Discovering all most specific sentences by randomized algorithms

Dimitrios Gunopulos* Heikki Mannila† Sanjeev Saluja‡

September 13, 1996

Abstract

Data mining can in many instances be viewed as the task of computing a representation of a theory of a model or a database. In this paper we present a randomized algorithm that can be used to compute the representation of a theory in terms of the most specific sentences of that theory. In addition to randomization, the algorithm uses a generalization of the concept of hypergraph transversal. We apply the general algorithm in two ways, for the problem of discovering maximal frequent sets in 0/1 data, and for computing minimal keys in relations. We present some empirical results on the performance of these methods on real data. We also show some complexity theoretic evidence of the hardness of these problems.

1 Introduction

Data mining has recently emerged as an active area of investigation and applications [7]. The goal of data mining can briefly and informally be stated as "development of efficient algorithms for finding useful high-level knowledge from large masses of data". The exact formal meaning of what constitutes "useful high-level knowledge" is determined by particular application, in particular by what type of knowledge one is looking for in the raw data. The design of data mining algorithms typically combines methods and tools from database theory, machine learning, statistics and combinatorial mathematics.

A large part of current research in data mining can be viewed as addressing instances of the following computational problem: given a language, a frequency

*Address: MPI Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: gunopulo@mpi-sb.mpg.de

†Address: University of Helsinki, Department of Computer Science, FIN-00014 Helsinki, Finland. E-mail: Heikki.Mannila@cs.helsinki.fi. Work supported by Alexander von Humbold Stiftung and the Academy of Finland.

‡Address: MPI Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: saluja@mpi-sb.mpg.de

criterion, and a database, find all sentences from the language that are true in the database and satisfy the frequency criterion. Typically, the frequency criterion states that there are sufficiently many instances in the database satisfying the sentence. Examples of scenarios where this formulation works include the discovery of association rules, strong rules, episodes, and keys. Using this *theory extraction formulation* [12, 13, 15] one can formulate general results about the complexity of algorithms for various data mining tasks.

The known algorithms for instances of above mentioned problem typically operate in a bottom-up fashion: starting with the simplest, most general sentences from the language which satisfy the truth and frequency criterion, successively bigger and more specific true sentences are discovered until no bigger and more specific sentences can be found. In this bottom up approach, the knowledge of the smaller true sentences discovered already is used to focus the search for bigger true sentences. In some applications, it is enough to compute only the most specific true sentences satisfying the requirements because they determine the theory uniquely i.e. they capture the information of all the true sentences. In these applications, an alternative feasible approach is to search directly the most specific sentences without going through the less specific true sentences. When the most specific true sentences are large, the number of true sentences which are discovered before discovering the most specific sentences is very large, which makes the above approach very time consuming.

In this paper, we first present a randomized algorithm which efficiently finds a random most specific true sentence. By running the randomized algorithm several times, one can expect to find a sizable collection of most specific true sentences. To extend this collection of most specific sentences to the complete set of most specific true sentences, one must cut down the search space, so that the search focuses only on the undiscovered most specific true sentences. We suggest a method which is applicable to the situations where discovering all most specific true sentences involves finding an anti-chain i.e. a collection of unrelated elements of a partial order. Our method uses the simple fact that if some sets from an anti-chain are known, then every unknown set in the anti-chain must contain a *minimal transversal* of the *complements* of the We also present an algorithm to compute all transversals of a given hypergraph (i.e. a collection of subsets of a finite sets).

Next, we propose to use the randomized algorithm along with the transversal computation algorithm to compute all most specific true sentences. The algorithm alternates between finding some undiscovered random most specific true sentences and finding transversals of the complements of the discovered most specific true sentences, until no new most specific true sentence can be found. We demonstrate this method by applying it to the following problems: (i) computation of all maximal frequent sets of a $\{0,1\}$ matrix for a given threshold (ii) computation of all minimal keys in a database.

Computation of maximal frequent sets is a fundamental data mining problem which is required in discovering association rules [1, 2]. Computation of minimal keys is important for semantic query optimization, which leads to fast query processing

in database systems [14, 11, 5, 17]. It turns out that our algorithms for finding maximal frequent sets and minimal keys find all maximal frequent sets and minimal keys respectively. We point out (Section 9) that whenever the computation involves finding all maximal or minimal elements of an ideal or filter respectively, then an analogous randomized algorithm will work.

Note that the computation of sentences of a theory is an enumeration problem i.e. the computation involves listing combinatorial substructures related with the input. For listing problems, one of the definitions of efficiency is that the running time of algorithm be bounded by a polynomial function of input and output sizes. Such an algorithm is called as an output-polynomial time algorithm. For both problems that we discuss in this paper, output-polynomial algorithms are unknown. In the absence of such provably efficient algorithms, we view our algorithms in the paper as efficient alternatives which can perform well in practice.

The rest of this paper is organized as follows. In Section 2 we present a model of data mining which formally defines the theory extraction problem. In Section 3 we formulate our general algorithm in this setting. Section 4 adapts our algorithm to the well-studied problem of finding maximal frequent sets, and also gives some complexity-theoretic evidence of hardness of this problem. Empirical results on the behavior of the algorithm are given in Section 5.

In Section 6 we adapt the general algorithm of Section 3 to the problem of finding keys of relations; empirical results for this method are given in Section 7. In Section 8, we present the algorithm for computing all minimal transversals that we have used in the algorithms for computing all maximal frequent sets and minimal keys. In Section 9, we discuss the scope of our algorithms and point out some directions of further work.

2 Data mining as theory extraction

The model of knowledge discovery that we consider is the following [12, 15, 13]. Given a database $\mathbf{r}$, a language $\mathcal{L}$ for expressing properties or defining subgroups of the data, and a frequency criterion q for evaluating whether a sentence $\varphi \in \mathcal{L}$ defines a sufficiently large subclass of $\mathbf{r}$. The computational task is to find the theory of $\mathbf{r}$ with respect to $\mathcal{L}$ and q, i.e., the set $Th(\mathcal{L}, \mathbf{r}, q) = \{\varphi \in \mathcal{L} \mid q(\mathbf{r}, \varphi) \text{ is true}\}$.

We are not specifying any satisfaction relation for the sentences of $\mathcal{L}$ in $\mathbf{r}$: this task is taken care of by the frequency criterion q. For some applications, $q(\mathbf{r}, \varphi)$ could mean that φ is true or almost true in $\mathbf{r}$, or that φ defines (in some way) a sufficiently large or otherwise interesting subgroup of $\mathbf{r}$.

Obviously, if $\mathcal{L}$ is infinite and $q(\mathbf{r}, \varphi)$ is satisfied for infinitely many sentences, (an explicit representation of) all of $Th(\mathcal{L}, \mathbf{r}, q)$ cannot be computed feasibly. Therefore for the above formulation to make sense, the language $\mathcal{L}$ has to be defined carefully. In case $\mathcal{L}$ is infinite, there are alternative ways of meaningfully defining feasible computations in terms of dynamic output size, but we do not concern ourselves with these scenarios. In this paper we assume that $\mathcal{L}$ is finite.

3 A randomized algorithm for computing $Th(\mathcal{L}, \mathbf{r}, q)$

We make the following assumption about language $\mathcal{L}$. There is a partial order $\preceq$ on the set of sentences of $\mathcal{L}$ such that q is monotone with respect to $\preceq$, that is, for all $\psi, \theta \in \mathcal{L}$ with $\theta \preceq \psi$ we have: if $q(\mathbf{r}, \psi)$, then $q(\mathbf{r}, \theta)$. [1] Denote by $rank(\psi)$ the *rank* of a sentence $\psi \in \mathcal{L}$, defined as follows. If for no $\theta \in \mathcal{L}$ we have $\theta \prec \psi$, then $rank(\psi) = 0$, otherwise $rank(\psi) = 1 + \max\{rank(\theta) \mid \theta \prec \psi\}$. For $T \subset \mathcal{L}$, let T_i denote the set of the sentences of $\mathcal{L}$ with rank i.

The level-wise algorithm [15] for computing $Th = Th(\mathcal{L}, \mathbf{r}, q)$ proceeds by first computing the set Th_0 consisting of the sentences of rank 0 that are in Th. Then, assuming Th_i is known, it computes a set of *candidates*: sentences ψ with rank $i+1$ such that all θ with $\theta \prec \psi$ are in Th. For each one of these candidates ψ, the algorithm calls the function q to check whether ψ really belongs to Th. This iterative procedure is performed until no more sentences in Th are found.

This level-wise algorithm has been used in various forms in finding association rules, episodes, sequential rules, etc. [2, 3, 16, 15].

The drawback with this algorithm is that it always computes the whole set $Th(\mathcal{L}, \mathbf{r}, q)$, even in the cases where a condensed representation of Th using *most specific sentences* would be useful. Given Th, a sentence $\psi \in Th$ is a *most specific* sentence of Th, if for no $\theta \in Th$ we have $\psi \prec \theta$. Denote by $MTh = MTh(\mathcal{L}, \mathbf{r}, q, \preceq)$ the set of most specific sentences of $Th(\mathcal{L}, \mathbf{r}, q)$ with respect to $\preceq$.

Our general algorithm for computing MTh is based on repeatedly computing random undiscovered most specific sentences in Th. After every computation of one or more random new most specific sentence in MTh, we compute the *minimal-orthogonal-elements (abbr. min-ortho-element)* with respect to the collection of most specific sentences found so far. The *minimal orthogonal elements* are the sentences in $\mathcal{L}$, which are defined using a generalization of the concept of transversals of a hypergraph [6].

A *hypergraph* is collection of subsets (i.e. *hyperedges*) of a finite set (*the vertex set*). A *transversal* of the hypergraph is a minimal set of vertices which intersects every hyperedge of the hypergraph. A *min-ortho-element* with respect to a set S of most specific sentence is a sentence ψ such that it is unrelated to any sentences in S under $\preceq$ and for no sentence $\phi \prec \psi$, this property holds. Note that if there is a sentence γ in MTh which is not in S, then there is a *min-ortho-element* X with respect to S such that $X \preceq \gamma$. Therefore after computation of *min-ortho-elements* the search can be limited to these sentences which are above the *min-ortho-elements* with respect to $\preceq$.

We first give the algorithm to compute a random most specific sentence from Th. Denote $\psi \prec_1 \theta$, if $\psi \prec \theta$ and for no φ we have $\psi \prec \varphi \prec \theta$; in this case, we say that θ is an *immediate specialization* of ψ.

[1] Note that this description is a fairly severe one. For example, a q defined in terms of statistical significance does not satisfy this condition.

Algorithm A_Random_MSS Find a random most specific sentence from Th.

1. $i := 0$.
2. $\psi :=$ **true**.
3. While (there is an immediate specialization θ of ψ such that $q(\mathbf{r}, \theta)$ holds) do: select such a θ randomly and let $\psi := \theta$.
4. Output ψ.

The algorithm assumes that **true** $\in \mathcal{L}$, and proceeds to specialize it successively until a most specific sentence is found. In Step 2, if ψ is initialized with an arbitrary sentence $s \in Th$ instead of "true", then the algorithm will find a random most specific sentence s' such that $s \preceq s'$.

Denote by Algorithm A_Random_MSS(ψ_{init}) the parameterized version of the Algorithm A_Random_MSS, which starts by initializing ψ with the sentence ψ_{init}.

We now give the general algorithm for finding all most specific sentences.

Algorithm All_MSS Finding all most specific sentences in Th.

1. Run Algorithm A_Random_MSS("true"), k_1 times and let S be the set of most specific sentences found.
2. While new most specific sentences are found:
   (a) Compute the set X of all *min-ortho-elements* with respect to S.
   (b) For each sentence $x \in X$:
   Run Algorithm A_Random_MSS(x), k_2 times and add any new most specific sentence found to S.
3. Output S.

The parameters k_1 and k_2 control the number of iterations of the randomized algorithm in Steps 1 and 2b. Since the running time of the algorithm depends on these values, there values are to be chosen suitably depending on the application of the algorithm.

Computing the *min-ortho-elements* is in general computationally non-trivial. In the experiments the randomized algorithm (in Step 1 above) typically produces a good approximation to the collection MTh and so only a few iterations of transversal computation are needed. Additionally, it is useful to notice that the transversal computation does not look at the data, only at elements of $\mathcal{L}$; if the input data is large, a complicated computation on $\mathcal{L}$ can still be much cheaper than just reading the data once.

4 Finding frequent sets using a randomized algorithm

In this section, we discuss how to adapt the algorithms of the previous section to find maximal frequent sets of a {0,1} matrix and threshold value σ. We first discuss association rules and how frequent sets arise in computation of association rules.

Given a 0-1 relation r with attributes R (i.e. a $\{0,1\}$ matrix, whose set of columns is R), an association rule is an expression $X \Rightarrow B$, where $X \subseteq R$ and $B \in R$. The intuitive meaning of such a rule is that if a row has a 1 in all attributes of X, then it tends also to have a 1 in column B.

An association rule has two values *support* and *confidence* associated with it, which are defined as follows. Given a set X of attributes of a relation r, *frequency* f(X,r) of X in r is the number of rows in r for which all attributes in X have a 1. The *support* of X in r is the fraction of these rows among all the rows of r. Given a rule $X \Rightarrow B$, the support of the rule is defined to be the support of $X \cup \{B\}$. The confidence of the rule is the fraction $f(X \cup \{B\}, r)/f(X, r)$.

The problem of mining association rules is to compute all association rules in a 0-1 relation such that the support of a rule is at least σ and the confidence at least γ. The first step in computing such association rules is to find all subsets of attributes (i.e. columns), whose support is at least σ. Such subsets are called the *frequent sets* of the relation r with threshold σ (or σ-frequent sets). A *maximal σ-frequent set* X of relation r is an σ-frequent set of r such that no proper superset of X is an σ-frequent set of r. The collection of all σ-frequent sets (resp. maximal σ-frequent sets) of relation r is denoted by $Fr(r, \sigma)$ (resp. $MFr(r, \sigma)$). [2] Note that to identify all σ-frequent sets, it is enough to compute $MFr(r, \sigma)$ because every frequent set is a subset of some maximal frequent set and conversely every subset of a maximal frequent set is a frequent set.

The computational problem that we study in this section is the following.

Problem 1 Given a 0-1 relation r over attributes R, and a support value $\sigma \in [0, 1]$, find all maximal σ-frequent sets of r.

We start by presenting two results which show the computational hardness of the above problem.

Theorem 2 The problem of finding the number of σ-frequent sets of a given 0-1 relation r and a threshold $\sigma \in [0, 1]$ is #P-hard.

Proof: We show a polynomial time reduction from the problem of computing the number of satisfying assignments of a monotone-2CNF formula to the problem of computing the number of frequent sets. Since the problem of computing the number of satisfying assignment of monotone-2CNF formulae is known to be #P-hard [19], this will show the #P-hardness of the frequent set counting problem.

[2] The collection of all maximal s-frequent sets of a matrix and threshold s, is an ideal of the boolean lattice over the set of columns (attributes) of the matrix.

A monotone-2CNF formula is a boolean formula in conjunctive normal form in which every clause has at most two literals and every literal is unnegated. Given a monotone-2CNF formula f with m clauses and n variables, construct an $m \times n$ {0,1} matrix M as follows. Value of $M_{i,j}$ is 0 iff the j^{th} variable is present in i^{th} clause. An assignment of variables falsifies f iff the set of columns corresponding to variables with value 1, form a frequent set of M with threshold $\frac{1}{n}$. Therefore, the number of frequent sets of M with threshold $\frac{1}{n}$ is (2^n - the number of satisfying assignments of f). This completes the reduction. ∎

Note that the above result still does not rule out the possibility of an output polynomial algorithm for computing all maximal frequent sets.

The next result shows the hardness of computing a large frequent set.

Theorem 3 The problem of deciding if there is a maximal σ-frequent set with at least t attributes for a given 0-1 relation r, and a threshold $\sigma \in [0, 1]$, is NP-complete.

Proof: It is easily seen that the problem is in NP. To show the NP-hardness, we show a polynomial time reduction from the Balanced Bipartite Clique problem to the above problem. Since the Balanced Bipartite Clique is known to be NP-hard, the result will follow ([8]).

Given a bipartite graph $G = (V_1, V_2, E)$, a balanced clique of size k is a complete bipartite graph with exactly k vertices from each of V_1 and V_2. The Balanced Bipartite Clique problem is, given a bipartite graph G and a positive integer k, check if there exist a balanced bipartite clique of size k.

Given a bipartite graph G with and a positive integer k, let n_1 and n_2 be the number of vertices in V_1 and V_2 respectively. Define an $n_1 \times n_2$ {0,1} matrix M as follows. $M_{i,j}$ is 1 iff i^{th} vertex of V_1 is connected to the j^{th} of V_2. Then there is a bipartite clique of size k in G iff there is a frequent set of M of size at least k with threshold $\frac{k}{n_1}$. ∎

The above theorem rules out the possibility of an efficient algorithm which outputs the maximal frequent sets in the decreasing order of their size.

We now discuss a refinement of the algorithm of Section 3 for computing all maximal frequent sets. To use the framework of Section 3, we define $\mathcal{L} = \{X \mid X \subseteq R\}$, and let $q(r, X)$ be true iff $s(X, r) \geq \sigma$. Next, the relation $\preceq$ is defined by $X \preceq Y$ iff $X \subseteq Y$; it is easy to see that the monotonicity condition holds. We also have $X \preceq_1 Y$ iff $Y = X \cup \{A\}$ for some $A \in R$. A most specific sentence corresponds to a maximal frequent set.

A useful way to think about the maximal σ-frequent sets problem is the lattice that is formed by the subsets of R. The level i of the lattice includes all subsets of size i, and two subsets are connected if they are on consecutive levels and one is the subset of the other (see Figure 1.) Note that the collection of all maximal σ-frequent sets of a matrix and threshold σ, is an ideal of the boolean lattice over the set of columns (attributes) of the matrix. The lattice view makes also the drawbacks of the level-wise algorithm evident: it can be that Fr is large, but MFr is quite small.

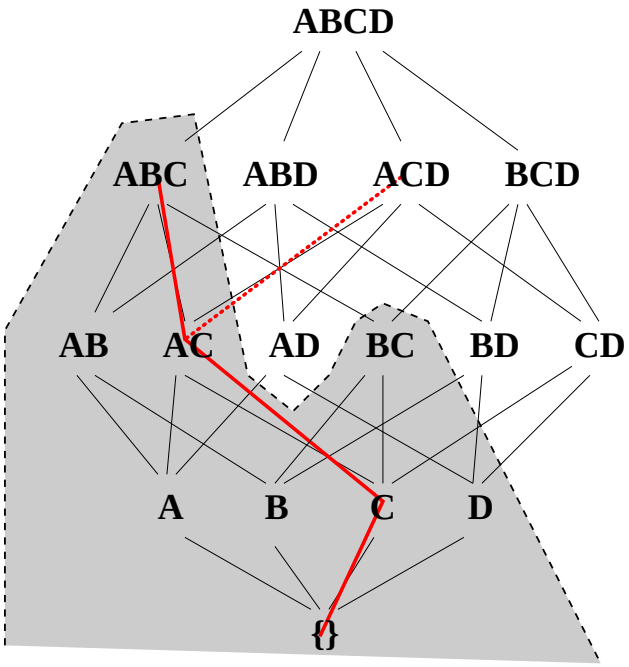

Figure 1: A relation with four attributes. The shaded area represents the σ-frequent sets. The solid line represents a run of the algorithm for the permutation C, A, D, B.

To apply the general algorithm (Algorithm A_Random_MSS), we can use the lattice structure efficiently: the process can be seen as a random walk in the lattice. Given X, in order to select a random sentence/set Y such that $X \preceq_1 Y$ the only thing we have to do is to get a random element $A \in R \setminus X$ and let $Y = X \cup \{A\}$.

Once a collection of maximal σ-frequent sets is found, any new maximal σ-frequent set cannot be subset or superset of a known maximal σ-frequent set. It follows that any new maximal σ-frequent set must include a set of attributes that is not a subset of any of the maximal σ-frequent sets found so far. Similarly for any new maximal σ-frequent set, there must be set of attributes such that it intersects every known maximal σ-frequent set and does not intersect the the new maximal σ-frequent set. We can express these conditions more succinctly using the concept of *minimal transversal* of a hypergraph. A *minimal* transversal of a hypergraph is a transversal of the hypergraph, such that no proper subset of it is a transversal. Therefore if we view a given collection C of maximal σ-frequent sets as a hypergraph, then for any new maximal σ-frequent set F, the following two conditions hold:
(i) F is a transversal of the hypergraph whose edges are the complements of all subsets in C.
(ii) Complement of F is a transversal of C.

The general problem of finding a set F satisfying the above two conditions for a given collection C of subsets is a non-trivial problem. In fact, it follows from the co-NP hardness of the Hypergraph Saturation problem ([6]) that this problem is NP-hard. Thus we cannot hope to find a set F satisfying conditions (i) and (ii). Rather

we note that given a collection C of maximal σ-frequent sets, any new maximal σ-frequent set must contain a minimal transversal of the hypergraph whose edges are the complements of subsets in C. Therefore to discover new maximal σ-frequent set, an approach is to start with a minimal transversal of the above type and extend it to a superset which is a maximal σ-frequent set. If every minimal transversal of the above type is considered for this extension, then we are sure that every new maximal σ-frequent set has a nonzero chance of being discovered.

We now present the algorithm in detail. First we give the algorithm A_Random_MFS, which finds a single random maximal σ-frequent set containing a given set S of attributes. This algorithm corresponds to the parameterized version of the algorithm A_Random_MSS.

Algorithm A_Random_MFS(S) Given a {0,1} matrix M with attributes $R = \{A_1, \ldots, A_{|R|}\}$ and n tuples (rows), a threshold σ and the set S of attributes $\{A_{S_1}, ..., A_{S_l}\}$; find a maximal σ-frequent set F containing all the attributes in S.

1. Find a permutation p of $(1, \ldots, |R|)$ such that for $i \leq |S|$, $p(i) = S_i$, and for $i > |S|$, p is a random permutation of the atributes in the set $R \setminus S$.
2. Set $X = \emptyset$.
3. For $i = 1$ to $|R|$:
   (a) If $X \cup \{A_{p(i)}\}$ is a σ-frequent set, add $A_{p(i)}$ to X.
4. Return X

The following theorem shows the basic properties of the algorithm.

Theorem 4 Let S be a σ-frequent set of relation r. Then the algorithm A_Random_MFS(S) finds the lexicographically first (according to the ordering given by p) maximal σ-frequent set containing attributes in S. Further its time complexity is $O(|r|)$.

Proof: The basic operation of the algorithm is to add a new attribute in the σ-frequent set X. We keep the set of rows $\alpha(X, r)$ that support X as a vector $s = (s_1, \ldots, s_m)$. When attribute R_i is considered, we take the intersection of s and the i-th column of r. This is the support of the set $X \cup R_i$. This process takes $O(m)$ time, so the total running time of the algorithm is $O(m|R|) = O(|r|)$, linear to the size of the relation r.

Note that with respect to a given permutation, a maximal frequent set F_1 is lexicographically smaller than another maximal frequent set F_2, if the smallest attribute (w.r.t. the order of attributes defined using permutation) in the symmetric difference of F_1 and F_2 is in F_1.

It is clear that the output set X is a maximal σ-frequent set. Assume that it is not the lexicographically first maximal σ-frequent set with respect to the ordering

p that contains S. S has to be a frequent set itself, and all the attributes of S are in the beginning of p, they will all be included in X is Step 3. Thereafter, the algorithm will add greedily into X attributes in the order given by p. Let $LF = \{R_{LF_1}, \ldots, R_{LF_k}\}$ be the lexicographically first maximal σ-frequent set with the attributes sorted according to p, and let P_i be the first attribute that is included to LF but not X. But the set $\{R_{LF_1}, \ldots, R_i\}$ is a frequent set, and therefore the algorithm would add P_i to X when it was considered. It follows that at the end of the algorithm F will represent the lexicographically smallest maximal σ-frequent set containing S. ∎

We recall the property that we mentioned earlier about maximal frequent sets that are outside a given collection of maximal frequent sets.

Lemma 5 Let C be a collection of maximal σ-frequent sets of a relation, and F be a maximal σ-frequent set not in C. Then there exists a minimal transversal T of the hypergraph defined by the complements of the sets in C such that $T \subseteq F$.

Proof: A transversal of a hypergraph $G = (V, E)$ is a set of vertices that intersects all the edges of the hypergraph. A minimal transversal is a minimal such set.

Any new maximal σ-frequent set F cannot be a superset or a subset of an existing maximal σ-frequent set. F must therefore intersect the complements of all the complements of the sets in $MFS(r, \sigma)$. This means it must intersect all the edges of $G_{R,S}$, and so it must be a transversal. ∎

We now give the final algorithm which uses algorithm A_Random_MFS to find all maximal frequent sets. After finding some of the maximal frequent sets, it computes all the minimal transversals of the hypergraph as defined in the lemma, to focus the search on undiscovered maximal frequent sets. In the algorithm below, we have omitted the details of how transversals are computed. We discuss it separately in Section 8.

Algorithm All_MFS Given a {0,1} relation r in the form of an $n \times m$ matrix M and a threshold σ, find all maximal σ-frequent sets. Parameters k_1, k_2 are positive integers.

1. Preprocess the matrix to remove all the columns (i.e. attributes) in which the number of 1's is less than σn.

2. Run algorithm A_Random_MFS(ϕ) k_1 times and let C be the set of maximal σ-frequent sets discovered in these runs.

3. While new maximal frequent sets are found:

   (a) Compute the set X of all minimal transversals of the hypergraph defined by complements of sets in C.

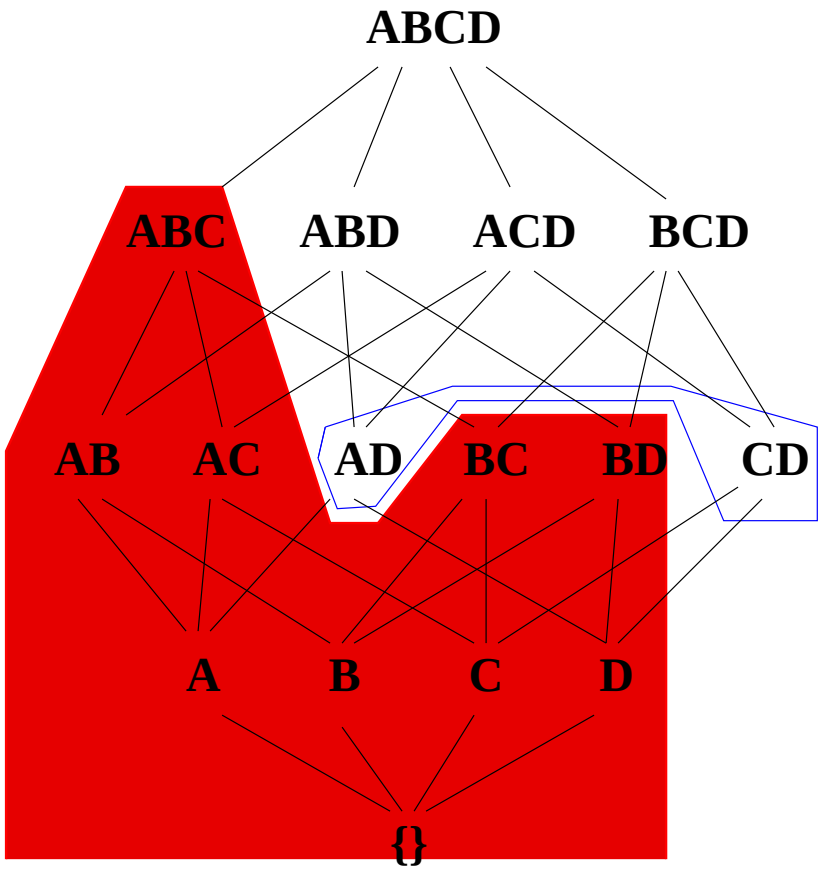

Figure 2: The lower shaded area represents the known σ-frequent sets. Any new maximal σ-frequent set must be a superset of $\{A, D\}$ or $\{C, D\}$.

(b) For each $x \in X$:
Run algorithm A_Random_MFS(x) k_2 times and add any new maximal frequent set found to C.

4. Output S.

It is somewhat surprising that even though the search steps 2 and 3b are randomized, the algorithm for all positive integer values of k_1, k_2 actually finds *all* maximal σ-frequent sets before stopping.

Theorem 6 The algorithm All_MFS finds all maximal σ-frequent sets of the input matrix M.

Proof: After the preprocessing in Step 1, each remaining attribute in the matrix is a frequent set. Since k_1 is positive, Step 2 results in a nonempty collection S of maximal σ-frequent sets. Let S be the (nonempty) collection of maximal σ-frequent sets, which are output in Step 4 at the end of the algorithm. Since the algorithm exited the while loop, it must be that in the last iteration of the while loop, S remained unchanged. Equivalently, for no transversal x of the complements of sets

in S, any maximal σ-frequent set containing x was found in Step 3b. Suppose to the contrary, there is a maximal σ-frequent set F, which is not in S. Then by Lemma 5, there is a minimal transversal (say x) of the complements of sets in S, such that F contains x. Therefore x must be a σ-frequent set. So in the iteration of the for loop at Step 3b, corresponding to x, algorithm A_Random_MFS(x) must have found at least one new maximal frequent set (Theorem 4). This contradicts the claim made earlier that S was unchanged in the last iteration of the while loop. Therefore every maximal σ-frequent set must be in S. ∎

5 Some experimental results for finding maximal frequent sets

We have implemented the algorithm A_Random_MFS, and we used the implementation to find maximal frequent sets in real data sets taken from the University of Helsinki. In these data sets each column represents a course offered, and the rows represent students. A given column has a 1 for each student that took this course and 0 for the rest. We have used two different threshold values, and we try to determine the rate at which the probabilistic algorithm finds new maximal frequent sets. We compare our results with the output of the level-wise algorithm ([1, 2]).

The preliminary results of our experiments are summarized in the following tables.

Matrix Size	σ (no. of rows)	Runs	$MFSs$ found	$MFSs$ present	Time (sec)
2670×20	100	500	78	93	15
2670×20	100	1000	86	93	26
2670×20	100	2000	88	93	50
2670×20	100	4000	89	93	99
2670×20	400	100	23	23	5
2670×20	400	500	23	23	14
2836×129	100	500	178	315	56
2836×129	100	1000	244	315	108
2836×129	100	2000	283	315	208
2836×129	100	4000	303	315	409
2836×129	400	100	27	27	18
2836×129	400	500	27	27	64

Matrix Size	s (no. of rows)	$MFSs$ found	Time (sec)
2670×20	100	93	355
2670×129	400	23	6
2836×20	100	315	1512
2836×129	400	27	10

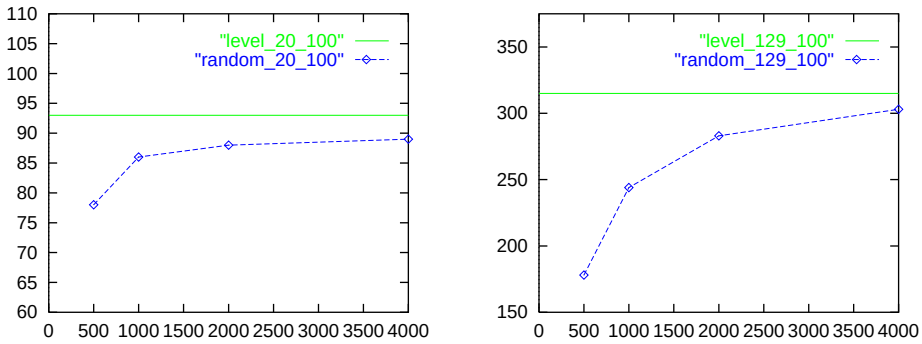

Figure 3: These two graphs plot the number of MFS found after running the algorithm from 500 to 4000 times on the smaller dataset (left) and on the larger dataset (right).

In the first table we present the runs of the randomized algorithm. The two datasets have sizes of 2670×20 and 2836×129 respectively, and the threshold value was set to 100 and 400 rows. We run the algorithm for 500 to 4000 times before colecting the different maximal σ-frequent sets that had been found so far. The number of different maximal σ-frequent sets found is shown in the column $MFSs$ found. The next column shows the total number of maximal σ-frequent sets, as reported by the level-wise algorithm. In the second table we tabulate the results of the level-wise algorithm runs on the same datasets.

The implementation of our algorithm is in C++, and the running times for both algorithms were measured on a SPARCstation 5.

The experiments show that the randomized algorithm finds a big fraction of all the maximal frequent sets while the number of iterations is only about 5 times the total number of maximal frequent sets. In addition the randomized algorithm clearly outperforms the level-wise algorithm as long as the size of the maximal σ-frequent sets is relatively large. In our datasets this happens for a threshold value of 100.

However, as the number of iterations increase the number of discovered maximal frequent sets does not increase in proportion, but "levels off". By observing the datasets, we also noticed that the level-wise algorithm performs equivalently or slightly better when the size of maximal frequent sets are small.

Our observations suggest that by increasing the number of runs to a very large number, the advantage of finding more MFS is lost in the increase of the running time. So a better alternative can be to run the randomized algorithm a fixed number of times and then use the transversal computation to focus the search. We have implemented the algorithm for transversal computation and included it in an implementation of the algorithm All_MFS. We are currently testing it with more examples to see how much the computation of transversals help in speeding up the computation of all maximal frequent sets.

6 Finding Minimal Keys in Databases

In this section we discuss the computational problem of finding all minimal keys of a database and propose an algorithm for the problem based on the general algorithm of Section 3. We begin by defining what we mean by keys and describe an application in which it is useful to find all minimal keys. We view a relational database r as a matrix whose columns correspond to fields and rows correspond to records. Let R denote the set of all fields (i.e. columns of the matrix). Then a set $X \subseteq R$ is a *key* of r, if no two rows of r agree on every attribute in X. A *minimal key* is a key such that no proper subset of it is a key. Note that every key must contain some minimal key and conversely every superset of a minimal key is a key. Therefore the collection of all minimal keys of a database is a succinct representation of the set of all keys of the database. Note the distinction between our definition of key and the more standard definition of (primary) key of a database [18]. A (primary) key is a key (w.r.t. our definition) of the database throughout the life of the database and is maintained so by the database manager. However an arbitrary key by our definition, may be so at current state of the database and may not exist to be so after an update of the database. The computational problem that we consider here is the following:

Problem 7 Given a database, compute all minimal keys that exist currently. □

As has been discussed in [4], the knowledge of all minimal keys existing currently in the database can help in semantic query optimization i.e. in the process by which a database manager substitutes a computationally expensive query by a semantically equivalent query which can be processed much faster.

Before we discuss our algorithms for this problem, we give two results which show its computational hardness.

Theorem 8 The problem of finding the number of all keys of a given database is #P-hard.

Proof: We prove the result in two steps. First we show a polynomial time reduction from the problem of computing the number of satisfying assignments of a monotone-2CNF formula to the problem of computing the number of set-covers of a family of sets. Then we show a polynomial time reduction from the problem of computing the number of set covers of family of sets to the problem of computing the number of keys of a database. Since the problem of computing the number of satisfying assignments of a monotone-2CNF formula is #P-hard [19], this will imply the #P-hardness of the problem of computing the number of keys of a database.

Recall that given a family of sets each of which is subset of a finite *universe* set, a *set cover* is a collection of sets from the family such that every element of the universe is in at least one of the sets in the collection. Given a monotone-2CNF formula with m clauses and n variables, construct a family of n sets $S_1, ..., S_n$ each of which is a subset of the set {1,2,...,m}, as follows. The set S_i contains j iff i^{th}

variable is present in j^{th} clause. It is easily seen that a satisfying assignment of the monotone-2CNF formula corresponds to a unique set cover of the family of sets and vice versa, by picking S_i in the set cover iff the i^{th} variable has value 1 in the assignment. Therefore the number of satisfying assignments of the monotone-2CNF formula is exactly the number of set covers of the family of sets. This completes the first reduction.

We now discuss the second reduction. Given a family of sets $S_1, ..., S_n$ each of which is a subset of the universe set {1,2,...,m}, construct a relational database as follows. The database has m fields $f_1, ..., f_m$ and $n+1$ records $R_0, ..., R_n$. The record R_0 will have value 0 in every field. For $1 \leq i \leq n$ and $1 \leq j \leq m$, the field f_j of record R_i will have value i if element i is present in the set S_j otherwise it will have value 0. Note that a collection $S_{i_1}, S_{i_2}, ..., S_{i_c}$ (for some c) of sets from the family will be a set cover iff the collection of fields $f_{i_1}, ...f_{i_c}$ is a key of the database. Therefore the number of set covers of the given family of sets is same as the number of keys of the database. This completes the second reduction and the proof of the theorem. ∎

The following theorem shows that counting the number of minimal keys is not easier than counting the number of all keys.

Theorem 9 The problem of finding the number of all minimal keys of a given database is #P-hard. [3]

Proof: Once again we show two polynomial time reductions. The first reduction is from the problem of computing the number of minimal vertex covers of a graph to the problem of computing the number of minimal set covers of a family of sets. The second reduction is from the problem of computing the number of minimal set covers of a family of sets to the problem of computing the number of minimal keys of the database. Since the problem of computing the number of minimal vertex covers of a graph is known to be #P hard [19], this will imply the result.

Recall that a vertex cover of a graph G is a set of vertices of G such that every edge of G is incident on at least one vertex in the set. Given a graph G with n vertices and m edges, define a family of sets $S_1, ..., S_n$ each of which is a subset of the set {1,2,...,m}, as follows. The set S_i has element j iff the j^{th} edge of the graph is incident on the i^{th} vertex. Note that a collection of sets $S_{i_1}, ..., S_{i_c}$ (for some c) from the family is a minimal set cover iff the set of vertices $\{i_1, ..., i_c\}$ is a minimal vertex cover of G. Therefore the number of minimal vertex covers of G is same as the number of minimal set covers of the family. This completes the first reduction.

For the second reduction, we use the same reduction which was used as second reduction in the proof of Theorem 8. With respect to the reduction, note that a

[3] In another paper [9], we discuss the related problem of finding the smallest key in a database. We give a simple and efficient (polynomial time) approximation algorithm which finds a key whose size is provably at most $O(log(n))$ times the size of the smallest key. Further we show that, it is NP-hard to obtain a polynomial time approximation to the problem within a factor which is asymptotically better than $O(log(n))$.

collection $S_{i_1}, ..., S_{i_c}$ is a *minimal* set cover of the family iff the set of fields $\{f_{i_1}, ..., f_{i_c}$ is a *minimal* key of the database. Therefore the number of minimal set covers of the family is same as the number of minimal keys of the database. ∎

We now discuss our algorithm for discovering all minimal keys of a database. To keep an analogy with the problem of discovering maximal frequent sets, we will use the notion of an *anti-key*. An *anti-key* in a database is a set of fields which is complement of some key of the database. A maximal anti-key is an anti-key such that no proper superset of it is an anti-key. Note that a set of fields is a maximal anti-key iff its complement is a minimal key. Therefore the problem of finding all minimal keys of a database is equivalent to the problem of finding all maximal anti-keys of the database. To keep presentation analogous to maximal frequent sets, we will henceforth in this section talk only of the problem of finding all maximal anti-keys of a database. Note that the collection of all maximal anti-keys of a database forms an ideal of the boolean lattice over the fields of the database.

We first present an algorithm for finding a random maximal anti-key containing a given set of fields.

Algorithm A_Random_MAK(S) Given a database in the form of an $n \times m$ matrix M and a set $S = \{f_{i_1}, ..., f_{i_s}\}$ of fields of the database, find a random maximal anti-key which contains all the fields in the set, provided there exists one.

1. Pick a random permutation of the set $\{1, 2, ..., m\}$. Set m pointers to the columns (i.e. fields) of the matrix according to the permutation so that we can assume without loss of generality that the columns of the matrix are in the order defined by the permutation.

2. Compute right_to_left profile matrix $RL_{n \times m}$ as follows:

   (a) Consider the m^{th} column of M. Relabel the values in this column with consecutive positive integers starting from 1 so that identical values are labeled with the same integer and different values are labeled with distinct integers. For all i, define $RL_{i,m}$ to be the integer labeling the value in $M_{i,m}$.

   (b) For $j = m - 1$ to 1:
   Consider the n pairs defined by the values in j^{th} column of M and $(j + 1)^{th}$ column of RL. Relabel the pairs with consecutive positive integers starting from 1 so that identical value pairs are labeled with the same integer and different value pairs are labeled with distinct integers. For all i, define $RL_{i,j}$ to be the integer labeling the pair in $(M_{i,j}, RL_{i,(j+1)})$.

3. Initialize the left_to_right profile array $LR_{n \times 1}$ to be all 0's. Initialize A to be empty set.

4. For $j = 2$ to m:

(a) Consider the n pairs defined by values in LR and the j^{th} column of RL. Label all the pairs with consecutive positive integers starting from 1 so that identical pairs are labeled with the same integer and different pairs are labeled with distinct integers.

(b) If the labeling uses all integers from $\{1, 2, ..., n\}$ then $A = A \cup \{j\}$ else

i. Consider the n pairs defined by the entries in LR and the $(j-1)^{th}$ column of M. Label the pairs with consecutive positive integers so that identical pairs are labeled with the same integer and different pairs are labeled with distinct integers. For all i Update the value of $LR_{i,1}$ to be the integer labeling the i^{th} pair.

ii. If the labeling uses all integers from $\{1, ..., n\}$ then $A = A \cup \{j, j+1, ..., m\}$ and go to Step 5.

5. For $k = 1$ to $|S|$: If $f_k \notin A$ then Stop.

6. Output A.

We now claim that the above algorithm outputs the lexicographically first maximal anti-key with respect to the random permutation containing the fields in the input set. First we point out some properties of the algorithm.

Observation: Let j be an integer from {1,2,...,m}. Consider the n tuples formed by taking the projection of the database with respect to the columns {j,j+1,...,m}. Then the j^{th} column of RL represents the distinctness of these n tuples i.e. $RL_{i_1,j}$ and $RL_{i_2,j}$ are different iff the i_1^{th} and i_2^{th} tuples are distinct. This follows by a simple induction on j.

Observation: At the end of any iteration of the for loop (i.e. just before Step 5), the array LR represents the distinctness of the n tuples formed by columns in A. This follows by a simple induction on the loop variable j.

Theorem 10 For a given set S of fields, suppose there is an anti-key which contains all the fields in S. Then the algorithm A_Random_MAK outputs the lex-smallest maximal anti-key with respect to the random permutation and which contains all the fields in S. Further, assuming that the time to access any field of any record is constant, the running time of the algorithm is $O(nm)$.

Proof: Recall that with respect to a given permutation, an antikey K_1 is lexicographically smaller than another antikey K_2, if the smallest field (w.r.t. the order of fields defined using permutation) in the symmetric difference of K_1 and K_2 is in K_1. Note that Step 4 of the algorithm maintains the invariant that the set of fields which are not in A, form a key of the database (note that the invariant is true before the beginning of the loop because A is empty and the set of all fields is certainly a key of the database). Further while scanning the fields in the order of the permutation, it "greedily" puts the fields in the set A i.e. if the set$\{1, 2, ..., m\} - A - \{j\}$ is a key then j is also put in A. Therefore at the end of the for loop, the set A is the lex-smallest maximal anti-key.

For analyzing the time complexity, note that the Step 2 makes one pass of the whole database and hence need $O(nm)$ time. Similarly Step 4 makes one pass of the database. Other steps take $O(m)$or $O(n)$ time. Therefore the total time complexity of the algorithm is $O(nm)$. ∎

We now give the complete algorithm for finding all maximal antikeys, which is analogous to the algorithm for finding all maximal frequent sets. First we point out that an analogue of the Lemma 5 holds also for the case of maximal anti-keys.

Lemma 11 Given a collection C of maximal anti-keys of a database, let K be a maximal anti-key not in C. Then there exists a minimal transversal T of the hypergraph defined by the complements of the sets in C such that $T \subseteq K$.

Proof: The proof follows the proof of Lemma 5, and is omitted here. ∎

In the algorithm All_MAK, once again, we ignore the details of how to find all minimal transversals of a hypergraph and postpone them to next section.

Algorithm All_MAK Given a relational database in the form of $n \times m$ matrix M and find all maximal anti-keys. Parameters k_1, k_2, which are positive integers.

1. Run algorithm A_Random_MAK(ϕ) k_1 times and let C be the set of maximal anti-keys discovered in these runs.
2. While new antikeys are being found:
   (a) Compute the set X of all minimal transversals of the hypergraph defined by complements of subsets in C.
   (b) For each $x \in X$: Run algorithm A_Random_MAK(X) k_2 times and add any new maximal anti-key found to C.
3. Output C.

We can now claim the following theorem.

Theorem 12 The algorithm All_MAK finds all maximal anti-keys (and hence minimal keys) of the input database.

Proof The proof of the theorem follows the proof of Theorem 6 and is omitted here. ∎

7 Some experimental results on finding keys

In this section we present some experimental results obtained from an earlier implementation of a somewhat different algorithm to compute all keys. This algorithm uses the same general scheme, but attempts to compute minimal keys directly, instead of computing maximal antikeys first.

We implemented this algorithm, A_Random_K in C++. To test the algorithm, we used two different relations which have a lot of keys. We are interested in finding how many keys the randomized algorithm Find-Key can discover before we have to perform the expensive traversal computation. The first table shows the results of these experiments.

Matrix Size	Runs	MK found	MK present	Time (sec)
83×12	500	57	58	7.2
83×12	1000	58	58	14.1
128×22	1000	417	1252	30
129×22	2000	588	1252	60.5

We also implemented the algorithm All_K and tested it on the above data sets. The following table summarizes the experimental results.

Matrix size	Number of keys found	Number of keys present	Time (sec)
58×12	58	58	142.9
128×22	1252	1252	21591.2

We remark that the second test case is an exceptionally complex one in terms of the size and structure of minimal keys of the input relation. We are planning more experiments to see the performance for large inputs and with respect to the level-wise algorithm.

8 Computing all minimal transversals of a hypergraph

The general problem of finding all minimal transversals of a hypergraph in output polynomial time is still an open problem. It corresponds to computing all maximal independent sets of a hypergraph because every minimal transversal is complement of a maximal independent set and vice versa. While it is known how to compute all maximal independent sets of a *graph* in output polynomial time, the corresponding problem for hypergraphs seems notoriously difficult [10].

In this section, we propose a heuristic for computing all minima transversals of a hypergraph. We have used this heuristic to algorithm in the implementation of the algorithms All_MFS and All_MAK.

In the algorithm below, we assume that the hypergraph with m hyperedges and n vertices (i.e. m subsets of {1,...,n}) is given in the form of an $n \times m$ {0,1} matrix, where the $(i,j)^{th}$ entry is 1 iff vertex i is present in the edge j.

Algorithm All_Minimal_Transversals Given a hypergraph in the form of an $n \times m$ {0,1} matrix M, compute all minimal transversals of it.

1. Compute the lookahead matrix $LM_{n \times m}$ as follows:

   (a) For i = 1 to m and j = n to 1:
   Set value of $LM_{i,j}$ to be the column number of the leftmost 1 among $\{M_{i \times j}, M_{i \times (j+1)}, ..., M_{i \times n}\}$.

2. For j = 1 to n:

   (a) Initialize boolean array $covered_edges_{m \times 1}$ to all 0's. Initialize boolean array $transversal_{1 \times n}$ to all 0's.

   (b) Call Compute_Transversals$(j, 0, covered_edges, transversal)$.

The procedure Compute_Transversals extends the current partial transversal stored in the input boolean array *transversal* by picking element j and those bigger than j. A variable *count* gives the number of edges already covered by the current partial transversal in the array *transversal*. If the value of *count* is equal to m then all edges are covered, and the array *transversal* stores a transversal of the hypergraph. We output this transversal if it is also minimal.

Procedure Compute_Transversals$(j, count, covered_edges, transversal)$

1. If $count = m$ then: check if the transversal represented by array *transversal* is minimal and if yes, then output it and return else simply return.

2. Let list L contain every value i such that $covered_edge_{i,1}$ is 0.

3. For every value x in L:

   (a) Define *new_element* to be the value in $LM_{x,i}$.

   (b) Include *new_element* in the transversal and update the value of the array *covered_edges*.

   (c) Call Compute_Transversals$(new_element, count, covered_edge, transversal)$

The Step 2 picks out all those elements to the right of j (elements whose index is larger than j) which cover some uncovered edge. Each of these elements is used to extend the current transversal and then the same process is repeated recursively until all edges are covered. Once all the edges are covered i.e. the *count* is m, then

the transversal in the array *transversal* is printed provided it is a minimal transversal. The check for minimality is done by checking that no subset of it with one fewer element, is also a transversal. We remark that this can be done efficiently by computing the left_to_right profile matrix and then making a right_to_left pass of the columns, similar to the algorithm A_Random_MAK. We now claim the correctness of the above algorithm.

Theorem 13 The algorithm All_Transversals computes all minimal transversals of the hypergraph given by matrix M.

Proof: Consider any arbitrary minimal transversal T of the hypergraph. Let the index set of columns corresponding to it be $\{c_1, ..., c_t\}$ such that $c_1 < ... < c_t$. We claim that the transversal will be output in the call to procedure Compute_Transversal with $j = c_1$. Note that after picking c_1 in the partial transversal, c_2 will be in the list picked at Step 2. This is because by minimality of T, there will be some edge covered by c_2 which is not covered by c_1. So one of the iteration of For loop (Step 3) will have (c_1, c_2) as partial transversal. By a similar argument c_3 will be in the list of elements picked at Step 2 in the recursive call to procedure Compute_Transversal with (c_1, c_2) as the partial transversal. Continuing above argument, it follows that there will be a recursive call to Compute_Transversals in which the partial transversal will be exactly $(c_1, ..., c_t)$. Since the value of *count* in this call will be m, this call will output T. ∎

9 Discussion

We have given a randomized algorithm for computing the representation of a theory in terms of its most specific sentences. We have also proposed an approach based on transversal computation to focus the search of the randomized algorithm on undiscovered sentences. This can be combined with the randomized algorithm to give an algorithm which can (provably) find all the most specific sentences. We have illustrated the application of the algorithm in two important data mining scenarios: computing all maximal σ-frequent sets of a {0,1}relation with threshold σ and computing all minimal keys of a database.

We have shown that algorithms All_MFS and All_MAK compute all of MFS and MAK respectively, although they are randomized algorithms. We now show that this happens because the collection of all σ-frequent sets for a given matrix and σ, (and similarly all maximal anti-keys of a database) forms an ideal in the boolean lattice of all attributes (resp. fields). It turns out that whenever the collection to be computed is the set of maximal sets of an ideal in the boolean lattice and the membership in the ideal can be checked efficiently, an analogous randomized algorithm will find all maximal sets.

Theorem 14 Let $\mathcal{I}$ be an ideal in boolean lattice over universe {1,...,n}. Assume that for an arbitrary set $S \subset \{1, ..., n\}$, we can check efficiently if $S \in \mathcal{I}$. Then there

is a randomized algorithm analogous to All_MFS (or All_MAK) which will find all maximal sets of $\mathcal{I}$.

Proof: Define an analogous of A_Random_MFS(S), which given a set $S \subseteq \{1, ..., n\}$, gets a random permutation of the elements $\{1, ..., n\} - S$ and then uses it to find a random maximal element of the ideal containing S. The algorithm will start with S and try to grow greedily the set to a superset by using the remaining elements in the order of permutation and maintaining the invariant that the current S is always in $\mathcal{I}$. Note that if there exists one such maximal set containing S, then $S \in \mathcal{I}$ and so the algorithm will start off by including all of S which will satisfy the invariant to begin with and eventually come up with some maximal set containing S.

Now also define another algorithm which is analogous to All_MFS which starts by running the previous algorithm a few times to compute a nonempty collection of maximal sets of $\mathcal{I}$ and then alternates between computing the minimal transversals of the complements of the known maximal elements and using the previous algorithm to compute for every minimal transversal a random maximal set containing the transversal. Since an analogue of Lemma 5 holds for the maximal sets of any ideal, it follows similar to the proofs of Theorems 6 and 12, that when the while loop of the algorithm is exited, the collection of maximal sets found includes all maximal elements of $\mathcal{I}$. ∎

We have conducted some experiments with our algorithms which indicate the benefits of using randomization over the earlier known level-wise approach. Though the preliminary results of the experiments show our algorithms to be promising, a lot more remains to be done to substantiate these promises. These include the scalability analysis and the tradeoff of transversal computation for focusing search against the level wise approach.

An issue we have not explored yet is to find a more efficient way to use transversals to guide the search. Currently we compute all the minimal transversals from scratch every time some new sentences is discovered. Some alternative strategies may result in faster algorithms. One possibility is to compute only one transversal at a time instead of all of them. Another possibility is to avoid computing the minimal transversals from scratch every time. Since the hypergraph (whose minimal transversals are computed) in a given iteration contains the hypergraphs for all previous iterations, it may be possible to compute the new set of minimal transversals by scanning the old set and dropping those which are no longer transversals and then computing only the newly formed transversals.

Another interesting possibility we plan to explore is to use the randomized algorithm in combination with the level-wise algorithm. The randomized algorithm for maximal σ-frequent sets can be used to select the right range for σ, as a preprocessing step to the level-wise algorithm of Agrawal et. al. [2].

10 Acknowledgements

We are thankful to Jordan Gergov for his comments on an earlier version of the paper.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'93)*, pages 207 – 216, May 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307 – 328. AAAI Press, Menlo Park, CA, 1996.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *International Conference on Data Engineering*, Mar. 1995.

[4] S. Bell. Deciding distinctness of query results by discovered constraints. *Manuscript.*

[5] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. Technical Report LS-8 14, Universität Dortmund, Fachbereich Informatik, Lehrstuhl VIII, Künstliche Intelligenz, 1995.

[6] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278 – 1304, Dec. 1995.

[7] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI Press, Menlo Park, CA, 1996.

[8] M. Garey and D. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[9] D. Gunopulos, H. Mannila, and S. Saluja. On some problems related with keys in relations. *Manuscript (In preparation)*, 1996.

[10] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.

[11] A. J. Knobbe and P. W. Adriaans. Discovering foreign key relations in relational databases. In *Workshop Notes of the ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*, pages 94 – 99, Heraklion, Crete, Greece, Apr. 1995.

[12] H. Mannila. Aspects of data mining. In *Workshop Notes of the ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*, pages 1–6, Heraklion, Crete, Greece, Apr. 1995.

[13] H. Mannila. Data mining: machine learning, statistics, and databases. In *Proceedings of the 8th International Conference on Scientific and Statistical Database Management, Stockholm*, 1996. To appear.

[14] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies. *Data & Knowledge Engineering*, 12(1):83 – 99, Feb. 1994.

[15] H. Mannila and H. Toivonen. On an algorithm for finding all interesting sentences. In *Cybernetics and Systems Research '96*, Vienna, Austria, Apr. 1996. To appear.

[16] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 210 – 215, Montreal, Canada, Aug. 1995.

[17] J. Schlimmer. Using learned dependencies to automatically construct sufficient and sensible editing views. In *Knowledge Discovery in Databases, Papers from the 1993 AAAI Workshop (KDD'93)*, pages 186 – 196, Washington, D.C., 1993.

[18] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Rockville, MD, 1988.

[19] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.