DEPENDENCY INFERENCE

(Extended Abstract)

*Heikki Mannila**

University of Helsinki, Department of Computer Science
Tukholmankatu 2, SF-00250 Helsinki, Finland

Kari-Jouko Räihä

University of Tampere, Department of Computer Science
P.O. Box 607, SF-33101 Tampere, Finland

Abstract. The problem of generating a cover for the set of functional dependencies that hold in a given relation is studied. The problem is an instance of the general problem of concept learning. It has applications e.g. in relational database design and in query optimization. A straightforward solution algorithm is shown to require exponential time for all inputs. We show that for some relations this time requirement is unavoidable, i.e., there are small relations where an exponential number of nontrivial dependencies hold. However, such relations are rare in practice. An algorithm is then developed that works efficiently with respect to the size of its input and its output.

1. Introduction

We consider the following *dependency inference* problem:

> Given a relation r, find a set of functional dependencies that logically determines all the functional dependencies holding in r.

The problem area of inferring general rules from instances of data has become popularly known as *machine learning* [MCM83, MCM86] or *knowledge acquisition*. In our case the simple and regular form of data (a relation) makes the problem particularly attractive. The concept (a set of dependencies) fitting the data (the relation) exactly always exists. Thus, for our problem, the learning can be solved exactly, whereas in inductive learning there is always a possibility of error.

Our interest in the dependency inference problem arouse from database design, which is traditionally based on constraints that the stored data must satisfy. Such constraints are abstract entities, and it is easy for the database designer to overlook some constraints that should hold. In our previous work we have proposed the use of examples to help the designer in getting a better intuitive idea about the consequences of the constraints. Algorithms for generating the examples are given in [BDFS84] and [MR86] for the relational model and in [Ma87] for the entity-relationship model.

The first proposal to use example relations in database design was made by Silva and Melkanoff [SM81]. They suggested the use of a single example relation generated for the universal relation scheme. In [MR86] we argue that using a separate example for each relation scheme is more useful, and study in detail problems related to the use and generation of examples. Our approach has been implemented [MRK87]. Another system supporting the use of examples is described in [BGM85]; however, the paper contains no algorithms for dealing with the example relations.

The role of the examples is more profound than just being an illustration of a prospective design. The best examples satisfy exactly the set of constraints required by the designer, i.e., they are Armstrong relations [Ar74, Fa82] (or corresponding ER-instances). That is, the example and the set of constraints are simply dual, equally powerful representations of the same facts. Incorrect constraints are easier to spot from the list of constraints, whereas the examples are better in revealing missing constraints.

To make full use of this duality, we need algorithms for transforming the two representations into each other. The example generation problem is fairly well understood, but the opposite direction has received less attention. Related questions have, however, been posed by others.

Delgrande [Del87] studies the problem of finding supporting evidence for the validity or invalidity of a given integrity constraint. His work differs from ours in several aspects. First, his integrity rules can be very general: they are arbitrary expressions in a (slightly weakened) relational algebra. We concentrate on dependencies, since they are the most important integrity rules for (re)designing the database scheme. Second, Delgrande uses only a part of the database to find the supporting evidence. He assumes that the algorithm is applied during the actual use of the database for 'monitoring' the validity of the constraints, and therefore the entire database is prohibitively large. Instead, we assume that the database has been explicitly constructed to help the designer, and it is therefore small. Finally and most importantly, Delgrande only considers finding evidence for the dependencies proposed by the user or designer. No attempt is made to automatically *generate* the set of all valid dependencies. (Considering the general form of constraints, such an attempt would not even be realistic.)

Similarly, Borgida, Mitchell and Williamson ([BW85], [BMW86]) suggest a method for automatically maintaining a set of exceptions to the integrity constraints. When sufficiently many exceptions are found, possible corrections to the constraints (based on some heuristics) are proposed to the designer. Again, no rules are automatically inferred.

Another, completely different application of dependency inference arises in query optimization. Traditionally queries are optimized with respect to a given set of constraints that any instance of the database must satisfy. However, the particular instance existing at the time of query evaluation may well satisfy additional dependencies that could be used to speed up query evaluation. One

---

*) On leave from the University of Tampere, Department of Computer Science. The work of this author was supported by the Academy of Finland.

algorithm specifically designed for instance based query optimization is presented in [Dec87]. If the dependency inference problem can be solved efficiently, its time usage can be subsumed by the savings achieved in query evaluation.

The rest of the paper is organized as follows. A naive solution to the dependency inference problem is presented in Section 2, where the exponential time complexity of this algorithm is also proved. Section 3 shows that in some cases we cannot do any better: there are small relations with large nontrivial dependency sets. However, in practical situations the naive algorithm behaves poorly, since most relations do not have this property. Therefore the remaining sections are devoted to developing a practical algorithm. Section 4 introduces some useful concepts that summarize the properties of a relation from the point of view of dependency inference. Section 5 contains rules and algorithms for simplifying this summary information, and outlines a practical algorithm for dependency inference. Section 6 makes some additional observations about the behavior of the algorithm. Open questions are listed in Section 7.

2. A naive algorithm for dependency inference

We start this section by introducing some notation. Let r be a relation over a relation scheme R. If F is a set of functional dependencies, then $r \models F$ means that all dependencies of F hold in r. If $X \rightarrow Y$ is a single dependency, $r \models X \rightarrow Y$ means $r \models \{X \rightarrow Y\}$. The set of all dependencies holding in r is denoted by $dep(r)$, i.e.,

$$X \rightarrow Y \in dep(r) \text{ if and only if } r \models X \rightarrow Y.$$

The dependency $X \rightarrow Y$ is a consequence of F, denoted $F \models X \rightarrow Y$, if for all r, $r \models F$ implies $r \models X \rightarrow Y$.

If F and G are equivalent dependency sets, i.e. all the dependencies of G are consequences of F and vice versa, we say that F is a cover of G (and G is a cover of F). In general, $dep(r)$ has several equivalent covers of varying size, and we are interested in finding a small cover.

In this section we study a straightforward way of inferring a cover of $dep(r)$ for a given relation r. Consider the following algorithm.

```
1.  F := ∅;
2.  for all subsets X ⊆ R do
3.      for all attributes A ∈ R − X do
4.          if r ⊨ X → A then F := F ∪ {X → A};
```

Clearly, F contains only dependencies that hold in r. Moreover, since the algorithm examines all possible nontrivial dependencies of the form $X \rightarrow A$, F will in the end contain all such dependencies that hold in r. Since all dependencies $X \rightarrow Y$ can be derived from dependencies with a single attribute on the right hand side, F is a desired cover of $dep(r)$.

To analyze the time complexity of the naive algorithm, let us use n to denote $|R|$ (the number of attributes in R) and p to denote $|r|$ (the number of tuples in r). The dependency $X \rightarrow A$ can be chosen in $n \cdot 2^{n-1}$ ways. Testing whether $r \models X \rightarrow A$ takes time $O(p^2 \cdot |X|)$, if each pair of tuples is checked individually. Another possibility is to sort the rows of r on X before making each test. Sorting takes time $O(|X| \cdot p \cdot \log p)$, and checking that the ordered relation satisfies $X \rightarrow A$ takes time $O(|X| \cdot p)$. Since $|X| \leq n$, the total complexity of this alternative is therefore

$$O(n \cdot 2^{n-1} \cdot n \cdot p \cdot \log p) = O(n^2 \cdot 2^{n-1} \cdot p \cdot \log p).$$

This exponential time requirement is clearly much too big to make the naive algorithm useful. For a relation with 10 tuples and 10 attributes the bound (albeit pessimistic) would be about $1.5 \cdot 10^6$. For a relation of realistic size the time usage is intolerable.

Our goal is to come up with an algorithm that performs better. In most cases the poor performance of the naive algorithm is caused by the fact that it generates a huge number of redundant dependencies. Making the algorithm generate only nonredundant dependencies is already a big improvement over the naive version. Unfortunately, there are still cases that are inherently hard, i.e. cases where the size of *any* correctly inferred cover is big. This is shown in the next section.

3. Size bounds for the inferred dependency sets

Theorem 1. For each n there exists a relation r over R such that $n = |R|$, $|r| = O(n)$, and each cover of $dep(r)$ has $\Omega(2^{n/2})$ dependencies. □

In practice, relations that have inherently large *dep*-sets should be rare. The fact that $dep(r)$ has only large covers implies that either r has many different keys, or the relation scheme is highly unnormalized (since lots of non-key dependencies hold). Both situations are unlikely.

Because of Theorem 1 we have to be modest in our search for an efficient algorithm for the dependency inference problem. However, there is still room for improving the naive algorithm, which takes exponential time even in the *best* case. We would like to find an algorithm that works in polynomial time with respect to both $|r|$ and the size (i.e., the number of dependencies) of a minimum cover of $dep(r)$ (the cover having the fewest dependencies).

Such an algorithm will have to start by inspecting the entire relation r and by gathering suitable information for producing a cover of $dep(r)$. Any algorithm (such as the one given in [MR86]) that simply steps through all pairs of tuples in an arbitrary order and at each step maintains a cover of the dependency set that still has not been invalidated, is bound to be inefficient. This follows from the proof of Theorem 1.

In the next section we will study what kind of essential information can be efficiently obtained from r to produce a small cover of $dep(r)$.

4. Necessary sets

In this section we describe an algorithm for the dependency inference problem. The remaining sections will concentrate on modifications and analysis of this method.

The algorithm is based on the following concept. Let r be a fixed relation over attribute set R. Let s and t be two tuples from r. Define

$$disag(s, t) = \{B \in R \mid s[B] \neq t[B]\}.$$

Suppose an attribute A belongs to $disag(s, t)$ and let $X = disag(s, t) - \{A\}$. Then X is a *necessary set* for A. Let $nec(A)$ denote the collection of all necessary sets for A, i.e.

$$nec(A) = \{\, disag(s, t) - \{A\} \mid s, t \in r, A \in disag(s, t) \,\}.$$

Example 1. Consider the following relation r over $R(ABCD)$:

A	B	C	D
0	0	0	0
1	1	0	0
0	2	0	2
1	2	3	4

For this relation,
$nec(A) = \{\{B\}, \{B, C, D\}, \{B, D\}, \{C, D\}\}$,
$nec(B) = \{\{A\}, \{D\}, \{A, C, D\}, \{A, D\}, \{C, D\}\}$,
$nec(C) = \{\{A, B, D\}, \{B, D\}, \{A, D\}\}$,
$nec(D) = \{\{B\}, \{A, B, C\}, \{A, B\}, \{B, C\}, \{A, C\}\}$. □

The term necessary set comes from the observation that if two rows s and t differ in attribute A (i.e., $A \in disag(s, t)$), then for any set X, if $X \rightarrow A$ holds in r, X must contain some attribute in $disag(s, t)$. This is formalized in the following.

Let $A \in R$ be an attribute. Consider the family of all nontrivial sets $Z \subseteq R$ such that $Z \rightarrow A$ holds in the relation r:

$$lhs(A) = \{ Z \subseteq R \mid r \models Z \rightarrow A \text{ and } A \notin Z \}.$$

Moreover, for any set $S = \{X_1, \ldots, X_k\}$ define

$$\beta(S) = \{ Y \subseteq R \mid X \cap Y \neq \emptyset \text{ for all } X \in S \}$$

and

$$\alpha(S) = \{ \{B_1, \ldots, B_k\} \mid B_i \in X_i \text{ for } 1 \leq i \leq k \}.$$

Lemma 1. $lhs(A) = \beta(nec(A))$. □

Lemma 2. $dep(r) = \{Y \rightarrow A \mid Y \subseteq R, A \in R - Y, X \cap Y \neq \emptyset \text{ for all } X \in nec(A) \}$. □

Lemma 2 gives us an alternative naive way of computing the set $dep(r)$: we first compute the necessary sets, and then use them to generate all the dependencies. The first step is reasonably fast:

Lemma 3. The collection $\cup \{ nec(A) \mid A \in R \}$ can be computed in time $O(p^2n^2)$, where $p = |r|$ and $n = |R|$. □

But the remaining step again requires us to step through all possible dependencies, and thus it is too slow. However, it is easy to improve on this. Let $S = \{X_1, \ldots, X_k\}$ be a collection of attribute sets. Denote

$$gendep(S) = \{ X \rightarrow A \mid X \in \alpha(S) \}.$$

Lemma 4. The collection $\cup \{ gendep(nec(A)) \mid A \in R \}$ is a cover of $dep(r)$. □

Hence it is possible to get a representation for $dep(r)$ by generating all dependencies $X \rightarrow A$, where A is in R and X contains an attribute from each set in $nec(A)$. This can be substantially faster than considering all the dependencies. The next section discusses ways to obtain further improvements.

5. Simplification lemmas

There are two problems with the use of the *gendep*-sets to the dependency inference problem as outlined in the end of Section 4. The first is that the generation process may produce a lot of unnecessary dependencies. E.g., in Example 1 we had $nec(B) = \{\{A\}, \{D\}, \{A, C, D\}, \{A, D\}, \{C, D\}\}$. The only dependency one really has to produce for B is $AD \rightarrow B$; however, the set $gendep(B)$ contains also the dependency $ACD \rightarrow B$. It is easy to construct more extreme examples where exponentially many unnecessary dependencies are produced.

Example 2. Let

$$nec(A) = \{\{B\}, \{B, C_1, D_1\}, \{B, C_2, D_2\}, \ldots, \{B, C_k, D_k\}\}.$$

Then the only nonredundant dependency is $B \rightarrow A$, but a simple generation method would produce 2^k dependencies. □

The second problem is that the generation method does not consider the interaction of dependencies.

Example 3. Suppose we have a relation where exactly the dependencies $B_1 \rightarrow B_2, B_3 \rightarrow B_4, \ldots, B_{2k-1} \rightarrow B_{2k}$ and $B_2B_4 \ldots B_{2k} \rightarrow A$ hold (that is, we have an Armstrong relation for this set of dependencies). Then the family $nec(A)$ contains the sets

$$\{B_1, B_2\}, \{B_3, B_4\}, \ldots, \{B_{2k-1}, B_{2k}\}.$$

This leads into the generation of 2^k dependencies with right hand side A; one would be enough. □

In this section we consider mainly the first source of inefficiency. The second (and seemingly more difficult) one will be addressed in Section 6.

First we give a result which enables us to reduce the size of the sets $nec(A)$ drastically.

Lemma 5. Suppose $U, V \in nec(A)$ and $U \subseteq V$. Then $gendep(nec(A) - \{V\})$ is a cover of $gendep(nec(A))$. □

That is, if one necessary set includes another, the larger set can be removed without changing the resulting set of dependencies. In Example 1 the use of Lemma 5 would reduce the necessary set collections to

A: $\{\{B\}, \{C, D\}\}$,
B: $\{\{A\}, \{D\}\}$,
C: $\{\{A, D\}, \{B, D\}\}$, and
D: $\{\{B\}, \{A, C\}\}$.

Lemma 5 enables us to preprocess the $nec(A)$-sets by removing unneeded sets from them. Denote by $nec'(A)$ the subfamily of $nec(A)$ left after all unnecessary sets have been removed using Lemma 5.

While replacing $nec(A)$ by $nec'(A)$ yields a considerably smaller dependency set, there is still room for improvement because of the product construction used in the generation of dependencies. In our example e.g. $\alpha(nec'(C)) = \{\{A, B\}, \{A, D\}, \{B, D\}, \{D\}\}$, which produces four dependencies in $gendep(nec'(C))$. However, only $AB \rightarrow C$ and $D \rightarrow C$ are really needed. We show next how the redundant sets can be pruned during the generation.

We first define a subfamily consisting of such left hand sides which do not contain any other left hand side:

$$nrlhs(A) = \{ Z \in lhs(A) \mid \text{for all } Y \in lhs(A), Y \subseteq Z \text{ only if } Y = Z \}$$

(nr comes from nonredundant).

An algorithm which does not take the interconnection of attributes into account can at best produce the dependencies $Z \rightarrow A$, where $Z \in nrlhs(A)$, for each attribute $A \in R$. Next we show how our basic algorithm can do this. The following lemma relates $nrlhs(A)$ to $nec'(A)$.

Lemma 6. $nrlhs(A) \subseteq \alpha(nec'(A)) \subseteq lhs(A)$. □

Our task is to recover the set $nrlhs(A)$ from the collection $nec'(A)$. We give a result showing how the $nec'(A)$-family can be pruned during the generation of the dependencies.

Let $gendep(nec'(A), B)$ denote the dependencies $Y \rightarrow A \in gendep(nec'(A))$ such that $B \in Y$. Hence

$$gendep(nec'(A)) = \cup \{ gendep(nec'(A), B) \mid B \in R \}.$$

Lemma 7. Let $X \in nec'(A)$, and let $Z \in nec(A)$ such that $Z \neq X$. If $B \in Z \cap X$, then $gendep(nec'(A) - \{Z\}, B)$ is a cover of $gendep(nec'(A), B)$. □

Thus, if we have started to generate the dependencies in a cover of $gendep(nec'(A))$ by choosing an attribute B from some set X in $nec'(A)$, we can leave out all sets $Z \in nec'(A)$ which contain B. New attributes are chosen repeatedly until all the sets in $nec'(A)$ are represented. Then backtracking is used (and the corresponding sets Z are reintroduced) until all the alternatives have been produced.

If the left hand sides in $nrlhs(A)$ are disjoint, the algorithm is particularly efficient, since no backtracking is required.

Theorem 2. If the sets in $nrlhs(A)$ are disjoint, the complexity of the above algorithm is polynomial in $|R|$. □

6. Global considerations

In the previous section we considered only the problem of inferring the left hand sides determining a given attribute. In this section we consider the interaction of attributes. The following lemma is useful.

Lemma 8. Let $A, B, C \in R$ and suppose $nec'(B) = \{X\}$, where $C \in X$. Suppose $Z \in nec'(A)$ and $B, C \in Z$. Then Z can be replaced by $Z - \{C\}$ in $nec'(B)$ without altering the closure of the resulting dependency set. □

The use of Lemma 8 takes care of cases resembling Example 3.

The size of the cover for $dep(r)$ produced by our methods can be bounded, if we know that r is in normal form, say BCNF. This result is based on Theorem 2 and the fact that in BCNF schemes dependencies do not interact.

Theorem 3. If the relation is in BCNF and the keys are disjoint, the general algorithm finds them in polynomial time. □

The point of Theorem 3 is not that this special case is singularly important or difficult to solve using specialized methods. Rather, it shows that the general algorithm works efficiently, even in this case.

7. Concluding remarks

We have considered the problem of inferring the functional dependencies holding in a given relation. We started by discussing the connections and applications of this problem. We showed that the trivial algorithm is hopelessly slow, and proved that the result can sometimes be exponentially large, no matter what algorithm is used. We outlined a general method which works fast for relations satisfying few nontrivial dependencies.

Several open problems remain. The analysis of the general algorithm is by no means complete. In general, dealing with an example relation is not necessarily easier than dealing with a set of dependencies: for example, in [BDFS84] it is shown that the problem of determining whether a relation has a key of size less than a given integer is NP-complete. This states some limitations which any dependency inference algorithm must face.

There are also interesting extensions to this problem. One could consider inferring other types of dependencies, like multivalued dependencies or inclusion dependencies [CFP84]. The latter class seems fairly easy. Another useful direction is to consider the *incremental version* of the dependency inference problem: given a relation r, a set of dependencies that is a cover of $dep(r)$, and a tuple t, find a cover of $dep(r \cup \{t\})$. This version of the problem is especially important in incremental database design.

From the point of view of concept learning, a functional dependency $X \rightarrow A$ is a rule of the following form: if something ($t[X] = s[X]$) holds, then something else ($t[A] = s[A]$) holds. The necessary set approach seems to give interesting learning algorithms for other classes of rules, too. This is a topic of a future paper.

References

Ar74 W.W. Armstrong, Dependency structures of database relationships. *Information Processing 74*, Proceedings of IFIP Congress 74, J.L. Rosenfeld (ed.), North-Holland Publ. Co., Amsterdam, 1974, 580-583.

BDFS84 C. Beeri, M. Dowd, R. Fagin, R. Statman, On the structure of Armstrong relations for functional dependencies. *J. ACM* 31, 1 (1984), 30-46.

BMW86 A. Borgida, T. Mitchell, K.E. Williamson, Learning improved integrity constraints and schemas from exceptions in data and knowledge bases. *On Knowledge Base Management Systems*, M.L. Brodie and J. Mylopoulos (eds.), Springer-Verlag, 1986, 259-286.

BW85 A. Borgida, K. Williamson, Accommodating exceptions in databases, and refining the schema by learning from them. *Proceedings of the Eleventh International Conference on Very Large Data Bases*, August 1985, 72-81.

BGM85 M. Bouzeghoub, G. Gardarin, E. Metais, Database design tools: an expert system approach. *Proceedings of the Eleventh International Conference on Very Large Data Bases*, August 1985, 82-95.

CFP84 M.A. Casanova, R. Fagin, C.H. Papadimitriou, Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences* 28 (1984), 29-59.

Dec87 R. Dechter, Decomposing an n-ary relation into a tree of binary relations. *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1987.

Del87 J.P. Delgrande, Formal bounds on the automatic generation and maintenance of integrity constraints. *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1987.

Fa82 R. Fagin, Armstrong databases. IBM Research Report RJ3440, San Jose, Calif., May 1982.

Ma87 H. Mannila, Generating example databases from ER-schemes. Manuscript, 1987.

MR86 H. Mannila, K.-J. Räihä, Design by example: an application of Armstrong relations. *Journal of Computer and System Sciences* 33, 2 (October 1986), 126-141.

MRK87 H. Mannila, K.-J. Räihä, M. Kantola, Design-by-Example: a user guide. Manuscript, 1987.

MCM83 R.S. Michalski, J.G. Carbonell, T.M. Mitchell (eds.), *Machine Learning: an Artificial Intelligence Approach.* Tioga, 1983.

MCM86 R.S. Michalski, J.G. Carbonell, T.M. Mitchell (eds.), *Machine Learning: an Artificial Intelligence Approach, Vol. II.* Morgan Kaufmann, 1986.

SM81 A.M. Silva, M.A. Melkanoff, A method for helping discover the dependencies of a relation. *Advances in Data Base Theory, Vol. 1*, H. Gallaire, J. Minker, J.M. Nicolas (eds.), Plenum Press, 1981, 115-133.