FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances

Catharine Wyss, Chris Giannella, and Edward Robertson

Computer Science Department, Indiana University, Bloomington, IN 47405, USA
{crood,cgiannel,edrbtsn}@cs.indiana.edu

Abstract. Discovering functional dependencies (FDs) from an existing relation instance is an important technique in data mining and database design. To date, even the most efficient solutions are exponential in the number of attributes of the relation (n), even when the size of the output is *not* exponential in n. Lopes et al. developed an algorithm, Dep-Miner, that works well for large n on randomly-generated integer-valued relation instances [LPL 00a]. Dep-Miner first reduces the FD discovery problem to that of finding minimal covers for hypergraphs, then employs a levelwise search strategy to determine these minimal covers. Our algorithm, FastFDs, instead employs a *depth-first, heuristic driven* search strategy for generating minimal covers of hypergraphs. This type of search is commonly used to solve search problems in Artificial Intelligence (AI) [RN 95]. Our experimental results indicate that the levelwise strategy that is the hallmark of many successful data mining algorithms is in fact significantly surpassed by the depth-first, heuristic driven strategy FastFDs employs, due to the inherent space efficiency of the search. Furthermore, we revisit the comparison between Dep-Miner and Tane, including FastFDs. We report several tests on distinct benchmark relation instances, comparing the Dep-Miner and FastFDs hypergraph approaches to Tane's partitioning approach for mining FDs from a relation instance. At the end of the paper (appendix A) we provide experimental data comparing FastFDs with a third algorithm, *fdep* [FS 99].

1 Introduction

Functional dependencies (FDs) are a well-studied aspect of relational database theory [AHV 95]. Originally, the study of FDs was motivated by the fact that they could be used to express constraints which hold on a relation schema *independently* of any particular instance of the schema (for example, a business rule). Recently, a new research direction for FDs has emerged: the *dependency discovery* problem.[1] Given a relation schema, R, and an instance of the schema, r, determine all FDs which hold over r. Our paper addresses this problem.

[1] Like [FS 99], we do not use the term "dependency inference" to avoid confusion with the problem of inferring the dependencies implied by a given set of dependencies (which is not the problem of interest in this paper).

We develop an algorithm, *FastFDs*, for finding the canonical cover of the set of FDs ($\mathcal{F}_r$) of a given relation instance (r). FastFDs is based on a result in [MR 87,MR 94], showing that finding $\mathcal{F}_r$ is equivalent to finding the minimal covers of each of a set of hypergraphs (one for each attribute) constructed from the *difference sets* of the relation instance.[2] FastFDs carries out the following steps for each attribute, $A \in R$: (1) construct the difference set hypergraph, $\mathfrak{D}_r^A$, (2) compute the set of minimal covers of $\mathfrak{D}_r^A$ using depth-first, heuristic-driven search. We have implemented FastFDs and performance tested it on several distinct classes of benchmark databases (§1.3).

1.1 Motivations

Motivations for addressing the dependency discovery problem arise in several areas of endeavor: (a) data mining, (b) database archiving, and (c) data warehousing and OLAP.

Data mining concerns the semi-automated finding of interesting patterns in large collections of data to guide future decision making [RG 00]. Certainly the FD discovery problem meets this description. Paraphrasing from [HKP+ 99], consider a database of chemical compounds and their outcomes on a collection of bioassays. The discovery that an important property (such as carcinogenicity) of a compound depends functionally on some structural attributes can be extremely valuable.

Given a large database to be archived, discovered FDs can be used to save storage space. The database can be normalized with respect to these FDs and the normalized database stored alongside the FDs. The normalized database will likely be significantly smaller than the original database.

Data warehouses are typically much larger than other kinds of databases and are typically accessed and analyzed by OLAP query tools [RG 00]. Data warehouses and OLAP tools are based on the *data cube* multidimensional data model ([HK 2001] section 2.2). Discovered FDs can provide valuable semantic information which can be used to save time in the evaluation of OLAP operations on data cubes (for example, a drill-down).

Example 1. Consider the following 4D data cube representing the minimum distance traveled to work by the residents of the state of Indiana. The dimensions are `occupation` (for example, executive, administrative, and managerial occupations; sales occupations; etc.), `means` (train, bus, automobile, etc.), `city`, and `age`. The measure is minimum number of miles traveled. Thus, a cell (*sales occupations, automobile, Indianapolis, 18-25*) containing 15 indicates that people aged 18-25 in sales occupations in Indianapolis who travel by automobile to work, at a minimum, travel 15 miles.

Assume that `occupation`, `city` $\rightarrow$ `means` is an FD in the current snapshot of the cube. Further assume that the 2D cuboid, `occupation`, `city`, is also maintained in the warehouse due to some previous request. Suppose a drill-down query on `occupation`, `city`, and `means` is issued (i.e. find the minimum

[2] These were originally introduced in [MR 87] and termed *necessary sets*.

distance traveled by `occupation`, `city`, and `means`). The FD above implies that no aggregations need be carried out (the measures from the 2D cuboid need not be changed); this may result in significant savings of time. □

As an aside, we point out the following application of the dependency discovery problem in database design. Mannila et al. developed a database design and analysis tool called "Design-By-Example" [KMR+ 92] which makes use of discovered FDs. Their tool, among other things, allows a database designer to see all of the FDs which hold on an example table of a database. The designer can then modify the database schema if necessary, for example to normalize with respect to some of these FDs. The schema and instance, in this application, are quite small. As a result, less efficient (albeit arguably simpler) algorithms for finding FDs can be used than FastFDs.

1.2 Related Work

The first examination of the dependency discovery problem appeared in [MR 87] (full version [MR 94]). Algorithms were presented for finding a cover for the set of discovered dependencies. Their worst case complexity is exponential in R (although their behavior in practice was not investigated in [MR 94]). This exponential complexity was shown to be input optimal in the sense that the number of dependencies in any cover can be exponentially larger than the size of the relation instance. Thus, we cannot hope to find an input efficient algorithm for solving the problem in the worst case. However, output efficient algorithms may still be found.

Since [MR 87], research has gone into developing algorithms which behave well in practice: [FS 99,LPL 00a,HKP+ 99], among others. Several of these algorithms have been implemented and performance tested on both benchmark and synthetic data. Three recent algorithms of interest are Tane [HKP+ 99], *Dep-Miner* [LPL 00a,LPL 00b], and *fdep* [FS 99]. Both Tane and Dep-Miner search the attribute lattice in a *levelwise* fashion.[3] FastFDs differs from Tane in two respects: (1) Tane is not based on the hypergraph approach, and (2) Tane searches the subset lattice levelwisely, whereas FastFDs uses a depth-first search strategy. FastFDs differs from Dep-Miner only in that Dep-Miner employs a levelwise search to find hypergraph covers, whereas FastFDs uses a depth-first search strategy.

Two versions of fdep are implemented and tested in [FS 99]: bottom-up and bi-directional. The bottom-up version is shown to be superior so we do not consider the bi-directional version (henceforth, fdep, refers to the bottom-up version). Fdep first computes the *maximal negative cover* of the dependencies then iterates through this cover, refining the cover of the FDs at each iteration.[4]

[3] The computation of discovered dependencies levelwisely is an instance of the generalized algorithm for data mining given in [MT 97]. Another example instance is the *a priori* algorithm for finding association rules [AMS+ 94].

[4] The maximal negative cover of r is the set of all $X \to A$ such that $r \not\models X \to A$ but, for all strict supersets $\hat{X} \supsetneq X$, $r \models \hat{X} \to A$.

After one pass through the negative cover the minimal cover is computed. Note that the relation instance is only used to compute the "auxiliary information" contained in the negative cover and is not used to guide the search for $\mathcal{F}_r$ directly. This is similar to FastFDs and Dep-Miner, as both use the auxiliary information contained in the difference sets, so the search is similarly not guided by the relation instance itself. However, fdep does not employ a levelwise or a depth-first search of the subset lattice as FastFDs and Dep-Miner do. Rather, fdep utilizes techniques for learning general logical descriptions within a hypotheses space; examples of methods within this general framework that are akin to the fdep method are *version spaces* and *generalization/specialization hierarchies* [RN 95].

1.3 Purpose and Primary Contributions

The purposes of this paper are to introduce a novel method (FastFDs) for addressing the dependency discovery problem and to report the findings of performance testing of the three algorithms Dep-Miner, TANE, and FastFDs. We report the relative performance of fdep in appendix A. Our two primary contributions are as follows.

1. We introduce the algorithm FastFDs, which uses heuristic-driven, depth-first search to compute minimal FDs from a relation instance.
2. We report the results of several tests of the programs Dep-Miner, TANE, and FastFDs on three distinct classes of relation instances:
   (a) Random integer-valued instances of varying correlation factors,
   (b) random Bernoulli relation instances, and
   (c) existing "real-life" relations available online at the Machine Learning (ML) Repository site [MM 96].

Our experiments reveal that FastFDs is competitive for all of these relation instance classes when compared to TANE and Dep-Miner. Both FastFDs and Dep-Miner compute difference sets using the same code. Omitting this computation, FastFDs is significantly faster than Dep-Miner on the three classes of instances (tables 3, 4, 6). Thus, the heuristic-driven, depth-first search appears inherently more efficient than the levelwise calculation of covers of difference sets.

2 The Algorithm FastFDs

In this section, we describe the algorithm FastFDs. In §2.1, we present definitions and results establishing a connection between minimal hypergraph covers and the *canonical cover* of the set of FDs of r, $\mathcal{F}_r$. In §2.2, we describe a method, genDiffSets, for computing the difference sets based on [LPL 00a]. In §2.3, we present an algorithm for finding minimal covers of the difference sets, findCovers, and indicate how FastFDs uses findCovers to compute $\mathcal{F}_r$.

2.1 Definitions and Basic Results

Let R be a relation schema and r an instance of R. Let $n = |R|$ and $m = |r|$. In what follows, we will use n and $|R|$, m and $|r|$ interchangeably as necessary, to facilitate ease of reading.

Definition 1. *Let $X, Y \subseteq R$.*

1. *$X \to Y$ is a* functional dependency (FD) *over r iff for all tuples $t_1, t_2 \in r$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$. We write $r \models X \to Y$.*
2. *$X \to Y$ is* trivial *iff $Y \subseteq X$.*
3. *$X \to Y$ is* minimal *iff (i) $r \models X \to Y$ and (ii) $Z \subsetneq X$ implies $r \not\models Z \to Y$.*

Definition 2. *The* canonical cover *for the set of FDs that hold over r is*

$$\mathcal{F}_r = \{X \to A \mid X \subseteq R, A \in R, r \models X \to A, A \notin X, \text{ and } X \to A \text{ is minimal}\}.$$

FastFDs assumes an alternate characterization of FDs, which relies on the concept of a *difference set.*

Definition 3. *1. For $t_1, t_2 \in r$, the* difference set *of t_1 and t_2 is*

$$D(t_1, t_2) = \{B \in R \mid t_1[B] \neq t_2[B]\}.$$

2. The difference sets of r *are*

$$\mathfrak{D}_r = \{D(t_1, t_2) \mid t_1, t_2 \in r,\ D(t_1, t_2) \neq \emptyset\}.$$

3. Given a fixed $A \in R$, the difference sets of r modulo A *are*

$$\mathfrak{D}_r^A = \{D - \{A\} \mid D \in \mathfrak{D}_r \text{ and } A \in D\}.$$

The set $\mathfrak{D}_r^A$ will provide an alternate characterization of FDs over r with RHS A (see lemma 1, below).

	A	B	C	D	E	F
t_0	1	1	1	1	1	1
t_1	0	0	0	1	1	1
t_2	0	0	1	0	1	1
t_3	0	1	0	1	1	0
t_4	0	1	1	0	0	1
t_5	0	0	0	0	1	1

Table 1. Example relation instance, r_0.

Example 2. The example relation, r_0, in table 1 is used to illustrate definition 3. The table depicts a Bernoulli relation instance over schema A, B, C, D, E, F. In this case,

1. $D(t_0, t_3) = \{ACF\}$ and $D(t2, t3) = \{BCDF\}$ (for example).
2. $\mathfrak{D}_{r_0} = \{ABC, ABD, ACF, ADE, ABCD, CD, BF, BCDE, D, BCDF, BE, C, CDEF, BDF, BCE\}$.
3. $\mathfrak{D}^A_{r_0} = \{BC, BD, CF, DE, BCD\}$. □

Definition 4. *Let $\mathcal{P}(R)$ be the power set of R and $\mathcal{X} \subseteq \mathcal{P}(R)$. Then $X \subseteq R$* covers *$\mathcal{X}$ iff $\forall\, Y \in \mathcal{X}, Y \cap X \neq \emptyset$. Furthermore, X is a* minimal cover *for $\mathcal{X}$ in case no $Z \subsetneq X$ covers $\mathcal{X}$.*

Consider $X \subseteq R$, $A \notin X$ that covers $\mathfrak{D}^A_r$. Let $D \in \mathfrak{D}^A_r$. Then $X \cap D \neq \emptyset$. Thus, X distinguishes any two tuples that disagree on A. On the other hand, this is exactly what it means that $r \models X \to A$. Thus, we have:

Lemma 1. *Let $X \subseteq R$ and $A \notin X$. Then $r \models X \to A$ iff X covers $\mathfrak{D}^A_r$.*

This lemma implies the main result of this section:

Theorem 1 ([MR 87,MR 94]). *Let $X \subseteq R$, $A \in R - X$, and r be a relation instance over R. $X \to A$ is a minimal functional dependency over r if and only if X is a minimal cover of $\mathfrak{D}^A_r(r)$.*

Theorem 1 reduces the problem of computing $\mathcal{F}_r$ to the problem of finding minimal covers of $\mathfrak{D}^A_r$ for each attribute $A \in R$. In fact, any cover of $\underline{\mathfrak{D}^A_r} = \{D \in \mathfrak{D}^A_r \mid D' \in \mathfrak{D}^A_r \text{ and } D' \subseteq D \Rightarrow D' = D\}$ is also a cover of $\mathfrak{D}^A_r$. Thus, we need only retain *minimal* difference sets and compute minimal covers of $\underline{\mathfrak{D}^A_r}$.

Example 3. Revisiting example 2, careful inspection shows that $\underline{\mathfrak{D}^A_{r_0}} = \{BC, BD, CF, DE\}$. The minimal covers of $\underline{\mathfrak{D}^A_{r_0}}$ are $\{CD, BCE, BDF, BEF\}$. Hence the minimal FDs with RHS A are: $CD \to A, BCE \to A, BDF \to A, BEF \to A$. □

As a further optimization, if there is a pair of tuples that disagree *only* on attribute A (i.e. $\emptyset \in \mathfrak{D}^A_r$), then any $X \to A$ that holds over r must be trivial. This allows FastFDs and Dep-Miner to terminate early for such cases.

The idea of computing $\mathcal{F}_r$ by computing minimal covers of $\mathfrak{D}^A_r$ was first identified in [MR 87] (full version [MR 94]). There, the authors note that the elements of $\mathfrak{D}^A_r$ form edges in a *hypergraph*; thus, the problem of computing $\mathcal{F}_r$ reduces to the problem of computing minimal covers for hypergraphs. A systematic study of complexity issues of many hypergraph problems relevant to computer science is given in [EG 95]. A connection between a general framework for many data mining problems (including the dependency discovery problem) and the problem of finding hypergraph traversals is studied in [GKMT 97].

This method also underpins the Dep-Miner algorithm of [LPL 00a] (full version [LPL 00b]). The main difference between FastFDs and Dep-Miner is that Dep-Miner employs a levelwise technique to compute these covers, whereas FastFDs instead employs a heuristic-based depth-first search strategy.

2.2 Computing Difference Sets

FastFDs first computes the difference sets $\mathfrak{D}_r$ from r and R, using the procedure genDiffSets (figure 1, below). Then, for each attribute A, $\underline{\mathfrak{D}_r^A}$ is computed from $\mathfrak{D}_r$ and the minimal cover of $\underline{\mathfrak{D}_r^A}$ is computed using the procedure findCovers (figure 4). In this subsection we describe genDiffSets and in §2.3 we describe findCovers. The computation of $\underline{\mathfrak{D}_r^A}$ from $\mathfrak{D}_r$ involves a standard minimization technique (such as sorting) and is not explicitly described.

One of the features of Tane is that its complexity is linear with respect to $|r|$ [HKP+ 99]. On the other hand, there are $|r|(|r|-1)/2$ possible difference sets, so computing these difference sets takes time $O(|R||r|^2)$. In [LPL 00b], a method is given for reducing the time needed to compute difference sets from r in some cases. This method involves partitioning the relation, much like the partitioning that occurs at the first level in the Tane algorithm. The method of [LPL 00b] computes *agree sets*. Agree sets are the natural dual (under complementation) of difference sets. As such, the difference set definitions translate naturally into agree set counterparts.

Definition 5. *1. Given $t_1, t_2 \in r$, the* agree set *for t_1 and t_2 is*

$$A(t_1, t_2) = \{B \in R \mid t_1[B] = t_2[B]\}.$$

2. The agree sets of r *are*

$$\mathfrak{A}_r = \{A(t_1, t_2) \mid t_1, t_2 \in r\}.$$

The next lemma (lemma 2) relates $\mathfrak{D}_r$ to $\mathfrak{A}_r$. Given $\mathcal{X} \subseteq \mathcal{P}(R)$, we define

$$\mathcal{X}^{\complement} = \{R - X \mid X \in \mathcal{X}\}.$$

Lemma 2. *Given relation instance r over schema R and $A \in R$, $\mathfrak{D}_r = (\mathfrak{A}_r)^{\complement}$.*

Proof. Let $D(t_1, t_2) \in \mathfrak{D}_r$, then, by definition, $t_1[B] = t_2[B]$ for all $B \in R - D(t_1, t_2)$. Thus, $R - D(t_1, t_2) \in \mathfrak{A}_r$, so, $D(t_1, t_2) \in (\mathfrak{A}_r)^{\complement}$. On the other hand, let $X \in (\mathfrak{A}_r)^{\complement}$, then, $R - X \in \mathfrak{A}_r$. So, there exist $t_1, t_2 \in r$ such that $R - X = A(t_1, t_2)$. Thus, by definition, $t_1[B] \neq t_2[B]$ for all $B \in X$. So, $X \in \mathfrak{D}_r$. □

To compute $\mathfrak{D}_r$, we first compute the agree sets of r, $\mathfrak{A}_r$. We then complement elements of $\mathfrak{A}_r$ to arrive at $\mathfrak{D}_r$. The reason for this seemingly roundabout method of computing $\mathfrak{D}_r$, is that candidates for computing agree sets are generated not from pairs of tuples from the original database, but from pairs of tuples extracted from *stripped partitions* [LPL 00a]. For random integer-valued relations, the number of tuples in the stripped partitions is significantly less than the number of tuples in the entire database, speeding up the computation significantly.[5]

[5] For random Bernoulli relations, these "optimizations" in fact perform *worse* than a brute-force pair calculation approach, due to the fact that partition strips are (on average) half as long as the relation instance itself.

Definition 6. *Given $A \in R$, we define the following.*

1. *Tuples $t_1, t_2 \in r$ are* equivalent modulo A, *i.e. $t_1 \sim_A t_2$ iff $t_1[A] = t_2[A]$.*
2. *For $t \in r$, let $[\![t]\!]_A$ be the equivalence class of t under $\sim_A$. Then the* stripped partition [LPL 00a] *of A with respect to r is*

$$\pi_A = \{[\![t]\!]_A \mid t \in r \text{ and } |[\![t]\!]_A| > 1\}.$$

3. *The* stripped partition database [LPL 00a] *for r is*

$$\Pi_r = \bigcup_{A \in R} \pi_A.$$

4. *The* maximal stripped partition database *for r is*

$$\overline{\Pi_r} = \{\pi \in \Pi_r \mid \pi' \in \Pi_r \text{ and } \pi \subseteq \pi' \Rightarrow \pi' = \pi\}.$$

The following lemma is shown in [LPL 00b]:

Lemma 3. *Let $t_1, t_2 \in r$. If t_1 and t_2 do* not *appear together in some stripped partition $\pi \in \overline{\Pi_r}$, then t_1 and t_2 disagree on every attribute.*

Thus, it suffices to look at tuple pairs of individual equivalence classes in each $\pi \in \overline{\Pi_r}$ to calculate the set $\mathfrak{A}_r$. This insight is behind the algorithm in figure 1, genDiffSets, which generates the difference sets for a relation instance r.

method genDiffSets:
input: schema R and r a relation instance over R
output: difference sets for r, $\mathfrak{D}_r$
// *Initialize*:
1. $resDS := \emptyset$;
2. $strips := \emptyset$;
3. $tmpAS := \emptyset$;
// *Compute stripped partitions for all attributes*:
4. **for** $A \in R$ **do**
5.   compute stripped partitions for A and add to $strips$;
// *Compute agree sets from stripped partitions*:
6. **for** $\pi \in strips$ **do**
7.   **for** $t_i \in \pi$ **do**
8.   **for** $t_j \in \pi$, $j > i$ **do**
9.   add $A(t_i, t_j)$ to $tmpAS$
// *Complement agree sets to get difference sets*:
10. **for** $X \in tmpAS$ **do**
11.   add $R - X$ to $resDS$;

Fig. 1. genDiffSets: computing difference sets from a relation instance.

GenDiffSets uses a simplified method for calculating difference sets, compared to [LPL 00b]. We have found that this simpler method, implemented in C++ performs well for random integer-databases (§3.1).

2.3 Finding Minimal Covers of $\underline{\mathfrak{D}_r^A}$

The FastFDs algorithm utilizes a search procedure called *findCovers* (figure 4). The search method findCovers uses is as follows. Every subset of R which does not contain attribute A is a potential candidate for a minimal cover of $\underline{\mathfrak{D}_r^A}$. Consider a search tree representing a simple, "brute-force" method which generates the subsets of R not containing A in a depth-first, left-to-right fashion. Such a search tree is shown in figure 2, for $R = \{A, B, C, D, E, F\}$. Each node in the tree represents a subset of $R - \{A\}$, given by the labeling along the path from the root to that node. There are $2^{|R|-1}$ such nodes. Note that in generating the subsets without repeats, we have "ordered" the attributes lexically: $B > C > D > E > F$.

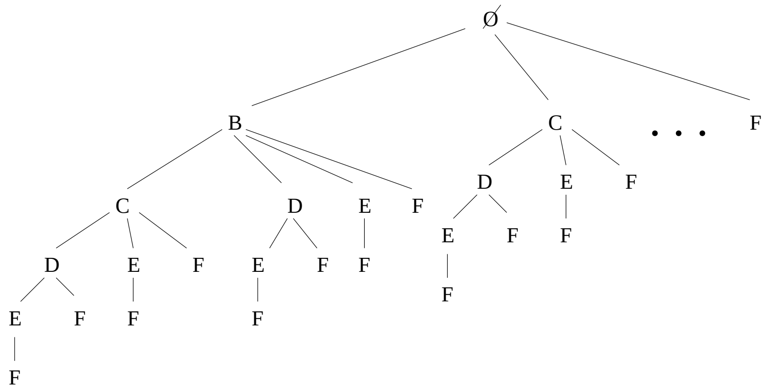

Fig. 2. Generating subsets of $R - \{A\}$.

The optimized search method findCovers uses constructs a search tree similarly using an attribute ordering, however this new ordering changes as we descend in our search. At each node in our new search tree, we *order the remaining attributes according to how many difference sets they cover* (that have not been already covered at a node above this point). We break ties in this ordering lexically. This search pattern is illustrated in figure 3 for the attribute A and $\underline{\mathfrak{D}_{r_0}^A}$ from example 3. At each node in the search tree, we track the remaining difference sets and the ordering of the remaining attributes.

The method shown in figure 3 is the search strategy used by the method findCovers, which underpins the FastFDs algorithm (figure 4). Note that leaf nodes appearing in figure 3 are enumerated. Each leaf node represents one of two base cases in our search.

1. If we arrive at a node where there are still difference sets left to cover, but no attributes are left to unfold, fail (there are no FDs down this branch) – see leaves 5, 7, and 8 in figure 3.

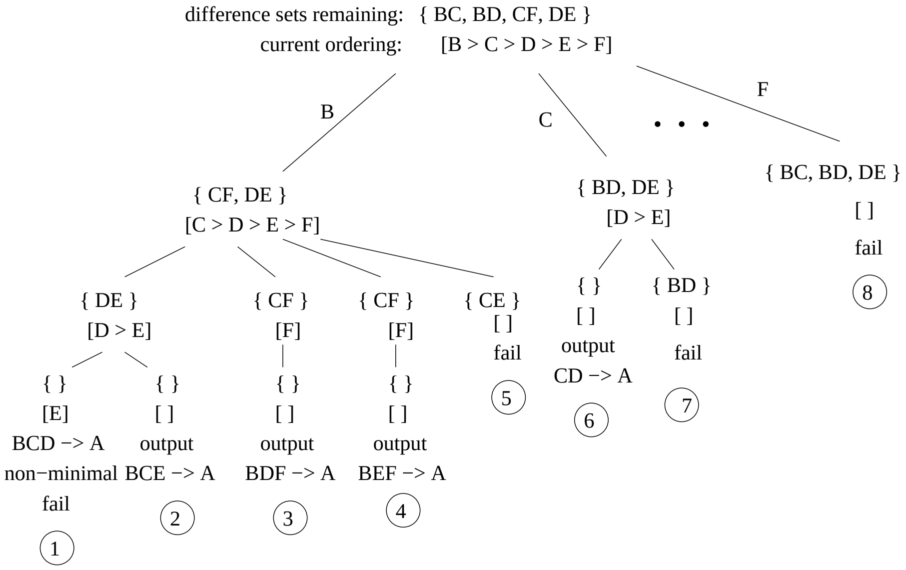

Fig. 3. Searching the subset lattice for minimal covers.

2. If we arrive at a node where there are no difference sets left, this implies one of two situations.
   (a) The FD may not be minimal; in this case fail (leaf 1). The minimality check involves testing each immediate subset of the current LHS to see if this subset also provides a legitimate LHS.
   (b) Otherwise, output the subset built along the path to the current leaf as a LHS for an FD for the current attribute (leaves 2, 3, 4, and 6).

We use a simple, greedy heuristic at each node: we will search the attributes that cover the most remaining difference sets first. This heuristic works well in practice (see §3), but can be fooled into performing excess work. Note that in figure 3 (leaf 1), the FD $BCD \rightarrow A$ is not minimal, since $CD \rightarrow A$ is also an FD (leaf 6). In this case, it is the way we break ties (lexically) that causes the extra branch; in general, the excess work is not straightforward to characterize. Thus, we must perform a minimality check on each LHS as it is generated, before we output the corresponding FD.[6]

As seen in figure 3, the heuristic used at each node is based on a partial ordering of R. We now formally define this ordering. Let $\mathcal{X} \subseteq \mathcal{P}(R)$. For $A \in R$,

[6] Another source of wasted work is the branches we traverse that do not contain a cover of the difference sets. We can avoid some of this work by inserting a "cover check" at each node that determines if the remaining attributes in the order indeed cover the difference sets. In practise, this reduces the number of recursive calls by roughly 30%, but the savings is moot because adding the cover check at each node causes the program to take longer than simply traversing these "fail" branches.

we define the measure $\mathbf{Cov}(\mathcal{X}, A) = |\{Z \in \mathcal{X} | A \in Z\}|$. The attribute ordering, $>$ is as follows. For $A, B \in R$:

$$A > B \text{ iff } \begin{cases} \mathbf{Cov}(\mathcal{X}, A) > \mathbf{Cov}(\mathcal{X}, B) \text{ or} \\ \mathbf{Cov}(\mathcal{X}, A) = \mathbf{Cov}(\mathcal{X}, B) \text{ and } A \text{ is lexicographically larger than } B. \end{cases}$$

The heuristic-based, depth first search seen in figure 3 is encapsulated in the recursive method findCovers (figure 4).

To calculate $\mathcal{F}_r$, we search a tree such as that in figure 3 for each attribute $A \in R$. Note that if $\underline{\mathfrak{D}_r^A} = \emptyset$, no tuples differed on A, so the only minimal FD for A is $\emptyset \to A$. Also, as previously mentioned, if $\emptyset \in \underline{\mathfrak{D}_r^A}$, there are no non-trivial FDs for attribute A. The algorithm FastFDs (figure 5) intercepts these cases, and sets up the findCovers recursion correctly for $A \in R$.

method findCovers:
input: attribute $A \in R$ (RHS)
  original difference sets, $\underline{\mathfrak{D}_r^A}$
  difference sets not thus far covered, $\mathfrak{D}_{curr}$
  the current path in the search tree, $X \subseteq R$
  the current partial ordering of the attributes, $>_{curr}$
output: minimal FDs of the form $Y \to A$

<u>Base Cases</u>:
if $>_{curr} = \emptyset$ but $\mathfrak{D}_{curr} \neq \emptyset$ **then**
  return; // *no FDs here*
if $\mathfrak{D}_{curr} = \emptyset$ **then**
  if no subset of size $|X| - 1$ of X covers $\underline{\mathfrak{D}_r^A}$ **then**
    output $X \to A$ and return;
  else return; // *wasted effort, non-minimal result*

<u>Recursive Case</u>:
for attributes B in $>_{curr}$ in order **do**
  $\mathfrak{D}_{next}$:= difference sets of $\mathfrak{D}_{curr}$ not covered by B;
  $>_{next}$ is the total ordering of $\{\hat{B} \in R | \hat{B} >_{curr} B\}$ according to $\mathfrak{D}_{next}$;
  findCovers(A, $\underline{\mathfrak{D}_r^A}$, $\mathfrak{D}_{next}$, $X \cup \{B\}$, $>_{next}$);

Fig. 4. findCovers: find minimal covers for $\underline{\mathfrak{D}_r^A}$

Complexity of FastFDs Recall that $|r| = m$, $|R| = n$, and let $|\mathcal{F}_r| = K$. Let us consider the steps involved in FastFDs.

1. First, we must compute the difference sets for r, $\mathfrak{D}_r$; this takes $O(nm^2)$ time. To compute $\underline{\mathfrak{D}_r^A}$ from $\mathfrak{D}_r$, we minimize over sets $X \in \mathfrak{D}_r$ which contain A. Let $d = \overline{|\mathfrak{D}_r|}$, then minimization takes time $O(d \log(d))$; in terms of m and n, this is $O(nm^2 log(nm^2))$.

```
method FastFDs:
input: relation instance r with schema R
output: canonical cover of minimal FDs over r, F_r

1. 𝔇_r := genDiffSets(R,r);
2. for A ∈ R do
3.     compute 𝔇_r^A (underlined) from 𝔇_r;
4.     if 𝔇_r^A (underlined) = ∅ then
5.         output ∅ → A;
6.     else if ∅ ∉ 𝔇_r^A (underlined) then
7.         >_init is the total ordering of R − {A} according to 𝔇_r^A (underlined);
8.         findCovers(A, 𝔇_r^A (underlined), 𝔇_r^A (underlined), ∅, >_init);
```

Fig. 5. FastFDs: compute minimal FDs from a relation instance.

2. Given that $|\mathcal{F}_r| = K$, the complexity of findCovers is $O((1+w(n))K)$, where $w(n)$ is a function representing the wasted work due to our imperfect search heuristic.[7]
3. Since we call findCovers for each attribute $A \in R$, we have that the main loop in FastFDs (lines 2-8) takes time $O(n(1+w(n))K)$.

Altogether, our method has worst case time complexity

$$O(nm^2 + nm^2 \log(nm^2) + n(1+w(n))K).$$

However, the space complexity of FastFDs is exponentially less than that of its levelwise-search counterparts, Dep-Miner and Tane. FastFDs is limited to $O(dn)$ space. The measure d is bounded by $m(m-1)/2$ and in practice seems to be much less than this. The levelwise approaches of Tane and Dep-Miner consume an amount of space directly proportional to the largest set of candidates generated, s_{max}. Thus, FastFDs is significantly faster than the levelwise approach for large n, since the size of s_{max} can be exponential in n. This trend is especially manifested in the case of Bernoulli relation instances where the average length of FDs is predicted to be $n/2$ or larger.

3 Experimental Results

FastFDs has been implemented in C++. The program has been tested on a 700MHz Athlon system with 128MB RAM, running Red Hat Linux 6.1.[8] We have also implemented our own version of Dep-Miner, which uses the same code as FastFDs for reading input relations and calculating $\underline{\mathfrak{D}_r^A}$. Our versions of both Dep-Miner and FastFDs run entirely in main memory. The version of Tane we use is available on the web [Tane].[9]

[7] For $n \leq 60$, $w(n) < 3$ on the benchmark databases from §3.
[8] The source code for FastFDs is available from the authors upon request.
[9] We also indicate the performance of Tane/mem in the following tests.

3.1 Integer-Valued Relation Instances

Our first set of experiments involved randomly-generated integer relation instances. Table 2 (below) illustrates the performance of TANE, Dep-Miner and FastFDs on such instances. Each instance is generated according to a *correlation factor* (CF), as in [LPL 00a]. The higher the CF, the fewer distinct values appear in each column.[10] As the CF increases, the length of the stripped partitions in our difference set computation also increases; thus the difference set calculation becomes more time-consuming as the CF progressively increases.

Instance	Time (seconds)					
	CF = 0.0		CF = 0.5		CF = 0.9	
$\|r\|$	Dep/Fast†	TANE‡	Dep/Fast†	TANE‡	Dep/Fast†	TANE‡
50,000	5	13	6	10	6	7
100,000	12	38	13	27	13	17
150,000	19	78	21	50	19	29
200,000	25	133	29	78	26	41
250,000	33	207	37	115	33	57
300,000	40	478	46	360	41	158
350,000	48	∘	69	476	54	196

†Dep-Miner and FastFDs took the same amount of time ($\pm 0.01s$).
‡TANE/MEM took approximately the same amount of time as TANE ($\pm 1s$).
∘ indicates TANE took longer than 2 hours and was aborted.

Table 2. Results on Random Integer-Valued Relation Instances; $|R| = 20$.

Qualitative Analysis of Table 2. Our results in table 2 are in line with those reported in [LPL 00a]. However, there are several considerations which our tests on random integer-valued instances reveal.

- Less than 0.1% of the computation time for Dep-Miner and FastFDs is spent searching for FDs; over 99.9% is spent computing difference sets.
- As the CF increases, TANE's performance comes progressively closer to that of Dep-Miner and FastFDs (figure 6).

We believe the results in table 2 illustrate the difference between the best case of the difference set computation (which appears linear in $|R|$ and $|r|$, and occurs only once for each instance) and the fact that TANE must compute a linear partition at each stage of it's computation.

In conclusion, our results on randomly generated integer-valued instances with 20 attributes indicate that Dep-Miner and FastFDs are 2-3 times faster for CFs from 0.0-0.9.[11]

[10] The range of distinct values for each attribute is $(1 - CF) * num\_tuples$.
[11] Similar conclusions hold for 1-31 attributes. The version of TANE we utilized [Tane] is limited to less than 32 attributes.

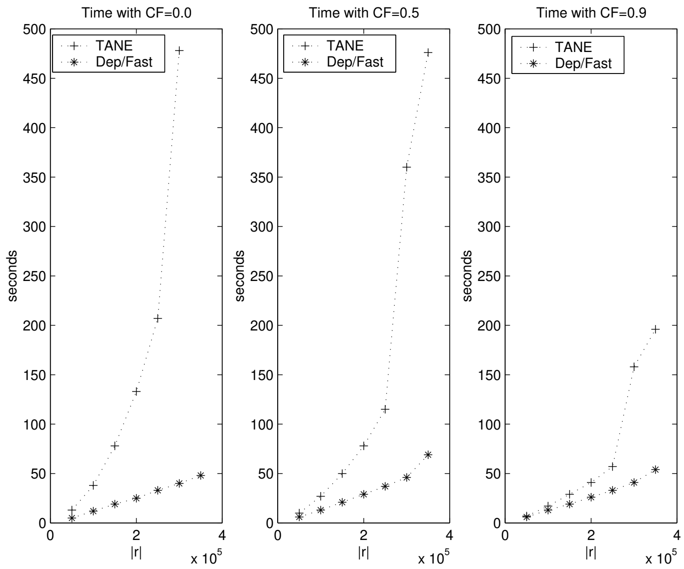

Fig. 6. Results on random integer-valued instances (table 2).

Instance	Time† (seconds)					
	CF=0.0		CF = 0.5		CF = 0.9	
$\lvert R\rvert$	FastFDs	Dep-Miner	FastFDs	Dep-Miner	FastFDs	Dep-Miner
10	0‡	0‡	0‡	0‡	0‡	0‡
20	0‡	0.1	0.1	0.2	0.1	0.2
30	0.2	0.3	0.8	1.6	1.0	2.1
40	0.6	1.2	3.7	9.5	5.5	11.0
50	2.3	3.8	14.3	27.5	20.8	48.0
60	5.6	8.6	41.6	81.2	64.7	134.4

†Time reported is time to search for FDs.

‡Less than $0.1s$.

Table 3. Results on Random Integer-Valued Relation Instances; $|r|$ fixed at 10,000; the time to compute difference sets is omitted.

Qualitative Analysis of Table 3. Table 3 illustrates the relative performance of FastFDs and Dep-Miner for a fixed number of tuples (10,000) as the number of attributes varies from 10 to 60.[12] Note that as the CF increases, FastFDs becomes progressively more efficient than Dep-Miner. For a CF of 0.0, FastFDs is $1\frac{2}{3}$ times as efficient, whereas for a CF of 0.9, FastFDs is over twice as efficient. Note that as the CF increases, the average length of the minimal FDs computed increases slightly, from under 2 for a CF of 0.0 to over 3 for a CF of 0.9. This trend may help explain the comparatively better performance of FastFDs as the CF increases, and is born out by our experiments with Bernoulli instances (§3.2), where Dep-Miner becomes drastically less efficient than FastFDs as the average length of the minimal FDs increases to $|R|/2$.

As we will see in the next sections (§3.2, §3.3), random integer-valued relation instances with low CFs do not appear to be good predictors of performance on real-life instances such as those from the ML repository. Bernoulli relation instances appear notably better in this regard.

3.2 Bernoulli Relation Instances

Our second set of experiments involved randomly generated Bernoulli relation instances. A Bernoulli relation involves only two values (usually 0 and 1). In this respect, Bernoulli relations are similar to the "market basket" relations considered in the frequent itemset problem [AMS+ 94].

Random Bernoulli instances are an interesting testbed for programs finding minimal FDs. Whereas random integer-valued instances seem quite amenable to the search methods of FastFDs, Dep-Miner and TANE, there tend to be few minimal FDs. This is not the case in Bernoulli instances. The average length of minimal FDs in random Bernoulli databases has been investigated and is straightforward to quantify [DKM+ 95]. For n attributes, the expected average length of minimal FDs is $n/2$ when the number of tuples $m = 2^{n/4}$. This length represents the maximal possible number of subsets at any one level in the power-set lattice of n attributes, and in practice the number of minimal FDs returned when $m = 2^{n/4}$ increases exponentially with n.

Since the case of minimal FDs of size $n/2$ involves relatively small-sized databases for n less than 30, the impact of the $O(nm^2)$ difference set calculation is minimized for Dep-Miner and FastFDs. Whereas in the random integer-valued databases in §3.1 less than 0.1% of the computation time of Dep-Miner and FastFDs is spent searching for FDs, in random Bernoulli instances where $m = 2^{n/4}$, over 99% of the time is spent searching for FDs (as opposed to computing difference sets).

Another reason this particular case is interesting is that, since there are so many subsets of size $n/2$, levelwise algorithms require exponential space to store the candidate sets at intermediate levels near to $n/2$. Thus, this test case allows

[12] In Table 3, the time for computing difference sets is omitted, since it is identical for both Dep-Miner and FastFDs.

us to see the impact of the exponential space usage of the levelwise algorithms, versus the $O(nm^2)$ space usage of FastFDs.

Table 4 (below) illustrates the relative performance of TANE, Dep-Miner and FastFDs for the case of Bernoulli databases, where $m = 2^{n/4}$ and n ranges from 20 to 28.

Instance			Time (seconds)		
$\lvert R\rvert$	$\lvert r\rvert = 2^{\lvert R\rvert/4}$	$\lvert\mathcal{F}\rvert$	fastFDs	TANE†	Dep-Miner
20	32	7.6×10^4	2	5	195
21	38	1.4×10^5	5	15	601
22	45	2.6×10^5	11	36	1893
23	54	5.4×10^5	28	84	5169
24	64	1.0×10^6	69	392	14547
25	76	2.0×10^6	171	•	38492
26	91	3.9×10^6	428	•	•
27	108	7.4×10^6	1081	•	•
28	128	1.6×10^7	3167	•	•

†TANE/MEM exhibited similar results.

• indicates program ran out of main memory.

Table 4. Results on Bernoulli Relation Instances.

Qualitative Analysis of Table 4. The number of FDs is increasing exponentially with n; thus it is not surprising that the time of all three programs does as well. However, here we can see the impact of the exponential space usage on TANE and Dep-Miner. Initially (for n from 20 to 23), the computation time of TANE is doubling, similarly to FastFDs computation time. Then, at $n = 24$, we see a jump in the execution time of TANE, corresponding to heavy utilization of disk swap space. For $n \geq 25$, execution of TANE was aborted after it ran out of memory. The results for Dep-Miner are similar. Although Dep-Miner's space usage seems slightly better than TANE's, Dep-Miner's performance is over two orders of magnitude worse than FastFDs.

In contrast, FastFDs is never in danger of running out of memory. [13] This is crucial, both because memory is a scarcer resource than CPU cycles, and because the CPU cycles are utilized much more efficiently when swapping is avoided.

Remark 1. One salient result of our tests is that for Bernoulli instances, the difference set calculation is less efficient than for random integer-valued instances for larger m. In fact, for Bernoulli instances, a brute-force approach to computing difference sets is *more* efficient than the optimized approach given in [LPL 00a]. Thus, there are most likely better optimizations for computing the difference sets for k-valued relation instances (small k).

[13] Memory usage remains below 0.5MB for all n shown.

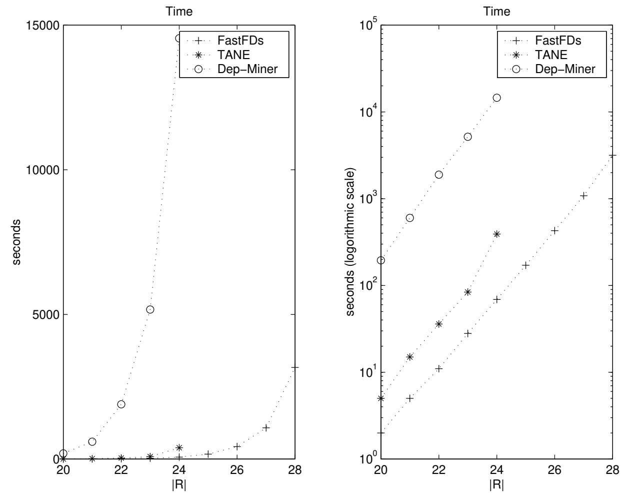

Fig. 7. Results of table 4: right is logarithmic scale for the y-axis (execution time).

3.3 Machine Learning Repository Relation Instances

Our third set of experiments involved non-synthetic relations extracted from the Machine Learning (ML) Repository, online [MM 96]. Table 5 shows the performance of FastFDs, TANE and Dep-Miner on 9 relation instances extracted from the ML Repository.[14] Table 6 indicates the relative performance of Dep-Miner and FastFDs on these same 9 relations with the effects of the difference set calculation time factored out.

Qualitative Analysis of Tables 5 and 6. First, compare the running times of FastFDs and Dep-Miner. FastFDs meets or exceeds Dep-Miner's performance in all cases. As $|R|$ becomes larger, FastFDs becomes increasingly faster than Dep-Miner.

Now compare the running times of FastFDs and TANE. Here, the comparison is not as straightforward. For large $|r|$, note the lengthy times of FastFDs. This is due to the $|r|^2$ difference set calculation. On the other hand, as the number of attributes ($|R|$) grows, TANE experiences serious difficulties due to excessive memory consumption. For the Horse-Colic database ($|R| = 28$), FastFDs finds the minimal FDs quickly; TANE runs out of memory. The best method for finding minimal FDs in real-life situations might involve a tradeoff between TANE (for small $|R|$ and large $|r|$) and FastFDs (for small $|r|$ or larger $|R|$).

[14] For all three programs, the tests were run on preprocessed, purely integer-valued versions of these ML relations, as in TANE. In addition, the three programs were tested on the following instances and each returned in less than 0.1 second: Liver, Abalone, Pima, Tic-Tac-Toe, Wisconsin Breast Cancer, Echocardiagram and Housing.

ML Instance				Time (seconds)			
Name	$\lvert R\rvert$	$\lvert r\rvert$	$\lvert\mathcal{F}\rvert$	FastFDs	Tane	Tane/mem	Dep-Miner
Chess	**7**	28,056	1	160	1	$0^{\dagger}$	160
Flare	**13**	1389	0	5	11	1	5
Adult	**15**	48,842	66	7269	1722	•	7269
Credit	**16**	690	1099	2	1	$0^{\dagger}$	6
Letter Recognition	**17**	20,000	61	1577	2487	•	1645
Lymphography	**19**	148	2730	4	41	9	91
Hepatitis	**20**	155	8250	3	13	5	212
Automobile	**26**	205	4176	1	179	81	∘
Horse Colic	**28**	300	128,726	123	•	•	•

† indicates program took less than 0.01 seconds to complete.
∘ indicates execution time exceeded 3 hours and was aborted.
• indicates program ran out of memory.

Table 5. Results on ML Repository Relation Instances.

ML Instance				Time (seconds)	
Name	$\lvert R\rvert$	$\lvert r\rvert$	$\lvert\mathcal{F}\rvert$	FastFDs	Dep-Miner
Chess	**7**	28,056	1	$0^{\dagger}$	$0^{\dagger}$
Flare	**13**	1389	0	0.5	0.5
Adult	**15**	48,842	66	4.4	4.8
Credit	**16**	690	1099	1.2	5.2
Letter Recognition	**17**	20,000	61	858.1	926.1
Lymphography	**19**	148	2730	4.3	90.8
Hepatitis	**20**	155	8250	3.0	212.0
Automobile	**26**	205	4176	1.0	∘
Horse Colic	**28**	300	128,726	122.6	•

∘ indicates execution time exceeded 3 hours and was aborted.
• indicates program ran out of main memory.
† indicates time was less than $0.01s$.

Table 6. Results on relation instances from table 5; the time for computing difference sets is omitted.

4 Conclusions and Further Directions

In this paper, we presented a novel search method, FastFDs, for computing minimal FDs from difference sets using heuristic-driven, depth-first search.

Our experimental results (§3) indicate that FastFDs is competitive for each of the following classes of benchmark relation instances: (1) random integer-valued instances of varying correlation factors, (2) random Bernoulli instances, and (3) real-life ML Repository relation instances. In fact, our experiments indicate that for wide relations (large R), FastFDs is significantly better for all classes, due to the inherent space efficiency of the depth-first search method. For narrow relations, a better choice may be the TANE algorithm for class (3) relation instances.

It is interesting that the heuristic-based, depth-first search methods common in solutions to Artificial Intelligence (AI) problems are usually eschewed by the data mining community, because they have a tendency *not* to scale up to the massive amounts of data the data mining programs must be run on. However, our experiments show that for the case of computing minimal covers of $\mathfrak{D}_r^A$, in fact the AI search strategy fares significantly better than the canonical levelwise approach. In fact, for the case of Bernoulli databases, when the FDs computed are of average length $\geq |R|/2$, the heuristic-driven approach is astoundingly more efficient (§3.2).

The space efficiency of FastFDs makes it a natural algorithm for parallelization. One obvious multi-threaded version of FastFDs would involve delegating the search for FDs with RHS A to distinct processors for distinct attributes A.

The performance of FastFDs depends crucially on the simple, greedy heuristic we have chosen for computing minimal hypergraph covers. Our heuristic works for finding minimal covers for *generic* hypergraphs; however, not every hypergraph can be realized from a relation instance. Additional constraints are implied by the relation instance which are not reflected in the current heuristic. Further study of these constraints with the aim of designing better heuristics seems an interesting path to pursue.

Another useful direction for further study (pointed out in [MR 87]) is to consider the incremental dependency inference problem: given, r, the canonical cover of the set of dependencies $\mathcal{F}_r$, and a tuple t, find the canonical cover of $r \cup \{t\}$ (or $r - \{t\}$). To the best of our knowledge, this problem has not yet been addressed, and seems reasonably challenging.

Finally, it seems reasonable that most applications of the dependency inference problem do not require all dependencies but only the "interesting" ones. For example, dependencies with small left-hand sides are likely to be more useful that ones with large left-hand sides. Characterizing the interestingness of dependencies and tailoring algorithms to finding these dependencies is a good direction for future work.

Acknowledgments

The authors thank the following individuals for contributions to this paper: Felix Wyss, Dennis Groth, Mehmet Dalkilic, Juha Kärkkäinen, Stéphane Lopes, and the anonymous reviewers of DaWaK 2001. The databases in §3.3 were obtained from the ML Repository online [MM 96]. The lymphography database was obtained from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data.

References

[AHV 95] Abiteboul, Serge; Hull, Richard and Vianu, Victor. *Foundations of Databases*. Addison-Wesley, Reading Massachusetts, 1995.

[AMS+ 94] Agrawal, Rakesh; Mannila, Heikki; Srikant, Ramakrishnan; Toivonen, Hannu and Verkamo, A.I. "Fast Discovery of Association Rules." *Advances in KDD*, AAAI Press, Menlo Park, CA, pg. 307-328, 1996.

[DKM+ 95] Demetrovics, J; Katona, G; Miklos, D; Seleznjev, O. and Thalheim, B. "The Average Length of Keys and Functional Dependencies in (Random) Databases." *Lecture Notes in Computer Science*, vol. 893, 1995.

[EG 95] Eiter, Thomas and Gottlob, Goerg. "Identifying the Minimal Traversals of a Hypergraph and Related Problems." *SIAM Journal of Computing*, vol. 24, no. 6, pg. 1278-1304, 1995.

[FS 99] Flach, Peter and Savnik, Iztok. "Database Dependency Discovery: a Machine Learning Approach." *AI Comm.* vol. 12, no. 3, pg 139-160.

[GKMT 97] Gunopulos, Dimitrios; Khardon, Roni; Mannila, Heikki; and Toivonen, Hannu. "Data Mining, Hypergraph Traversals, and Machine Learning (extended abstract)", PODS, 1997, pg 209-216.

[HK 2001] Han, Jiawei and Kamber, Micheline. "Data Mining: Concepts and Techniques", Morgan Kaufmann, 2001.

[HKP+ 99] Huhtala, Ykä; Kärkkäinen, Juha; Porkka, Pasi and Toivonen, Hannu. "TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies." *The Computer Journal*, vol. 42, no. 2, 1999.

[KMR+ 92] Kantola, Martti; Mannila, Heikki; Räihä, Kari-Jouko and Siirtola, Harri. "Discovering Functional and Inclusion Dependencies in Relational Databases." *Journal of Intelligent Systems*, vol. 7, pg. 591-607, 1992.

[LPL 00a] Lopes, Stephane; Petit, Jean-Marc and Lakhal, Lotfi. "Efficient Discovery of Functional Dependencies and Armstrong Relations." *Proceedings of ECDT 2000.* Lecture Notes in Computer Science, vol 1777.

[LPL 00b] Lopes, Stephane; Petit, Jean-Marc and Lakhal, Lotfi. "Efficient Discovery of Functional Dependencies and Armstrong Relations" (full version). http://libd2.univ-bpclermont.fr/publications, Technical report, LIMOS, 1999.

[MR 87] Mannila, Heikki and Räihä, Kari-Jouko. "Dependency Inference (Extended Abstract)", *Proceedings of VLDB 1987*, Brighton, pg. 155-158, 1987.

[MR 94] Mannila, Heikki and Räihä, Kari-Jouko. "Algorithms for Inferring Functional Dependencies from Relations", *Data & Knowledge Engineering*, 12, pg. 83-99, 1994.

[Man 97] Mannila, Heikki. "Methods and Problems in Data Mining." *Proceedings of the ICDT*, pg. 83-99, January 1997.

[MT 97] Mannila, Heikki and Toivonen, Hannu. "Levelwise Search and Borders of Theories of Knowledge", *Data Mining and Knowledge Discovery*, vol. 1, pg. 241-258, 1997.

[MM 96] Merz, C.J. and Murphy, P.M. UCI machine learning databases. `http://www.ics.uci.edu/~mlearn/MLRepository.html`. Irvine, CA: University of California, Dept. of Information and Comp. Sci. 1996.

[RG 00] Ramakrishnan, Raghu and Gehrke, Johannes. *Database Management Systems.* McGraw-Hill, 2000.

[RN 95] Russell, Stuart J. and Norvig, Peter. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[Tane] The TANE and TANE/MEM source code is available on the web at `http://www.cs.helsinki.fi/research/fdk/datamining/tane`.

[Fdep] The **fdep** source code is available on the web at `http://www.cs.bris.ac.uk/~flach/fdep/`.

A The Program fdep

There is another candidate program for finding FDs in a relation instance – fdep [FS 99]. This algorithm was developed by Flach and Savnik, and has been experimentally compared to TANE [HKP+ 99], [FS 99]. The approach fdep uses for finding FDs is based on canonical AI methods for learning general logical descriptions within hypotheses spaces [RN 95,FS 99]. Experiments show that, in fact, a pure bottom up search method within this framework is superior, when contrasted with a top-down search method, or bi-directional search [FS 99]. We ran experiments showing fdep's performance on the test canon of relation instances. The version of fdep we tested is the version available on the web [Fdep]. Tables 7 and 8 (below) show the performance of fdep on random integer-valued and Bernoulli relations.

The fdep search framework does not fit into the levelwise or breadth-first heuristic frameworks discussed above. For certain small relations, the fdep search method outperforms both TANE and FastFDs. However, an intrinsic drawback of the fdep approach is that its complexity can be exponential in the *size of the input relation instance.* This is a severe limitation from the database point of view, albeit the fdep algorithm is quite successful from an AI theory point of view.

Qualitative Analysis of Tables 7 and 8. The figures in table 7 illustrate fdep's poor performance as the number of tuples in the input relation increases beyond 1000. On relations of the size shown in table 7, the algorithms FastFDs, TANE, and Dep-Miner all complete in less than 1 second (table 2 contains results for Dep-Miner, TANE, and FastFDs for relations of size $\geq 50,000$). Table 8 illustrates fdep's exponential space requirements for Bernoulli relations. The performance of FastFDs, TANE, and Dep-Miner are shown for comparison.

The performance of fdep on the ML relations is exhibited in table 9. The performance of TANE and FastFDs is shown for comparison.

Instance	Time (seconds)		
$\|r\|$	CF = 0.0	CF = 0.5	CF = 0.9
10	$0^{\dagger}$	$0^{\dagger}$	1
100	$0^{\dagger}$	1	3
500	2	3	4
1,000	10	11	11
5,000	258	257	258
10,000	1035	1225	1779

† indicates execution time less than $0.5s$.

Table 7. Results for fdep on Random Integer-Valued Relation Instances; $|R| = 20$.

Instance		Time (seconds)			
$\|R\|$	$\|r\| = 2^{\|R\|/4}$	FastFDs	Tane	fdep	Dep-Miner
20	32	2	5	26	195
21	38	5	15	74	601
22	45	11	36	•	1893
23	54	28	84	•	5169
24	64	69	392	•	14547

• indicates program ran out of memory.

Table 8. Results for fdep on Bernoulli Relation Instances.

Qualitative Analysis of Tables 9 and 10. The results in table 9 show that fdep takes more time than the other programs for the ML repository relations. The only exceptions are, notably, the Lymphography and Horse Colic relations. These relations exhibit a fairly large number of attributes (19 and 28, respectively), yet a small number of tuples (148 and 300, respectively). Table 10 (below) shows the result of running fdep on larger versions of the Horse Colic relation (i.e. containing duplicate tuples). Note that the poor performance of fdep as the relation increases in size becomes apparent. Fdep quickly becomes infeasible as the size of the relation increases beyond 10,000 tuples. These experiments bear out the results of [HKP+ 99], even for the improved solely bottom-up version of fdep.

ML Instance				Time (seconds)			
Name	$\lvert R \rvert$	$\lvert r \rvert$	$\lvert \mathcal{F} \rvert$	FastFDs	Tane	Tane/mem	fdep
Liver	**7**	345	25	$0^\dagger$	$0^\dagger$	$0^\dagger$	$0^\dagger$
Chess	**7**	28,056	1	160	1	$0^\dagger$	2129
Pima	**9**	768	153	$0^\dagger$	$0^\dagger$	$0^\dagger$	2
Abalone	**9**	4177	137	$0^\dagger$	$0^\dagger$	1	52
Tic-Tac-Toe	**10**	958	18	$0^\dagger$	$0^\dagger$	1	4
Wisc. breast cancer	**11**	699	46	$0^\dagger$	$0^\dagger$	$0^\dagger$	2
Echocardiagram	**13**	130	448	$0^\dagger$	$0^\dagger$	$0^\dagger$	$0^\dagger$
Flare	**13**	1389	0	5	11	1	9
Housing	**14**	506	478	$0^\dagger$	$0^\dagger$	$0^\dagger$	2
Credit	**16**	690	1099	2	1	$0^\dagger$	4
Letter Recognition	**17**	20,000	61	1577	2487	•	4312
Lymphography	**19**	148	2730	4	41	9	1
Hepatitis	**20**	155	8250	3	13	5	3
Automobile	**26**	205	4176	1	179	81	2
Horse Colic	**28**	300	128,726	123	•	•	90

† indicates program took less than 0.5 seconds to complete.
• indicates program ran out of memory.

Table 9. Results on ML Repository Relation Instances.

Instance		Time (seconds)
Name	$\lvert r \rvert$	fdep
Lymphography	148	1
Lymphography × 10	1,480	22
Lymphography × 100	14,800	4307
Horse Colic	300	90
Horse Colic × 10	3,000	228
Horse Colic × 100	30,000	42035

Table 10. Results on ML Relation Instances – Repeated Tuples.