The Discovery of Frequent Patterns with Logic and Constraint Programming

NITTAYA KERDPRASOP and KITTISAK KERDPRASOP
Data Engineering Research Unit,
School of Computer Engineering, Suranaree University of Technology,
111 University Avenue, Nakhon Ratchasima 30000
THAILAND
nittaya@sut.ac.th, kittisakThailand@gmail.com

Abstract: - The basic goal of data mining is to discover patterns occurring in the databases, such as associations, classification models, sequential patterns, and so on. In this paper we focus on the problem of frequent pattern discovery, which is the process of searching for patterns such as sets of features or items that appear in data frequently. Such frequent patterns can reveal associations, correlations, and many other interesting relationships hidden in a database. Most of frequent pattern mining systems in the market are too generic and become inefficient when set of patterns is large and the frequent patterns are very long. A new trend in data mining is a scalable method that uses constraints to guide the system in its search for interesting patterns. Our main research objective is the development of constraint-based mining methodology and this paper presents the preliminary results of our study and prototype development. We present the implementation of frequent pattern mining system based on declarative programming paradigm using logic programming and constraint logic programming. The comparative performance studies on speed and memory usage of logic versus constraint programming are also reported in the paper.

Key-Words: - Frequent pattern discovery, Association mining, Logic programming, Prolog, Constraint logic programming, ECLiPSe.

1 Introduction
Data mining is the nontrivial process of extracting implicit, previously unknown and potentially useful information from data [7]. Traditionally, the analysis of data employs machine learning and statistical techniques to find hidden patterns and correlations in the data. Data mining methods are broadly defined depending on the specific research objective and involve different classes of mining tasks including regression, classification, clustering, identifying meaningful associations between data attributes. The later mining task refers to association mining, or market basket analysis [8] in the retail business domain, which is the main focus of our research.

Association mining is a popular method for discovering relations between features or variables in large databases and presenting the discovered results as a set of if-then rules, such as [milk, bread] => {butter} to indicate that if a customer buys milk and bread, he or she is more like to buy butter as well. Association rule generation process is composed of two major phases: frequent itemset mining and rule generation. Frequent itemset mining is to find all items or features that are frequently occurred together. It is an import phase of association mining because it is a difficult task to search all possible itemsets. We thus pay attention to the design and implementation of frequent itemset discovery by applying the concept of constraint logic programming. Our implementation is based on the ECLiPSe constraint system.

The organization of this paper is as follows. The problem of frequent pattern discovery is defined in Section 2. Then the logic-based and constraint programming implementation of frequent pattern discovery is explained and demonstrated in Section 3. Experimentation and results are presented in Section 4. Finally, Section 5 concludes the paper with discussion of our future research direction.

2 The Discovery of Frequent Patterns
Frequent pattern discovery refers to an attempt to find regularities or common relations frequently occurred in a database. A set of market basket transactions [8] is a common database used in frequent pattern analysis. A transactional database is in a table format (as shown in Figure 1).
Each row is a transaction, identified by a transaction identifier \( TID \). A transaction contains a set \( I \) of items bought by a customer. The problem of frequent discovery is defined as the search for recurring relationships or correlations between items in a database [1], [2]. Let \( I = \{ i_1, i_2, i_3, ..., i_m \} \) be a set of \( m \) items and \( DB = \{ T_1, T_2, T_3, ..., T_n \} \) be a transactional database of \( n \) transactions and each transaction contains items in \( I \). A pattern is a set of items that occur in a transaction. The number of items in a pattern is called the length of the pattern. The pattern such as \{ Beer, Cereal, Diaper \} is thus a pattern of length three or a 3-item pattern.

To search for all valid patterns of length 1 up to \( m \) in large database is computationally expensive. It can be seen in Figure 2 that a transactional database containing different combinations of five items \( I = \{ \text{Beer(B)}, \text{Cereal(C)}, \text{Diaper(D)}, \text{Milk (K)}, \text{Egg(E)} \} \) can generate a search space of \( 2^5 - 1 = 31 \) possible patterns (excluding the empty set, or 0-item). Thus, for a set \( I \) of \( m \) different items, the search space for all distinct patterns can be as huge as \( 2^m - 1 \).

To reduce the size of the search space, the \textit{support} measurement has been introduced [1],[2]. The support \( s(P) \) of a pattern \( P \) is defined as a number of transactions in \( DB \) containing \( P \). Thus, \( s(P) = |\{ T | T \in DB, \ P \subseteq T \}| \).

A pattern \( P \) is called \textit{frequent pattern} if the support value of \( P \) is not less than a predefined minimum support threshold \( \text{minS} \). It is the \( \text{minS} \) constraint that helps reducing the computational complexity of frequent pattern generation. Suppose we specify \( \text{minS} = \frac{2}{5} = 0.4 \) on a set of transactions shown in Figure 1, then the pattern \{Egg\} is infrequent and so do all the set of patterns having Egg(E) as their member, that is, \{E, BE, CE, DE, EM, BCE, BDE, BEM, CDE, CEM, DEM, BCDE, BCEM, BDEM, CDEM, BCDEM\}. All the infrequent patterns can be pruned to reduce the search space. This pruning strategy is called an anti-monotone property and has been applied as a basis for searching frequent patterns in the well known algorithm Apriori [1], [2].

The Apriori-like algorithms find all frequent itemsets by generating supersets of previously found frequent itemsets. This generate-and-test method is computationally expensive. Han et al [9] proposed a different divide-and-conquer approach based on the prefix-tree structure that consumes less memory space. Toivonen [13] employed sampling techniques to deal with frequent pattern mining from large databases. Zaki et al [14] tackled the problem by means of parallel computation. Some researchers [4], [6], [10], [11], [12] consider the issue of search space reduction through the concept of constraints. Our research is in the same direction. We consider the problem of mining frequent patterns within a setting of constraint logic programming using the ECLiPSe system [3]. Constraints can play an important role in improving the performance of mining algorithm. The problem of constraint-based pattern mining can therefore be formulated as the discovery of all patterns in a given dataset that satisfy the specified constraints. In the next section, we present work in progress of problem modelling and solving with respect to the frequent pattern discovery in a transactional database.

### 3 Logic and Constraint Programming Implementation

A problem of frequent pattern discovery is to determine how often a candidate pattern occurs. A pattern is a set of items co-occurrence across a database. Given a candidate pattern, the task of pattern matching is then applied to search for its frequency looking for the patterns that are frequent enough. The outcome of this search is frequent patterns that suggest strong co-occurrence relations between items in the dataset.

#### 3.1 Logic Program

The search for patterns of interest can be efficiently programmed using the logic-based language such as Prolog. In Prolog, the feature of pattern matching
can be defined through the use of arguments. For example, the following program [5] demonstrates the length function (in Prolog it is called predicate instead of function) to count the number of elements in a list. Last argument is normally a placeholder for an output. Variables in Prolog start with an uppercase letter such as Xs, L, X. Each statement in Prolog is called a clause and every statement ends with period. The symbol ‘,’ is a logical connective AND. The symbol ‘:-’ is an implication and it may be pronounced as ‘if’. Thus the last statement of length predicate may be read as “length of the list (X|Xs) is L is length of the list (Xs) is M and L is M+1.”

length([], 0).  -- pattern 1: length of an empty list is 0
length([X|Xs], L) :- length(Xs, M), L is M+1.  -- pattern 2: length of a list whose first element is X and remainder is Xs is 1+ length of xs

In declarative language such as Prolog, programs are sets of definitions and recursion is the main control structure of the program computation. In imperative or procedural languages, such as C and Java, programs are sequences of instructions and loops are the main control structure. A logic programming language like Prolog is a declarative language in which programs are sets of predicate definitions. Predicates are true or false when applied to an object or set of objects. A predicate definition has a dual meaning: (1) it describes what is the case, and (2) it describes the way to compute something.

Declarative languages are mathematically sound. It is easy to prove that a declarative program meets its specification, which is an important requirement in software industry. Declarative style makes a program better engineered, that is, easier to debug, easier to maintain and modify, and easier for other programmers to understand. The following is the coding of frequent pattern mining program in Prolog language with a simple transactional database as appeared in Section 2.

```prolog
% % FrequentPattern.pl
% call :- r1.
% ?- r2(X).
% ?- clear.
tid(5).  % all transactions=5
s(3).  % support=3 or 0.6
ni([b,c,d,e,m]).  % all items
item([c, m]).
item([b, c, d, e]).
item([b, d, m]).
item([b, c, d, m]).
item([d, m]).
r1 :- ni(X), cL1(X).
r2(X) :- cC2(X).
r3(X) :- cC3(X).
clear :- retractall(l1(_)), retractall(l2(_)), retractall(l3(_)).
cL1([]).  % Create L1
clL1([H | T]) :- findall(X, f([H], X), L),
length(L, Len),
Len >= 2 , ! ,
write('ok-head-H-len='Len),
nl ,cL1(T), assert(l1([H|X], Len))
; cL1(T).

% Create C2, L2
cC2(X) :- l1((X|Xs), l2(X2,...)),
X \==X2, write(X-X2),
union(X, X2, Res),
assert(c2(Res)) ,
retract(l1([X|Xs])) , nl.
crC2(L) :- findall(X, c2(X), L).
cL2([]).
cL2([H | T]) :- findall(X, f([H], X), L),
length(L, Len),
Len >= 2 , !,
write('ok-head-H-len='Len),
nl ,cL2(T), assert(l2([H|X], Len))
; cL2(T).
cC3(X) :- l2((X, L), l2(X2,...)),
X \==X2, write(X-X2),
union(X, X2, Res),
assert(c3(Res)) ,
retract(l2([X, L])) , nl.
crC3(L) :- findall(X, c3(X), L).
cL3([]).
cL3([H | T]) :- findall(X, f([H], X), L),
length(L, Len),
Len >= 2 , !,
write('ok-head-H-len='Len),
nl ,cL3(T), assert(l3([H, L]))
; cL3(T).
f(H, X) :- item(X), subset(H, X).
```

3.2 Constraint Logic Program

Constraint logic programming is a declarative programming style that combines the features of logic programming and constraint propagation to solve combinatorial and optimization problems. A constraint logic program is an extension of logic program by including constraints in the body of the clauses. Common structure of a constraint program is consisted of the part to define variables and constraints on variables and the part to search for a valid value on each variable. This is the style of constraint-and-search. Figure 3 demonstrates the different between logic program and constraint program on the problem of map coloring [3].
% Map coloring problem
% Prolog style:
% Generate-and-test
colour(red).
colour(green).
colour(blue).
colour(yellow).
colour_LP([A,B,C,D]) :-
colour(A),
colour(B),
colour(C),
colour(D),
A \= B,
A \= C,
A \= D,
B \= C,
B \= D,
C \= D.

% Map coloring problem
% CLP style:
% Constrain-and-search
:- lib(fd).
colour_CLP([A,B,C,D]) :-
colour(A),
colour(B),
colour(C),
colour(D),
A \= B,
A \= C,
A \= D,
B \= C,
B \= D,
C \= D.

Figure 3. Logic program versus constraint logic program.

The following implementation is the coding of frequent pattern mining in constraint logic programming using the ECLiPSe system (http://www.eclipseclp.org). The data set is transactional database containing itemset I as appeared in Section 2. The constraint problem can be formulated as: given a transactional database D and a minimum support threshold \( \sigma \), find all frequent itemsets \( FS \) such that \( FS = \{ X \subseteq \mathcal{I} | \text{support}(X) \geq \sigma \} \). A screenshot of ECLiPSe system when a minimum support has been set to be 0.3 is shown in Figure 4.

% Frequent pattern discovery with constraint
:-lib(listut).
:-dynamic(c/2).
item([b, c, d, e, m]).
t([c, m]).
t([b, d, m]).
t([b, c, d, m]).
t([d, m]).
solve(Sigma) :-
    retract_all(c(_,_)),
    findall(X, t(X), AllTrans),
    length(AllTrans, NoTrans),
    MinSup is Sigma*NoTrans,
    item(IT),
    % scan in all transactions
    (foreach(Y, IT), param(MinSup)
        do findall(Y, (t(T), subset(Y, T)), Res),
            length(Res, Len),
            (Len >= MinSup -> writeln(Y-Len),
                assert(c(KK,Y1)) ; true)
    ),
    % end solve
    myunion([X,Y], Z) :- union(X,Y,Z).

Figure 4. Screenshot of ECLiPSe constraint system.

4 Experimentation
We comparatively study the performance of our implementations of frequent pattern discovery using Prolog and ECLiPSe constraint logic programming. All experimentations have been performed on a 796 MHz AMD Athlon notebook with 512 MB RAM and 40 GB HD. We select four datasets from the UCI Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.htm) to test the speed and memory usage of the programs. The details of selected datasets are summarized in Table 1.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>File size</th>
<th># Transactions</th>
<th># Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote</td>
<td>13.2 KB</td>
<td>300</td>
<td>17</td>
</tr>
<tr>
<td>Chess</td>
<td>237 KB</td>
<td>2,130</td>
<td>37</td>
</tr>
<tr>
<td>DNA</td>
<td>252 KB</td>
<td>2,000</td>
<td>61</td>
</tr>
<tr>
<td>Mushroom</td>
<td>916 KB</td>
<td>5,416</td>
<td>23</td>
</tr>
</tbody>
</table>

The frequent pattern discovery programs have been tested on each dataset with various minS values, ranging from 0.005 to 0.5. Performance comparisons of logic programming (LP) and constraint logic programming (CLP) in terms of speed and memory usage on four datasets are shown in Figures 5 and 6, respectively. It can be noticed from the experimental results that on both speed and memory usage comparison CLP performs better than LP. However, the degree of difference is insignificant.

5 Conclusion
Frequent pattern discovery is one major problem in the areas of data mining and business intelligence. The problem concerns finding frequent patterns hidden in a large database. Finding such frequent patterns has become an important task because it reveals associations, correlations, and many other interesting relations among items in the databases. We suggest that the problem of frequent pattern discovery can be efficiently and concisely implemented with high-level declarative language such as Prolog. Coding in declarative style takes less effort because pattern matching is a fundamental feature supported by most logic-based languages. The implementation of Apriori algorithm using Prolog confirms our hypothesis about conciseness of the program. We also extend our study by implementing the algorithm with constraint logic programming paradigm using the ECLiPSe system. The performance studies also support our intuition on efficiency because our constraint-based implementation is slightly more efficient than the conventional logic programming implementation in terms of speed and memory usage.

This preliminary study supports our belief regarding constraint-based declarative programming paradigm towards frequent pattern discovery. We focus our future research on the design of constraint formulating and processing to optimize the speed and storage requirement. We also consider the extension of the algorithm in the course of concurrency to improve its performance.

Figure 5. Speed comparison of logic program versus constraint logic program
Acknowledgment
This work has been supported by grants from the National Research Council of Thailand (NRCT) and Suranaree University of Technology via the funding of Data Engineering Research Unit.

References: