Mining Frequent Binary Expressions

Toon Calders* and Jan Paredaens

Universiteit Antwerpen,
Departement Wiskunde-Informatica,
Universiteitsplein 1, B-2610 Wilrijk, Belgium.
{calders,pareda}@uia.ua.ac.be

Abstract. In data mining, searching for frequent patterns is a common basic operation. It forms the basis of many interesting decision support processes. In this paper we present a new type of patterns, *binary expressions*. Based on the properties of a specified binary test, such as reflexivity, transitivity and symmetry, we construct a generic algorithm that mines all frequent binary expressions.
We present three applications of this new type of expressions: mining for rules, for horizontal decompositions, and in intensional database relations.
Since the number of binary expressions can become exponentially large, we use data mining techniques to avoid exponential execution times. We present results of the algorithm that show an exponential gain in time due to a well chosen pruning technique.

1 Introduction

In data mining, searching for frequent patterns is a basic operation. It forms the basis of many interesting decision support processes. Most data mining algorithms first start searching frequent patterns. In association rule mining [1], frequent itemsets are mined.

In this paper we will present a new type of patterns, *binary expressions*, that will be the basis of three applications. A binary expression is a conjunction of binary tests between attributes. An example of such an expression using the test $<$ is $(1 < 2) \wedge (2 < 3)$, expressing that attribute 1 is smaller than attribute 2 and attribute 2 is smaller than attribute 3. A binary expression will be called *frequent* iff the number of tuples satisfying the expression is bigger than a given threshold. Based on the properties of a specified binary test, such as reflexivity, transitivity and symmetry, we construct a generic algorithm that searches all frequent binary expressions. The properties are used to avoid syntactically different, but semantically equal expressions. The following two different expressions

$$(1 < 2) \wedge (2 < 3)$$

$$(1 < 2) \wedge (2 < 3) \wedge (1 < 3)$$

* Research Assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen).

will select exactly the same tuples. This is due to the fact that the binary test $<$ is transitive. We will give a method to avoid generating both expressions.

In this paper we present three applications that have the mining of frequent binary expressions in common. The first one is rule mining. A *binary association rule* is a rule $X \rightarrow Y$, where X and Y are binary expressions. Just like in association rule mining, we define notions of *support* and *confidence* for this type of rules. The similarities with association rules will be elaborated in Section 3.

The second application is in making horizontal decompositions. Horizontal decompositions have already been studied extensively[3] and are important in the context of distributed databases. When we want to make a horizontal decomposition, it is important to find a good criterion to split the relation. We will use the frequent binary expressions to make an optimal decomposition of a relation, based on target sizes of the fragments.

A third application is mining with *intensional database relations*[4]. In *Inductive Logic Programming* (ILP), the mining base typically contains intensional relations besides the traditional extensional relations. In this context, in the mining process, the intensional relations can be viewed as tests, in addition to the traditional tests such as $<, =, \ldots$, and rules that contain intensional relations can be mined in much the same way as other tests.

The outline of the paper will be as follows: in Section 2 binary expression, equivalence of expressions and some other notions are formally defined. In Section 3 the applications mentioned above are studied. In Section 4 we give some properties of the search space of the algorithm. In Section 5 we present a generic algorithm to find all frequent binary expressions. In Section 6 some experimental result of the algorithm are given. These results show good scalability properties of the algorithm. Section 7 concludes the paper.

An extended version of this paper is available as [2].

2 Definitions

Before we elaborate the three applications given in the introduction, we define formally the notions of respectively a relation, a binary test, a binary expression and equivalence of expressions

First we fix the relations we will consider. We only look at relations where all attributes have the same domain $\mathcal{U}$ [1]. $\mathcal{U}$ is a, possibly infinite, recursive set. We use an unnamed perspective; i.e. we will refer to the attributes by their number.

Definition 1. *An $n-ary$ relation is a finite subset of $\mathcal{U}^n$.*
A binary test[2] *θ over $\mathcal{U}$ is a recursive subset of $\mathcal{U} \times \mathcal{U}$. When $(u_1, u_2) \in \theta$, we will write $u_1 \theta u_2$.*

We now define the notion of an expression.

[1] $\mathcal{U}$ stands for *Universe.*
[2] We use the name *binary test* instead of *relation*, to avoid confusion with *database relations*. Actually, a binary test is just a relation in the mathematical sense.

Definition 2. *Let θ be a binary test. A (θ, n)-expression (θ and n will be omitted when clear from the context) is a conjunction of $(i\theta j)$'s, where $1 \leq i, j \leq n$. The set of all (θ, n)-expressions will be denoted by $\mathcal{E}(\theta, n)$.*

The previous definition gave the syntax of an expression. The next definition gives the semantics of expressions.

Definition 3. *Let $\mathcal{R}$ be an n-ary relation, e is a (θ, n)-expression. $\sigma_e \mathcal{R} = \{r \in \mathcal{R} \mid (\forall (i\theta j) \text{ in } e) r(i)\theta r(j)\}$[3] is the* selection on e of $\mathcal{R}$.

We are now ready to state the problem of mining all frequent binary expressions.

Definition 4. *Let $\mathcal{R}$ be an n-ary relation. The* frequency *of the binary expression $e \in \mathcal{E}(\theta, n)$, denoted freq$(e, \mathcal{R})$ is $\frac{|\sigma_e \mathcal{R}|}{|\mathcal{R}|}$.*
Let t be a number between 0 and 1. A binary expression $e \in \mathcal{E}(\theta, n)$ is t-frequent iff freq$(e, \mathcal{R}) \geq t$. (t will be omitted if clear from the context.)
The solution of the freq$(\mathcal{R}, t, \theta)$-problem is the set of all t-frequent (θ, n)-expressions.

Example 1. Consider the following relation:

1	2	3	4
1	2	1	4
1	5	6	2
1	5	1	1

The solution of the *freq*$(\mathcal{R}, \frac{2}{3}, <)$-problem is the set $\{1 < 2, 1 < 4, 3 < 2, 4 < 2, 1 < 2 \wedge 1 < 4, 1 < 2 \wedge 3 < 2, 1 < 2 \wedge 4 < 2\}$.

3 Applications

In this section we describe three applications of mining frequent binary expressions.

3.1 Rule Discovery

First we define a binary association rule.

Definition 5. *A* binary association rule *is a rule $X \rightarrow Y$, where X and Y are (θ, n)-expressions.*
The support *of the rule $X \rightarrow Y$ is freq$(X \wedge Y, \mathcal{R})$.*
The confidence *of the rule $X \rightarrow Y$ is $\frac{\mathit{freq}(X \wedge Y, \mathcal{R})}{\mathit{freq}(X, \mathcal{R})}$.*

Example 2. Consider the relation given in Example 1. The support of the binary association rule $1 < 4 \rightarrow 1 < 2$ is $\frac{2}{3}$. The confidence is 1.

[3] $r(i)$ denotes the i-th component of r; e.g. $(a, b, c)(2) = b$.

There are multiple similarities between association rules and binary association rules. Both rules give frequent dependencies that hold within the tuples themselves. Unlike for example roll-up dependencies [9], that describe relations between different tuples, association rules and binary association rules relate properties of attributes. In association rule mining, frequent itemsets can be considered as a conjunction of unary predicates. In this setting, binary association rules are a straightforward extension of the unary predicates to binary predicates. A binary association rule finds associations between binary predicates, where association rules find associations between unary predicates.

3.2 Horizontal Decompositions

Horizontal decompositions are very important for distributed databases. In many cases it is desirable to fragment the database over different locations. In that case it is important to find good criteria to divide the database. We will call this a split-problem. The solution to a split-problem is an expression that selects a fraction of the tuples whose cardinality is as close to the given goal as possible.

Example 3. Consider the relation given in Example 1. $3 < 2$ is a solution for the split-problem where the goal is $\frac{1}{2}$, and the binary test $<$, since $|\sigma_{3<2}\mathcal{R}|$ is as close to $\frac{|\mathcal{R}|}{2}$ as possible.

3.3 Intensional Database Relations

In *Inductive Logic Programming* (ILP) [4], mining conjunctions with intensional relations besides extensional relations is very common. The mining base used in Logic Programming typically contains a number of extensional relations and some intensional relations. The intensional relations are given by a set of describing rules in a logic programming language, for example Prolog or Datalog. In the context of mining, the intensional relations can be viewed as tests, in addition to the traditional tests such as $<, =, \ldots$

Example 4. Suppose the following logic program is given:

```
Related(X,Y):-Parent(X,Y);
Related(X,Z):-Related(X,Y) & Related(Y,Z);
Related(X,Y):-Related(Y,X);
Related(X,X);
```

From the last three rules we can conclude that the binary relation `Related` is transitive, symmetric, and reflexive.

In this example the intensional relation `Related` is in fact a binary test. Using this similarity, we can apply all results we obtain for mining binary expressions to this case. Suppose for example than we have a predicate `King` and we use the binary test `Related`. We could for example find that the expression `Related`(X,Y)&`King`(X)&`King`(Y) is frequent. Because we know that `Related` is symmetric, we know that testing `Related`(X,Y)&`Related`(Y,X)&`King`(X)&`King`(Y) is redundant.

4 The Search Space

The *freq*$(\mathcal{R}, t, \theta)$-problem is essentially a search-problem. We want to find all frequent binary expressions in the search space $\mathcal{E}(\theta, n)$. For all binary tests θ, the number of expressions in $\mathcal{E}(\theta, n)$ is $2^{(n^2)}$, since the number of pairs of attributes is n^2, and for every pair (x, y), $x\theta y$ is present or absent. However, it is not always necessary to consider all expressions. When there are equivalent expressions, there is no need to consider them all.

Example 5. $1 = 2 \wedge 1 = 3$ is equivalent to $1 = 2 \wedge 2 = 3$.

In Tab. 1, for some binary tests and different number of attributes, the total number of non-equivalent elements in the search space is given. For example, for the equality and 3 attributes, the search space is $\{1 = 1, 1 = 2, 1 = 3, 2 = 3, 1 = 2 = 3\}$. Therefore, in Tab. 1, the row for $n = 3$ contains 5, the size of the search space. The value of $2^{(n^2)}$ is also given for each value of n.

Table 1. Size of the search space for some binary tests

n	$<$	$\leq$	$\neq$	$=$	2^{n^2}
1	2	1	2	1	2
2	3	4	2	2	16
3	19	29	8	5	512
4	219	355	64	15	65536

To exploit the equivalence of expressions we need some properties of the expressions to decide when two expressions are equivalent. Based on these properties we will construct a mechanism to avoid generation of equivalent expressions.

Definition 6. *A binary test θ has property*
P_1 = reflexive *iff for all $1 \leq i \leq n$, $(i\theta i)$ holds.*
Q_1 = anti-reflexive *iff for all $1 \leq i \leq n$, $(i\theta i)$ does* not *hold.*
P_2 = symmetric *iff for all $1 \leq i, j \leq n$, if $(i\theta j)$ then also $(j\theta i)$ holds.*
Q_2 = anti-symmetric *iff for all $1 \leq i, j \leq n$, if $(i\theta j)$, then $(j\theta i)$ does* not *hold.*
P_3 = transitive *iff for all $1 \leq i, j, k \leq n$, if $(i\theta j)$ and $(j\theta k)$, then also $(i\theta k)$ holds.*
Q_3 = anti-transitive *iff for all $1 \leq i, j, k \leq n$, if $(i\theta j)$ and $(j\theta k)$, then $(i\theta k)$ does* not *hold.*

From definition 6, it results that for each i, P_i holds or Q_i, or none. This means that there exist $3^3 = 27$ possible combinations. However, only 16 of them really exist.

Definition 7. *Let θ be a binary test, and let $P \subseteq \{P_1, P_2, P_3\}$ be the set of P-properties of θ. An expression $e \in \mathcal{E}(\theta, n)$* is closed *iff every conjunct $(i\theta j)$ that*

is necessary by the properties of P appears in e.
Let $Q \subseteq \{Q_1, Q_2, Q_3\}$ be the set of Q-properties of θ. An expression $e \in \mathcal{E}(\theta, n)$ is valid *iff all conjuncts that are forbidden by the properties of Q, do not appear in e.*

Example 6. Clearly, $e = (1 < 2) \wedge (2 < 3)$ is not closed since $1 < 3$ is necessary by the transitivity and does not appear in e. On the other hand $e' = (1 < 2) \wedge (2 < 3) \wedge (1 < 3)$ is closed. e and e' are both valid expressions. $(1 < 2) \wedge (2 < 1)$ is not valid, since the anti-symmetry forbids $(2 < 1)$ when $(1 < 2)$ is present.

Lemma 1. *Given a valid (θ, n)-expression e, there is a* unique *valid and closed expression e', that is equivalent with e. e' is obtained by augmenting e with all conjuncts that are necessary by the properties of P of θ. e' is called the* closure *of e.*

In example 6, e' is the closure of e. It is clear now that in every equivalence class of expressions there is a unique closed expression. Since we have to test only one expression of each equivalence class, for solving the *freq*$(\mathcal{R}, t, \theta)$-problem, it is sufficient to test each closed expression.

5 Algorithm

In this section we describe an algorithm that finds all frequent binary expressions given a binary test and a relation. Basically, the algorithm performs a levelwise search as described in [6]. The levelwise algorithm is a generate-and-test algorithm. It highly depends on a *monotonicity principle* saying, roughly speaking, that whenever e_1 is more specific than e_2, and the result of e_2 is too small then the result of e_1 is also too small. The next proposition states this monotonicity principle.

Proposition 1. *Let e_1 and e_2 be two expressions, $\mathcal{R}$ is a relation, and e_1 is more specific than e_2, then $|\sigma_{e_1}\mathcal{R}| \leq |\sigma_{e_1}\mathcal{R}|$.*

Consider the following situation: We want to solve the *freq*$(\mathcal{R}, \frac{1}{2}, <)$-problem, and we know that the expression $1 < 2$ is not frequent. Then, using Proposition 1, we know that $1 < 2 \wedge 1 < 3$ cannot be frequent, since $1 < 2 \wedge 1 < 3$ is more specific than $1 < 2$. So, in this situation there is no need to count the frequency of $1 < 2 \wedge 1 < 3$. We can *prune* the expression $1 < 2 \wedge 1 < 3$.

The search space of our algorithm will consist of all closed and valid expressions. In Fig. 1 a part of the search space for *freq*$(\mathcal{R}, 3, <)$ is showed. When we use the term *children of an expression*, we mean the expressions that are next more specific in the lattice.

Our algorithm will try to prune as much of the search space as possible. We start with the most general expression of our search space, and we iteratively test more specific expressions, without ever evaluating those expressions that cannot be frequent given the information obtained in earlier iterations. More precisely, the search space is traversed level by level, from general to specific. In each

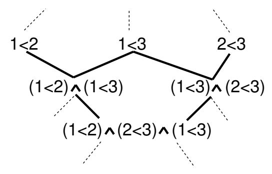

Fig. 1. A part of the search space

iteration, the set *candidates* will contain the candidate frequent expressions. An "apriori trick" is used; if the frequency of e is below the threshold, and $e' \preceq e$, then we know a priori that e' must fail the frequency threshold. For this reason, all expressions that failed the frequency threshold are stored in the set $TooLow$. This gives us the framework of Fig. 2, which actually is a *levelwise search* [6]. Steps 3 to 7 are testing the candidates against the database and bookkeeping. In step 8 the children of the frequent candidates are generated as the candidates for the next iteration. In step 9, we use the apriori trick to prune away candidates that cannot be frequent due to information obtained in previous iterations.

```
1.  $candidates = \{\top\}$; $Output = \{\}$; $TooLow = \{\}$
2.  while($candidates \neq \{\}$) do
3.  Test          Test $candidates$ against the database.
4.                $fcan = \{c \in candidates \mid c$ is frequent$\}$
5.                $nfcan = candidates - fcan$
6.                $Output = Output \bigcup fcan$
7.                $TooLow = TooLow \bigcup nfcan$
8.  Generate      $candidates = \bigcup_{p \in fcan} \{c \mid c$ is a child of $p\}$
9.  Prune         $candidates = candidates - \{c \mid \exists n \in TooLow : c \preceq n\}$
10. end while
```

Fig. 2. Algorithm for finding frequent expressions

5.1 Testing

In the test-phase, the frequencies of the candidates are tested against the database. The calculation of the frequency of an expression is very costly, since we need to iterate over all tuples in the relation to count the number of tuples that satisfy the expression. To limit the overhead, all candidates in an iteration are tested in the same run over the database.

5.2 Generation

In the generation phase, we need to generate all closed and valid children of the frequent candidates. As can be seen in Fig. 1, all children are generated by adding one conjunct, and taking the closure. However, not all conjuncts can be used for generating children; in Fig. 1, the closure of $(1 < 2)$, augmented with $(2 < 3)$ is $(1 < 2) \wedge (2 < 3) \wedge (1 < 3)$, and this is no child of $(1 < 2)$, since $(1 < 2) \wedge (1 < 3)$ lies between them. In the generation phase this problem is handled.

In the framework of the algorithm, in the generation phase, all children of the frequent candidates are generated. It is however sufficient that every expression only generates a subset of its children, as long as for every expression there is still at least one generating parent. We only generate those children that are induced by a sublattice of the search space.

Not generating all children does no harm; still all expressions are generated. On the other hand, not generating all children has a couple of advantages.

– In step 8. of the algorithm, all expressions generated by frequent candidates are added as new candidates. Probably lots of duplicates are generated. These duplicates need to be removed. The less children are generated, the less duplicates need to be removed.
– By not generating all children, some early pruning is applied. We will discuss this in more detail in the subsection on pruning.

From this discussion we can conclude that ideally each expression has exactly one generating parent. This is the case when the spanning sublattice is a tree.

The generation phase is discussed more in-depth in [2], where we introduce two strategies for the generation, ρ_1 and ρ_2.

5.3 Pruning

A basic operation of the algorithm is the pruning. It is essential that this operation is performed as efficiently as possible. The pruning implies that for every expression e that is generated in step 8 of the algorithm, we need to investigate whether there is an expression l in $TooLow$ such that $e \preceq l$. If this is the case, we can prune e.

From previous research, we can conclude that a *trie* is a good structure to store sequences. A trie uses common prefixes between the sequences to store them more efficiently. We are not going into detail on tries, for more elaborated work on tries, we refer to [5] and [7].

Step 8 is not the only step in which pruning occurs. When all *generating* parents of an expression are infrequent, the expression will never be generated, even when there are other parents that are frequent. Thus, the less parents a node has, the bigger the chance that it never will be generated if some of its parents are infrequent. This type of pruning is called *early pruning*. When an expression is not pruned early, it can still be pruned in step 8 of the algorithm. This situation occurs when at least one generating parent is frequent, and at least one other parent is infrequent. Due to the monotonicity principle the expression will be pruned.

6 Experimental Results

In this section we present some experimental results. We implemented ρ_1 and ρ_2. For a definition and a discussion of these two algorithms, we refer to [2]. The source code of both implementations can be obtained at `http://cc-www.uia.ac.be/u/calders/`.

6.1 Effectiveness of Pruning

In Fig. 3 (left), a lower bound for the total number of closed and valid expressions in $\mathcal{E}(<, n)$ is given for $n = 2, 4, \ldots, 20$ for reference. In Fig. 3 (right), some tests on a randomized dataset are given for increasing number of attributes. The number of expressions that are examined by our algorithms is given for a threshold 0.3, and for increasing number of attributes. Note that the scale of the graph representing the total size of the search space is logarithmic. The number of expressions examined by the algorithms in this example is exponentially less than the total number of elements in the search space.

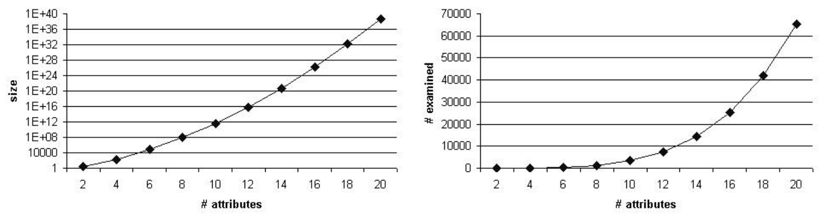

Fig. 3. The size of the search space (left) versus the number of expressions that were investigated (right)

6.2 Scalability

In Fig. 4, the running time of the two algorithms is measured. When the number of attributes grows, refinement operator ρ_2 becomes much more efficient than ρ_1. In the left graph, a threshold of 0.4 was used, and the binary test was $<$. In the right graph, the binary test $=$ was used, and the test was done with the refinement operator ρ_2, with a threshold of 0.5. In both graphs, the dataset was randomly generated. The number of values in $\mathcal{U}$ was 2 in the right graph, and 7 in the left one.

7 Conclusion

Binary expressions are an interesting type of patterns for data mining. In this paper we presented three applications of frequent binary expressions; binary

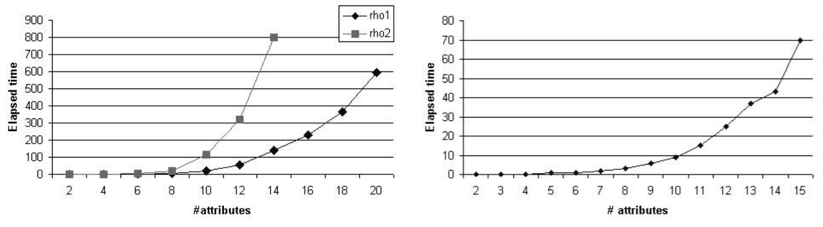

Fig. 4. Scalability in the number of attributes

rules, that essentially are extensions of association rules to binary predicates, horizontal decompositions and the mining of intensional database relations. We presented and tested an algorithm for finding frequent binary expressions. The algorithm exploited background information such as reflexivity, transitivity and symmetry of the binary tests to optimize the search.

References

[1] R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, Washington, D.C., 1993

[2] T. Calders, and J. Paredaens. Mining Binary Expressions: Applications and Algorithms. Technical Report, Universiteit Antwerpen, Belgium, June 2000.

[3] P. De Bra. Horizontal decompositions based on functional-dependency-set-implications. In *ICDT*. Springer-Verlag, 1986.

[4] L. Dehaspe. Frequent pattern discovery in first-order logic. PhD thesis, Katholieke Universiteit Leuven, Belgium, Dec. 1998.

[5] J. Han, J.Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD*, 2000

[6] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. In *Data Mining and Knowledge Discovery 1(3)*, 1997.

[7] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining access patterns efficiently from web logs. In *PAKDD*, 2000.

[8] M. Y. Vardi. The decision problem for database dependencies. In *Inf. Proc. Letters 12(5)*, 1981.

[9] J. Wijsen, R. Ng, and T. Calders. Discovering roll-up dependencies. In *Proc. ACM SIGKDD*, 1999.