$,j) with $j \in$ *unbound*. Thus each internal 1nodes has at most $nItems$ child 1node (except the root, every 1node has a parent 1node).

Algorithm 7 DFS(Itemset I, projected Database $\mathcal{D}_I$) \$ initially $I = \emptyset$ and $\mathcal{D}_I = \mathcal{D}$

```
1: F = {I}
2: determine a total order R on the items in D_I $ ≃ variable heuristic
3: for i occuring in D_I do $ ≃ decision step
4:     create D_{I∪i} from D_I $ ≃ p(I, i)
5:     D_{I∪i} contains:
6:     - only transactions in D_I that contain i
7:     - only items in D_{I∪i} that are frequent and higher than i in chosen order R
8:     F = F ∪ DFS(I ∪ i, D_{I∪i})
9: return F
```

The different type of DFS algorithms differ on the choice of how they store the projected database. Usually they use either FP-trees [13] [1] or tid-lists. The first algorithm using this approach was the Eclat algorithm. Recently multiple algorithms have been based on this approach among them two famous examples are Charm and LCM [28, 25] . LCM algorithms tends to outperform

[1] I will not show how to store a database using FP-trees in this document.

every other frequent itemset miners and closed itemset miners especially on the problem of closed itemset mining. One of the LCM implementation won the best implementation award in FIMI 2004. FIMI is a workshop of programming competition of data mining and last competition occured in 2004. In 2003 it is an implementation of FP-tree who won the competition.
The LCM algorithm compares very well to my approach and I will show that they are almost equivalent.

3.1.1 Eclat Algorithm

The idea of the algorithms based on the Eclat algorithm [27, 28, 26] is to store the following database for an itemset I:

$$\mathcal{D}_I = \{(j, \mathcal{T}(I \cup j)) = \mathcal{D}_I(j) \mid the\ information\ \mathcal{D}_I(j)\ will\ be\ usefull\}$$

When I say $\mathcal{D}_I(j)$ will be useful, what it means is that we will need it to generate every closed or closed frequent itemsets $I'\ s.t.\ I' \supseteq I$. Eclat algorithm generates the projected database $\mathcal{D}_{I\cup i}$ from $\mathcal{D}_I$ as follows:

$$\mathcal{D}_{I\cup i} = \{(j, \mathcal{D}_I(j) \cap \mathcal{D}_I(i)) \mid (j, \mathcal{D}_I(j)) \ \in\ \mathcal{D}_I\ and\ |\mathcal{D}_I(j) \cap \mathcal{D}_I(i))| \geq \theta\ and\ j \geq i\}$$

When I note $j \geq i$ what it means is that j appears after i in the created order R. Let $\mathcal{S}$ be the set of solutions found by Eclat, his pseudo-code is presented in the Algorithm8.
When we only use the uncommented part we solve the FIMP thus we have $\mathcal{S} = \mathcal{F}$. If we uncomment the end of the pseudo code we solve the CFIMP thus we have $\mathcal{S} = \mathcal{C}$. An example of solving the FIMP with algorithm 8 can be found in figure **??**. This figure will be explained in subsection "Comparison Of The CP And The Eclat Search Tree" but it is useful to look at it before what comes next.

The old versions of Eclat algorithm had a major drawback solved by the LCM algorithm,

Algorithm 8 Eclat(I, $\mathcal{D}$, T) $ initially $I = \emptyset$, T = $\mathcal{I}$ and at each call $T = \mathcal{T}(I)$

```
1: S = {I}
2: determine a total order R on the items in D
3: for (i, D_I(i)) ← D_I do
4:     D_{I∪i} = ∅ $ create projected database D_{I∪i} from D_I
5:     T' = D_I(i) $ = T(I ∪ i)
6:     for (j, D_I(j)) ← D_I with j > i do
7:         D_{I∪i}(j) = T' ∩ D_I(j) $ = T(I ∪ i ∪ j)
8:         if |D_{I∪i}(j)| ≥ θ then
9:             D_{I∪i} = D_{I∪i} ∪ (j, D_{I∪i}(j))
10:    S = S ∪ DFS(I ∪ i, D_{I∪i}, T')
11: $When the following lines are uncommented, this algorithm solves the CFIMP. These lines
    remove I from S if it is unclosed.
12: $ IisClosed = false
13: $ if ∄ (i, D_I(i)) ∈ D_I s.t. |T| == |D_I(i)| then
14: $     if ∄ C ∈ Closed s.t. C ⊇ I and Freq(C) == Freq(I) then
15: $         IisClosed = true
16: $         add I to Closed
17: $ if ! IisClosed then
18: $     S = S \ I
19: return S
```

they had to sometimes execute line 14 which can take up to $O(\mathcal{C})$ time complexity. As in these algorithm we lose the necessary information to check if I is closed, we have to maintain a repository $\mathcal{C}losed$ of already found closed itemset. In line 14 we check if no already found closed itemset is a superset of I of equivalent frequency. This operation is performed after every closed itemset that are possibly superset of I are found. The different Eclat algorithms differ in the way they execute the condition of line 14. Each of them use different clever data structures to store the already found closed itemsets and to execute the condition of line 14. Line 13 is an example of how to avoid the test of line 14. For example Charm [28] algorithm uses a more clever approach to avoid performing the test of line 14 too often.

3.1.2 Comparison of the CP and the Eclat Search Tree

Figure 3.1 is very useful to see the similarities between the execution of the CP algorithms and the Eclat based algorithms. First it is interesting to see that the CP search tree has the same shape as the Eclat search tree if we collapse each nodes containing "same $I(D)$" with their parent node. We can also see that each 1nodes shown in green, correspond to a unique frequent itemset. For any frequent itemset $I \cup i$ if we look at its corresponding one node $n(I, i)$, its set unbound is the same as the set of items occuring in $\mathcal{D}_{I \cup i}$. The gray edges correspond to the 0propagations which cost almost no computation time. In section3.1.4 I will use all the ideas discussed hear. I will also show that to each **1propagation $p(I, i)$** (occuring in the **black edges**) correspond the computation of the projected database $\mathcal{D}_{I \cup i}$ and that these computations are very similar.

It is interesting to notice that if we uncomment the lines 13 and 14 of Algorithm9 (which will be explained in next section), the only non empty projected database is $\mathcal{D}_{\emptyset}$.
It is also interesting to notice that if we order the items in increasing value of their frequency, then there is fewer computation needed. For example if we take the order B E A C D, we need less computation than if we use the order D C A E B.

3.1.3 Eclat With Sets Algorithm

Algorithm 9 is half way between my algorithms and Eclat algorithm and helps catch the similarities between the two of them. This algorithm uses the following sets:

I: is the itemset currently considered. It is equivalent to $I(D)$. Every solutions under the current node will contain I

unbound: is the set of items that could be added to I to form a solution.
This is equivalent to the set of unbound variables in my algorithms.

boundTo0: is the set of items that we do not consider to add to I anymore. No items in boundTo0 will be part of any solution under the current node.

test$_I$: is the set of items that are still useful for further tests. For example in my ReifiedFrequencyWithClosure propagator it is the set unboundNotInClosure and in my LCMColumn propagator it is the set unboundOrBoundTo0Frequent. In the Eclat algorithm it corresponds to the set of items that occur in $\mathcal{D}_I$.

If we uncomment the lines 13, 14, in 9, we obtain the same search as when using the ReifiedFrequencyWithClosure propagator. Otherwise we obtain the same search as in 8 and as when using the ReifiedFrequency propagator. In the second case we have test = unbound = the set of items that occur in $\mathcal{D}_I$. When we loop over unbound or test, the items i are taken following the order R.

Figure 3.1: Comparison of the CP and the Eclat search tree for the database and the minimum frequency threshold given below.

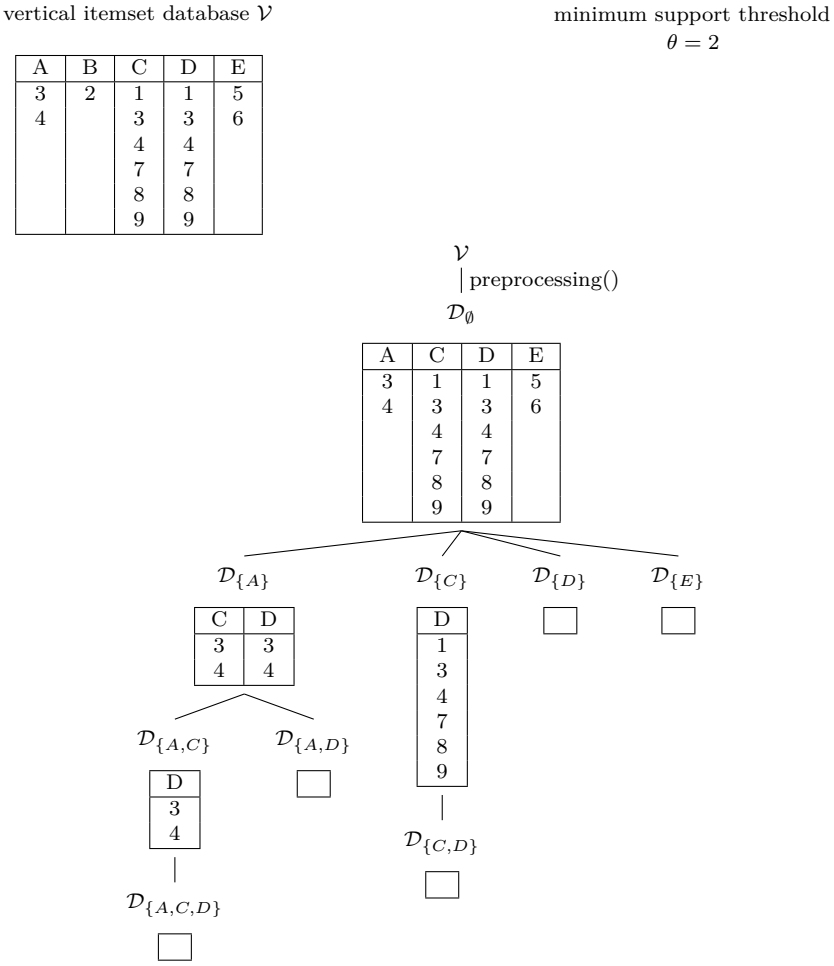


Figure 3.2: **Eclat search tree**:

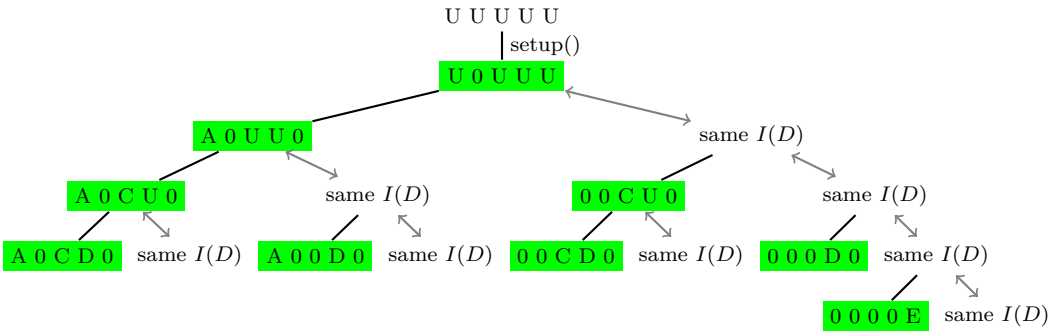


Figure 3.3: **CP search tree obtained when using the reified frequency propagator**: .

Algorithm 9 EclatWithSets(I, unbound, boundTo0, test$_I$, T) \$ initially $I = \emptyset$, unbound $= \mathcal{I}$, boundTo0 $= \emptyset$, test$_I = \mathcal{I}$, T $= \mathcal{I}$ and at each call $T = \mathcal{T}(I)$

1: $\mathcal{S} = \{I\}$
2: determine a total order R on the items in $\mathcal{D}$ \$ $\simeq$ variable heuristic.
3: unbound' = unbound, boundTo0' = boundTo0
4: **for** $i \leftarrow unbound$ **do** \$ lines 4 and 5 $\simeq$ decision step
5: $I' = I \cup i$, $ubound' = unbound' \setminus i$, $boundTo0' = boundTo0' \cup i$
6: $T' = T \cap \mathcal{V}_i$ \$ $= \mathcal{T}(I \cup i)$ \$ lines 6 to 14 $\simeq p(I, i)$
7: $test_{I'} = \{j \in test_I | \ j > i\}$
8: **for** $j \leftarrow test_{I'}$ **do**
9: freqWithIt $= |T' \cap \mathcal{V}_j|$ \$ $= \mathcal{T}(I \cup i \cup j) = |T' \cap \mathcal{D}_I(j)|$
10: **if** $freqWithIt < \theta$ **then**
11: $test_{I'} = test_{I'} \setminus j$
12: $unbound' = unbound' \setminus j$, $boundTo0' = boundTo0' \cup j$
13: \$ **else if**$(freqWithIt == \theta)$
14: \$ $test_{I'} = test_{I'} \setminus j$
15: $\mathcal{S} = \mathcal{S} \cup DFS(I', unbound', bounTo0', test', T')$

3.1.4 Nearly Equivalent Execution Time Demonstration

In this subsection I will demonstrate that the execution time of the Eclat algorithm and the resolution of my model with the reified frequency constraint are nearly equivalent because of the great similarities in the operation executed by these algorithms.
The proof of the nearly equivalent execution time of the LCM algorithm and the CP search with the LCMLine propagator is very similar to this one and would take too much space to write thus I will not write it.

The structure of the proof goes as follows:
firstly I show that **1propagations** (2) $\Leftrightarrow$ (2) **1nodes** (1) $\Leftrightarrow$ (1) $\mathcal{F} \Leftrightarrow$ **nodes of Eclat search tree** (3) $\Leftrightarrow$ (3) **database projections**. The order of my proof follows the number indicated. I start with 1nodes (1) $\Leftrightarrow$ (1) $\mathcal{F}$ which means that I will create a one-to-one mapping between 1nodes and frequent itemsets.

Secondly I will show that for every 1propagations, the operation executed in the database projection that corresponds to it in the one-to-one mapping I built are very similar (**1propagations** $\Leftrightarrow_{time}$ **database projections**). Therefore, the time needed to compute all the 1propagations is almost equivalent to the time needed to compute all the database projections.

I will finally show that we can ignore the time lost in the 0propagations (**ignore 0propagations**). Therefore we can conclude that the two execution times are nearly equivalent

1nodes (1) $\Leftrightarrow$ (1) $\mathcal{F}$**:** I will show that 1nodes $\Leftrightarrow \mathcal{F}$ by creating the one-to-one mapping $n(I, i) \leftrightarrow I \cup i$ which means that$n(I, i)$ corresponds to the frequent itemset $I \cup i$ in the mapping I create and vice versa.

- $\Rightarrow$ As by the support existence property when we bind every unbound variables to 0 we obtain a unique frequent itemset, the function that takes a 1node and follows the 0 branches

under it until it reaches a leaf[2] and returns its corresponding frequent itemset is an injective function from 1nodes to frequent itemsets.

- $\Leftarrow$ As every frequent itemset correspond to a leaf of the search tree, the function that takes a solution node and returns the 1node obtained by following the 0 branches above the solution node until we reach a 1node,[3] is an injective function from frequent itemset to 1nodes.

1propagations (2) $\Leftrightarrow$ (2) **1nodes:** We can create a one-to-one mapping $p(I,i) \leftrightarrow n(I,i)$ between the 1nodes and the 1propagations by associating every 1nodes $n(I,i)$ to its corresponding 1propagation $p(I,i)$ provoked by the decision to bind I_i to 1.

nodes of Eclat search tree (3) $\Leftrightarrow$ (3) **database projections:** We can create a one-to-one mapping between the database projections done when we add i to the itemset I and the node corresponding to the itemset $I \cup i$.

1propagations $\Leftrightarrow_{time}$ **database projections:** The bijection I created between 1propagations and database projections associates each 1propagation $p(I,i)$ to the computation of the database projection $\mathcal{D}_{I \cup i}$, because in the mappings I created I have $p(I,i) \leftrightarrow n(I,i) \leftrightarrow I \cup i \leftrightarrow$ node that contains $\mathcal{D}_{I \cup i} \leftrightarrow$ computation of $\mathcal{D}_{I \cup i}$.

Let's show that these two computations are very similar to do so I assume that the intersections and cardinality computation are the only costly operations.
Let's define T as follows T = $\mathcal{T}(I)$ = the set maintained by the coverage data structure.
First of all, at the beginning of the 1propagation $p(I,i)$ I have the redundant computation T' = T$\cap \mathcal{V}_i$ due to the fact that I do not maintain a projected database in memory but just the coverage of $I(D)$.
Then I have to compute $|T|$ for the failure of frequency rule, this is not executed by Eclat because the items are added one by one whereas in the CP environment, another constraint may have bounded multiple variable to 1 since the last propagate call.
I will now show that the computation needed to enforce the reified frequency filtering rule (lines 14 to 17 of Algorithm4) which is the most costly operation of the 1propagations is nearly equivalent to the computation needed to compute $\mathcal{D}_{I \cup i}$ (lines 6 to 9 of Algorithm8). It is clear that when using RSparBitSets, the computation $T' \cap \mathcal{V}_j$ is very similar in terms of execution time to the computation $\mathcal{D}_I(i) \cap \mathcal{D}_I(j)$. The cardinality computations $|T' \cap \mathcal{V}_j|$ and $|\mathcal{D}_{I \cup i}(j)|$ are the same.
Thus if $unbound == \{j \mid (j, \mathcal{D}_I(j)) \in \mathcal{D}_I \text{ and } j > i\}$, the execution time of computing the projected database and enforcing the reified frequency filtering rule are almost equivalent.

$unbound == \{j \mid (j, \mathcal{D}_I(j)) \in \mathcal{D}_I \text{ and } j > i\}$.

- $\subseteq$: $j \in unbound \Rightarrow j > i$ as the search decides on the first unbound variables. As the reified frequency filtering rule has been unforced we know that $i \in unbound \Rightarrow \mathcal{T}(I \cup j) \geq \theta$ thus $(j, \mathcal{D}_I(j)) \in \mathcal{D}_I$.
- $\supseteq$: $j > i$ means that j has not been bound by a decision step. $(j, \mathcal{D}_I(j)) \in \mathcal{D}_I \Rightarrow \mathcal{T}(I \cup j) \geq \theta$ thus j has not been bound by the reified frequency constraint so it must be unbound.

□

[2]If the node is already a leaf return direcly its corresponding itemset.
[3]If the solution node is a 1node we return it.

ignore 0propagations: I assume that each 0propagation can be forgotten because their time complexity is much smaller than the time complexity of the 1propagations and because there is as many 0propagation as 1propagations, I can thus say that the execution time of the Eclat based algorithms and of my algorithms are almost equivalent.

3.1.5 LCM Algorithm

LCM (Linear time Closed pattern Miner) [25, 26] was the first algorithm based on the principles of Eclat able to enumerate all the closed patterns without using any memory space for the previously found patterns.[4] There has been a lot of papers about solving the FIMP and it took a lot of research and time to arrive to an algorithm able to avoid the kind of test shown in line 14 of Algorithm8. The previous algorithms had a $\mathcal{O}(\parallel \mathcal{T} \parallel + |\mathcal{C}| \times nItems)$ space complexity and a $\mathcal{O}(\parallel \mathcal{T} \parallel \times \mathcal{F})$ time complexity. With $\forall\ T \subseteq \mathcal{T} : \parallel T \parallel = \sum_{t \in T} |t|$
To sum up very briefly the ideas shown in the paper [25], what they show with a quite long proof is that they are able to build a search tree for which each internal node correspond to a different closed itemsets thus they have a search tree containing $\mathcal{O}(\mathcal{C} * nItems)$ nodes. And if we call lines 2 to 6 of Algorithm 10, propagations, then there are only $\mathcal{O}(\mathcal{C} * nItems)$ propagations needed to find all the closed itemset. And as they show the difference between $|\mathcal{C}|$ and $|\mathcal{F}|$ is often up to exponential thus their algorithm is much more efficient than the previous ones. I will discuss LCM complexity in more details in the section Complexities. What is interesting is that even without knowing about how the LCM algorithm works I designed the LCMLine propagator 6 and I was able to easily prove its correctness. Moreover it is easy to see that when using this propagator, each internal 1nodes $n(I, i)$ of the search tree correspond to a different closed itemset as $p(I, i)$ creates a closed itemset Q at line 11 and if it can not modify D so that $I(D) =$ Q, then it fails. Thus for each internal 1node $n(I, i)$, we have $I(D)$ is the closed frequent itemset $Closure(I \cup i)$. Therefore we can create the injective function from internal nodes to closed frequent itemsets that takes a node, follows the 0propagations under it until it reaches a leaf and then returns the closed frequent itemset in the leaf[5]. As to each internal 1node correspond at most $nItems$ child 1nodes, we arrive to the conclusion that my algorithm is able to enumerate all the closed frequent itemset by using $\mathcal{O}(\mathcal{C} * nItems)$ propagations.

So I was able by using constraint programming to easily see how I can design, without knowing about LCM, an algorithm almost equivalent to LCM, to prove that it is correct and to prove its time complexity in a much easier way than they did for LCM. It leads me to think that using CP is a very intuitive tool to reason about the problem to solve.

Implementation of LCM

The pseudo-code of LCM 10 uses two definitions that I am going to introduce.
The first is **I(i)** which is the subset of I consisting of the elements smaller than i in the order created by the variable heuristic. $I(i) = I \cap \{1, ..., i\}$.
The second is $core_i(I)$ which is the minimum index i[6] such that $\mathcal{T}(I(i)) = \mathcal{T}(i)$ ($core_i(\perp) = 0$).

Once again I created an algorithm half way between LCM and the CP-search, along with one of my propagators. The Idea of these algorithms that I call the withSets algorithms is to easily see the similarities between CP and exact algorithms. It can be used to transform easily

[4] It does not uses something similar to lines 12 to 18 of Algorithm8.

[5] We could return directly the closed frequent itemset of the node but the idea of following the 0propagations until we reach a leaf is here to show that we have an injective function.

[6] Once again according to the order created by the variable heuristic which is equivalent to the order they create for the items.

an exact DFS algorithm as a propagator by first transforming it to a WithSets version and then to the corresponding propagator, the other way around is also true.

Algorithm 10 LCM(I : closed frequent itemset, $\mathcal{D}_\mathcal{I}$) $ initially $I = \emptyset$

```
1: output I
2: remove items that are infrequent from D_I $⇔ reified frequency filtering rule.
3: remove items of I from D_I
4: for i := core_i(I) + 1 to nItems with i occuring in D_I do
5:     I' = Closure(I ∪ i) $⇔ closure filtering rule.
6:     if I(i − 1) == I'(i − 1) then $⇔ failure of closedness rule.
7:         LCM(I', D_I)
```

Algorithm 11 LCMWithSets(I, unbound, boundTo0, T) $ initially $I = \emptyset$, unbound = $\mathcal{I}$, boundTo0 = $\emptyset$, test$_I$ = $\mathcal{I}$, T = $\mathcal{I}$ and at each call $T = \mathcal{T}(I)$

```
1: output I
2: unbound' = unbound, boundTo0' = boundTo0
3: for i ← unbound do$⇔ {i ≥ core_i(I) + 1 to nItems with i occuring in D_I}
4:     I' = I ∪ i, ubound' = unbound' \ i, boundTo0' = boundTo0' ∪ i
5:     T' = T ∩ V_i, stop = false
6:     Q = ⋂_{t ∈ T'} H_t $ ⇔ Q = Closure(I)
7:     for j ← Q if !stop do
8:         if j ∈ unbound then $ closure filtering rule.
9:             unbound' = unbound' \ j, I' = I' ∪ j
10:        if j ∈ boundTo0 then $ failure of closedness rule.
11:            stop = true
12:    for j ← unbound' if !stop do $ reified frequency filtering rule
13:        freqWithIt = |T' ∩ V_j|
14:        if freqWithIt < θ then
15:            unbound' = unbound' \ j, boundTo0' = boundTo0' ∪ j
16:    if !stop then
17:        LCM(I', unbound', bounTo0', T')
```

3.2 Complexities

I will discuss here the complexities of the different algorihms presented.
This section proves the following theorems:

Theorem 1. ***Mining $\mathcal{F}$****: Given a dataset $\mathcal{D}$, enumerating the frequent itemsets takes $\mathcal{O}(\mathcal{T}(I) * nItems)$ time for each frequent itemset $I \in \mathcal{F}$ when using the reified frequency or reified frequency with closure propagator.*

Theorem 2. ***Mining $\mathcal{C}$****: Given a dataset $\mathcal{D}$, enumerating the closed frequent itemsets takes $\mathcal{O}(\mathcal{T}(I) * nItems^2)$ (resp. $\mathcal{O}(\| \mathcal{T}(I) \| \times nItems)$) time for each closed frequent itemset $I \in \mathcal{C}$ when using the LCMColumn or GAC (resp. LCMLine) propagator.*

Theorem 3. ***memory space:*** *Each propagator of this thesis need $\mathcal{O}(\ nItems \times (nItems + nTrans))$ memory space to find $\mathcal{S}$.*

For the following demonstrations, I will use the fact that for any itemset I and transaction set T', computing $\mathcal{T}(I) \cap T'$ or $Freq(I)$ is done in $\mathcal{O}(\mathcal{T}(I))$ when using the Reversible Sparse

Bit-Set coverage.

3.2.1 FIMP

To compute the complexity of finding $\mathcal{F}$ when using the reified frequency propagator 4 I need to use a little trick. To each node n(I$\cup j$,i), I remove the the first intersection happening in line 7 of the associated propagation p(I$\cup j$,i) as its complexity is $\mathcal{O}(\mathcal{T}(I \cup j))$. Once this intersection is removed, p(I$\cup j$,i) is done in $\mathcal{O}(nItems * \mathcal{T}(I \cup i \cup j))$. But the removed intersections must be taken into account. To do it I add them to the parent propagation $p(I, i)$. As each $n(I, i)$ has at most $nItems$ child, I add at most $nItems$ intersections of complexity $\mathcal{O}(\mathcal{T}(I \cup i))$ to $p(I, i)$, thus I generate each $I \in \mathcal{F}$ in $\mathcal{O}(\mathcal{T}(I) * nItems)$. Adding the success check doesn't modify the complexity. This complexity is the same complexity as Eclat.

3.2.2 CFIMP

LCMLine propagator: As in the complexity analysis of LCM Algorithm10 they replaced the reified frequency test of line 2 by a failure of frequency test, I will do the same and remove the lines 17 to 20 of Algorithm6 to compute the complexity of finding $\mathcal{C}$ when using the LCM line expression propagator.
Doing so, the complexity of any $p(I, i)$ is the complexity of computing the closure of $I \cup i$ in line 11 of Algorithm6 which is $\mathcal{O}(\| \mathcal{T}(I) \|)$. As there is an injective function from internal nodes $n(I, i)$ to closed patterns ($n(I, i)$ corresponds to $Closure(I \cup i)$) and to each internal nodes there is at most $nItems$ corresponding child 1nodes whose propagation time are $\mathcal{O}(\| \mathcal{T}(I) \|)$, I generate each $I \in \mathcal{C}$ in $\mathcal{O}(\| \mathcal{T}(I) \| \times nItems)$ which is the same complexity as LCM.

LCMColumn propagator: The complexity of the LCMColumn5 propagator for $p(I, i)$ is $\mathcal{O}(\mathcal{T}(I \cup i) \times nItems)$[7]. As there is an injective function from non-failed 1nodes to closed itemsets that maps each non-failed 1node $n(I, i)$ to the closed itemset $Closure(I \cup i)$ and each of these non-failed 1nodes are the father of at most $nItems$ failed 1nodes, we can produce each closed frequent item I in $\mathcal{O}(\mathcal{T}(I) \times nItems^2)$.

GAC propagator: As the GAC propagator 13 produces a search tree without any failure, we can form a one-to-one mapping between 1propagations and closed frequent itemsets. The 1propagation $p(I, i)$ corresponds to the closed itemset $I' = Closure(I \cup i)$, we can also make a one-to-one correspondence with the 0propagation starting at the parent node of $n(I, i)$. These two propagations together can be computed in $\mathcal{O}(\mathcal{T}(I \cup i) \times nItems^2)$. Thus the theoretical complexity is not better than for the LCMColumn propagator. Moreover, the LCMColumn propagator usually performs a very good pruning thus in its complexity, the factor $nItems$ coming from the fraction $\frac{total\ number\ of\ 1propagations}{number\ of\ 1propagations\ leading\ to\ a\ non-failed\ node}$ is very close to 1. It explains why this propagator always outperforms the GAC propagator.

3.2.3 Spacial Complexity

As I do not modifie $\mathcal{D}$, I just need to keep one version of the dataset in memory which takes $\mathcal{O}(nItems \times nTrans)$. To store one version of my RSparBitSet coverage and the vector of

[7] Once again we can use the trick of removing the first intersection $\mathcal{T}(I) \cap i$ and add the corresponding computation time to the parent node.

variables I, I need $\mathcal{O}(nItems)$ memory space. As the maximal depth of my search tree is $nItems$, the total spacial complexity is $\mathcal{O}(\ nItems \times (nItems + nTrans))$.

Chapter 4

Experimental Results

The experiments conducted here are comparing my best implementations[1] with the state of the art CP methods and specialized methods. I will use LCM[2] [25] to represent the state of the art specialized miner as it was always the fastest specialized methods in the experiments conducted by CP4IM [14], and it was the only specialized method used in GCFCIM [19]. I ran my experiments on a MacBook Pro with a 2,8 GHz Intel Core i5 processor and 8 Go of RAM. My implementations use the state-of-the-art OscaR solver [22].

Finding my best propagator: In A we can find a serie of experiments conducted to determine the best propagotors of this thesis.
For the **FIMP** the fastest propagators are using the frequency success rule but as explained in 2.2.6, I will not use them for my experiments. I selected the **reified frequency with closure propagator** for my experiments as it is the fastest propagator without the frequency success rule.

For the **CFIMP**, the LCMColumn propagator was always much faster than the GAC propagator. On the small and dense datasets it is much faster than the LCMLine propagator and slower on large and sparse datasets. I will therefore use the **LCMColumn** propagator for my experiments.

Variable heuristic: All the experiments use the variable heuristic $f(I_i) = Freq(i)$, that sorts the variables in increasing order of the frequency of their corresponding item. I tried a few more clever heuristics but their effect is almost nonexistent .

4.1 Datasets

In table 4.1, I give the characteristics of the datasets used for my experiments. For each dataset I report the number of items $nItems$, the number of transactions $nTrans$, the average size of a transaction $\widehat{|\mathcal{T}|}$, its density $\sigma = \frac{\|\mathcal{T}\|}{nItems \times nTrans}$ and its size (size $= nItems \times nTrans$). Let's the support *sup.* be the value $\theta = sup. \times nTrans$, in the fraction $\frac{|\mathcal{F}|}{|\mathcal{C}|}$, $\mathcal{F}$ and $\mathcal{C}$ are computed with a 10% *support*. I don't give the value of $\frac{|\mathcal{F}|}{|\mathcal{C}|}$ for T10I4D100K and retail because they are not representative[3] and not useful for the following analysis. The datasets are presented in increasing size order.

[1] the codes are available here: https://bitbucket.org/projetsJOHN/fim-cp
[2] I used LCM-v5.3 available here http://research.nii.ac.jp/ uno/codes.htm
[3] With a 10% *support* there is only one (resp. 10) frequent itemset for T10I4D100K (resp. retail).

The datasets soybean, anneal, splice-1 and mushroom were found on the website of CP4IM[4] The datasets T10I4D100K and retail were found on the FIMI repository[5].

Dataset	$nItems$	$nTrans$	$\widehat{\lvert\mathcal{T}\rvert}$	σ	$\frac{\lvert\mathcal{F}\rvert}{\lvert\mathcal{C}\rvert}$	size
soybean	50	630	16	32%	9.5	31 500
anneal	93	812	42	45%	120	75 516
splice-1	287	3 190	60	21%	1	915 530
mushroom	119	8 124	21.5	18%	47.3	966 756
T10I4D100K	1 000	100 000	10	1%	—	100 000 000
retail	16 470	88 162	10	0.06%	—	1 452 028 140

Table 4.1: Dataset Charasteristics.

4.2 Comparison with CP4IM and LCM on Small Datasets

For the comparisons I used the available distribution of CP4IM[6]. As explained later CP4IM is unable to manage the two large datasets T10I4D100K and retail. This section will therefore compare LCM, my codes and CP4IM only on the soybean, anneal, splice-1 and mushroom datasets. In Figure4.1 we can see that even if I have promising results on the splice-1 dataset, my performances are still faraway from LCM. The performance improvements over CP4IM are nevertheless important.

ReifiedFrequencyWithClosure and LCMColumn vs CP4IM: As explained in [14], CP4IM has an inefficient representation of sparse data which explains the bigger performance gap when the density decreases.

ReifiedFrequencyWithClosure and LCMColumn vs LCM: When comparing my implementations to LCM, it is important to keep in mind that the performances of LCM are well known to be very implementation dependent. So even if as explained in my theoretical analysis, my algorithms are very similar to LCM, the implementation tricks can have a dramatic effect on the performances. I think that most of my results can be explained by one implementation trick that I do not use.
In the implementation of LCM that I used for my experiment, during their database projection, they merge the identical transactions into one. The effect of this implementation trick is most visible when using the anneal dataset. Indeed the transactions in anneal are very similar to one another thus a lot of merges can be achieved when using this dataset. I think that it explains the huge performance gap between the reified frequency with closure propagator and LCM for the anneal dataset. The transactions in soybean, splice-1 and mushroom look also quite similar, but I think that this merging operation has more impact for soybean (resp. mushroom) because $\widehat{|\mathcal{T}|} = 16$ (resp. $\widehat{|\mathcal{T}|} = 21.5$) while for splice-1 $\widehat{|\mathcal{T}|} = 60$. I also think that the value of $nTrans$ for mushroom leads this merging operation to a greater impact.

Intuitively we can see that when a dataset has a lot of very similar transactions, $|\mathcal{C}|$ is much smaller than $|\mathcal{F}|$. And when mining a dataset with very similar transactions, we will jump from one closed itemset to its much bigger children closed itemset. Doing so the coverage of the current itemset changes more frequently than when solving the FIMP. The time invested in merging equivalent transactions will therefore have a less lasting positive impact on the

[4]https://dtai.cs.kuleuven.be/CP4IM/datasets/
[5]http://fimi.ua.ac.be/data/
[6]https://dtai.cs.kuleuven.be/CP4IM/download.php

future computations. This could explain why the performance gap between LCM and my implementations is smaller when solving the CFIMP.

I think that my approach might be an alternative to LCM for datasets like splice-1 where there is few transactions to merge.

4.3 Comparison with GCFCIM, LCM and CP4IM on Small and Large Datasets for the CFIMP

As the implementation of GCFCIM is not available yet[7], I had to use directly the results that they present in [19]. All their experiments were "conducted on an Intel Xeon E5-2680 @ 2.5 GHz with 128 Gb of RAM". As explained in [19], the model used by CP4IM [14] is very memory consuming. This explains the OOM (Out Of Memory) of Table4.2. So even with 128 Gb of RAM CP4IM is unable to tackle the CFIMP for the two large datasets. The LCM algorithm usually runs faster on my computer than their but not CP4IM (I think that this is due to the memory issues). I think that the implementation of GCFCIM might run a little bit faster on my computer but certainly not more than two time faster.

GAC vs ClosedPattern-DC: These two propagator use equivalent rules and are both GAC so the performance improvements of my GAC propagator can not be explained by a better pruning. I think that the improvements comes first from the use of the very efficient Reversible Sparse Bit-Set data structure. The second source of performance improvements comes from the redundant operation that I was able to avoid by using different rules equivalent in terms of filtering.

LCMColumn vs GAC and ClosedPattern-DC: The last column of table 4.3 computes the fraction $\frac{number\ of\ nodes\ for\ the\ LCMColumn\ propagator}{number\ of\ nodes\ for\ the\ GAC\ propagator}$. The values in this column suggest that even if not perfect, the pruning of the LCMColumn and LCMLine propagators is already very good. Actually when using the LCMColumn propagator, the 0propagations are much lighter in terms of computation than the 1propagations, which is not the case when using the GAC and the ClosedPattern-DC propagators thus there is fewer costly propagations occuring when using the LCMColumn propagator than when using the two others. Therefore, we could somehow divide the time complexity bound of the LCMColumn propagator by $nItems$ for each generated closed frequent itemset I leading to a $\mathcal{O}(\mathcal{T}(I) * nItems)$ upper bound vs a $\mathcal{O}(\mathcal{T}(I) * nItems^2)$ upper bound for the GAC and the ClosedPattern-DC propagators. It explains the large performance difference between these propagators.

number of nodes for the GAC and ClosedPattern-DC propagators: Let's nSols be the number of solutions found by the solver. When using OscaR [22], the number of nodes for a GAC propagator is $2\times$ nSols -2 and we can observe in table 4.3 that my GAC propagator respects this equation. In the mail exchange I had with the designers of GCFCIM, they said that if their propagator is General Arc Consistant it should have $2\times$ nSols -1 nodes which is the case for the splice-1 and T10I4D100K datasets. For the mushroom dataset, the inconsistent number of nodes is explained by the fact that they use a slightly different mushroom dataset than mine. For the retail dataset no value of θ can explain their results.

[7] My teacher and me exchanged some mails with the designers of GCFCIM and the conclusion was that they will make their implementation public in september.

LCMColumn and LCMLine vs ClosedPattern-WC The pruning performed by these propagator should be equivalent but the experimental results are in contradiction with this theoretical truth. I think that in GCFCIM they made a mistake in the implementation of their ClosedPattern-WC propagator. This mistake gave them wrong experimental results in which the pruning done by the ClosedPattern-DC propagator is much better than the pruning of the ClosedPattern-WC propagator. I think that from this erroneous result they thought that the rule they invented[8] to make their propagator GAC was an interesting finding. In the introduction of their paper [19] they say "Experiments on several known large datasets show that our approach clearly outperforms the reified model used in CP4IM [3] and achieves scalability, which is a major issue for CP approaches. This is an expected result of the fact that ClosedPattern insures domain consistency, which guarantee to produce a closed pattern in a backtrack-free manner.". My theoretical analysis and experimental results go clearly against such conclusions (at least for the six different dataset I used).

dataset	*sup.*	LCM	Column	Line	GAC	DC	WC	CP4IM
splice-1	31	40.1	**51.1**	87.7	225.6	**400.1**	502.7	3753.2
mushroom	4	0.37	**1.1**	7.6	1.9	**6.3**	9.9	41.1
T10I4D100K	10	1.6	19.9	**11.2**	288.9	**905.1**	1 471.3	*OOM*
retail	44	0.4	16.3	**7.6**	306.9	2 645	**2 352.1**	*OOM*

Table 4.2: **Comparison of the execution times for the CFIMP.** The green columns contain the execution times of the LCMColumn (Column), LCMLine (Line) and GAC (GAC) propagators. The blue columns contain the execution time of the ClosedPattern-DC (DC) and ClosedPattern-WC (WC) presented in GCFCIM [19].

dataset	*sup.*	nSols	GAC	DC	Column	WC	$\frac{Column}{GAC}$
splice-1	31	6727378	13454754	13454755	13521502	754067617	1,0049
mushroom	4	186718	373434	407763	657410	7983119	1,7604
T10I4***	10	283398	566794	566795	723534	130673093	1,2765
retail	44	19699	39396	40229	39468	114345005	1,0018

Table 4.3: **Analysis of the number of nodes.** The green columns contain the number of nodes for the LCMColumn (Column) and GAC (GAC) propagators. The blue columns contains the number of nodes for the ClosedPattern-DC (DC) and ClosedPattern-WC (WC) presented in GCFCIM [19]. The last column contains the fraction $\frac{number\ of\ nodes\ for\ the\ LCMColumn\ propagator}{number\ of\ nodes\ for\ the\ GAC\ propagator}$.

4.4 Conclusion

I introduced in this thesis multiple new propagators for solving the FIMP and the CFIMP. The experiments show that they clearly outperform previous CP approaches. The newly introduced frequency success rule shows to be useful on the standard task of finding $\mathcal{F}$.

In this thesis, new insights have been provided with regard to the use of ideas coming from the specialized algorithms to solve the CP problems, and conversely, by building algorithms halfway between the two, referred here as "withSets algorithms".
It is easy to create a withSet algorithm based on my propagators and to transform it to a specialized algorithm, the opposite is also true, at least for the specialized algorithms based on Eclat [27, 25]. The theoretical analysis showed that the instructions performed when solving the FIMP (resp. the CFIMP) are nearly equivalent to the ones used when solving Eclat (resp. LCM).

In this thesis, the following bounds for my propagators were demonstrated:

- the computational time of generating each itemset $I \in \mathcal{F}$ is $\mathcal{O}(\mathcal{T}(I) \times nItems)$ and each $I \in \mathcal{C}$ is $\mathcal{O}(\mathcal{T}(I) \times nItems^2)$.

[8] This rule is equivalent to the simple GAC filtering rule presented in 13

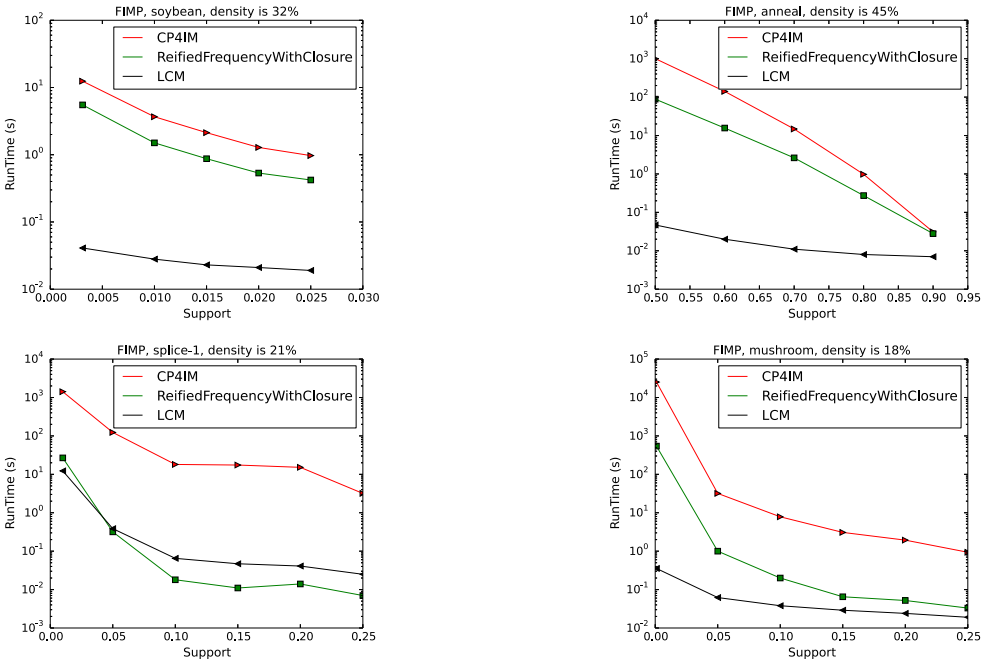
- the spacial complexity is $\mathcal{O}(nItems \times (nItems + nTrans)$ for both the FIMP and the CFIMP.

My theoretical analysis and the experimental results give strong evidences about the irrelevance of using the newly introduced filtering rule in GCFCIM [19].

The analysis of the experiments gives good insights about which implementation trick used by LCM [25, 26] could explain a large portion of the performance gap with my propagators. The experimental results on the splice-1 dataset show that using the reversible sparse bit-set data structure could be a good alternative over the data structures used by LCM when solving the FIMP or CFIMP on datasets with transactions very different to one another.

Given the limited framework of this thesis, further research would be required to enhance my algorithms with the implementation tricks present in LCM.

Frequent Itemset Mining Problem



Closed Frequent Itemset Mining Problem

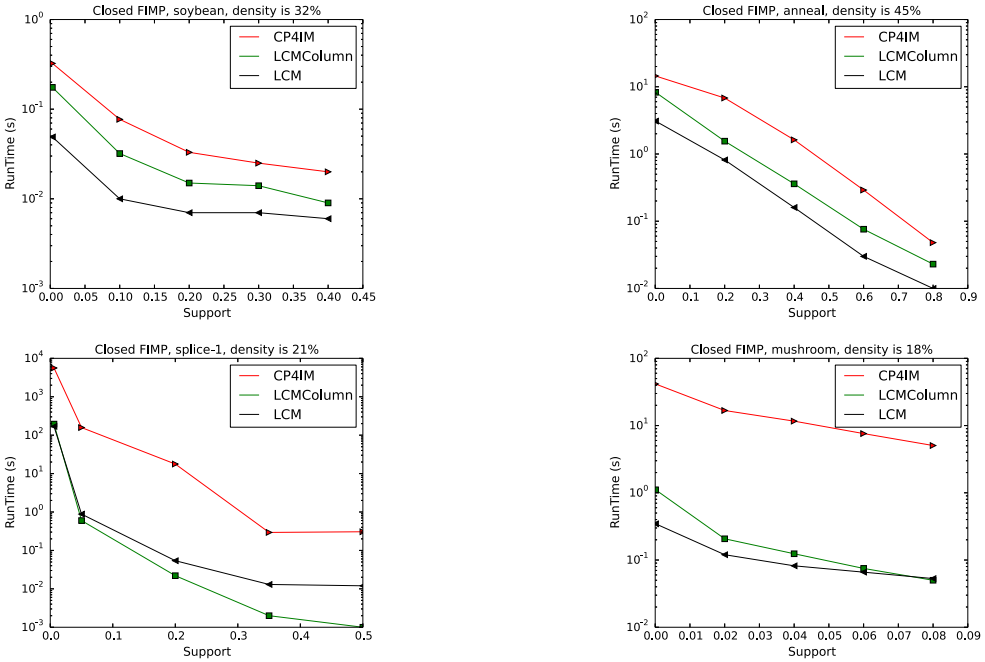


Figure 4.1: ReifiedFrequencyWithClosure and LCMColumn vs LCM vs CP4IM.

Bibliography

[1] Scala standard library 2.11.6. `http://www.scala-lang.org/api/current/index.html#scala.Long`.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[3] John OR Aoga, Tias Guns, and Pierre Schaus. An efficient algorithm for mining frequent sequence with constraint programming. *ECML-PKDD 2016*, 2016.

[4] Ferenc Bodon. A fast apriori implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03), volume 90 of Workshop Proceedings*, 2003.

[5] Francesco Bonchi and Claudio Lucchese. Extending the state-of-the-art of constraint-based pattern discovery. *Data Knowl. Eng.*, 60(2):377–399, February 2007.

[6] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):59–69, 1993.

[7] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. *SIGMOD Rec.*, 26(2):265–276, June 1997.

[8] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 204–212, New York, NY, USA, 2008. ACM.

[9] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[10] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. *CP2016*, 2016.

[11] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[12] Google. Google optimization tools, 2015. Available from `https://developers.google.com/optimization/`.

[13] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets, 2003.

[14] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining: A constraint programming perspective. *Artificial Intelligence*, 175(12-13):1951–1983, August 2011.

[15] Tias Guns, Siegfried Nijssen, and Luc De Raedt. k-Pattern set mining under constraints. *IEEE Transactions on Knowledge and Data Engineering*, 25(2):402–418, February 2013.

[16] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[17] Amina Kemmar, Samir Loudni, Yahia Lebbah, Patrice Boizumault, and Thierry Charnois. PREFIX-PROJECTION global constraint for sequential pattern mining. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, pages 226–243, 2015.

[18] Mehdi Khiari, Patrice Boizumault, and Bruno Crémilleux. *Constraint Programming for Mining n-ary Patterns*, pages 552–567. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[19] Mehdi Maamar, Nadjib Lazaar, Samir Loudni, and Yahia Lebbah. A global constraint for closed itemset mining. *CP2016*, 2016.

[20] Jean-Baptiste Mairy and Yves Deville. LINGI2365: Constraint programming. University Lecture, 2015.

[21] Siegfried Nijssen and Tias Guns. Integrating constraint programming and itemset mining. In *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part II*, pages 467–482, 2010.

[22] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[23] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014. Available from `http://www.choco-solver.org`.

[24] Guns Tias. *Declarative Pattern Mining using Constraint Programming*. PhD thesis, KATHOLIEKE UNIVERSITEIT LEUVEN, january 2002.

[25] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. *An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases*, pages 16–31. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[26] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, OSDM '05, pages 77–86, New York, NY, USA, 2005. ACM.

[27] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 326–335, New York, NY, USA, 2003. ACM.

[28] Mohammed J. Zaki and Ching jui Hsiao. Charm: An efficient algorithm for closed itemset mining. pages 457–473, 2002.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**