Database dependency discovery: a machine learning approach

Peter A. Flach
*Department of Computer Science,
University of Bristol,
Bristol BS8 1UB, United Kingdom,*
Peter.Flach@bristol.ac.uk

Iztok Savnik
*Faculty of Computer and Information Science,
University of Ljubljana,
1000 Ljubljana, Slovenia,*
Iztok.Savnik@fri.uni-lj.si

Database dependencies, such as functional and multivalued dependencies, express the presence of structure in database relations, that can be utilised in the database design process. The discovery of database dependencies can be viewed as an induction problem, in which general rules (dependencies) are obtained from specific facts (the relation). This viewpoint has the advantage of abstracting away as much as possible from the particulars of the dependencies. The algorithms in this paper are designed such that they can easily be generalised to other kinds of dependencies.
Like in current approaches to computational induction such as inductive logic programming, we distinguish between top-down algorithms and bottom-up algorithms. In a top-down approach, hypotheses are generated in a systematic way and then tested against the given relation. In a bottom-up approach, the relation is inspected in order to see what dependencies it may satisfy or violate. We give a simple (but inefficient) top-down algorithm, a bi-directional algorithm, and a bottom-up algorithm. In the case of functional dependencies, these algorithms have been implemented in the FDEP system and evaluated experimentally. The bottom-up algorithm is the most efficient of the three, and also outperforms other algorithms from the literature.

Keywords: Induction, attribute dependency, database reverse engineering, data mining.

1. Introduction

Dependencies between attributes of a database relation express the presence of structure in that relation, that can be utilised in the database design process. For instance, a functional dependency $X \rightarrow Y$ expresses that the values of attributes X uniquely determine the value of attributes Y. The way Y depends on X can thus be stored in a separate relation with attributes $X \cup Y$. Furthermore, Y can be removed from the original relation, which can be reconstructed as the join of the two new relations over X. In this way the implicit structure of the relation is made explicit. In fact, the relation between X and Y does not need to be functional: all that is needed for a lossless decomposition is that the way Y depends on X is independent of the remaining attributes. This has led to the concept of a multivalued dependency $X \rightarrow\!\!\!\rightarrow Y$.

Traditionally, database dependencies were considered to be part of the data model provided by the database designer. However, they may also be retrieved from the extensional data. One reason for doing so can be that the data model, or parts of it, has been lost or is no longer accurate, so that some form of *reverse engineering* is required. Another reason may be that certain dependencies were not foreseen by the database designer, but do occur in practice. Once they have been discovered, they may be utilised for restructuring the database, as indicated above, but also for query optimisation. In this paper we address this problem of dependency discovery, understood as characterising the set of dependencies that are satisfi ed by a given collection of data.[1]

Some previous work has been done on algorithms for dependency discovery, mostly restricted to discovery of functional dependencies [18, 14, 20, 13]. In this paper we propose some new algorithms for discovery of functional dependencies, and we study the new problem of discovery of multivalued dependencies. The major contribution, however, is the elabora-

[1]We avoid using the ambiguous term ‘dependency inference’, which has been used in the literature both for dependency discovery and for the problem of constructing dependencies that are implied by given dependencies, which is not the problem we are dealing with in this paper.

tion of the connection between the problem of dependency discovery and the problem of inductive learning or learning from examples from the field of machine learning. Through this novel perspective we are able to develop our algorithms in a systematic and principled way. Furthermore, the algorithms are formulated in very general terms, relegating the particulars of the kind of dependency that is being induced to specific sub-procedures. As a consequence, the algorithms we develop can be adapted to discover other kinds of dependencies than the functional and multivalued dependencies considered in this paper.

This paper summarises and extends results from [6, 28, 29, 7, 8, 30].

1.1. Overview of the paper

In Section 2 we review the main concepts and notation from the relational database model. Some additional insights are gained through a reformulation in terms of clausal logic. Section 3 gives a brief introduction to the field of computational induction, and indicates how database dependency discovery fits in. Section 4 is the main part of the paper, giving top-down, bi-directional, and bottom-up algorithms for induction of functional and multivalued dependencies. The algorithms are formulated so that they can be easily adapted to other kinds of dependencies. In Section 5 we discuss implementation details of the algorithms. In particular, we define a datastructure for succinct representation of large sets of functional dependencies. In Section 6 we discuss relations of our results with other published work, and in Section 7 we report some experimental results. Section 8 concludes.

2. Preliminaries

This section provides the theoretical backgrounds of database dependencies. In Section 2.1 we review the main concepts and notation from the relational database model. In Section 2.2 we provide an alternative, logical view, through which we derive some additional theoretical results needed further on in the paper.

2.1. The relational model

Our notational conventions are close to [17]. A *relation scheme* R is an indexed set of *attributes* $A_1, \ldots, A_n$. Each attribute A_i has a *domain* D_i, $1 \leq i \leq n$, consisting of *values*. Domains are assumed to be countably infinite. A *tuple* over R is a mapping $t : R \rightarrow \bigcup_i D_i$ with $t(A_i) \in D_i$, $1 \leq i \leq n$. The values of a tuple t are usually denoted as $\langle t(A_1), \ldots, t(A_n) \rangle$ if the order of attributes is understood. A *relation on* R is a set of tuples over R. We will only consider finite relations. Any expression that is allowed for attributes is extended, by a slight abuse of symbols, to sets of attributes, e.g., if X is a subset of R, $t(X)$ denotes the set $\{t(A) \mid A \in X\}$. We will not distinguish between an attribute A and a set A containing only one attribute. A set of values for a set of attributes X is called an X-value. In general, attributes are denoted by uppercase letters (possibly subscripted) from the beginning of the alphabet; sets of attributes are denoted by uppercase letters (possibly subscripted) from the end of the alphabet; values of (sets of) attributes are denoted by corresponding lowercase letters. Relations are denoted by lowercase letters (possibly subscripted) such as n, p, q, r, u; tuples are denoted by t, t_1, t_2, ... If X and Y are sets of attributes, their juxtaposition XY means $X \cup Y$. If S is a set, $|S|$ denotes its cardinality.

Informally, a functional dependency from attributes X to attributes Y expresses that the Y-value of any tuple from a relation satisfying the functional dependency is uniquely determined by its X-value. In other words, if two tuples in the relation have the same X-value, they also have the same Y-value.

Definition 1 (Functional dependency) *Let R be a relation scheme, and let X and Y be subsets of attributes from R. The expression $X \rightarrow Y$ is a* functional dependency *over R* (fd *for short*). *If Y is a single attribute the fd is called* simple.
A pair of tuples t_1 and t_2 over R violates *a functional dependency $X \rightarrow Y$ if $t_1(X) = t_2(X)$ and $t_1(Y) \neq t_2(Y)$.*
A relation r over R violates *a functional dependency $X \rightarrow Y$ if some pair of tuples $t_1 \in r$ and $t_2 \in r$ violates $X \rightarrow Y$; t_1 and t_2 are called* witnesses. *r* satisfies *an fd if r does not violate it.*

The following properties are immediate from Definition 1.

Proposition 1 (Properties of fds) (1) *A relation satisfies a functional dependency $X \rightarrow Y$ iff it satisfies $X \rightarrow A$ for every $A \in Y$.*
(2) *If a relation satisfies an fd $X \rightarrow Y$, it also satisfies any functional dependency $Z \rightarrow Y$ with $Z \supseteq X$.*
(3) *If a relation satisfies an fd $X \rightarrow Y$, it also satisfies any fd $X \rightarrow Y'$ with $Y' = Y - X$.*
(4) *If a functional dependency is satisfied by a relation r, it is also satisfied by any relation $r' \subseteq r$.*

(1) allows us, without loss of generality, to restrict attention to simple functional dependencies with a single dependent attribute. (2) indicates that functional dependencies with minimal left-hand sides convey the most information. (3) indicates that left-hand side attributes are redundant on the right-hand side. (4) states that the set of functional dependencies satisfi ed by a relation decreases monotonically when the relation increases.

Multivalued dependencies generalise functional dependencies by stipulating that every X-value determines a *set* of possible Y-values. For instance, if a relation describes events that occur weekly during a given period, this relation satisfi es a multivalued dependency from day of week to date: given the day of week, we can determine the set of dates on which the event occurs. For instance, if the Computer Science course and the Artifi cial Intelligence course are both taught on a Wednesday during the fall semester, and there is a CS lecture on Wednesday September 7, while there is an AI lecture on Wednesday December 7, then there is also a CS lecture on the latter date and an AI lecture on September 7.

Definition 2 (Multivalued dependency) *Let R be a relation scheme, let X and Y be subsets of attributes from R, and let Z denote $R - XY$. The expression $X \twoheadrightarrow Y$ is a* multivalued dependency *over R* (mvd *for short*).
Given an mvd $X \twoheadrightarrow Y$, the tuple determined *by a pair of tuples t_1 and t_2 over R is defined by $t_3(XY) = t_1(XY)$ and $t_3(Z) = t_2(Z)$. Given a relation r, a pair of tuples $t_1 \in r$ and $t_2 \in r$* violates *a multivalued dependency $X \twoheadrightarrow Y$ if $t_1(X) = t_2(X)$ and the tuple determined by t_1 and t_2 is not in r.*
A relation r over R violates *a multivalued dependency $X \twoheadrightarrow Y$ if some pair of tuples $t_1 \in r$ and $t_2 \in r$ violates $X \twoheadrightarrow Y$; t_1 and t_2 are called* positive witnesses, *and the tuple determined by them is called a* negative witness. *r* satisfi es *an mvd if r does not violate it.*

Thus, a relation r satisfi es an mvd $X \twoheadrightarrow Y$ if for every pair of tuples $t_1 \in r$ and $t_2 \in r$ such that $t_1(X) = t_2(X)$ the tuple t_3 determined by t_1 and t_2 is in r as well. Note that by exchanging t_1 and t_2, there should also be a tuple $t_4 \in r$ with $t_4(XY) = t_2(XY)$ and $t_4(Z) = t_1(Z)$. Furthermore, notice that a pair of tuples can only violate an mvd in the context of a particular relation, while an fd can be violated by a pair of tuples in isolation. This difference has certain consequences for the induction algorithms developed in Section 4.

The following properties are immediate from Defi - nition 2.

Proposition 2 (Properties of mvds) (*1*) *A relation satisfies a multivalued dependency $X \twoheadrightarrow Y$ iff it satisfies $X \twoheadrightarrow Z$ with $Z = R - XY$.*
(*2*) *If a relation satisfies an mvd $X \twoheadrightarrow Y$, it also satisfies any mvd $Z \twoheadrightarrow Y$ with $Z \supseteq X$.*
(*3*) *If a relation satisfies an mvd $X \twoheadrightarrow Y$, it also satisfies any mvd $X \twoheadrightarrow Y'$ with $Y' = Y - X$.*
(*4*) *If a relation satisfies a functional dependency $X \rightarrow Y$, it also satisfies the multivalued dependency $X \twoheadrightarrow Y$.*

(1) states that mvds are pairwise equivalent. This gives us a sort of "normal form" which is however much weaker than for fds: any mvd $X \twoheadrightarrow Y$ partitions a relation scheme R into three subsets $X, Y, R - XY$. Such a partition is called a *dependency basis* for X [17]. It can be generalised to represent a set of mvds with identical left-hand side. (2) and (3) demonstrate that, analogously to fds, multivalued dependencies with minimal left-hand sides are most informative, and that left-hand side attributes are redundant on the right-hand side. (4) states that functional dependencies are indeed special cases of multivalued dependencies. Notice that Proposition 1 (4) does not extend to multivalued dependencies, since an mvd requires the existence of certain tuples in the relation. Thus, if r satisfi es a certain mvd, some subset of r may violate it.

Functional dependencies and multivalued dependencies are jointly referred to as *dependencies*. The attributes found on the left-hand side of an attribute dependency are called *antecedent attributes*, those on the right-hand side *consequent attributes*. The set of dependencies over a relation scheme R is denoted DEP_R. If r is a relation over relation scheme R, the set of dependencies satisfi ed by r is denoted $DEP_R(r)$. Single dependencies are denoted by Greek lowercase letters such as ϕ, ψ, χ; sets of dependencies are denoted by Greek uppercase letter such as Φ, Ψ, Ξ.

As we see from Propositions 1 and 2, the set $DEP_R(r)$ contains a lot of redundancy. A cover of $DEP_R(r)$ is a subset from which all other dependencies satisfi ed by r can be recovered. In order to defi ne this formally we need the notion of entailment between (sets of) dependencies. In the database literature (e.g. [17, 32, 19]) the entailment structure of functional and multivalued dependencies is usually defi ned by means of inference rules like the following (*cf.* Proposition 1 (2), Proposition 2 (1,2)):

from $X \rightarrow Y$ infer $X \rightarrow Z$ for $Z \supseteq Y$

from $X \twoheadrightarrow Y$ infer $X \twoheadrightarrow Z$ for $Z \supseteq Y$

from $X \twoheadrightarrow Y$ infer $X \twoheadrightarrow R - XY$

If we view dependencies as logical statements about the tuples in a relation, as will be discussed in Section 2.2 below, there is no need to axiomatise the entailment structure of functional and multivalued dependencies, as they are, so to speak, hardwired into the logical representation. A cover of a set of dependencies Φ is thus simply any subset that is satisfied by exactly the same relations.

Definition 3 (Cover) *Let R be a relation scheme, and let Φ be a set of dependencies over R. $\Psi \subseteq \Phi$ is a* cover *of Φ if the set of relations over R satisfying Ψ is equal to the set of relations over R satisfying Φ. A cover Ψ is* minimal *if for every proper subset $\Xi \subset \Psi$, there is a relation over R satisfying Ξ but not Ψ.*

We are usually interested in a cover of $DEP_R(r)$, but sometimes we are also interested in a cover of $DEP_R - DEP_R(r)$, i.e. the set of dependencies that are violated by r.

2.2. The logical view

There is a well-known correspondence between statements from the relational model and first-order predicate logic [5, 10, 27]. The basic idea is to interpret an n-ary relation r as a Herbrand interpretation for an n-ary predicate `r`. Given this Herbrand interpretation, we can translate a relational statement like $X \rightarrow Y$ to a logical formula ϕ, such that r satisfies the relational statement if and only if the Herbrand interpretation is a model for ϕ.

For instance, given a relation r with relational scheme $R = \{A, B, C, D\}$, the functional dependency $AC \rightarrow D$ corresponds to the logical statement

```
D1=D2 :- r(A,B1,C,D1),r(A,B2,C,D2)
```

In words: if two tuples from relation r agree on their AC-values, they also agree on their D-value. Similarly, the multivalued dependency $A \twoheadrightarrow BD$ corresponds to the logical statement

```
r(A,B1,C2,D1) :- r(A,B1,C1,D1),
                 r(A,B2,C2,D2)
```

In words: if two tuples t_1 and t_2 from relation r agree on their A-values, then there exists a third tuple t_3 in the relation which has the same A-value as t_1 and t_2, while inheriting its BD-values from t_1 and its C-value from t_2.

We now proceed to define the logical analogues of the main relational concepts from Section 2.1. Let $R = \{A_1, \ldots, A_n\}$ be a relation scheme with associated domains D_i, $1 \leq i \leq n$. With R we associate an n-ary predicate `r`, and with each attribute A_i we associate a countable set of typed variables $X_i, Y_i, Z_i, \ldots$ ranging over the values in domain D_i.[2]

Definition 4 (Clause) *A* relational atom *is an expression of the form $\mathtt{r}(X_1, \ldots, X_n)$. An* equality atom *is an expression of the form $X_i = Y_i$. A* literal *is an atom or the negation of an atom. A* clause *is a set of literals, understood as a disjunction. A* definite clause *is a clause with exactly one positive literal. Definite clauses are conveniently written in the form* `H:-B`*, where* `H` *is the positive literal and* `B` *is a comma-separated list of the negative literals.*

Definition 5 (Substitution) *A* substitution *is a mapping from variables to values from their associated domains. Given a relation r, a substitution* satisfies *an atom $\mathtt{r}(X_1, \ldots, X_n)$ if the tuple resulting from applying the substitution to the sequence $\langle X_1, \ldots, X_n \rangle$ is in r. A substitution satisfies an atom $X_i = Y_i$ if both variables are mapped to the same value. Given a relation r, a substitution* violates *a clause if it satisfies all negative atoms in the clause but none of the positive atoms. A relation r* satisfies *a clause if no substitution violates it.*

Definition 6 (Dependencies in logic) *Let $X \rightarrow A$ be a functional dependency, then the associated logical statement is* `H:-B1,B2`*, constructed as follows.* `B1` *and* `B2` *are two relational atoms that have identical variables for attributes in X, and different variables otherwise.* `H` *is the equality literal* `A1=A2`*, where* `A1` *and* `A2` *are the variables associated with attribute A occurring in* `B1` *and* `B2`*, respectively.*
Let $X \twoheadrightarrow Y$ be a multivalued dependency, then the associated logical formula is `L3:-L1,L2`*, constructed as follows.* `L1` *and* `L2` *are two relational atoms that have identical variables for attributes in X, and different variables otherwise.* `L3` *is the relational atom that*

[2] We adopt the Prolog convention of treating any string starting with a capital as denoting a variable.

has the same variables for attributes in X as `L1` *and* `L2`*, the same variables for attributes in Y as* `L1`*, and the same variables as* `L2` *for the remaining attributes. If ϕ is a relational statement, then the corresponding logical formula is denoted $[\![\phi]\!]$.*

The following Proposition is immediate from the above construction.

Proposition 3 (Logical vs. relational view) *Let R be a relation scheme, r a relation over R, and ϕ a dependency. r satisfies dependency ϕ, as defined in Definitions 1 and 2, if and only if r satisfies formula $[\![\phi]\!]$, as defined in Definitions 6 and 5.*

As an illustration of the advantage of the logical view, consider the multivalued dependency $A \rightarrow\!\!\!\rightarrow C$ which is translated to the statement

```
r(A,B2,C1,D2) :- r(A,B1,C1,D1),
                 r(A,B2,C2,D2)
```

By re-ordering the literals in the body of the latter clause and renaming the variables, it is readily seen that it is in fact equivalent to the clause above representing the mvd $A \rightarrow\!\!\!\rightarrow BD$. Thus, we see that the equivalence between mvd's $X \rightarrow\!\!\!\rightarrow Y$ and $X \rightarrow\!\!\!\rightarrow R - XY$ is obtained as a logical consequence of the chosen representation, and does not need to be axiomatised separately. It is easily checked that this holds true for all properties stated in Propositions 1 and 2. Thus, a major advantage of the logical view is that meta-statements about attribute dependencies are much easier verified.

In particular, we will make use of the following Proposition.

Proposition 4 (Entailment) *Let ϕ and ψ be two fds or two mvds. ϕ entails ψ (that is, every relation that satisfies ϕ also satisfies ψ) iff there is a substitution θ such that $[\![\phi]\!]\theta = [\![\psi]\!]$.*

The substitution θ will have the effect of unifying two variables in the body of the clause, associated with the same attribute. Translating the result back to a dependency, this corresponds to an extension of the antecedent.

Corollary 5 *A functional dependency $X \rightarrow Y$ entails another fd $V \rightarrow W$ iff there exists $Z \subseteq R$ such that $V = XZ$ and $Y - Z \subseteq W \subseteq Y$.*
A multivalued dependency $X \rightarrow\!\!\!\rightarrow Y$ entails another mvd $V \rightarrow\!\!\!\rightarrow W$ iff there exists $Z \subseteq R$ such that $V = XZ$, and either $Y - Z \subseteq W \subseteq Y$ or $(R - XY) - Z \subseteq W \subseteq (R - XY)$.

The significance of Corollary 5 is established by the only-if halves, by which it extends Propositions 1 and 2. Note that the attributes with which the antecedent is extended may remain in the consequent, but clearly they would be redundant. We will need this result in order to prove the completeness of our induction algorithms.

It must be noted that a disadvantage of the logical view with respect to the relational view is that the logical view can refer to attributes only by their position in a relational scheme, whereas the relational model provides a metalevel on which attributes can be referred to by their names. Therefore we usually employ the relational terminology in this paper, and assume that logical concepts like entailment have been adequately reflected upwards to the relational metalevel.

3. Dependency discovery as an induction task

The general problem dealt with in this paper is to characterise the set of dependencies that are satisfied by a given set of tuples. We provide a novel perspective on this problem by viewing it as an induction task. *Induction* is the process of inferring general hypotheses from specific information. It is the main inference step in concept learning from examples, which is the problem of finding a concept definition that correctly classifies all positive and negative examples, while maximising expected accuracy on unseen instances [4].

Induction can also be employed to infer non-classificatory hypotheses, that describe properties (rather than definitions) of the data. From the database perspective one would refer to such properties as *integrity constraints*. Since attribute dependencies constitute a form of database constraints, we will concentrate on this second non-classificatory (also referred to as *confirmatory* [8] or *descriptive*) form of induction.

Broadly speaking, induction algorithms can be classified as belonging to either of two main types: *bottom-up* algorithms, that take the data as starting point for hypothesis construction, and *top-down* algorithms, that embody a generate-and-test approach. These two types of induction algorithms are reviewed in Section 3.1. In Section 3.2 we introduce the main tool for structuring the search space in computational induction: the generality ordering.

3.1. Top-down vs. bottom-up induction

It is customary in computational induction to assume that the *hypothesis space* (the set of possible hypotheses) is known beforehand. Under this assumption, the simplest induction algorithm, reproduced below as Algorithm 1, generates the possible hypotheses H_i one-by-one, halting when the current hypothesis agrees with all examples.

```
**Algorithm 1 (Enumerative induction)**
**Input**: an indexed set of hypotheses $\{H_i\}$,
and a set of examples $E$.
**Output**: a hypothesis $H$ agreeing with $E$.
begin
    $i := 0$;
    while $H_i$ does not agree with $E$
    do $i := i + 1$ od
    output $H_i$
end.
```

In the case of dependency discovery, $\{H_i\}$ would be an enumeration of the set of attribute dependencies over a given relation. For any but the simplest hypothesis space this naive approach is obviously infeasible. The main problem is that, even if the enumeration is done in some systematic way, no information is extracted from it: every hypothesis is treated as if it were the first one. If we view the algorithm as a search algorithm, the search space is totally flat: every hypothesis is an immediate descendant of the root node, so solutions are only found at leaves of the search tree.

If solutions are to be found higher up in the search space, we need a strict partial order that structures the hypothesis space in a non-trivial way. For instance, in concept learning from examples a suitable search order can be defined in terms of generality of the possible concept definitions: more specific definitions are only tried after the more general ones have been found inappropriate (*top-down* algorithms). We will further discuss the generality ordering in Section 3.2.

Bottom-up induction algorithms construct hypotheses directly from the data. They are less easily viewed as search algorithms, and are in some cases even deterministic. For instance, when inducing a definition for the `append`-predicate we may encounter the facts

```
append([a],[],[a])
```

and

```
append([1,2],[3,4],[1,2,3,4])
```

If we assume that these are generated by the same clause, we can construct the clause head

```
append([A|B],C,[A|D])
```

by anti-unification, and then proceed to find appropriate restrictions for the variables. Since the data can be viewed as extremely specific hypotheses, such algorithms climb the generality ordering from specific to general, and are thus in some sense dual to top-down algorithms.

It should be noted that some algorithms, most notably those inducing classification rules, induce a single possible hypotheses agreeing with the data, while others result in a description of the set of all hypotheses agreeing with the data. The latter holds for all algorithms in this paper. Alternatively, we can determine the set of all hypotheses *refuted* by the data, which can be seen as a dual induction problem with the reverse generality ordering.

3.2. The generality ordering

The generality ordering arose from work on concept learning from examples. Briefly, *concept learning* can be defined as inferring an intensional definition of a predicate from positive examples (ground instances) and negative examples (non-instances). The induction task is to construct a concept definition such that the extension of the defined predicate contains all of the positive examples and none of the negative examples. This induces a natural generality ordering on the space of possible definitions: a predicate definition is said to be *more general* than another if the extension of the first is a proper superset of the second.

Notice that this definition of generality refers to extensions and is thus a semantical definition. For the generality ordering to be practically useful during the search for an appropriate hypothesis we need a syntactical analogue of this semantical concept. In theory it would be desirable that the semantical and syntactical generality ordering coincide, but in practice this ideal is unattainable. Even if the set of all extensions forms a Boolean algebra, the syntactical hypothesis space is often not more than a partially ordered set, possibly with infinite chains. The complexity of the learning task depends for a great deal on the algebraic structure of the hypothesis space that is reflected by the syntactical generality ordering.

In general, a predicate definition establishes both *sufficient* and *necessary* conditions for concept membership. In practice, however, the necessary conditions are usually left implicit. This means that the hypothesis language is restricted to definite clauses, as is customary in inductive logic programming [23, 24, 2]. It furthermore means that the extension of the defined predicate is obtained from the least Herbrand model of its definition, using negation as failure. Consequently, one predicate definition is more general than another if the least Herbrand model of the first is a model of the second, which in fact means that the first logically entails the second.

For predicate definitions consisting of a single, non-recursive clause, logical entailment between predicate definitions is equivalent to θ-subsumption: a clause θ-*subsumes* another if a substitution can be applied to the first, such that every literal occurs, with the appropriate sign, in the second.[3] In the general case of recursive clauses θ-subsumption is strictly weaker than logical entailment [11]. However, for the restricted language of attribute dependencies θ-subsumption is quite sufficient, as demonstrated by Proposition 4. Note that clauses representing attribute dependencies always have the same number of literals.

In the last 15 years, an abundance of algorithms for induction of predicate definitions in first-order clausal logic has been developed [23, 24, 2]. A top-down induction algorithm starts from a set of most general sentences, specialising when a sentence is found too general (i.e. it misclassifies a negative example). In definite clause logic, there are two ways to specialise a clause under θ-subsumption: to apply a substitution, and to add a literal to the body. Alternatively, a bottom-up algorithm operates by generalising a few selected examples, for instance by applying some variant of anti-unification.

As has been said before, in this paper we are interested in non-classificatory induction. Contrasting with induction of classification rules, induction of integrity constraints does not allow for an interpretation in terms of extensions.[4] We simply want to find the set of all hypotheses that agree with the data. For instance, given a relation r we may want to find the set $DEP_R(r)$ of all dependencies satisfied by r. Another instance of this problem is the construction of a logical theory axiomatising a given Herbrand model m.

Even if there is no extensional relation between data and hypothesis, the notion of logical entailment can again be used to structure the search space, since we are usually interested in finding a cover of the set of all possible hypotheses, which only needs to contain the most general integrity constraints satisfied by the data (see Definition 3). An important difference in comparison with induction of predicate definitions is that satisfaction of an integrity constraint does not depend on other sentences.[5] This gives rise to a simple yet general top-down algorithm, that is given below as Algorithm 2.

The most interesting differences, and the major contributions of the present paper, manifest itself when considering bottom-up approaches to induction of integrity constraints, that operate more in a data-driven fashion. It is in general not possible to construct a satisfied integrity constraint from only a small subset of the data, because such a constraint may be violated by other parts of the data. It is however possible to construct, in a general and principled way, a number of dependencies that are *violated* by the data. Proceeding in this way we can extract from the data, in one run, all the information that is needed to construct a cover for the set of satisfied dependencies. This makes a data-driven method more feasible than a top-down method when the amount of data is relatively large. The details can be found below in Sections 4.2 and 4.3.

4. Algorithms for inducing dependencies

We are now ready to present the main induction algorithms for dependency discovery. The specifics of a particular kind of dependency will as much as possible be relegated to sub-procedures, in order to obtain specifications of the top-level algorithms that are generally applicable. We start with the conceptually simpler top-down algorithm in Section 4.1, followed by a bi-directional algorithm in Section 4.2, and a pure bottom-up algorithm in Section 4.3.

[3]This definition, and the term θ-subsumption, was introduced in the context of induction by Plotkin [25, 26]. In theorem proving the above version is termed subsumption, whereas θ-subsumption indicates a special case in which the number of literals of the subsumant does not exceed the number of literals of the subsumee [16].

[4]An alternative view of induction of integrity constraints is obtained if we view the data (e.g. a database relation or a Herbrand model) as one example, i.e. a member of the extension of the set of satisfied integrity constraints, which re-establishes the extensional interpretation of generality [3]. A disadvantage of this alternative view is that it reverses the intuitive relation between generality and entailment: an integrity constraint with more Herbrand models is logically weaker.

[5]In contrast, when inducing a recursive predicate definition what we are constructing is a *set* of interrelated sentences, which is clearly much more complicated than the construction of several independent sentences.

4.1. A top-down induction algorithm

We want to find a cover of $DEP_R(r)$ for a given relation r. The basic idea underlying the top-down approach is that such a cover is formed by the most general elements of $DEP_R(r)$. Therefore, the possible dependencies are enumerated from general to specific. For every dependency thus generated, we test whether the given relation satisfies it. If it does not, we schedule all of its specialisations for future inspection; if it does, these specialisations are known to be satisfied also and need not be considered. In the context of database dependencies, the process can be slightly optimised because the witnesses signalling the violation of a dependency may indicate that some of its specialisations are also violated. For instance, let $R = \{A, B, C, D\}$, let r contain the tuples $\langle a, b_1, c, d_1\rangle$ and $\langle a, b_2, c, d_2\rangle$, and consider the fd $A \rightarrow B$ which is violated by r. Its specialisations are $AC \rightarrow B$ and $AD \rightarrow B$, but the first of these is violated by the same pair of witnesses.

Algorithm 2 (Top-down induction of dependencies)
Input*: a relation scheme R and a relation r over R.*
Output*: a cover of $DEP_R(r)$.*

```
begin
    DEPS := $\emptyset$;
    Q := initialise($R$);
    while $Q \neq \emptyset$
    do D := next item from Q; Q := Q − D;
        if some witnesses $t_1, \ldots, t_n$ from r violate D
        then Q := Q $\cup$ spec(R,D,$t_1, \ldots, t_n$)
        else DEPS := DEPS $\cup$ $\{D\}$
        fi
    od
    output DEPS
end.
```

Algorithm 2 is basically an agenda-based search algorithm, specialising items on the agenda Q until they are no longer violated by r. Notice that the thus constructed set $DEPS$ may still contain redundancies; a non-redundant cover may be constructed in some more or less sophisticated way.

Algorithm 2 works for both functional and multivalued dependencies; the only difference lies in the initalisation of the agenda, and the way dependencies are violated and specialised. The correctness of the algorithm depends on the following conditions on the procedure $spec()$:

- every dependency D' returned by $spec(D, t_1, \ldots, t_n)$ is a specialisation of D, i.e. D entails D' but D' does not entail D;
- every dependency entailed by D and not violated by r is entailed by some dependency returned by $spec(D, t_1, \ldots, t_n)$.

The first condition is called *soundness* of the specialisation procedure; the second condition is called *relative completeness* (it is not complete in the sense that minimal specialisations of D that are violated by the same witnesses are not returned).

Theorem 6 *Algorithm 2 returns a cover of $DEP_R(r)$ if the procedure $initialise()$ returns the set of most general dependencies, and the procedure $spec()$ is sound and relatively complete.*

Proof. Let $DEPS$ be the set of dependencies returned by Algorithm 2 when fed with relation r. Clearly, only dependencies not violated by r are included in $DEPS$. The algorithm halts because every dependency has only a finite number of specialisations. It remains to prove that every dependency satisfied by r is entailed by some dependency in $DEPS$. This follows from the fact that the agenda is initialised with the set of most general dependencies and the relative completeness of the specialisation procedure. ▮

For functional dependencies the parameters of Algorithm 2 are instantiated as follows. By Proposition 1 we can restrict attention to simple fds. In accordance with Corollary 5 the agenda is initialised with $\emptyset \rightarrow A$ for every $A \in R$. Furthermore, according to Definition 1 an fd $X \rightarrow A$ is violated by two witnesses t_1 and t_2 that agree on X but not on A. Again according to Corollary 5, $X \rightarrow A$ must be specialised by extending the antecedent; without compromising relative completeness we can ignore A (since the fd $XA \rightarrow A$ is tautological), and also any attribute on which the witnesses agree (since this specialised fd would be refuted by the same witnesses). This results in the following relatively complete specialisation algorithm.

Algorithm 3 (Specialisation of functional dependencies)
Input*: a relation scheme R, a functional dependency $X \rightarrow A$ over R, and two witnesses t_1, t_2 violating $X \rightarrow A$.*
Output*: the non-trivial minimal specialisations of $X \rightarrow A$ not violated by the same witnesses.*

```
proc fd-spec
    SPECS := $\emptyset$;
    DIS := the set of attributes for which
            $t_1$ and $t_2$ have different values;
    for each $B \in$ DIS−$A$
    do SPECS := SPECS $\cup$ $XB \rightarrow A$ od;
    output SPECS
endproc.
```

Theorem 7 *Algorithm 3 is a sound and relatively complete specialisation algorithm.*

Proof. Clearly, $X \rightarrow A$ entails $XB \rightarrow A$ but not *vice versa*, which demonstrates soundness.
Let $V \rightarrow W$ be a non-tautological fd entailed by $X \rightarrow A$, then by Corollary 5 $V \supseteq X$ and $W = A$. If furthermore $V \rightarrow W$ is not contradicted by the same witnesses, then the witnesses do not agree on all attributes in V. Let $B \neq A$ be an attribute on which they disagree, then Algorithm 3 will return $XB \rightarrow A$, which entails $V \rightarrow W$. ▮

Example 1 (Top-down induction of fds) *Consider the following relation r over $R = \{A, B, C, D\}$ (taken from [20]):*

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

The initial fd $\emptyset \rightarrow A$ is violated by the first two tuples; the only specialisation not violated by the same witnesses is $B \rightarrow A$. This fd is in turn violated by the last two tuples, and possible specialisations are $BC \rightarrow A$ and $BD \rightarrow A$. These two dependencies are satisfied by r and output by Algorithm 2.
Similarly, the initial fd $\emptyset \rightarrow B$ is specialised to $A \rightarrow B$, and subsequently to $AD \rightarrow B$ if the first and third tuple are taken as witnesses, or alternatively to $AC \rightarrow B$ and $AD \rightarrow B$ if the second and fourth tuple are taken instead. In the latter case, $AC \rightarrow B$ is specialised to $ACD \rightarrow B$, which is output by Algorithm 2 but may be removed because it is entailed by $AD \rightarrow B$.

For multivalued dependencies the parameters of Algorithm 2 are instantiated as follows. According to Corollary 5 the most general mvd's are $\emptyset \twoheadrightarrow Y$ for every $Y \subseteq R$. In order to avoid redundancies we can ignore $Y = \emptyset$ and $Y = R$. Furthermore, we only need to consider one of each pair $\emptyset \twoheadrightarrow Y$ and $\emptyset \twoheadrightarrow R - Y$; one way to achieve this is to choose a designated attribute $A \in R$, and to initialise the agenda with $\emptyset \twoheadrightarrow AY$ with $Y \subset R$. For instance, for $R = \{A, B, C, D\}$ and A as designated attribute the agenda is initialised with $\emptyset \twoheadrightarrow A$, $\emptyset \twoheadrightarrow AB$, $\emptyset \twoheadrightarrow AC$, $\emptyset \twoheadrightarrow AD$, $\emptyset \twoheadrightarrow ABC$, $\emptyset \twoheadrightarrow ABD$, and $\emptyset \twoheadrightarrow ACD$; while with designated attribute D the most general mvd's are $\emptyset \twoheadrightarrow D$, $\emptyset \twoheadrightarrow AD$, $\emptyset \twoheadrightarrow BD$, $\emptyset \twoheadrightarrow CD$, $\emptyset \twoheadrightarrow ABD$, $\emptyset \twoheadrightarrow ACD$, and $\emptyset \twoheadrightarrow BCD$ (clearly, both sets of mvd's are logically equivalent).

Furthermore, according to Definition 2, given a relation r an mvd $X \twoheadrightarrow Y$ is violated by two positive witnesses t_1 and t_2 if they agree on X but the negative witness determined by them is not in r. Again according to Corollary 5, $X \twoheadrightarrow Y$ must be specialised by moving an attribute from Y or from $R - XY$ to X. Without compromising relative completeness we can ignore any attribute on which the positive witnesses agree (since this specialised mvd would be refuted by the same witnesses). Furthermore, we can also ignore the case in which Y or $R - XY$ is a singleton (since this would result in a tautological mvd).

This results in the following relatively complete specialisation algorithm.

Algorithm 4 (Specialisation of mvds)
Input: *a relation scheme R, an mvd $X \twoheadrightarrow Y$ over R, and two positive witnesses t_1, t_2 and a negative witness t_3 violating $X \twoheadrightarrow Y$.*
Output: *the non-trivial minimal specialisations of $X \twoheadrightarrow Y$ not violated by the same witnesses.*

```
proc mvd-spec
    SPECS := $\emptyset$;
    DIS13 := attributes for which $t_1$ and $t_3$ have different values;
    DIS23 := attributes for which $t_2$ and $t_3$ have different values;
    for each $B \in$ DIS13 such that
            $Y \neq \{B\}$ and $(R - XY) \neq \{B\}$
    do SPECS := SPECS $\cup$ $XB \twoheadrightarrow Y - B$ od;
    for each $B \in$ DIS23 such that
            $Y \neq \{B\}$ and $(R - XY) \neq \{B\}$
    do SPECS := SPECS $\cup$ $XB \twoheadrightarrow Y - B$ od;
    output SPECS
endproc.
```

Theorem 8 *Algorithm 4 is a sound and relatively complete specialisation algorithm.*

Proof. Clearly, $X \twoheadrightarrow Y$ entails $XB \twoheadrightarrow Y - B$ but not *vice versa*, which demonstrates soundness.
Let $V \twoheadrightarrow W$ be an mvd entailed by $X \twoheadrightarrow Y$, then by Corollary 5 $V \supseteq X$. If furthermore $V \twoheadrightarrow W$ is not contradicted by the same witnesses, then the positive witnesses do not agree on all attributes in V. Let B be an attribute on which they disagree, then either $B \in DIS13$ or $B \in DIS23$, and consequently Algorithm 4 will return $XB \twoheadrightarrow Y - B$, which entails $V \twoheadrightarrow W$. ▮

Example 2 (Top-down induction of mvds) *Consider the relation r from Example 1. The initial mvd $\emptyset \twoheadrightarrow D$ is violated by the first and the third tuple, since they determine the tuple $\langle 0, 2, 0, 0 \rangle$, which is not in the relation. We thus have $DIS13 = \{B\}$ and $DIS23 = \{D\}$, hence the only non-trivial specialisation not violated by the same witnesses is $B \twoheadrightarrow D$. This mvd is in turn violated by the last two tuples, and possible special-*

isations are $AB \rightarrow\!\!\!\rightarrow D$ and $BC \rightarrow\!\!\!\rightarrow D$. These two dependencies are satisfied by r and output by Algorithm 2. Notice that the corresponding fd's $AB \rightarrow D$ and $BC \rightarrow D$ are also valid – the course of actions by which the mvd's are found is in this case very similar to the course of actions by which the corresponding fd's are found.
Similarly, the initial mvd $\emptyset \rightarrow\!\!\!\rightarrow B$ is specialised to $A \rightarrow\!\!\!\rightarrow B$, and subsequently to $AD \rightarrow\!\!\!\rightarrow B$ if the first and third tuple are taken as witnesses, or alternatively to $AC \rightarrow\!\!\!\rightarrow B$ and $AD \rightarrow\!\!\!\rightarrow B$ if the second and fourth tuple are taken instead. In the latter case, $AC \rightarrow\!\!\!\rightarrow B$ – while invalid – cannot be specialised anymore.

4.2. *A bi-directional induction algorithm*

Algorithm 2 is a generate-and-test algorithm: it generates dependencies from general to specific, testing each generated dependency against the given relation. We now describe an alternative approach which differs from the top-down approach in that the test phase is conducted against the set of least general *violated* dependencies, rather than against the relation itself. The basic idea is to construct, for each pair of tuples, the least general dependencies for which the tuples are a pair of violating witnesses. For instance, let $R = \{A, B, C, D\}$ and consider the tuples $t_1 = \langle a, b_1, c, d_1 \rangle$ and $t_2 = \langle a, b_2, c, d_2 \rangle$. The fd's $A \rightarrow B$, $A \rightarrow D$, $C \rightarrow B$, $C \rightarrow D$, $AC \rightarrow B$, and $AC \rightarrow D$ are all violated by t_1, t_2; of these, the last two are least general. If we iterate over all pairs of tuples in relation r, we have constructed a cover for the set of dependencies that are *violated* by r; we will call such a cover a *negative cover*.[6] Since the negative cover is calculated in a bottom-up fashion, while the positive cover is calculated from the negative cover in a top-down manner as before, we call this the *bi-directional* induction algorithm.

The following procedure calculates this negative cover.

Algorithm 5 (Negative cover)
Input: *a relation scheme R and a relation r over R.*
Output: *a cover of the dependencies violated by r.*

```
proc neg-cover
    NCOVER := $\emptyset$;
    for each pair of tuples $t_1, t_2$ from r
    do NCOVER := NCOVER $\cup$ violated(R,r,$t_1$,$t_2$) od
    output NCOVER
endproc.
```

[6] Negative covers are called anti-covers in [12].

In practice the procedures *neg-cover*() and *violated*() will be merged, in order to prevent addition of dependencies for which a less general version is already included in the negative cover.

The correctness of Algorithm 5 depends on the procedure $violated(R, r, t_1, t_2)$, which calculates the set of dependencies for which t_1 and t_2 are violating witnesses (the relation r is only needed in the case of multivalued dependencies). This procedure treats the attributes on which the tuples agree as antecedent attributes, and the remaining attributes as consequent attributes.

Algorithm 6 (Violated fds)
Input: *a relation scheme R and two tuples t_1, t_2 over R.*
Output: *the least general fd's over R violated by t_1, t_2.*

```
proc fd-violated
    VIOLATED := $\emptyset$;
    DIS := attributes for which $t_1$ and $t_2$ have different values;
    AGR := R−DIS;
    for each $A \in$ DIS
    do VIOLATED := VIOLATED $\cup$ $AGR \rightarrow A$ od;
    output VIOLATED
endproc.
```

Notice that the fd-initialisation procedure for the top-down approach is a special case of Algorithm 6, with $DIS = R$ and $AGR = \emptyset$.

Theorem 9 *Algorithm 6 correctly calculates the set of least general fd's over R violated by two tuples t_1 and t_2.*

Proof. Clearly, any fd output by Algorithm 6 is violated by t_1 and t_2. Conversely, let $X \rightarrow A$ be an fd violated by t_1 and t_2, then $X \subseteq AGR$ and $A \in DIS$, hence $X \rightarrow A$ entails $AGR \rightarrow A$ which is output by Algorithm 6. ∎

Example 3 (Negative cover for fds) *Consider again the relation r from the previous examples:*

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

For construction of the negative cover we have 6 pairs of tuples to consider.
The first and second tuple result in violated fd's $CD \rightarrow A$ and $CD \rightarrow B$.
The first and third tuple result in violated fd's $AC \rightarrow B$ and $AC \rightarrow D$.

The first and fourth tuple result in violated fd's $\emptyset{\rightarrow}A$, $\emptyset{\rightarrow}B$, $\emptyset{\rightarrow}C$, and $\emptyset{\rightarrow}D$. (*Notice that, if the negative cover is built incrementally by merging Algorithms 5 and 6, as it is implemented in practice, the first, second, and fourth of these fd's will be found redundant when compared with previously found fd's.*)
The second and third tuple result in violated fd's $C{\rightarrow}A$, $C{\rightarrow}B$, and $C{\rightarrow}D$ (*which are all redundant in comparison with fd's found earlier*).
The second and fourth tuple result in violated fd's $A{\rightarrow}B$, $A{\rightarrow}C$, and $A{\rightarrow}D$ (*of which only the second is non-redundant*).
Finally, the third and fourth tuple result in violated fd's $B{\rightarrow}A$, $B{\rightarrow}C$, and $B{\rightarrow}D$.
A non-redundant negative cover consists of the following violated fd's: $B{\rightarrow}A$, $CD{\rightarrow}A$, $AC{\rightarrow}B$, $CD{\rightarrow}B$, $A{\rightarrow}C$, $B{\rightarrow}C$, $AC{\rightarrow}D$, and $B{\rightarrow}D$.

For multivalued dependencies the procedure *mvd-violated*() is slightly more involved, since for a pair of positive witnesses and a possibly violated mvd we need to consider one or both tuples determined by the positive witnesses, and check whether these are indeed negative witnesses not contained in r.

Algorithm 7 (Violated mvds)
Input: *a relation scheme R, a relation r over R, and two tuples t_1, t_2 over R.*
Output: *the least general mvd's violated by t_1, t_2 given r.*

```
proc mvd-violated
    VIOLATED := $\emptyset$;
    DIS := attributes for which $t_1$ and $t_2$ have different values;
    AGR := $R-DIS$;
    A := some attribute in DIS;
    for each $Z \subset (DIS - A)$
    do MVD := $AGR \twoheadrightarrow AZ$;
        $t_3$ := tuple determined by $t_1, t_2$ given MVD;
        if $t_3 \notin r$
        then VIOLATED := VIOLATED $\cup$ MVD
        else $t_4$ := tuple determined by $t_2, t_1$ given MVD;
            if $t_4 \notin r$
            then VIOLATED := VIOLATED $\cup$ MVD
            fi
        fi
    od;
    output VIOLATED
endproc.
```

Note that the construction of MVD in Algorithm 7 is again a variant of the initialisation procedure for mvd's in the top-down approach, by putting $DIS = R$ and $AGR = \emptyset$.

Theorem 10 *Algorithm 7 correctly calculates the set of least general mvd's over R violated by two tuples t_1 and t_2 given relation r.*

Proof. Clearly any mvd added to $VIOLATED$ is violated by t_1 and t_2 given r. Conversely, let $X{\twoheadrightarrow}Y$ be an mvd violated by t_1 and t_2, then $X \subseteq AGR$; therefore the tuples determined by r_1 and t_2 given $X{\twoheadrightarrow}Y$ are the same as determined by t_1 and t_2 given $AGR{\twoheadrightarrow}Y - AGR$. Hence the latter mvd – which is entailed by $X{\twoheadrightarrow}Y$ according to Corollary 5 – will be output by Algorithm 7. ∎

Example 4 (Negative cover for mvds) *We proceed with the running example. For construction of the negative cover for mvd's we have 6 pairs of tuples to consider. We will assume that the designated attribute* (A *in Algorithm 7*) *is always the lexicographically first attribute in DIS.*
For the first and second tuple we have $DIS = \{A, B\}$ and $AGR = \{C, D\}$, hence with A as designated attribute the only potentially violated mvd is $CD{\twoheadrightarrow}A$.[7]
The tuple determined by the first and second tuple in r given $CD{\twoheadrightarrow}A$ is $\langle 0, 1, 0, 0\rangle$, which is indeed outside r, hence the mvd is added to the negative cover.
The first and third tuple result in violated mvd $AC{\twoheadrightarrow}B$.
The first and fourth tuple result in violated mvd's $\emptyset{\twoheadrightarrow}A$, $\emptyset{\twoheadrightarrow}AB$, $\emptyset{\twoheadrightarrow}AC$, $\emptyset{\twoheadrightarrow}AD$, $\emptyset{\twoheadrightarrow}ABC$, $\emptyset{\twoheadrightarrow}ABD$, and $\emptyset{\twoheadrightarrow}ACD$. (*Notice that only $\emptyset{\twoheadrightarrow}ABD$ is non-redundant in the light of the mvd's found previously.*)
The second and third tuple result in violated mvd's $C{\twoheadrightarrow}A$, $C{\twoheadrightarrow}AB$, and $C{\twoheadrightarrow}AD$ (*which are all redundant*).
The second and fourth tuple result in violated mvd's $A{\twoheadrightarrow}B$, $A{\twoheadrightarrow}BC$, and $A{\twoheadrightarrow}BD$ (*of which only the last is non-redundant*).
Finally, the third and fourth tuple result in violated mvd's $B{\twoheadrightarrow}A$, $B{\twoheadrightarrow}AC$, and $B{\twoheadrightarrow}AD$.
A non-redundant negative cover consists of the following violated mvd's: $B{\twoheadrightarrow}A$, $CD{\twoheadrightarrow}A$, $AC{\twoheadrightarrow}B$, $A{\twoheadrightarrow}C$, $B{\twoheadrightarrow}C$, and $B{\twoheadrightarrow}D$.

We can now give the main bi-directional algorithm. It starts with construction of the negative cover, and proceeds by specialising each dependency until none of them entails any element of the negative cover.

[7] With B as designated attribute it would be the logically equivalent mvd $CD{\twoheadrightarrow}B$.

Algorithm 8 (Bi-directional induction of dependencies)
Input: *a relation scheme R and a relation r over R.*
Output: *a cover of $DEP_R(r)$.*

```
begin
    DEPS := $\emptyset$;
    Q := initialise($R$);
    NCOVER := neg-cover($R$,$r$);
    while $Q \neq \emptyset$
    do D := next item from Q; Q := Q $-$ D;
        if D entails some element ND in NCOVER
        then Q := Q $\cup$ spec(D,ND)
        else DEPS := DEPS $\cup$ $\{D\}$
        fi
    od
    output DEPS
end.
```

The procedure *spec*() is similar to the specialisation procedures given above as Algorithms 3 and 4, with the only distinction that specialisation is guided by a dependency ND not to be entailed, rather than a pair or triple of violating witnesses; we omit the details.

The reader will notice that Algorithm 8 is in fact quite similar to Algorithm 2, but there is an important difference: dependencies on the agenda are tested with respect to the negative cover, *without further reference to the tuples in r*. This bi-directional approach is more feasible if $|r|$ is large compared to $|R|$.

Theorem 11 *Let r be a relation over scheme R, let NCOVER be the negative cover constructed by Algorithm 5, and let D be a dependency over R. D violates r iff D entails some element ND of NCOVER.*

Proof. (if) If $ND \in$ *NCOVER*, then ND is contradicted by some witnesses in r. If D entails ND, then D is contradicted by the same witnesses, hence D is violated by r.
(only if) Suppose D is violated by r, then the dependency ND constructed from any witnesses by Algorithm 5 will be entailed by D. ▮

In the light of Theorem 11, the correctness of Algorithm 8 follows from the correctness of Algorithm 2.

Corollary 12 *Algorithm 8 returns a cover of $DEP_R(r)$ if the procedure $initialise()$ returns the set of most general dependencies, and the procedure $spec()$ is sound and relatively complete.*

Example 5 *In Example 3 the following non-redundant negative cover was found: $B{\rightarrow}A$, $CD{\rightarrow}A$, $AC{\rightarrow}B$, $CD{\rightarrow}B$, $A{\rightarrow}C$, $B{\rightarrow}C$, $AC{\rightarrow}D$, and $B{\rightarrow}D$. The initial fd $\emptyset{\rightarrow}A$ entails $CD{\rightarrow}A$ in the negative cover, and is therefore specialised to $B{\rightarrow}A$. This fd is found to be in the negative cover in the next iteration, and is specialised to $BC{\rightarrow}A$ and $BD{\rightarrow}A$. These two dependencies do not entail any element of the negative cover, and are output by Algorithm 8.*
In the case of mvds, in Example 4 the following non-redundant negative cover was found: $B{\rightarrow\!\!\!\rightarrow}A$, $CD{\rightarrow\!\!\!\rightarrow}A$, $AC{\rightarrow\!\!\!\rightarrow}B$, $A{\rightarrow\!\!\!\rightarrow}C$, $B{\rightarrow\!\!\!\rightarrow}C$, and $B{\rightarrow\!\!\!\rightarrow}D$. The initial mvd $\emptyset{\rightarrow\!\!\!\rightarrow}B$ entails $AC{\rightarrow\!\!\!\rightarrow}B$ in the negative cover, and is specialised to $D{\rightarrow\!\!\!\rightarrow}B$. In turn, this mvd entails $CD{\rightarrow\!\!\!\rightarrow}A$ in the negative cover, and is specialised to $AD{\rightarrow\!\!\!\rightarrow}B$.

Notice that the agenda cannot be initialised with *NCOVER* because, although every most general satisfied dependency is the specialisation of a violated dependency (otherwise it would not be most general), it may not be the specialisation of a *least general* violated dependency that is included in the negative cover.

Example 6 *The most general satisfied fd's $BC{\rightarrow}A$ and $BD{\rightarrow}A$ (see Example 1) are indeed specialisations of an fd in the negative cover,* viz. *$B{\rightarrow}A$. However, the most general satisfied fd $AD{\rightarrow}B$ is not.*
Similarly, the most general satisfied mvd's $BC{\rightarrow\!\!\!\rightarrow}A$ and $BD{\rightarrow\!\!\!\rightarrow}A$ (see Example 2) are indeed specialisations of an mvd in the negative cover, viz. *$B{\rightarrow\!\!\!\rightarrow}A$. However, the most general satisfied mvd $D{\rightarrow\!\!\!\rightarrow}C$ is not.*

4.3. A bottom-up induction algorithm

The pure bottom-up induction algorithm is a simple variation on Algorithm 8. Instead of generating the positive cover from the negative cover in a top-down, generate-and-test manner, we iterate once over the negative cover, considering each least general violated dependency only once. Experimental results in Section 7 demonstrate that this yields a considerable improvement in practice.

Algorithm 9 (Bottom-up induction of dependencies)
Input: *a relation scheme R and a relation r over R.*
Output: *a cover of $DEP_R(r)$.*

```
begin
    DEPS := initialise($R$);
    NCOVER := neg-cover($R$,$r$);
    for each ND $\in$ NCOVER
    do TMP := DEPS;
        for each D $\in$ DEPS entailing ND
        do TMP := TMP $-$ D $\cup$ spec(D,ND) od
        DEPS := TMP
    od
    output DEPS
end.
```

The bottom-up algorithm thus contains two nested loops, the outer one of which iterates over the negative cover, ensuring that its elements are inspected only once. Notice that by exchanging the two loops we obtain (more or less) the bi-directional algorithm again.

Example 7 *In Example 3 the following non-redundant negative cover was found: $B{\rightarrow}A$, $CD{\rightarrow}A$, $AC{\rightarrow}B$, $CD{\rightarrow}B$, $A{\rightarrow}C$, $B{\rightarrow}C$, $AC{\rightarrow}D$, and $B{\rightarrow}D$. $B{\rightarrow}A$ causes specialisation of the initial fd $\emptyset{\rightarrow}A$ to $C{\rightarrow}A$ and $D{\rightarrow}A$. The next element of the negative cover $CD{\rightarrow}A$ causes specialisation of these to $BC{\rightarrow}A$ and $BD{\rightarrow}A$. $AC{\rightarrow}B$ causes specialisation of $\emptyset{\rightarrow}B$ to $D{\rightarrow}B$, subsequently specialised to $AD{\rightarrow}B$ by virtue of the fourth element of the negative cover. Continuation of this process leads to construction of the positive cover in one iteration over the negative cover.*

5. Implementation

The algorithms described above have been implemented in two separate systems: the system `fdep` for induction of functional dependencies, and the program `mdep` which realizes algorithms for induction of multivalued dependencies. The system `fdep` is implemented in GNU C and has been used by the authors for the discovery of functional dependencies from a variety of domains.[8] The program `mdep` is implemented in Sicstus Prolog. It serves as an experimental prototype for the study of algorithms for the discovery of multivalued dependencies.

The programs `fdep` and `mdep` are described in the following sub-sections. Section 5.1 presents the implementation of the algorithms for induction of functional dependencies. It includes the description of the data structure which is used for storing sets of dependencies, as well as handling of missing values and noise. In Section 5.2 we present the main aspects of the program `mdep` and some experiences obtained with the prototype. Experiments with `fdep` are described separately in Section 7.

[8]The system is available from the `fdep` homepage at `http://www.cs.bris.ac.uk/~flach/fdep/`.

5.1. Algorithms for inducing functional dependencies

The top-down induction algorithm for functional dependencies turned out to be computationally too expensive for inducing functional dependencies from larger relations. The main weakness of this algorithm is its method for testing the validity of the hypotheses; they are tested by considering all pairs of tuples from the input relation. The method for testing the validity of a functional dependency is considerably improved in the bi-directional induction algorithm.

The bi-directional algorithm uses the same method for the enumeration of hypotheses as the top-down algorithm: in both algorithms hypotheses (i.e., functional dependencies) are specialised until the valid dependencies are reached. The main improvements of the bi-directional algorithm are in the method for testing the validity of the dependencies, and for specialisation of refuted functional dependencies. First, instead of checking the hypotheses on the complete relation, they are tested using the negative cover (the set of most specific invalid dependencies). Secondly, the hypotheses are specialised in the bi-directional algorithm using the invalid dependencies which are the cause for the contradiction of the hypotheses.

Since every hypothesis is checked for its validity by searching the negative cover, an efficient representation of the set of dependencies is of significant importance for the performance of the induction algorithm. The set of dependencies is represented in `fdep` using a data structure called *FD-tree* which allows efficient use of space for the representation of sets of dependencies, and at the same time provides fast access to the elements of the set. FD-tree is presented in the following sub-section.

FD-tree is also used for the representation of the set of valid hypotheses which are the result of the algorithm. The valid hypotheses are added to the set as they are generated by the enumeration based on specialisation. Initially, the set of valid dependencies is empty. Each time a new valid dependency is generated, it is first tested against the existing set of valid dependencies. The valid hypothesis is added to the existing set of dependencies only if it is not covered by the existing set of dependencies. As in the case of the set of invalid dependencies representing the negative cover, efficient access to the elements of the set of valid dependencies is important for the efficient performance of the algorithm.

A problem which appears with the bi-directional algorithm lies in the enumeration of the hypotheses.

The hypotheses are enumerated from the most general functional dependencies to more specific dependencies as presented by Algorithm 8. The algorithm can be seen as performing depth-first search where the specialisation of the refuted candidates is based on invalid dependencies from the negative cover. Ideally, each of the hypotheses would be enumerated and tested against the negative cover only once. However, since the invalid dependencies are used for specialisation of refuted hypotheses, the enumeration can not be guided in a way which avoids testing some hypotheses more than one time. This is one of the main reasons for a bad performance of the algorithm for some domains.

Finally, the bottom-up algorithm has the best performance among the presented algorithms. As presented in Algorithm 9, the bottom-up algorithm starts with the positive cover which includes the most general dependencies. The positive cover is refined in each step by considering only one invalid dependency. A single FD-tree is used for the representation of the positive cover during the iteration through the invalid dependencies from the negative cover. In each step, only those dependencies from the positive cover which are refuted by the currently considered invalid dependencies are affected. Each of these dependencies is removed from the FD-tree (positive cover) and their specialisations which are not contradicted by the currently considered invalid dependency are added to the same FD-tree. Therefore, the algorithm generates only those hypotheses which are actually needed to avoid the contradiction of the invalid dependency considered in one step of the algorithm. In a way, this algorithm is the closest approximation of the intuitive idea of computing the positive cover by “inverting” the negative cover. In Section 7 we present empirical support for this analysis.

5.1.1. Data structure for representing sets of functional dependencies

Each of the presented algorithms for induction of functional dependencies manipulates sets of valid and/or invalid dependencies. In the previous sections it was assumed that covers for valid and invalid functional dependencies are represented simply as lists. In this section we introduce a data structure which is used for storing sets of functional dependencies and which allows efficient implementation of operations for searching more specific and more general dependencies of a given dependency.

To describe the data structure for representing sets of dependencies, we first introduce the *antecedent tree*.

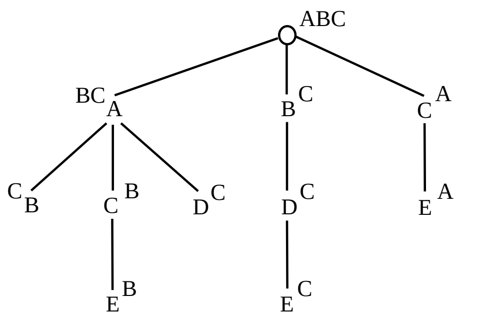

Fig. 1. An example of an FD-tree representing a set of functional dependencies.

We assume that attributes are totally ordered, so that for each pair of attributes $A_i, A_j \in R (i \neq j)$ we have that either A_i is higher than A_j or the opposite.

Definition 7 (Antecedent tree) *Given a relation scheme R, an* antecedent tree over R *is a tree with the following properties:*

1. *every node of the tree, except the root node, is an attribute from R;*
2. *the children of the node A_i are higher attributes; and*
3. *the children of the root are all attributes.*

Each path starting at the root of an antecedent tree represents a set of attributes, that will be interpreted as the antecedent of a functional dependency. The nodes of an antecedent tree are now labelled with sets of consequent attributes, as follows.

Definition 8 (FD-tree) *Given a relation scheme R and a set of functional dependencies F over R, the* FD-tree *of F is defined as follows: for every fd $X \rightarrow B \in F$ there is a path representing X, and each node from the path is labelled by the consequent attribute B. For every attribute $A \in R$, the unlabelled subtree consisting of all the nodes in FD-tree that include A in its label is called the A*-subtree.

Note that the dependencies composing the A-subtree can be identified in the FD-tree by visiting a subtree of nodes that are labelled by attribute A. An example of the FD-tree which is used for the representation of the set of functional dependencies $AB \rightarrow C$, $ACE \rightarrow B$, $AD \rightarrow C$, $BDE \rightarrow C$, $CE \rightarrow A$ is presented in Figure 1.

There are two important operations on the FD-tree which are used when generating the positive and negative covers. The first operation can be defined as follows: given an arbitrary dependency, search for a more specific dependency in the FD-tree. If the operation completes successfully, the result of the operation is a *more specific* dependency found in the set of dependencies. The second operation is similar: given an arbitrary dependency, it searches in the FD-tree for a dependency that is more general than the input dependency. The first operation is called *exists-specialisation* and the second *exists-generalisation*. Since both algorithms work in a similar manner only the operation *exists-specialisation* is detailed below as Algorithm 10. The description of the operation *exists-generalisation* can be found in [28].

Algorithm 10 (exists-specialisation)
Input: *the root of an FD-tree denoted as $Tnode$, and a dependency $X \rightarrow A$*
Output: *a dependency $Y \rightarrow A$ from FD-tree that is more specific than $X \rightarrow A$, if it exists*

```
function exists-specialisation( $Tnode, X, Y, A$ ): boolean;
begin
    exists-specialisation := false;
    if $X$ is empty
    then
        $Y$ := $Y$ + { Path from $Tnode$ to arbitrary leaf
                    of $Tnode$'s $A$-subtree };
        exists-specialisation := true;
        output $Y \rightarrow A$;
    else
        $X_1$ := first attribute from the list $X$;
        $Y_1$ := last attribute from the list $Y$;
        foreach $B \in [(label(Y_1) + 1) .. label(X_1)]$
        do if $B = X_1$ then $X := X - \{X_1\}$ fi ;
            if exists-specialisation( $Tnode$.child[$B$],$X$,
                                      $Y + \{B\}$,$A$ )
            then exists-specialisation := true;
                    exit;
            fi ;
        od;
    fi ;
end.
```

Algorithm 10 searches the FD-tree for a dependency $Y \rightarrow A$ that is more specific than the input dependency $X \rightarrow A$. Suppose that the antecedent of the input dependency is composed of attributes $X_1 X_2 \ldots X_k$ and the antecedent of the dependency $Y \rightarrow A$ is composed of attributes $Y_1 Y_2 \ldots Y_l$. The search process is completed, if a path from the root to a leaf of the A-subtree is found, such that the set of attributes Y forming the path includes the set of attributes X.

The core of Algorithm 10 can be described as follows. Suppose that each attribute from the set $\{X_1, \ldots, X_{i-1}\}$ matches one of the nodes on the path from the root of the A-subtree to the node Y_{j-1}. In this step, the algorithm searches for the descending node of the node Y_{j-1} that would form the next attribute in the path Y. Only attributes in the range from Y_{j-1} to the attribute X_i are considered. The reason for choosing the lower bound of the range is obvious, since descending nodes describe higher attributes. Similarly, there is no reason for investigating nodes that are higher than the attribute X_i, since the attribute X_i would be missing in such a path. If the next attribute on the path Y is the attribute X_i, than the next attribute from the list X (X_{i+1}) is considered in the next step of the algorithm. In the case that the next attribute on the path Y is not the attribute X_i, it is assumed that X_i will be matched later in the subtree.

Empirical results suggested that the average time needed for the operation *exists-specialisation* does not exceed $O(c * |R|)$, where c is a constant between 1 and 3.

5.1.2. Missing values and noise

`fdep` incorporates simple mechanisms for handling incomplete and/or incorrect data.[9] First, the *unknown* attribute values are in `fdep` treated as separate values which do not match any of the legal attribute values. Hence, the unknown attribute values are not used when generating the negative cover. For example, the tuples $[a_1, b_1, c_1]$ and $[a_1, ?, c_2]$ ('?' denotes unknown value), which have the schema (A, B, C), yield the invalid dependency $A \rightarrow C$. Since we do not know if dependencies $A \rightarrow B$ and $AB \rightarrow C$ are valid (invalid), we simply do not take them into account when the invalid dependencies contradicted by a given pair of tuples are enumerated.

Secondly, `fdep` implements a simple method for treating noisy data. During the generation of negative cover, the program counts the number of pairs of tuples which are the evidence for the contradiction of an invalid functional dependency. The user can define a threshold T which represents a number of (permitted) contradictions of an functional dependency which are treated as noise. Hence, if a functional dependency is contradicted by N pairs of tuples, and $N < T$, then we treat this dependency to be valid. In `fdep` we use the percentage to specify the number of permitted contradicting pairs of tuples. For example, if the threshold for the permitted contradicting pairs of tuples in a relation with n tuples is p, then the number of permitted

[9] See [15] for an approach to the related issue of approximate dependency inference.

contradictions is $n * (n-1) * p/200$. The experiments presented below include some results obtained when the threshold is used for eliminating the noise in data.

5.2. *Algorithms for inducing multivalued dependencies*

The algorithms for induction of multivalued dependencies were implemented in a prototype written in Sicstus Prolog; the prototype is referred to as `mdep`. The main intentions with the prototype were to study the properties of the presented algorithms and, in particular, to study the enumeration methods for the generation of hypotheses (multivalued dependencies) and the data structures for the representation of the sets of multivalued dependencies. This sub-section presents the work in progress.

For practical reasons, the multivalued dependencies are in the prototype represented using the dependency basis [1]. For a given set of attributes X, the *dependency basis of* X with respect to the set of all multivalued dependencies valid in a relation, written $DB(X)$, is a set composed of pairwise disjoint sets of attributes $\{Y_1, \ldots, Y_n\}$ such that the right-hand side Y of an arbitrary valid multivalued dependency $X \twoheadrightarrow Y$ is the union of some sets from $DB(X)$. The definition of the dependency basis and the presentation of its properties can be found in [1, 32].

The most important reason for using the dependency basis in our prototype is the efficient representation of dependencies. Namely, for a given left-hand side of the multivalued dependency, there exists a single dependency basis $DB(X)$ which determines all valid multivalued dependencies. As a consequence, a set of multivalued dependencies can be represented in a tree which is defined similarly to the above presented FD-tree. Furthermore, the dependency basis can serve as a suitable base for the definition of a method for enumerating multivalued dependencies. A detailed description of the use of the dependency basis in the algorithms for inducing multivalued dependencies is presented in [30].

In comparison to the algorithms for discovery of functional dependencies, in the multivalued case the space of hypotheses is more complex in terms of the relationships among the hypotheses and in terms of the number of possible multivalued dependencies. The program `mdep` can be used to induce multivalued dependencies from relations which contain up to 10 attributes and only few hundred of tuples. For example, `mdep` needs about 4 minutes on an HP4000 workstation to compute the satisfied multivalued dependencies from the *Lenses database* [31] which includes 6 attributes and 150 tuples. However, the results show that the prototype can serve as a basis for the development of a system which can induce multivalued dependencies from larger relations.

6. Related work

Most of the published work on dependency discovery has been done by Mannila and co-workers, who concentrated on inferring functional dependencies [18, 14, 20]. There are a number of resemblances between the algorithms of Mannila *et al.* and ours; however, their algorithms make more use of the particulars of functional dependencies, and are thus less easily generalised to other kinds of database constraints.

We will follow the survey in [20]. Algorithm 1 there, a naive algorithm enumerating and testing all possible fd's in no particular order, is similar to our enumeration Algorithm 1, restricted to fd-discovery. This is naive in the sense that it does not make use of the generality ordering; like our Algorithm 1, it is intended as a starting point for further discussion, rather than as a practically feasible algorithm.

Algorithm 2 bears some resemblance to our top-down Algorithm 2 restricted to fd's; however, there are some major differences. First of all, the algorithm of Mannila and Räihä assumes the consequent attribute to be fixed. This can be done because for fd's the hypothesis space consists of disconnected parts, one for each consequent attribute. However, this is not true in the general case; for instance, it is not true for mvd's. Running the algorithm for each possible mvd consequent separately would result in unnecessary duplication of work.

The most important difference between the two algorithms is that the loops iterating over all possible hypotheses and over all possible pairs of violating tuples are interchanged. In our Algorithm 2, we pop a possible fd off the agenda and then iterate over all possible pairs of tuples to see if this particular fd is satisfied. In contrast, the algorithm of Mannila and Räihä considers every possible pair of tuples in turn and updates the agenda accordingly. This means that their algorithm is in fact more of a bottom-up algorithm. It is hard to see, however, how to generalise this algorithm to other kinds of dependencies, since the test for satisfaction of a dependency is not relegated to a subroutine (as in our top-down algorithm), but hardwired into the top-level algorithm itself.

The authors of [20] then proceed with an algorithm based on transversals of hypergraphs. The algorithm employs the notion of a necessary set, which seems to be related to our negative cover. If $X \rightarrow A$ is refuted by two tuples t_1 and t_2, then the fd must be specialised by adding an attribute on which t_1 and t_2 disagree to the antecedent. Mannila and Räihä call the set of such attributes a *necessary set* for A. For every satisfied fd $Y \rightarrow A$, Y must contain at least one attribute of each necessary set for A. This means that we are only interested in *minimal* necessary sets, for which we compute minimal hitting sets (or transversals).

Example 8 *Consider again the relation r from Example 1, originally from [20]:*

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

The minimal necessary sets for A are B and CD.
The minimal necessary sets for B are A and D.
The minimal necessary sets for C are BD and AD.
The minimal necessary sets for D are B and AC.
The most general dependencies are given by the minimal hitting sets: $BC \rightarrow A$, $BD \rightarrow A$, $AD \rightarrow B$, $AB \rightarrow C$, $D \rightarrow C$, $AB \rightarrow D$, and $BC \rightarrow D$.

Clearly, the calculation of necessary sets is done bottom-up. There also seems to be some relation between necessary sets and the negative cover, which was computed in Example 3 as consisting of $B \rightarrow A$, $CD \rightarrow A$, $AC \rightarrow B$, $CD \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$, $AC \rightarrow D$, and $B \rightarrow D$. The antecedents of the first two and last two violated fd's correspond to necessary sets, while also in the remaining cases there seems to be some relation. However, it should be noted that the calculation of a necessary set is is necessarily based on a different pair of tuples than the calculation of the antecedent of a violated fd, since in the first case the tuples are required to disagree on the attributes, while in the latter case they are required to agree. Further investigations are needed on this point.

Finally, Mannila and Räihä give an algorithm for computing the most general satisfied dependencies based on consecutive sorts. This top-down algorithm is interesting because it maintains both a set of fd's known to be satisfied, and a set of fd's known to be violated, resembling our negative cover.

Recently, Mannila and Toivonen have generalised the fd-discovery problem to the problem of "finding all interesting sentences" [21]. The problem is stated in general terms:

> given a database $\mathbf{r}$, a language $\mathcal{L}$ for expressing properties or defining subgroups of the data, and an interestingness predicate q for evaluating whether a sentence $\phi \in \mathcal{L}$ defines an interesting subclass of $\mathbf{r}$. The task is to find the theory of $\mathbf{r}$ with respect to $\mathcal{L}$ and q, i.e., the set $Th(\mathcal{L}, \mathbf{r}, q) = \{\phi \in \mathcal{L} \mid q(\mathbf{r}, \phi) \text{ is true}\}$.

This problem can be instantiated to finding all fd's $X \rightarrow A$ over a relation scheme R that are satisfied by a given relation r as follows: $\mathcal{L} = \{X \mid X \subseteq R\}$, and $q(r, X)$ iff $X \rightarrow A$ is satisfied by r.

Mannila and Toivonen now give a general algorithm for constructing $Th(\mathcal{L}, \mathbf{r}, q)$ by employing a *specialisation relation* $\preceq$ that is monotone with respect to q, i.e. if $q(\mathbf{r}, \phi)$ (ϕ is interesting) and $\phi' \preceq \phi$ (ϕ' is less special than ϕ), then $q(\mathbf{r}, \phi')$ (ϕ' is interesting). Given this specialisation relation, their algorithm enumerates sentences from general to specific in order to find all sentences that satisfy the interestingness predicate. In the case of fd-discovery the specialisation relation of Mannila and Toivonen coincides[10] with our generality ordering, so that their algorithm actually enumerates functional dependencies from weak to strong, starting with largest antecedents. Alternatively, one could redefine the interestingness predicate so that $q(r, X)$ stands for "$X \rightarrow A$ is violated by r". Consequently, the algorithm of Mannila and Toivonen would find the set of violated fd's (the maximal negative cover).

Like our algorithms, the algorithm of Mannila and Toivonen abstracts away from the particulars of the induced sentences. One difference between their approach and ours is that they don't attempt to find a *cover* for the set of interesting sentences, but simply output all of them. It would however not be difficult to adapt their algorithm in this sense. Another difference is that they only consider a generate-and-test approach, since a bottom-up approach requires assumptions about the nature of the induced sentences.

A slightly different approach to the discovery of functional dependencies is taken by Huhtala and his colleagues [13]. Levelwise search [22] of the lattice of attribute sets is used for the enumeration of hypotheses.

[10]That is, while we call the most easily refuted dependencies 'most general', they call the most easily satisfied sentences 'most general'.

When the algorithm is processing a set of attributes X then all the dependencies $X - A \rightarrow A$, where $A \in X$, are tested. Testing the validity of hypotheses is based on *partitions*. A partition of r under X is the set of equivalence classes of tuple identifiers defined with respect to the values of attributes from X. The partitions are built incrementally in accordance with the traversal of the search algorithm through the lattice. The actual test of the validity of a hypothesis (functional dependency) is reduced to a simple comparison of the sizes of the partitions which are computed at the previous level and at the current level of the lattice.

In order to optimize the number of considered hypotheses, the space of hypotheses is pruned. As in the case of the computation of partitions, pruning in TANE algorithm takes advantage of the information computed at the previous levels of the lattice. There is a slight difference between fdep and TANE in the way the search space is pruned. fdep uses the generality ordering as the only means for pruning, while TANE uses the generality ordering and some additional criteria for pruning the search space. These criteria prune the dependencies $X \rightarrow A$, where an attribute $B \in X$ depends on some subset of $X - B$, and the dependencies which include a super-key on the left-hand side. Such dependencies are in fdep filtered after the complete positive cover is computed. The filtering algorithm simply removes all dependencies for which there exists a more general dependency in the positive cover.

7. Experiments

In this section we describe some empirical results regarding the efficiency of our fdep system. The experiments were done on a set of databases including the databases available from the UCI Machine Learning Repository [31], and the set of medical databases originating from University Medical Center at University of Ljubljana. All experiments were done on a Pentium II Personal Computer with 64MB of dynamic memory, operating under the LINUX operating system.

In Section 7.1 we describe our experiments to compare the different algorithms proposed in this paper. In Section 7.2 we compare the performance of two of our algorithms with TANE [13], currently the most efficient fd-discovery system to be found in the literature. In Section 7.3 we discuss the utility of the discovered fds in some domains.

7.1. *Comparison of the top-down, bi-directional and bottom-up fd-algorithms*

The parameters which were observed in experiments are: the number of relation tuples $|r|$, the number of attributes $|R|$ describing the relation, the percentage of permitted contradicting pairs of tuples ct, the number of dependencies in the negative cover n_{neg}, the number of dependencies which form the positive cover[11] n_{pos}, the CPU time used for construction of the negative cover t_1, the CPU time t_2 used for construction of the positive cover using the bi-directional algorithm, and the CPU time t_3 used for construction of the positive cover using the bottom-up algorithm. The top-down algorithm is not included in the experiments because the time for the computation of a positive cover exceeds 2 hours in all cases. The CPU time is measured using the UNIX `time` command; we report the CPU time used by a process.

The results of our experiments are presented in Table 1. The first thing to note is that the bottom-up algorithm significantly outperforms the bi-directional algorithm in all experiments. The main reason for this lies, as already stated above, in the method for the enumeration of the hypotheses. The enumeration used in the bottom-up algorithm considers each particular invalid dependency only once. Further, it generates only those hypotheses which are necessary to avoid contradiction of each particular invalid dependency. The enumeration used in the bi-directional algorithm can be seen as depth-first traversal of the space of hypotheses which is organized in a lattice. Even though the specialisation of refuted dependencies is based on the invalid dependencies, which heavily reduces the number of generated hypotheses, the same hypothesis can be generated more than once.

Secondly, in all experiments the time needed to compute the negative cover is greater than the time needed to compute the positive cover from the negative cover using the bottom-up algorithm. Since generation of invalid dependencies from a pair of tuples requires constant time, the complexity of the algorithm for the computation of the negative cover is dictated by the complexity of enumeration of all pairs of tuples from the input relation which is $O(n^2)$ where n is the number of relation tuples. For this reason, the algorithm may be

[11] These covers are minimal in the sense that they don't contain two fds such that one entails the other, but some fds may be redundant because they are entailed by two or more other fds in the cover. Our cover is a positive border as defined by [22].

Domain	$\|r\|$	$\|R\|$	ct	n_{neg}	n_{pos}	t_1	t_2	t_3
Rheumatology	362	17	0	507	1061	7.8	27.3	0.3
Rheumatology	362	17	0.01	2625	3159	9.2	577.5	1.5
Rheumatology	362	17	0.1	4970	5150	15.8	1859.1	2.0
Rheumatology	362	17	1	4022	4326	40.5	509.4	1.2
Rheumatology	362	17	3	2463	2967	60.9	133.0	0.6
Rheumatology	362	17	5	1744	2249	70.8	50.6	0.4
Primary tumor	339	18	0	141	224	4.7	1.75	0.1
Primary tumor	339	18	0.01	1110	1971	4.8	651.9	1.4
Primary tumor	339	18	0.1	4522	6355	9.1	19701.8	4.1
Primary tumor	339	18	1	5789	6641	26.0	8998.1	3.0
Primary tumor	339	18	3	3767	4965	42.6	2488.5	1.6
Primary tumor	339	18	5	2142	3412	49.7	347.8	0.8
Lymphography	148	19	0	883	2727	2.3	86.7	1.3
Lymphography	148	19	0.01	1971	5490	2.6	564.9	2.9
Lymphography	148	19	0.1	6362	13809	6.0	2285.4	6.5
Lymphography	148	19	1	5952	11794	19.7	679.3	2.9
Lymphography	148	19	3	2774	6053	28.8	117.5	1.0
Lymphography	148	19	5	1590	3766	32.4	34.2	0.5
Hepatitis	155	20	0	1151	4480	2.0.	110.9	1.4
Hepatitis	155	20	0.01	1338	4597	2.1	199.0	1.4
Hepatitis	155	20	0.1	1359	3327	2.9	250.6	0.9
Hepatitis	155	20	1	2264	3206	5.9	166.8	1.4
Hepatitis	155	20	3	2619	3820	10.1	305.8	1.5
Hepatitis	155	20	5	2225	3480	12.8	250.5	1.1
Wisconsin breast cancer	699	11	0	48	48	5.1	0.01	0.03
Wisconsin breast cancer	699	11	0.01	58	108	5.1	0.02	0.04
Wisconsin breast cancer	699	11	0.1	57	70	5.2	0.01	0.02
Wisconsin breast cancer	699	11	1	120	187	5.3	0.02	0.07
Wisconsin breast cancer	699	11	3	130	200	5.3	0.03	0.07
Wisconsin breast cancer	699	11	5	112	227	5.3	0.01	0.08
Annealing data	798	39	0	375	1584	13.0	17.6	0.3
Annealing data	798	39	0.01	719	2007	13.6	52.8	0.5
Annealing data	798	39	0.1	422	1504	14.3	9.1	0.25
Annealing data	798	39	1	163	724	15.1	0.8	0.08
Annealing data	798	39	3	90	497	15.7	0.4	0.04
Annealing data	798	39	5	68	443	16.0	0.2	0.05

Table 1
Comparison of the runtimes of the bi-directional and bottom-up fd-algorithms for various datasets, and the impact of the noise threshold on the size of covers and on the computation time of algorithms.

of limited practical use for the discovery of dependencies from very large relations (depending on the number and the type of attributes, as well).

Table 1 shows that in most cases, the time for computing the negative cover increases with the increase of the noise threshold. One of the reasons for this is the increased number of invalid dependencies in the negative cover which appears because of the relaxed condition for the invalid dependencies. In particular, the size of the negative cover increases when the actual invalid dependencies are dispersed into a number of more general invalid dependencies. Furthermore, when the noise threshold is greater than zero, the time for computing the negative cover includes the time used for eliminating "weak" invalid dependencies from the negative cover. The most expensive operation involved in filtering of the negative cover is counting the pairs of tuples which are the evidence for the contradiction of a given invalid dependency.

Furthermore, the time needed to compute the positive cover from the negative cover depends considerably on the noise threshold. At the beginning, the time increases very quickly as the algorithm treats a higher percentage of contradicting tuples as noise. After this, the time for the computation of the positive cover slowly decreases (as the threshold increases). The increase of the computation time can be explained by the increased number of functional dependencies in the positive cover. Similarly as was the case for the negative cover, the increase of the noise threshold causes the increase of the number of specific valid dependencies which are otherwise contradicted by a small number of pairs of relation tuples. In other words, the actual dependencies are dispersed into a number of

Domain	$\|r\|$	$\|R\|$	n_{pos}	`fdep`: $t_1 + t_2$	`fdep`: $t_1 + t_3$	Tane	Tane/MEM
Abalone data	4177	9	137	155.5	155.5	0.24	0.16
Annealing data	798	39	1584	30.6	13.3	*	*
Hepatitis	155	20	4480	112.9	3.4	15.4	11.6
Horse colic	300	28	81638	‡	72.3	*	*
Lymphography	148	19	2727	89.0	3.6	23.6	20.1
Primary tumor	339	18	224	6.4	4.8	29.0	28.6
Rheumatology	362	17	1061	35.1	8.1	6.0	5.6
Wisconsin breast cancer	699	11	48	5.2	5.2	0.25	0.22

Table 2
Empirical comparison of algorithms. The symbol ‡ denotes that the time for the computation of positive cover exceeds 2 hours. The asterix denotes that the program is unable to compute the positive cover due to the amount of the system memory required.

(weaker) valid dependencies. The decrease of the computation time in the second phase is a consequence of the decreased number of invalid dependencies in the negative cover as well as the shape of these invalid dependencies; only a small number of very specific invalid dependencies supported with the large number of contradicting pairs of tuples remain in the negative cover. As a consequence, the positive cover contains a small number of very general valid dependencies.

7.2. Comparison with TANE

The comparison of the performance of the bidirectional and bottom-up algorithms with the TANE algorithm [13] for discovery of functional dependencies from relations is presented in Table 2. The TANE algorithm was chosen for comparison because it is publicly available as well as because of its performance – as presented in [13], the TANE algorithm performs better than most existing implementations of algorithms for discovery of functional dependencies. Table 2 presents two versions of the TANE algorithm: the version denoted `Tane` which uses file representation of the relation, and the version denoted `Tane/MEM` which represents relations in the dynamic memory.

The results show that the TANE algorithm performs better when the number of the attributes of the input relation is small. In the case of the Wisconsin breast cancer and the Abalone data domains, the small number of tuples coincides with the small number of dependencies in the positive cover which results in a good performance of the levelwise algorithm. When the number of attributes of the input relation increases, the performance of `fdep` is better than the performance of the TANE algorithm. The reason for this lies in the large number of hypotheses which have to be considered by the levelwise algorithm in the case that the number of attributes is large – the set of generated hypotheses includes dependencies from the most general sentences to the most specific valid dependencies (positive cover). `fdep` performs good if the relation has large number of attributes, while, as pointed out in the previous section, its performance depends considerably on the number of tuples in the input relation.

7.3. Utility of dependencies

Knowledge discovery algorithms like the ones presented in this paper typically output all valid rules in a domain. This raises the question which of the discovered rules are actually useful for a practical task. While this is not the topic of this paper, we have started work on quantifying the novelty of rules. The novelty measure we used quantifies the extent to which the null-hypothesis of statistical independence between antecedent and consequent of an fd should be rejected. We used this heuristic to rank the induced fds.

Here are some of the highest ranked rules in several domains:

Lymphography:

```
[block_lymph_c,regeneration,lym_nodes_enlar,no_nodes]->[block_lymph_s]
[lymphatics,by_pass,regeneration,lym_nodes_enlar]->[lym_nodes_dimin]
```

Primary tumor:

```
[class,histologic_type,degree_of_diffe,brain,skin,neck]->[axillar]
[class,histologic_type,degree_of_diffe,bone_marrow,skin,neck]->[axillar]
[class,histologic_type,degree_of_diffe,bone,bone_marrow,skin]->[axillar]
```

Hepatitis:

```
[liver_firm,spleen_palpable,spiders,ascites,bilirubin]->[class]
[liver_big,liver_firm,spiders,ascites,varices,bilirubin]->[class]
[anorexia,liver_firm,spiders,ascites,varices,bilirubin]->[class]
```

Wisconsin breast cancer:

```
[uni_cell_size,se_cell_size,bare_nuclei,normal_nucleoli,mitoses]->[class]
[uni_cell_shape,marginal_adhesion,bare_nuclei,normal_nucleoli]->[class]
[uni_cell_size,marginal_adhesion,se_cell_size,bare_nuclei,normal_nucleoli]->[class]
```

Notice that in the last two domains the induced fds determine the class attribute. If we wanted to run a machine learning algorithm to obtain a predictor for the class attribute, such fds could be used to remove irrelevant attributes.

8. Conclusions

In this paper we have addressed the problem of discovering database dependencies from extensional data. We have approached this problem from a machine learning perspective by viewing it as an induction problem. This enabled us to employ a major tool from the computational induction field, *viz.* the generality ordering. We have distinguished between top-down approaches, which embody a generate-and-test approach, specialising dependencies that are found too general, bottom-up approaches, that start by generalising from the data, and bi-directional approaches, that mix elements from the two. We have developed practical algorithms, proved their correctness, and discussed their implementation. We have also presented experimental results, indicating that the pure top-down approach is infeasible, and the pure bottom-up approach outperforms the bi-directional approach. In comparison with the TANE system, our `fdep` system performs better for relations with many attributes.

Care has been taken so as to design the main algorithms as general as possible, pushing details about functional and multivalued dependencies to the lower levels of specialisation and generalisation routines. In this way, the algorithms may be adapted for other kinds of dependencies or constraints. We have compared our algorithms with other published algorithms for discovery of functional dependencies, and found that there are similarities as well as differences. These differences can partly be explained by our motivation to keep the algorithms as flexible as possible, whereas algorithms designed by others were not intended to go beyond fd-discovery. Yet, there does not seem to be a major penalty to be paid for this flexibility in terms of decreased effectiveness.

We suspect that our bottom-up approach through negative covers is also applicable for other machine learning tasks. For instance, suppose we know that `human(john)` is true but `woman(john)` is false. From this we can construct the clause `woman(X):-human(X)`, which is violated — we can consider it as a part of the negative cover, and use it to construct a more specific clause such as `man(X);woman(X):-human(X)`. An initial attempt can be found in [9].

Acknowledgements

This work was partially supported by the Esprit Long Term Research Project 20237 (*Inductive Logic Programming 2*), the PECO Network ILPnet, and a Joint Project with Central/Eastern Europe funded by the Royal Society. The work of Iztok Savnik has been supported by Research Project J2-7627 financed by the Ministry of Science and Technology, Republic of Slovenia. The comments of two anonymous reviewers have been helpful.

References

[1] C. Beeri, ‘On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases’, *ACM Trans. on Database Systems* 5:3, 1980, pp. 241-259.

[2] L. De Raedt (ed.), *Advances in Inductive Logic Programming*, IOS Press, 1996.

[3] L. De Raedt, ‘Logical settings for concept-learning’, *Artificial Intelligence* 95:1, 1997, pp. 187-201.

[4] T.G. Dietterich, B. London, K. Clarkson & G. Dromey, ‘Learning and inductive inference’, *Handbook of Artificial Intelligence*, Vol. III, P. Cohen & E.A. Feigenbaum (eds.), William Kaufmann, 1982.

[5] R. Fagin, ‘Horn clauses and database dependencies’, *J. ACM* 29, 1982, pp. 952-985.

[6] P.A. Flach, ‘Inductive characterisation of database relations’, *Proc. Fifth Int. Symposium on Methodologies for Intelligent Systems ISMIS’90*, Z.W. Ras, M. Zemankowa, M.L. Emrich (eds.), North-Holland, Amsterdam, 1990, pp. 371-378. Also ITK Research Report No. 23, Tilburg University.

[7] P.A. Flach, ‘Predicate invention in Inductive Data Engineering’, *Proc. European Conference on Machine Learning ECML’93*, P. Brazdil (ed.), Lecture Notes in Artificial Intelligence 667, Springer-Verlag, 1993, pp. 83-94.

[8] P.A. Flach, *Conjectures — an inquiry concerning the logic of induction*, PhD thesis, Tilburg University, 1995.

[9] P.A. Flach, ‘Normal forms for Inductive Logic Programming’, *Proc. International Workshop on Inductive Logic Programming ILP’97*, N. Lavrač and S. Džeroski (eds.), Lecture Notes in Artificial Intelligence 1297, Springer-Verlag, 1997, pp. 149-156.

[10] H. Gallaire, J. Minker & J.-M. Nicolas, ‘Logic and databases: a deductive approach’, *Computing Surveys* 16:2, 1984, pp. 153-185.

[11] G. Gottlob, ‘Subsumption and implication’, *Inf. Proc. Letters* 24, 1987, pp. 109-111.

[12] G. Gottlob & L. Libkin, ‘Investigations on Armstrong Relations, Dependency Inference, and Excluded Functional Dependencies’, *Acta Cybernetica* 9:4, 1990, pp. 385-402.

[13] Y. Huhtala, J. Kärkkäinen, P. Porka & H. Toivonen, *Efficient Discovery of Functional and Approximate Dependencies Using Partitions* (*Extended version*), Report C-1997-79, Dept. of Computer Science, University of Helsinki, 1997.

[14] M. Kantola, H. Mannila, K.-J. Räihä & H. Siirtola, ‘Discovering Functional and Inclusion Dependencies in Relational Databases’, *Int. Journal of Intelligent Systems* 7, 1992, pp. 591-607.

[15] J. Kivinen & H. Mannila, ‘Approximate inference of functional dependencies from relations’, *TCS* 149:1, 1995, pp. 129-149.

[16] D.W. Loveland & G. Nadathur, ‘Proof procedures for logic programming’, *Handbook of Logic in Artificial Intelligence*

and Logic Programming, Vol. 5, D.M. Gabbay, C.J. Hogger & J.A. Robinson (eds.), Oxford University Press, 1998, pp. 163-234.

[17] D. Maier, *The theory of relational databases*, Computer Science Press, Rockville, 1983.

[18] H. Mannila & K.-J. R¨aih¨a, 'Dependency inference, *Proc. 13th Int. Conf. on Very Large Databases*, 1987, pp. 155-158.

[19] H. Mannila & K.-J. R¨aih¨a, *The design of relational databases*, Addison-Wesley, Wokingham, 1992.

[20] H. Mannila & K.-J. R¨aih¨a, 'Algorithms for inferring functional dependencies from relations', *Data & Knowledge Engineering* 12, 1994, pp. 83-99.

[21] H. Mannila & H. Toivonen, 'On an algorithm for fi nding all interesting sentences', *Proc. Cybernetics and Systems '96*, R. Trappl (ed.), 1996, pp. 973-978.

[22] H. Mannila & H. Toivonen, *Levelwise search and borders of theories in knowledge discovery*, Report C-1997-8, Dept. of Computer Science, University of Helsinki, 1997.

[23] S.H. Muggleton (ed.), *Inductive Logic Programming*, Academic Press, 1992.

[24] S.H. Muggleton & L. De Raedt, 'Inductive Logic Programming: theory and methods', *Journal of Logic Programming* 19-20, 1994, pp. 629-679.

[25] G. Plotkin, 'A note on inductive generalisation', *Machine Intelligence 5*, B. Meltzer & D. Michie (eds.), North-Holland, 1970, pp. 153-163. 1970.

[26] G. Plotkin, 'A further note on inductive generalisation', *Machine Intelligence 6*, B. Meltzer & D. Michie (eds.), North-Holland, 1971, pp. 101-124.

[27] R. Reiter, 'Towards a logical reconstruction of relational database theory'. In *On conceptual modelling: perspectives from Artifi cial Intelligence, databases and programming languages*, M.L. Brodie, J.A Mylopoulos & J.W. Schmidt (eds.), Springer-Verlag, Berlin, 1984 pp. 191-233.

[28] I. Savnik, *Induction of Functional Dependencies from Relations*, IJS Report 6681, Jožef Stefan Institute, 1993.

[29] I. Savnik & P.A. Flach, 'Bottom-up induction of functional dependencies from relations', *Proc. AAAI '93 Workshop on Knowledge Discovery in Databases*, G. Piatetsky-Shapiro (ed.), 1993, pp. 174-185.

[30] I. Savnik & P.A. Flach, *Discovery of Multivalued Dependencies from Relations*, Technical Report, Jožef Stefan Institute, in preparation.

[31] C.J. Merz, P.M. Murphy, *UCI repository of machine learning databases*, University of California, Dept. of Information and Computer Science, 1996.
http://www.ics.uci.edu/~mlearn/MLRepository.html

[32] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume 1, Computer Science Press, Rockville, 1988.