The iZi Project: Easy Prototyping of Interesting Pattern Mining Algorithms

Frédéric Flouvat[1], Fabien De Marchi[2], and Jean-Marc Petit[2]

[1] University of New Caledonia, PPME, F-98851, Noumea, New Caledonia
frederic.flouvat@univ-nc.nc
[2] Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69621, France
fabien.demarchi@liris.cnrs.fr
[3] Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205, F-69621, France
jean-marc.petit@insa-lyon.fr

Abstract. In the last decade, many data mining tools have been developed. They address most of the classical data mining problems such as classification, clustering or pattern mining. However, providing classical solutions for classical problems is not always sufficient.

This is especially true for pattern mining problems known to be "representable as set", an important class of problems which have many applications such as in data mining, in databases, in artificial intelligence, or in software engineering. A common idea is to say that solutions devised so far for classical pattern mining problems, such as frequent itemset mining, should be useful to answer these tasks. Unfortunately, it seems rather optimistic to envision the application of most of publicly available tools even for closely related problems.

In this context, the main contribution of this paper is to propose a modular and efficient tool in which users can easily adapt and control several pattern mining algorithms. From a theoretical point of view, this work takes advantage of the common theoretical background of pattern mining problems isomorphic to boolean lattices. This tool, a C++ library called *iZi*, has been devised and applied to several problems such as itemset mining, constraint mining in relational databases, and query rewriting in data integration systems. According to our first results, the programs obtained using the library have very interesting performance characteristics regarding simplicity of their development. The library is open source and freely available on the Web.

1 Introduction

In the last decade, many data mining tools have been developed [1]: standalone algorithm implementations [2,3], packages [4], libraries [5], complete softwares with GUI [6,7] or inductive databases prototypes [8,9] . They address most of the classical data mining problems such as classification, clustering or pattern mining.

T. Theeramunkong et al. (Eds.): PAKDD Workshops 2009, LNAI 5669, pp. 1–15, 2010.

However, providing classical solutions for classical problems is not always sufficient. For example, frequent itemset mining (FIM) is a classical data mining problems with applications in many domains. Many algorithms and tools have been proposed to solve this problem. Moreover, several works, such as [10], shown that FIM algorithms can be used as a building block for other, more sophisticated pattern mining problems. This is especially true for pattern mining problems known to be "representable as set" [10], an important class of problem which have many applications such as in data mining (e.g. frequent itemset mining and variants [11,12]), in databases (e.g. functional or inclusion dependency inference [13,14]), in artificial intelligence (e.g. learning monotone boolean function [15]), or in software engineering (e.g. software bug mining [16]).

In this setting, a common idea is to say that algorithms devised so far should be useful to answer these tasks. Unfortunately, it seems rather optimistic to envision the application of most of publicly available tools for frequent itemset mining, even for closely related problems. For example, frequent essential itemset mining [17] (as well as other conjunctions of anti-monotone properties) is very closely related to FIM. Actually, only the predicate test is different. In the same way, mining keys in a relational database is a pattern mining problem where, from a theoretical point of view, FIM strategies could be used. However, in both cases, users can hardly adapt existing tools to their specific requirements, and have to re-implement the whole algorithms.

Paper contribution. In this context, the main contribution of this paper is to propose a modular and efficient tool in which users can easily adapt and control several pattern mining algorithms. From a theoretical point of view, this work takes advantage of the common theoretical background of pattern mining problems isomorphic to boolean lattices. This tool, a C++ library called *iZi*, has been devised and applied to several problems such as itemset mining, constraint mining in relational databases and query rewriting in data integration systems. According to our first results, the programs obtained using the library have very interesting performance performance characteristics regarding simplicity of their development. The library is open source and freely available on the Web.

Paper organization. Section 2 discusses the value of our proposition w.r.t. existing related works. Section 3 introduces the *iZi* library. This section presents the underlined theoretical framework, points out how state of the art solutions can be exploited in our generic context, and describes the architecture of the *iZi* library. A demonstration scenario is presented in Section 4. Experimentations are described in Section 5. The last section concludes and gives some perspectives of this work.

2 Related Works

One may notice that algorithm implementations for pattern mining problems are "home-made" programs, see for example implementations available in FIMI workshops [2,3].

Packages, libraries, software, inductive databases prototypes have also been proposed, for instance Illimine [4], DMTL [5], Weka [6], ConQuest [8] and [9].

Except DMTL, they provide classical algorithms for several data mining tasks (classification, clustering, itemset mining...). However, their algorithms are very specific and could not be used to solve equivalent or closely related problems. For example, even if most of these tools implement an itemset mining algorithm, none of them can deal with other interesting pattern discovery problems. Moreover, their source codes are not always available.

DMTL (Data Mining Template Library) is a C++ library composed of algorithms and data structures optimized for frequent pattern mining. Different types of frequent patterns (sets, sequences, trees and graphs) using generic algorithms implementations are available. Actually, DMTL supports any types of patterns representable as graphs. Moreover, the data is decoupled of the algorithms, and can be stored in memory, files, Gigabase databases (an embedded object relational database), and PSTL [18] components (a library of persistent containers). This library currently implements an exploration strategy: a depth-first approach (*eclat*-like [19]). Moreover, some support for breadth-first strategies is also provided. These algorithms could be used to mine all the frequent patterns of a given database.

To our knowledge, only the DMTL library has objectives close to *iZi*. Even if objectives are relatively similar w.r.t. code reusability and genericity, the motivations are quite different: while DMTL focuses on patterns genericity w.r.t. the frequency criteria only, *iZi* focuses on a different class of patterns but on a wider class of predicates. Moreover, *iZi* is based on a well established theoretical framework, whereas DMTL does not rely on such a theoretical background. However, DMTL encompasses problems that cannot be integrated into *iZi*, for instance frequent sequences or graphs mining since such problems are not isomorphic to a boolean lattice. The iZi library is complementary to DMTL since it offers the following new functionalities:

1. any monotone predicate can be integrated in *iZi*, while DMTL "only" offers support for the "frequent" predicate;
2. the structure of the patterns does not matter for *iZi*, while the patterns studied by DMTL must be representable as graphs (e.g. inclusion dependencies cannot be represented in DMTL);
3. while DMTL only gives all frequent patterns, *iZi* is able to supply different borders of "interesting" patterns (positive and negative borders). These borders are the solutions of many pattern mining problems. Moreover, end-users often do not care about all the patterns and prefer a smaller representation of the solution.

3 A Generic and Modular Solution for Patterns Discovery

3.1 A Generic Theoretical Framework

The theoretical framework of [10] formalizes *enumeration problems under constraints*, i.e. of the form "enumerate all the patterns that satisfy a condition".

When the condition must be verified in a data set, the word "enumerate" is commonly replaced by "extract". Frequently, the problem specification requires that patterns must be maximal or minimal w.r.t. some natural order over patterns.

Consequently, common characteristics of these problems are: 1) the predicate defining the interestingness criteria is monotone (or anti-monotone) with respect to a partial order $\preceq$ over patterns, 2) there exists a bijective function f from the set of patterns to a boolean lattice and its inverse f^{-1} is computable, and 3) the partial order among patterns is preserved, i.e. $X \preceq Y \Leftrightarrow f(X) \subseteq f(Y)$.

3.2 Algorithms

The classical way to solve pattern mining problems is to develop ad-hoc solutions from scratch, with specialized data structures and optimization techniques. If such a solution leads to efficient programs in general, it requires a huge amount of work to obtain a sound and operational program. Moreover, if problem specifications slightly change over time, a consequent effort should be made to identify what parts of the program should be updated.

One of our goal is to factorize some algorithmic solutions which can be common to any pattern mining problem representable as sets.

Currently, many algorithms from the multitude that has been proposed for the FIM problem could be generalized and implemented in a modular way, from well knowns *Apriori* algorithm [20] or depth-first approaches, to more sophisticated dualization-based algorithms (*Dualize and Advance* [21] or *ABS* [14,22]).

However, some algorithms don't fit in this framework because they are not based on a clear distinction between the exploration strategy and the problem. For example, FP-growth like algorithms [23] cannot be used into this framework since their strategy is based on a data structure specially devised for FIM. In the same way, condensed representations based algorithms like LCM [24] cannot be applied to any pattern mining problem representable as sets.

The need to have multiple strategies in a pattern mining tool is twofold. First, note that the type of solution discovered by each algorithm is specific. For example, the *Apriori* algorithm discover (without any overhead) the theory and the two borders, whereas dualization-based algorithms "only" discovers the two borders. Since depending on the studied problem, we might be interested in either the theory, or the positive border, or the negative border, it is necessary to have multiple strategies to enable the discovery of the required solution. Secondly, as shown by the FIMI workshops, the algorithms performance depend on dataset/problem characteristics. For example, the *Apriori* algorithm is more appropriate when the theory is composed of relatively small elements, i.e. solutions are small patterns. Consequently, several algorithms must be integrated into a pattern mining tool to have the best performances according to problem properties.

3.3 The iZi Library

Based on the theoretical framework introduced in Section 3.1, we propose a C++ library, called *iZi*, for this family of problems. The basic idea is to offer a toolkit

providing several efficient, robust, generic and modular algorithm implementations. The development of this library takes advantage of our past experience to solve particular pattern mining problems such as frequent itemsets mining, functional dependencies mining, inclusion dependencies mining and query rewriting.

Architecture. Figure 1 represents the architecture and the "workflow" of our library: The *algorithm* is initialized (*initialization* component) with patterns corresponding to singletons in the set representation, using the data (*data access* component). Then, during the execution of the algorithm, the *predicate* is used to test each pattern against the data. Before testing an element, the algorithm use the *set transformation* component to transform each set generated into the corresponding pattern.

This architecture is directly derived from the studied framework and has the main advantage of decoupling algorithms, patterns and data. Only the *predicate*, *set transformation* and *initialization* components are specifics to a given problem. Consequently, to solve a new problem, users may have to implement or reuse with light modifications some of these components.

The *algorithm* component represents generic algorithm implementations provided with the library and used to solve pattern mining problems. As shown in Figure 1, algorithms are **decoupled from the problems** and are a **black box for users**. Each algorithm can be used directly to solve any problem fitting in the framework without modifications. This leads to the rapid construction of robust programs without having to deal with low level details. Currently, the library offers a levelwise algorithm [10], a dualization-based algorithm, and two other variants of these algorithms. These variants globally have the same strategy but explore the search space in a different way (top-down exploration instead of bottom-up) which is more appropriate for some predicates. Finally, depth-first strategies are also currently being integrated.

Another important aspect of our library is that data access is totally decoupled of all other components (see Figure 1). Currently, data access in most of the other implementations is tightly coupled with algorithm implementations and

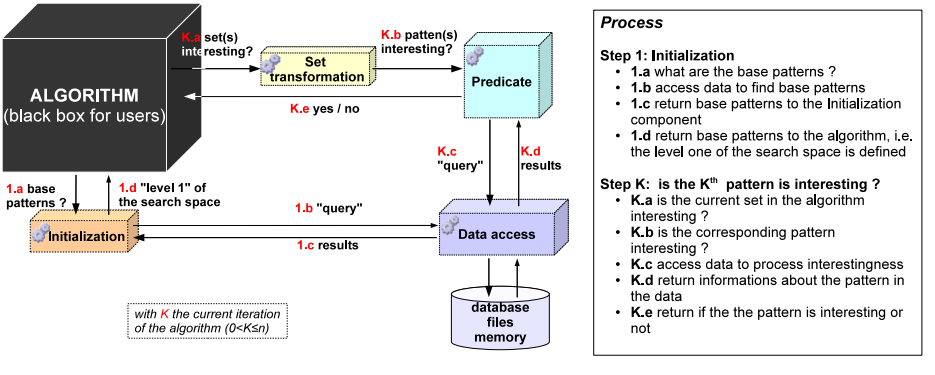

Fig. 1. *iZi* "workflow"

predicates. Consequently, algorithms and "problem" components can be used with different data formats without modifications.

Figure 2 presents how the library works for the IND (INclusion Dependency) mining problem. We suppose in this example that the algorithm used is the levelwise strategy.

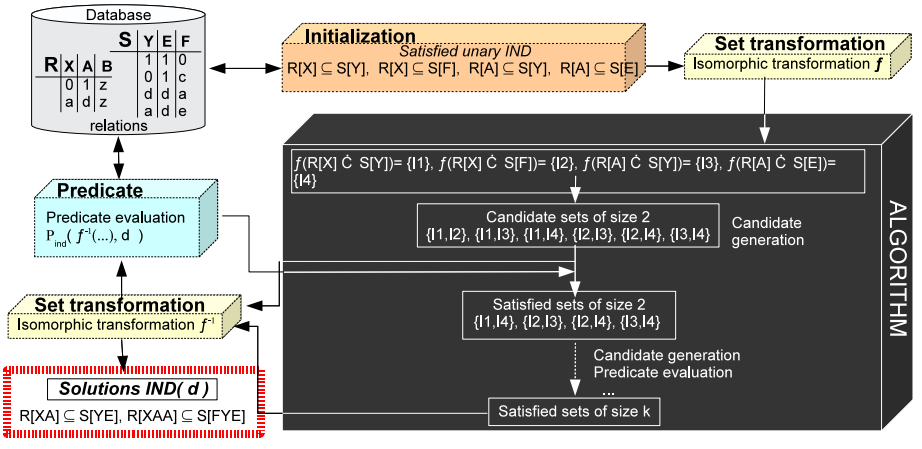

Fig. 2. IND mining example

Data structures. Since internally each algorithm only manipulates sets, we use a data structure based on prefix-tree (or trie) specially devoted to this purpose [25]. For example, Figure 3 represents the prefix-tree data structure associated to the set $\{\{A, C\}, \{A, D, F\}, \{A, D, G\}, \{A, E\}, \{D, E\}, \{E, F, G\}, \{E, G\}\}$.

They have not only a power of compression by factorizing common prefix in a set collection, but are also very efficient for candidate generation. Moreover, prefix-trees are well adapted for inclusion and intersection tests, which are basic operations when considering sets. Of course, as for algorithms, one can imagine

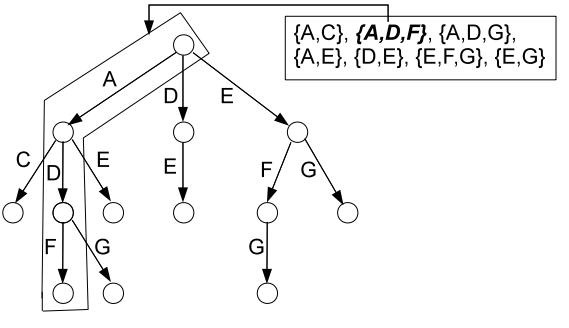

Fig. 3. Example of trie data structure

to extend our library with alternative structures for sets, like bitmaps. The use of indexes is also an important issue but not considered yet.

Note that template trie container and iterator are provided with the library. Actually, two trie implementations are available with the library: one optimized for data compression and one optimized for data search. Their implementation have been mapped on the implementations of the standard C++ STL (Standard Template Library) containers. This class also contains an implementation of an incremental algorithm, based on trie data structures, for the minimal transversals computation of an hypergraph.

Implementation issues. Figure 4 presents, from an implementation point of view, a UML model of the library. In particular, this model specifies how patterns and sets interact with the other components: patterns are used in problem specific components and sets are used internally by the algorithms. This model also points out the possibility to do predicate composition which is the case in many applications (e.g. itemset mining using conjunction of monotone constraints). For data access, this model distinguishes two cases: input data and output data. Input data is used by the predicate to test patterns and is totally independent of the algorithms. Output data is used by the algorithm to output the solutions (theory and/or positive border and/or negative border).

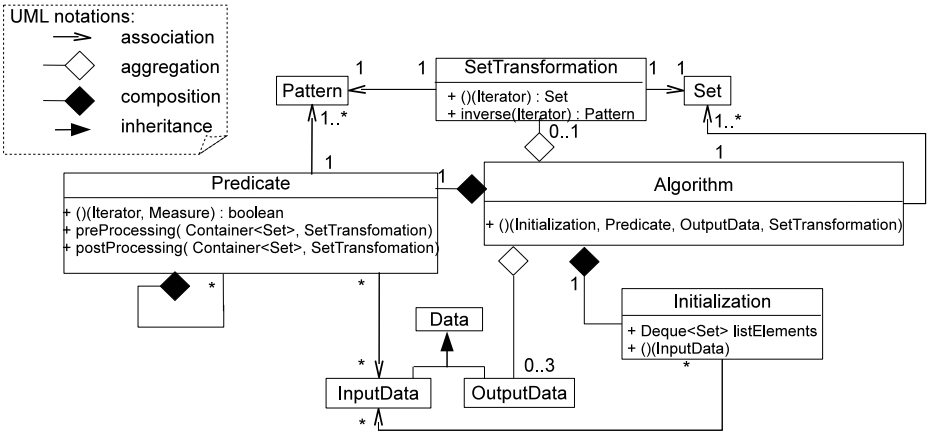

Fig. 4. iZi UML model

Moreover, thanks to this model and to the object-oriented paradigm, users can also implement algorithm and predicate variants/refinements, i.e. use inheritance to define new algorithms or predicate based on existing ones. Figure 5 presents an example of algorithm and predicate variants/refinements already implemented. In this figure, the *frequent* class represents the frequent itemset mining predicate, and the *frequent essential* class represents the predicate for a condensed representation of frequent itemsets. In the same way, the *Dualization* class represents the dualization based algorithm provided in *iZi* and the *ReverseDualization* class represents a variant of this algorithm changing the exploration strategy.

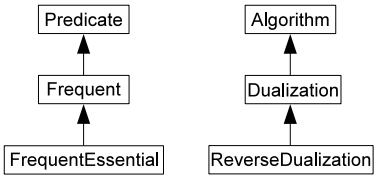

Fig. 5. Example of algorithm and predicate variants/refinements

In our context, another interesting property is method overloading which can be used to optimize some predicates sub-methods w.r.t. specific data structures. For example, the support counting method is crucial for frequent itemset mining algorithms. Using method overloading, it is for example possible to have a generic support counting method and one optimized for trie data structures. Thanks to this property, it is possible to have a good trade-off between components genericity and algorithms performances.

Finally, to solve a new problem, users only have few implementations constraints. For example, their *predicate* and *set transformation* classes have to be functors (i.e. function objects) with the same signature for the *operator()* method, and *output data* classes must only have a *push_back()* method. To facilitate these developments, abstract base classes are provided with the library as well as sample components.

4 Demonstration Scenario

From our past experience in the development of pattern mining algorithms, we note that the adaptation of existing implementations is extremely difficult. In some cases, it implies the redevelopment of most of the implementation and could take more time than developing a new program from scratch.

As shown in Table 1, many problems have been implemented in our library along with several data types and data sources components. For itemset mining, the format considered is the FIMI file format which has been defined by the FIMI workshops [2,3] to store transactional databases in a text file. This data format is widely used for this family of problems. For constraint mining in relational databases, components have also been developed to access data in files

Table 1. Problems and data sources experimented with *iZi*

Data type	Problem	File (format)	DBMS
tabular	inclusion dependencies [14]	CVS FDEP [26]	MySQL
tabular	keys [10]	CVS FDEP [26]	MySQL
binary	frequent itemsets [11]	FIMI [2,3]	
binary	frequent essential itemsets [17]	FIMI [2,3]	
set	sub-problems of query rewriting [27]	specific	

(CSV format of *Excel* and *FDEP* format defined by [26]) and in the *MySQL* DBMS. For query rewriting in integration systems, we have studied two combinatorial sub-problems (i.e. two different predicates). The data access and output components processed specific file formats.

As indication, the use of our library to implement a program for the key mining problem in a relational database has been done in less than one working day. Based on these components, the following scenario will show the simplicity of solving a new problem using *iZi*.

Let us suppose that a user wants to solve a new pattern mining problem using *iZi*: for example inclusion dependencies mining. **First, the user has to check some theoretical aspects:**

1. **Is the problem an "enumeration problem under constraint"?**
   An *inclusion dependency* (IND) is an expression of the form $R[X] \subseteq S[Y]$, where R and S are relation schemas of a same database schema D. Such a constraint ensures that, for any relations r and s over R and S, any X-value into r is a Y-value into s . If Y is a key in S, then X is a *foreign key* in R. Inclusion dependency discovery is a way to discover foreign keys and other more general semantic constraints. It can be stated as follows:

   IND mining problem *(referred to as IND): Let d be a database over a schema D, extract (maximal) inclusion dependencies satisfied in d.* Let $IND(d) = \{R[X] \subseteq S[Y] \mid R, S \in D, R[X] \subseteq S[Y]$ is satisfied in $d\}$.

2. **What are the patterns, the partial order and the predicate? Is the predicate (anti-)monotone?**
   a. The pattern language $\mathcal{L}_{ind}$ is composed of all the IND expressions that can be expressed into a database schema.
   b. The predicate $P_{ind}(R[X] \subseteq S[Y], d)$ is true, if $\pi_X(r) \subseteq \pi_Y(s)$ (with π the projection operator of the relational algebra).
   c. From a well known inference rule for INDs [28], if an IND is satisfied, then any IND obtained by applying the same projection on the left and right-hand sides is satisfied. As an example, if $R[ABC] \subseteq S[EFG]$ is satisfied, then the following INDs (not exhaustive) are satisfied: $R[A] \subseteq S[E], R[B] \subseteq S[F], R[C] \subseteq S[G], R[BC] \subseteq S[FG], R[CB] \subseteq S[GF]$... Consequently, the partial order is defined by projections over INDs.

   Considering the partial order defined by projections over INDs, the predicate $P_{ind}(R[X] \subseteq S[Y], d)$ is anti-monotone (see [14] for the proof).

   The IND mining problem can be reformulated as follows [14]:

$$IND(d) = \mathcal{B}d^{+}(Th(\mathcal{L}_{ind}, d, P_{ind}))$$

3. **What is the function f that guarantees the isomorphism with a boolean lattice ?** (see [14] for more details on this point)
   The search space of IND is not a boolean lattice at all. As an example, consider the two INDs $R[X] \subseteq S[Y]$ and $R[X'] \subseteq T[Z]$. They do not have an

upper bound (i.e. a common specialization), such as $R[XX'] \subseteq S[YZ]$ for $R[X] \subseteq S[Y]$ and $R[X'] \subseteq S[Z]$, since they don't consider the same relations. To solve this, we have to consider the subproblems $IND(r, s)$ for each pair of relations {r,s} in d. However, the search spaces of these subproblems are still not boolean lattices. For example $R[A] \subseteq S[E]$ and $R[B] \subseteq S[F]$ have two possible least upper bound, which are $R[AB] \subseteq S[EF]$ and $R[BA] \subseteq S[FE]$. In order to fit each subproblem into a boolean lattice context, we define the function f which transforms any IND into the set of all unary INDs (i.e. INDs between single attributes) obtained by projection. Thus, $f(R[AB] \subseteq S[EF]) = \{R[A] \subseteq S[E]; R[B] \subseteq S[F]\}$. Now, the desirable property is that f must be a bijection between IND search space and the powerset of all unary INDs. However:

- f is not a one-to-one function, since $f(R[AB] \subseteq S[EF]) = f(R[BA] \subseteq S[FE])$. The solution is to restrict the IND search space to INDs with a sorted left-hand side. Thanks to the "permutation inference rule", this restriction leads to no loss of knowledge [28].
- f is not surjective, since e.g. $f^{-1}(\{R[A] \subseteq S[E]; R[B] \subseteq T[G]\})$ cannot be defined. To cope with this problem, one needs to mine INDs from pairs of relations one by one. Moreover, duplicate attributes must be allowed in IND definition as it is done in [29].

With the above restrictions, one can easily verify that f is an isomorphism between IND search space and the powerset of unary INDs.
The search space C of INDs over (R, S) is defined by: $C(R, S) = \{R[< A_1...A_n >] \subseteq S[< B_1...B_n >] \mid \forall 1 \leq i < j \leq n, (A_i < A_j) \vee (A_i = A_j \wedge B_i < B_j)\}$ where $n = min(|R|, |S|)$.

Let I_1 be the set of unary INDs over R. The function $f : C \longrightarrow \mathcal{P}(I_1)$ is defined by: $f(i) = \{j \in I_1 \mid j \preceq i\}$. The function $f : C \longrightarrow \mathcal{P}(I_1)$ is bijective and its inverse function f^{-1} is computable. Moreover, given i and j two IND expressions of C, $i \preceq j \Leftrightarrow f(i) \subseteq f(j)$.

Consequently, f is an isomorphism from $(C, \preceq)$ to $(\mathcal{P}(I_1), \subseteq)$, that is to say that the search space of INDs is representable as sets.

Let $\mathcal{L}_{ind} = C(R, S)$, the search space of $IND(r, s)$ is isomorphic to a boolean lattice, and the function f is $f : C \longrightarrow \mathcal{P}(I_1)$ (see [14] for the proof).

This example is a typical case: the problem becomes representable as sets by restricting the language to be used to define the search space (without any loss of knowledge thanks to patterns properties).

Secondly, the user has to develop (or adapt) several components:

4. **the *data access* component.** Suppose in this scenario that the data is stored in a MySQL database, and that a component for this data source is already implemented.
5. **the *initialization* component**, which will initialize unary INDs using databases schemas.

6. **the *set transformation* component**, which will transform an IND in a set of unary INDs (and inversely).
7. **the *predicate* component**, which will test if the IND in parameter is satisfied in the database (using the *data access* component).

Note that as shown by their source code, all these components are simple with few lines of code. Moreover, if some of them are already developed, the user can directly reuse them without modifications. See as an example Figure 6 for an implementation of the predicate component for IND mining in a MySQL database.

```
template< class DBMS>
class SatisfiedIND: public Predicate
{
  protected:
     //! Pointer on the dbms
     DBMS * mydbms;

  public:
     //! Constructor
     /*!
       \param inDbms pointer on the DBMS and the db studied
     */
     SatisfiedIND_DBMS( DBMS * inDbms )
     {
        mydbms = inDbms ;

        if( mydbms->get_relation(1) && mydbms->get_relation(2) )
        {
          // store a parameretrized query  to test inclusion
          // dependencies between the two input relations
          string query = "select count(*) from "
             +  mydbms->get_relation(1)->name
             + " where ( var1 ) not in( SELECT distinct var2 FROM "
             +  mydbms->get_relation(2)->name +" )" ;
          mydbms->store_query( (char *)(query.c_str()) );
        }
     }

      //! Operator that test if an inclusion dependency is satisfied
      //!  or not in two relations
      /*!
        \param itCand iterator on the pattern to test
        \param mesCand measure associated wih the pattern and
processed in the predicate
      */
      template< class Iterator, class Measure >
      bool operator() ( Iterator  itCand, Measure & mesCand );
};
```

```
#include "SatisfiedIND.h"

//! Operator that test if an inclusion dependency is satisfied or
//! not in two relations
template< class DBMS>
template< class Iterator, class Measure >

bool SatisfiedIND<DBMS>::operator()(Iterator  itCand,
Measure & mesCand)
{
    //search the attributes in the left part of the IND
    string left= itCand->left[0] ;
    for( int i = 1; i < itCand->left.size(); i++)
              left+=","+itCand->left[i] ;

    //search the attributes in the right part of the IND
    string right= itCand->right[0] ;
    for( int i = 1; i < itCand->right.size(); i++)
              right+=","+itCand->right[i] ;

    left= " "+left+" ";
    right=" "+right+" ";

    //replace the variables by the attributes of the IND
    mydbms->replace_in_query(" var1 ", (char *)(left.c_str()) );
    mydbms->replace_in_query(" var2 ", (char *)(right.c_str()) );

    //execute the query
    string nb_notin = mydbms->exec_query();

    //re initialize the variables for the next predicate test
    mydbms->replace_in_query( (char *)(left.c_str())," var1 ");
    mydbms->replace_in_query( (char *)(right.c_str())," var2 ");

    // test if  values of the first projection are in the second one
    if( nb_notin == "0" )   return true ;  // the IND is satisfied
    else  return false ;
}
```

Fig. 6. Example of IND predicate implementation

From this moment, the user can directly use any algorithm provided with *iZi* in his/her source codes, compile and execute the algorithm to find all satisfied INDs.

5 Experimentations

Our motivation here is to show that our generic library has good performance characteristics w.r.t. specialized and optimized implementations.

We present some experimental results for frequent itemset mining, since it is the original application domain of the algorithms we used and the only common problem with DMTL. Moreover, many resources (algorithms implementations,

datasets, benchmarks...) are available on Internet [30] for frequent itemset mining. For other problems such as key mining, even if algorithms implementations are sometimes available, it is difficult to have access to the datasets. As an example, we plan to compare *iZi* with the proposal in [31] for key mining. Unfortunately, neither their implementation, nor their datasets have been made available in time.

Implementations for frequent itemset mining are very optimized, specialized, and consequently very competitive. The best performing ones are often the results of many years of research and development. In this context, our experimentations aims at proving that our generic algorithms implementations behave well compared to specialized ones. Moreover, we compare *iZi* to the DTML library, which is also optimized for frequent pattern mining.

Experiments have been done on some FIMI datasets [2,3] on a pentium 4.3GHz processor, with 1 Go of memory. The operating system was Ubuntu Linux 6.06 and we used gcc 4.0.3 for the compilation. We compared our *Apriori* generic implementation to two others devoted implementations: one by B. Goethals [32] and one by C. Borgelt [33]. The first one is a quite natural version, while the second one is, to our knowledge, the best existing *Apriori* implementation, developed in *C* and strongly optimized. Then, we compared "*iZi Apriori*" and "*iZi* dualization based algorithm" to the eclat implementation provided with DMTL.

In Table 2, three *Apriori* implementations are compared w.r.t. their execution times (in milliseconds) for datasets *Connect* (129 items and 67 557 transactions), *Pumsb* (2 113 items and 49 046 transactions) and *Pumsb∗* (2 088 items and 49 046 transactions). One can observe that our generic version has good performance with respect to other implementations. These results are very encouraging, in regards of the simplicity to obtain an operational program.

In Table 3, iZi and DMTL are compared w.r.t. their execution times (in milliseconds) for the same datasets. Even if DMTL is optimized and specialized for the frequent predicate, algorithm implementations of *iZi* have good performances w.r.t. eclat DMTL . The difference between the two libraries is mainly due to the algorithm used during the experimentations. This could be easily confirmed by looking at the performances of *Apriori*, *Eclat* and *dualization based* algorithms observed during FIMI benchmarks [30].

Table 2. Comparison of three *Apriori* implementations (in milliseconds)

	Apriori iZi	Apriori Goethals	Apriori Borgelt
Connect 90%	23 000	133 000	1 000
Pumsb 90%	18 000	14 000	1 000
Pumsb* 60%	2 000	4 000	1 000

Table 3. Comparison of iZi and DMTL implementations (in milliseconds)

	Apriori iZi	ABS iZi	eclat DMTL
Connect 90%	23 000	8 000	17 000
Pumsb 90%	18 000	18 000	8 000
Pumsb* 60%	2 000	2 000	5 000

6 Discussion and Perspectives

In this paper, we have considered a classical problem in data mining: the discovery of interesting patterns for problems known to be *representable as sets*, i.e. isomorphic to a boolean lattice. In addition to the interest of our library to solve new problems, *iZi* is also very interesting for algorithm benchmarking. Indeed, thanks to the modularity of *iZi*, it is possible to test several data representations (e.g. prefix tree or bitmap) or several predicates, with the same algorithm source code. Thus, it enables a fair comparison and test of new strategies. *iZi* has also been used for educational purpose. Using the library, students can better understand where the key issues are in pattern mining. For example, for frequent itemset mining, they often underestimate the importance of support counting in the algorithm performance. By allowing to easily change the strategy used for support counting, *iZi* enables to better understand how this affects algorithms performances.

To our knowledge, this is the first contribution trying to bridge the gap between fundamental studies in data mining around inductive databases [10,21,34] and practical aspects of pattern mining discovery. Our work concerns plenty of applications from different areas such as databases, data mining, or machine learning.

Many perspectives exist for this work. First, we may try to integrate the notion of *closure* which appears under different flavors in many problems. The basic research around concept lattices [35] could be a unifying framework. Secondly, we are interested in integrating the library as a plugin for a data mining software such as Weka [6]. Analysts could directly use the algorithms to solve already implemented problems or new problems by dynamically loading their own components. Finally, a natural perspective of this work is to develop a declarative version for such mining problems using query optimization techniques developed in databases [36].

References

1. Goethals, B., Nijssen, S., Zaki, M.J.: Open source data mining: workshop report. SIGKDD Explorations 7, 143–144 (2005)
2. Bayardo Jr., R.J., Zaki, M.J.(eds.): FIMI 2003, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19. CEUR Workshop Proceedings, vol. 90, CEUR-WS.org (2003)
3. Bayardo Jr., R.J., Goethals, B., Zaki, M.J. (eds.): FIMI 2004, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1. CEUR Workshop Proceedings, vol. 126, CEUR-WS.org (2004)
4. Han, J.: Data Mining Group: IlliMine project. University of Illinois Urbana-Champaign Database and Information Systems Laboratory (2005), `http://illimine.cs.uiuc.edu/`
5. Hasan, M., Chaoji, V., Salem, S., Parimi, N., Zaki, M.: DMTL: A generic data mining template library. In: Workshop on Library-Centric Software Design (LCSD 2005), at OOPSLA 2005 conference, San Diego, California (2005)

6. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)
7. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: Yale: rapid prototyping for complex data mining tasks. In: Eliassi-Rad, T., Ungar, L.H., Craven, M., Gunopulos, D. (eds.) KDD, pp. 935–940. ACM, New York (2006)
8. Bonchi, F., Giannotti, F., Lucchese, C., Orlando, S., Perego, R., Trasarti, R.: Conquest: a constraint-based querying system for exploratory pattern discovery. In: Liu, L., Reuter, A., Whang, K.Y., Zhang, J. (eds.) ICDE, p. 159. IEEE Computer Society, Los Alamitos (2006)
9. Blockeel, H., Calders, T., Fromont, É., Goethals, B., Prado, A., Robardet, C.: An inductive database prototype based on virtual mining views. In: Li, Y., Liu, B., Sarawagi, S. (eds.) KDD, pp. 1061–1064. ACM, New York (2008)
10. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Min. Knowl. Discov. 1, 241–258 (1997)
11. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Buneman, P., Jajodia, S. (eds.) SIGMOD Conference, pp. 207–216. ACM Press, New York (1993)
12. Mannila, H., Toivonen, H.: Multiple uses of frequent sets and condensed representations (extended abstract). In: KDD, pp. 189–194 (1996)
13. Koeller, A., Rundensteiner, E.A.: Heuristic strategies for inclusion dependency discovery. In: Meersman, R., Tari, Z. (eds.) OTM 2004, Part II. LNCS, vol. 3291, pp. 891–908. Springer, Heidelberg (2004)
14. De Marchi, F., Flouvat, F., Petit, J.M.: Adaptive strategies for mining the positive border of interesting patterns: Application to inclusion dependencies in databases. In: Boulicaut, J.-F., De Raedt, L., Mannila, H. (eds.) Constraint-Based Mining and Inductive Databases. LNCS (LNAI), vol. 3848, pp. 81–101. Springer, Heidelberg (2006)
15. Angluin, D.: Queries and concept learning. Machine Learning 2, 319–342 (1987)
16. Li, Z., Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Wermelinger, M., Gall, H. (eds.) ESEC/SIGSOFT FSE, pp. 306–315. ACM, New York (2005)
17. Casali, A., Cicchetti, R., Lakhal, L.: Essential patterns: A perfect cover of frequent patterns. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 428–437. Springer, Heidelberg (2005)
18. Gschwind, T.: Pstl-a c++ persistent standard template library. In: COOTS, pp. 147–158. USENIX (2001)
19. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: KDD, pp. 283–286 (1997)
20. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) VLDB, pp. 487–499. Morgan Kaufmann, San Francisco (1994)
21. Gunopulos, D., Khardon, R., Mannila, H., Saluja, S., Toivonen, H., Sharm, R.S.: Discovering all most specific sentences. ACM Trans. Database Syst. 28, 140–174 (2003)
22. Flouvat, F., De Marchi, F., Petit, J.M.: ABS: Adaptive Borders Search of frequent itemsets. In: [3]
23. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Chen, W., Naughton, J.F., Bernstein, P.A. (eds.) SIGMOD Conference, pp. 1–12. ACM, New York (2000)
24. Uno, T., Asai, T., Uchida, Y., Arimura, H.: Lcm: An efficient algorithm for enumerating frequent closed item sets. In: [2]

25. Bodon, F.: Surprising results of trie-based fim algorithms. In: [3]
26. Flach, P.A., Savnik, I.: Database dependency discovery: A machine learning approach. AI Commun. 12, 139–160 (1999)
27. Jaudoin, H., Flouvat, F., Petit, J.M., Toumani, F.: Towards a scalable query rewriting algorithm in presence of value constraints. Journal on Data Semantics 12, 37–65 (2009)
28. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
29. Mitchell, J.C.: The implication problem for functional and inclusion dependencies. Information and Control 56, 154–173 (1983)
30. Goethals, B.: Frequent itemset mining implementations repository, `http://fimi.cs.helsinki.fi/`
31. Sismanis, Y., Brown, P., Haas, P.J., Reinwald, B.: Gordian: Efficient and scalable discovery of composite keys. In: Dayal, U., Whang, K.Y., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y.K. (eds.) VLDB, pp. 691–702. ACM, New York (2006)
32. Goethals, B.: Apriori implementation. University of Antwerp, `http://www.adrem.ua.ac.be/~goethals/`
33. Borgelt, C.: Recursion pruning for the apriori algorithm. In: [3]
34. Boulicaut, J.F., Klemettinen, M., Mannila, H.: Modeling kdd processes within the inductive database framework. In: Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 1999. LNCS, vol. 1676, pp. 293–302. Springer, Heidelberg (1999)
35. Ganter, B., Wille, R.: Formal Concept Analysis. Springer, Heidelberg (1999)
36. Chaudhuri, S.: Data mining and database systems: Where is the intersection? IEEE Data Eng. Bull. 21, 4–8 (1998)