Frequent Pattern Discovery with Constraint Logic Programming

Nittaya Kerdprasop and Kittisak Kerdprasop

***Abstract*—**Constraint logic programming is a declarative programming style combining the features of logic programming and constraint propagation to solve combinatorial and optimization problems such as resource allocation, scheduling, and routing. We consider the problem of mining frequent patterns within a setting of constraint logic programming approach. Frequent patterns are patterns such as sets of features or items in transactions that appear frequently. Such patterns can reveal associations, correlations, and many other interesting relationships hidden in a dataset. Constraints can play an important role in improving the performance of mining algorithms. The problem of constraint-based pattern mining can be formulated as the discovery of all patterns in a given dataset that satisfy the specified constraints. We present implementation of problem modeling and solving with respect to pattern mining in knowledge discovery in databases.

***Keywords*—** Frequent pattern discovery, Itemset mining, Logic-based programming, Constraint programming, ECLiPSe constraint system.

I. Introduction

THE main goal of data mining is to extract hidden knowledge from data [8]. Knowledge is a valuable asset to most organizations as a substantial source to enhance understanding of data relationships and support better decisions to increase organizational competency. Automatic knowledge acquisition can be achieved through the availability of the knowledge induction component. The induced knowledge can facilitate various knowledge-related activities ranging from expert decision support, data exploration and explanation, estimation of future trends, and prediction of future outcomes based on present data. The methodology of knowledge induction is known as knowledge discovery in databases, or data mining.

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 [9] 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 [11], [12], [14], [19] and then 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 [13], [21]. 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 methodology 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.

II. Concepts and Techniques of Frequent Pattern Discovery

Frequent patterns are patterns such as sets of features or items that appear in data frequently. Frequent pattern mining focuses on the discovery of relationships or correlations between items in a dataset. A set of market basket transactions [1], [2] is a common dataset used in frequent pattern analysis. A dataset is typically in a table format. Each row is a transaction, identified by a transaction identifier or a *TID*. A transaction contains a set of items bought by a customer. A set of transactions might be organized in either an enumerated (dense), or a sparse binary vector format [6]. In either format a dataset can be processed horizontally or vertically. Figure 1 illustrates the data organization formats of a simple market basket dataset.

Manuscript received June 5, 2011: Revised version received August 2, 2011. This work was supported by grants from the National Research Council of Thailand (NRCT) and Suranaree University of Technology through the funding of Data Engineering Research Unit.

N. Kerdprasop is an associate professor and the director of Data Engineering Research Unit, School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (phone: +66-44-224-432; fax: +66-44-224-602; e-mail: nittaya@sut.ac.th).

K. Kerdprasop is with the School of Computer Engineering and Data Engineering Research Unit, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (e-mail: KittisakThailand@gmail.com).

TID	Items
1	{Cereal, Milk}
2	{Beer, Cereal, Diaper, Egg}
3	{Beer, Diaper, Milk}
4	{Beer, Cereal, Diaper, Milk}
5	{Diaper, Milk}

(a) Horizontally enumerated format

Item IDs

B	C	D	E	M
2	1	2	2	1
3	2	3		3
4	4	4		4
		5		5

TID

(b) Vertically enumerated format

Item IDs

TID	B	C	D	E	M
1	0	1	0	0	1
2	1	1	1	1	0
3	1	0	1	0	1
4	1	1	1	0	1
5	0	0	1	0	1

(c) Horizontal binary vector

Item IDs

TID	B	C	D	E	M
1	0	1	0	0	1
2	1	1	1	1	0
3	1	0	1	0	1
4	1	1	1	0	1
5	0	0	1	0	1

(d) Vertical binary vector

Fig. 1 horizontal and vertical organization schemes of transaction database

In a horizontally enumerated data organization (Figure 1(a)), each transaction contains only items positively associated with a customer purchase. It is a simplistic representation of market basket data because it ignores other information such as the quantity of purchased items or the profit of item sold. A horizontally enumerated format is sometimes referred to as a *TidLists* dataset organization.

In a vertical organization of items bought enumeration (Figure 1(b)), each column stores an ordered list of *TIDs* of the transactions that contain an item. This format of a dataset occupies that same space as the horizontally enumerated format.

Figures 1(c) and 1(d) represent a binary vector format. A value in each vector cell is 1 if the item is present in a transaction and 0 otherwise. A binary vector format is referred to as a *TidSets* dataset organization.

Recent attention has been given to the influence of data organization on the performance of the process of frequent pattern discovery. A vertical vector organization has been proven an efficient layout for the problem of frequent pattern discovery, but it suffers from the memory usage. In this paper, we study the performance of frequent pattern discovery based on the horizontal item organization.

In frequent pattern mining, we are interested in analyzing connections among items. A collection of zero or more items is called an itemset. For example, the first transaction in Figure 1 contains the itemset {Cereal, Milk}. Since this set contains two items, it is called a 2-itemset. An itemset can be an empty set, a 1-itemset, a 2-itemset, and so on. Figure 2 shows all combinations of distinct itemsets from the set of items {B, C, D, E, M}, where B = Beer, C = Cereal, D = Diaper, E = Egg, and M = Milk.

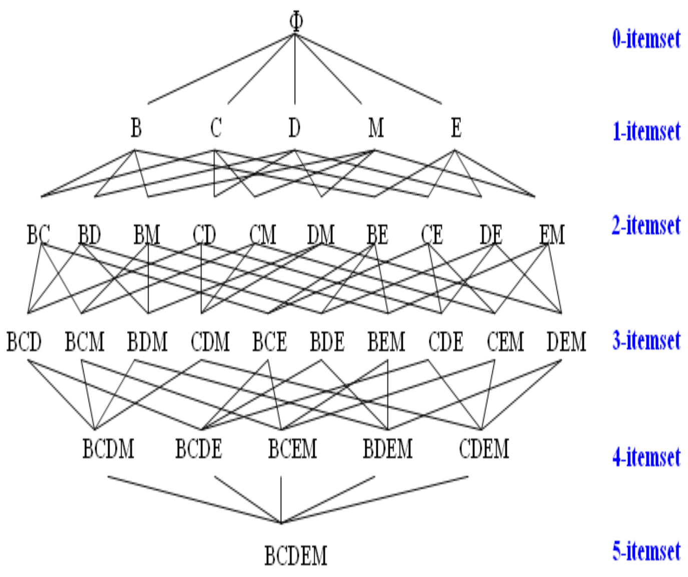


Fig. 2 a lattice of possible frequent patterns from the five distinct items

The discovery of interesting relationships hidden in large datasets is the objective of frequent pattern mining. The uncovered relationships can be represented in the form of association rules. An association rule is an inference of the form $X \rightarrow Y$, where X and Y are non-empty disjoint itemsets. To form association rules, we consider only valid itemsets. An itemset is valid if it really occurs in a transaction. For instance, from a dataset shown in Figure 1 an itemset {Egg, Milk} is invalid because none of the customers buy both eggs and milk.

The identification of all valid itemsets is computational expensive. It can be seen from Figure 2 that a dataset of I items has 2^I distinct itemsets. To reduce the search space, the measurements of *support* and *confidence* are used to constrain the mining process.

The constraint *support* forces the mining process to discover only relationships that occur frequently, while *confidence* constrains the reliability of the inference made by a rule.

The support count for an itemset Z, denoted as $\sigma(Z)$, is the number of transactions that contain a particular itemset Z. As an example, consider a dataset in Figure 1, there are three transactions (*TID* 2, 3, 4) that contain the item Beer, thus $\sigma(\text{Beer}) = 3$. Given the definition of support count, the metrics support and confidence of the association rule $X \rightarrow Y$ can be defined as follows [9].

$$\textit{Support},\ s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N},$$

where N is the number of all transactions.

$$\textit{Confidence},\ c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}.$$

Given a dataset as shown in Figure 1, an example of association rule is the statement that "customers who buy beer also buy diaper, with 60% supporting transactions and 100% confidence." An itemset is called a *frequent itemset* if its support is greater than or equal user-specified support threshold (called *minSup*).

An association rule generated from frequent itemset with the confidence greater than or equal a confidence threshold (called *minConf*) is considered a valid association rule. With the pre-specified *minSup* and *minConf* metrics, the problem of association rule discovery can be stated as follows: Given a set of transactions, find all the rules having support $\geq$ *minSup* and confidence $\geq$ *minConf*. This problem can be decomposed into two subtasks:

(1) Frequent itemset generation: find all itemsets that satisfy the *minSup* threshold.

(2) Rule generation: generate from frequent itemsets all high confidence rules.

It is the *minSup* constraint that helps reducing the computational complexity of frequent itemset generation. Suppose we specify *minSup* = 2/5 = 40% on a set of transactions shown in Figure 1; the item {Egg} is infrequent. As a result, all supersets of {Egg} are also infrequent. All infrequent itemsets can then be pruned to reduce the search space (see Figure 3).

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 detail of this algorithm is given in Figure 4.

The Apriori-like algorithms find all frequent itemsets by generating supersets of previously found frequent itemsets. This generate-and-test method is computational expensive. Han et al [10] proposed a different divide-and-conquer approach based on the prefix-tree structure that consumes less memory space. Toivonen [20] employed sampling techniques to deal with frequent pattern mining from large databases. Zaki et al [22] tackled the problem by means of parallel computation.

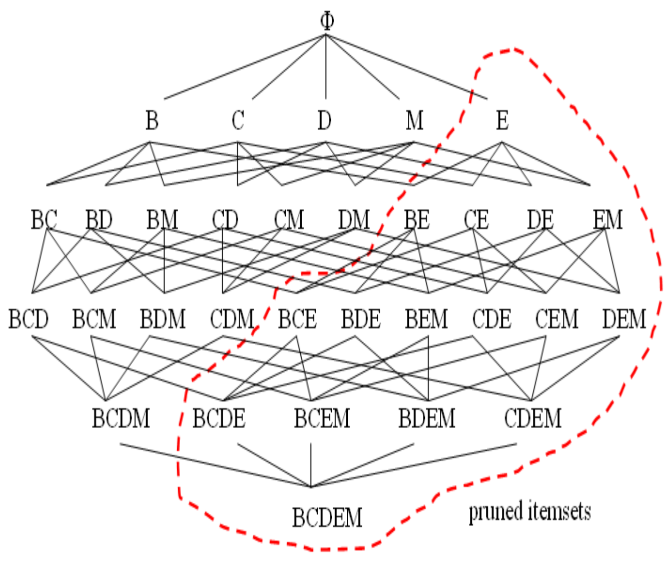


Fig. 3 a reduced search space after pruning an infrequent item E

```
Algorithm Frequent pattern discovery
Input: Transaction database, DB, of itemset I
       Minimum support threshold, minSup
Output: Sets of frequent patterns of length 1 to k,
        P1, .., Pk

1.  $P_1 = \{x \mid x$ is an item in $I$ and $s(P) \geq minSup\}$
                                        // 1-item pattern

2.  For ($k=1$; $P_k \neq \emptyset$; k++) do
2.1     $C_{k+1}$ = Generate_candidate($P_k$)
2.2     For each $T_i \in$ DB do
2.2.1       Increment the count $c$ of all candidates in $C_{k+1}$
                that are contained in $T_i$
2.3     $P_k = \{c \mid c \in C_k$ and c.count $\geq minS\}$
3.  Return $\cup_k P_k$   // return all sets of frequent patterns


Algorithm Generate_candidate

Input:  Pattern at current level, $P_{i-1}$
Output: Pattern in larger level, $C_i$

1.  $C_i = \emptyset$   // initialize candidate frequent pattern set
                        // as an empty set

2.  For each pattern $J$ in $P_{i-1}$ do

2.1     For each pattern $K$ in $P_{i-1}$ and $K \neq J$ do
2.1.1       If i-2 of the elements in $J$ and $K$ are equal then
2.1.2           If all subsets of $\{K \cup J\}$ are in $P_{i-1}$ then
2.1.3               $C_i = C_i \cup \{K \cup J\}$

3.  Return $C_i$
```

Fig. 4 a pseudo code of Apriori algorithm [1], [2] Some researchers [4], [7], [16], [17], [18] consider the issue of search space reduction through the concept of constraints. Our research is in the same direction as De Raedt et al [7]. We consider the problem of mining frequent patterns within a setting of constraint logic programming using the ECLiPSe constraint 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 modeling and solving with respect to the frequent pattern discovery in a transaction database.

III. Logic-based and Constraint-based Programming Paradigms

The 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. We suggest that such pattern oriented task can be efficiently implemented with logic-based languages.

A. Logic Program

In logic programming, a statement is called a clause, which is a disjunction of literals (atomic symbols or their negations) such as $p \vee q$ and $\neg p \vee r$. A statement is in clausal form if it is a conjunction of clauses such as $(p \vee q) \wedge (\neg p \vee r)$. Logic programming is a subset of first order logic in which clauses are restricted to Horn clauses.

A Horn clause, named after the logician Alfred Horn [15], is a clause that contains at most one positive literal such as $\neg p \vee \neg q \vee r$. Horn clauses are widely used in logic programming because their satisfiability property can be solved by resolution algorithm (an inference method for checking whether the formula can be evaluated to true).

A Horn clause with no positive literal, such as $\neg p \vee \neg q$, which is equivalent to $\neg(p \wedge q)$, is called *query* in Prolog and can be interpreted as '*:- p, q*' in which its value (true/false) to be proven by resolution method. A clause that contains exactly one positive literal such as *r* is called a *fact* representing a true statement, written in clausal form as '*r :-*' in which the condition part is empty and that means *r* is unconditionally true. Therefore, facts are used to represent data.

A Horn clause that contains one positive literal and one or more negative literals such as $\neg p \vee \neg q \vee r$ is called a *definite clause* and such clause can equivalently written as $(p \wedge q) \rightarrow r$ which in turn can be represented as a Prolog *rule* as *r :- p, q.* The symbol '*:-*' is intended to mean '$\leftarrow$', which is implication in first-order logic (it stands for 'if'), and the symbol ',' represents the operator $\wedge$ (or 'AND').

In Prolog, rules are used to define procedures and a Prolog program is normally composed of facts and rules. Running a Prolog program is nothing more than posing queries to obtain true/false answers. The advantages of using logic programming are the flexible form of query posing and the additional information regarding variable instantiation obtained from the Prolog system once the query is evaluated to be true.

The symbols *p, q, r* are called *predicates* in first-order logic programming and they can be quantified over variables such as *r(X) :- p(X,Y), q(Y).* This clause has the same meaning as $\forall X\ (p(X,Y) \wedge q(Y) \rightarrow r(X))$.The scope of variables is within a clause (delimit the end of clause with a period). Horn clauses are thus the fundamental concept of logic programming.

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 place holder for an output.

Variables in Prolog start with an uppercase letter such as Xs, L, X. The first statement of the length predicate declares the fact that the length of an empty list is zero. The last statement of length predicate can 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 transaction database of five items as appeared in Figure 1(a).

```
%%  FrequentPattern.pl
%         call ?-  r1.
%              ?-  r2(X).
%              ?-  clear.
tid(5).                % all transactions=5
s(3).                  % support=3 or 0.6
n([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 :-  n(X),  cL1(X).
r2(X) :-  cC2(X).
r3(X) :-  cC3(X).

clear :- retractall(l1(_)),
         retractall(c1(_)),
         retractall(c2(_)) ,
         retractall(l2(_)),
         retractall(c3(_)),
         retractall(l3(_)).
cL1([]).      % Create L1
cL1([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], Len)))
     ;
     cL1(T).

     % Create C2, L2
cC2(X) :-
     l1((X,_)),
     l1((X2,_)),
     X \==X2,
     write(X-X2),
     union(X, X2, Res),
     assert(c2((Res))) ,
     retract(l1((X,_))) ,
     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,Len)))
     ;
     cL2(T).

cC3(X) :-
      l2((X,_)),
      l2((X2,_)),
      X \==X2,
      write(X-X2),
      union(X, X2, Res),
      assert(c3((Res))) ,
      retract(l2((X,_))) ,
      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, Len)))
      ;
      cL3(T).
   f(H, X) :- item(X),
          subset(H, X).
```

The Prolog implementation of frequent itemset mining uses the findall, assert, and retract predicates, which are higher order. *Higher-order predicate* is a predicate in a clause that can quantify over other predicate symbols [5]. As an example, besides the rule *r(X):- p(X,Y), q(Y)*, if we are also given the following five Horn clauses (or facts): *p(1, 2). p(1, 3). p(5, 4). q(2). q(4).*

By asking the query: *?- r(X)*, we will get the response as '*true*' and also the first instantiation information as *X=1*. If we want to know all instantiations that make *r(X)* to be true, we may ask the query: *?- findall(X, r(X), Answer).*We will get the response: *Answer = [1,5]*, which is a set of all answers obtained from the predicate *r(X)* according to the given facts. The predicate symbol *findall* quantifies over the variables *X*, *Answer*, and the predicate *r*. The predicate *findall* is thus called a higher-order predicate.

B. 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. The structure of constraint logic program is as follows:

```
solve(Variables) :-
        setup_constraints(Variables),
        search(Variables).
```

Figure 5 demonstrates the different between logic program and constraint program on the same problem of map coloring [3]. The problem is given four colors and four regions, the program has to provide coloring scheme such that two consecutive regions have different colors.

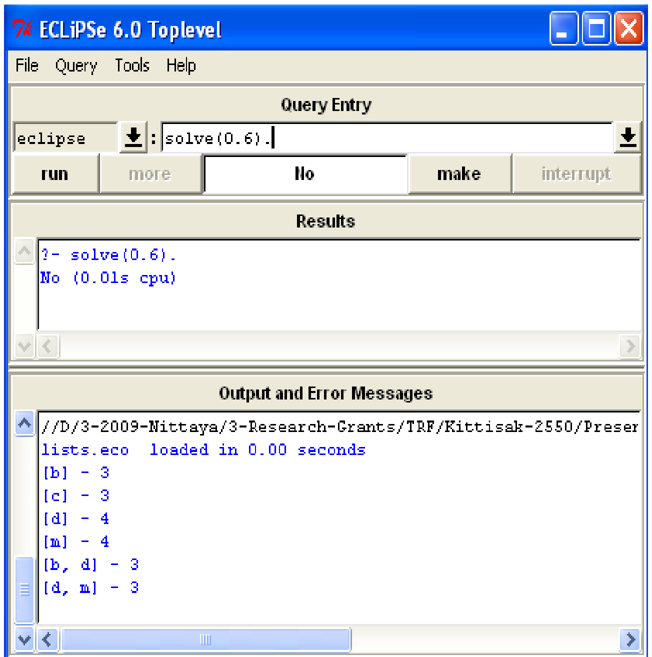
```
% Map coloring problem
%    Prolog style: Generate-and-test
color(red).
color(green).
color(blue).
color(yellow).
color_LP([A,B,C,D]) :-
                    color(A),
                    color(B),
                    color(C),
                    color(D),
                    A \= B,    A \= C,    A \= D,
                    B \= C,    B \= D,
                    C \= D.
```

```
% Map coloring problem
%    CLP style: Constrain-and-search
:- lib(fd).
color_CLP([A,B,C,D]) :-
             [A,B,C,D]::[red, green, blue, yellow],
             alldifferent([A,B,C,D]),
             labeling([A,B,C,D]).
```

Fig. 5 Logic programming versus constraint logic programming

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 Figure 1(a). The constraint problem can be formulated as: given a transactional database D and a minimum support threshold σ, find all frequent itemsets FS such that FS = { X ⊆ I | support(X) ≥ σ }. Screenshots of ECLiPSe system when a minimum support has been set to be 0.3 and 0.6 are shown in Figure 6.

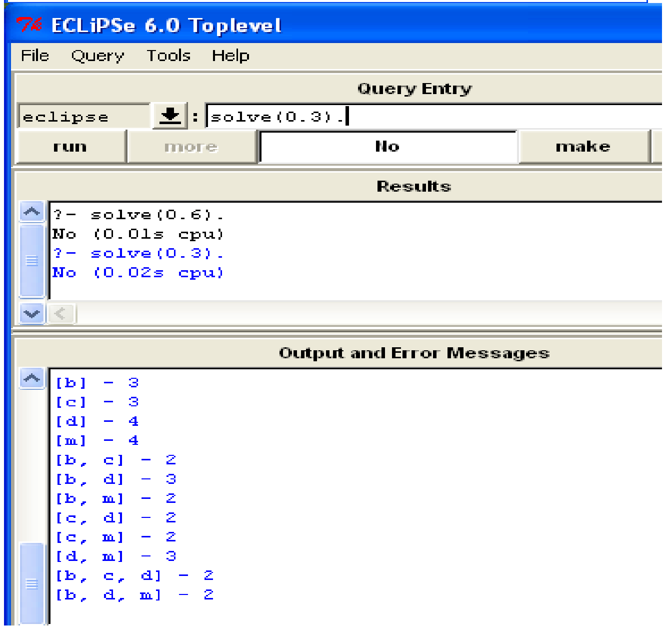
```
% Frequent pattern discovery with constraint
:-lib(listut).
:-dynamic(c/2).
item([[b], [c], [d], [e], [m]]).
t([c, m]).
t([b, c, d, e]).
t([b, d, m]).
t([b, c, d, m]).
t([d, m]).
solve(Sigma) :-      %   Sigma=MinSupport
    retract_all(c(_,_)),
    findall(X,t(X),AllTrans),
    length(AllTrans,NoTrans),
    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(1,Y)) ;  true) ),
    (for(K,1,5), param(MinSup)
      do (findall(Y, c(K,Y), RR),
          KK is K+1,
           findall(R1, (subset(S, RR),
                        myunion(S, R1),
                        length(R1,Len),
                        Len = KK),  Res11),
           maplist(flatten, Res11, Res12),
           remove_dups(Res12, Res1),
          (Res1=[] -> fail ; true), % break when [ ]
          (foreach(Y1, Res1), param(KK),
                          param(MinSup)
            do findall(Y1, (t(T), subset(Y1, T)), Res2),
               length(Res2,Len1) ,
               (Len1 >= MinSup ->
                         writeln(Y1-Len1),
                         assert(c(KK,Y1)) ; true) )
            )  ).    % end solve
myunion([X,Y], Z) :- union(X,Y,Z).
```



```
solve(0.6).
```

```
?- solve(0.6).
No (0.01s cpu)
```

```
//D/3-2009-Nittaya/3-Research-Grants/TRF/Kittisak-2550/Preser
lists.eco  loaded in 0.00 seconds
[b] - 3
[c] - 3
[d] - 4
[m] - 4
[b, d] - 3
[d, m] - 3
```



```
solve(0.3).
```

```
?- solve(0.6).
No (0.01s cpu)
?- solve(0.3).
No (0.02s cpu)
```

```
[b] - 3
[c] - 3
[d] - 4
[m] - 4
[b, c] - 2
[b, d] - 3
[b, m] - 2
[c, d] - 2
[c, m] - 2
[d, m] - 3
[b, c, d] - 2
[b, d, m] - 2
```

Fig. 6 Screenshots of ECLiPSe constraint system with 0.6 (top screen) and 0.3 (bottom) minimum support values, respectively

IV. Experimentation

We comparatively study the performance of our implementations of frequent pattern discovery using logic programming (SWI Prolog) and constraint logic programming (ECLiPSe constraint system). 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 standard UCI Machine Learning Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html) to test the speed and memory usage of the programs. The details of selected datasets are summarized in Table 1.

TABLE I
DATASET CHARACTERISTICS

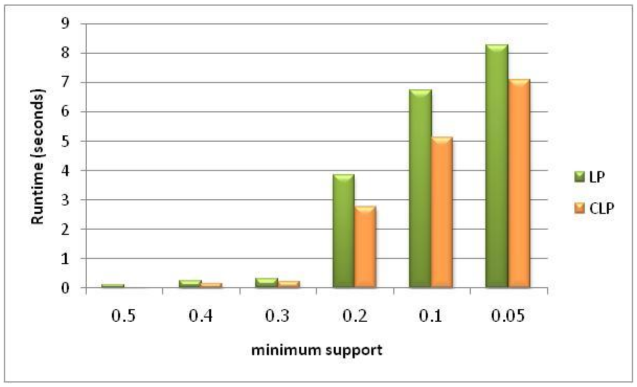
Dataset	File size	# Transactions	# Items
Vote	13.2 KB	300	17
Chess	237 KB	2,130	37
DNA	252 KB	2,000	61
Mushroom	916 KB	5,416	23

The frequent pattern discovery programs have been tested on each dataset with various *minSup* 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 7 and 8, 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.

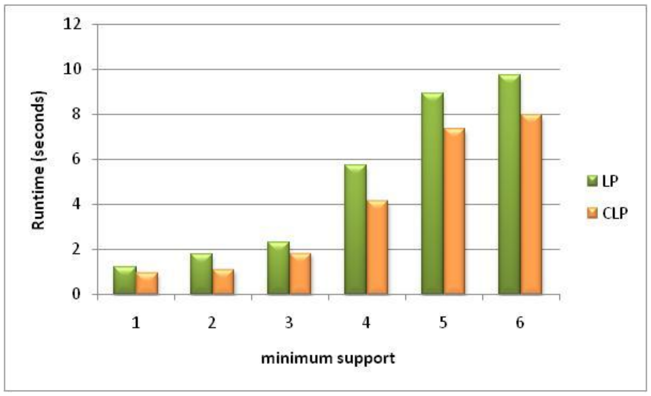
V. Conclusion

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.

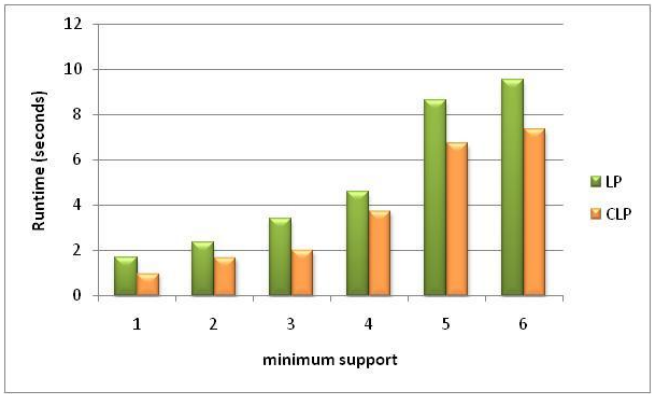
We also extend our study by implementing the algorithm with constraint logic programming paradigm using the ECLiPSe constraint system. The performance studies also support our intuition on the issue of efficiency because our constraint-based implementation is slightly more efficient than the conventional logic programming implementation in terms of speed and memory usage.



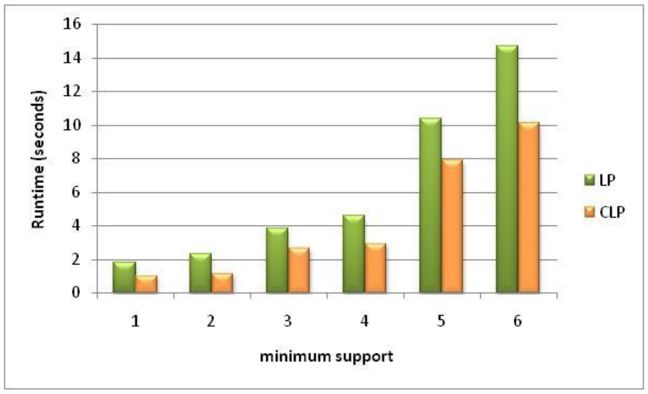
(a) Vote data



(b) Chess data

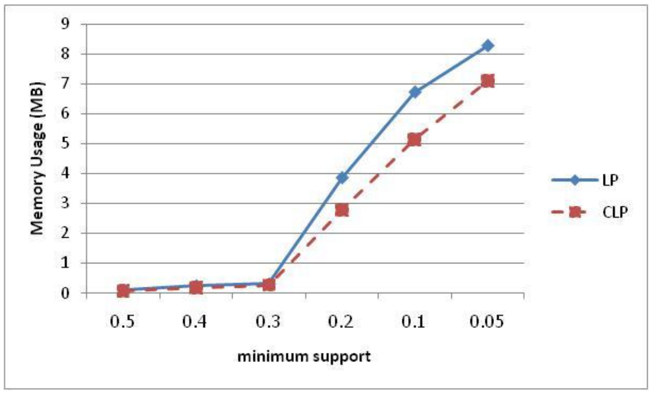


(c) DNA data

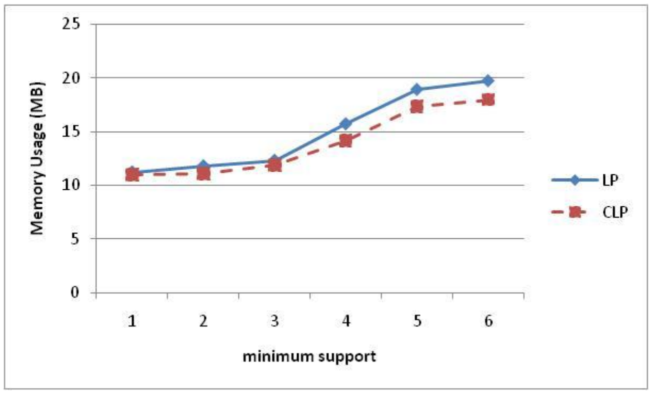


(d) Mushroom data

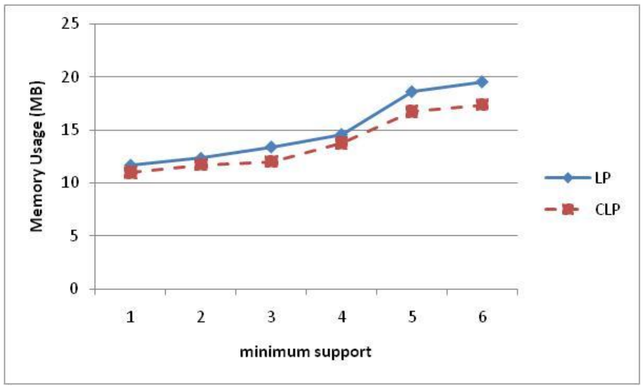
Fig. 7 speed comparison of logic program versus constraint logic program



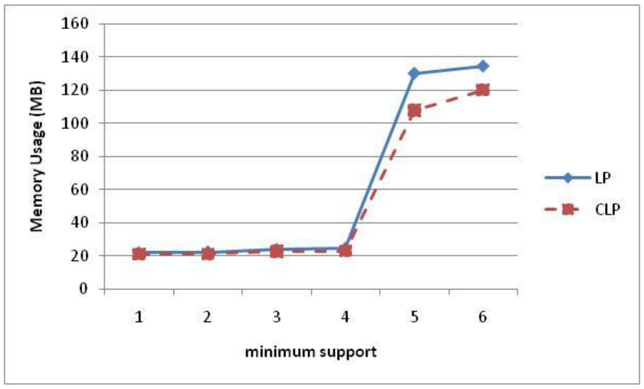
(a) Vote data



(b) Chess data



(c) DNA data



(d) Mushroom data

Fig. 8 Memory usage comparison of logic program versus constraint logic program

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.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM SIGMOD*, 1993, pp. 207-216.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. VLDB*, 1994, pp. 487-499.

[3] K. R. Apt and M. Wallace, *Constraint Logic Programming using ECLiPSe*, Cambridge University Press, 2007.

[4] S. Bistarelli and F. Bonchi, "Soft constraint based pattern mining," *Data and Knowledge Engineering*, vol. 62, 2007, pp. 118-137.

[5] I. Bratko, *Prolog Programming for Artificial Intelligence*, 3rd ed., Pearson, 2001.

[6] A. Cegler and J. Roddick, "Association mining," *ACM Computing Surveys*, vol.38, no.2, 2006.

[7] L. De Raedt, T. Guns, and S. Nijssen, "Constraint programming for itemset mining," in *Proc. KDD*, 2008, pp. 204-212.

[8] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus, "Knowledge discovery in databases: an overview," *AI Magazine*, vol. 13, no. 3, 1992, pp. 57-70.

[9] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, 2nd ed., Morgan Kaufmann, 2006.

[10] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. ACM SIGMOD*, 2000, pp. 1-12.

[11] J. Hu and X. Li, "Association rules mining including weak-support modes using novel measures," *WSEAS Transactions on Computers*, vol. 8, issue 3, 2009, pp. 559-568.

[12] M.C. Hung, S.Q. Weng, J. Wu, and D.L. Yang, "Efficient mining of association rules using merged transactions," *WSEAS Transactions on Computers*, vol. 5, issue 5, 2006, pp. 916-923.

[13] N. Kerdprasop and K. Kerdprasop, "Recognizing DNA splice sites with the frequent pattern mining technique," *International Journal of Mathematical Models and Methods in Applied Science*, vol.5, issue 1, 2011, pp. 87-94.

[14] R. Kuusik and G. Lind, "Algorithm MONSA for all closed sets finding: basic concepts and new pruning techniques," *WSEAS Transactions on Information Science & Applications*, vol. 5, issue 5, 2008, pp. 599-611.

[15] S.-H. Nienhuys-Cheng and R.D. Wolf, *Foundations of Inductive Logic Programming*, Springer, 1997.

[16] J. Pei and J. Han, "Can we push more constraints into frequent pattern mining?" in *Proc. ACM SIGKDD*, 2000, pp. 350-354.

[17] J. Pei, J. Han, and L. Lakshmanan, "Pushing convertible constraints in frequent itemset mining," *Data Mining and Knowledge Discovery*, vol. 8, 2004, pp. 227-252.

[18] R. Srikant, Q. Vu, and R. Agrawal, "Mining association rules with item constraints," in *Proc. KDD*, 1997, pp. 67-73.

[19] H. Sug, "Discovery of multidimensional association rules focusing on instances in specific class," *International Journal of mathematics and Computers in Simulation*, vol. 5, issue 3, 2011, pp. 250-257.

[20] H. Toivonen, "Sampling large databases for association rules," in *Proc. VLDB*, 1996, pp. 134-145.

[21] G. Yu, S. Shao, and X. Zeng, "Mining long high utility itemsets in transaction databases," *WSEAS Transactions on Information Science & Applications*, vol. 5, issue 2, 2008, pp. 202-210.

[22] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "Parallel algorithm for discovery of association rules," *Data Mining and Knowledge Discovery*, vol. 1, 1997, pp. 343-374.

Nittaya Kerdprasop is an associate professor and the director of Data Engineering research unit, school of computer engineering, Suranaree University of Technology, Thailand. She received her B.S. in radiation techniques from Mahidol University, Thailand, in 1985, M.S. in computer science from the Prince of Songkla University, Thailand, in 1991 and Ph.D. in computer science from Nova Southeastern University, U.S.A., in 1999. She is a member of IAENG, ACM, and IEEE Computer Society. Her research of interest includes Knowledge Discovery in Databases, Data Mining, Artificial Intelligence, Logic and Constraint Programming, Deductive and Active Databases.

Kittisak Kerdprasop is an associate professor at the school of computer engineering and one of the principal researchers of Data Engineering research unit, Suranaree University of Technology, Thailand. He received his bachelor degree in Mathematics from Srinakarinwirot University, Thailand, in 1986, master degree in computer science from the Prince of Songkla University, Thailand, in 1991 and doctoral degree in computer science from Nova Southeastern University, USA, in 1999. His current research includes Data mining, Machine Learning, Artificial Intelligence, Logic and Functional Programming, Probabilistic Databases and Knowledge Bases.