Mining Binary Expressions: Applications and Algorithms

Toon Calders* Jan Paredaens

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

Technical report TR0008, June 2000

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.

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

Contents

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][2][3][7], frequent itemsets are mined. In episode rule mining [4][12], frequent episodes are mined. In this paper we present a new type of patterns, *binary expressions*, that is 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 is *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

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

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

select exactly the same tuples. This is due to the fact that the binary test $<$ is transitive. We 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. Just like frequent itemsets form the basis of association rules, binary expressions form the basis of a new type of rules. 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 are elaborated in Section 3.

The second application is in making horizontal decompositions. Horizontal decompositions have already been studied extensively[5] 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 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*[6]. 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 is 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.

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 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 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 write $u_1 \theta u_2$. $\triangleleft$

We now define the notion of an expression.

Definition 2 Let θ be a binary test. A (θ, n)-expression (θ and n is 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 is denoted by $\mathcal{E}(\theta, n)$. $\triangleleft$

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. $\sigma_e \mathcal{R} = \{r \in \mathcal{R} \mid (\forall (i,j)\ in\ e) r(i) \theta r(j)\}$[3] is the *selection on e of $\mathcal{R}$*. $\triangleleft$

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*$(e)$ 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*$(e) \geq t$. (t is 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. $\triangleleft$

Example 1 Consider the relation given in Fig. 1. The solution of the *freq*$(\mathcal{R}, \frac{5}{7}, <)$-problem is the set $\{1 < 2, 1 < 4\}$. $\triangleleft$

[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.

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

1	2	3	4	5
1	2	2	4	1
1	5	6	2	1
1	5	1	3	7
3	5	6	2	3
2	7	2	4	5
3	2	4	4	6
6	2	3	5	5

Figure 1: The relation $\mathcal{R}$

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 $freq(X \wedge Y)$.
The *confidence* of the rule $X \rightarrow Y$ is $\frac{freq(X \wedge Y)}{freq(X)}$. ◁

Example 2 Consider the relation given in Fig. 1. The support of the binary association rule $1 < 2 \rightarrow 1 < 4$ is $\frac{4}{7}$. The confidence is $\frac{4}{5}$. ◁

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 [16] or approximate dependencies [9][10], 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. In this paper we study how one can apply data mining to find expressions such that the number of tuples that satisfy the expression approximates as good as

possible an in advance fixed goal. We 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 Fig. 1. $3 < 4$ is a solution for the split-problem where the goal is $\frac{1}{2}$, and the binary test $<$, since $|\sigma_{3<4}\mathcal{R}|$ is as close to $\frac{|\mathcal{R}|}{2}$ as possible. ◁

In the following definition the split-problem is formalized.

Definition 6 Let $\mathcal{R}$ be a relation with n attributes, g a number between 0 and 1, and θ a binary test. $e \in \mathcal{E}(\theta, n)$ is a solution of the $Split(\mathcal{R}, g, \theta)$-problem if $|\frac{|\sigma_e\mathcal{R}|}{|\mathcal{R}|} - g|$ is minimal; i.e. there is no other expression e' in $\mathcal{E}(\theta, n)$ such that $|\frac{|\sigma_{e'}\mathcal{R}|}{|\mathcal{R}|} - g|$ is smaller. ◁

3.3 Intensional Database Relations

In *Inductive Logic Programming* (ILP) [6], 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):-Father(X,Y);
Related(X,Y):-Mother(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. We now define formally when two expressions are equivalent. Therefore, we introduce an ordering on the set of expressions, based on query containment [15].

Definition 7 Let $e_1, e_2 \in \mathcal{E}(\theta, n)$, θ is a binary test, n is a positive integer.

- e_1 is *more specific* than e_2, denoted $e_1 \preceq e_2$, iff for every n-ary relation $\mathcal{R}$ the following holds: $\sigma_{e_1}\mathcal{R} \subseteq \sigma_{e_2}\mathcal{R}$[4].
- e_1 is *more general* than e_2 iff $e_2 \preceq e_1$.
- We write $e_1 \prec e_2$ iff $e_1 \preceq e_2$ and not $e_2 \preceq e_1$.
- e_1 and e_2 are *equivalent* iff $e_1 \preceq e_2$ and $e_2 \preceq e_1$.

◁

Example 5 $(1 < 2) \wedge (2 < 3) \prec (1 < 2)$
$(1 < 2) \wedge (2 < 3) \prec (1 < 3)$
$1 = 2 \wedge 1 = 3$ 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 (i.e. equivalent expressions are considered equal). 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, for the equality. The value of $2^{(n^2)}$ is also given for each value of n. It is clear that the number of non-equivalent expressions with for example $<$ is much smaller than the total number of expressions. So, when we search for an optimal solution it is a good idea to exploit the equivalence between expressions. When we neglect this fact, we are doomed to search a space with up to 2^{n^2} expressions.

In definition 7, equivalence is introduced as a semantic notion. Two expressions are equivalent, if for all relations they select the same subsets. This definition cannot be used in a practical solution. 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 construct a mechanism to avoid generation of equivalent expressions. In this paper we restrict ourselves to combinations of the properties (anti-) transitivity, (anti-) symmetry and (anti-) reflexivity.

[4] $e_1 \preceq e_2$ iff query σ_{e_1} is contained in σ_{e_2}.

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

Definition 8 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. $\triangleleft$

Remark that a binary test cannot have both properties P_n and Q_n. On the other hand it can have only P_n, or only Q_n, or none of the two. Indeed, $=$ has property P_2, $<$ has property Q_2, and $\leq$ has P_2 nor Q_2. Hence, for each $n = 1, 2, 3$, there are three possibilities for a binary test; P_n, Q_n or none. This means that there could be at most 27 possibilities for a binary test. Table 2 shows however that only 16 of them really exist, and it gives an example for each of them.

Definition 9 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. $\triangleleft$

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. $\triangleleft$

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 split problem $freq(\mathcal{R}, t, \theta)$, it is sufficient to test each closed expression.

Table 2: Combinations of reflexivity, symmetry and transitivity. A and B indicate attributes, $1, 2, \ldots$ are used as constants

Reflexive	Symmetric	Transitive	Possible relation
yes	yes	yes	$A = B$
yes	yes		$\|A - B\| \leq 1$
yes	yes	anti	**Impossible**
yes		yes	$A \geq B$
yes			$A - B \leq 1$
yes		anti	**Impossible**
yes	anti	yes	**Impossible**
yes	anti		**Impossible**
yes	anti	anti	**Impossible**
	yes	yes	$(A = 1) \wedge (B = 1)$
	yes		$(A = 1) \vee (B = 1)$
	yes	anti	**Impossible**
		yes	$A \leq 1 \leq B$
			A(B+1)=2
		anti	**Impossible**
	anti	yes	**Impossible**
	anti		**Impossible**
	anti	anti	**Impossible**
anti	yes	yes	$A\theta B$ always false
anti	yes		$A \neq B$
anti	yes	anti	$\|A - B\| = 1$
anti		yes	**Impossible**
anti			$((AB = 0) \wedge (A + B \neq 0)) \vee (A < B)$
anti		anti	$\|A - B\| = 1 \vee A - B = 3$
anti	anti	yes	$A < B$
anti	anti		$A < B < 3A$
anti	anti	anti	$A - B = 1$

An important property, that we use in our algorithm is that the projection of closed expressions is closed.

Definition 10 Let θ be a binary test.
Let $e \in \mathcal{E}(\theta, n)$, $I \subseteq \{1, \ldots, n\}$. $\pi_I e$, *the projection of* e *on* I, is the expression that contains all conjuncts $(i\theta j)$ of e where $i, j \in I$.
$\triangleleft$

Proposition 1 *If* $e \in \mathcal{E}(\theta, n)$ *is closed, then also is* $\pi_I e$*, for all* $I \subseteq \{1, \ldots n\}$.

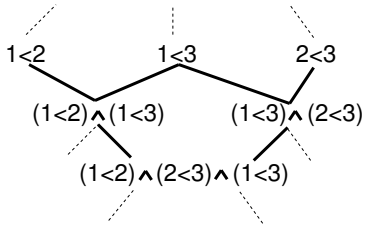

Figure 2: A part of the search space

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 [13]. The levelwise algorithm is a generate-and-test algorithm. It highly depends on a *monotonicity principle* saying, roughly speaking, that whenever $e_1 \preceq e_2$, 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 2 *Let e_1 and e_2 be two expressions, $\mathcal{R}$ is a relation, and $e_1 \preceq e_2$, 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 2, we know that $1 < 2 \wedge 1 < 3$ cannot be frequent, since $1 < 2 \wedge 1 < 3 \preceq 1 < 2$. So, in this situation there is no need to count the frequency of $1 < 2 \wedge 1 < 3$. We can *prune* the expression $1 < 2 \wedge 1 < 3$.

Another important aspect of the algorithm is that only the closed and valid expressions are evaluated. All other expressions are equivalent to such an expression. So, the search space of our algorithm consists of all closed and valid expressions. The ordering $\preceq$ induces a lattice-structure on this search space. We can proof that `true` is always the unique top element in this lattice, and we denote this top element by $\top$. In Fig. 2 a part of the search space of the $freq(\mathcal{R}, 3, <)$ is showed. When we use the term *children of an expression*, we mean the expressions that are next more specific in the lattice. An important result of the fact that we only consider closed expressions is the next proposition.

Proposition 3 *Let e_1, e_2 be* closed *and* valid *expressions. The following two statements are equivalent:*

- $e_1 \preceq e_2$
- $\{(i\theta j) \mid (i\theta j)$ *is a conjunct in* $e_2\} \subseteq \{(i\theta j) \mid (i\theta j)$ *is a conjunct in* $e_1\}$

Example 7 Suppose we want to test whether $(1 < 2) \wedge (2 < 3) \wedge (1 < 3) \wedge (1 < 4) \preceq (1 < 3) \wedge (2 < 3) \wedge (1 < 4)$. This is equivalent with the question whether $\{(1 < 3), (2 < 3), (1 < 4)\} \subseteq \{(1 < 2), (2 < 3), (1 < 3), (1 < 4)\}$. ◁

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

Figure 3: Algorithm for finding frequent expressions

Our algorithm tries 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 iteration, the set *candidates* contains 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. 3, which actually is a *levelwise search* [13]. 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.

The three important operations in this framework are the testing of candidates, the generation of new candidates and the pruning.

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.

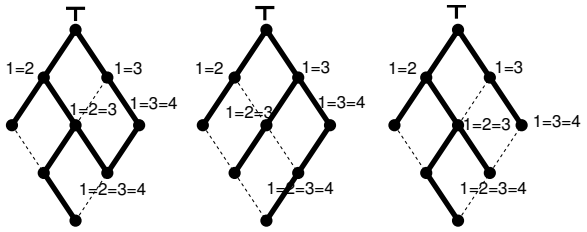

Figure 4: Three spanning lattices of a searchspace

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. 2, all children are generated by adding one conjunct, and taking the closure. However, not all conjuncts can be used for generating children; in Fig. 2, 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.

Example 8 In Fig. 4 some examples of this are given. The bold dots represent the elements of the search space; these are the closed and valid expressions. The bold lines represent the ways the generation takes place. In the first lattice, for example, the top-element generates both its children. The right child only generates a subset of its children. In the second search space, the spanning sublattice is a tree. ◁

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 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.

In Fig. 6 and 7, two functions that describe sublattices of the search space are given. The functions give for every expression its successors; i.e. when ρ is such an expression, and $c \in \rho(p)$, then p is a generating parent for c. These functions correspond with the so-called *refinement operators* in [6]. The first function ρ_1 always defines a spanning tree. The second function, ρ_2, does not

	1	2	3	4	5
1	**1**	**4**	**9**	**16**	**25**
2	2	3	8	15	24
3	5	6	7	14	23
4	10	11	12	13	22
5	17	18	19	20	21

Figure 5: The numbers assigned to the edges by the function number(i,j). The square corners are bold

$number(1,1) = 1$
$number(1,j) = number(j-1,1) + 1$
$number(i,j) = number(i-1,j) + 1, i \leq j$
else $number(i,j) = number(i,j+1) + 1$
$Edge(m) = (i,j)$ iff $number(i,j) = m$
$SquareCorner(i,j)$=true iff $i = 1$.
$Last(i,j)$=true iff $i = j = n$.

$GenerateClosedSuccessors(e,m)$
output: set $\mathcal{S}$.
while ((closed(e) and not Last(Edge(m))) or not $SquareCorner(Edge(m))$) do

1. m=m+1
2. if $(e \wedge Edge(m))$ is valid then
   - if $(e \wedge Edge(m))$ is closed then
     Add $e \wedge Edge(m)$ to $\mathcal{S}$
     else Add $GenerateClosedSuccessors(e \wedge Edge(m), m)$ to $\mathcal{S}$.

end while

$m = max\{number(i,j) \mid (i\theta j) \in e\}$
$\rho_1(e) = GenerateClosedSuccessors(e,m)$

Figure 6: Refinement operator ρ_1

$Generate(e, m)$

$candidates = \{(i\theta j) \notin e$, with $m \in \{i, j\}\}$
$implies = \{\}$
for all tests $(i\theta j) \in candidates$ do

- Calculate the closure of $e \wedge (i\theta j)$
- Add $(i\theta j) \rightarrow (i'\theta j')$ to implies for all $(i'\theta j') \in candidates \cap cl(e) \wedge (i\theta j)$.

end for
for all tests t in $candidates$ do

- if $\exists t' \in candidates$: and $t \rightarrow t'$ in $implies$ and $t' \rightarrow t$ not in implies then remove t from candidates.
- if $\exists t' \in candidates : number(t') < number(t)$ and both $t \rightarrow t'$ and $t' \rightarrow t$ in $implies$ then remove t from candidates.

end for
$Generate(e, m) = \{cl(e \wedge t) \mid t \in candidates\}$

$m = max\{i \mid (i, j) \in e \vee (j, i) \in e\}$
$\rho_2(e) = \bigcup_{i \leq m} Generate(e, i)$

Figure 7: Refinement operator ρ_2

define a spanning tree, but is on the other hand more efficient in generating successors. In the experiments we compare the two functions. The proof that ρ_1 and ρ_2 are correct, is beyond the scope of this paper.

5.2.1 Comments on the refinement operators

As said earlier, the refinement operators calculate children of nodes. Using these refinement operators, we can, starting with the top-element, enumerate all elements in the search space by going top-down. ρ_1 and ρ_2 are designed to go through the search space $\mathcal{E}(\theta, n)$. Thus, given an expression, ρ_1 and ρ_2 calculate some of its children. In order to be able to iterate over the full search space, for every expression $e \neq \top$, there must be an expression p such that $\rho_*(p)$ contains e.

In both ρ_1 and ρ_2, every possible conjunct $i\theta j$ in an expression is assigned a unique number; $number(i, j)$. In this way the conjuncts are ordered. This order is chosen as showed in Fig. 5. So, all conjuncts $i\theta j$ with both i and j smaller than k, come before all conjuncts (i', j') with either $i' \geq k$ or $j' \geq k$.

$SquareCorner(i, j) = true$ means that the conjunct $i\theta j$ is the last conjunct with both attributes smaller than or equal to j. Every expression can now be expressed as a set of numbers. Using Proposition 1, we have: if $number(i, j) = sc$ and $SquareCorner(i, j)$, and $E = \{c_1, c_2, \ldots, c_m\}$ represents a closed and valid expression e, then $E' = \{c \in E \mid c \leq sc\}$ represents also a closed and valid expression, since E' represents the projection of e on $\{1, \ldots, j\}$. Both ρ_1 and ρ_2 rely heavily on this observation.

ρ_1 is based on brute force searching of children of an expression. To avoid multiple generations of the same expression, ρ_1 generates only the children e' of e, such that the numbers of the conjuncts in e' that aren't in e are bigger than the biggest number of a conjunct in e. It is clear that in this way every expression can only be generated by one parent. ρ_1 adds conjuncts until the expression is complete or until a square corner is reached. When this corner is reached and the expression is not complete, every super-expression will not be complete, since the incomplete expression will be a projection. For the exact definition of ρ_1, see Fig. 6.

ρ_2 calculates first the impact of adding a certain conjunct. For example: adding $1 < 2$ to $2 < 3$ implies $1 < 3$. Therefore, $1 < 2 \rightarrow 1 < 3$ will be stored in *implies*. After this step, only the conjuncts that do not imply other conjuncts are added. For example: $1 < 3$ is added to $2 < 3$, but $1 < 2$ is not added to $2 < 3$, because $1 < 2 \rightarrow 1 < 3$ is in implies. For the exact definition of ρ_2, see Fig. 7.

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. Since the algorithm only generates closed

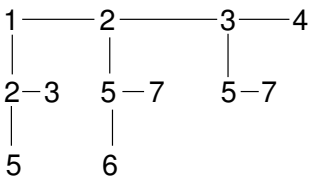

Figure 8: A trie containing the items 125,13,256,27,35,37 and 4

expressions, the test $e \preceq l$ is equal to the test $\{(i\theta j) \mid (i\theta j)$ is a conjunct in l$\} \subseteq \{(i\theta j) \mid (i\theta j)$ is a conjunct in 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. In Fig. 8 an example of a trie is given. A trie is in fact a tree, in which the sequences are stored as paths from the root to the leaves. The test whether a sequence is a subsequence of a sequence in the trie can be done very efficiently. We are not going into detail on tries, for a more elaborated work on tries, we refer to [8] and [14].

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 parent 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 is pruned.

Early pruning does not compromise the completeness of the algorithm; i.e. still all frequent expressions are generated. This is due to the monotonicity principle; whenever an expression is frequent, also its generating parents are frequent. By induction we can now conclude that every frequent expression is generated.

In Appendix B we give an example showing the algorithm.

6 Complexity

In this section we give both theoretical and empirical complexity results. For reasons of simplicity we assume in this section that in the input relations, equal rows can appear more than once; i.e. relations are rather bags than sets.

6.1 Theoretical Complexity Results

Definition 11 Let θ be a binary test. An instance of the $FREQ_\theta$-problem is a 3-tuple $(\mathcal{R}, t, k)$. $FREQ_\theta(\mathcal{R}, t, k)$ is true iff there is an expression $e \in \mathcal{E}(\theta, n)$ with $freq(e, \mathcal{R}) \geq t$, and the number of conjuncts in e is at least k. ◁

Proposition 4 *Let θ be a binary test. If θ can be decided in polynomial time, then $FREQ_\theta$ is in NP.*

Proof. Note, that when an expression is frequent, all its sub-expressions are frequent too. Thus, if there is an expression of length at least k, there is also an expression of exactly length k. We can guess an expression of length k. The evaluation of this expression can be done in polynomial time, because θ can be decided in polynomial time, and the number of conjuncts is at most n^2, with n the number of attributes of $\mathcal{R}$. ◁

The previous proposition states that the $FREQ_\theta$-problem can be solved in non-deterministic polynomial time. Whether this is a tight upper bound on the complexity depends on θ. For example, suppose that $a\theta b$ is always false, then $FREQ_\theta$ can obviously be solved in constant time. The next proposition however, states that under reasonable assumptions, $FREQ_\theta$ is NP-complete.

Proposition 5 *If there exist $a, b, c, d \in \mathcal{U}$ such that $a\theta b$, $c\theta d$, $a \not\theta c$, $a \not\theta d$, $b \not\theta c$, and $b \not\theta d$, then $FREQ_\theta$ is NP-complete.*

Proof. We will show that under the this assumption, we can always reduce the following problem to $FREQ_\theta$:

> Given a transaction database $\mathcal{D}$, a threshold t and k, is there an itemset with at least k items that has a frequency of at least t in $\mathcal{D}$?

This problem is well-known to be NP-complete [11]. The reduction works as follows: the input $(\mathcal{D}, t, k)$ of the frequent itemset problem is reduced to a input of the $FREQ_\theta$-problem $(R_\mathcal{D}, t', k)$.
We will now give the construction of $R_\mathcal{D}$. This construction is illustrated in Fig. 9. $R_\mathcal{D}$ is a relation over $A_1, B_1, A_2, B_2, \ldots, A_n, B_n$, with n the number of items in $\mathcal{D}$. For each transaction T, $R_\mathcal{D}$ contains a row with $A_i = a$ for all i, $B_i = b$ if the i-th item is in T, and $B_i = c$ else. In this relation, the expression $A_{i_1}\theta B_{i_1} \wedge A_{i_2}\theta B_{i_2} \wedge \ldots A_{i_m}\theta B_{i_m}$ has frequency t iff the itemset $\{i_1, i_2, \ldots, i_m\}$ has frequency t in $\mathcal{D}$.

There is however still a problem: there will also be frequent expressions that contain conjuncts of the form $A_i\theta B_j$ with $i \neq j$, $A_i\theta A_j$ and $B_i\theta B_j$. Hence, additionally, $R_\mathcal{D}$ contains $|\mathcal{D}| + 1$ times the following rows: for each $1 \leq i \leq n$, $R_\mathcal{D}$ contains r, with $r.A_i = c$, $r.B_i = d$, and $r.A_k = a$, $r.B_k = b$ for all $k \neq i$. In

1	2	3	4	5
0	1	1	1	0
1	1	1	1	0
1	1	1	0	1
1	0	0	0	0
1	1	1	1	0
1	1	0	1	1
1	1	1	1	0
0	0	1	0	1
1	0	0	0	1
1	1	1	1	0

t=0.5
k=3

is reduced to

A_1	B_1	A_2	B_2	A_3	B_3	A_4	B_4	A_5	B_5
1	2	1	1	1	1	1	1	1	2
1	1	1	1	1	1	1	1	1	2
1	1	1	1	1	1	1	2	1	1
1	1	1	2	1	2	1	2	1	2
1	1	1	1	1	1	1	1	1	2
1	1	1	1	1	2	1	1	1	1
1	1	1	1	1	1	1	1	1	2
1	2	1	2	1	1	1	2	1	1
1	1	1	2	1	2	1	2	1	1
1	1	1	1	1	1	1	1	1	2

and 11 times

1	1	2	2	2	2	2	2	2	2
2	2	1	1	2	2	2	2	2	2
2	2	2	2	1	1	2	2	2	2
2	2	2	2	2	2	1	1	2	2
2	2	2	2	2	2	2	2	1	1

with $k' = 6$, and $t' = \frac{60}{65} = 0.307...$

The frequent itemset $\{2, 3, 4\}$ corresponds with the frequent expression $(A_2 = B_2) \wedge (B_2 = A_2) \wedge (A_3 = B_3) \wedge (B_3 = A_3) \wedge (A_4 = B_4) \wedge (B_4 = A_4)$.

Figure 9: Illustration of the construction of $R_{\mathcal{D}}$

these $(|\mathcal{D}| + 1)n$ extra rows, every row satisfies $A_i \theta B_i$, for all i. For every other pair of attributes C, D, there are at least $|\mathcal{D}| + 1$ rows that do not satisfy $C \theta D$, except for $B_i \theta A_i$. If $b \theta a$ and $d \theta c$ are both true, also all $B_i \theta A_i$ are true in the extra rows, otherwise at least $|\mathcal{D}| + 1$ rows do not satisfy $B_i \theta A_i$.

Then, for every expression that only contains conjuncts of the form $A_i \theta B_i$ (or $B_i \theta A_i$ if $b \theta a$ and $d \theta c$ are both true), the number of rows in $R_{\mathcal{D}}$ that satisfies them, is the number of transactions in $\mathcal{D}$ that satisfy them, plus $(|\mathcal{D}| + 1)n$. For every other expression e, the number of rows satisfying e is at most $|\mathcal{D}| + (|\mathcal{D}| + 1)n - (\mathcal{D} + 1) = (|\mathcal{D}| + 1)n - 1$. Therefore, with $t' = \frac{t|\mathcal{D}| + (|\mathcal{D}|+1)n}{(|\mathcal{D}|+1)n + |\mathcal{D}|}$, there is a frequent expression with i (or $2i$ if $b \theta a$ and $d \theta c$ are both true) conjuncts iff there is a frequent itemset of size i and vice verse. So, in the case $b \theta a$ and $d \theta c$ are both true, we choose $k' = 2k$, else $k' = k$. ◁

6.2 Experimental Results

In this section we present some experimental results. We implemented both refinement operators ρ_1 and ρ_2. The source code of both implementations can be obtained at `http://cc-www.uia.ac.be/u/calders/`.

Effectiveness of Pruning In Fig. 10 (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. The calculation of this lower bound is given in Appendix A. In Fig. 10(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.

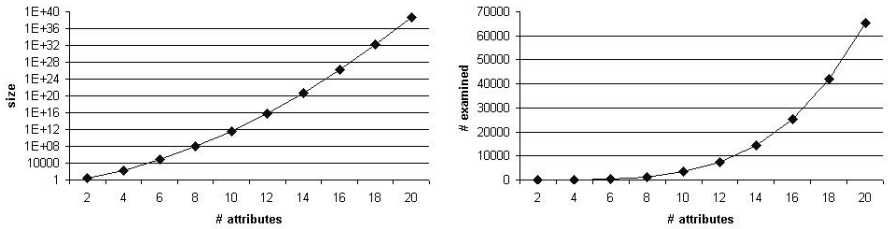

Figure 10: The size of the search space versus the number of expressions that were investigated

Scalability In Fig. 11, 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.

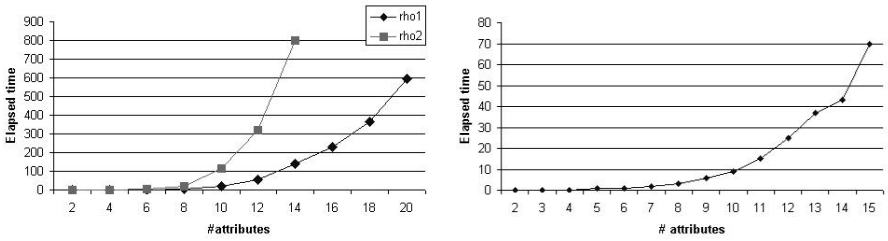

Figure 11: Scalability in the number of attributes

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

References

[1] A. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB*, Santiago, Chile, 1994.

[2] 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

[3] R. Agrawal, H.Manilla, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, chapter 12. 1996.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *ICDT*. IEEE Computer Society, 1995.

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

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

[7] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. VLDB*, Zürich, Switzerland, 1995.

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

[9] Y. Huhtala, J.Kärkkäinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDT*. IEEE Computer Society, 1998.

[10] M. Kantola, H. Mannila, K.-J. Räihä, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7, 1992.

[11] Y.-D. Kwon, R. Nakanishi, M. Ito, M. Nakanishi. Computational Complexity of Finding Meaningful Association Rules. *IEICE Trans. Fundamentals Vol.E82-A No.9 pp.1945-1952*, september 1999

[12] H. Mannila and H. Toivonen. Discovery of frequent episodes in event sequences. In *Data Mining and Knowledge Discovery 1(3)*, 1997.

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

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

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

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

A Approximation of the number of partial orders

Proposition 6

$$\alpha_m^n = 2^{nm} - 1 - \sum_{\substack{1 \le k \le n,\, 1 \le l \le m \\ k+l < n+m}} \binom{n}{k} \binom{m}{l} \alpha_l^k \qquad (1)$$

$$\beta_m^n = 2^{nm} - 1 - \sum_{l=1}^{m-1} \binom{m}{l} \beta_l^n \qquad (2)$$

$$\alpha_m^1 = \alpha_1^n = 1$$

$$\beta_m^1 = \beta_1^n = 1$$

$$\gamma(i_1, \ldots, i_s) = \alpha_{i_2}^{i_1} \beta_{i_3}^{i_2} \beta_{i_4}^{i_3} \ldots \beta_{i_s}^{i_{s-1}} \qquad (3)$$

$$N(n) = \sum_{s=2}^{n} \sum_{\substack{1 \le i_1, \ldots i_s \le n \\ \sum i_j \le n}} \binom{n}{i_1} \binom{n - i_1}{i_2} \cdots \binom{n - i_1 - i_2 - \ldots - i_{s-1}}{i_s} \gamma(i_1, \ldots, i_s) \qquad (4)$$

$N(n)$ is a lower bound for the number of partial orders on a set of n elements.

Proof. We denote the number of binary relations between the set A (with n elements) and B (with m elements) such that each element of A and each element of B participates in the relation, by α_m^n. We clearly have (1), which defines all α_m^n recursively.

We denote the number of relations between the sets A and B such that each element of B participates in the relation by β_m^n. We clearly have (2), which defines β_m^n recursively.

Consider now C, a set with n elements and $\mathcal{C}$ a sequence $C_1, C_2, \ldots, C_s$ of nonempty, disjoint subsets of C. Let C_j have i_j elements. We associate to $\mathcal{C}$ the set of those partial orders where

1. all direct successors of elements in C_{i-1} are in C_i;
2. each element in C_i has at least one predecessor, $2 \le i \le s$;
3. each element of C_1 has at least one direct successor;
4. only elements of $C_1, \ldots, C_s$ are involved in the partial order.

Clearly the number of partial orders associated to $\mathcal{C}$ is (3).

Since the sets of partial orders that are associated to two different sequences are disjoint, we have at least $N(n)$ different partial orders on C. ◁

In table A, the value of $N(n)$ is shown for $n = 1, \ldots, 20$.

Table 3: Values of N(n).

n	$N(n)$
1	1
2	3
3	19
4	195
5	3031
6	67263
7	2086099
8	89224635
9	5254054111
10	426609529863
11	47982981969979
12	7507894696005795
13	1641072554263066471
14	502596525992239961103
15	216218525837808950623459
16	13088716738583188114006475
17	1116532187631668288631416369 11
18	134349872458038183085622069028183
19	22822803527454864652004538948366 2539
20	54764261537847173488740261986903 5943475

B Example run of the algorithm

In this appendix we give a large sample run of the algorithm described in the paper. All steps of the algorithm are discussed, and both refinement operators ρ_1 and ρ_2 are considered.

$\mathcal{R} =$

1	2	3	4
1	2	2	4
1	5	6	2
1	5	1	3
3	5	6	2
2	7	2	4
3	2	4	4
6	2	3	5

, $threshold = \frac{3}{7}$, $\theta =$"$<$"

Initialization

- $candidates = \{\top\}$
- $Output = \{\}$
- $TooLow = \{\}$

Iteration 1

Test $fcan = \{\top\}$, $nfcan = \{\}$, $Output = \{\top\}$, TooLow=$\{\}$

Generate and Prune $candidates = \{ (1 < 2),\ (1 < 3),\ (1 < 4),\ (2 < 1),\ (3 < 1),\ (4 < 1),\ (2 < 3),\ (2 < 4),\ (3 < 2),\ (4 < 2),\ (3 < 4),\ (4 < 3)\}$

Iteration 2

Test $Output = fcan = \{ (1 < 2),\ (1 < 3),\ (1 < 4),\ (2 < 3),\ (2 < 4),\ (4 < 2),\ (3 < 4)\}$
$TooLow = nfcan = \{ (2 < 1),\ (3 < 1),\ (4 < 1),\ (3 < 2),\ (4 < 3)\}$

Generate and Prune $candidates = \{ (1 < 2 \wedge 1 < 3),\ (1 < 2 \wedge 1 < 4),\ (1 < 2 \wedge 4 < 2),\ (1 < 2 \wedge 3 < 4),\ (1 < 3 \wedge 1 < 4),\ (1 < 3 \wedge 2 < 3),\ (1 < 3 \wedge 2 < 4),\ (1 < 3 \wedge 4 < 2),\ (1 < 4 \wedge 2 < 3),\ (1 < 4 \wedge 2 < 4),\ (1 < 4 \wedge 3 < 4),\ (2 < 3 \wedge 2 < 4),\ (2 < 4 \wedge 3 < 4)\}$

Iteration 3

Test $fcan = \{\ (1 < 2 \wedge 1 < 3),\ (1 < 2 \wedge 1 < 4),\ (1 < 2 \wedge 4 < 2),\ (1 < 2 \wedge 3 < 4),\ (1 < 3 \wedge 1 < 4),\ (1 < 3 \wedge 2 < 3),\ (1 < 4 \wedge 3 < 4)\}$
$nfcan = \{\ (1 < 3 \wedge 2 < 4),\ (1 < 3 \wedge 4 < 2),\ (1 < 4 \wedge 2 < 3),\ (1 < 4 \wedge 2 < 4),\ (2 < 3 \wedge 2 < 4),\ (2 < 4 \wedge 3 < 4)\}$

Generate and Prune $candidates = \{\ (1 < 2 \wedge 1 < 3 \wedge 1 < 4),\ (1 < 2 \wedge 1 < 4 \wedge 3 < 4)\}$

Iteration 4

Test $fcan = \{\ (1 < 2 \wedge 1 < 4 \wedge 3 < 4)\}$
$nfcan = \{(1 < 2 \wedge 1 < 3 \wedge 1 < 4)\}$

Generate and Prune $candidates = \{\}$
