THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

Benjamin Negrevergne

Thèse dirigée par **Marie-Christine Rousset,**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'**EDMSTII.**

A Generic and Parallel Pattern Mining Algorithm for Multi-Core Architectures

Thèse soutenue publiquement le **29 Novembre 2011**,
devant le jury composé de :

Mots-clefs : Fouille de données, extraction de motifs fréquents, systèmes d'ensembles accessibles, algorithmes parallèles, calcul haute performance, évaluation de performances, architectures multi-cœurs.

Résumé : Dans le domaine de l'extraction de motifs, il existe un grand nombre d'algorithmes pour résoudre une large variété de sous problèmes sensiblement identiques. Cette variété d'algorithmes freine l'adoption des techniques d'extraction de motifs pour l'analyse de données. Dans cette thèse, nous proposons un formalisme qui permet de capturer une large gamme de problèmes d'extraction de motifs. Pour démontrer la généralité de ce formalisme, nous l'utilisons pour décrire trois problèmes d'extraction de motifs : le problème d'extraction d'itemsets fréquents fermés, le problème d'extraction de graphes relationnels fermés ou le problème d'extraction d'itemsets graduels fermés.

Ce formalisme nous permet de construire ParaMiner qui est un algorithme générique et parallèle pour les problèmes d'extraction de motifs. ParaMiner est capable de résoudre tous les problèmes d'extraction de motifs qui peuvent être décrit dans notre formalisme. Pour obtenir de bonne performances, nous avons généralisé plusieurs optimisations proposées par la communauté dans le cadre de problèmes spécifique d'extraction de motifs. Nous avons également exploité la puissance de calcul parallèle disponible dans les architectures parallèles.

Nos expériences démontrent qu'en dépit de la généricité de ParaMiner ses performances sont comparables avec celles obtenues par les algorithmes les plus rapides de l'état de l'art. Ces algorithmes bénéficient pourtant d'un avantage important, puisqu'ils incorporent de nombreuses optimisations spécifiques au sous problème d'extraction de motifs qu'ils résolvent.

Thèse effectuée au Laboratoire d'Informatique de Grenoble
Laboratoire LIG - UMR 5217
Maison Jean Kuntzmann - 110 av. de la Chimie - Domaine Universitaire de
Saint-Martin-d'Hères - BP 53
38041 Grenoble cedex 9 - France
Tel. : +33 (0)4 76 51 43 61 - Fax : +33 (0)4 76 51 49 85

Keywords: Data mining, pattern mining, accessible set systems, parallel algorithms, high performance computing, performance evaluation, multi-core architectures.

Abstract: In the pattern mining field, there exist a large number of algorithms that can solve a large variety of distinct but similar pattern mining problems. This variety prevent broad adoption of data analysis with pattern mining algorithms. In this thesis we propose a formal framework that is able to capture a broad range of pattern mining problems. We illustrate the generality of our framework by formalizing three different pattern mining problems: the problem of closed frequent itemset mining, the problem of closed relational graph mining and the problem of closed gradual itemset mining.

Building on this framework, we have designed ParaMiner, a generic and parallel algorithm for pattern mining. ParaMiner is able to solve any pattern mining problem that can be formalized within our framework. In order to achieve practical efficiency we have generalized important optimizations from state of the art algorithms and we have made ParaMiner able to exploit parallel computing platforms.

We have conducted thorough experiments that demonstrate that despite being a generic algorithm, ParaMiner can compete with the fastest ad-hoc algorithms.

Contents

Chapter 1

Introduction

In 1994, Agrawal and Srikant took the list of the sale records of a retail store and tried to discover knowledge about buying habits of customers. They did so by extracting recurring patterns from the list of receipts. These patterns consisted in sets of items frequently occurring together in customer baskets. For example the pattern $\{cereals, milk\}$ is likely to be a frequent one since most people buy milk with their cereals.

The frequent sets of items and their frequency can later be turned into *association rules* in order to know whether customers who bought milk are likely to buy cereals, or the opposite. In this context, the association rule: $cereals \rightarrow milk$ – stating that people buying cereals are likely to buy milk – is the most likely one. The association rules extracted from a retail store dataset are considered as valuable knowledge to rearrange items in shelfs and improve sales.

Back then Agrawal's work on pattern mining was pushed by the progress in bar-code technology and the increasing data storage capacities. Nowadays, recording devices are ubiquitous and almost every piece of information is digitally recorded. In the scientific field, collecting tremendous amounts of experimental data has become a standard. At the CERN in Geneva, the Large Hadron Collider produces over *15 million gigabytes* of experimental data *every year*. Other scientific applications in chemistry, meteorology and micro biology generate similar amounts of data.

For centuries, results of scientific experiments were carefully analyzed by experts with strong knowledge in the field. However, no human beings are capable of tackling the tremendous amounts of data generated by modern scientific experiments. Hence computer programs are now required to assist the domain experts to analyze their experimental data. These programs can help to discover relevant informations from an ocean of mostly noisy data.

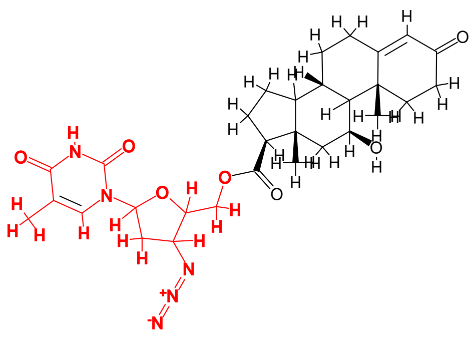
Since 1994, the pattern mining community has grown and gain in diversity. It is today an important topic of computer science addressed by many scientific publications in international conferences. In order to harness the diversity of the pattern mining problems, researchers have proposed various techniques that fall into two categories: the statistical approach and the structural approach.

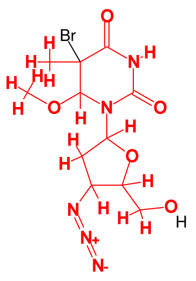
The traditional approach to handle large amounts of data is to use statistics. Nowadays, statistical approaches for pattern mining are widely used to mine for knowledge in very large graphs such as the web, social networks or even gene networks. In this context, so called statistical graph mining algorithms are used to compute general informations such as statistical distributions of the node degrees or such as typical topologies of small clusters[KTF09, MN08]. This approach allows to infer global properties about the input graph and spot representative or dissimilar patterns in this graph. Statistical approaches can tackle very large input datasets and provide informative knowledge over the dataset.

The other approach is in the direct vein of Agrawal's pattern mining algorithm. It consists in extracting algebraic substructures such as sets but also sequences or graphs from the input dataset. Relevant substructures are identified according to whatever property is meaningful to the application. In most applications the frequency of a pattern is the basic property to discriminate irrelevant patterns, but not always. In this thesis we exclusively deal with this approach of pattern mining so called *structural pattern mining.*

1.1 Structural pattern mining

Pushed by the interesting results obtained in the context of market basket analysis, experts from other scientific domains with large set of experimental data, started to show interest in pattern mining techniques. In collaboration with researchers from the pattern mining community, they have elaborated new algorithms to extract meaningful patterns from their datasets.


(a) anti-HIV active molecule 1


(b) anti-HIV active molecule 2

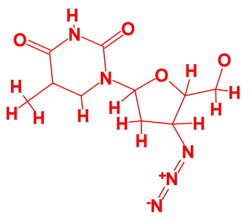

(c) anti-HIV molecular pattern

Figure 1.1: *Azidothymidine* (c) is a molecular pattern that occurs in many anti-HIV molecules such as (a) and (b).

For example, chemical engineers were interested in extracting characteristic substructures from datasets of chemical compounds. In [KDRH01] Kramer et al. have mined a dataset of molecules represented as graphs. Those molecules had been previously tested for their capability to protect human cells from the HIV infection (an example of such molecules

Date	Temp. (°C)	Pressure (hPa)	Wind direction	Wind speed (km/h)
May 26 2011	**17.6**	**1021.20**	300	**57**
May 27 2011	18.5	1021.30	310	57
May 28 2011	**20.4**	**1018.20**	320	**51**
May 29 2011	**28.5**	**1012.80**	110	**26**
May 30 2011	18.9	1014.80	290	67
May 31 2011	**16.5**	**1026.50**	310	**77**

Figure 1.2: Meteorological records of climate taken from May 25 2011 to May 31 2011, at 6pm, in France. The gradual pattern $\{T°\uparrow, P\downarrow, Wind\uparrow\}$ is observable day 26, 28, 29 and 31.

is shown in figure 1.1 (a) and (b)). The goal was to extract the substructures commonly occurring in anti-HIV molecules (e.g. figure 1.1 (c)). Although in [KDRH01] Kramer et al. were only able to extract linear fragments of this compound, other approaches developed by Inokuchi et al. in [IWM00] and Yan et al. in [YH02] are capable of extracting full graph patterns such as in figure 1.1 (c). Graph mining is a useful application to analyze many other types of dataset such as web logs or gene networks datasets.

Yet some datasets were still out of reach. Recently in [DJLT09], Di-Jorio et al have conducted work in order to analyze datasets consisting in large amounts of quantitative data. The goal is to extract correlated variations of quantitative values. For example, the dataset in figure 1.2 is a list of records from various climatic sensors. Given this dataset, one may want to know if there exists any correlation between the measured values. A fine analysis of this dataset reveals that in 67% of the records, when the temperature increases the pressure decreases and the wind speed increases. *Gradual pattern mining* is able to extract such co-variations formalized as $\{T°\uparrow, P\downarrow, Wind\uparrow\}$. Di-Jorio in[DJLT09] have worked on gradual pattern mining in order to extract co-variations in datasets with hundreds of attributes and thousands of records. This type of pattern is helpful to mine survey database, data streams or network sensors readings.

Because of the diversity of the datasets and the patterns to extract, most people having interest in pattern mining have developed their own ad-hoc algorithm adapted to their needs. Although the problems addressed by those algorithms look quite different, they are all different instances of the same problem: structural pattern mining. In this thesis, we define the problem of structural pattern mining as follows:

Given a dataset, a pattern *structure definition* and a pattern *selection criterion*, extract the set of patterns made up of all the structures occurring in the dataset satisfying the selection criterion.

- The *dataset* is the input data to mine. In the context of basket market analysis, the dataset is a list of receipts where each receipt is a set of items purchased together. It can also be a set of molecules, when mining for substructures in chemical compounds, or the list of records from climatic sensors when mining for gradual patterns.

- The *pattern structure definition* specifies the structure of the patterns to extract. It is set according to dataset structure and the application needs. For example, in the context of market basket analysis, patterns are subsets of the set of available items. When mining molecular compounds, the patterns are labeled graphs where vertices are labeled with chemical elements names in a given set and edges are labeled with covalent bound types. The pattern structure definition intentionally defines the set of all the *candidate patterns.*

- The *pattern selection criterion* is formulated by the application experts. A candidate pattern must meet this criterion in order to be an actual pattern. It discriminates patterns relevant to the application from irrelevant ones. The frequency is commonly used to discard irrelevant patterns, however it can be combined with other requirements such as *the pattern must be a connected graph* (see. Section 2.2.2).

When it's clear from context, pattern mining will be used as a shortcut for structural pattern mining.

Pattern mining is a very difficult problem that raises many important challenges such as:

- Encoding and preprocessing of raw data.
- Handling the combinatorial explosion of the number of candidate patterns.
- Filtering and analyzing patterns that are extracted.

In this thesis we address the second challenge by generalizing in a principled way several optimizations and by incorporating them into a generic and parallel pattern mining algorithm.

1.2 Scope of this thesis

The standard method to output the set of patterns is to *generate* candidate patterns with respect to the pattern structure definition, and then *test* if they occur in the dataset *and* satisfy the selection criterion. However the number of candidate patterns to test is theoretically tremendous. For example if a retail store, selling 1000 distinct items, wants to mine its sale records to extract frequent sets of items, the number of candidate patterns is as big as $2^{1000}(\sim 10^{300})$. Generating and testing this amount of candidate patterns is not feasible in practice.

To reduce the number of candidate patterns and simplify the process of testing them, most pattern mining algorithms were designed to take advantage of the specificities of patterns and datasets to mine.

In [AS94], Agrawal and Srikant address the problem of mining frequent subsets by exploiting the anti-monotonicity property of the frequent sets. This property states that any set including an infrequent subset is also infrequent. Based on this property, Agrawal's algorithm avoids generating all the candidate patterns including one or more infrequent subset. This principle was later adapted to mine other types of frequent patterns such as frequent graphs in [IWM00].

In [HPY00], Han et al. have proposed FP-GROWTH, a depth-first-search recursive algorithm which starts from a frequent set and efficiently computes the frequent supersets of this set. In FP-GROWTH items occurring in the dataset are stored in a prefix tree like structure called *FP-tree.* FP-GROWTH avoids costly database scans by building for each recursive call a new FP-tree representing only the sub-dataset relevant to the recursive call being processed.

Since larger frequent sets are less represented in the dataset, FP-trees get smaller as the algorithm gets deeper in the recursive calls. This approach allowed FP-GROWTH to tackle big datasets with a divide and conquer approach. An improved version of this technique is used in the fastest frequent set mining algorithms, LCM[UKA04].

Reducing the number of candidate patterns may not be sufficient to tackle large datasets because the number of patterns can be large as well. In order to avoid the combinatorial explosion of the number of patterns, recent algorithms focus on the extraction of *closed patterns* only. Closed patterns were introduced by Pasquier et al. in [PBTL99] in the context of mining frequent sets. A frequent set is closed if and only if there exist no strict supersets occurring in the dataset with an equal frequency. Mining closed frequent sets represents no loss of information over mining frequent sets. It is indeed possible to derive the identity and the frequency of any frequent set from the set of closed sets. It is also more concise; in practice the number of closed frequent sets can be orders of magnitude smaller than the number of frequent sets. Closed pattern mining is a major issue to reach efficiency in pattern mining, thus various types of closure operator were defined for other types of patterns such as trees, graphs and even gradual patterns.

In order to tackle bigger datasets researchers have worked on pattern mining algorithms able to exploit parallel architectures. The problem has attracted more attention since the parallelism became truly ubiquitous with the advent of *multi-core architectures*. Indeed, almost every processor available nowadays embeds two or more computing *cores* providing true parallelism at low cost. However naive parallelizations of pattern mining algorithms perform poorly on multi-core platforms due to load imbalance or excessive memory consumption. The problem of designing pattern mining algorithms for multi-core architectures has been addressed by several research papers in the context of frequent set mining [LOP07, NTMU10], tree mining [TP09] or graph mining [BPC06]. These papers have shown that dynamic work distribution strategies and customized data structures can reduce the load imbalance and the memory consumption. Both are required to achieve good scalability with the number of cores used to run the algorithm.

The additional computational power available in multi-core architectures, together with the algorithmic improvements mentioned above, should allow to tackle many real world datasets. However, very few pattern mining algorithms fully integrate these research works. Indeed, the lack of an unified definition for pattern mining problems make any improvement hardly transposable from one pattern mining problem to another. In addition most of these improvements do not coexist well together. For example, enumeration of closed patterns breaks the algorithmic properties that allowed FP-GROWTH to tackle the problem with a divide and conquer approach. Without the divide and conquer approach, the problem must be handled globally, leading to unwanted communication and synchronization when it comes to parallel algorithms. As a consequence, most application experts do not have access to adequate and efficient algorithms to mine their datasets.

1.3 Contributions of this thesis

In this thesis, we aim at generalizing the main improvements proposed over the years to mine large specific datasets into a single generic algorithm. This includes efficient pattern enumeration strategies, closed pattern mining, divide and conquer methods to tackle the dataset and parallelism.

Our contributions are the following:

A generic definition of the structural pattern mining problem. Following Boley et al. in [BHPW07] or Arimura and Uno in [AU09], we define the problem of enumerating closed patterns as the problem of enumerating sets satisfying constraints in a set system. This definition comes with the guarantee that the problem can be solved in polynomial delay and space, if the underlying set system is *strongly accessible.* We extended this definition to pattern mining by formalizing and incorporating the notion of patterns *occurring* in a dataset. We show that this definition is sufficient to capture many different pattern mining problems such as frequent itemset mining, gradual pattern mining, relational graph mining.

A generic and parallel algorithm for structural pattern mining. PARAMINER is an algorithm able to solve any pattern mining problem that can be expressed according to the definition mentioned above. In order to tackle large-scale datasets PARAMINER, successfully addresses several important issues of pattern mining. It efficiently solves the problem of parallel enumeration of closed patterns. It also generalizes several of the most important optimizations introduced in ad-hoc algorithms such as *database reduction.* PARAMINER is then proven to be correct for any pattern mining algorithm expressed according to our definition.

A parallelism engine for multi-core architectures adapted to pattern mining algorithms. Load imbalance and high memory consumption are two important problems observed with almost any pattern mining algorithm. We address these problems by proposing MELINDA, a parallelism engine for pattern mining algorithms. MELINDA is able to accurately drive the execution of a parallel pattern mining algorithm according to a *strategy* expressed with abstract concepts. It was designed based on an extensive study conducted over several pattern mining applications involving different types of patterns and datasets. Although MELINDA is the parallelism engine in use in PARAMINER, it was designed independently and is used in other parallel pattern mining algorithms[NTMU10].

1.4 Outline

This thesis is organized as follows:

- Chapter 2 provides our definition of pattern mining and the formal background on which it is built.
- Chapter 3 describes PARAMINER, our generic and parallel algorithm for pattern mining.
- Chapter 4 is an experimental validation of PARAMINER. In this chapter, we report on thorough experiments that we have conducted in order to evaluate PARAMINER's efficiency.

- We present in Chapter 5 the state of the art in generic pattern mining and the most recent works in parallel pattern mining.

- We conclude and present several perspectives in Chapter 6.

Chapter 2

A generic framework for mining patterns in a dataset

Contents

The naive approach to output a set of patterns is to generate all the *candidate patterns* matching the pattern structure definition, then to test which candidate patterns satisfy the selection criterion. However generating and testing the set of candidate patterns is intractable because the number of structures is combinatorial with the number of possible structure components. For example, the number of candidate itemsets that can be generated over a set of n distinct items is as big as 2^n. The number of candidate patterns is even larger with other pattern structure definitions such as sequence-based or graph-based structure definitions.

In most pattern mining problems, the candidate patterns can be partially ordered by an inclusion relation. Thus the set of candidate patterns has a directed acyclic graph (a *DAG*) structure. Even if this DAG is too big to be entirely constructed, it is an important resource to efficiently explore the set of candidate patterns. Indeed, the generation of large amounts of non-meaningful candidate patterns can be avoided given the results of tests performed on a small number of candidate patterns.

The structured exploration of the set of candidate pattern is however insufficient to achieve reasonable performances when the number of meaningful patterns itself is large. In order to cope with this problem, Pasquier et al.([PBTL99]) have proposed to mine *closed patterns* only. The set of closed patterns is a lossless representation of the set of all the meaningful patterns, that can be one to several order of magnitude smaller.

Exploring the DAG structure formed by the candidate patterns in order to extract closed patterns only is a complex task. Existing pattern mining algorithms are typically driven by an *enumeration strategy* to ensure exhaustive and non redundant enumeration of all the closed patterns. Until recently the work on enumeration strategies was lacking theoretical foundations. Enumeration strategies for most pattern mining algorithms were designed in an ad-hoc way, based on specific properties of the search space and the patterns mined.

In this chapter we first provide a generic formal framework to address the problem of pattern mining in which the patterns are represented as sets of elements. We show how it is able to capture several assorted pattern mining problems such as relational graph mining or gradual pattern mining. We then present the work of Boley et al.([BHPW10]) and Arimura and Uno([AU09]), on closed pattern enumeration, and show how it can be used to define an efficient and generic enumeration strategy for enumerating closed patterns in a parallel algorithm. This will introduce the ParaMiner algorithm that will be extensively presented in the next chapter.

2.1 Formal background

2.1.1 Dataset

In our setting, any dataset is defined as a sequence of transactions over a finite *ground set* of elements.

Definition 2.1 (Dataset)
Given a ground set E, a dataset $\mathcal{D}_E$ *is sequence of transactions $[t_1, t_2, \dots, t_n]$ where each transaction is a subset of the ground set E. The set of transaction indices is called the* tid set *and is denoted $T_{\mathcal{D}_E}$.*

We also use the following notations:

- $\mathcal{D}_E(i)$, with $i \in T_{\mathcal{D}_E}$ denotes the transaction t_i in $\mathcal{D}_E$
- $|\mathcal{D}_E|$ denotes the number of transactions in $\mathcal{D}_E$
- $||\mathcal{D}_E|| = \sum_{i=1}^{i \leq |\mathcal{D}_E|} |\mathcal{D}_E(i)|$ denotes the size of $\mathcal{D}_E$.

Many application datasets can be directly stored in this form. For instance, in the context of market basket analysis [AS94], if the ground set is the set of all available items such as $E = \{apple, beer, chocolate \dots\}$, one can store each receipt, that is each set of items purchased together, as a transaction. The set $\mathcal{D}_E$ of all the transactions is a dataset for this application. An example is shown in Figure 2.1.

receipt #	1	2	3
	apple beer choco- late	apple beer	apple choco- late

$\Rightarrow$

$E = \{apple, beer, chocolate\}$

$\mathcal{D}_E =$
$[\{apple, beer, chocolate\},$
$\{apple, beer\},$
$\{apple, chocolate\}]$

Figure 2.1: A dataset for frequent itemset mining in the context of market basket analysis. The ground set E is the set of available items, each transaction in $\mathcal{D}_E$ is a set of items purchased together.

In the context of gene network analysis [YZH05], given a set of genes denoted G, the ground set E is the cartesian product $G \times G$ of pairs of genes representing all the possible gene interactions in a given set G of genes. In the dataset, each transaction is a set of interactions observed during one experiment. An example is shown in Figure 2.2.

2.1.2 Candidate patterns

In our setting a *candidate pattern* is simply defined as a subset of the ground set.

Definition 2.2 (candidate pattern)
A candidate pattern is any subset of the ground set E.

The support set of a candidate pattern is the sub-dataset made of the transactions including this candidate pattern.

Definition 2.3 (support set)
Given a dataset $\mathcal{D}_E$ and a candidate pattern $X \subseteq E$, the support set of X denoted $\mathcal{D}_E[X]$ is the sequence of transactions in $\mathcal{D}_E$ including X.

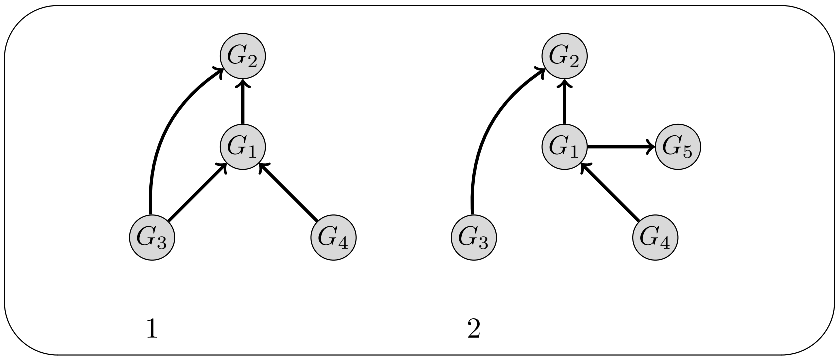
Definition 2.4 (tid support set)
In addition we define the tid support set *of X, denoted $\mathcal{D}_E[\![X]\!]$ as the set of indices of the transactions in $\mathcal{D}_E[X]$: $\mathcal{D}_E[\![X]\!] = \{t \in T_{\mathcal{D}_E} | X \subseteq D_E(t)\}$.*

If a candidate pattern $X \subseteq E$ has a non empty tid set in a dataset $\mathcal{D}_E$, we say that X *occurs* in $\mathcal{D}_E$.

We will use the following proposition in several places: the support set of the union of two candidate patterns is the intersection of the support sets of each candidate pattern.

Proposition 2.1
For every $X, Y \subseteq E$: $\mathcal{D}_E[X \cup Y] = \mathcal{D}_E[X] \cap \mathcal{D}_E[Y]$.

Proof: From Definition 2.3, the support set of $X \cup Y$ is the set of transactions t such that $X \cup Y \subseteq t$. The candidate pattern $X \cup Y$ is included in a transaction if and only if



$$\Downarrow$$

$$E = \{(G_1, G_2), (G_1, G_3), \ldots, (G_5, G_4)\}$$

$$\mathcal{D}_E = [\{(G_1, G_2), (G_3, G_1), (G_3, G_2), (G_4, G_1)\}, \{(G_1, G_2), (G_1, G_5), (G_3, G_2), (G_4, G_1)\}]$$

Figure 2.2: A dataset of relational graphs, each node is a gene, there is an arc between two genes G_X, and G_Y when the gene G_X interacts (i.e. has an influence) with the gene G_Y. The ground set E is the set of all the possible interactions between the set of genes, each transaction in $\mathcal{D}_E$ is a set of interactions between genes.

X and Y are both included in this transaction. Hence the support set $\mathcal{D}_E[X \cup Y]$ is the set of transactions that are in $\mathcal{D}_E[X]$ and in $\mathcal{D}_E[Y]$. $\mathcal{D}_E[X \cup Y] = \mathcal{D}_E[X] \cap \mathcal{D}_E[Y]$.

2.1.3 Selection criterion

The selection criterion is specified according to the application needs. It provides a way to discriminate candidate patterns meaningful to the application from irrelevant ones. The selection criterion is defined as follows:

Definition 2.5 (selection criterion)
The selection criterion *denoted Select is a user-specified predicate. Given a candidate pattern $X \subseteq E$ and a dataset $\mathcal{D}_E$, the selection criterion Select$(X, \mathcal{D}_E)$ returns true if and only if the candidate pattern X is to be retained as a pattern in $\mathcal{D}_E$.*

In many pattern mining applications, the selection criterion is based on frequency in order to extract frequent or infrequent patterns from the dataset. For example in applications such as the basket market analysis, the patterns to be retained are the ones occurring in at least ε transactions. In this pattern mining problem, the selection criterion can be specified as follows: $Select(X, \mathcal{D}_E) \equiv |\mathcal{D}_E[X]| \geq \varepsilon$.

In other applications, other properties may be involved in the selection criterion. For example, given a gene network dataset such as the one presented in Figure 2.2 the connectivity of the final graph-pattern is an important concern. Indeed in Figure 2.2, although the candidate pattern $\{(G_4, G_1), (G_3, G_2)\}$ (Figure 2.3) is frequent, it does not represent

any connected graph and is therefore not meaningful to model a gene interaction network. Thus one can add the connectivity constraints in the selection criterion:
$Select(X, \mathcal{D}_E) \equiv |\mathcal{D}_E[X]| \geq \varepsilon \wedge X$ *is a connected set of arcs.*

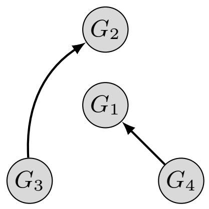


Figure 2.3: The candidate pattern $\{(G_4, G_1), (G_3, G_2)\}$ is not a pattern because it is not connected.

We define the concept of meaningful *pattern* as follows.

Definition 2.6 (Meaningful pattern)
Given a dataset $\mathcal{D}_E$ built over a ground set E, a selection criterion Select, a candidate pattern $X \subseteq E$ is a meaningful pattern in $\mathcal{D}_E$ if and only if:

1. *X occurs in $\mathcal{D}_E$*
2. *X satisfies the selection criterion in the dataset: $Select(X, \mathcal{D}_E) = true$.*

When it is clear from context the term *pattern* will stand for *meaningful pattern.*

We denote by $\mathcal{F} \subseteq 2^E$ the set of meaningful patterns.

2.1.4 Closed patterns

Although each pattern in $\mathcal{F}$ is meaningful, the whole set of meaningful patterns may provide redundant information. Closed patterns were proposed by Pasquier et al. in [PBTL99] in the context of frequent itemset mining, to reduce the redundancy among the set of the patterns. The set of closed pattern is a lossless representation of the set of patterns.

For example, consider the itemsets $I_1 = \{chocolate\}$ and $I_2 = \{beer, chocolate\}$ occurring in the dataset $\mathcal{D}_E$ in Figure 2.1. I_1 and I_2 are both frequent for a given support threshold $\varepsilon = 2$. However $I_2 = \{beer, chocolate\}$ is frequent *implies* that $I_1 = \{chocolate\}$ is frequent, hence the if I_2 is in $\mathcal{F}$, adding I_1 provides no additional information.

In the context of frequent itemset mining, a frequent itemset I is closed in a dataset $\mathcal{D}_E$ if and only if it is the biggest pattern with the support set $\mathcal{D}_E[I]$. Considering the two itemsets $I_1 = \{chocolate\}$ and $I_2 = \{beer, chocolate\}$ from $\mathcal{D}_E$ in Figure 2.1, I_1 and I_2 are both frequent for $\varepsilon = 2$, and they both share the same support set $\{\mathcal{D}_E(1), \mathcal{D}_E(3)\}$, however I_1 is not closed because there exists I_2 such that $I_1 \subset I_2$ and $\mathcal{D}_E[I_1] = \mathcal{D}_E[I_2]$. I_2 is closed.

The principle of closed patterns can be extended to other types of patterns. For example, the two graph patterns P_1 and P_2 in Figure 2.4, are both connected and frequent subgraphs in the graph dataset from Figure 2.2 (with $\varepsilon = 2$). P_1 and P_2 share the same support set, however P_2 is a superset of P_1 hence P_1 is not a closed pattern. P_2 is the biggest pattern in the dataset with the support set $\{\mathcal{D}_E(1), \mathcal{D}_E(2)\}$, hence it is closed.

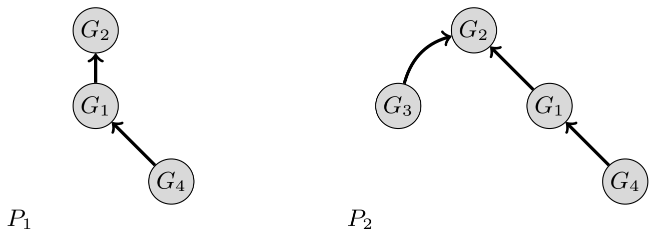


Figure 2.4: Two 2-frequent and connected subgraphs extracted from the graph dataset in Figure 2.2. Only (b) is closed.

In our setting, we define a closed pattern as follows:

Definition 2.7 (Closed pattern)
A meaningful pattern $P \in \mathcal{F}$ is closed *if and only if there does not exist any strict superset of P that is a pattern in $\mathcal{F}$ with the same support set.*

We also define the closure of a pattern as follows:

Definition 2.8 (Closure of a pattern)
For a pattern $P \in \mathcal{F}$, a closed pattern $Q \in \mathcal{F}$ is a closure of P if and only if $P \subseteq Q$ and $\mathcal{D}_E[P] = \mathcal{D}_E[Q]$.

Proposition 2.2
Every pattern in $\mathcal{F}$ admits at least one closure.

Proof: We suppose that there exists a pattern P that does not have a closure. From Definition 2.8, $P \neq \emptyset$ and there does not exist a closed pattern Q such that $P \subseteq Q$ and $\mathcal{D}_E[P] = \mathcal{D}_E[Q]$. Therefore P itself is not closed or else $Q = P$ would be a closure of P. From the definition of a closed pattern, if P is not closed, there exist at least one strict superset Q of P that is a pattern in $\mathcal{F}$ such that $\mathcal{D}_E[P] = \mathcal{D}_E[Q]$. Therefore P admits Q as a closure which contradicts the initial statement that P admits no closure. Hence there exists a closure for every $P \in \mathcal{F}$. $\square$

This definition does not guarantee the uniqueness of the closure of a pattern. However, in Theorem 2.1, we exhibit a sufficient condition for guaranteeing it. This condition express that a given property $(P1)$, depending on the dataset $\mathcal{D}_E$ and the definition of the set $\mathcal{F}$ of patterns, holds.

Theorem 2.1
Let $(P1)$ be the following property:

$(P1)$ *For every two patterns Q and $Q' \in \mathcal{F}$,*
if:
i) $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$ *(Q and Q' have the same support set)*
ii) *there exists $Z \subseteq Q \cap Q'$ such that $Z \neq \emptyset$ and $Z \in \mathcal{F}$ ($Q \cap Q'$ includes a non empty pattern)*
then $Q \cup Q' \in \mathcal{F}$ *($Q \cup Q'$ is also a pattern)*

If $(P1)$ is satisfied, then every pattern has a unique closure.

Proof: Suppose that there exists a pattern P that admits two closures denoted Q and Q'. From Definition 2.8, it is guaranteed that: (1) Q and Q' are patterns, (2) $P \subseteq Q$ and $P \subseteq Q'$, and (3) $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$. Hence, according to Property $(P1)$ (applied with $Z = P$), $Q \cup Q'$ is also a pattern. Since $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$, $\mathcal{D}_E[Q \cup Q'] = \mathcal{D}_E[Q] \cap \mathcal{D}_E[Q'] = \mathcal{D}_E[Q] = \mathcal{D}_E[Q']$. However, if Q is closed, there exist no strict super set of Q that admit the same support set. Hence $Q = Q \cup Q'$. The same goes for Q', and thus $Q = Q'$. Therefore the closure of any pattern P is unique. $\square$

Other characterizations of the closure uniqueness have been published. In particular the property of *confluence* has been introduced by [BHPW10]: A set of patterns is confluent if and only if the union of two patterns having a non empty pattern in their intersection is also a pattern.

Definition 2.9 (Confluence [BHPW10])
Given a set $\mathcal{F}$ of patterns defined over a ground set E, $\mathcal{F}$ is confluent if and only for all $I, X, Y \in \mathcal{F}$ with $\emptyset \neq I \subseteq X$ and $I \subseteq Y$, it holds that $X \cup Y \in \mathcal{F}$.

It is shown in Theorem 7 from [BHPW10] that the set of patterns defined over a ground set is confluent if and only if the closure is well defined (exists and is unique) for **every** dataset defined over the same ground set.

It may seem that Theorem 7 is stronger than Theorem 2.2. This is not the case. Our Theorem 2.2 applies to most of existing pattern mining problems whereas Theorem 7 does not apply on the most basic pattern mining problem that is the problem of frequent itemset mining. Indeed this problem does not verify the confluence property even thus the closure is unique.

This does not contradict the fact that the confluence property is a necessary condition in Theorem 7 for the existence and uniqueness of closure when $\mathcal{F}$ is defined independently of the dataset.

For example:
Let $E = \{a, b\}$ be a ground set and $\mathcal{F} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$, $\mathcal{D}_E = [\{a\}, \{b\}]$. The closure of $\{a, b\}$ does not exist. This is due to the fact that in contrast with our definition, the patterns defined in [BHPW10] are not required to occur in the dataset.

The Property $(P1)$ that we have introduced in Theorem 2.2 is more specific than the confluence property but applies to most pattern mining problems encountered in practice as we will show in Section 2.2 including the problem of frequent itemset mining. Therefore this Property $(P1)$ is the right property to characterize the pattern mining problem for which the closure is unique.

In this case, we denote $Clo(P, \mathcal{D}_E)$ the closure operator that, for every $P \in \mathcal{F}$, associates a pattern with its closure. We also denote $\mathcal{C}$, the set of closed patterns that are the closure of a pattern in $\mathcal{F}$: $\mathcal{C} = \{Q \in \mathcal{F} | \exists P \in \mathcal{F}, Q = Clo(P, \mathcal{D}_E)\}$. Algorithm 1, is a generic algorithm that computes the closure of any pattern P by augmenting P with elements from the intersection of the transactions in the support set of P.

Algorithm 1 A generic closure operator

- **Require:** a ground set E, a dataset $\mathcal{D}_E$, a selection criterion *Select* and a pattern P.
- **Ensure:** returns the unique closure Q of P.

```
1: $Q \leftarrow P$
2: //while there exists e such that $Q \cup \{e\} \in \mathcal{F}$
3: while $\exists e \in \cap\mathcal{D}_E[P] \setminus Q$ such that $Select(Q \cup \{e\}, \mathcal{D}_E)$ do
4:     $Q \leftarrow Q \cup \{e\}$
5: end while
6: return $Q$
```

In practice, this algorithm may be very costly and can be replaced by more efficient specific algorithms relying on characterizations of the closure operator that exploit the specificities of the problem.

2.1.5 Formal problem statement

In this thesis, we address the problem of closed pattern mining when the closure is unique and computable by a closure operator *Clo*. It can be stated as follows.

Definition 2.10 (Closed pattern mining problem)
Given a ground set E, a dataset $\mathcal{D}_E$, a selection criterion, Select and closure operator Clo extract from $\mathcal{D}_E$ all the closed patterns, that is any set $P \subseteq E$ such that:

1. *P occurs in $\mathcal{D}_E$ ($\mathcal{D}_E[P] \neq \emptyset$)*

2. *$Select(P, \mathcal{D}_E) = true$*

3. *$Clo(P, \mathcal{D}_E) = P$.*

2.2 Formalization of different specific pattern mining problems

In this section, we show how the generic framework presented in the former section can capture several existing pattern mining problems by using an adequate encoding. This encoding can be direct (e.g. frequent itemset mining) or more complex (e.g. gradual itemset mining).

2.2.1 Mining closed frequent itemsets (FIM)

Ground set & dataset: Encoding a frequent itemset input dataset is direct and has been explained in Figure 2.1.

Selection criterion: A subset of the ground set E is a pattern if occurs in at least ε transactions (for a given constant ε). For any $P \subseteq E$, $\mathit{Select}(P, \mathcal{D}_E) \equiv |\mathcal{D}_E[P]| \geq \varepsilon$.

Theorem 2.2 (Closure uniqueness for the FIM problem)
The Property $(P1)$ is satisfied for the FIM problem.

The proof relies on the following Lemma, which will be further reused.

Lemma 2.1
Let Q and Q' be two candidate patterns such that $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$. Let us show that if Q or Q' is frequent, then $Q \cup Q'$ is also frequent.

Proof: If Q is frequent $|\mathcal{D}_E[Q]| \geq \varepsilon$ and $|\mathcal{D}_E[Q]| \geq \varepsilon$. From Proposition 2.1 $\mathcal{D}_E[Q \cup Q'] = \mathcal{D}_E[Q] \cap \mathcal{D}_E[Q']$, hence $\mathcal{D}_E[Q \cup Q'] = \mathcal{D}_E[Q] \geq \varepsilon$. □

The proof the Theorem 2.2 is a direct consequence of the Lemma: For every two patterns Q and Q' such that $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$, $Q \cup Q'$ is frequent, hence $Q \cup Q'$ is a pattern and $(P1)$ holds. □

Closure operator: As it has been proved in [PBTL99], the closure of a pattern P is the intersection of the transactions in the support set of P: $\mathit{Clo}(P, \mathcal{D}_E) = \bigcap \mathcal{D}_E[P]$.

2.2.2 Mining closed frequent connected relational graphs (CRG)

A relational graph is a labelled graph in which all the node labels are distinct. Such graphs can represent gene networks as well as social networks [YZH05]. An example of a relation graph dataset has been presented in Figure 2.2.

The problem of mining frequent connected relation graph can be stated as follows: Given a set of vertices V and a set of relational graphs $G_1, \ldots, G_n$ where each G_i is a relational graph (V, E_i) defined using the nodes in V, extract the connected sub-graphs occurring in at least ε input graphs.

The problem of extracting frequent closed connected relational graphs can be captured in our setting as follows:

Ground set: The ground set E is a set of pairs in $V \times V$, each pair is used to represent an edge connecting two nodes in V.

Dataset: The dataset $\mathcal{D}_E = [t_1, \ldots, t_n]$ is a sequence of transactions where for all $i \in [1, n]$, the transaction $\mathcal{D}_E(t_i)$, represents the input graph G_i. Each element in the transaction t_i is a pair representing an edge in G_i.

Selection criterion: Given a pattern G, $\mathit{Select}(G, \mathcal{D}_E)$ returns true if and only if:

- G is connected.
- $|\mathcal{D}_{Eg}[G]| \geq \varepsilon$ (for a given constant ε)

Theorem 2.3 (Closure uniqueness for the CRG problem)
The Property $(P1)$ is satisfied for the CRG *problem.*

Proof: Let Q and Q' be two patterns in $\mathcal{F}$ such that $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$. and let Z be a pattern in $\mathcal{F}$ such that $Z \subseteq Q \cap Q'$ and $Z \neq \emptyset$. As patterns, Q, Q' and Z are connected set of edges. Q is connected implies that for every edges $e, e' \in Q$, there exists a path $[e, \ldots, e']$, connecting e and e'. Let e, be an edge in $Q \cap Q'$ (This edge always exists because $Q \cap Q' \supseteq Z \neq \emptyset$) then $\forall e' \in Q$ and $e'' \in Q'$, there exists a path $[e', \ldots, e, \ldots, e''] \in Q \cup Q'$. Hence $Q \cup Q'$ is connected. From Lemma 2.1, we have that $Q \cup Q'$ is frequent in $\mathcal{D}_E$. $Q \cup Q'$ is a pattern. The Property $(P1)$ is satisfied. □

Closure operator: The closure of a graph P is the set of edges connected to P occurring in every transactions of the support set of P. It can be computed with Algorithm 2. This algorithm is specialization of Algorithm 1: in Line 3, we only perform a connectivity test rather than a complete call to *Select*, because by construction $Q \cup \{e\}$ is frequent.

Algorithm 2 Closure operator for the CRG problem

- **Require:** a graph pattern P and a dataset $\mathcal{D}_E$
- **Ensure:** returns the unique closure Q of P.

```
1: $Q \leftarrow P$
2: //while there exists e such that e is connected to Q
3: while $\exists e \in \cap\mathcal{D}_E[P] \setminus Q$ such that Q is connected e do
4:     $Q \leftarrow Q \cup \{e\}$
5: end while
6: return Q
```

2.2.3 Mining closed frequent gradual itemsets (GRI)

The problem of mining gradual itemsets consists in mining attributes co-variations in numerical datasets [ALYP10]. Consider the numerical database Figure 2.1.

Place	Temperature in °C	Electric consumption in W
p_1	0	2000
p_2	10	1000
p_3	20	500
p_4	30	1500

Table 2.1: Example of a numerical database

When considering the records p_1, p_2 and p_3, it appears that an increase in temperature is correlated with a decrease in electric consumption. This co-variation of the temperature and the electric consumption can be represented by the *gradual itemset* $(T^{\uparrow}, EC^{\downarrow})$ (where T stands for *Temperature* and EC stands for *Electric Consumption.* This gradual itemset is *respected* by the sequence of records $[p_1, p_2, p_3]$. Note that symmetrically, $(T^{\downarrow}, EC^{\uparrow})$ is respected by $[p_3, p_2, p_1]$.

Let $A = \{a_1, \ldots, a_m\}$ be a set of attributes and $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a set of records where each record p_i with $i \in [1, n]$ stores a numerical value for every attribute in A. The problem of mining closed frequent gradual itemsets can be represented in our framework by considering as ground set the variations of attributes and by encoding transactions and patterns as subsets of attribute variations verifying some constraints. The encoding is the following:

Ground set: E is the set of attributes variations: $E = \{a_1^{\uparrow}, a_1^{\downarrow}, \ldots, a_m^{\uparrow}, a_m^{\downarrow}\}$.

Dataset: In the dataset $\mathcal{D}_E$, there are as many transactions as pairs of records $(p_i, p_j) \in \mathcal{P}$ with $i, j \in [1, n]$ and $i \neq j$. A transaction has as identifier (p_i, p_j) if it contains the variation for every attribute in A between the record p_i and p_j. We will denote the corresponding transaction $t_{(p_i, p_j)}$: for every attribute $a \in A$, $a^{\uparrow} \in t_{(p_i, p_j)} \Leftrightarrow p_i[a] \leq p_j[a]$ ($p[a]$ denoting the value of attribute a for record p), $a^{\downarrow} \in t_{(p_i, p_j)}$ otherwise. The corresponding encoded dataset for the database in Table 2.1 is shown in Table 2.2.

$$\begin{array}{rl}
t_{(p_1,p_2)}: & [\{T^{\uparrow}, EC^{\downarrow}\}, \\
t_{(p_1,p_3)}: & \{T^{\uparrow}, EC^{\downarrow}\}, \\
t_{(p_1,p_4)}: & \{T^{\uparrow}, EC^{\downarrow}\}, \\
t_{(p_2,p_1)}: & \{T^{\downarrow}, EC^{\uparrow}\}, \\
t_{(p_2,p_3)}: & \{T^{\uparrow}, EC^{\downarrow}\}, \\
t_{(p_2,p_4)}: & \{T^{\uparrow}, EC^{\uparrow}\}, \\
t_{(p_3,p_1)}: & \{T^{\downarrow}, EC^{\uparrow}\}, \\
t_{(p_3,p_2)}: & \{T^{\downarrow}, EC^{\uparrow}\}, \\
t_{(p_3,p_4)}: & \{T^{\uparrow}, EC^{\uparrow}\}, \\
t_{(p_4,p_1)}: & \{T^{\downarrow}, EC^{\uparrow}\}, \\
t_{(p_4,p_2)}: & \{T^{\downarrow}, EC^{\downarrow}\}, \\
t_{(p_4,p_3)}: & \{T^{\downarrow}, EC^{\downarrow}\}]
\end{array}$$

Table 2.2: Encoding for the database in Table 2.1

Selection criterion: Given a constant ε, and a candidate pattern $G = \{a_{g_1}^{v_1}, \ldots, a_{g_k}^{v_k}\}$ with $g_1 < \ldots < g_k$, and $v_1, \ldots, v_k$ variations of the form $\uparrow$ or $\downarrow$, G is a pattern if it is contained in at least ε transactions in $\mathcal{D}_E$ whose identifiers form a *path.* A sequence of transactions identifiers $[(p_{i_1}, p_{j_1}), \ldots, (p_{i_n}, p_{j_n})]$ forms a path if $\forall k \in [1, n[$, $p_{j_k} = p_{i_{k+1}}$. When it is clear from context, we say that the transactions form a path when their identifiers forms a path.

In addition, to account for the symmetry of this problem, we discard from the set of patterns, any pattern $G = \{a_{g_1}^{v_1}, \ldots, a_{g_k}^{v_k}\}$ such that v_1 is not $\uparrow$.

Given a pattern $G = \{a_{g_1}^{v_1}, \ldots, a_{g_k}^{v_k}\}$, *Select*$(G, \mathcal{D}_E)$ returns true if and only if:

- G is contained in at least ε transactions whose tid form a path, $(C1)$

- G is empty or, the first variation v_1 of a_{g_1} in G is $\uparrow$. $(C2)$

Theorem 2.4 (Closure uniqueness for the GRI problem)
The Property $(P1)$ is satisfied for the GRI *problem.*

Proof: Let Q and Q' be two patterns in $\mathcal{F}$ such that $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$ and Z be another pattern in $\mathcal{F}$ such that $Z \subseteq Q \cap Q'$ and $Z \neq \emptyset$. Since Q is a pattern, there exists at least ε transactions including Q whose tids form a path. The same transactions also contains Q' because $\mathcal{D}_E[Q] = \mathcal{D}_E[Q']$, thus $Q \cup Q'$ is also included in the same transactions whose tids form a path. Therefore $Q \cup Q'$ is pattern if and only if the first variation v_1 of $a_{g1}^{v_1}$ is $\uparrow$, which is always granted since Q and Q''s first variations are both $\uparrow$. Hence $Q \cup Q' \in \mathcal{F}$. The Property $(P1)$ is satisfied. □

Closure operator: We recall that the attributes $a_1, \ldots, a_m$ are ordered. The closure of a pattern P is the intersection of the transactions in $\mathcal{D}_E[P]$, from which we remove the descending variations of the attributes before the first attribute with an ascending variation.

Algorithm 3 Closure operator for the GRI problem encoding attribute variations

- **Require:** a gradual itemset pattern P and a dataset $\mathcal{D}_E$ encoding attribute variations
- **Ensure:** returns the unique closure Q of P.

```
1: $Q_{max} \leftarrow \cap \mathcal{D}_E[P]$
2: $a_1 \leftarrow$ the first attribute with an ascending variation in $Q_{max}$
3: $Q \leftarrow Q_{max}$
4: while $\exists a^{\downarrow} \in Q$ such that $a$ precedes $a_1$ in the order of the attributes do
5:     $Q \leftarrow Q \setminus \{a^{\downarrow}\}$
6: end while
7: return $Q$
```

Proof: We show that the closure operator defined in Algorithm 3 for the GRI problem is equivalent to the generic closure operator defined in Algorithm 1.

Let a_1 be the first attribute with an ascending variation in $\bigcap \mathcal{D}_E[P]$.

First, let us show that for any $a^{\downarrow} \in \bigcap \mathcal{D}_E[P]$ such that a is before a_1 in the order of the attributes, there does not exist $S \subset \bigcap \mathcal{D}_E[P]$ such that $a^{\downarrow} \in S$ and $P \cup S$ is a pattern.

Suppose that $P \cup S$ is a pattern: its first variation is ascending and therefore there is $b^{\uparrow}$, where $b < a < a_1$. This contradicts the fact that a_1 is the first attribute with an ascending variation in $\bigcap \mathcal{D}_E[P]$. Therefore the closure of P cannot include any element in the set R of elements suppressed in Line 5 of Algorithm 3.

Second, let us show that $S' = \bigcap \mathcal{D}_E[P] \setminus R$ is the closure of P. P being a pattern, it appears in at least ε transactions of $\mathcal{D}_E[P]$ whose tids form a path. By construction S' has the same support set as P and therefore S' also appears in at least ε transactions forming a path.

By removing R from $\bigcap \mathcal{D}_E[P]$ to build S', it is guaranteed that there is no descending variation on attributes before a_1 in S'. Therefore S' is a pattern.

By definition elements that are not in $\bigcap \mathcal{D}_E[P]$, cannot be in the closure of P, hence S' is maximal and thus is the closure of P. □

Note that the closure operator defined in Algorithm 3 is much simpler than the one defined in [DJLT09]. this is due to our set-based encoding of the problem of closed frequent gradual itemset mining.

2.3 Closed pattern enumeration

In most pattern mining algorithms an *enumeration strategy* ensures that every pattern is outputted once and only once. In our framework, we have designed an enumeration strategy for closed patterns by exploiting the structure of the set of patterns defined by the augmentation relation defined as follows.

Definition 2.11 (Pattern augmentation)
A pattern Q is an augmentation *of a pattern P if there exists $e \in Q \setminus P$ such that $Q = P \cup \{e\}$.*

The set of patterns together with the augmentation relation form a strict partial order with $\perp$ as its minimum element, thus having a directed acyclic graph (DAG) structure. Given this DAG, one enumeration strategy is to explore the set of candidate patterns following an *enumeration tree*, spanning the closed patterns (see Figure 2.5, shaded boxes).

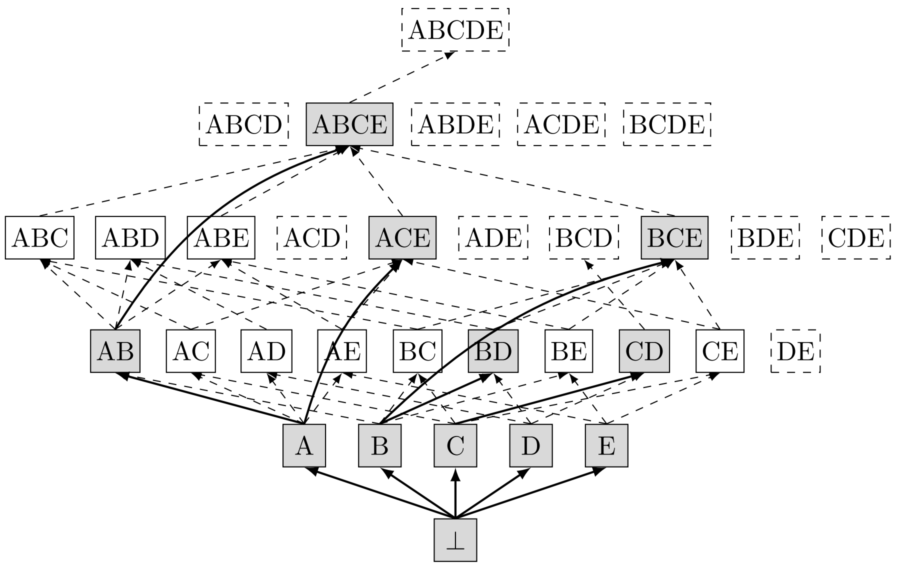


Figure 2.5: A DAG representation of a set $\mathcal{F}$ of patterns defined over a ground set $E = \{\text{A}, \text{B}, \text{C}, \text{D}, \text{E}\}$. In the boxes, the label ACD stands for $\{\text{A}, \text{C}, \text{D}\}$. Dashed boxes are candidate patterns, solid boxes are meaningful patterns and shaded boxes are closed pattern. Each dashed edge connects a pattern and its augmentation. The enumeration tree over the set $\mathcal{C}$ of closed patterns, is presented in solid edges.

This enumeration strategy however, does not ensure space efficiency. Indeed, since the nodes of the enumeration tree are the closed patterns in $\mathcal{C} \subseteq 2^E$, the size of the tree can be exponential with the size of the ground set E. Relying on a memory representation of the tree to ensure the soundness and the completeness of the enumeration strategy is thus inefficient in terms of memory. In addition keeping in memory an up-to-date representation of the enumeration tree, is inadequate to parallel exploration because it requires extensive synchronization and communication which can drastically reduce the concurrency, hence the overall performances of the parallel algorithm.

The problem of designing polynomial space enumeration strategies for which soundness and completeness are not based on a memory representation of the enumeration tree, is a crucial issue for a parallel algorithm. However, this problem was lacking theoretical foundation until recent work of Boley et al. in [BHPW07], then Arimura and Uno in [AU09]. They have investigated the problem of enumerating closed patterns by modeling patterns as sets in a *set system* and they have shown that it is always possible to build a polynomial space enumeration strategy if the set system satisfies some *accessibility properties.*

2.3.1 Patterns as sets in a set system

Definition 2.12 (Set system)
A set system is an ordered pair $(E, \mathcal{F})$ where E is a set of elements and $\mathcal{F} \subseteq 2^E$ is a family of subsets of E.

In the context of pattern mining, E is the ground set, and $\mathcal{F}$ is the set of patterns. We now present three properties of set systems that are observed in most pattern mining problems (discussed in Section 2.3.3). These so called *accessibility properties* are key properties to build time and space efficient enumeration strategies. They are defined here from the least strong to the strongest(as stated in Proposition 2.3).

Definition 2.13 (Accessible set system)
A set system $(E, \mathcal{F})$ is accessible *if for every non-empty $X \in \mathcal{F}$, there exists some $e \in X$ such that $X \setminus \{e\} \in \mathcal{F}$.*

Definition 2.14 (Strongly accessible set system)
A set system $(E, \mathcal{F})$ is strongly accessible *if it is accessible and if for every $X, Y \in \mathcal{F}$ with $X \subset Y$, there exists some $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{F}$.*

Definition 2.15 (Independence set system)
A set system $(E, \mathcal{F})$ is an independence *set system if $Y \in \mathcal{F}$ and $X \subseteq Y$ together imply $X \in \mathcal{F}$.*

The intuition behind the notion of accessibility is that when a set system is accessible, there is a way to reach each pattern by repeatedly augmenting smaller patterns, starting from the empty set (denoted $\perp$). Finding this way is increasingly difficult as we relax the accessibility constraints from independence to accessibility.

The independence property of a set system guarantees that *any* subset of a pattern is also a pattern. For instance, this property is satisfied for the problem of frequent itemset mining (proven in [BHPW07]).

Figure 2.6(a) shows an independence set system. As can be seen, if ABC is a pattern, then any subset of ABC is also a pattern. It is thus possible to reach ABC by repeatedly augmenting A or B or C with any other element in ABC.

The independence property of a set system is not always satisfied. For instance: the encoding proposed in Section 2.2 for the problem of mining closed connected relational graphs. In this problem the subset of a connected graph is not guaranteed to be connected, hence not every subset of a pattern is also a pattern. We prove later in Theorem 2.6 that the set system associated with the CRG problem is *strongly accessible*.

Figure 2.6(b) shows a strongly accessible set system. Contrary to the independence set system, it is impossible to reach ABC by augmenting A with B, because AB is not a pattern.

However, the *strong* accessibility guarantees that for *any* subset of ABC *that is a pattern*, there exist at least one augmentation path toward ABC. Hence we can reach ABC by augmenting A, B, C, AC or BC.

In contrast with both previous examples, the set system in Figure 2.6(c) is accessible, but neither independent nor strongly accessible. In such set systems, the fact that A is a pattern and a subset of ABC does not guarantee that there an augmentation path between A and ABC. However it is guaranteed that there exist at least one such path in the whole set system. We cannot reach ABC by augmenting A, but we can by augmenting B, C or BC.

Proposition 2.3 (Relationship between accessibility properties)
Let $(E, \mathcal{F})$ be a set system:

1. *if it is independent, then it is strongly accessible,*
2. *if it is strongly accessible, then it is accessible.*

Proof:

1. Any independence set system is also strongly accessible: Let $S = (E, \mathcal{F})$ be an independence set system. For every X, Y in $\mathcal{F}$, with $X \subset Y$ let e be any element in $Y \setminus X$. Since S is an independence set system, Y belongs to $\mathcal{F}$ implies that $X \cup \{e\} \subseteq Y$ belongs to $\mathcal{F}$ as well. Therefore for every $X, Y \in \mathcal{F}$ with $X \subset Y$, there exist e such that $X \cup \{e\}$ is in $\mathcal{F}$. S is strongly accessible.
2. By construction, any strongly accessible set system is also accessible.

□

2.3.2 Building the enumeration tree in accessible set system

In order to efficiently build the tree in Figure 2.5, we assume a total order $<_E$, associated with the ground set E. Since any arbitrary is adequate, there is no loss of genericity. The

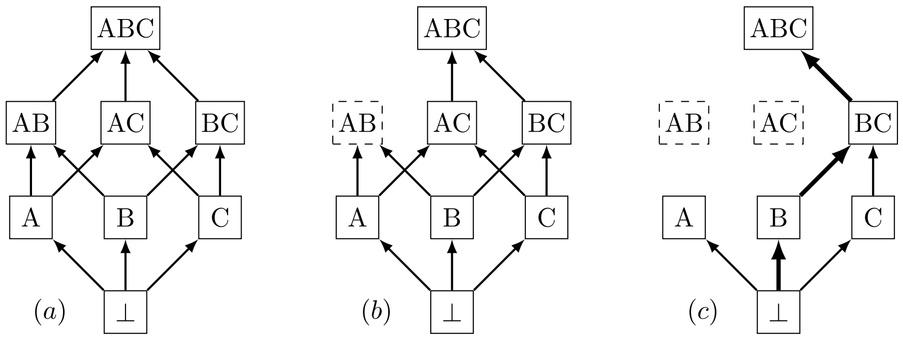


Figure 2.6: (a): an *independence* set system. Each pattern P in $\mathcal{F}$ can be reached by augmenting any subset of P; (b): a *strongly accessible* set system, there is one way to reach any pattern in $\mathcal{F}$ from any of its subset in $\mathcal{F}$; (c): an accessible set system, there is a minimum of one way in the whole DAG. Here, although there is no way to reach ABC from A, there is one way to reach it from B or C.

set of candidate patterns 2^E, can be totally ordered by a lexicographical order built from the order $<_E$. Since $\mathcal{F}$ and $\mathcal{C}$ are both subsets of 2^E they are totally ordered as well. We denote $<_{\mathcal{F}}$ the total order of $\mathcal{F}$ and $\mathcal{C}$, or simply $<$ when it is clear from context.

This order allows to identify for each closed pattern a unique *first parent*, defined as follows:

Definition 2.16 (First parent)
Let P be a closed pattern, and Q the closure of an augmentation $P \cup \{e\}$ of P such that $P < P \cup \{e\}$. P is the first parent *of Q if there does not exist a closed pattern $P' < P$ and an element e' such that $P' < P' \cup \{e'\}$ and Q is the closure of $P' \cup \{e'\}$.*

Given this definition, we can build the enumeration tree and correctly and completely enumerate the set $\mathcal{C}$ by going through it with a recursive algorithm such as the Algorithm 4. The *enum_clo*() procedure in the Algorithm 4 generates the augmentations of a closed pattern P and performs a recursive call if the augmentation has P as first parent, in other words, if there is an arc between P and its augmentation in the enumeration tree.

In Algorithm 4, the first parent test is performed by the $is\_first\_parent()$ boolean function. Given a pattern P and the closure Q of an augmentation of P, $is\_first\_parent(P, Q)$ will return true, if and only if P is the unique first parent of Q.

In any accessible set system, if there exists a polynomial space implementation for $Select$, Clo and is_first_parent, then the algorithm in Algorithm 4 performs in polynomial space([AU09]).

First parent detection

The first parent detection is an increasingly difficult task as we relax the constraints on the accessibility of the set system from independence to accessibility.

When the set system formed by the pattern is *independent*, all the subsets of a (closed) pattern are also patterns, thus checking whether P is the first parent of Q (obtained by closure of an augmentation $P \cup \{e\} > P$) can be reduced to check whether $Q > P$ (see [BHPW07]).

Algorithm 4 Enumerate closed patterns in accessible set systems

- **Require:** An accessible set system $(E, \mathcal{F})$, a closure operator Clo and a dataset $\mathcal{D}_E$.
- **Ensure:** Outputs the set $\mathcal{C}$ of all the closed pattern in $\mathcal{F}$.

```
1: $enum\_clo(Clo(\perp, \mathcal{D}_E), \mathcal{D}_E)$
2: procedure $\textbf{\textit{enum\_clo}}(P, \mathcal{D}_E)$
```

- **Require:** A closed pattern P, the dataset $\mathcal{D}_E$.
- **Ensure:** Outputs the set of all the closed patterns that have P as an ancestor in the enumeration tree.

```
3:     output $P$
4:     //Generate all the augmentations of $P$.
5:     for all $e \in E$ such that $P \cup \{e\} \in \mathcal{F}$ do
6:         if $is\_first\_parent(P, Clo(P \cup \{e\}, \mathcal{D}_E)$ then
7:             //Recursive call if the augmentation's first parent is $P$.
8:             $enum\_clo(Clo(P \cup \{e\}, \mathcal{D}_E), \mathcal{D}_E)$
9:         end if
10:    end for
11: end procedure
```

When the set formed by the patterns is only *accessible*, testing whether a given closed pattern P is the first parent of a pattern Q is a difficult task that may lead to a reverse generation of a whole branch of the enumeration tree to find the actual first parent. Although Arimura and Uno have shown in [AU09] how to design a polynomial space implementation of $is\_first\_parent()$, it has been shown in [BHPW10] that the problem of enumerating closed pattern in accessible set systems is time-intractable in the general case.

When the set system formed by the patterns is *strongly accessible*, this test is complex, but simpler than in accessible systems. It can be done by keeping track of the minimal augmentations (w.r.t. to $<_{\mathcal{F}}$) that were performed on the branch ending to P. If any of these augmentation is included in Q, then it is granted that Q has been expanded from a pattern with a lower value in another branch of the enumeration tree.

This holds because in a strongly accessible set system, any lower subset of Q that is a pattern in $\mathcal{F}$, can be expanded toward Q. In Figure 2.6 (b) if A belongs to $\mathcal{F}$, it is granted that ABC can be reached by repeatedly augmenting A. This does not holds in the accessible set system in Figure 2.6 (c).

Boley et al. in [BHPW07] store the elements used to augment closed patterns that occurred previously in the branch in an *exclusion list* denoted EL. The $is\_first\_parent(P, Q)$ test can be implemented by testing whether $EL \cap Q$ is empty, if it is, P is the first parent of Q.

In the Figure 2.7, we start the exploration with B because A is not a pattern. So far $EL = \emptyset$. The next closed pattern ABC, which is the closure of $\{\text{B}\} \cup \{\text{A}\}$ has B as first parent, and indeed $\text{ABC} \cap EL = \emptyset$ hence it is outputted. Since we augmented B with A we add A to EL, and proceed to augment B towards BC a similar way. At this point A is a valid augmentation for BC, but ABC's first parent is not BC, indeed $EL \cap \text{ABC} = A \neq \emptyset$, hence it not outputted, and we avoided the duplicate generation of ABC.

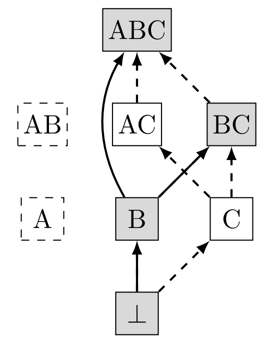


Figure 2.7: A strongly accessible set system

A polynomial space enumeration strategy is presented in Algorithm 5.

Algorithm 5 Enumerate closed patterns in strongly accessible set systems

- **Require:** An accessible set system $(E, \mathcal{F})$, a closure operator Clo.
- **Ensure:** Output the set $\mathcal{C}$ of all the closed pattern in $\mathcal{F}$.

```
1: $enum\_clo(Clo(\perp, \mathcal{D}_E), \mathcal{D}_E, \emptyset)$
2: procedure $\textbf{\textit{enum\_clo}}(P, \mathcal{D}_E, EL)$
```

- **Require:** A closed pattern P, the dataset $\mathcal{D}_E$, and an exclusion list EL.
- **Ensure:** Outputs the set of all the closed patterns that have P as an ancestor in the enumeration tree.

```
3:      output $P$
4:      //Generate all the augmentation of $P$.
5:      for all $e \in E$ such that $P \cup \{e\} \in \mathcal{F}$ do
6:          //detect if $P$ is $P \cup \{e\}$'s first parent.
7:          if $Clo(P \cup \{e\}, \mathcal{D}_E) \cap EL = \emptyset$ then
8:              //Recursive call if $P$ is the first parent.
9:              $enum\_clo(Clo(P \cup \{e\}, \mathcal{D}_E), \mathcal{D}_E, EL)$
10:         end if
11:         $EL := EL \cup \{e\}$
12:     end for
13: end procedure
```

2.3.3 Accessibility in pattern mining problems

We discuss here the accessibility properties of the problems presented in Section2.2.

In order to prove the strong accessibility of several pattern mining problems 2.2, we need the following theorem.

Theorem 2.5 (Set system intersection)
Given two set systems $S_1 = (E, \mathcal{F}_1)$ and $S_2 = (E, \mathcal{F}_2)$ defined over the same ground set E, if S_1 is independent and S_2 is strongly accessible, the set system $S_3 = (E, \mathcal{F}_1 \cap \mathcal{F}_2)$ is

strongly accessible.

Proof: First, let Y be a subset of E in $\mathcal{F}_3$, and let X be any subset of Y. Since $\mathcal{F}_3 = \mathcal{F}_1 \cap \mathcal{F}_2$, Y is also in $\mathcal{F}_1$ and $\mathcal{F}_2$. As a subset of Y, X is also in $\mathcal{F}_1$, therefore any subset X of Y belongs to $\mathcal{F}_3$ if and only if it belongs to $\mathcal{F}_2$.

We now show that S_3 is accessible. $\mathcal{F}_2$ is strongly accessible thus accessible, therefore there exists $e \in Y$ such that $Y \setminus \{e\} \in \mathcal{F}$. However $Y \setminus \{e\}$ is a subset of Y and belongs to $\mathcal{F}_2$, therefore it also belongs to $\mathcal{F}_3$. S_3 is accessible.

In a similar way, we show that S_3 is also strongly accessible. Since $\mathcal{F}_2$ is strongly accessible there exists some $e \in X \setminus Y$ such that $X \cup \{e\} \in \mathcal{F}_2$. However, $X \cup \{e\}$ is a subset of Y and therefore also belongs to $\mathcal{F}_3$. S_3 is accessible and for any $Y, X \in \mathcal{F}_3$, there exists $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{F}_3$ therefore S_3 is strongly accessible. $\square$

Frequent itemsets (FIM)

The set system $(E, \mathcal{F})$ formed by the frequent itemsets have been proven to be an independence set system in [BHPW10].

Frequent Connected Relational Graphs (CRG)

Theorem 2.6 (Accessibility in CRG)
The set system associated with the CRG *problem is strongly accessible for every dataset* $\mathcal{D}_E$*.*

We denote $(E, \mathcal{F}_0)$, with $\mathcal{F}_0 = \mathit{Select}(2^E, \mathcal{D}_E)$ the set system formed by all the sets $X \subseteq E$ satisfying the selection criterion. That is, any X such that:

- X is a frequent set of edges in $\mathcal{D}_E$,
- X is a connected set of edges.

Let $\mathcal{F}_1$ be the set of sets $X \subseteq E$ such that X is a frequent set of edges and let $\mathcal{F}_2$ the set of sets $Y \subseteq E$ such that Y is a connected set of edges. Hence, $\mathcal{F} = \mathcal{F}_1 \cap \mathcal{F}_2$ is set of all the frequent and connected sets of edges $\mathcal{F}_0 = \mathcal{F}_1 \cap \mathcal{F}_2$.

We observe that mining frequent sets of edges is equivalent to mine frequent set of items, hence the set system $(E, \mathcal{F}_1)$ formed by all the frequent set of edges is an independence set system.

The set system $(E, \mathcal{F}_2)$ formed by all the connected sets of edges have been proven to be strongly accessible in [BHPW10].

According Theorem 2.5, the set system formed by all the frequent and connected graphs $(E, \mathcal{F}_0 = \mathcal{F}_1 \cap \mathcal{F}_2)$ is strongly accessible.

Gradual itemsets (GRI)

Theorem 2.7 (Accessibility in GRI)
The set system associated with the GRI *problem is strongly accessible for every dataset* $\mathcal{D}_E$*.*

Proof: We denote $(E, \mathcal{F}_0)$, with $\mathcal{F}_0 = \mathit{Select}(2^E, \mathcal{D}_E)$ the set system formed by all the set $X \subseteq E$ satisfying the selection criterion. That is, any $X = \{a_{x_1}^{v_1}, \dots, a_{x_k}^{v_k}\}$ such that:

- X is contained in at least ε transactions whose tid form a path. $(P1)$
- X is empty or, the first variation v_1 of a_{x_1} in X is $\uparrow$. $(P2)$

Let $\mathcal{F}_1$ be the set of sets $X \subseteq E$ satisfying $(P1)$ and, $\mathcal{F}_2$ the set of sets $Y \subseteq E$ satisfying $(P2)$. The set of sets satisfying both conditions is $\mathcal{F}_0 = \mathcal{F}_1 \cap \mathcal{F}_2$.

We show that $(E, \mathcal{F}_1)$ is an independence set system and that $(E, \mathcal{F}_2)$ is a strongly accessible set system. Hence we can apply Theorem 2.5.

$(E, \mathcal{F}_1)$ **is an independence set system:** $\forall Y \in \mathcal{F}_1$ with $X \subseteq Y$, $Y \in \mathcal{F}_1$ if and only if there exists at least ε transaction in $\mathcal{D}_E[Y]$ whose tid form a path. Since $X \subseteq Y$, $\mathcal{D}_E[Y] \subseteq \mathcal{D}_E[X]$, the same path also exists in $\mathcal{D}_E[X]$. $X \in \mathcal{F}_1$, and thus the set system F_1 an independence set system.

$(E, \mathcal{F}_2)$ **is strongly accessible:** We recall that $(E, \mathcal{F}_2)$ is strongly accessible if and only if:

1. it is accessible
2. for every $X, Y \in \mathcal{F}$ with $X \subset Y$, there exists some $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{F}$.

We first prove that the set system $(E, \mathcal{F}_2)$ is accessible. From $(P2)$, $Y = \{a_{y_1}^{v_1}, \dots, a_{y_k}^{v_k}\} \in \mathcal{F}_2$ implies that the variation v_1 of the first attribute a_{y_1} in Y, is $\uparrow$. We show that there always exists $a_{y_i}^{v_i} \in Y$, with $i \in [1, k]$, such that $Y \setminus \{a_{y_i}^{v_i}\}$ still satisfies $(P2)$. For every $Y \in \mathcal{F}_2$, if $|Y| \geq 2$, there exists at least one $a_{y_i}^{v_i} \in Y$ with $i \neq 1$. Hence there exists $a_{y_i}^{v_i}$, such that $X = Y \setminus \{a_{y_i}^{v_i}\}$ contains $a_{y_1}^{v_1}$. Since $a_{y_1}^{v_1}$, is the variation for the first attribute in X, and $v_1 = \uparrow$, $X = Y \setminus \{a_{y_i}^{v_i}\}$ satisfies $(P2)$. In addition, if $|Y| = 1$, $X = Y \setminus \{a_{y_1}^{v_1}\} = \emptyset$ for every $e \in Y$, which is granted to be in $\mathcal{F}_2$ since the restriction $(P2)$ does not applies to $\emptyset$. Therefore for every $Y \in \mathcal{F}_2$ there exists at least one $a_{y_i}^{v_i}$ such that $Y \setminus \{a_{y_i}^{v_i}\} \in \mathcal{F}_2$. (E, F_2) is accessible. The set system $(E, \mathcal{F}_2)$ is accessible.

We show that for every $X, Y \in \mathcal{F}$ with $X \subset Y$, there exists some $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{F}$. Let $X = \{a_{x_1}^{v_1}, \dots, a_{x_k}^{v_k}\}$ and $Y = \{a_{y_1}^{v_1}, \dots, a_{y_p}^{v_p}\}$ be two patterns in $\mathcal{F}_2$ with $X \subset Y$, since (E, F_2) is accessible, it also is strongly accessible if and only if $\forall X, Y \in \mathcal{F}_2$ with $X \subset Y$, there exists $a_{y_i}^{v_i} \in Y \setminus X$, such that $X \cup \{a_{y_i}^{v_i}\} \in \mathcal{F}_2$.

- If $|Y| - |X| = 1$ then there exists only one $a_{x_i}^{v_i} \in Y \setminus X$ and $Y \setminus \{a_{x_i}^{v_i}\} = X$. Since $X \in \mathcal{F}_2$, $Y \setminus \{a_{x_i}^{v_i}\} = X$ is in $\mathcal{F}_2$ as well.
- If $|Y| - |X| \geq 2$, let $a_{y_i}^{v_i}$ be any attribute variation in $Y \setminus X$ with $a_{y_i}^{v_i} \neq a_{y_1}^{v_1}$. Then $Y \setminus \{a_{y_i}^{v_i}\}$ admits $a_{y_1}^{v_1}$ as a first variation with $v_1 = \uparrow$. $Y \setminus \{a_{y_i}^{v_i}\} \in \mathcal{F}_2$.

$(E, \mathcal{F}_2)$ is a strongly accessible.

From Theorem 2.5, if $(E, \mathcal{F}_1)$ is an independence set system, and $(E, \mathcal{F}_2)$ is a strongly accessible set system, the set system $(E, \mathcal{F}_0 = \mathcal{F}_1 \cap \mathcal{F}_2)$ is strongly accessible.

2.4 Discussion

The framework presented in this chapter offers an interesting base for a generic and parallel pattern mining algorithm. Several pattern mining problems such as frequent itemset mining, closed connected graph mining and gradual pattern mining can be successfully captured with a reasonably sized ground set. In addition, we have shown that a *generic* closure operator can be defined when the union of two patterns having a non-empty pattern in their intersection and *having the same support set* is a pattern. This property (property $(P1)$ of our Theorem 2.1) is more appropriate than the confluence property in [BHPW10] for characterising the pattern mining problems in which the closure operator is well defined. This is due to the fact that our definition of pattern, in contrast with [BHPW10] requires patterns to *occur* in the dataset which is an obvious requirement when data is involved.

Any pattern mining problem formulated into our framework comes with an underlying set system. The structural properties of the underlying set system is a sufficient information to design an efficient enumeration strategy that does not rely on the problem definition. We showed in this chapter that the independence property of a set system is too strict to capture most pattern mining problems, and the accessibility property is too loose to build efficient pattern enumeration strategies. The strong accessibility is a fair compromise.

Given that the set system is strongly accessible, the set theory can be used to design efficient polynomial space enumeration strategies. In addition to the space complexity guaranteed it offers several benefits that are major issues to a parallel pattern mining algorithm, the main one being a cutting plane to distribute the work among several processing units without synchronization.

Although these are important improvements they are far from enough to obtain a generic and practically efficient algorithm for pattern mining. Indeed, dealing with large datasets is another main issue of pattern mining that have not been addressed yet.

Chapter 3

The ParaMiner algorithm

Contents

In this section, we present ParaMiner, a parallel algorithm able to solve every pattern mining problem formalized in the framework described in the previous chapter.

The enumeration strategy implemented in ParaMiner is built on the principles of pattern enumeration in strongly accessible set systems recalled in the previous chapter. This enumeration strategy consists in exploring a tree covering the closed patterns. The strong accessibility of the set system formed by the set of patterns guarantees that it is possible to build every branch of the tree without sharing data with other branches. This is a major benefit for a parallel algorithm in which communication between the different processing units can reduce the concurrency and the performance of the algorithm.

Although this enumeration strategy is a good base for building a parallel pattern mining algorithm, it is not sufficient to achieve practical efficiency. The selection criterion and the closure operator are performed millions of times along the execution of the algorithm, hence their efficiency is also a critical issue. Since these two operations typically require to access the dataset, providing efficient access to the relevant information is mandatory to be able to handle medium or large datasets.

In order to propose a solution to this problem, pattern mining researchers have made an important observation: most patterns occur in a small part of the dataset only, hence the whole dataset is not required to compute the selection criterion for a given set of candidate patterns. In [HPY00] Han et al. have built FP-Growth, a recursive frequent itemset mining algorithm. In FP-Growth, each recursive call inputs a frequent itemset and outputs a set of larger frequent itemsets. In order to count itemset's frequency efficiently, each recursive call holds a tree representation of the dataset called a *conditional pattern tree* containing only the support set of the input pattern. Later in [UKA04], Uno et al. have adapted this technique to closed itemset mining by proposing *database reduction*. In their LCM algorithm all the computations required to augment a closed pattern are performed in a sub-dataset instead of in the full dataset. A sub-dataset is built for each closed pattern. Each sub-dataset is a copy of the initial dataset where the transactions and elements not required to compute the augmentations of the closed pattern are removed. In this chapter we show how to generalize database reduction to our generic framework and how it can be used to speed up the computation of *Clo* and *Select*. In order to be be consistent with our framework we call this technique data*set* reduction.

Distributing the computations among the available processing units is a critical issue in any parallel algorithm. Indeed, different distributions strategies may significantly impact the parallel efficiency of the algorithm. However ParaMiner as well as many other data mining algorithms is *data-driven*. It means that the execution of the algorithm is driven by the input data and the intermediary results. Hence depending on the problem and the data at hand, the optimal strategy for distributing the computations and the data may differ. To address this problem we have designed Melinda which is a parallelism engine adapted to data-driven algorithms. Melinda assumes that tasks are produced all along the execution of the algorithm and can be consumed in any order. With Melinda, anyone with basic programming skills can build and evaluate new distribution strategies. We show that Melinda can be used to design distribution strategies that fit the memory sub-system of the execution platform and improve the run time or the memory consumption.

In this chapter we first present the ParaMiner algorithm in Section 3.1 and show how to combine the enumeration strategy presented in the previous chapter with a dataset reduction. The dataset reduction technique and other optimizations is discussed in the following Section 3.2. In Section 3.3, we show how to break up the exploration of the enumeration tree into tasks, we then present Melinda and show how it can be used to efficiently run ParaMiner on multi-core platforms.

ParaMiner is a generalization of the PLCM algorithm that we have previously designed and implemented on top of Melinda. PLCM is a parallel algorithm designed for making LCM parallel. LCM [UKA04] is the state of the art fastest sequential algorithm to mine closed frequent itemsets. To avoid redundancy in this thesis report, we have chosen to focus on describing ParaMiner. For details on PLCM, we refer to [NTMU10].

3.1 ParaMiner: main algorithm

In ParaMiner, the enumeration tree is explored in a bottom-up fashion starting from $\perp$ as in the enumeration strategy described in the previous chapter. Given a closed pattern P, the *expand*() procedure (Algorithm 7) is in charge of outputting every closed patterns that is a descendent of P in the enumeration tree.

The input parameters of the *expand*() procedure are:

- a closed pattern P,
- the *reduced dataset* $\mathcal{D}_P^{reduced}$ of P (see Section 3.2),
- a copy of the exclusion list EL that is required to perform the first parent test as described in Section 2.3.2.

Each individual call to $expand(P, \mathcal{D}_P^{reduced}, EL)$ outputs the set of closed patterns that have P as first parent.

ParaMiner starts the exploration of the enumeration tree by calling the *expand*() procedure on $\perp$. When *expand*() finds a closed pattern Q, it builds the reduced dataset $\mathcal{D}_Q^{reduced}$ of Q by calling the *reduce*() function with the parent dataset $\mathcal{D}_P^{reduced}$ as a first argument. The *expand*() procedure is then recursively called to build the augmentations of Q (as illustrated in Figure 3.1).

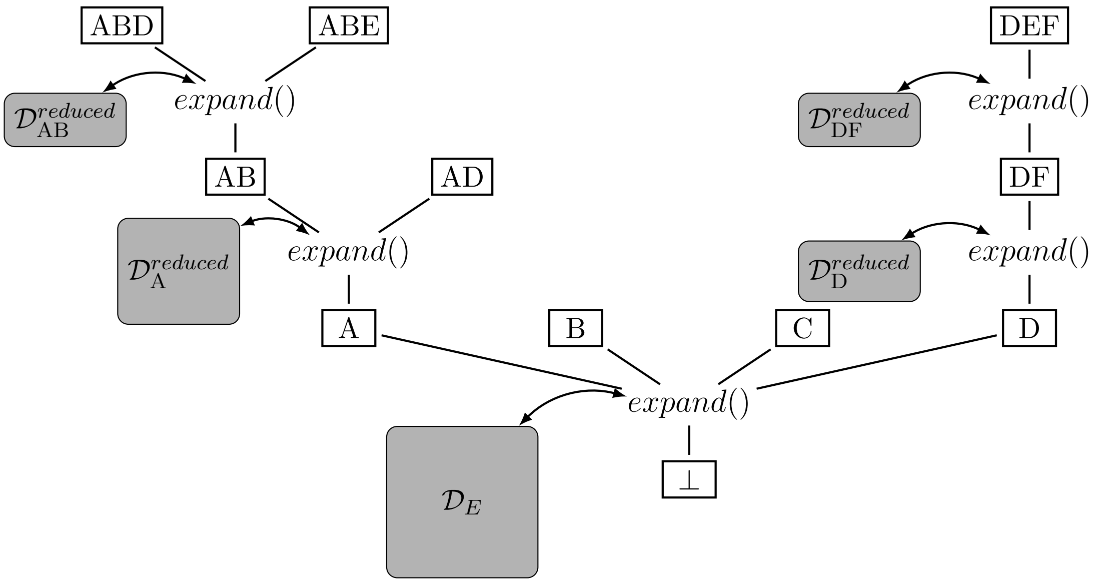

ABD
ABE
DEF
$\mathcal{D}_{AB}^{reduced}$
$expand()$
$\mathcal{D}_{DF}^{reduced}$
$expand()$
AB
AD
DF
$\mathcal{D}_{A}^{reduced}$
$expand()$
$\mathcal{D}_{D}^{reduced}$
$expand()$
A
B
C
D
$expand()$
$\mathcal{D}_E$
$\perp$

Figure 3.1: The *expand*() procedure explores the search space following the enumeration tree (white boxes), and builds a reduced dataset for each new node of the enumeration tree (shaded boxes).

expand() builds the augmentations of a pattern P by testing (Line 3) which candidate pattern $P \cup \{e\}$ satisfies the selection criterion (given as input). However we only perform this test for the elements e that occurs in the reduced dataset of P. This is not a limitation compared to Algorithm 5 because we build the reduced dataset $\mathcal{D}_P^{reduced}$ of P such that it is guaranteed that if e does not occurs in $\mathcal{D}_P^{reduced}$, then the candidate pattern $P \cup \{e\}$ would either not occur in the initial dataset – hence fail the pattern test (Line 5 in Algorithm 5) – or fail the first parent test (Line 7 in Algorithm 5).

If $P \cup \{e\}$ satisfy the selection criterion *Select*, ParaMiner applies the closure operator, *Clo* (given as input) to get the closure Q of $P \cup \{e\}$. For each one of them, Line 5 checks whether P is their *first parent* by a simple intersection test with the exclusion list EL (as in Algorithm 5).

The closed patterns are outputted as soon as they are produced in Line 7. The *expand*() procedure is then called Line 9, on each Q and together with its corresponding reduced dataset $\mathcal{D}_Q^{reduced}$. The reduced dataset is build Line 8 by the *reduce*() function described in Algorithm 8, Section 3.2), and the same exclusion list EL.

In Line 9 instead of performing the recursive call the *expand*() procedure, we *spawn* a new task that can be performed on a different core. We discuss this parallelization strategy in Section 3.3.

Algorithm 6 ParaMiner

- **Require:** ground set E, selection criterion $Select$, closure operator Clo, dataset $\mathcal{D}_E$
- **Ensure:** Outputs all closed patterns occurring in $\mathcal{D}_E$.

```
1: output $Clo(\perp, \mathcal{D}_E)$
2: $expand(Clo(\perp, \mathcal{D}_E), \mathcal{D}_E, \emptyset)$
```

Algorithm 7 Expanding a closed pattern P

```
1: procedure expand($P$, $\mathcal{D}_P^{reduced}$, $EL$)
```

- **Require:** A closed pattern P, a reduced dataset $\mathcal{D}_P^{reduced}$, an exclusion list EL.
- **Ensure:** Output all closed patterns that are descendent of P in the enumeration tree.

```
2:     for all $e$ such that $e$ occurs in $\mathcal{D}_P^{reduced}$ do
3:         if $Select(P \cup \{e\}, \mathcal{D}_P^{reduced})$ then
4:             $Q \leftarrow Clo(P \cup \{e\}, \mathcal{D}_P^{reduced})$
5:             if $EL \cap Q = \emptyset$ then
6:                 //P is Q's the first parent
7:                 output $Q$
8:                 $\mathcal{D}_Q^{reduced} \leftarrow reduce(\mathcal{D}_P^{reduced}, e, EL)$
9:                 spawn $expand(Q, \mathcal{D}_Q^{reduced}, EL)$
10:                $EL \leftarrow EL \cup \{e\}$
11:            end if
12:        end if
13:    end for
14: end procedure
```

ParaMiner's efficiency depends on the *reduce*() function, we explain in details the principles of the dataset reduction and provide the complete algorithm in the next section. We then prove ParaMiner's soundness.

3.2 Optimizations based on dataset reduction

ParaMiner is based on the same principles of Boley et al.'s enumeration strategy of closed patterns. In fact, if in Algorithm 7, $\mathcal{D}_P^{reduced}$ and $\mathcal{D}_Q^{reduced}$ (respectively appearing in Line 1, 2, 3, 4, 8 and 9) were replaced by $\mathcal{D}_E$ (the full dataset), we would get exactly the *enum_clo* algorithm in Algorithm 5 proven to be sound and complete by Boley et al in [BHPW10]. However, the practical efficiency of ParaMiner (that will be demonstrated in Chapter 4) relies on the fact that the full dataset is replaced by appropriate reduced datasets in the computation of *Clo* (Line 4 of Algorithm 7), *Select* (Line 3 of Algorithm 7) and in the elements chosen to expand closed pattern (Line 2 of Algorithm 7). In Section 3.2.1, we provide the algorithm computing the *reduce*() function at the core of ParaMiner and we show why the resulting reduced datasets are sufficient for the computation of the closure operator (under some constraints on *Select*). In Section 3.2.2, we explain how to build appropriate indexes for the reduce datasets that are the basis for testing and counting the occurrences of the patterns in the (reduced) dataset, we then show how indexes can be used to efficiently compute support sets and pattern's closures.

3.2.1 Dataset reduction: principles and properties

In the enumeration strategy implemented by *expand*() and thus in ParaMiner, a closed pattern Q is obtained by an appropriate augmentation of a closed pattern P (called its first parent).

Dataset reduction is a recursive process: we explain in the next two paragraphs how the reduced dataset of a closed pattern Q is obtained by by removing transactions and elements from the reduced dataset of its first parent P.

The algorithm implementing the corresponding *reduce*() function is shown in Algorithm 8.

In the following, we denote e the element used to obtain Q by augmenting P: $Q = Clo(P \cup \{e\})$.

Algorithm 8 The dataset reduction algorithm

```
1: function reduce($\mathcal{D}_P^{reduced}$, $e$, $EL$)
• Require: The reduced dataset $\mathcal{D}_P^{reduced}$ of the parent $P$ of $Q$, the augmenting element
    $e$ such that $Q = Clo(P \cup \{e\}, \mathcal{D}_E)$, the exclusion list $EL$.
• Ensure: Returns the reduced dataset of $Q$: $\mathcal{D}_Q^{reduced}$
2:     $\mathcal{D}_Q^{reduced} \leftarrow \mathcal{D}_P^{reduced}[\{e\}]$
3:     // Suppress elements from transactions.
4:     for all $G \in partition(\mathcal{D}_Q^{reduced}, EL)$ do
5:         for all $e \in EL$ do
6:             if there exists $t' \in G$ such that $e \notin t'$ then
7:                 Suppress $e$ from all the transactions in $G$
8:             end if
9:         end for
10:    end for
11:    return $\mathcal{D}_Q^{reduced}$
12: end function
```

Removing transactions: It is done in Line 2 of Algorithm 8, and consists in initializing $\mathcal{D}_Q^{reduced}$ with the support set of $\{e\}$ in $\mathcal{D}_P^{reduced}$. This first reduction step was initially introduced by FP-Growth in [HPY00], it consists in removing from the dataset any transaction not including Q.

Suppressing elements from transactions: By construction, elements in the exclusion list EL cannot appear in patterns enumerated from Q. These elements however cannot be directly suppressed from $\mathcal{D}_Q^{reduced}$, because some of them are needed to compute the closure of patterns enumerated from Q (Line 4 of Algorithm 7) and check that these patterns have an empty intersection with EL (Line 5). The elements of the exclusion list EL that can be safely removed from transactions in $\mathcal{D}_Q^{reduced}$ are the elements that are guaranteed not to appear in any closed pattern that is a descendent of Q in the enumeration tree.

For example, given the ground set $E = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$, let $\{\mathrm{C}\}$ be a closed pattern, and $EL = \{\mathrm{A}, \mathrm{B}\}$ an exclusion list. Considering the following dataset (each line is a transcription):

$$\mathcal{D}_{\{\mathrm{C}\}}^{reduced} = \boxed{\begin{array}{cccc} & & \downarrow & \\ \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \mathrm{A} & & \mathrm{C} & \mathrm{D} \\ & & \mathrm{C} & \end{array}}$$

In this dataset, the only superset of $\{\mathrm{C}\}$ that can be a closed pattern and a descendent of $\{\mathrm{C}\}$ in the enumeration tree is the candidate pattern $\{\mathrm{C}, \mathrm{D}\}$. Any other superset of Q would necessarily include A or B which belong to the exclusion list, hence it would fail the first parent test (Line 5 in Algorithm 7).

However A must be kept in $\mathcal{D}_Q^{reduced}$ because it is a possible element of the closure of $\{\mathrm{C}, \mathrm{D}\}$. Indeed $\{\mathrm{C}, \mathrm{D}\}$ and $\{\mathrm{A}\}$ have the same support set, hence if $Select(\{\mathrm{A}, \mathrm{C}, \mathrm{D}\}) = true$, then the closure of $\{\mathrm{C}, \mathrm{D}\}$ is $\{\mathrm{A}, \mathrm{C}, \mathrm{D}\}$. Without A in the dataset the closure of $\{\mathrm{C}, \mathrm{D}\}$ cannot be computed correctly and the first parent test is not guaranteed to fail as it should be. This scenario could lead to the generation of $\{\mathrm{C}, \mathrm{D}\}$ as a closed pattern even if it is not one.

B does not have the same support set as any superset of $\{\mathrm{C}\}$ that is a descendent of $\{\mathrm{C}\}$ in the enumeration tree, therefore it can be removed from $\mathcal{D}_Q^{reduced}$ without altering the soundness of PARAMINER.

For determining easily the elements that cannot belong to any closure of a descendent of Q, $\mathcal{D}_Q^{reduced}$ is partitioned (by the function $partition(\mathcal{D}_Q^{reduced}, EL)$ called Line 4 in Algorithm 8) in groups of sets of transactions that have the same elements except elements of EL (the *partition*() function is detailed in Algorithm 9). For each group G of the partition, we suppress from each of its transactions the elements of EL that do not appear in all the transactions of the group. Such elements will not belong to the closure of any pattern Q' further produced from augmentations of Q and supported by the transactions in G. This is done in Lines 5–9 of Algorithm 8. Note that this is a generalization of the so-called *prefix intersection* optimization at the core of LCM [UAUA04].

The following theorem characterizes the dataset reduction. It is the key to guarantee PARAMINER's soundness. It states that the transaction identifiers (*tid support sets*, see Definition 2.4, Page 11) are preserved by dataset reduction, and the elements that belong to *all* the transactions in the support set of a pattern augmentation are also preserved.

Theorem 3.1
Let $Q = Clo(P \cup \{e\}, \mathcal{D}_P^{reduced})$ *and* $\mathcal{D}_Q^{reduced}$ *be the result of* $reduce(\mathcal{D}_P^{reduced}, e, EL)$. *If* e' *occurs in* $\mathcal{D}_Q^{reduced}$ *and* $(Q \cup \{e'\}) \cap EL = \emptyset$ *then:*

- $\mathcal{D}_E[\![Q \cup \{e'\}]\!] = \mathcal{D}_Q^{reduced}[\![\{e'\}]\!]$.

- $\bigcap \mathcal{D}_E[Q \cup \{e'\}] = \bigcap \mathcal{D}_Q^{reduced}[\{e'\}]$.

Algorithm 9 The partition function used in *reduced*()

```
1: function partition($\mathcal{D}_Q^{reduced}$, $EL$)
• Require: A dataset $\mathcal{D}_Q^{reduced}$, an exclusion list $EL$
• Ensure: Returns sets of transactions that are equal when considering only the elements
    that are not in $EL$.
2:     $\mathcal{T} \leftarrow \mathcal{D}_Q^{reduced}$
3:     for all $t \in \mathcal{T}$ do
4:         $G \leftarrow \{t\}$
5:         $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
6:         for all $t' \in \mathcal{T}$ do
7:             if $t \setminus EL = t' \setminus EL$ then
8:                 //$t$ and $t'$ have the same set of non $EL$-elements.
9:                 //They belong to the same group of transactions.
10:                $G \leftarrow G \cup \{t'\}$
11:                $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t'\}$
12:            end if
13:        end for
14:        $\mathcal{G} \leftarrow \mathcal{G} \cup \{G\}$
15:    end for
16:    return $\mathcal{G}$
17: end function
```

Proof: Let us prove the first item. Let i be a tid in $\mathcal{D}_E[\![Q \cup \{e'\}]\!]$: it is the identifier of a transaction t in $\mathcal{D}_E$ including $Q \cup \{e'\}$. t cannot have been removed by Line 2 of *reduce*() because the support set of $\mathcal{D}_E[Q \cup \{e'\}]$ is included in $\mathcal{D}_E[Q]$. In addition the elements in $Q \cup \{e'\}$ cannot have been removed from t by Line 7 because $(Q \cup \{e'\}) \cap EL = \emptyset$. Therefore the resulting transaction t' obtained by removing elements from t necessarily is in $\mathcal{D}_Q^{reduced}[e']$. Therefore i is also in $\mathcal{D}_Q^{reduced}[\![e']\!]$.

Conversely, since $e' \notin EL$, e' occurs in $\mathcal{D}_Q^{reduced}$ and thus there exists a tid i in $\mathcal{D}_Q^{reduced}[\![e']\!]$. Let t' the transaction identified by i in $\mathcal{D}_Q^{reduced}[\![e']\!]$ and let t the transaction identified by i in $\mathcal{D}_E$. By construction: $Q \cup \{e'\} \subseteq t'$ and $t' \subseteq t$, and thus $Q \cup \{e'\} \subseteq t$. Therefore i is in $\mathcal{D}_E[\![Q \cup \{e'\}]\!]$.

Let us now prove the second item. Let a in $\bigcap \mathcal{D}_E[Q \cup \{e'\}]$. By construction each group of transactions in $\mathcal{D}_E$ including $Q \cup \{e'\}$ also includes a. Hence the condition in Line 6 in Algorithm 8 is not satisfied for a. Therefore a is not suppressed by *reduce*() from any transaction in $\mathcal{D}_Q^{reduced}[\{e'\}]$ and thus it belongs to $\bigcap \mathcal{D}_Q^{reduced}[\{e'\}]$.

Since $e' \notin EL$, e' occurs in $\mathcal{D}_Q^{reduced}$ and thus there exists a in $\bigcap \mathcal{D}_Q^{reduced}[\{e'\}]$. Suppose that a is not in $\bigcap \mathcal{D}_E[Q \cup \{e'\}]$: there exists a transaction t in $\mathcal{D}_E[Q \cup \{e'\}]$ including $Q \cup \{e'\}$ that does not include a. This transaction cannot have been removed by Line 2 of *reduce*() because the support set of $\mathcal{D}_E[Q \cup \{e'\}]$ is included in $\mathcal{D}_E[Q]$. In addition the elements in $Q \cup \{e'\}$ cannot have been removed from t by Line 7 because $(Q \cup \{e'\}) \nsubseteq EL$. Therefore the resulting transaction t' obtained by removing elements from t necessarily is in $\mathcal{D}_Q^{reduced}[\{e'\}]$. Since a is not in t it cannot be in t' either. This contradicts the fact a is in $\bigcap \mathcal{D}_Q^{reduced}[\{e'\}]$. □

We focus now on the soundness of ParaMiner. As we will show further, the soundness of ParaMiner is guaranteed if the *Select* predicate is *decomposable* (Definition 3.1). This definition states that $\mathit{Select}(P, \mathcal{D}_E)$ can be decomposed in two constraints, one on the pattern P and the other on the *tid support set* of P in $\mathcal{D}_E$ (see Definition 2.4).

Definition 3.1 (Decomposability of *Select*)
Given a ground set E, the Select predicate is decomposable if and only if there exist a constraint $C1$ and a constraint $C2$ such that for every dataset $\mathcal{D}_E$ and every pattern P, $\mathit{Select}(P, \mathcal{D}_E) \equiv C1(P) \wedge C2(\mathcal{D}_E[\![P]\!])$, where $\mathcal{D}_E[\![P]\!]$ is the tid support set of P in $\mathcal{D}_E$.

To prove the soundness of ParaMiner, we need to prove the following lemmas.

Lemma 3.1 states that the closure Q of a pattern $P \cup \{e\}$ computed Line 4 in Algorithm 7 is the same in $\mathcal{D}_E$ and in $\mathcal{D}_P^{reduced}$ ($\mathit{Clo}(P \cup \{e\}, \mathcal{D}_E) = \mathit{Clo}(P \cup \{e\}, \mathcal{D}_P^{reduced})$).

Lemma 3.2 states that the *expand*() function of Algorithm 7 returns the same output as the *enum_clo*() function (Algorithm 5, Page 26).

Those two lemmas (and thus the Theorem 3.2) are true under the following condition ($P2$) on the predicate *Select*, that defines what the patterns are and thus varies depending on the pattern mining problem to solve. ($P2$) holds if the *Select* predicate can be checked on reduced datasets (instead of the full dataset). Based on Theorem 3.1 (first item), ($P2$) is true if the *Select* predicate is decomposable.

Condition (P2): For every call to $\mathit{expand}(P, \mathcal{D}_P^{reduced}, EL)$, for every element e not in EL and occurring in $\mathcal{D}_P^{reduced}$, and for every $S \subset \bigcap \mathcal{D}_E[P \cup \{e\}]$, $\mathit{Select}(P \cup \{e\} \cup S, \mathcal{D}_E) \equiv \mathit{Select}(P \cup \{e\} \cup S, \mathcal{D}_P^{reduced})$.

Lemma 3.1
Let P be a closed pattern and $\mathcal{D}_P^{reduced}$ its reduced dataset. If e is an element such that $P \cup \{e\}$ is a pattern in $\mathcal{D}_P^{reduced}$ (e occurs in $\mathcal{D}_P^{reduced}$ and $\mathit{Select}(P \cup \{e\}, \mathcal{D}_P^{reduced})$) then, if the Select predicate is decomposable, the following property holds: $\mathit{Clo}(P \cup \{e\}, \mathcal{D}_P^{reduced}) = \mathit{Clo}(P \cup \{e\}, \mathcal{D}_E)$.

Proof: We show that the generic closure operator provided in Algorithm 1, Page 16 applied with $\mathcal{D}_P^{reduced}$ as the input dataset returns the same result than the one provided with $\mathcal{D}_E$.

We denote $Q(\mathcal{D}_E)$ the result of Algorithm 1 applied to $(P \cup \{e\}, \mathcal{D}_E)$ and $Q(\mathcal{D}_P^{reduced})$ the result of Algorithm 1 applied to $(P \cup \{e\}, \mathcal{D}_P^{reduced})$.

Let e' an element of $Q(\mathcal{D}_E)$, we show that e' is also in $Q(\mathcal{D}_P^{reduced})$.

e' is in $Q(\mathcal{D}_E)$ implies the following:

- $e' \in \bigcap \mathcal{D}_E[P \cup \{e\}]$
- let S be the set of elements added before e (i.e. in former iterations of the while loop in Line 4 in Algorithm 1. It is granted that $\mathit{Select}(P \cup \{e\} \cup S \cup \{e'\}, \mathcal{D}_E)$ is true.

Therefore if $e' \notin Q(\mathcal{D}_P^{reduced})$ either:

- $e' \notin \bigcap \mathcal{D}_P^{reduced}[P \cup \{e\}]$: this cannot be the case because e' occurs in $\mathcal{D}_P^{reduced}$ and $P \cup \{e\} \nsubseteq EL$ hence Theorem 3.1 applies with $Q = P$.
- there exists S' such that $Select(P \cup \{e\} \cup S' \cup \{e'\}, \mathcal{D}_P^{reduced})$ is false whereas $Select(P \cup \{e\} \cup S' \cup \{e'\}, \mathcal{D}_E)$ is true. This is impossible from condition $(P2)$ and $S = S' \cup \{e'\}$.

Conversely we show that if e' is an element of $Q(\mathcal{D}_P^{reduced})$, e' is also in $Q(\mathcal{D}_E)$.

If $e' \notin Q(\mathcal{D}_E)$ either:

- $e' \notin \bigcap \mathcal{D}_E[P \cup \{e\}]$: this cannot be the case because e' occurs in $\mathcal{D}_E$ and $P \cup \{e\} \nsubseteq EL$ hence Theorem 3.1 applies with $Q = P$.
- there exists S' such that $Select(P \cup \{e\} \cup S' \cup \{e'\}, \mathcal{D}_E)$ is false whereas $Select(P \cup \{e\} \cup S' \cup \{e'\}, \mathcal{D}_P^{reduced})$ is true. This is impossible from condition $(P2)$ and $S = S' \cup \{e'\}$.

Therefore $Q(\mathcal{D}_E) = Q(\mathcal{D}_P^{reduced})$. □

Lemma 3.2
If the Select predicate is decomposable, then: for every argument $(P, \mathcal{D}_P^{reduced}, EL)$ of $expand(P, \mathcal{D}_P^{reduced}, EL)$, $expand(P, \mathcal{D}_P^{reduced}, EL) = enum\_clo(P, \mathcal{D}_E, EL)$.

Proof: Let Q be an output of $enum\_clo(P, \mathcal{D}_E, EL)$: it is of the form $Clo(P \cup \{e\}, \mathcal{D}_E)$ where $P \cup \{e\}$ is a pattern and $e \notin EL$.

Suppose that it is not outputted by $expand(P, \mathcal{D}_P^{reduced}, EL)$. According to Algorithm 7 it means that:

- either e does not occur in $\mathcal{D}_P^{reduced}$ (Line 2 in Algorithm 7) which is impossible: since e does not belong to EL, it cannot have been suppressed by the dataset reduction returning $\mathcal{D}_P^{reduced}$.
- or $Select(P \cup \{e\}, \mathcal{D}_P^{reduced})$ is false (Line 3 in Algorithm 7). This cannot be the case if the condition $(P2)$ holds.
- or $Clo(P \cup \{e\}, \mathcal{D}_P^{reduced}) \cap EL \neq \emptyset)$ (Line 5 in Algorithm 7). This cannot be the case since according to Lemma 3.1: $Clo(P \cup \{e\}, \mathcal{D}_P^{reduced}) = Clo(P \cup \{e\}, \mathcal{D}_E)$ and $Clo(P \cup \{e\}) \cap EL = \emptyset$ ($P \cup \{e\}$ is outputted by $enum\_clo$ only if $Clo(P \cup \{e\}) \cap EL = \emptyset$ according to Line 7 in Algorithm 5).

Conversely, we prove that a Q outputted by $expand(P, \mathcal{D}_P^{reduced}, EL)$ (Algorithm 7) is also outputted par $enum\_clo(P, \mathcal{D}_P^{reduced}, EL)$ (Algorithm 5, Page 26).

If Q is outputted in Line 7 in Algorithm 7 it means that there exists an e such that:

1. e occurs in $\mathcal{D}_P^{reduced}$ and $Select(P \cup \{e\}, \mathcal{D}_P^{reduced})$ is true, and
2. $EL \cap Clo(P \cup \{e\}, \mathcal{D}_P^{reduced}) = \emptyset$.

From 1., according to the property $(P2)$, the condition $P \cup \{e\} \in \mathcal{F}$ Line 5 in Algorithm 5 is satisfied.

From 2., and Lemma 3.1 stating that $Clo(P \cup \{e\}, \mathcal{D}_P^{reduced}) = Clo(P \cup \{e\}, \mathcal{D}_E)$, the condition Line 7 in Algorithm 5 is also satisfied.

Therefore Q is outputted by $enu\_clo(P, \mathcal{D}_P^{reduced}, EL)$

Theorem 3.2 (Soundness of ParaMiner)
ParaMiner *computes the set of all closed patterns if the Select predicate is decomposable.*

Proof: It is a direct consequence of Lemma 3.2 and from the fact that *enum_clo*() (Algorithm 5, Page 26) is in fact a rephrasing of the Boley et al.'s Algorithm 1 in [BHPW10], which has been shown to compute the set of closed patterns.

We now prove that the *Select* predicate is decomposable for the FIM, CRG and GRI pattern mining problems introduced in Section 2.2. It guarantees that ParaMiner is sound and complete for these problems.

Theorem 3.3
ParaMiner *is sound and complete for the* FIM, CRG *and* GRI *problems.*

Proof: It is straightforward to show that the *Select* predicate for the FIM problem is decomposable, since $Select(P, \mathcal{D}_E)$ is true if and only if P is frequent in $\mathcal{D}_E$. Therefore $Select(P, \mathcal{D}_E) \equiv |\mathcal{D}_E[\![P]\!]| \geq \varepsilon$ (where $\mathcal{D}_E[\![P]\!]$ is the tid support set of P and ε is the frequency threshold).

It is also easy to show that the *Select* predicate is decomposable for the CRG problem, since $Select(P, \mathcal{D}_E)$ is true if and only if P is a connected graph and frequent in $\mathcal{D}_E$. Therefore $Select(P, \mathcal{D}_E) \equiv is\_connected(P) \wedge |\mathcal{D}_E[\![P]\!]| \geq \varepsilon$.

Finally let us consider the GRI problem. Let us recall that in this problem P is a pattern if and only if:

1) its first element is of the form $a^{\uparrow}$

2) and if its tid support set contains a *path* whose size is greater than ε.

Note that the first condition 1) is a constraint that depends only on the pattern P while the condition 2) is a constraint that depends only on the tid support set of P in $\mathcal{D}_E$. This corresponds exactly to requirements for *Select* to be decomposable (Definition 3.1). $\square$

3.2.2 Indexing

An execution of ParaMiner requires a large number of accesses to the dataset to check if a candidate pattern must be outputted or not. For example, computing the support set of a candidate pattern is a common operation in ParaMiner. In each call to *expand*(), it is required at least once for every element that occurs in the reduced dataset (Line 2 in Algorithm 7). However to compute the support set of a pattern, one needs to perform a full pass over the dataset. This makes to many passes over the dataset to expect good performances. A common solution is to have an index which associate each element with the set of transactions in which it occurs, saving computation at the cost of memory space.

Indexes are very common to improve the performances of information finding in large databases. They are redundant representations of the initial data organized a different way in order to provide faster accesses. This technique was first introduced in LCM under the name of *occurrence deliver*.

Given a reduced dataset $\mathcal{D}_P^{reduced}$, the index of $\mathcal{D}_P^{reduced}$ denoted $Index_{\mathcal{D}_P^{reduced}}$ associates each element e occurring in $\mathcal{D}_P^{reduced}$ with the set of the transaction identifiers of the transactions that include e. For any element e, $Index_{\mathcal{D}_P^{reduced}}[e] = \mathcal{D}_p^{reduced}[\![e]\!]$. In ParaMiner, the index of a reduced dataset is built with the *build_index*() function presented in Algorithm 10. We build an index for each new reduced dataset (Line 8 of Algorithm 7, omitted for clarity) and use it to improve the computation of tid sets.

Algorithm 10 Computing the index of a reduced dataset.

```
1: function build_index($\mathcal{D}_P^{reduced}$)
• Require: A reduced dataset $\mathcal{D}_P^{reduced}$.
• Ensure: Returns the index of $\mathcal{D}_P^{reduced}$.
2:     for all transaction $t$ in $\mathcal{D}_P^{reduced}$ do
3:         for all element $e$ in $t$ do
4:             Add the transaction identifier of $t$ to $Index_{\mathcal{D}_P^{reduced}}[e]$
5:         end for
6:     end for
7:     return $Index_{\mathcal{D}_P^{reduced}}$
8: end function
```

In ParaMiner, we check if a an element e occurs in a dataset $\mathcal{D}_P^{reduced}$ (Line 2 in Algorithm 7) by checking whether $Index_{\mathcal{D}_P^{reduced}}[e]$ is not empty.

The index can also be used to speed up the computation of the closure operators. For example, in the generic closure operator (Algorithm 1) as well as in every problem-specific closure algorithms that we have proposed in Section 2.2, the intersection $\bigcap \mathcal{D}_P^{reduced}[Q]$ is required to perform the closure of a pattern Q. Given the index $Index_{\mathcal{D}_P^{reduced}}$ we can efficiently compute this intersection by checking the elements that have the same tid set as Q. The algorithms to compute the intersection is provided in Algorithm 11.

Algorithm 11 Computing the intersection of the support set with indexes.

```
• Require: The index $Index_{\mathcal{D}_P^{reduced}}$ of a dataset $\mathcal{D}_P^{reduced}$, and a pattern $Q$ that is an
  augmentation of $P$ ($Q = P \cup \{e\}$).
• Ensure: Returns the intersection of the support set of $Q$ in $\mathcal{D}_P^{reduced}$.
1: $I \leftarrow \emptyset$
2: for all $e'$ such that $e'$ occurs in $\mathcal{D}_P^{reduced}$ do
3:     //$Q = P \cup \{e\}$, therefore $e'$ belongs to $\bigcap \mathcal{D}_P^{reduced}[Q]$ if it has the same tid support
   set than $e$ in $\mathcal{D}_P^{reduced}$.
4:     if $Index_{\mathcal{D}_P^{reduced}}[e] = Index_{\mathcal{D}_P^{reduced}}[e']$ then
5:         //$e'$ belongs to $\bigcap \mathcal{D}_P^{reduced}[Q]$.
6:         $I \leftarrow I \cup \{e'\}$
7:     end if
8: end for
9: return $I$
```

3.3 Optimizations based on parallelism

The algorithmic optimizations presented in the previous section are mandatory to be able to tackle large datasets. However in order to further improve the performances of ParaMiner and make it usable for practical data-mining problems, it is also important to consider the architecture of modern execution platforms.

In this thesis we focus on platforms with multi-core architectures, that is platforms with more than one processing units (cores) and a main memory that is accessible from every cores. This covers a broad range of execution platforms, from everyone's desktop computer with 2 to 8 cores, to departmental computing servers with up to several hundreds of cores.

In order to be able to run efficiently on these platforms, and to be able to scale efficiently on larger platforms, ParaMiner must be a parallel algorithm. In order to be parallel, the whole computation must be split into independent tasks that can be performed onto different cores. In Section 3.3.1 we first explain how to split the computations in ParaMiner.

Once the global computation has been split into tasks, we must distribute the tasks to the cores available. If the computations are not fairly distributed among the cores *load balancing* issues may arise. If the computations are fairly distributed but the *data* in memory is accessed in a very disorganized manner, *poor data locality* issues may arise. Both reduce the performances of the algorithm, and should be avoided. However improving load balancing typically reduces the data locality and conversely, therefore finding the correct task distribution is a complex problem.

An adequate distribution strategy can be found with a good knowledge of both the algorithm behavior and the execution platform. In order to find efficient task distribution strategies for the range of problems that can be solved by ParaMiner, we designed Melinda. Melinda is the parallel execution engine used to run ParaMiner on multi-core architectures. In Melinda the task distribution strategy can be easily changed in order to fit the algorithm and the execution platform. We present Melinda in Section 3.3.2 and show how it can be used to improve ParaMiner's performances in Section 3.3.3.

3.3.1 Parallel exploration of the enumeration tree

ParaMiner outputs the set of closed pattern by following the enumeration tree described in the previous chapter, therefore splitting the exploration of the enumeration tree into tasks is the natural approach to distribute the computations among the available processing units. In ParaMiner each task is in charge of expanding a node of the enumeration tree (i.e. a call to *expand*()). Each task may create additional tasks to explore the child nodes of the tree. When all the tasks have been completed, that is when no more tasks are pending, the exploration of the enumeration tree is also complete.

Creating and executing tasks comes with a computational overhead. It is worth it if this overhead is small relatively to the computation time required to complete the tasks. However in the deeper levels of the enumeration tree, the calls to *expand*() are typically much simpler due to dataset reduction. When the time required to complete a task becomes close to the time required to create and handle it (i.e. within the same order of magnitude), it is not worth creating a new task. In order to avoid the creation of small tasks, no new tasks are created to perform the descendent calls to *expand*() over a fixed level in the enumeration tree, instead further calls are performed within the same task. More precisely over this fixed depth, *spawn* in Algorithm 12, Line 9 becomes a recursive call to *expand*(), and any further call is performed *sequentially* within the same tasks. An example of splitting is shown in Figure 3.2.

Algorithm 12 ParaMiner's *expand*() procedure with depth controlled *spawn*.

```
1: procedure expand($P$, $\mathcal{D}_P^{reduced}$, $EL$, $depth$)
• Require: A closed pattern $P$, a reduced dataset $\mathcal{D}_P^{reduced}$, an exclusion list $EL$, the depth in the enumeration tree.
• Ensure: Output all closed patterns that are descendent of $P$ in the enumeration tree.
2:     for all $e$ such that $e$ occurs in $\mathcal{D}_P^{reduced}$ do
3:         if $Select(P \cup \{e\}, \mathcal{D}_P^{reduced})$ then
4:             $Q \leftarrow Clo(P \cup \{e\}, \mathcal{D}_P^{reduced})$
5:             if $EL \cap Q = \emptyset$ then
6:                 //P is Q's the first parent
7:                 output $Q$
8:                 $\mathcal{D}_Q^{reduced} \leftarrow reduce(\mathcal{D}_P^{reduced}, e, EL, depth + 1)$
9:                 if $depth \leq depth\_threshold$ then
10:                    spawn $expand(Q, \mathcal{D}_Q^{reduced}, EL, depth + 1)$ //Spawn a new task.
11:                else
12:                    $expand(Q, \mathcal{D}_Q^{reduced}, EL, depth + 1)$ //Perform the recursive call inside the task.
13:                end if
14:                $EL \leftarrow EL \cup \{e\}$
15:            end if
16:        end if
17:    end for
18: end procedure
```

It is important to note that each task can be performed without communicating with other tasks. This property directly derives from the polynomial space enumeration strategy and the first parent test presented in the previous chapter. This is an important benefit for ParaMiner because task-wise communications are time consuming and induce dependencies among the tasks. Since there is no task dependencies in ParaMiner, no particular order is required to ensure its soundness.

However, the tasks should be distributed among the cores in order to preserve ParaMiner from important parallel issues such as load unbalance or poor cache locality.

Poor cache locality

The main memory is an important resource that is shared by every cores of the processors. However, the delay required to fetch a value from the main memory to the core is long when compared to the delay required to perform any other operation on this value (e.g. an arithmetic operation). In order to reduce this delay, the cores embed caches memories. A cache contains a copy of the memory space that has been recently accessed by the core.

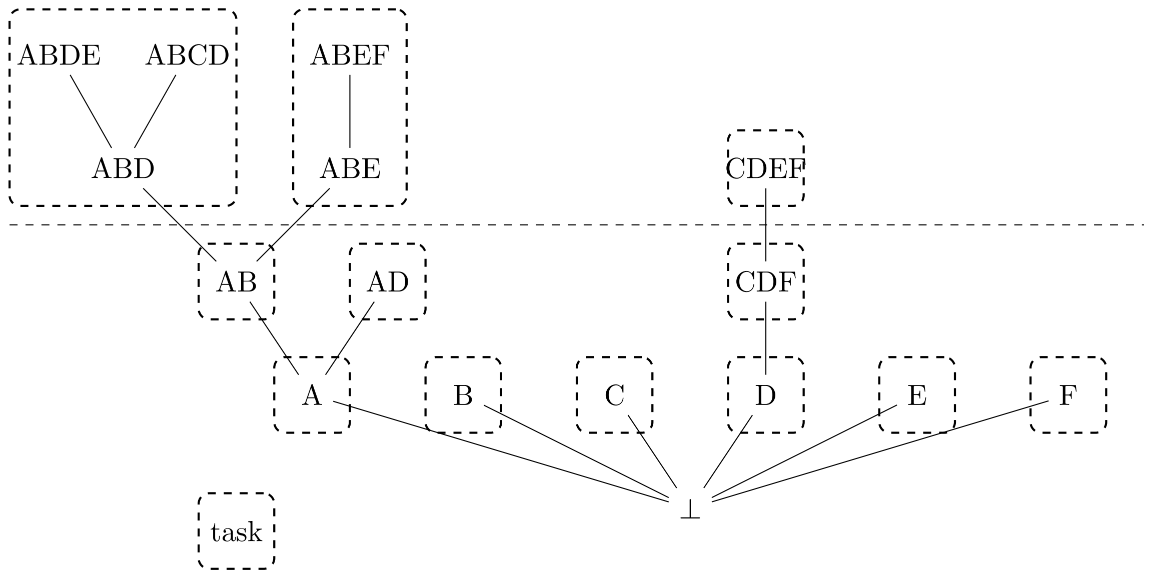


Figure 3.2: An enumeration tree split into tasks, over depth 2 in the enumeration tree (dashed line), the expands calls are performed sequentially into a single task.

If the cache holds a copy of a value, the delay required to access this value is two to three orders of magnitude smaller than in the main memory. If the value is not contained in the cache, it has to be loaded from the main memory into the cache. Therefore in order to keep the cores busy and ensure a good computational throughput, it is important to keep the number of out-of-cache memory accesses as low as possible.

In ParaMiner numerous tasks exhibit *data-affinities*. Two tasks share affinities, if they access the same data-structures from the memory. For example, when a task create a dataset (Line 8 in Algorithm 12), this dataset will be further used by the child task. In order to reduce the number of out-of-cache memory accesses, it is important to execute those task on the same core because the cache already contains the dataset.

Conversely if the tasks sharing data-affinities are executed on different cores the same dataset may have to be transfered twice from the memory to distinct caches, this is an important concern because the canal used to transfer data from the memory to the cache is a shared resource, hence subject to contention when a lot of data transfers are performed simultaneously by several cores. Without a special care, an algorithm may not be able to scale on a large computation platforms with a large number of cores.

Preserving the cache locality implies distributing the tasks with respect to the core on which they have been created, this constraint on the way to distribute the tasks may lead to load unbalance.

Unpredictable tasks size and load unbalance

Load unbalance is another well known problem occurring in many parallel algorithms. It occurs when cores are assigned tasks that requires different amount of time to be completed. If one core is assigned a short task and the other a much longer task, the first core will stay idle until the second core has finished processing its task. This will reduce the number of tasks completed per time unit, hence increase the execution time. The Figure 3.3 shows two executions of the same set of tasks on a two cores platform. The execution in (a) shows load unbalance (core 1 is idle during the hatched section) whereas the execution in (b) does not. As a consequence, even if the total amount of computation required is identical in both scenarios, the time required to perform all the tasks is shorter in (b) than in (a).

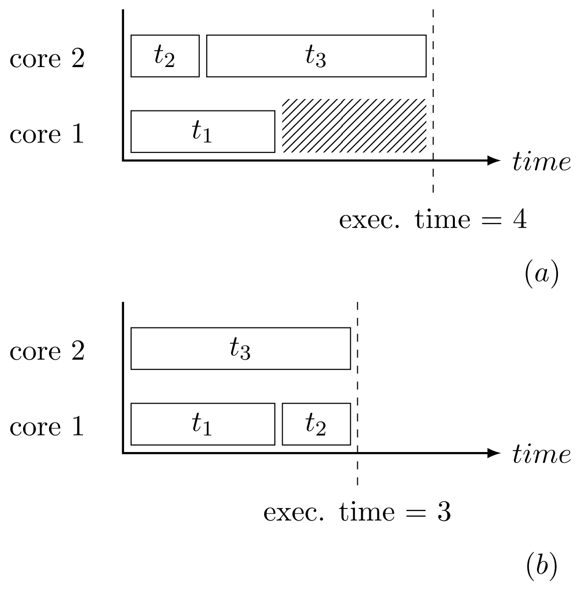


Figure 3.3: Unbalanced execution (a) vs. balanced execution (b).

An execution of ParaMiner is driven by the enumeration tree which depends on the input dataset. As a consequence, the number of tasks is unknown in advance and the amount of time required to complete each task is highly variable. For example in Figure 3.2, the task in charge of expanding AB spawns more and possibly larger tasks, than the task in charge of expanding E. Since the enumeration tree is unknown until the tasks are completed, no a priori distribution of the tasks can be made at the initialization of the algorithm. We call algorithms whose execution flow is driven by their input data *data driven algorithms*. For those algorithms, naive task distribution strategies fail to provide correct load balancing (cf. Related Works in Chapter 5) and dynamic and adaptive distributions must be used to avoid this issue.

Discussion

Load unbalance and cache locality are two orthogonal problems and solving one can worsen the other, preventing any performance increase. A good distribution strategy is typically a fair tradeoff between load unbalance and cache locality.

The pattern mining problems that can be solved by ParaMiner are not equally subject to cache locality and load unbalance. For example, load unbalance happens in frequent tree and graph mining applications[TP09, BPC06] whereas memory issues are frequent in itemset mining[NTMU10]. These papers have proposed efficient ad-hoc solutions designed after an extensive study of a given algorithm and a given problem. (cf. Related Work in

Chapter 5). However those solutions are hardly generalizable to ParaMiner which has to be efficient for pattern mining problems that can exhibit very different behaviors.

In order to find a satisfying solution for distributing the tasks in ParaMiner, Melinda, the parallelism engine used to drive the distribution and the execution of the tasks in ParaMiner can be parameterized with various task distributions strategies. In Melinda, creating and adapting strategies is fast and simple.

Although it is illustrated with ParaMiner within the next section, Melinda has been used as the execution engine of several other pattern mining algorithms such as DigDag[TTN+07, NTM08], PLCM[NTMU10], and PGLCM[DLT10].

3.3.2 Melinda: a parallel engine adapted to pattern mining algorithm

Melinda has been inspired by the Linda model designed by Gelernter in [Gel85]. The main components in Melinda are *tuples* and the *tuple-space*. Tuples are records with named fields where each field is defined according to the algorithm needs (in our case: ParaMiner). The tuple-space is a memory space that can contain tuples only. The tuples can be dropped in, or withdrawn from the tuple-space using two primitives, respectively *put*() and *get*(). In order to avoid errors due to concurrent accesses to the tuple-space, the *put*() and *get*() primitives are implemented with the adequate synchronization. It is important to note that there is no other mean to access to the tuple-space.

When there are no tuples in the tuple-space, the *get*() function is blocking, and the core calling the function is put on a waiting queue. The program terminates when all the cores are pending in the waiting queue.

In ParaMiner, we create a new tuple for each task. We recall that a task is a node or a sub-tree of the enumeration tree. Since each node of the enumeration tree is discovered by a call to the *expand*() procedure, each task can be defined as the list of parameters of a call to *expand*(). Therefore, in order to use ParaMiner together with Melinda, we create a tuple containing this list of parameters. In addition, to perform the splitting as it is described in Figure 3.2, we also store in the tuple the depth in the tree at which it has been generated. Any tuple in ParaMiner is a 4-fields tuple $[Q, \mathcal{D}_Q^{reduced}, EL, depth]$, where Q is the closed pattern, the $\mathcal{D}_Q^{reduced}$ is the reduced dataset of Q, EL is the exclusion list and $depth$ the depth generation. Notice that in practice, we only store in the tuple a pointer to the dataset, this to avoid expensive copies of the datasets into the tuple-space.

In order to execute the tasks, each core executes a function that pulls tuples from the tuple-space with the *get*() primitive (Algorithm 13 procedure *run_tasks*()). For each tuple extracted, the core calls the *expand*() procedure with the list of parameters contained in the tuple. Notice that each call to *expand*() may create new tuples if necessary. This carries on until the tuple-space is empty.

With this settings, the **spawn** instruction Line 9 in Algorithm 7 can be implemented as shown in Algorithm 13 (only the lines relevant to spawning a new task are shown here, i.e. Lines 9 to 13 in 12).

Melinda ensures correct data sharing among the cores and global termination detection. Since it hides low level details it can be used by programmers with no experience with parallel programming.

Algorithm 13 The mechanism used to implement the *spawn* instruction in Algorithm 7.

• The *expand*() procedure with tuples

```
1: procedure expand($P$, $\mathcal{D}_P^{reduced}$, $EL, depth$)
   ...
2:     //In order to spawn a task, creates a tuple and put it in the tuple-space.
3:     if $depth \leq depth\_threshold$ then
4:         $put([Q, \mathcal{D}_Q^{reduced}, EL, depth + 1])$
5:     else
6:         $expand(Q, \mathcal{D}_Q^{reduced}, EL, depth)$
7:     end if
8: end procedure
```

• The function executed by the cores

```
1: procedure run_tasks
2:     while tuple-space is not empty do
3:         $[P, \mathcal{D}, E, depth] = get()$
4:         $expand(P, \mathcal{D}, E, depth)$
5:     end while
6: end procedure
```

In the original Linda, the tuples are heterogeneous and can be queried by specifying the fields values. A tuple is extracted from the tuple-space only if the specified fields matches. This approach is of interest for ParaMiner and other data-driven algorithms because the tuples contains relevant information that can be exploited to build an adequate distribution of the tuples.

However, Melinda must be able to handle a large number of tuples and an intensive usage of *get*() and *put*(). In addition, we do not need such an advanced querying system, instead, we need a way to drive the tuple distribution that can be tuned easily in order to match the pattern mining problem and the execution platform. In Melinda, we use *internals*. Internals are arbitrary subdivisions of the tuple-space that can be used to sort tuples.

3.3.3 Optimizing ParaMiner's performances with Melinda's tuple distribution strategies

When threads are pulling tuples from the tuple-space, Melinda is in charge of distributing them according to a strategy that can be redefined by the user. If no strategy is provided, the tuples are retrieved in a *first in first out* manner. However, different strategies can be used to preserve data locality or to improve the load balancing.

In order to define new strategies, the tuple-space is divided into internals. Each internal is a disjoint section of the tuple-space and can be used to group tuples together according to whatever trait is relevant to the algorithm. For example, tuples can be grouped according to the size of their reduced dataset. In addition to this, Melinda provide information relative to the execution platform. Therefore the tuples can also be grouped according to platform information such as the identifier of the core that have generated the tuple.

A strategy is then defined by a pair of functions (*distribute*(), *retrieve*()). The distribute function is called by the *put*() primitive, right before putting the tuple inside the tuple-space (Line 2 in Algorithm 14). The distribute function returns an integer that is used as a internal identifier. If the internal does not exists it is created dynamically. Conversely, when an internal is empty it is destroyed. The *retrieve*() function is called by the *get*() primitive. The integer value returned by *retrieve*() is used as an internal identifier (Line 2 in Algorithm 15) from which *get*() picks the tuple to return. If the internal does not exists or is empty, a tuple is taken from any other internal. This way the soundness of the algorithm is not altered.

Algorithm 14 The *put*() primitive.

```
1: function put(t: tuple)
2:     $internal\_id \leftarrow distribute(t)$
3:     if $Internals[internal\_id]$ does not exists then
4:         $create\_internal$
5:     end if
6:     $I \leftarrow Internals[internal\_id]$
7:     Stores $t$ in internal $I$.
8: end function
```

Algorithm 15 The *get*() primitive.

```
1: function get
2:     $internal\_id \leftarrow retrieve()$
3:     while $Internals[internal\_id]$ is empty or does not exist do
4:         $internal\_id \leftarrow next\_internal\_id$ //Pick another internal id.
5:     end while
6:     //$Internals[internal\_id]$ is a non-empty internal.
7:     return the first tuple $t$ from $Internal[internal\_id]$.
8: end function
```

Now we can easily implement a strategy to improve the load imbalance issue mentioned in Section 3.3.1. For example, we are aware that tuples with large datasets typically require more time to be completed than tuples with small datasets. In Algorithm 16 the distribute function stores large datasets in Internal 1 and small datasets in Internal 2 (Line 3 to 6). The retrieve function returns 1, therefore if the corresponding internal is not empty, the tuples will be retrieved from it, otherwise they will be retrieved from another one (here the Internal 2). It has been proven by Graham et al. in [Gra66] that this way of distributing the tasks provides better load balancing than a random task distribution strategy.

Another strategy can be used to promote local memory accesses and preserve cache locality. In ParaMiner, if a task spawns another task, they require accesses to the same data-structures: a parent call to *expand*() creates the sub-dataset that is used by the child call to *expand*(). However if the child task is performed on a core with a different cache, the dataset has to be transfered again into the other cache, which costs time. The strategy proposed in Algorithm 17 ensure that a tuple is delivered to the core that has generated the tuple, as far as possible. It does so by storing the tuples from the same cores within the same internal.

Algorithm 16 MELINDA strategy to improve the load balancing.

```
function Distribute(tuple:[$P, \mathcal{D}, EL, depth$])
    //Stores the tuples with large datasets in internal 1 and the tuples with small datasets in internal 2.
    if $||\mathcal{D}||$ is large then
        return 1
    else
        return 2
    end if
end function
function Retrieve()
    //Retrieve tuple with large datasets in priority.
    return 1
end function
```

Algorithm 17 MELINDA strategy to promote local memory accesses.

```
function Distribute(tuple:[$P, \mathcal{D}, EL, depth$])
    return $core\_identifier$
end function
function Retrieve()
    return $core\_identifier$
end function
```

3.4 Conclusion

In this section, we have proposed PARAMINER, which is an algorithm adapted to pattern mining problems as defined in the previous chapter. Although PARAMINER is generic it is made efficient by generalizing state of the art algorithmic optimizations such as the database reduction technique and indexing techniques. In addition PARAMINER is a parallel algorithm hence it can benefit from modern parallel architectures. As a consequence any one with basic programming skills and a dataset to mine can benefit from state of the art pattern mining techniques *and* from parallelism.

In addition, pattern mining is a challenging problem for the parallelism community. Indeed naive parallelizations of pattern mining typically fail to scale even on machines with few cores, and even with a ad-hoc algorithm it is sometime hard to reach the maximal speed on a given execution platform due to the complex interactions between the algorithm behavior and hardware components. In the past years various researchers from both communities have worked together to build efficient ad-hoc parallel algorithms (see Section 5). Thanks to the MELINDA executing engine used in PARAMINER, we can quickly benefit from these works by implementing strategies into MELINDA. In addition, PARAMINER together with MELINDA is an interesting framework for quick experimentations regarding an unknown pattern mining problem. It also makes the experiments comparable with other execution strategies or other pattern mining problems. Hence it is and valuable tool for a better understanding of parallel pattern mining algorithms.

Within the next section, we validate the efficiency of PARAMINER by comparing it with other ad-hoc algorithms.

Chapter 4

Experiments

Contents

In this chapter we report on thorough experiments that we have conducted to evaluate ParaMiner's performances in terms of execution times and also in terms of scalability on large computation platforms. We give the details of our experimental settings in Section 4.1.

In Section 4.2, we experimentally demonstrate the efficiency of dataset reduction by measuring the gain offered by this optimization presented in the previous chapter. The experiments show that dataset reduction makes drastically faster the computation of closed patterns and that it is a key optimization to tackle large datasets.

In Section 4.3, we demonstrate the benefit of parallelism on several types of computation platforms. With the raise of multi-core processors, parallelism is now available in most standard computers. We thus evaluate the gain offered by parallelism on a 4-core laptop computer. These experiments show that ParaMiner can fully exploit this new form of computational power.

We also demonstrate that ParaMiner can benefit from larger computation platforms by conducting experiments on a 32-core computation server with 64GiB of memory. Thanks to this additional computational power, ParaMiner successfully tackle the problem of gradual itemset mining on real world datasets.

Those larger platforms have complex architectures and exploiting them efficiently have been an active research topic for decades. Designing efficient parallel pattern mining algorithms is a particularly challenging problem due to irregular and memory intensive computations. Experiments reveal important parallelism issues also identified in [TP09, GBP$^+$05] or [NTMU10] that prevent ParaMiner from reaching the theoretical performance of the computing platforms. From those experiments, we propose solutions and demonstrate their feasibility with Melinda strategies.

In Section 4.4, we demonstrate experimentally that ParaMiner is generic *and* efficient, thanks to the above optimizations. For doing so, we compare the performances of ParaMiner with several state of the art specific algorithms designed for particular pattern mining problems. Although these algorithms are the fastest available, ParaMiner is competitive in all cases. For the problem of gradual itemset mining, it outperforms the state of the art algorithm by several orders of magnitude.

4.1 Experimental settings

We have implemented ParaMiner in `C++`. The *Select* and *Clo* operators for the different problems proposed in this thesis are also implemented in `C++` although it is technically possible to interface ParaMiner with other common programming languages such as `Java` or `Python`. The Melinda library is implemented in `C`. Unless otherwise mentioned, the Melinda strategy in use is the default strategy where the tuples are pushed into and pulled from the tuple-space in the *first-in, first-out* order.

All the algorithms including algorithms from different authors are compiled with the `gcc` compiler with the same settings and with compiler optimizations fully enabled (`-O3` flag in `gcc`). They are executed on computers running the `GNU/Linux` operating system.

We execute the algorithms on two distinct computing platforms namely *Laptop* and *Server*. Their hardware configuration are presented in Table 4.1. *Laptop* is a high-end laptop computer with four cores. Its fairly standard configuration is similar to what most data-miners use as their main computer. Experiments conducted on this platforms thus represent what anyone can get by running ParaMiner on his/her own computer.

	Laptop	*Server*
# cores	4	32
Memory (GiB)	8	64
Processor type	Intel Core i7 X900	4 x Intel Core i7 X7560
Processor frequency (GHz)	2	2.27
Cache size (MiB)	8	4 × 24
Memory bus bandwidth (GiB/s)	7.9	9.7

Table 4.1: Specifications of the computation platform

Server is a computation server with 32 cores and 64 GiB of memory. It is four times the size of *Laptop* in terms of number of cores and eight times the amount of memory available. However it is important to note that other important characteristics are not dimensioned in these proportions. For example the *bus memory bandwidth* is roughly the same on both platforms: 7.9 GiB/s on *Laptop* vs 9.7 GiB/s on *Server*. This will imply important issues discussed later in this chapter.

4.2 Experimental evaluation of dataset reduction

In order to evaluate the impact of ParaMiner's dataset reduction, we first measure the average *reduction factor*. The reduction factor of a given reduced dataset $\mathcal{D}_P^{reduced}$, is the ratio between its size and the size of the input dataset $\mathcal{D}_E$: $reduction\_factor = \frac{||\mathcal{D}_E||}{||\mathcal{D}_P^{reduced}||}$.

We compute the average reduction factor for all the reduced datasets built during an execution of ParaMiner. Given a dataset $\mathcal{D}_E$ and a set of closed patterns $\mathcal{C}$ in $\mathcal{D}_E$, the average reduction factor can be computed with the following formula:

$$average\_reduction\_factor = \frac{\sum_{P\in\mathcal{C}} ||\mathcal{D}_E||/||\mathcal{D}_P^{reduced}||}{|\mathcal{C}|}$$

In the following experiments we also present the *average dataset size*, which is given by the following formula: $average\_dataset\_size = \frac{\sum_{P\in\mathcal{C}} ||\mathcal{D}_P^{reduced}||}{|\mathcal{C}|}$.

4.2.1 Reduction factors for the FIM problem

We compute the average reduction factors for two datasets, namely: BMS-WebView-2 and Accidents. These two datasets are from the FIMI repository [Goe03] which contains a large number of itemset mining datasets commonly used to evaluate the performances of frequent itemset mining algorithms. The characteristics of several datasets from the FIMI repository are shown in Table 4.2.

In addition to traditional metrics such as the size of the ground set or the number of transactions, the *density* has been introduced by [GNDR11] as the ratio between the largest dataset that is possible to build with a given ground set and a given number of transactions, and the number of actual elements occurring in this dataset. It is important

dataset name	ground set size $\lvert E\rvert$	# transactions $\lvert\mathcal{D}_E\rvert$	dataset size $\lvert\lvert\mathcal{D}_E\rvert\rvert$	Density (%) $\frac{\lvert E\rvert\lvert\mathcal{D}_E\rvert}{\lvert\lvert\mathcal{D}_E\rvert\rvert}\times 100$
BMS-WebView-2	3,340	77,512	320,601 (1.2MiB)	0.14
T40I10D100K	942	100,000	3,960,507 (15MiB)	4.20
Connect	129	67,557	2,904,951 (11MiB)	33.33
Mushroom	119	8,124	186,852 (0.7MiB)	19.33
Accidents	468	340,184	11,500,870 (43MiB)	7.22

Table 4.2: Characteristics of FIM datasets available in the FIMI repository. The bracketed value in the size field is an approximation of the size of the dataset once loaded into the memory.

to consider the density because some algorithms exhibit variable performances depending on the density[UKA05, NTMU10].

We run our first experiments on BMS-WebView-2 and Accidents because they are both fairly large and real datasets. BMS-WebView-2 is *very sparse*, i.e. it has a low density, whereas Accidents is *very dense*.

We present in Figure 4.1 (*a*) the average dataset size and in (*b*) the average reduction factor for ParaMiner on BMS-WebView-2 executed with various frequency thresholds. The frequency thresholds are given relatively to the size of the input dataset decreasingly from 0.09% to 0.02%. Executions of ParaMiner with lower frequency thresholds outputs more closed patterns and are thus more complex problems.

In Figure 4.2 (*a*) we present the average dataset size and the average reduction factor in (*b*) for the dense dataset Accidents. Due to its higher density, the number of closed patterns in accidents is much higher and is thus tackled with much higher frequency thresholds from 90% to 20% in order to have a problem difficulty comparable with the previous experiment.

These results show that the dataset can be reduced up to 600 times in the sparse dataset BMS-WebView-2 (*a*), and up to 20,000 times with the dense dataset Accidents (*b*). For both kinds of datasets, dataset reduction is thus able to efficiently reduce the size of the dataset, alleviating the computations needed to compute *Select* and *Clo* (cf. Section 4.2.3).

It is important to note that the reduction factor increases as we reduce the frequency threshold. This is due to the fact that the reduced dataset of a pattern only contains the transactions that include this pattern (See Paragraph 3.2.1, Page 35). Hence the reduced dataset of an itemset included in a small number of transactions is typically much smaller than the original dataset. This makes the dataset reduction a key optimization for tackling large datasets with low frequency threshold.

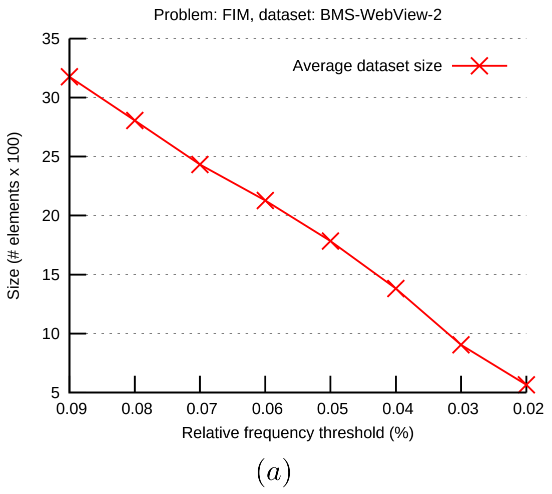
The rest of the reduction is achieved by removing elements that belong to the exclusion list (described in Paragraph 3.2.1). This is particularly effective when performing reduction on patterns occurring on the right side of the enumeration tree where the exclusion lists are large. It is not uncommon to have a single call to *reduce*() reducing a large dataset to few elements.

4.2.2 Reduction factors for crg and gri

In order to compute the reduction factors achieved by ParaMiner for the relational graph mining problem (crg), we use a real world gene network dataset: Hughes. This dataset is made of DAGs, where each DAG represents a potential gene interaction network built with the algorithm in [IGM01] from *micro array experiments* [HMJ$^+$00]. In this dataset, there are 1000 graphs, having an average of 245 genes and 270 edges.

The datasets used for graduals itemset mining (gri), I500 and I4408, are also real datasets from DNA experiments describing gene expressions in the framework of breast cancer. I500 has 500 attributes and 109 lines, I4408 has 4408 attributes and 109 lines as well. Mining these datasets is a complex process: in order to evaluate the reduction factor, we only use the smallest dataset: I500.

Once they are encoded into our framework these datasets can be compared with the same



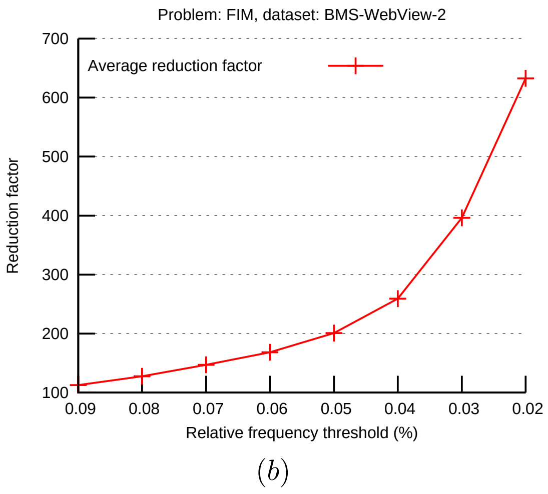
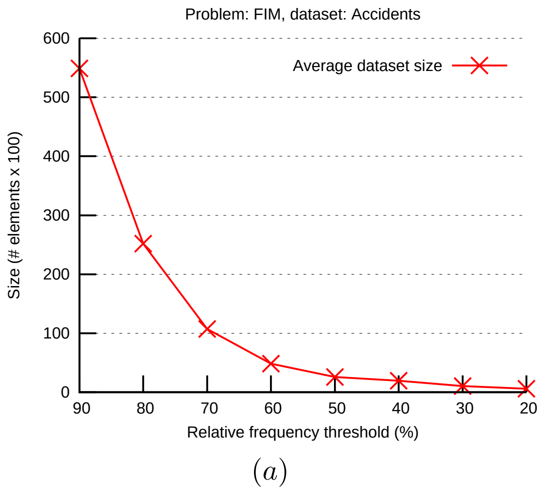


Figure 4.1: Impact of dataset reduction for the FIM problem on BMS-WebView-2 (sparse). Average reduced dataset size in (a), and average reduction factor in (b).



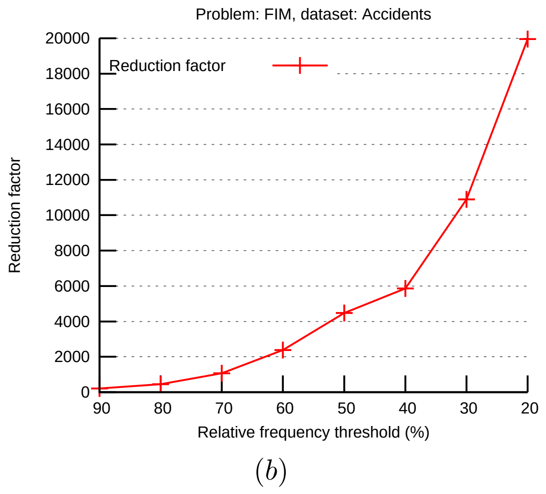


Figure 4.2: Impact of dataset reduction for the FIM problem on Accidents (dense). Average reduced dataset size in (a), and average reduction factor in (b).

dataset name	ground set size $\lvert E\rvert$	# transactions $\lvert\mathcal{D}_E\rvert$	dataset size $\lvert\lvert\mathcal{D}_E\rvert\rvert$	Density (%) $\frac{\lvert E\rvert\lvert\mathcal{D}_E\rvert}{\lvert\lvert\mathcal{D}_E\rvert\rvert}\times 100$
Hughes	794	1,000	270,985	34.13
I500	1008	11,556	5,824,224	50
I4408	8824	11,556	50,985,072	50

Table 4.3: Characteristics of encoded datasets: Hughes(CRG problem), I500 and I4408 (GRI problem).

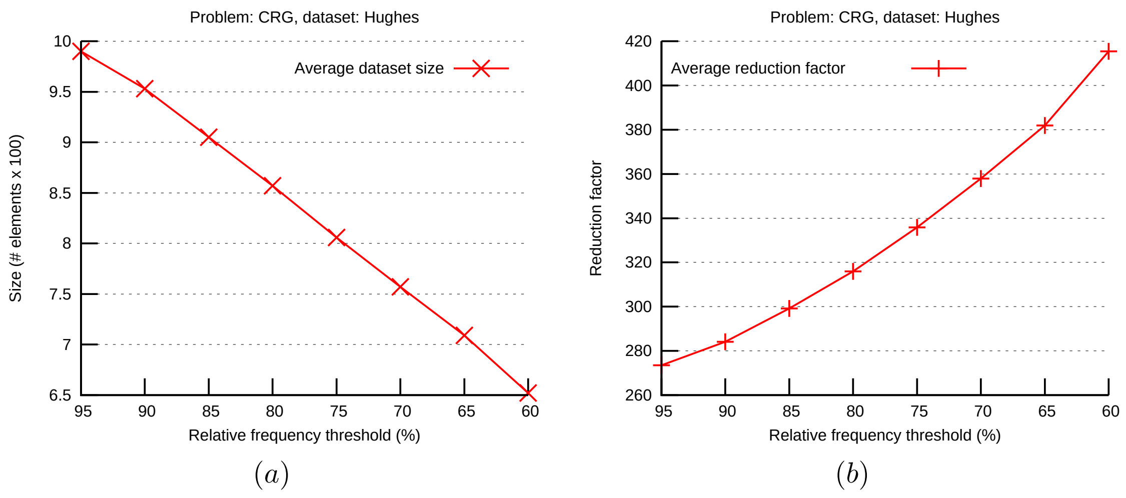


Figure 4.3: Impact of dataset reduction for the CRG problem on Hughes. Average reduced dataset size in (a), and average reduction factor in (b).

metrics. The characteristics of the encoded datasets Hughes, I500, and I4408 are shown in Table 4.3.

The reducing factors for CRG and GRI are shown in Figure 4.4. Again, ParaMiner exhibits important reduction factors: over 400 times in the GRI problem, and over 85 times in the GRI problem. We demonstrate the important impact of the reduction on the execution times in the following section.

4.2.3 Performance impact of dataset reduction

In order to evaluate the performance impact of the dataset reduction we compare ParaMiner with ParaMiner$_{no\_dsr}$ which is the same algorithm except all the computations are performed within the original dataset.

More precisely, ParaMiner$_{no\_dsr}$ is Algorithm 7, Page 34 in which we have performed the following changes:

- $\mathcal{D}_P^{reduced}$ or $\mathcal{D}_Q^{reduced}$ are replaced by $\mathcal{D}_E$, line 3, 4, and 9.
- No call to *reduce*() is performed Line 8.
- Line 2: **forall** e such that e occurs in $\mathcal{D}_P^{reduced}$ **do** ...
  is replaced by: **forall** e such that $P \cup \{e\}$ occurs in $\mathcal{D}_E$ **do** ...

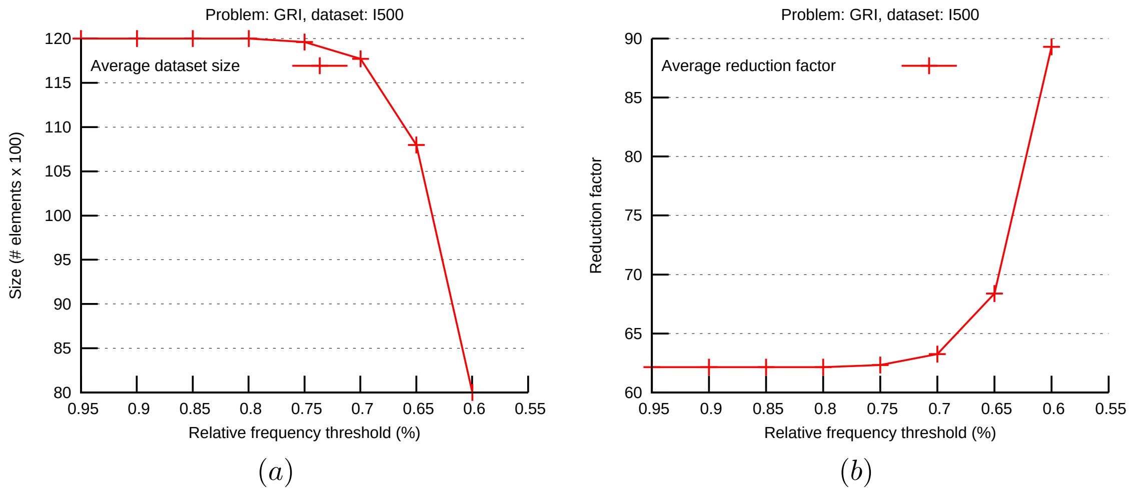


Figure 4.4: Impact of dataset reduction for the CRG problem on Hughes. Average reduced dataset size in (a), and average reduction factor in (b).

Replacing Line 2 that way is mandatory to ensure the soundness of ParaMiner$_{no\_dsr}$.

We then compare the efficiency of ParaMiner$_{no\_dsr}$ and ParaMiner by comparing the execution times required to mine closed frequent itemsets from the Mushroom dataset. Both algorithms are executed on *Laptop*, using only one core (sequential execution). The results are shown in Figure 4.5.

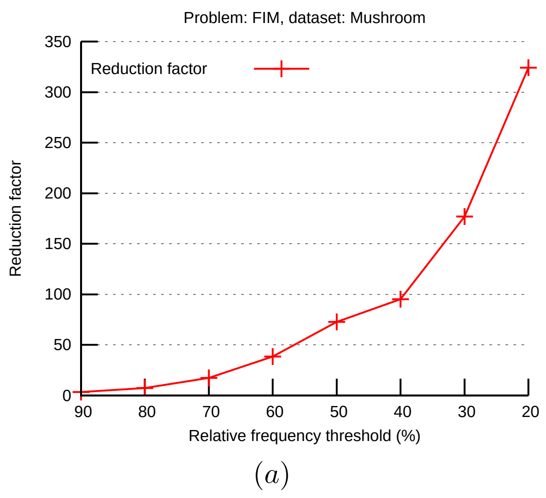
We do not run these experiments on Accidents or BMS-WebView-2 due to the limited capabilities of ParaMiner$_{no\_dsr}$: instead we run them on Mushroom which is a smaller dataset (see Table 4.2). The average reduction factors performed by ParaMiner on this dataset are shown in Figure 4.5 (a). The execution times shown in Figure 4.5 (b) clearly show that dataset reduction reduces the amount of computation required to tackle the dataset. As the consequence ParaMiner$_{no\_dsr}$ is outperformed by one to two orders of magnitude.

Although dataset reduction is an important optimization, the pattern mining problem is still a time consuming problem. We made ParaMiner a parallel algorithm in order to fully exploit the parallelism provided by multi-core architectures. In the following section we evaluate the benefits offered by parallelism.

4.3 Experimental evaluation of parallelism in ParaMiner

In order to demonstrate the gain of parallelism, we first run experiments on *Laptop*. These experiments show that by correctly exploiting multi-core architectures we can turn most standard desktop or laptop computer into a powerful computation platform able to tackle reasonably sized pattern mining problems. This is an interesting result to pattern mining practitioners that need pattern mining to be a process as interactive as possible.

We then evaluate ParaMiner's efficiency on larger computation platforms by experimenting on *Server* which is a 32-core computation server. This type of server can be useful to tackle more complex problems such as the GRI problem or problems with larger



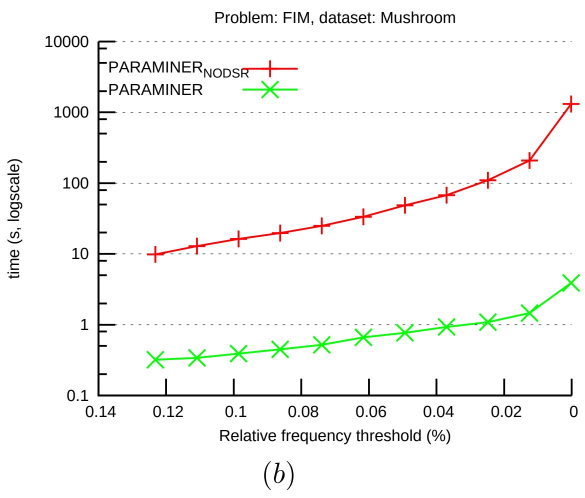


Figure 4.5: ParaMiner vs ParaMiner$_{no\_dsr}$, sequential execution times on the dataset Mushroom (a). The average reduction factor for this dataset is shown in (b).

dataset size and lower frequency thresholds. However due to the large number of interacting hardware components, these platforms can exhibit important issues such as contention on the communication channel connecting these components: the memory bus. After a first evaluation of ParaMiner's performance, we discuss the results and conduct further experiments to understand and ParaMiner's behavior on this type of platforms.

In order to evaluate the parallel performances of ParaMiner, we measure both the execution times and the *speedups*. The speedup is obtained by dividing the time required for an execution restricted to one core, by the time required for an execution exploiting a number n of cores.

$$speedup_n = \frac{\text{execution time using 1 cores}}{\text{execution time using } n \text{ cores}}$$

A parallel algorithm has a good speedup with n cores, the speedup approaches n. The speedup is a well recognized metric to evaluate the ability of an algorithm to *scale* up with a large number of cores.

4.3.1 Parallel performance evaluation of ParaMiner on *Laptop*

The execution times and the speedups of ParaMiner on BMS-WebView-2 and Accidents (fim) are shown in Figure 4.6 and Figure 4.7. The execution times and the speedups on Hughes (crg) are shown in Figure 4.8. Due to a large memory requirements, experiments for the gri problem were only conducted on *Server*.

In Figure 4.6 (a), 4.7 (a) and 4.8 (a), the upper curves show the execution times for an execution of ParaMiner restricted to one core, the lower curves show the execution times of ParaMiner exploiting all the available cores (four cores on *Laptop*). The (b) chart on the right side of each figure shows the corresponding speedup achieved.

We first observe that in both experiments the speedup is lower for the executions with higher frequency threshold values. However, those executions all complete within a very short time delay (less than five seconds). The lower speedup is due to the significant

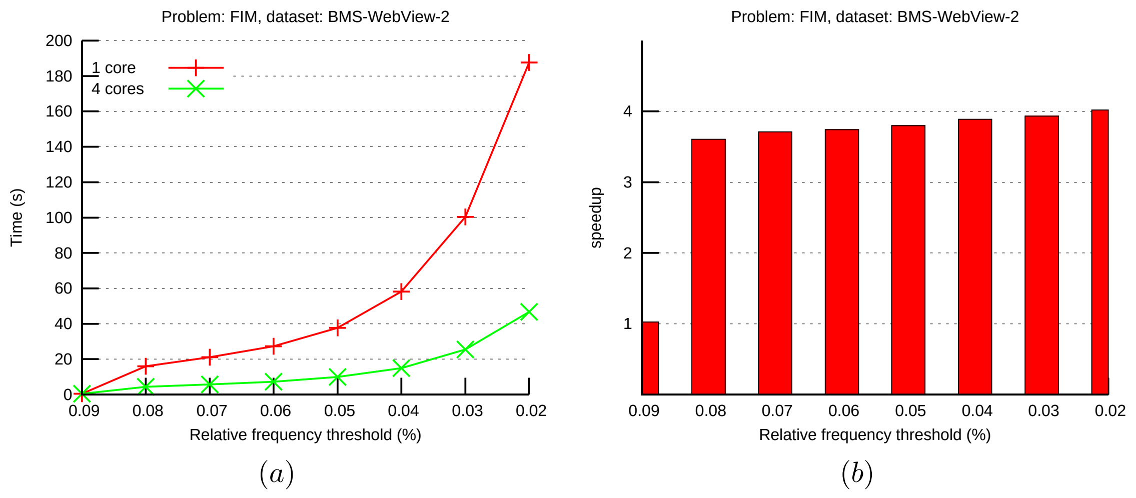


Figure 4.6: Runtimes on *Laptop*: Execution times for the FIM problem on BMS-WebView-2 (a) and speedups (b).

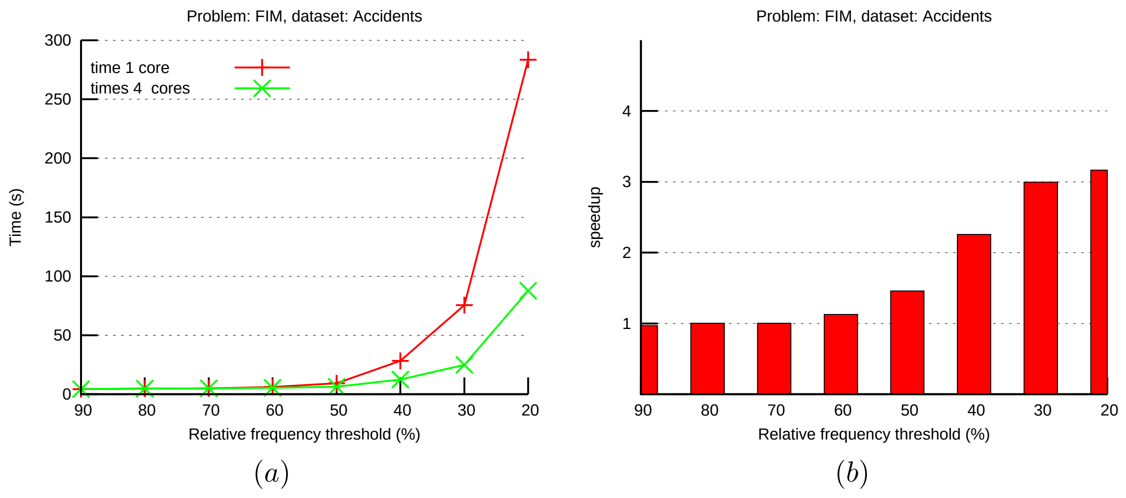


Figure 4.7: Runtimes on *Laptop*: Execution times for the FIM problem on Accidents (a) and speedups (b).

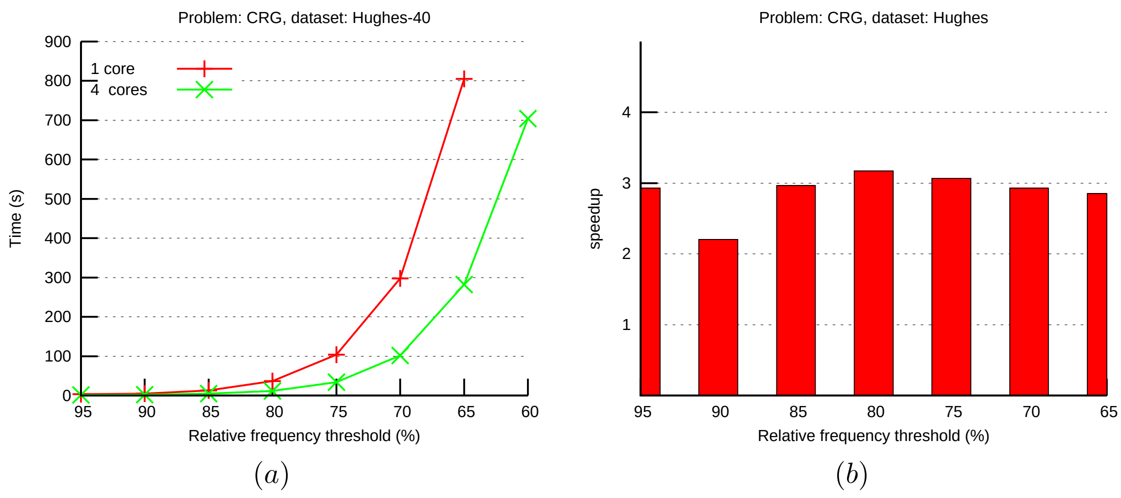


Figure 4.8: Runtimes on *Laptop*: Execution times for the CRG problem on Hughes (a) and speedups (b).

overheads required to create tasks and distribute them among the multiple cores. If the total computation time is short, it is not worth using many cores. However since this effect quickly disappears when tackling the datasets with lower threshold values, we do not consider this to be an important issue.

The speedup bars in Figure 4.6 (b), show that ParaMiner performs almost four times faster when exploiting the four cores available on *Laptop*. To flesh out these numbers on real data analysis scenarios, this means that a 12 hour computation, wasting one work day, can be done in three to four hours.

The speedup bars in Figure 4.7 (b), and 4.8 (b) show a 3 times speedup which is also an important benefit for the user of ParaMiner. However this is not the maximum reachable speedup on a computer with four cores. We observe similar behaviors on the *Server*, and thus we will discuss this in the following section by running additional experiments.

These experiments reveal the significance of designing parallel pattern mining algorithms. They are the only algorithms that can exploit the new form of computational power. By exploiting four cores instead of one, ParaMiner is able to perform three to four times faster.

4.3.2 Parallel performance evaluation of ParaMiner on *Server*

We now evaluate the scalability of ParaMiner on larger platforms. We run our experiments on a 32-core computation server. Thanks to this additional computational power, we are able to tackle more complex problems such as the GRI problem.

The execution times and the speedups of ParaMiner for the FIM problem are shown in Figure 4.9 and 4.10. The executions times and speedups for the CRG problem are shown in Figure 4.13. Finally, the execution times and speedups for the GRI problem are shown in Figure 4.11, Figure 4.12.

We first discuss the execution times and the speedup of ParaMiner for the GRI prob-

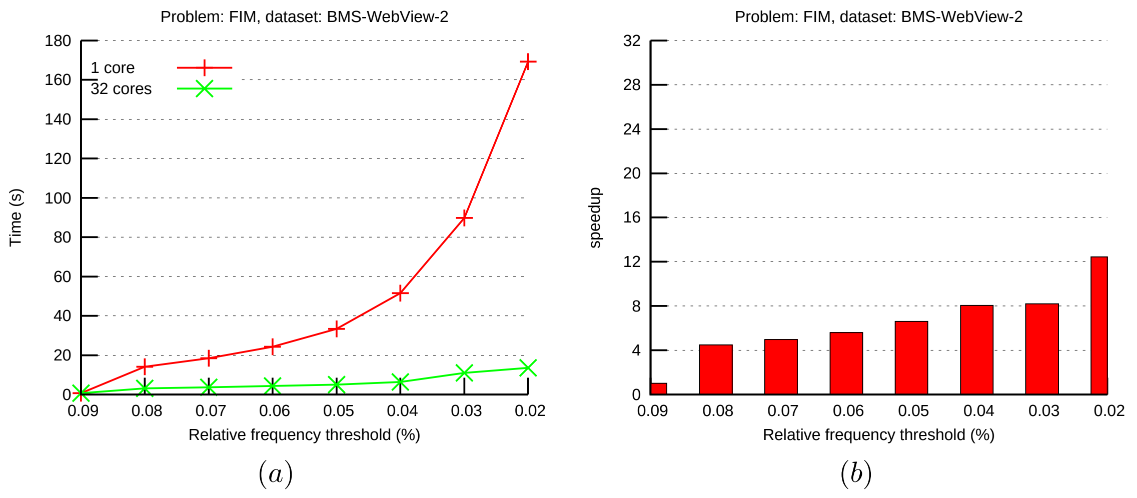


Figure 4.9: Runtimes on *Server*: Execution times for the FIM problem on BMS-WebView-2 (a) and speedups (b).

lem. ParaMiner exhibits good execution times for both datasets I500 and I4408. It is important to note that these dataset were both out of reach until recent work by Do in [DLT10]. On the larger dataset I4408, ParaMiner performs up to 30 times faster than on a single core execution. As a consequence, we are able to mine the dataset I4408 for graduals itemsets within few minutes instead of several hours.

The speedup reached by ParaMiner is less satisfying when it comes to mine frequent itemsets in BMS-WebView-2 and Accidents or relational graphs in Hughes. ParaMiner reaches a 4 times speedup but does not benefit further from additional cores.

Result interpretation

In order to provide an explanation of the behavior of ParaMiner on *Server*, we first monitor the core activity during an execution. The two charts in Figure 4.14 show each core activity during two execution of ParaMiner: one with BMS-WebView-2 (a) and one with Accidents (b). When a core switches from the inactive state to active state its activity curve moves upward and conversely. The program terminates when all the cores are shown inactive.

First, what we can see in the Figure 4.14 is that load imbalance is not an issue. Load imbalance occurs when some cores remains inactive while others are performing their last task. Figures 4.14 (a) and (b) clearly show that at the end of the execution, all the cores become inactive roughly at the same time.

However Figure 4.14 (b) reveals that most cores are inactive at the beginning of the execution of ParaMiner. Further investigation shows that this is the time required to load and pre-process the input dataset. In ParaMiner, this operation is done sequentially. It means that during this time, only one core is contributing to the overall progress of the execution. When the dataset is as large as Accidents, the time required to perform this step may be a significant fraction of the total time and this can impact the speedup. In this particular experiment the fraction of sequential time is about $\frac{1}{5}$ of the total time

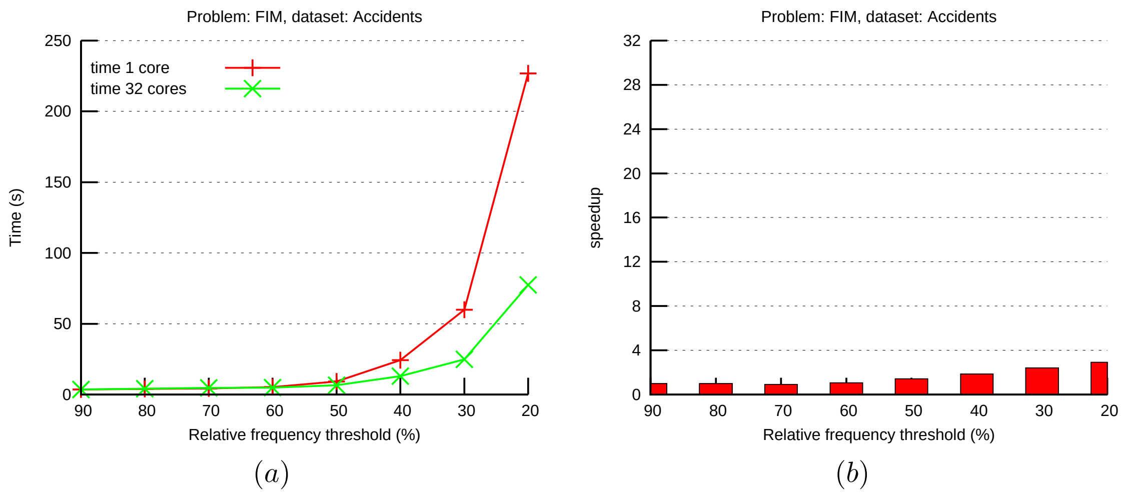


Figure 4.10: Runtimes on *Server*: Execution times for the FIM problem on Accidents (a) and speedups (b).

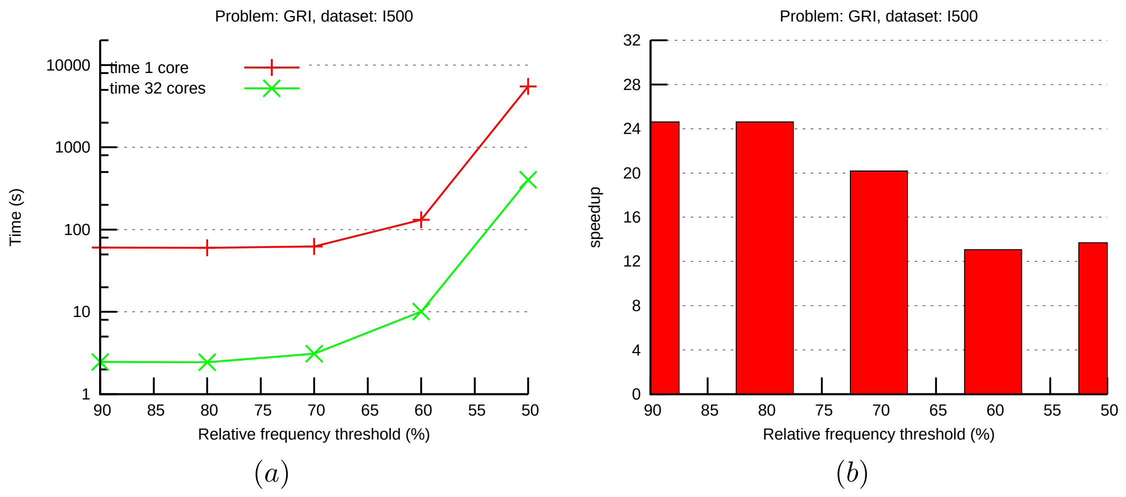


Figure 4.11: Runtimes on *Server*: Execution times for the GRI problem on I500 (a) and speedups (b).

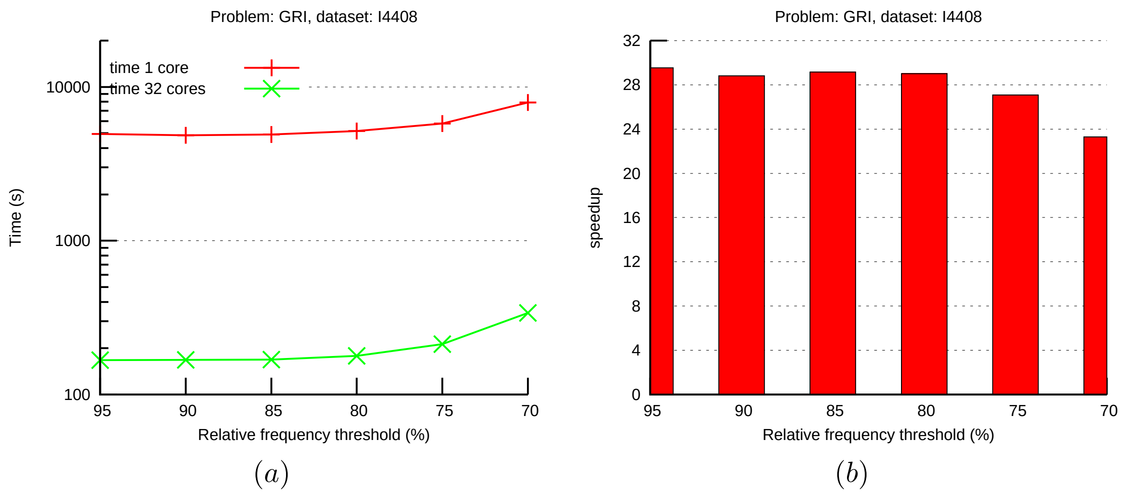


Figure 4.12: Runtimes on *Server*: Execution times for the GRI problem on 4408 (a) and speedups (b).

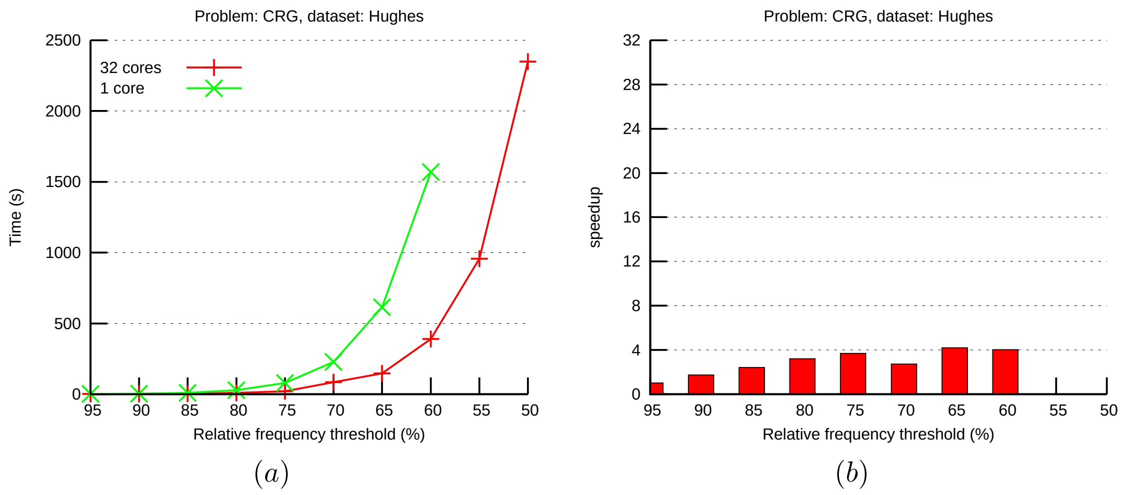


Figure 4.13: Runtimes on *Server*: Execution times for the CRG problem on Hughes (a) and speedups (b).

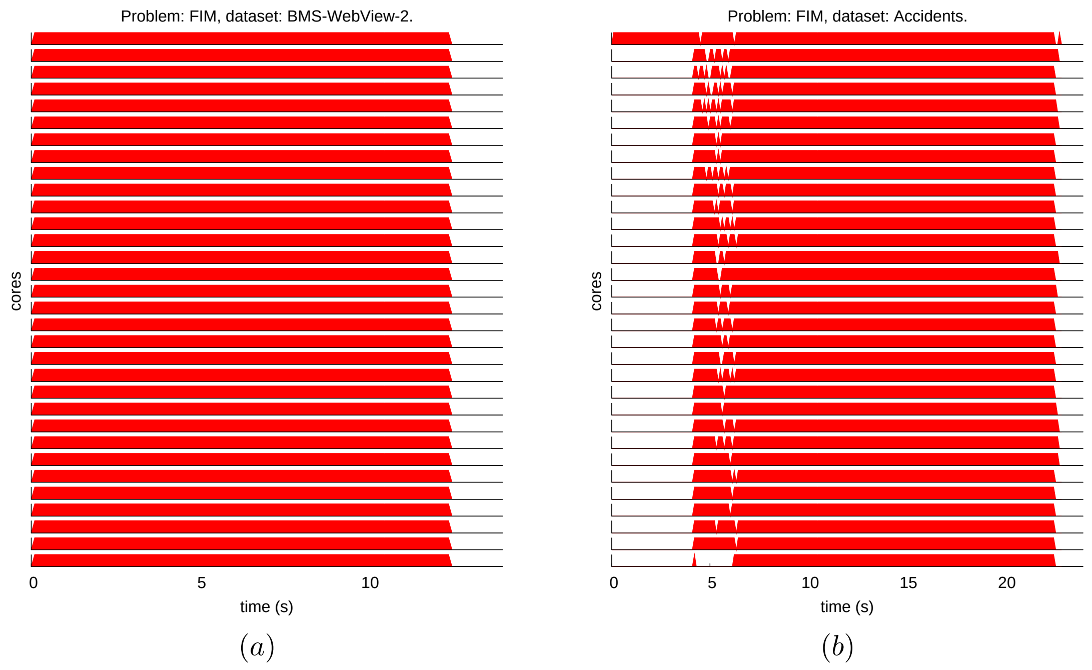


Figure 4.14: Core activity for BMS-WebView-2 mined with a frequency threshold of 0.08% (*a*) and Accidents mined with a frequency threshold of 40%: (*b*).

(4.2s/22.9s). The Aldahm's law state that the maximal speedup is bounded by the inverse of the fraction of sequential time ($max\_speedup = \frac{1}{P}$ where P is the portion of sequential time). Therefore the theoretical speedup is five for this execution.

According to Figure 4.10 ParaMiner's speedup on Accidents with a frequency threshold of 40% is above 3. In addition, ParaMiner on BMS-WebView-2 does not exhibit the same behavior and still does not reach the maximal speedup of 32. Hence this is not the only explanation for ParaMiner's behavior shown in speedup curves.

We recalled in the previous chapter that poor cache locality is another common issue in a parallel algorithm. Caches are useful to provide fast memory accesses to data. Indeed accesses to data that are already in the cache can be performed very quickly. Conversely when data are not in the cache, it must be fetched from the main memory and it implies a delay. Each memory access that has to fetch data from the main memory is called a *cache miss*. We say that an algorithm suffers from poor cache locality when cache misses are frequent and induce an important time penalty for the algorithm. In order to evaluate the cache locality in ParaMiner, we measure the number of memory accesses that are cache misses, and compare it to the total number of memory accesses. The results are shown in Figure 4.15.

The Figure 4.15 clearly shows that the number of cache misses increases when ParaMiner is executed with more cores. It start with 11% with a single core execution and reaches 45% percent when using 32 cores.

When the algorithm performs a lot of cache misses, the *memory bus* is frequently solicited. The memory bus is the hardware component used to transfer data from the memory to the cores. When the bus reaches its maximal bandwidth, it has to delay memory operations.

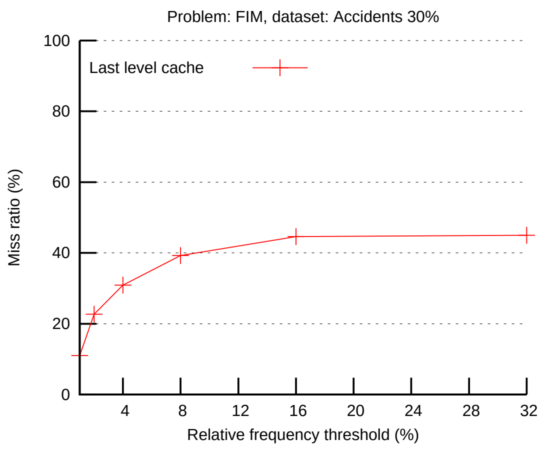


Figure 4.15: The percentage of cache misses memory operations in ParaMiner with 1 to 32 cores.

During this delay the cores are unable to execute any instruction, and the overall efficiency is impacted. The Figure 4.15 only reports the number of caches misses which is insufficient to detect these delays.

In Figure 4.16, we measure the amount of high-latency memory operations with a varying number of cores. Darker surfaces mean longer memory operations. This figure shows that as we increase the number of cores used to execute ParaMiner, memory operations are performed within more cycles. On this type of platform, an access to the main memory is performed in about 120 cycles. Therefore if a memory operation is performed in more than 120 cycles it means that it has been delayed.

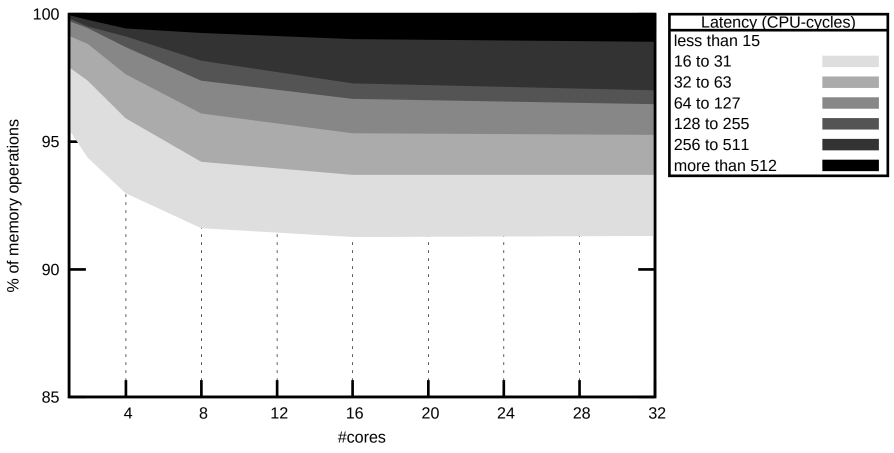


Figure 4.16: The increasing number of high latency memory operations.

If we focus on memory operations whose latency is above 128 cycles (three darker slices), we observe that the number of such operations is almost null when ParaMiner runs on a single core ($\sim 0.2\%$) but reaches 3.4% when ParaMiner is executed on 32 cores. This is symptomatic of a memory bus unable to sustain the memory accesses in time. This results are in line with the results of Ghosting in [GBP$^+$05], measured on a FP-Growth like frequent itemset mining algorithm.

This problem is known as the *memory wall* problem [WM95]: It is due to the fact that for twenty five years the computational throughput of processors has increased faster than the memory bus capabilities. Today, it is almost impossible to build a scalable parallel algorithm without putting work into the reduction of bus usage.

We have shown in this section that ParaMiner is particularly sensitive to the memory wall problem due to an extensive usage of memory operations. Similar results have been observed on other pattern mining algorithms in [TP09, GBP+05]. In the following section we propose a Melinda strategy to improve the cache locality and reduce the bus contention.

4.3.3 Melinda strategies to improve the cache locality

First, it important to mention that this work on Melinda strategies was conducted in collaboration with Serge Emteu in the context of a master internship [ETMT10].

It also important to mention that we designed and experimented new Melinda strategies on the PLCM algorithm and not on ParaMiner. PLCM is our parallel frequent itemset mining algorithm, which is also powered by Melinda. PLCM integrates a similar dataset reduction process although it is specific to frequent itemset mining. In addition, when ParaMiner is used to mine closed frequent itemset, both algorithms explore the search space according the same enumeration strategy.

We have shown in former experiments in [NTMU10] that PLCM also hit the *memory wall.* PLCM was preferred to ParaMiner to conduct these experiments because it features *bloc allocation.* Bloc allocation simply guarantees that large data structures such as the datasets are stored contiguously in the memory. It provides PLCM a more straightforward behavior in terms of memory accesses. This feature has not been integrated yet into ParaMiner. As a consequence, PLCM is more appropriate to experiment on memory accesses but the results are transposable to an implementation of ParaMiner using bloc allocation.

In this section we design and evaluate a Melinda strategy that takes advantage of the processor complex cache architecture in *Server.* We first present the cache architecture in *Server* then explain how we can exploit this knowledge to improve PLCM's behavior in terms of cache locality.

Cache architecture in *Server*

In order to reduce the number of accesses to the main memory most multi-core architectures comes with not just one, but a cascade of cache memory units. When data is requested by a core, the first cache level is queried, if the first level cache does not contain a copy of the data, the second level is queried and so on until the main memory. First levels units are fast but can only store a limited amount of data whereas last level are bigger but slower. A schematic representation of a processor of *Server* is presented in Figure 4.17.

Figure 4.17 shows that each processor on *Server* has three levels of cache: L1, L2 and L3. It also reveals that the last level of cache is shared among all the cores of the processor. It means that if a core fetches a particular data structure such as a dataset, this data structure is then available in this cache for all the other cores of the processor. Therefore

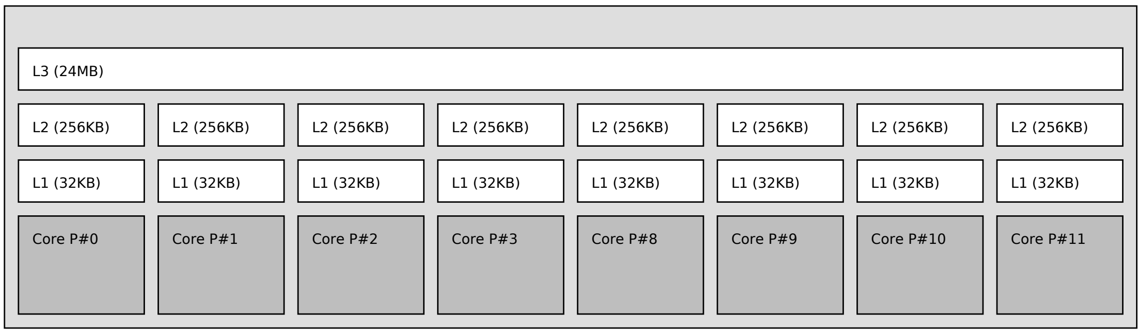


Figure 4.17: The caches architecture of one processor of *Server*. The whole computation platforms has four such processors.

if two tasks performs accesses to the same data structures in memory, they can benefit from being executed on two cores with the same cache. In our new strategy, we want to take advantage of this effect to reduce the amount of out-of-cache memory accesses. To do so we must identify what tasks perform accesses to the same data structures.

Data sharing in PLCM and ParaMiner

Data sharing in ParaMiner and PLCM occurs due to the internal representation of the reduced datasets. In order to limit the memory usage, the reduced dataset of a closed pattern shares common parts with its parent reduced dataset. As a consequence, parents and sibling tuples often access the same data structures in memory. Conversely tuples generated on different branches perform accesses to different data structure.

The idea behind the strategies proposed in this section is to group together on the same processor the tasks that perform accesses to the same data structures. The benefit is two fold: first the data structures only need to be loaded once into the cache; and second, if all the tasks perform access to the same data structures more cache memory is available to store these data structures. This is illustrated in Figure 4.18.

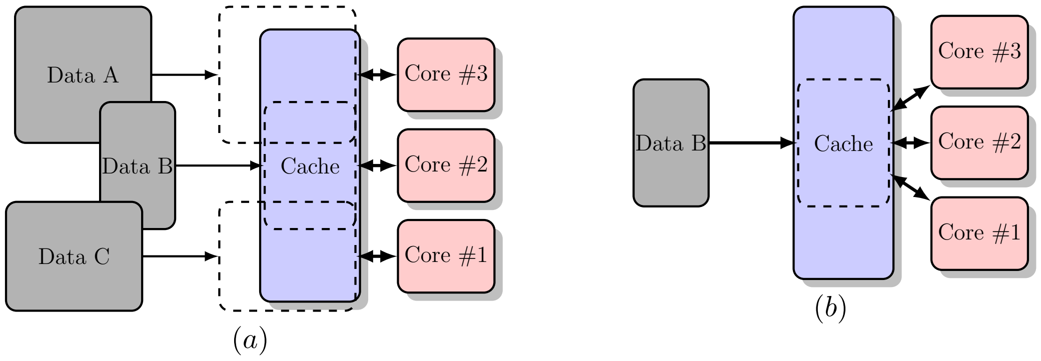


Figure 4.18: Bad strategy vs good strategy: In (*a*), all three cores access different data structures, and shared cache is overfull. In (*b*) the cache is correctly used.

In order to achieve the result in Figure 4.18 (*b*), we propose a Melinda strategy and measure the speedup achieved on two dense datasets: Accidents and Connect.

Our strategy S_{arch} consists in distributing the tuples generated in the same branch of the enumeration tree to the cores of the processors that share a cache. In order to avoid the computation of several branches simultaneously on the same processor, cores may have to wait until the branch is completed before exploring another branch.

The *distribute*() function of the strategy S_{arch} is shown in Algorithm 18. This function separates the depth 1 tuples from other tuples. The depth 1 tuples are the roots of new branches and are systematically stored in the first internal (Line 3).

Any tuple with a depth higher than 1 is by construction an internal node of a branch that must be taken by a core of the same processor. By returning the processor identifier in Line 8, we guarantee that tuples generated by different processors are not mixed together within the same internal.

The rest of the work is performed by the *reduce*() function presented in Algorithm 19. In this function, we essentially pull tuples from the internal *processor_id* until the branch is complete. The branch is complete when no more tuples are available (Line 3), and no more processor's core is active (Line 6). When the branch is completed, a root tuple is pulled from the internal 0 (Line 7), and the same process starts again until no more tuple are available in internal 0.

In order to prevent two branches from being processed simultaneously on the same processor, cores have to wait in Line 10 until the branch is completely processed. Cores wait until they are signaled by another core. Signals are sent when a new tuple is pushed into the internal (Algorithm 19 Line 6-7) or when a new tuple is retrieved from the first internal to explore a new branch (Line 6 in Algorithm 19).

Algorithm 18 MELINDA strategy S_{arch} (*distribute*).

```
1: function Distribute(tuple:[$P, \mathcal{D}, depth$])
2:     if $depth = 1$ then
3:         return 0 //Each depth 1 tuple is the first node of a new branch. We store these
       tuples in the first internal (internal 0).
4:     else
5:         $pending\_tuples[processor\_id] \leftarrow pending\_tuples[processor\_id] + 1$
6:         signal //Signal waiting cores that a new thread arrived.
7:         return $processor\_id$
8:     end if
9: end function
```

The Figure 4.19 compares the default strategy with this new strategy. The curve $S_{default}$ represents the speedup of PLCM with the default *fifo* strategy whereas S_{arch} is the speedup with our new strategy. This figures show that our S_{arch} strategy performs 24% better on Accidents and 28% better on Connect when running PARAMINER. This is an important benefit obtained without any modifications to the source code of PLCM and also without handling too low level concepts.

In Figure 4.19 (*b*) we also observe that with this new strategy, the performance are optimal with up to two active cores per processor (i.e. a total of 8 active cores). This suggests that over two cores per processor, the bus is unable to sustain the demand and has again to

Algorithm 19 MELINDA strategy S_{arch} (*retrieve*).

```
1:  $active\_cores[processor\_id] \leftarrow 0$
2:  function Retrieve()
3:      while $pending\_tuples[processor\_id] = 0$ do
4:          $active\_cores[processor\_id] \leftarrow active\_cores[processor\_id] - 1$
5:          if $active\_cores[processor\_id] = 0$ then
6:              $active\_cores[processor\_id] \leftarrow active\_cores[processor\_id] + 1$
7:              signal
8:              return 0 //Explore a new branch
9:          else
10:             wait //Wait until a new tuples arrives.
11:         end if
12:     end while
13:     $active\_cores[processor\_id] \leftarrow active\_cores[processor\_id] + 1$
14:     //Pulls a tuple of the branch and decrease the tuple count
15:     $pending\_tuples[processor\_id] \leftarrow pending\_tuples[processor\_id] - 1$
16: end function
```

delay the memory operations. However this strategy is an important step towards more cache locality.

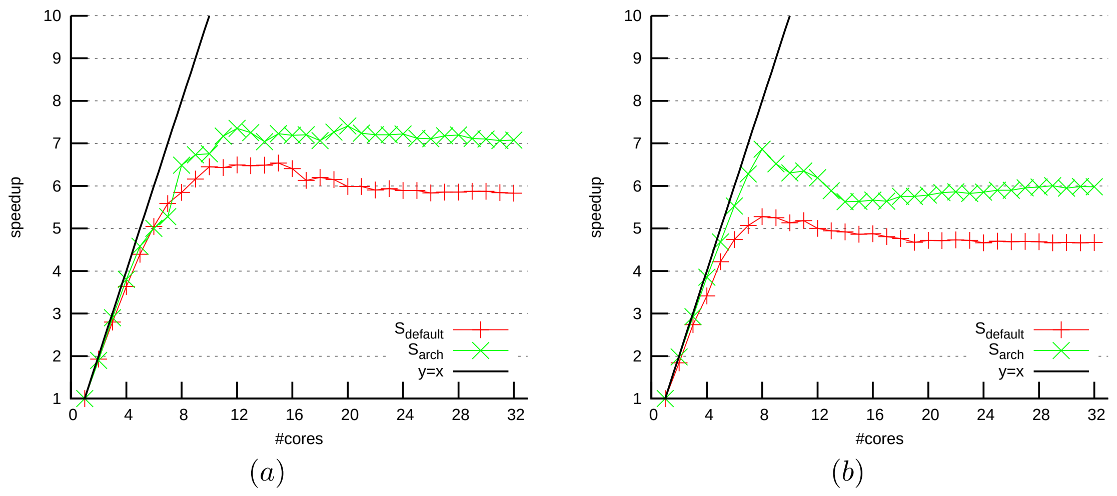
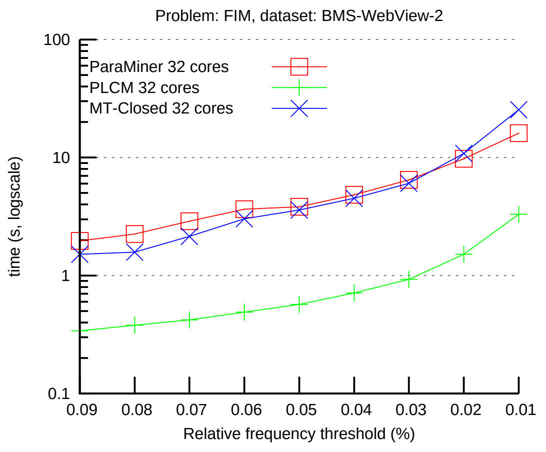


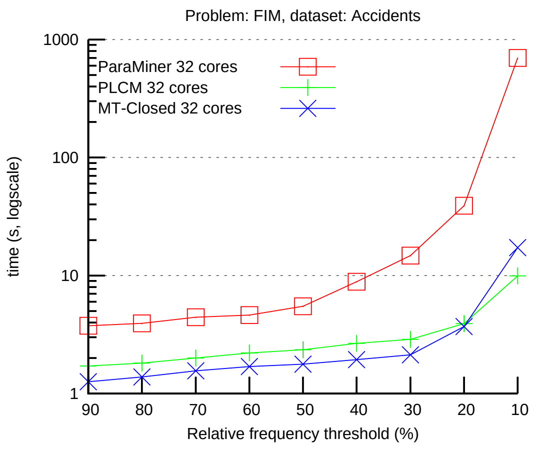
Figure 4.19: Speedup on Accidents (a) and Connect (b) with different MELINDA's strategies.

4.4 Comparative experiments of PARAMINER with ad-hoc algorithms

In this section we compare the performances of PARAMINER for mining frequent itemset and gradual itemset with state of the art ad-hoc algorithms. We were unable to run comparative experiments for the relational graph mining problem due to lack of implementation available.



(*a*)



(*b*)

Figure 4.20: Comparatives experiments for FIM: ParaMiner vs PLCM vs MT-Closed.

The algorithms that we compare to are the original implementations downloaded from author's website or directly retrieved from them. We recall that the algorithms were all compiled with the same compiler (`gcc`) and compiler settings. The execution times include the complete mining process. In ParaMiner, this includes reading and encoding the input dataset, computing the patterns and writing them into a file.

4.4.1 ParaMiner vs FIM algorithms

During the FIMI workshop [Goe03], the efficiency of dozens of FIM algorithms has been compared. LCM received the fastest algorithm award. Since PLCM is our parallel implementation of LCM, (whose efficiency was demonstrated in [NTMU10]), we chose to compare ParaMiner with PLCM. The DCI-Closed algorithm by Lucchese et al. [LOP04] is another very competitive algorithm. In fact, DCI-Closed is particularly efficient on dense datasets. In addition, DCI-Closed author's have built an efficient parallel implementation of DCI-Closed called MT-Closed [LOP07]. Thus we also chose to compare ParaMiner with MT-Closed.

The experiments were conducted on the two datasets BMS-WebView-2 (sparse) and Accidents (dense). The timing results are shown in Figure 4.20 (*a*) for BMS-WebView-2 and in Figure 4.20 (*b*) for Accidents.

On the sparse dataset, ParaMiner and MT-Closed exhibit very close performances. On this dataset, they are also both one order of magnitude slower than PLCM. On the dense dataset, ParaMiner is one order of magnitude slower than both PLCM and MT-Closed. This is a good result considering that PLCM and MT-Closed can exploit the problem specificity by ignoring infrequent elements for example. This reduces the search space and allows more aggressive dataset reduction. Although ParaMiner cannot make these assumptions for the sake of genericity, it still exhibits good execution times. It is worth noticing that for the same datasets, any implementation of Apriori is two to three orders of magnitude slower than ParaMiner.

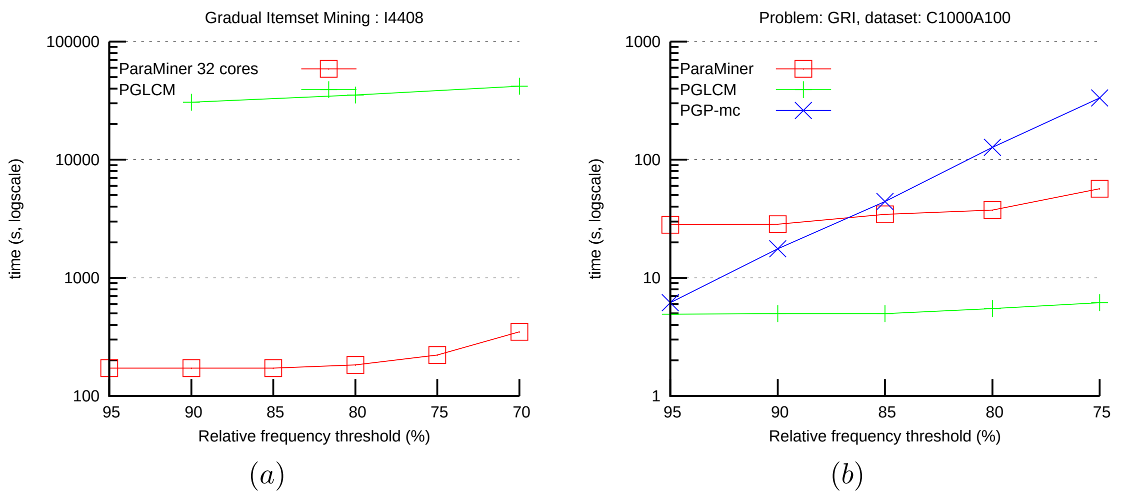


Figure 4.21: Comparatives experiments for GRI: ParaMiner vs PGLCM vs PGP-MC.

4.4.2 ParaMiner vs GRI algorithms

We now compare ParaMiner with the state-of-the-art parallel algorithms designed to solve the GRI problem. We run comparative experiments with two algorithms: PGP-MC by Laurent et al. [LNST10] and PGLCM [DLT10] by Do et al. The comparison with PGP-MC is not fair because it only mines frequent gradual itemsets (not necessarily closed). However it is shown here to evaluate the benefit of *closed* gradual itemset mining. PGLCM is the fastest closed gradual itemset mining algorithm available nowadays, its efficiency has been demonstrated in [DLT10].

We first experiment on the real gene expression database I4408 described in Table 4.3. The results are shown in Figure 4.21 (a).

On this database, PGP-MC cannot complete due to memory exhaustion. With a frequency threshold of 70%, PGLCM takes more than 11 hours to complete the computations whereas ParaMiner outputs the same result within less than six minutes. This is mostly due to the lack of dataset reduction technique in PGLCM. In [DLT10], the authors leave the problem of designing a dataset reduction for the GRI problem as open problem. Since any problem formalized into our framework can directly benefit from ParaMiner's dataset reduction, this problem is now solved.

Due to lack of dataset reduction PGLCM's efficiency is particularly sensitive to the high number of attributes in the I4408 database (4408). However this database has few number of records. In order to provide a fair comparison with PGLCM, we also run experiments on C1000A100. C1000A100 is a synthetic database with 1000 records and only 100 attributes. It was generated with a modified version of IBM Synthetic Data Generator for Association and Sequential Patterns also used in [DLT10] and [LNST10].

The results are shown in Figure 4.21 (b). This figure shows that although ParaMiner is one order of magnitude slower than PGLCM, this database can be tackled in less than a minute by both algorithms. PGP-MC which is competitive with high frequency thresholds values, is quickly outperformed when this frequency threshold is reduced. This is due to non-closed pattern mining in PGP-MC. Despite being a generic algorithm, ParaMiner is the best choice to solve the GRI problem.

4.5 Conclusion on the experimental evaluation of ParaMiner

In this chapter we have experimentally demonstrated the practical efficiency of dataset reduction and we have also shown the important benefit that can be achieved by exploiting efficiently the computational power offered by multi-core architectures.

This ability to exploit multi-core architectures combined together with optimizations such as dataset reduction make ParaMiner a competitive algorithm for many important pattern mining problem. As a consequence, for some important problems such as the GRI problem, ParaMiner is able to achieve much better performances than state of the art ad hoc algorithms. This demonstrates the interest of the generic approach compared to the traditional problem specific approach. It also validates the framework presented in Chapter 2 and the design choices made in Chapter 3.

For some problems, ParaMiner does not currently scale up to the large number of cores available in large multi-core platforms. We have identified with experiments that those problems are due to the memory wall problem and are provoked by poor cache locality. This is a challenging problem that has been observed and addressed by several research papers in parallel pattern mining such as [TP09], [GBP$^+$05] and [NTMU10]. We will present these works in Section 5. It is important to note that any improvement in ParaMiner implementation will instantaneously benefit to the broad range of problems that can be formalized into our framework.

Chapter 5

Related work

Contents

To the best of our knowledge, ParaMiner it is the first work to address simultaneously the problems of genericity and of parallelism. Separately, these two problems have attracted a lot of attention from the pattern mining community. We present in this chapter the state of the art in generic pattern mining and the most recent works in parallel pattern mining. For each family of approaches, we will compare it with ParaMiner in order to highlight the specificities of our approach w.r.t. the state of the art and the use cases where ParaMiner should be preferred over another approach.

5.1 Generic pattern mining

The notion of genericity in pattern mining have been tackled in two major ways. One of them, called *constraint-based* pattern mining, is rooted in the database world. It is based on the necessity to make as precise as possible queries on the dataset, in order to produce few patterns of high significance. These queries are expressed as *constraints* that the candidate patterns have to satisfy in order to be considered as a result. Constraint-based approaches are mostly adapted to transactional databases, possibly with numerical values. We present them in sub-section 5.1.1.

The other approach has an algorithmic root. It is built on the observation that most pattern mining algorithms share a similar structure, and tries to unify the algorithms in order to reduce the work of practitioners having a new pattern mining problem to handle. We call this approach the *toolbox approach.* Toolbox approaches are mostly adapted to databases with structured data: sequences, trees or graphs. We present them briefly in Section 5.1.2.

5.1.1 Constraint-based approaches

Constraint-based pattern discovery is a solution that was introduced as early as 1996 [SVA97] to mine more efficiently a reduced number of interesting patterns. The idea is to allow pattern mining system users to characterize the patterns of interest with various constraints. This guarantees that the practitioner will have less patterns, which correspond better to what is searched in the data. From the mining algorithm point of view, a large body of work has been dedicated to push as deeply as possible the constraints inside the mining algorithm, allowing to prune large portions of the search space and to dramatically reduce mining times (see [BL07] for a survey).

A constraint is a predicate $C : 2^E \times \mathcal{D}_E \times \mathcal{A} \mapsto \{true, false\}$. It is very similar to our *Select* predicate: the only syntactic difference is the presence of the set of attributes $\mathcal{A}$ in the constraints, however nothing prevents from using attribute values in our *Select* predicate. Most approaches presented in the literature consider that a candidate pattern is a meaningful pattern (following terminology of Chapter 2) if it is *frequent* and if it satisfies a user-defined constraint C (other than frequency). This acknowledges the fact that in practice frequency is needed, both to ensure the discovery of patterns having enough significance in the dataset and to guarantee pruning of the search space.

From the practitioner point of view, the constraints can either be written in any formalism, like in our approach, or a formalism can be imposed: [Sou06, SC05] give a rich set of primitives for expressing the constraints, and recently [GNR11] proposed to directly use the formalism of constraint programming.

For all these formalisms, the most important point is how deeply the constraint can be pushed into the algorithm in order to prune the search space and improve the mining time. This depends on properties that are verified by these constraints. Several classes of properties verified by the constraints have been studied in the literature: *anti-monotone* constraints [MT97], *monotone* constraints [MT97], *succinct* constraints [NLHP98], *convertible* constraints [PH00, PHL01], *loose anti-monotone* constraints [BL07] and more general constraints, such as the *area* constraint [SC05, Sou06].

Constraints and accessibility

The properties on constraints given above help to define how the search space of patterns verifying the constraint is organized. They are thus very similar to the accessibility properties presented in Chapter 2. As a side contribution of this thesis, we present the most studied constraint classes and show the accessibility property that they verify when possible.

For sake of simplification, in the following we simply write $C(X)$ to denote the value of constraint C applied to the candidate pattern X.

We first recall the definitions of all major constraint classes, with an example for each class.

Definition 5.1 (Anti-monotone constraint)
A constraint C is anti-monotone if for every pattern X and every candidate pattern Y with $Y \subseteq X$, $C(X)$ implies $C(Y)$.

Example: The well known frequency constraint, first defined in Apriori [AS94] is an anti-monotone constraint.

Definition 5.2 (Monotone constraint)
A constraint C is monotone if for every pattern X and every candidate pattern Y with $Y \supseteq X$, $C(X)$ implies $C(Y)$.

Example: Consider that each element is associated a *price* attribute. The constraint $C_M(X) \equiv sum(X.price) \geq 500$ is monotone for prices taking values in $\mathbb{R}^+$.

For the succinct constraints, we use the a reformulation by [Sou06], which is more understandable than the original formulation provided in [NLHP98].

Definition 5.3 (Succinct constraint)
For a ground set E, a constraint C is succinct if the exists $E_1 \subseteq E$,...,$E_n \subseteq E$ such that the set of patterns satisfying C can be expressed as unions and differences of the power sets $2^{E_1}, ..., 2^{E_n}$.

Example: Consider a set of elements E with a unique *type* attribute having categorical values such as $toy, food, tool \ldots$ and so on. Let X be a candidate pattern defined over E. Now consider the constraint $C_S(X) \equiv X.type \supseteq \{food, toy\}$, where $X.type$ denotes the union of all types of the elements in X. Informally, a candidate pattern satisfies C_S if it contains at least one element of type $food$ and one element of type *toy*. [BL07] have shown that this constraint is succinct because the set of patterns satisfying it can be defined as: $2^E - 2^{E_2} - 2^{E_3} - 2^{E_4} - 2^{E_2 \cup E_4} - 2^{E_3 \cup E_4}$, with:

- $E_2 = \{e \mid e \in E \text{ and } e.type = food\}$
- $E_3 = \{e \mid e \in E \text{ and } e.type = toy\}$
- $E_4 = \{e \mid e \in E \text{ and } e.type \neq toy \text{ and } e.type \neq food\}$

Definition 5.4 (Convertible constraint)
A constraint C is convertible anti-monotone if there exists an order R on the elements of the ground set E such that for all X satisfying C, it holds that any prefix of X also satisfies C.

A constraint C is convertible monotone if there exists an order R on the elements of the ground set E such that for all X not satisfying C, any prefix of X also does not satisfies C.

Example: Suppose that a unique attribute *price* with numerical values is defined for all elements of the ground set E. Let R be a value-descending order. Then $C_{CAM}(X) \equiv average(X.price) \geq v$ for a fixed value v is a convertible anti-monotone constraint [BL07].

With the same order, $C_{CM}(X) \equiv average(X.price) \leq v$ for a fixed value v is a convertible monotone constraint.

Definition 5.5 (Loose anti-monotone constraint)
A constraint C is loose anti-monotone if for every candidate pattern X with $|X| > 2$, $C(X)$ implies $\exists e \in X | C(X \setminus \{e\})$.

Example: The constraint $C_{LAM}(X) \equiv variance(X.A) \leq v$ for an attribute A and a fixed value v is loose anti-monotone [BL07].

Definition 5.6 (Area constraint)
Given a candidate pattern X, the area constraint for X is $C_A(X) \equiv support(X) \times length(X) \geq v$ for a fixed value v, where $length(X)$ is the number of elements in X.

This last constraint is in none of the classes defined above.

We now determine the accessibility property verified by each constraint.

Property 5.1
The set system associated to an anti-monotone constraint is independent.

Proof: We already have stated in Chapter 2 that the FIM problem was independent thanks to the anti-monotony of frequency. The proof of [BHPW10] can trivially be extended to all anti-monotone constraints. □

Property 5.2
The set system associated to constraint which is either:

- *monotone*
- *succinct*
- *convertible*
- *loose anti-monotone*
- *area*

is not accessible.

Proof: For each of the above constraint classes, consider the example constraints we have given for before. For the empty set, the constraints $C_M(\emptyset)$, $C_{CAM}(\emptyset)$, $C_{CM}(\emptyset)$, $C_{LAM}(\emptyset)$ are not defined. The constraints $C_S(\emptyset)$ and $C_A(\emptyset)$ are defined but are not satisfied. So in all cases as $\emptyset$ is not in the set system defined by the constraint, that set system cannot be accessible and the examples provided are counter-example for the accessibility of their class of constraints. □

The point that prevented to conclude to accessibility was in all cases the fact that $\emptyset$ could not satisfy the constraints due to their specific formulation. In practice, many constraints, especially aggregate constraints such as *sum*, *average*, *variance*, can only be satisfied by candidate patterns having a certain size. This means that at the beginning of the enumeration the technique to *"augment $\emptyset$ by one element"* cannot work, the enumeration has to make "jumps" to bigger candidate patterns. However once a pattern of minimal size is found, for some constraints the above enumeration technique can be used.

We thus propose to amend the notion of accessibility to such cases. This is a preliminary step towards the convergence of works of constrained pattern enumeration and works on accessible / strong accessible set system enumeration.

We first need to characterize the "minimal" patterns satisfying a constraint. This can be done with the notion of *lower border* introduced in 2007 for itemsets in [SY07]. We generalize below their definition for any set system.

Definition 5.7 (Lower border of a set system)
Let $(E, \mathcal{F})$ be a set system. The lower border of $\mathcal{F}$, denoted $BD^-_{\mathcal{F}}$, is the set of patterns such that:

i $BD^-_{\mathcal{F}} \subseteq \mathcal{F}$

ii For any two patterns $X, Y \in BD^-_{\mathcal{F}}$, $X \nsubseteq Y$ and $Y \nsubseteq X$ ($BD^-_{\mathcal{F}}$ is called an antichain)

iii For any pattern $X \in \mathcal{F}$, there exists at least one pattern $Y \in BD^-_{\mathcal{F}}$ such that $Y \subseteq X$.

Property 5.3
The lower border of a set system is unique.

Proof: Let $(E, \mathcal{F})$ be a set system, and let $BD1^-_{\mathcal{F}}$ and $BD2^-_{\mathcal{F}}$ be two lower borders for $(E, \mathcal{F})$. We will show that $BD1^-_{\mathcal{F}} = BD2^-_{\mathcal{F}}$.

($BD2^-_{\mathcal{F}} \subseteq BD1^-_{\mathcal{F}}$) Consider $X_2 \in BD2^-_{\mathcal{F}}$. Either X_2 is also in $BD1^-_{\mathcal{F}}$ or there exists $X_1 \in BD1^-_{\mathcal{F}}$ such that $X_1 \subset X_2$ (according to iii. in Definition 5.7). For this X_1, we have either:

- $X_1 \in BD2^-_{\mathcal{F}}$: as $X_1 \subset X_2$, this contradicts the antichain property of $BD2^-_{\mathcal{F}}$.
- or there exists $X'_2 \in BD2^-_{\mathcal{F}}$ such that $X'_2 \subset X_1$ (property iii. of Definition 5.7). Then we have $X'_2 \subset X_1 \subset X_2$, this contradicts the antichain property of $BD2^-_{\mathcal{F}}$.

We conclude that $X_2 \in BD1^-_{\mathcal{F}}$, hence $BD2^-_{\mathcal{F}} \subseteq BD1^-_{\mathcal{F}}$.

($BD1^-_{\mathcal{F}} \subseteq BD2^-_{\mathcal{F}}$) This case is symmetrical to the preceding case.

Hence $BD2^-_{\mathcal{F}} = BD1^-_{\mathcal{F}}$, the lower border is unique. □

We can now propose a weaker property of accessibility based on the lower border.

Definition 5.8 (Partial accessibility)
Let $(E, \mathcal{F})$ be a set system. $(E, \mathcal{F})$ is partially accessible if for every $X \in \mathcal{F} \setminus BD^-_{\mathcal{F}}$ there exists some $e \in X$ such that $X \setminus \{e\} \in \mathcal{F}$.

Definition 5.9 (Partial strong accessibility)
Let $(E, \mathcal{F})$ be a set system. $(E, \mathcal{F})$ is partially strongly accessible if it is partially accessible and if for every $X \subset Y$ with $X \in \mathcal{F}$ and $Y \in \mathcal{F} \setminus BD^-_{\mathcal{F}}$, there exists some $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{F}$.

As preliminary results, we show below the partial accessibility properties verified by some important constraint classes.

Property 5.4
Let C be a constraint and $(E, \mathcal{F})$ be its associated set system.

- *If C is monotone, then $(E, \mathcal{F})$ is partially strongly accessible.*
- *If C is convertible anti-monotone or loose anti-monotone, then $(E, \mathcal{F})$ is partially accessible.*

Proof: **Monotone constraints:** Let C be a monotone constraint with $(E, \mathcal{F})$ its associated set system. First let us prove the partial accessibility. Consider $X \in \mathcal{F} \setminus BD^-_{\mathcal{F}}$. By definition of the lower border, there exists at least one $Y \in BD^-_{\mathcal{F}}$ such that $Y \subset X$. Consider any element $e \in X \setminus Y$, and consider $Y' = Y \cup (X \setminus Y \setminus \{e\})$. We have $Y' = X \setminus \{e\}$ and by definition of monotony, as $C(Y)$ holds so does $C(Y')$. Hence the partial accessibility.

Let us now prove the partial strong accessibility. Let $X \subset Y$ be two patterns satisfying the constraint ($C(X) = C(Y) = true$). We have that $X \in \mathcal{F}$ and by the antichain property of lower border, $Y \in \mathcal{F} \setminus BD^-_{\mathcal{F}}$. Let us consider any $e \in Y \setminus X$. We have $X \subset X \cup \{e\}$ so by monotony property $C(X \cup \{e\}) = true$. Hence the set system is partially strongly accessible.

Convertible anti-monotone constraints: Let C be a convertible anti-monotone constraint with the order R and let $(E, \mathcal{F})$ be its associated set system. Let X be any pattern satisfying C, with $|X| = k > 1$ and $X \in \mathcal{F} \setminus BD^-_{\mathcal{F}}$. By definition of the lower border, X admits at least one subset which satisfies the constraint. Let X' be the $k-1$-prefix of X: by definition of convertible anti-monotone constraints $C(X') = true$, hence $e = X \setminus X'$ is such that $C(X \setminus \{e\}) = true$. Hence the partial accessibility.

Loose anti-monotone constraints: Let C be a loose anti-monotone constraint and let $(E, \mathcal{F})$ be its associated set system. Let $X \in \mathcal{F} \setminus BD^-_{\mathcal{F}}$ with $|X| > 2$. By definition of the lower border, X admits at least one subset which satisfies the constraint. And by definition of loose anti-monotony, there exists $e \in X$ such that $X \setminus \{e\} \in \mathcal{F}$. Hence the partial accessibility. □

The property stated above shows that the efficient pattern enumeration techniques presented in Chapter 2 can be exploited for a large portion of the search space of major constraints. This is important as these enumeration strategies have strong complexity guaranties, which is not the case of most other approaches in the literature of constraint-based pattern mining. One interesting perspective, developed in Chapter6, is to develop efficient hybrid algorithms that first compute the patterns of the lower border for the set system of a constraint and then switch to enumeration strategies adapted to accessible or strongly accessible set systems.

Constraints and closure

We have presented at the beginning of Chapter 2 the notion of *closed patterns*, which is essential to speedup the discovery of patterns while reducing the size of the output. Although there have been many works for mining closed patterns for itemsets, sequences, trees or graphs, there are comparatively few works on closed constrained pattern mining.

The difficulty of integrating constrained pattern mining and closed pattern mining is well illustrated in the following example coming from the 2004 paper of Bonchi and Lucchese [BL04], which is focused on itemsets. Consider a constraint for an itemset X:

$$C(X) \equiv isFrequent(X) \wedge isClosed(X) \wedge sum(X.price) \leq 22$$

Here the closure is considered as just another constraint, simplifying the writing of constraints. However, Bonchi and Lucchese point out that this constraint is ambiguous and can lead to two possible interpretations:

- mine all frequent closed itemsets having the additional property of having sum of prices less than 22 ;
- or mine all frequent itemsets having sum of prices less than 22 and which have the additional property of being closed w.r.t. the two other constraints.

As the closure is not a property that an itemset verifies on its own but a property of an itemset w.r.t. a set of itemsets, its the second interpretation which is correct. They formally characterize the associated problem statement, showing that their solution is the only one to provide a lossless representation of constrained frequent patterns for monotone and anti-monotone constraints. They also propose a FP-Growth algorithm based on Closet [PHM00], and show that for selective constraints their approach outperforms the original algorithm.

More recently, Soulet [Sou06] has proposed an approach based on intervals of patterns w.r.t the constraint during the search. Their approach is efficient at determining lower and upper bounds of patterns verifying a constraint. For instance if AB, ABC, ABD and $ABCD$ satisfy the constraint, only the interval $[AB, ABCD]$ will be returned. These intervals allow to both reduce the output and perform pruning during search space exploration. They have defined a closure for these intervals, allowing to improve their original algorithm performance. Their final algorithm, Music-dfs, outputs a set of closed intervals satisfying a constraint. Such a closure notion is interesting, as even if it differs from the traditional "closed pattern" notion, it can be applied to many different constraints such as the *area* constraint, and provides interesting pruning results.

Last, Guns et al. [GNR11] integrate the notion of closure directly into their constraint programming framework. Due to the way constraint programming approaches explore a search space, they state that their approach has a search space exploration very similar to that of LCM [UKA04].

Discussion: The integration of closure and constraints have been tackled in several different ways. Although these approaches show real performance improvements, there still lacks an unifying framework that would allow to fully benefit from closure with as many constraint classes as possible.

Dataset reduction and constraints

Dataset reduction is a crucial optimization for the performance of pattern mining algorithms. It has been shown in the case of ParaMiner by the experiments conducted in Chapter 4.

With the additional pruning that they allow on the search space, constraints can also benefit from dataset reduction. Bonchi and Lucchese detail in [BL07] for anti-monotone, monotone and loose anti-monotone constraints the reductions that can be performed. They explicitly provide the properties that must be checked on transactions to be pruned. Their experiments show up to three orders of magnitude of reduction in the datasets.

This approach is interesting, but comes with specific algorithms for different classes of constraints, and cannot be applied to all classes of constraints.

In [GNR11], Guns et al. advocate that their constraint programming based algorithms perform an operation very similar to dataset reduction with the concept of *state* of the search. However the data structures associated to this state currently have some overheads that are necessary for generic constraint programming, but that impairs performances for pattern mining.

Performances

Most papers about constrained pattern mining are interested in different classes of constraints, which make comparative experiments difficult to conduct. Hence these patterns first define a baseline by comparing their approach on the simple problem of mining frequent itemsets or closed frequent itemsets. In this case, the performances are similar to the frequent itemset mining that has been used as a basis to build the constraint mining algorithm.

We give below a quick positioning of ParaMiner w.r.t. the approaches that can extract closed frequent itemsets:

- CCIMiner [BL04] has similar performances as Closet for mining closed frequent itemsets. Closet has been show to be at least one order of magnitude slower that DCI-Closed, which is the sequential version of the parallel algorithm MT-Closed which ParaMiner has comparable performances for closed frequent itemset mining.
- The constraint-programming approach in [GNR11] reports performances on closed frequent itemset mining that are between one and two order of magnitude slower than LCM. This performance difference is similar to the performance difference between ParaMiner and PLCM which is a parallel algorithm based on LCM. However PLCM and ParaMiner can benefit from parallelism, thus having faster execution times than LCM and the constraint-programming approach in [GNR11] when using several cores.

Discussion

Constraint-based pattern mining research has proposed many interesting solutions to the problem of discovering a reduced number of high interest patterns. One key algorithmic point to efficiency is to exploit the properties of constraints in order to devise a good enumeration strategy. We have proposed above a new notion of *partial (strong) accessibility*, that could allow to bridge the gap between set-system based generic algorithms such as ParaMiner and the current research on pattern mining based on constraints.

In practice, picking the right algorithm depends on what constraints are significant to the application. Then another problem is the tradeoff between expressivity, guidance and performances. For instance, Soulet [Sou06] proposes a rich set of primitives for writing many different constraints. It is thus easy for a practitioner to use it for writing customized constraints with just basic mathematical knowledge. However, if the constraints are not very selective (such as frequency) the performances will not necessarily be optimal: for instance Music-dfs performances for mining frequent itemsets are between Apriori and Eclat. A more constrained problem is thus recommended in this case, especially with constraints such as the *area* constraint which are not handled in most other approaches.

The constraint programming approach of Guns and al. [GNR11] is based on the traditional languages of constraint programming such as Essence. They will have a great appeal for constraint programming specialists and to certain extent, to people with a strong mathematical background. However they might be more difficult to grasp for practitioners from other fields.

With ParaMiner we can write the *Select* predicate in any implementation language that can be interfaced with ParaMiner's implementation (currently `C++`). It is thus accessible to practitioners of many different domains, although the strong accessibility verification demands some mathematical skills. Thanks to parallelism ParaMiner is the fastest approach for the constraints it can handle. The relative simplicity of the algorithm make it a nice tool for data mining researchers to tinker with. For example, as seen in Chapter 4, ParaMiner applied to gradual itemset mining provides the fastest mining algorithm available to date for this problem, with minimal implementation effort.

Many constraint-based pattern mining approaches tend towards "final products" dedicated to some practitioners with specific constraints that they need to analyze in data. Such approaches culminate with the works of Bonchi and Lucchese [BL07] for efficiency and of Soulet [Sou06] for expressivity and ease of use.

Conversely, Guns et al. [GNR11] proposed a convergence with the more mathematical world of constraint programming, their work having the objective to be further extended and improved. ParaMiner has a similar vision, establishing a convergence between both generic pattern mining and the domains of closed pattern enumeration and parallelism, in order to provide a new basis to pattern mining researchers, with a heavy focus on computing efficiency.

5.1.2 Toolbox approaches

Since 2000, several prominent pattern mining researchers have devoted their work to mine patterns in *structured data*, i.e. data having a structure of sequence, tree or graph. For example the groups of Jiawei Han or the one of Mohamed J. Zaki have designed numerous notorious algorithms to mine various types of structured data: CloSpan [YHA03] and Spade [Zak01] for sequence mining, TreeMiner [Zak05a] or Sleuth [Zak05b] for tree mining and gSpan [YH02] for graph mining.

Jiawei Han's team proposed its algorithms in a toolbox called IlliMine[1], which include algorithms for mining frequent itemsets, sequences and graphs. The algorithms are not integrated together, so if the user wants to adapt an algorithm for a different pattern mining problem, he has to directly modify the code, which can be difficult.

Mohamed J. Zaki's team proposed a more integrated approach with the *Data Mining Template Library (*DMTL*)*[2] [AHCS⁺05]. In DMTL, the mechanic of classical pattern mining algorithms is abstracted, and a specific pattern mining task is instantiated by implementing specific sub-tasks needed for the mining:

- candidate generation, which boils down to take two existing patterns and join them to obtain a new candidate pattern with one more element ;
- isomorphism checking, which verifies that a candidate pattern has not been enumerated twice ;
- support counting, which returns the support of a candidate in the database.

This approach has been tested with patterns ranging from itemsets to graphs, and in [AHCS⁺05] it is shown that it can be easily used for cliques, which had not been addressed by a specific mining algorithm at that time. For genericity of implementation, the data structures and the functions make a heavy use of `C++` template mechanism.

Compared to ParaMiner, it is more easy to mine patterns with a complex structure in DMTL, as patterns in DMTL are natively expressed as graphs. ParaMiner's framework requires encoding the patterns as sets, which makes more difficult the handling of labelled sequences/trees/graphs having several times the same label. On the other hand, DMTL does not handle closed frequent patterns, dataset reduction or parallelism: its run-times are very slow. DMTL is also specialized for frequency, whereas ParaMiner can handle more general pattern interest measures. Last, if the C++ templates approach used in DMTL is very elegant from a design point of view, it can be difficult to master for non specialists.

It is also worth mentioning the works of Flouvat et al. with the iZi[3] library [FMP09]. iZi's goal is close to ParaMiner's: it deals with patterns representable as sets. It is focused on constrained-pattern mining problems where the constraint is anti-monotone or monotone. The constraint is given as a predicate, the user also needs to give a *set transformation* function that transforms the set representation of a pattern used by the algorithm into a pattern that can be checked by the predicate, and an *initialization component* that finds the patterns corresponding to singletons in the set representation. iZi's mining algorithm is based on Apriori, with FP-trees data structures in order to perform some dataset reduction.

One of the interest of iZi compared to ParaMiner is that it can give the upper and lower border of the results for a given predicate, which can be of interest for certain users. It also directly integrates monotone constraints. On the other hand, it does not manage closure and uses an outdated algorithmic base (Apriori). It is thus very slow, being beaten by ad-hoc Apriori implementations. It is thus several orders of magnitude slower than

[1]`http://illimine.cs.uiuc.edu`
[2]`http://dmtl.sourceforge.net`
[3]`http://liris.cnrs.fr/izi`

ParaMiner, and can only be used on small datasets or with high support values with the current implementation.

We presented in this section existing approaches for generic pattern mining and compare them with ParaMiner. ParaMiner is not only a generic pattern mining algorithm, it is also an efficient parallel pattern mining algorithm. We present below major works on parallel pattern mining and compare them with ParaMiner.

5.2 Parallel pattern mining

Pattern mining algorithms have always been time and memory consuming. This is a problem because the process of data analysis needs to be as interactive as possible. The only way to get smaller execution times and thus more interactivity is to exploit the additional computational power available in parallel computation platforms. Thus researchers in pattern mining have soon worked on the problem of designing efficient parallel pattern mining algorithms.

The works conducted by the pattern mining community to build parallel algorithms have shown that two problems are particularly important when designing parallel pattern mining algorithms. The first one consists in finding a good task decomposition. We will discuss the different approaches in Section 5.2.1. The second problem is to minimize data movement between the processing units and the memory, we will present several possible solutions proposed by the community in Section 5.2.2. In this thesis we focus on shared memory machines and more particularly on multi-core architectures. However, several works conducted before the emergence of multi-core architectures are also relevant to our problem, therefore we also present some work conducted on different parallel platforms such as clusters of computers.

5.2.1 Decomposing computations into tasks

Decomposing the computation into independent tasks and distributing them among available processing units has been shown to be a non trivial problem. Several works presented in this section show that naive task decomposition or naive task distribution strategies can lead to extensive communication/synchronization or load imbalance issues. Researchers have addressed this problem in the context of frequent itemset mining, graph mining, tree mining and closed itemset mining with variable success.

First works on parallel pattern mining have been conducted by Agrawal et al. [AS96]. They have proposed several parallel implementations of their Apriori algorithm for clusters of computers. Apriori is an iterative algorithm for mining frequent itemsets ([AS94]). In each loop, first a set of candidate patterns is generated, then the frequency of each candidate is evaluated. Agrawal et al. proposed several strategies to decompose the computations into independent tasks. We present two of them below.

- Count Distribution: The dataset is partitioned into equal parts and distributed to the nodes. Each node generates all the candidates and counts the support of these candidates on its local partition of the dataset. All the nodes exchange their local counts in order to obtain the global frequency count for each candidate.

- DATA DISTRIBUTION: The dataset is distributed on the nodes as in COUNT DISTRIBUTION. Each node generates a disjoint set of candidate patterns and counts their local frequency in its dataset partition. Each node then has to scan remote dataset partitions to compute the global frequency count of its candidates.

These two strategies are two different ways to split and distribute the work of APRIORI on several nodes. Agrawal et al. could show that COUNT DISTRIBUTION had the best result because it requires less communications between the nodes (only frequency counts are exchanged).

Zaki et al. in [ZPOL97b] proposed improvements upon the works of Agrawal et al. for parallel frequent itemset mining. They evaluate the performances of four parallel algorithms namely PAR-ECLAT, PAR-MAXECLAT, PAR-CLIQUE and PAR-MAXCLIQUE based on the sequential algorithms proposed in [ZPOL97a]. All the algorithms are based on the same principle: in an initialization phase the algorithms generate the frequent 2-itemsets and distribute them to the different processing units according to prefix based equivalence classes (PAR-ECLAT, PAR-MAXECLAT) or to Clique based equivalence classes (PAR-CLIQUE, PAR-MAXCLIQUE). Each processor has a copy of the sub dataset that is relevant to its equivalence class. This sub dataset is actually the support set of the 2-itemset originating the equivalence class. This is a simple version of the dataset reduction implemented in PARAMINER. It is well adapted to a depth first exploration of the search space. This is the first occurrence of this exploration technique for parallel pattern mining which has been used and improved later in many algorithms including PARAMINER.

Contrary to the previous approach by Agrawal et al. each processor can perform the computations without costly synchronization or communications. As a consequence this approach scales much better than the one of Agrawal et al. on large computation platforms.

Later in [BPC06], Buehrer et al. tackle the problem of designing a parallel algorithm for mining graph patterns. They proposed a parallel implementation of the well known GSPAN algorithm. This algorithm builds candidate graph patterns by growing frequent graph patterns with additional nodes. For this algorithm, naive decomposition of the search space based on the frequent 1-node subgraphs fails to provide correct load balancing because a single 1-node subgraph can originate over 50% of the total computation required. Buehrer et al. thus propose to use a dynamic task decomposition and a dynamic task distribution, explained below.

Buehrer and his colleagues evaluate several task distribution strategies, and retain the *distributed model* on the basis of experimental results. In this model each processor is assigned a task queue where it can push tasks into. If a processor is idle it pulls a task from its own queue in priority. But if the queue is empty it searches other queues in a round robin fashion.

The main contribution of this work lies in the dynamic task decomposition strategy adopted: the authors propose an *adaptive partitioning strategy* which commands the algorithm to create new tasks for a processor only when the number of pending tasks in its queue is below a given threshold. As a consequence, when the number of tasks is sufficient to balance the work, no new tasks are created. This reduces the overhead due to extensive task creation and it also limit data movements (see Section 5.2.2).

Their experiments show that their parallelization of GSPAN is able to reach a 22.5 to 27 fold speedup on a 32 cores machine. This demonstrates that adaptive partitioning successfully solves the problem of load imbalance in graph mining applications.

Building on this work Tatikonda et al. in [TP09] proposed a multi level work sharing approach that adaptively modulates the type and granularity of the tasks according to the platforms needs. This work was proposed in the context of a parallel tree mining algorithm. They define three levels of tasks: A first level task encompasses all the computations required to generate a complete equivalence class of tree, a second level task encompasses all the computations required to generate a single tree pattern and a last level task contains only a fraction of the computations required to generate a tree pattern. First and second level tasks can be dynamically subdivided into lower level tasks (i.e. smaller tasks). A dynamic task partitioning strategy switches to smaller tasks when load balancing is needed. Tatikonda et al. are able to achieve a 7.43 to 7.85 speed up on a computing platforms with 8 cores. It is worth noticing that is the first experiments that were conducted on an actual multi-core system. Previous experiments were conducted on traditional multi processors systems due to the lack of large scale multi-core architectures available. The performance of an algorithm may not be the same on the two types of platforms.

In [LOP07] Lucchese et al. have proposed to tackle the problem parallel *closed* frequent itemset mining. Lucchese et al. observe that closed pattern mining is a more complex problem because computing the closure *"needs a global view either of the dataset or of the collection of closed pattern mined so far"*. It makes the problem of task decomposition a tougher problem.

Their MT-Closed algorithm is based on DCI-Closed ([LOP04]) which recursively explores the set of frequent closed patterns. Each recursive call is parameterized with a closed frequent itemset, a PRE_SET and a POST_SET. The POST_SET contains the items that can be used to expand the current closed itemset whereas the PRE_SET contains the items that have been formerly used to expand the closed itemset. The PRE_SET is used to detect possible duplicate. It is worth noticing that POST_SET principle is very similar to the Exclusion List proposed by Boley et al. in [BHPW10] and used for enumeration in ParaMiner. In MT-Closed, every 2-itemset originates a parallel call to the recursive function. However since the PRE_SET is unknown at this time, it is set to $\emptyset$. This change, compared to the original DCI-Closed, may lead to generation of duplicate itemsets. In order to ensure its soundness, MT-Closed has to perform duplicates checks. Those duplicate checks are optimized by exploiting the SIMD vector instructions available in most modern processors.

The authors of MT-Closed evaluate static and dynamic task distribution strategies. They also implement an additional task decomposition system where an idle processor can *steal* work from another busy processor by subdividing the task into two new sub tasks. In this case splitting a task consists in distributing the set of possible extension (i.e. the POST_SET) to different cores. This technique is well known in by the parallel computing community as *work stealing* and has been shown to be adapted to irregular problems ([BL94]).

Their experiments show that static task decomposition and static task distribution does not provide correct load balancing. Static task decomposition with dynamic task distribution and dynamic task decomposition with dynamic task distribution provide better results with a slight advantage for the latter. However, for several datasets the algorithm is unable to show a good speedup. For instance, for the Connect dataset, the speedup is 4.5 on 8 cores.

Discussion

In ParaMiner we adopted a branch wise task decomposition strategy. This strategy can be parameterized according to the problem requirement to generate more or less tasks. Usually our tasks are fine grained enough to avoid load imbalance issues. Using work stealing strategies such as in [LOP07] is thus not necessary.

One of the main point of using Melinda for ParaMiner's task distribution is flexibility: it is easy to implement either a static or a dynamic task distribution strategy if the problem requires it. Interesting ideas such as Buehrer et al. load aware task distribution strategies can also be implemented as a Melinda strategy.

It is also worth noticing that the problem of task decomposition for closed pattern mining addressed by Lucchese et al in [LOP07] is solved in ParaMiner thanks to our pattern enumeration strategy. It allows independent exploration of different branches of the enumeration tree in any order. As a consequence we do not need to perform in ParaMiner any global operation such as duplicate checks.

5.2.2 Minimizing data movements

As we have shown in our experiments (Chapter 4) a correct task distribution is insufficient to guarantee performance. In parallel pattern mining algorithms reducing the amount of data movements between the main memory and the processing units is a critical issue. In the context of exploiting multi-core architectures this mostly consists in reducing the *bandwidth pressure*. We present in this section the most significant works in this direction.

In [GBP$^+$05], Ghoting et al. observe that over 50% of the operation performed are memory operations. This measure reveals the importance of having fast memory operations in pattern mining, which can be achieved through improving the cache usage. They also observe that an important number of architectural innovations embedded in new processors to bridge the gap between memory and cores are widely ignored by pattern mining algorithms. For example pointer based structures such as FP-Trees used in the FP-Growth algorithm does not combine well with caches and cache line prefetchers, which fail to provide the improvements they are supposed to. Ghoting at al. proposed *cache conscious prefix trees* which are essentially FP-Trees restructured to improve cache locality. The cache conscious prefix tree is a prefix tree were each node of the tree is stored in consecutive blocks in memory following the depth first search order. This way of storing the tree accommodates bottom up traversals which are frequent in FP-Growth.

In order to further improve the cache usage Ghoting et al. have also proposed the *path tiling* optimization. This optimizations is based on the observation that most FP-Trees do not fit into caches. Therefore different parts of the trees have to be loaded several times into the cache, once for each step of the computation that needs to traverse the tree. Path tiling consists in breaking the FP-Trees into *tiles*. Each tile is a subset of nodes stored consecutively in the memory. The authors have modified FP-Growth such that all the computations to be performed on a tile are performed consecutively. As a consequence each tile is loaded once into the cache and can be purged.

Thanks to its improvement and other optimizations, Ghoting et al. are able to reach 30% to 60% improvement in execution time compared to a fast implementation of FP-Growth.

Later in [TP09], Tatikonda and Parthasarathy observed that algorithms must maintain small *working sets* to deliver good performance. The working set is the amount of data actively used by the program during a particular phase of computation. Based on this observations they have conducted an extensive study to evaluate the working set size of several existing tree mining algorithms namely TreeMiner ([Zak02]), iMB3-T ([TDH$^+$06]) and Trips ([TPK06]) and chose to focus their work to the Trips algorithm that have the smallest working set.

In sequential algorithms it is common to trade space for improved execution times. For example a majority of tree mining algorithm maintain an *embedding list* to speedup the mining process. An embedding list is created for each new pattern and contains location information useful to locate the pattern occurrences in the input trees. They are used to speedup the process of testing whether a child pattern occurs in an input tree.

Embedding lists are redundant information and occupy an important amount of memory. In a sequential algorithm it is worth the cost as long as there is enough memory available to run the algorithm. However in a parallel algorithm the overheads induced by the additional memory transfers drastically reduce the performances. Instead of carrying the embedding list together with the pattern tree, Tatikonda et al. have proposed to build these embedding list *on the fly*. Although it increases the amount of computation required, it drastically reduces the working set size of their algorithm.

The authors of [TP09] have demonstrated the importance of reducing the working set by proposing a parallel implementation of Trip. Thanks to the reduced working set and other similar memory optimizations not detailed here, they are able to reduce the memory footprint of Trip by 366 times and improve achieved a near-linear speedup on a 8-core machine.

These results highlight the fact that modern algorithm must be designed by taking into consideration of both the computations and the memory transfers. This is in line with the experiments by Agrawal et al. in [AS96], who had better performances with their Count distribution algorithm with redundant computation but fewer data transfers. In [TP09], Tatikonda et al. summarize it by saying that: *"Essentially parallelization without identifying memory conscious optimizations [. . .] is extremely inefficient"*.

In [NTMU10], we presented the details of our parallel closed frequent itemset mining algorithm. PLCM is based on the sequential LCM algorithm ([UKA04]). In LCM the dataset reduction process involves a sorting algorithm. Transactions are sorted using an algorithm based on the principle of *radix sort*: in a first pass transactions' identifiers are stored into *buckets*, each bucket for a different number. Once the transaction identifiers are correctly arranged into buckets a second pass sorts the transactions. Several iterations are needed to completely sort the transactions, but the algorithm is able to achieve linear time sorting of the transactions by using additional memory (under the form of buckets). Although this strategy exhibit good performances in LCM algorithm we demonstrated that it was not a good way to sort the transactions in a parallel algorithm. Indeed this algorithm exhibit very random memory access patterns that prevent caches to be efficient. We designed a new dataset reduction algorithm based on the simpler in-place quick sort algorithm. Quicksort is slower in generally slower than radix sort, but it reduces the working set size and thus improve the locality. Thanks to this change we were able to achieve a 15% to 50% performance improvement in speedup.

Discussion

In ParaMiner, the working set size is reduced through dataset reduction. This technique has been proved to be very efficient in the sequential case. In a parallel setting results depends on the dataset and on the mining task at hand. For large datasets and simple mining tasks such as closed frequent itemsets many cache evictions occur, leading to bad locality and reduced speedups. In this case ParaMiner would benefit from the tiling technique proposed by [GBP+05] and described in this section. We further discuss this point in our Future Works, Chapter 6.

5.2.3 Conclusion on parallel pattern mining

Studying these work highlighted that the theoretical performances of the execution platform cannot be reached without deep changes in the algorithms. These changes are typically closely related to the algorithm itself and the execution plaforms. As a consequence we observe an important trend in the design of efficient parallel pattern mining algorithm: they tend to be more and more *architecture conscious*. It means that they are aware of the machine features and limits. For example *cache conscious* algorithms ([GBP+05]), load adaptive algorithms ([BPC06]), memory conscious algorithms ([TP09]) or algorithms using SIMD processor's instruction set [LOP07].

Designing this type of algorithm require a high expertise in high performance computing and as well as a good knowledge in pattern mining algorithms. For example in Ghoting et al. in [GBP+05] design new *FP-Trees* to improve the efficiency of the *hardware cache line prefetchers*. In addition these algorithm are usually not portable due to their dependency to the computing platform.

By designing Melinda we tried and succeeded to a certain extent to decouple the algorithmic issues and the platform details. All the interaction between the two are described in Melinda's strategy that can be understood with basic knowledge on the algorithm bahaviour and the platform architecture. Hopefully ParaMiner and Melinda can be used as an experimentation platform to experiment and exchange knowledge concerning both the algorithms and the computing plaforms.

Another important observation that can be made is that algorithms with higher computational requirements tend to exhibit better speedups than high-end optimized algorithms. Indeed, when more arithmetic instructions are executed, the bus is less frequently solicited and it is possible to overlap communication delays with computations. Conversely, when the amount of computations are reduced to the minimum required, it is harder to overlap communications delays with computations, consequently such algorithms are generally more bandwidth demanding. This phenomenon is illustrated by the fact that Count-Distribution by Agrawal et al. is able to achieve good speedups on clusters of computers with very low bandwidth and high latency memory operations whereas MT-Closed and PLCM exhibit some speedup issues on multi-core architectures with much higher bus capabilities. In [LOP07] Lucchese et al. compare an execution of MT-Closed with and without their *projection* optimization. This optimization is similar to dataset reduction and drastically reduces the amount of computations required to count the frequency of a candidate pattern. Although they do not discuss phenomenon in their publication, their experiments clearly show that with projections enabled, their algorithm has lower execution times but also marginally lower speedups. Our experiments on ParaMiner also illustrate this effect: ParaMiner exhibits a much better speedup on the complex gradual itemset mining problem than on the frequent itemset mining problem and the relational graph problem which are comparatively simpler problems.

Although one could be tempted to parallelize simpler implementations to achieve better speedups it is important to keep in mind that the parallelism can only offer a linear speedup with the number of cores whereas algorithmic optimizations can drastically reduce the algorithm complexity. For example, we recall that even if Count-Distribution exhibit good speedups, it is still orders or magnitude slower than modern algorithms. We will conclude by saying that algorithmic optimization must not be disregarded but rather adapted to collaborate with parallelism.

Chapter 6

Conclusion

In the pattern mining field, many ad-hoc algorithms solve similar problems. Each of these algorithms is highly optimized with respect to its problem and is difficult to adapt to variations of this problem. This unnecessarily complex setting holds back progress in pattern mining research and large scale adoption of data analysis with pattern mining.

Based on a set of important pattern mining problems we proposed a generic framework for pattern mining algorithms. This framework extends state of the art pattern enumeration strategies with the notion of dataset, central to pattern mining. We also proposed in this framework a new property to characterize pattern mining problems having a well defined closure operator.

We have shown that this framework captures many distinct pattern mining problems such as closed frequent itemset mining, closed frequent relational graph mining and closed frequent gradual itemset mining.

Our second contribution is an efficient generic and parallel pattern mining algorithm: ParaMiner. ParaMiner is able to solve any problem that can be expressed in our framework. One of our main achievement in ParaMiner is an efficient dataset reduction technique which generalizes several state of the art optimizations. Up to now these optimization could only be used in some ad-hoc algorithms. Another achievement is that ParaMiner is a parallel program, able to exploit parallel multi-core computers. We proposed Melinda as a parallelism engine for ParaMiner. Melinda is flexible enough to accommodate the various characteristics of the different pattern mining problems and the different computation platforms.

We have conducted in Chapter 4 a thorough experimental study to evaluate ParaMiner's efficiency. First we have shown that our dataset reduction technique was able to dramatically reduce the computation time. This allows ParaMiner to tackle real worlds datasets. Next, we have shown that ParaMiner could scale well on recent computers with four cores. On larger platforms we shown that ParaMiner hit the *memory wall* problem, well known in the parallelism community. We proposed solutions based on Melinda strategies. Last, we have shown ParaMiner is able to compete with carefully optimized ad-hoc algorithms such as MT-Closed or PLCM for the frequent itemset mining problems. For other problems such as the problem of gradual itemset mining, ParaMiner is even one order of magnitude faster than the state of the art algorithm PGLCM on real world datasets.

ParaMiner is the first generic pattern mining algorithm exhibit such performances

In the Chapter 5, we positioned our approach with other generic approaches based on constraints and could find a relationship between constraints and the accessibility properties exploited in ParaMiner. We also reviewed the main parallel pattern mining approaches. Most of these approaches tackle the complex problems that we encountered in our experiments through deep modifications of the algorithm which cannot yet be used in a generic framework. However we will discuss several ideas that can be retained for improvement in the perspectives bellow.

Any pattern mining problem encoded in our framework immediately benefits from an efficient and parallel algorithmic solution: ParaMiner. ParaMiner thus allows to freely experiment new pattern definitions, and test them on real datasets. For such new problems, ParaMiner can be the new *de facto* baseline algorithm. This allows for example to estimate how much gain a new ad-hoc approach can bring, helping in the progress of pattern mining research. As a consequence, ParaMiner is already used in other tasks of our laboratory. It is used to mine execution traces generated by *bit accurate* processor simulators. This work is conducted in the context of the Ph.D of Sofiane Lagraa (Date'11 paper in submission).

ParaMiner is also used in the Ph.D works of Patricia Lopez Cueva to explore a new definition of *periodic patterns* (SDM'12 paper in submission). Periodic pattern are used to mine traces generated by video decoding algorithms on embedded processors.

Future works

There exist a large variety of works that can be conducted to improve both ParaMiner and Melinda. Most of these work fall into three categories: the works to improve efficiency, usability and genericity.

Improving ParaMiner's/Melinda's efficiency

Since pattern mining typically requires long processing times, improving the efficiency is closely related to improving usability. We already put a lot of effort to keep ParaMiner efficient. We detail below approaches for further improvements.

As the number of cores available on multi-core architectures increases, performing more computations to reduce the amount of data movements is becoming more worthwhile. In ParaMiner we can adopt this design strategy by implementing *tiling* techniques such as the ones proposed in [GBP$^+$05] and described in Section 5.2.2. In ParaMiner it would consist in breaking up the larger datasets into chunks, and performing all the computations on a chunk before moving to the next one. It would require to rearrange the computations in ParaMiner which could lead to lower sequential performances but this additional cost would be absorbed by the speedup increase on large multi-core architectures.

The rationale behind the design of Melinda is to provide its user a way to control the distribution strategy. In Melinda *internals* are used to organize the tuples in the tuple-space. This method provides a simple way to sort the tuples and retrieve them efficiently. However there are no means to query a specific tuple in an internal. This choice was deliberate to ensure Melinda's efficiency. A different approach is to provide Melinda's users a more advanced tuple querying system. For example we can integrate in the tuple-space a simple database system in order to allow SQL like tuple querying. This would obviously induce an important additional cost when retrieving tuples, however it would also allow the implementation of a whole new range of more complex MELINDA's strategies. For example, we could implement a strategy that carefully distributes the tuples on the processor such that the sum of the datasets is small enough to fit in the cache memory.

Although we focus on multi-core architectures, it is worth noticing that PARAMINER can be modified to exploit larger parallel platforms such as clusters of computers. Thanks to our decomposition of the search space into independent tasks and the dataset reduction technique no task migration is required *a priori*. In practice however, it can be required to balance the work on cluster with thousands of nodes. To reduce the impact of task migration, we can try to estimate the amount of work associated with each task. For doing so, there exists simple heuristics based for example on the dataset size or the number of pattern extensions, but these heuristics typically fail to provide accurate estimations of the computation required to complete a task. A better idea is to use the work proposed by Boley et al. in [BG08]. In this work, the authors are able to accurately estimate the number of frequent itemsets in a dataset with a probabilistic method. If this approach is successful on frequent itemset mining algorithms, it is an interesting question to know whether this work can be generalized to other types of patterns.

Improving PARAMINER's usability

In order to use PARAMINER for a new pattern mining problem, a practitioner has first to define a selection criterion for this problem. Then he/she has to prove that the associated set system is strongly accessible in order to ensure that PARAMINER can discover all the closed patterns satisfying the selection criterion. This step requires mathematical skills and may prevent PARAMINER's adoption to some pattern mining practitioners. As a perspective we would like to provide the practitioner a way to write selection criterions that are guaranteed to be strongly accessible. We already have seen that the composition of strongly accessible and independence set system led to a strongly accessible set system. It would thus be interesting to provide a list of elementary selection criterions whose accessibility properties are known and a list of operators to combine them together, while preserving the strong accessibility. The rationale behind this idea is to have an algebra on strongly accessible set systems, like the relational algebra of the database world. With such an algebra we could define a declarative language to easily write strongly accessible selection criterion. This language could be to pattern mining what SQL is to the relational algebra.

Improving genericity

Our most important goal is to extend PARAMINER's genericity. So far, some important pattern mining problems cannot be solved efficiently.

First, we proposed in Chapter 5.1.1 the notion of partial (strong) accessibility. We have shown that this notion allows to express the accessibility property of many constraint based pattern mining problems. From partial (strong) accessibility we can deduce that when patterns are above the lower border of the set system, they can be enumerated efficiently with the enumeration strategy presented in Chapter 2. It would thus be interesting to propose hybrid algorithms that use state of the art techniques from constraint based pattern mining in order to discover the lower border of the set system and then switch to ParaMiner's enumeration strategy above the lower border. ParaMiner could be extended to operate this way, it would further increase the number of problem that can be solved efficiently with it.

Another problem of interest is the problem of mining frequent sequences in a sequence database. Any sequence (input sequence or pattern) can be encoded into sets by using precedences elements. For instance, the sequence $[a, b, d]$ can be encoded to the set $\{a \rhd b, a \rhd d, b \rhd d\}$.

In sequences however, the same element can be repeated several times. We can solve this problem by adding indexes, for example the sequence $[a, b, a]$ can be encoded with the following set: $\{a_1 \rhd b_1, a_1 \rhd a_2, b_1 \rhd a_2\}$.

However this encoding presents an important drawback, the encoding of the sequence $[b, a]$ is not included (w.r.t. set inclusion) in the encoding of the sequence $[a, b, a]$: $\{b_1 \rhd a_1\} \nsubseteq \{a_1 \rhd b_1, a_1 \rhd a_2, b_1 \rhd a_2\}$.

This problem can be solved with an adequate encoding of the input dataset, but the size of the resulting dataset would be combinatorial with the number of repeated elements, thus we do not consider this as an efficient solution.

In ParaMiner, the set inclusion is used to test if a pattern occurs in a transaction. To solve problem mention above, we can replace this set inclusion by a more flexible inclusion relation. For example, one that is able to detect that the coded sequence $\{b_1 \rhd a_1\}$ is included in $\{a_1 \rhd b_1, a_1 \rhd a_2, b_1 \rhd a_2\}$. Although it would not break the soundness of the enumeration strategy, this would inevitably invalidate some important optimizations in ParaMiner. However, it is an interesting question to know if we could adapt these optimizations and have good performances.

This approach to solve the sequence problem is based on the set system theory used in combination with a non trivial encoding of the sequence problem. Another interesting approach to solve the same problem is to extend the accessibility properties to other algebraic structures such as sequences or general graphs. We illustrate this idea bellow.

We observe that given a database of sequences, an augmentation relation similar to the one for patterns represented as sets can be defined (see Definition 2.11, Page 21). Since it is also possible to define an order on the sequences, the set of frequent sequences can be represented as a DAG in a fashion similar to the one presented in Figure 2.5, Page 21. In this DAG structural properties similar to the strong accessibility can be defined. It would be interesting to know to what extent it is possible to use such properties to efficiently enumerate closed sequences.

Bibliography

[AHCS^{+}05] M. Al Hasan, V. Chaoji, S. Salem, N. Parimi, and M.J. Zaki. Dmtl: A generic data mining template library. *Library-Centric Software Design (LCSD'05)*, page 53, 2005.

[ALYP10] Sarra Ayouni, Anne Laurent, Sadok Ben Yahia, and Pascal Poncelet. Mining closed gradual patterns. In *ICAISC*, pages 267–274, 2010.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[AS96] R. Agrawal and J.C. Shafer. Parallel mining of association rules. *Knowledge and Data Engineering, IEEE Transactions on*, 8(6):962–969, 1996.

[AU09] Hiroki Arimura and Takeaki Uno. Polynomial-delay and polynomial-space algorithms for mining closed sequences, graphs, and pictures in accessible set systems. In *SDM*, pages 1087–1098, 2009.

[BG08] Mario Boley and Henrik Grosskreutz. A randomized approach for approximating the number of frequent sets. In *Proceedings of the IEEE International Conference on Data Mining*, 2008.

[BHPW07] Mario Boley, Tamás Horváth, Axel Poigné, and Stefan Wrobel. Efficient closed pattern mining in strongly accessible set systems. In *Mining and Learning with Graphs (MLG)*, 2007.

[BHPW10] Mario Boley, Tamás Horváth, Axel Poigné, and Stefan Wrobel. Listing closed sets of strongly accessible set systems with applications to data mining. *Theor. Comput. Sci.*, 411(3):691–700, 2010.

[BL94] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. 1994.

[BL04] Francesco Bonchi and Claudio Lucchese. On closed constrained frequent pattern mining. In *ICDM*, pages 35–42, 2004.

[BL07] Francesco Bonchi and Claudio Lucchese. Extending the state-of-the-art of constraint-based pattern discovery. *Data Knowl. Eng.*, 60(2):377–399, 2007.

[BPC06] Gregory Buehrer, Srinivasan Parthasarathy, and Yen-Kuang Chen. Adaptive parallel graph mining for cmp architectures. In *ICDM*, pages 97–106, 2006.

[DJLT09] L. Di-Jorio, A. Laurent, and M. Teisseire. Mining frequent gradual itemsets from large databases. *Advances in Intelligent Data Analysis VIII*, pages 297–308, 2009.

[DLT10] Trong Dinh Thac Do, Anne Laurent, and Alexandre Termier. Pglcm: Efficient parallel mining of closed frequent gradual itemsets. In *ICDM*, pages 138–147, 2010.

[ETMT10] Serge Emteu, Maurice Tchuente, Jean-François Méhaut, and Alexandre Termier. Vers un partage de travail adapté a la hierachie de cache. Master's thesis, Université de Yaounde 1, 2010.

[FMP09] Frédéric Flouvat, Fabien De Marchi, and Jean-Marc Petit. The izi project: Easy prototyping of interesting pattern mining algorithms. In *PAKDD Workshops*, pages 1–15, 2009.

[GBP$^+$05] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the 31st international conference on Very large data bases*, pages 577–588. VLDB Endowment, 2005.

[Gel85] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[GNDR11] T. Guns, S. Nijssen, and L. De Raedt. Itemset mining: A constraint programming perspective. *Artificial Intelligence*, 175(12-13):1951–1983, 2011.

[GNR11] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175(12-13):1951–1983, 2011.

[Goe03] Bart Goethals. Fimi repository website. `http://fimi.cs.helsinki.fi/`, 2003.

[Gra66] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[HMJ$^+$00] T.R. Hughes, M.J. Marton, A.R. Jones, C.J. Roberts, R. Stoughton, C.D. Armour, H.A. Bennett, E. Coffey, H. Dai, Y.D. He, et al. Functional discovery via a compendium of expression profiles. *Cell*, 102(1):109–126, 2000.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM SIGMOD Record*, 29(2):1–12, 2000.

[IGM01] S. Imoto, T. Goto, and S. Miyano. Estimation of genetic networks and functional structures between genes by using bayesian networks and nonparametric regression. In *Pacific Symposium on Biocomputing 2002: Kauai, Hawaii, 3-7 January 2002*, page 175. World Scientific Pub Co Inc, 2001.

[IWM00] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. *Principles of Data Mining and Knowledge Discovery*, pages 13–23, 2000.

[KDRH01] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in hiv data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 136–143. ACM, 2001.

[KTF09] U. Kang, C.E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[LNST10] Anne Laurent, Benjamin Négrevergne, Nicolas Sicard, and Alexandre Termier. Pgp-mc: Towards a multicore parallel approach for mining gradual patterns. In *DASFAA (1)*, pages 78–84, 2010.

[LOP04] C. Lucchese, S. Orlando, and R. Perego. Dci closed: A fast and memory efficient algorithm to mine frequent closed itemsets. In *IEEE ICDM'04 Workshop FIMI'04*. Citeseer, 2004.

[LOP07] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Parallel mining of frequent closed patterns: Harnessing modern computer architectures. In *ICDM*, pages 242–251, 2007.

[MN08] S.A. Macskassy and C.C. Nanjo. Graph mining using graph pattern profiles. In *Proceedings of the 2008 International Conference on Artificial Intelligence*. Citeseer, 2008.

[MT97] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3):241–258, 1997.

[NLHP98] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained association rules. In *SIGMOD Conference*, pages 13–24, 1998.

[NTM08] Benjamin Negrevergne, Alexandre Termier, and Jean-François Méhaut. Algorithmes parallèles pour la fouille de donnée structurées. Master's thesis, Université Joseph Fourier, 2008.

[NTMU10] Benjamin Negrevergne, Alexandre Termier, Jean-Francois Mehaut, and Takeaki Uno. Discovering closed frequent itemsets on multicore: Parallelizing computations and optimizing memory accesses. In *HPCS*, pages 521–528, 2010.

[PBTL99] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416, 1999.

[PH00] Jian Pei and Jiawei Han. Can we push more constraints into frequent pattern mining? In *KDD*, pages 350–354, 2000.

[PHL01] Jian Pei, Jiawei Han, and Laks V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *ICDE*, pages 433–442, 2001.

[PHM00] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[SC05] Arnaud Soulet and Bruno Crémilleux. An efficient framework for mining flexible constraints. In *PAKDD*, pages 661–671, 2005.

[Sou06] Arnaud Soulet. *Un cadre générique de découverte de motifs sous contraintes fondées sur des primitives.* PhD thesis, Université de Caen, 2006.

[SVA97] Ramakrishnan Srikant, Quoc Vu, and Rakesh Agrawal. Mining association rules with item constraints. In *KDD*, pages 67–73, 1997.

[SY07] Xingzhi Sun and Philip S. Yu. Hiding sensitive frequent itemsets by a border-based approach. *JCSE*, 1(1):74–94, 2007.

[TDH$^+$06] H. Tan, T. Dillon, F. Hadzic, E. Chang, and L. Feng. Imb3-miner: Mining induced/embedded subtrees by constraining the level of embedding. *Advances in Knowledge Discovery and Data Mining*, pages 450–461, 2006.

[TP09] S. Tatikonda and S. Parthasarathy. Mining tree-structured data on multicore systems. *Proceedings of the VLDB Endowment*, 2(1):694–705, 2009.

[TPK06] S. Tatikonda, S. Parthasarathy, and T. Kurc. Trips and tides: new algorithms for tree mining. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 455–464. ACM, 2006.

[TTN$^+$07] A. Termier, Y. Tamada, K. Numata, S. Imoto, T. Washio, and T. Higuchi. Digdag, a first algorithm to mine closed frequent embedded sub-dags. In *Proceedings of Mining and Learning with Graphs Workshop (MLG'07)*, pages 41–45. Citeseer, 2007.

[UAUA04] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In *Discovery Science*, pages 16–31, 2004.

[UKA04] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

[UKA05] T. Uno, M. Kiyomi, and H. 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*, pages 77–86. ACM, 2005.

[WM95] W.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–20, 1995.

[YH02] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *Proc. 2002 of Int. Conf. on Data Mining*, 2002.

[YHA03] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *Proceedings of SIAM International Conference on Data Mining*, pages 166–177, 2003.

[YZH05] Xifeng Yan, Xianghong Jasmine Zhou, and Jiawei Han. Mining closed relational graphs with connectivity constraints. In *ICDE*, pages 357–358, 2005.

[Zak01] Mohammed J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42(1/2):31–60, Jan/Feb 2001. special issue on Unsupervised Learning.

[Zak02] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference Knowledge Discovery and Data Mining*, Jul 2002.

[Zak05a] M.J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, pages 1021–1035, 2005.

[Zak05b] Mohammed J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2):33–52, Mar/Apr 2005. special issue on Advances in Mining Graphs, Trees and Sequences.

[ZPOL97a] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug 1997.

[ZPOL97b] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1(4):343–373, Dec 1997. Special issue on Scalable High-Performance Computing for KDD.