Bottom-up Induction of Functional Dependencies from Relations

Iztok Savnik
Jožef Stefan Institute,
Jamova 39, 61000 Ljubljana, Slovenija
e-mail: iztok.savnik@ijs.si

Peter A. Flach
Institute for Language Technology & Artificial Intelligence,
Tilburg University, POBox 90153, 5000 LE Tilburg, the Netherlands
e-mail: flach@kub.nl

Abstract

Data dependencies express the presence of structure in database relations, that can be utilised in the database design process. The discovery of data dependencies can be viewed as an induction process. Like in induction, we can distinguish between top-down approaches and bottom-up approaches. In top-down approach, dependencies are generated and then tested against the given relation. Since each test requires $O(n^2)$ comparisons, where n is the number of tuples in relation, this can be computationally costly. We propose an alternative approach which differs from the top-down approach in that it starts with an analysis of the tuples in the relation: a *bottom-up* approach.

1 Introduction

Data dependencies are among the basic tools for modelling relational databases. They are used for the representation of constraints on the possible relations that can be instances of the relational scheme. Many types of data dependencies have been introduced and studied in the last two decades [8]. Of these types, functional and multivalued dependencies are the ones that are most commonly found in real environments. Consequently, functional and multivalued dependencies are quite extensively studied and applied in the database design process.

Usually, data dependencies are invented by the designer during the database design process. In the first step of the database design, the choice of the relational schemes that represent concepts in the Universe of Discourse is usually not influenced by the known data dependencies. They are used as constraints that can guide the detailed design process of the database conceptual scheme. In particular, data dependencies are used for checking database consistency, and for eliminating redundancy by decomposing the relation into smaller relations that still preserve the original information and obey the given set of data dependencies [8, 3].

In this paper, we concentrate on automating the construction of database dependencies from data. Such automated discovery of data dependencies from the existing relations can simplify the database design process and can be of a great help when relationships among the attributes of the relation are not obvious, due to the complex structure of the University of Discourse. Much work on the discovery of functional dependencies has been done by Mannila and Raiha; see [5] for an overview.

Discovery of database dependencies can also be viewed as an induction process, where the tuples in a relation represent instances of that relation, and dependencies represent hypotheses about the relation. In [3] it was shown how inductive learning techniques can be applied to the discovery of functional and multivalued dependencies. In [4] it was described how such induced dependencies could improve the design of the database. In [3], both types of dependencies are induced by a similar algorithm that incorporates the notion of *specialization* of the data dependency, which can be compared to the *refinement operator* as defined in the MIS [7]. Briefly, the algorithm starts with the set of the most general dependencies. Each dependency that is contradicted by the relation is replaced by the set of its specializations. One could call such an approach a generate-and-test or *top-down* approach.

We found the serious disadvantage of the algorithm proposed by Flach to be the $O(n^2)$ time complexity of the procedure for testing the validity of a given data dependency, where n is the number of tuples in the relation. Since the upper limit of the number of generated hypotheses grows exponentially with the number of attributes in the relation, this is a serious limitation that prevents application of the algorithm on larger database relations that usually appear in the real database environment.

In this paper, we propose an alternative approach which differs from the top-down approach in that it starts with an analysis of the tuples in the relation: a *bottom-up* approach. We have worked out this approach for the case of functional dependencies, but the idea is more general and can also be applied to multivalued dependencies. The basic idea behind the new algorithm for the induction of functional dependencies from the relation is derived from the fact, that all invalid dependencies that are contradicted by a given relation, can be identified by considering all pairs of relation tuples. The set of identified invalid dependencies can be represented by a *cover for invalid dependencies*, which includes only the most specific invalid dependencies. In comparison with the approach taken by Flach [3], the advantages of using the cover for invalid dependencies for the induction of valid dependencies are twofold. First, the complexity for testing the contradiction of the functional dependencies is reduced. Second, the specialization of the contradicted functional dependency is improved, since it is based on the most specific invalid functional dependency from the cover of invalid dependencies.

The paper is organised as follows. The Section 2 gives an overview of the concepts from the theory of relational databases, introduces the notion of *functional dependency* and defines the cover for valid dependencies. In the Sections 3 we define the cover for invalid dependencies and introduce the algorithm for calculating the cover for invalid dependencies is presented. The complete algorithm for the induction of functional dependencies from relations is described in the Section 4. The data structure intended for storing the cover of valid and invalid dependencies is proposed in the Section 5. The performance of the algorithm is analysed in the Section 6. We end the paper with some concluding remarks.

2 Preliminaries

Let the *relation* r be a subset of the Cartesian product of domains $D_1 \times D_2 \times \ldots \times D_k$. The *relational scheme* R is defined by the list of attributes $R = (A_1, A_2, \ldots, A_k)$, where the *domain* of the attribute A_i is D_i. Therefore, the relation is the set of ordered tuples $\{t_1, t_2, \ldots, t_n\}$, where the value of the i-th component of the tuple t_j is $t_j[A_i]$.

In general, attributes are denoted by the uppercase letters $A, B, \ldots$ and the set of attributes by the uppercase letters from the end of the alphabet $X, Y, Z, \ldots$. Relations are denoted by the lower-case letters $r, p, \ldots$. Tuples are denoted by the lower-case subscripted letters $t_1, t_2, \ldots$.

The functional dependency $X \rightarrow Y$ defines the constraint on the relation r, that has to be obeyed by each pair of relation tuples. The functional dependency $X \rightarrow Y$ is *valid* or X "*functionally determines*" Y, if for each pair of tuples $t_1, t_2 \in r$ it is not possible that the tuples agree on the components from the set of attributes X ($t_1[X] = t_2[X]$) and disagree on the components from the set of attributes Y ($t_1[Y] \neq t_2[Y]$). The dependency is *invalid*, if it is contradicted by two or more tuples of the relation r. The functional dependency will be in the following text denoted by *dependency*, since functional dependencies are the only type of dependencies considered in the paper.

Let r be a relation over the relational schema R and let F and G be the sets of dependencies that are satisfied by r. The sets F and G are *equivalent*, if any dependency in F can be deduced from the dependencies from G using Armstrong axioms [8] and vice versa. In other words, sets are equivalent, if the deductive closures of sets F and G are equal. Therefore, any set of functional dependencies that are satisfied by the relation r can be represented by an equivalent set of dependencies that is called a *cover* of the set of dependencies. The cover is *minimal* [8] for the set of dependencies F, if it is equivalent to the set of dependencies F and there is no redundant dependency with respect to the Armstrong axioms.

The *more-general-then* and the *more-special-then* relationships can be defined among functional dependencies with the equal right side of the dependency. Relationships can be effectively used in the algorithm for the induction of functional dependencies. As it will be shown latter, the more-general-then relationship is used as a clue for the enumeration of the hypotheses for dependencies.

Definition 1 *Let be X and Y sets of attributes such that $X \subseteq Y$, then the dependency $X \rightarrow A$ is* **more-general-then** *the dependency $Y \rightarrow A$ or $Y \rightarrow A$ is* **more-specific-then** *$X \rightarrow A$.*

Note, that if the dependency FD1 is more general than the dependency FD2, then we can conclude that FD2 is valid for a given relation r, if FD1 is valid and the opposite, the dependency FD1 is invalid if FD2 is invalid. For the purpose of the presentation of our algorithm for the induction of functional dependencies, we are interested in the construction of the cover, which is minimal considering the *more-general-then* relationship. Such cover will be denoted by the *positive cover*. The construction of the minimal cover for valid dependencies of the relation r, requires further work and is not the subject of this paper. The definition of the positive cover is the following.

Definition 2 *The set of dependencies F is a* **positive cover** *for the relation r iff*

1. *every $FD \in F$ is in the form of $X \rightarrow A$, where A is a single attribute,*
2. *for every valid dependency of the relation r, there exists the* **more general** *dependency that is an element of the positive cover.*

3 Cover for invalid dependencies

The validity of a dependency can be checked by searching for a pair of victim tuples, that contradict given dependency. Searching the contradicting pair of tuples in the relation r can be accomplished in time $O(n^2)$, which means that it is a time consuming operation when inducing dependencies from large relations.

To reduce the time complexity we introduce the notion of a *negative cover*, that include the set of invalid dependencies from which all dependencies that are contradicted by the given relation can be deduced. Analogous to the construction of the positive cover, the negative cover can be minimized using the *more-specific-than* relationship, which provides an ordering of the set of invalid dependencies with the same attribute on the right side. In this way, only the most specific invalid dependencies need to be stored in the negative cover. The negative cover can now be defined as follows.

Definition 3 *The set of invalid dependencies is the* **negative cover** *for the relation r iff*

1. *every right side of an invalid dependency is a single attribute,*
2. *for every invalid dependency that is contradicted by the relation r, there exists the* **more-specific** *invalid dependency that is an element of the negative cover.*

The contradiction test for the given dependency can now be accomplished by searching the negative cover for more specific invalid dependency. This is explained in the Section 4.

```
procedure find_invalid_fds;
begin
   invalid_fds := [];
   for each pair T1, T2 from r do begin
       split( T1, T2, X, Z );
       for each A in Z do
         if not exists_specialization( invalid_fds, X, Y, A ) then
            add( invalid_fds, X, A );
   end;
end {find_invalid_fds};
```

Figure 1: The procedure for the construction of the *negative cover*

The invalid dependencies of a given relation r can be identified by examining each pair of tuples and by constructing invalid dependencies that are contradicted by each of the selected pairs of tuples. Invalid dependencies are constructed from a pair of tuples t_1 and t_2 by splitting the set of attributes of the relation r into two sets. The first set of attributes Z includes those attributes A, where $t_1[A] \neq t_2[A]$ and the second set X collects those attributes B, where $t_1[B] = t_2[B]$. Invalid dependencies constructed from a given pair of tuples are dependencies of the form $X \rightarrow A$, where $A \in Z$.

The algorithm for the construction of the negative cover is presented in the Figure 1. Each invalid dependency constructed from the relation r is added to the negative cover, if it is not more general than an invalid dependency that is already in the negative cover. The existence of the more specific dependency in the negative cover for invalid dependencies is checked by the call of the procedure *exists_specialization*. For now, we suppose that the set of invalid dependencies is represented by the list data structure. More efficient data structure for the representation of the set of dependencies is described in the Section 6.

After the execution of the procedure *find_invalid_fds* there can still be some redundant dependencies in the constructed cover for invalid dependencies. They can be eliminated by filtering the cover, that results in the negative cover containing only the most specific invalid dependencies.

The time complexity of the given algorithm is $O(n * (n-1)/2 * h * nc)$, where n is the number of tuples in the relation, h is average number of constructed dependencies from one pair of tuples and nc is time needed for searching the negative cover for more specific dependency. Here, the unit of computation is the construction of one invalid functional dependency, which is multiplied by the time needed for checking the negative cover for the existence of a more specific dependency.

```
procedure find_positive_fds( X: attribute_set; A: attribute );
begin
   if exists_specialization( invalid_fds, X, Y, A ) then
      for each attribute Ai do
         if not (Ai in Y) and (Ai <> A) then
            find_positive_fds( X + [Ai], A )
   else
      if not exists_generalization( valid_fds, X, A ) then
         add( valid_fds, X, A );
end {find_positive_fds};
```

Figure 2: The procedure for the construction of the *positive cover*

4 Construction of the positive cover

The algorithm for the construction of the positive cover for valid dependencies is composed of two parts: hypothesis generation, and checking the validity of the hypothesis. The positive cover includes only the most general valid dependencies. An approach that is used for the generation of hypothesis is in order from the most general to more specific dependencies. The process of hypothesis specialization ends when the hypothesis is valid for a given relation.

Checking the validity of the dependency is realized using the negative cover. The dependency is invalid, if it is the generalization of an invalid dependency. Therefore, checking the consistency of the dependency is converted to the searching for the more specific dependency in the negative cover.

If the dependency is found to be invalid, it should be specialised by adding additional attributes to its left hand side. For example, the invalid dependency $A_1 \rightarrow A_4$, can be specialized by adding attributes A_2 or A_3 to the left side of the dependency. In this way, newly generated hypothesis are dependencies $A_1, A_2 \rightarrow A_4$ and $A_1, A_3 \rightarrow A_4$.

The specialization of the contradicted dependency is *guided* using the invalid dependency that is an element of the negative cover and was used for the contradiction of the given dependency. In this way, not all possible specializations of the contradicted dependency are generated, since it is not reasonable to generate specializations that are still generalizations of the invalid dependency that is used for the contradiction of the previous hypothesis. In terms of previous example, if $A_1, A_2 \rightarrow A_4$ is the invalid dependency from the negative cover that contradicts the hypothesis $A_1 \rightarrow A_4$, there is no reason for generating the hypothesis $A_1, A_2 \rightarrow A_4$.

The procedure *find_positive_fds* constructs the set of valid dependencies for a given right side of the dependency A. The algorithm is presented in the Figure 2. The parameters of the procedure are initially set to the most general dependency $[] \rightarrow A$, which is then checked and specialized by recursive calls. First, the validity of the functional dependency $X \rightarrow A$ is checked by searching the negative base for the more specific invalid dependency.

If the hypothesis is contradicted, then it is specialized considering the invalid dependency $Y \rightarrow A$, which is the part of the negative cover. In the case that the hypothesis is the valid functional dependency, the existence of more general valid dependency in the cover for valid dependencies is checked. If the hypothesis is not covered by the positive cover, than it is added to the cover.

After processing the procedure *find_positive_fds* there can still be some redundant dependencies in the constructed cover. The process of filtering the constructed positive cover is accomplished by examination of each dependency and removing those dependencies that are the specialization of the dependency from the cover.

The worst-case time complexity of the presented algorithm for the induction of the functional dependencies is $O(\boldsymbol{k} * 2^{2*\boldsymbol{k}-2})$, where $\boldsymbol{k}$ is the number of relation attributes. As will be seen in the following sections the complexity of searching more-specific-than or more-general-than dependency in the cover can be considerably reduced.

5 Data structure for the representation of the dependency cover

In the previous sections we supposed that the cover for valid and invalid functional dependencies are represented simply by the list of dependencies. Since every hypothesis (functional dependency) is checked for its validity by searching the negative and positive cover, better representation of the set of dependencies is of significant importance for the performance of the algorithm for the induction of functional dependencies.

To describe the data structure for the representation of the set of dependencies, we first introduce the *attribute-tree*. We suppose that attributes are ordered, so that for each pair of attributes $A_i, A_j \in R$ we can state that either A_i is higher than A_j or the opposite.

Definition 4 *The* **attribute-tree** *is a tree with the following properties:*

1. *Every node of the tree, except the root node, is an attribute.*
2. *The children of the node A_i are higher attributes.*
3. *The children of the root are all attributes.*

The attribute-tree can be used for the representation of the set of dependencies with the same right side of the dependency. The left side of each dependency is represented by the set of nodes, that are the elements of the path from the root of the tree to the one of its leaves. The tree representing the set of dependencies, which have the attribute A on the right side of the dependency, is denoted by the A-***subtree***. Note, that the single root represents the empty set.

The set of dependencies with different right sides of dependencies is represented by FD-***tree***, where the nodes that belong to the particular A-***subtree*** are labeled by the

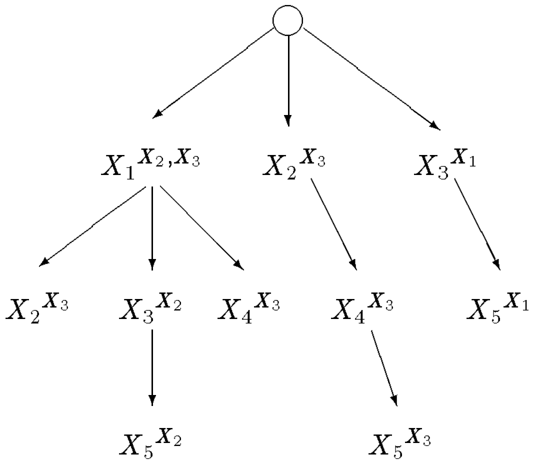

Figure 3: An example of tree representing the set of dependencies.

attribute A. Therefore, every ***FD-tree*** node is labeled by the set of attributes. The ***FD-tree*** is defined as follows.

Definition 5 *The* ***FD-tree*** *is an attribute-tree, where each dependency is represented as following:*

1. *Left side of the dependency $X \rightarrow A$ is represented by the path from the root to the leaf of the* ***A-subtree*** *and*
2. *each node from the path is labeled by the attribute A.*

Note, that the dependencies composing ***A-subtree***, can be identified in the ***FD-tree*** by visiting a subtree of nodes that are labeled by attribute A.

An example of the ***FD-tree*** which is used for the representation of the set of functional dependencies $\{\{X_1, X_2\} \rightarrow X_3$, $\{X_1, X_3, X_5\} \rightarrow X_2$, $\{X_1, X_4\} \rightarrow X_3$, $\{X_2, X_4, X_5\} \rightarrow X_3$, $\{X_3, X_5\} \rightarrow X_1\}$ is presented in the Figure 3.

There are two important operations on the ***FD-tree***, that are used when generating the positive and negative cover. The first operation can be defined as follows. Given the arbitrary dependency, search for the more specific dependency in the ***FD-tree***. If the operation completes successfully, the result of the operation is the *more specific* dependency found in the set of dependencies. The second operation is similar; given arbitrary dependency, it searches in the ***FD-tree*** for dependency that is more general than the input dependency. The first operation is called *exists_specialization* and the second *exists_generalization*.

```
function exists_specialization( Tnode, X, Y, A ): boolean;
begin
   exists_specialization := false;
   if Tnode <> NIL than
      if X is empty then
         Y := Y + [ Path from Tnode to the arbitrary A-subtree leaf ];
         exists_specialization := true;
         exit;
      else
         X1 := first attribute from the list X;
         Y1 := last attribute on the list Y;
         for each Atr labeled from label(Y1)+1 to label(X1) do
            if Atr = X1 then
               X := X-[X1];
            if exists_specialization( Tnode.child[ Atr ], X, Y+[Atr], A) then
               Y := Y+[Atr];
               exists_specialization := true;
               exit;
         od;
      fi;
   fi;
end {exists_specialization};
```

Figure 4: The procedure *exists_specialization*

Since both procedures operate in a similar manner and for the reason of the space limitations, only the procedure *exists_specialization* is described in the following subsection. The detailed description of the operation *exists_generalization* can be found in [6].

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

The procedure is presented in the Figure 4. The core of the procedure can be described as follows. Suppose that each attribute from the set $X_1, \ldots, X_{i-1}$ match 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 the matching with the attribute X_i would occur later in the subtree.

6 Performance

In this Section the performance of the procedure *exists_specialization* is analysed by studying the worst-case time complexity of the procedure. Second, the empirical results of running the program for the induction of functional dependencies on some real-world domains are presented.

6.1 Complexity of the operation *exists_specialization*

For the worst-case time complexity analysis of the operation *exists_specialization* we suppose that the FD-$tree$ is complete i.e. it includes all subsets of a given set of attributes $R = \{A_1, \ldots, A_k\}$. Given the input functional dependency $X_1, X_2, \ldots, X_l \rightarrow B$, the procedure *exists_specialization* searches for the more specific dependency in the FD-$tree$. The set of attributes $\{X_1, X_2, \ldots, X_l\}$ are ordered. The position of the attribute X_i in the ordering is denoted by $p(X_i)$. Only the results of the time complexity analysis are presented. The complete analysis can be found in [6].

The maximal number of visited FD-$tree$ nodes, during the execution of the procedure *exists_specialization* is specified by the following formula.

$$2^{p(X_l)-l+1} + 2^{p(X_{l-1})-l+1} + 2^{p(X_{l-2})-l+2} + \ldots + 2^{p(X_2)-2} + 2^{p(X_1)-1}$$

Since we can suppose that $p(A_i) = i$, we can see that in the worst case the procedure searches complete FD-$tree$ for the input dependency with the attribute A_l on the left side of the dependency. If the left side of the input dependency is the complete set of relation attributes $\{A_1, A_2, \ldots, A_l\}$, the number of visited vertices is $l + 1$, since the root of the FD-$tree$ is also included.

The empirical results showed that the average time needed for the operation *exists_specialization* does not exceed $O(c*k)$, where k is the number of the relation attributes and c is a constant. The average value of the constant c was in our experiments always around 1. The experiments were made on negative and positive covers for the relations that are presented in the following paragraphs.

6.2 Experimental results

The algorithm is implemented in the VAX Pascal programming language. It contains about 1500 lines of code. The domains that have been used for experiments include large number of attributes and tuples, so that they are comparable to real-world domains. The results of experiments are presented in the Figure 5. To reduce the number of induced

Domain	$\lvert r \rvert$	$\lvert R \rvert$	$\lvert X \rvert$	ct	t_1(CPU)	t_2(CPU)	N
Rheumatology	462	17	17	0	18min	27min	1191
Rheumatology	462	17	17	5	18min	23min	972
Rheumatology	100	17	17	0	1min	33min	2453
Rheumatology	100	17	17	5	1min	17min	1552
Rheumatology	100	17	7	5	1min	10min	1523
Lymphography	150	19	19	0	2min	16min	1248
Lymphography	150	19	10	0	2min	14min	1226
Lymphography	150	19	7	0	2min	7min	641
Lymphography	150	19	19	2	2min	7min	780

Figure 5: Experimental results

dependencies, the following two parametres can be set by the user. First, the number of attributes on the left side of dependencies, that are discovered by the program, can be limited. Secondly, since medical domains are noisy, the number of permited contradicting pairs of tuples can be set by the user.

Observed parameters are the following: the name of the domain, the number of relation tuples $|r|$, the number of attributes $|R|$ describing the relation, the maximal number of attributes on the left side of the discovered dependencies $X \rightarrow A$, denoted by $|X|$, the number of permited contradicting pairs of tuples ct, the CPU time used for the construction of the negative cover t_1, the complete CPU time used by the program t_2 and the number of discovered dependencies N.

7 Concluding remarks

The algorithm for the induction of the functional dependencies from relations was presented. The algorithm improves the performance of the algorithm proposed by Flach [3], by improving the part of the algorithm that checks the functional dependency for contradiction. For this purpose, the notions of invalid dependency and of negative cover were introduced. The improved performance of the algorithm allows its use in the real database environment.

Problems that require further work are the following. One of the most important problems concerns the large number of induced dependencies. Only some of the induced dependencies are meaningful and useful in the design process of the conceptual scheme of the modelling environment. Our further work will require the study of criteria that could eliminate useless functional dependencies. Another problem that is also closely connected to the elimination of the useless dependencies, is the elimination of the dependencies that can be deduced by the use of the Armstrong axiom expressing the transitivity property of the functional dependencies.

8 Acknowledgments

Part of this work has been carried out under the Esprit Basic Research Action 6020 (Inductive Logic Programming). Thanks are due to Heikki Mannila and two anonymous referees for useful discussions and suggestions.

References

[1] D.Angluin, C.H.Smith, *Inductive inference: theory and methods*, Computing Surveys 15:3, 238-269

[2] Catriel Beeri, *On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases*, ACM Trans. on Database systems, Vol.5, No.3, September 1980

[3] Peter A.Flach, *Inductive characterisation of database relations.* In Methodologies for Intelligent Systems 5, Z.W.Ras, M.Zemankowa, M.L.Emrich (eds.), North-Holland, Amsterdam, 371-378. Also ITK Research Report No.23, Tilburg University, November 1990

[4] Peter A.Flach, *Predicate invention in Inductive Data Engineering.* In Proc. European Conference on Machine Learning, P.Brazdil (ed.), Lecture Notes in Artificial Intelligence, Springer-Verlag.

[5] M.Kantola, H.Mannila, K.Raiha, H.Siirtola, *Discovering Functional and Inclusion Dependencies in Relational Databases*, Int. Journal of Intelligent Systems, Vol.7, 591-607, 1992

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

[7] E.Y.Shapiro, *Algorithmic program debugging*, MIT Press, 1983

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