Efficient Discovery of Functional and Approximate Dependencies Using Partitions

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen*
University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN–00014 University of Helsinki, Finland
Email: {yka.huhtala, juha.karkkainen, pasi.porkka, hannu.toivonen}@cs.helsinki.fi

Abstract

Discovery of functional dependencies from relations has been identified as an important database analysis technique. In this paper, we present a new approach for finding functional dependencies from large databases, based on partitioning the set of rows with respect to their attribute values. The use of partitions makes the discovery of approximate functional dependencies easy and efficient, and the erroneous or exceptional rows can be identified easily. Experiments show that the new algorithm is efficient in practice. For benchmark databases the running times are improved by several orders of magnitude over previously published results. The algorithm is also applicable to much larger datasets than the previous methods.

1 Functional and approximate dependencies

Functional dependencies are relationships between attributes of a relation: a functional dependency states that the value of an attribute is uniquely determined by the values of some other attributes. The discovery of functional dependencies from relations has received considerable interest (e.g., [2, 10, 17, 19, 11, 1, 6, 3]). Automated database analysis is, of course, interesting for knowledge discovery and data mining (KDD) purposes, and functional dependencies have applications in the areas of database management, reverse engineering [14, 20], and query optimization [21].

Formally, a *functional dependency* over a relation schema R is an expression $X \rightarrow A$, where $X \subseteq R$ and $A \in R$. The dependency *holds* or is *valid* in a given relation r over R if for all pairs of rows $t, u \in r$ we have: if $t[B] = u[B]$ for all $B \in X$, then $t[A] = u[A]$ (we also say that t and u *agree* on X and A). A functional dependency $X \rightarrow A$ is *minimal* (in r) if A is not functionally dependent on any proper subset of X, i.e., if $Y \rightarrow A$ does not hold in r for any $Y \subset X$. The dependency $X \rightarrow A$ is *trivial* if $A \in X$. The central task we consider is the following: given a relation r, find all minimal non-trivial dependencies that hold in r.

An approximate dependency [5] is a functional dependency that almost holds. Such dependencies arise in many databases when there is a natural dependency between attributes, but some rows contain errors or represent exceptions to the rule. The discovery of unexpected but meaningful approximate dependencies seems to be an interesting and realistic goal in many data mining applications.

There are many possible ways of defining the approximateness of a dependency $X \rightarrow A$. The definition we use is based on the minimum number of rows that need to be removed from the relation r for $X \rightarrow A$ to hold in r: the *error* $g_3(X \rightarrow A) = 1 - (\max\{|s| \mid s \subseteq r \text{ and } X \rightarrow A \text{ holds in } s\})/|r|$ [5]. The measure g_3 has a natural interpretation as the fraction of rows with exceptions or errors affecting the dependency. Given an error threshold ε, $0 \leq \varepsilon \leq 1$, we say that $X \rightarrow A$ is an *approximate dependency* if and only if $g_3(X \rightarrow A)$ is at most ε. In this paper, we also consider the approximate dependency inference task: given a relation r and a threshold ε, find all minimal non-trivial approximate dependencies.

We describe a new approach to the discovery of both functional and approximate dependencies. The major innovation is a novel way of determining whether a dependency holds or not. The idea is to maintain information about which rows agree on a set of attributes. Formally, the approach can be described using equivalence classes and partitions. A major advantage of the use of partitions is that it allows efficient discovery of approximate dependencies.

The algorithm is based on the levelwise algorithm that has been used in many data mining applications [12]. It starts from dependencies with a small left-hand side, i.e., from the ones that are not very likely to hold. The algorithm then works towards larger and larger dependencies, until the minimal dependencies that hold are found.

The worst case time complexity of the algorithm with respect to the number of attributes is exponential, but this is inevitable since the number of minimal dependencies can

*Also at Rolf Nevanlinna Institute, University of Helsinki.

be exponential in the number of attributes [10, 9]. However, if the number of rows increases but the set of dependencies stays the same, the time increases only linearly in the number of rows. To our knowledge, only one previous algorithm can claim this [18]. Other algorithms based on sorting could perhaps be implemented in linear time, e.g., by using hashing, but we are not aware of such implementations. The linearity makes the algorithm especially suitable for relations with large number of rows.

Experimental results show that the algorithm is effective in practice, and that it makes the discovery of functional and approximate dependencies feasible for relations with even hundreds of thousands of rows. Dependency discovery tasks that have been reported to take minutes or even hours are solved with the new algorithm in seconds or fractions of a second on a PC.

Related work Several algorithms for the discovery of functional dependencies have been presented [7, 2, 9, 18, 17, 11, 1]. We review these algorithms and compare them with our method in Section 6. The complexity of discovering functional dependencies has been studied in [8, 10, 9].

Approximate functional dependencies have been considered in [5, 15, 6, 3]. Kivinen and Mannila [5] define several measures for the error of a dependency, and derive bounds for discovering dependencies with errors. The measure g_3 is one of their measures.

The use of partitions to describe and define functional and approximate dependencies has been suggested in [3] parallel to our work. There the emphasis is on a conceptual viewpoint, and no algorithms are given.

Extended version An extended version of this article, with proofs and additional details, is available as [4]. An implementation of the algorithm can be obtained via the WWW page at http://www.cs.helsinki.fi/research/fdk/datamining/tane/.

2 Partitions and dependencies

Informally, a dependency $X \rightarrow A$ holds if all rows that agree on X also agree on A. Our approach to the discovery of dependencies is based on considering sets of rows that agree on some set of attributes. We describe this idea more formally by applying equivalence classes and partitions on relations.

Partitions Two rows t and u are *equivalent* with respect to a given set X of attributes if $t[A] = u[A]$ for all A in X. Any attribute set X partitions the rows of the relation into equivalence classes. We denote the *equivalence class* of a row $t \in r$ with respect to a given set $X \subseteq R$ by $[t]_X$, i.e., $[t]_X = \{u \in r \mid t[A] = u[A] \text{ for all } A \in X\}$. The set $\pi_X = \{[t]_X \mid t \in r\}$ of equivalence classes is a *partition* of r under X. That is, π_X is a collection of disjoint sets (equivalence classes) of rows, such that each set has a unique value for the attribute set X, and the union of the sets equals the relation r. The *rank* $|\pi|$ of a partition π is the number of equivalence classes in π.

Row Id	A	B	C	D
1	1	a	$	Flower
2	1	A	£	Tulip
3	2	A	$	Daffodil
4	2	A	$	Flower
5	2	b	£	Lily
6	3	b	$	Orchid
7	3	C	£	Flower
8	3	C	#	Rose

Figure 1: An example relation.

Example 1 Consider the relation in Figure 1. Attribute A has value 1 only on rows t_1 and t_2, so they form an equivalence class $[t_1]_{\{A\}} = [t_2]_{\{A\}} = \{1, 2\}$. The whole partition with respect to A is $\pi_{\{A\}} = \{\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}\}$. The partition with respect to $\{B, C\}$ is $\pi_{\{B,C\}} = \{\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6\}, \{7\}, \{8\}\}$. □

Partition refinement The concept of partition refinement gives almost directly functional dependencies. A partition π is a *refinement* of another partition π' if every equivalence class in π is a subset of some equivalence class of π'. We have the following lemma.

Lemma 1 *A functional dependency $X \rightarrow A$ holds if and only if π_X refines $\pi_{\{A\}}$.*

Example 2 Continuing Example 1, to find out whether the dependency $\{B, C\} \rightarrow A$ holds, we can compare the partitions $\pi_{\{B,C\}}$ and $\pi_{\{A\}}$ and check whether $\pi_{\{B,C\}}$ refines $\pi_{\{A\}}$. In the relation of Figure 1, the dependency holds since each equivalence class in $\pi_{\{B,C\}}$ is totally contained by some equivalence class in $\pi_{\{A\}}$.

The dependency $\{A\} \rightarrow B$ does not hold in the figure: the equivalence class $[t_3]_{\{A\}} = \{3, 4, 5\}$, for instance, is not contained in any equivalence class in $\pi_{\{B\}} = \{\{1\}, \{2, 3, 4\}, \{5, 6\}, \{7, 8\}\}$. □

There is an even simpler test for whether $X \rightarrow A$ holds or not. If π_X refines $\pi_{\{A\}}$, then adding A to X does not break any equivalence classes of π_X; thus $\pi_{X \cup \{A\}}$ equals π_X. On the other hand, since $\pi_{X \cup \{A\}}$ always refines π_X, $\pi_{X \cup \{A\}}$ cannot have the same number of equivalence classes as π_X unless $\pi_{X \cup \{A\}}$ and π_X are equal. We have shown the following lemma.

Lemma 2 *A functional dependency $X \rightarrow A$ holds if and only if $|\pi_X| = |\pi_{X \cup \{A\}}|$.*

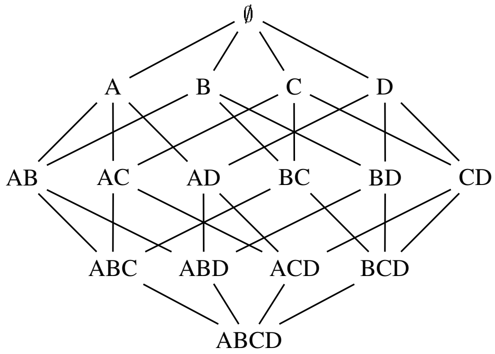

Figure 2: The set containment lattice for $\{A, B, C, D\}$: the search space of all possible left-hand sides.

Approximate dependencies Recall that the error $g_3(X \to A)$ of a dependency $X \to A$ is the minimum fraction of rows that must be removed from the relation for $X \to A$ to hold. The error $g_3(X \to A)$ can be computed from the partitions π_X and $\pi_{X \cup \{A\}}$ in the following way. Any equivalence class c of π_X is the union of one or more equivalence classes $c'_1, c'_2, \ldots$ of $\pi_{X \cup \{A\}}$, and the rows in all but one of the c'_is must be removed for $X \to A$ to hold. The minimum number of rows to remove is thus the size of c minus the size of the largest of the c'_is. Summing that over all equivalence classes c of π_X gives the total number of lines to remove. Thus, we have

$$g_3(X \to A) = \\ 1 - \sum_{c \in \pi_X} \max\{|c'| \mid c' \in \pi_{X \cup \{A\}} \text{ and } c' \subseteq c\}/|r|.$$

An algorithm to compute $g_3(X \to A)$ given the partitions π_X and $\pi_{X \cup \{A\}}$ is described in [4].

3 Search

To find all minimal non-trivial dependencies, we search through the space of non-trivial dependencies and test the validity and minimality of each dependency. The validity test uses partitions as described in the previous section. In addition, we need to do the following tasks efficiently: (1) compute partitions, and (2) test minimality.

The collection of all possible left-hand sides of dependencies is the collection of all attribute sets. They constitute a set containment lattice such as in Figure 2. In many data mining applications, such lattices have been searched successfully using a levelwise algorithm [12]. The levelwise algorithm starts the search from the singleton sets, and works its way through the lattice level by level until the minimal dependencies that hold are found. During this levelwise search, false dependencies are eliminated as early as possible, in order to reduce the search space. This pruning is described in Section 4.

We consider all possible right-hand sides with a single breadth-first or levelwise pass through the lattice. There is a one-to-one correspondence between the edges of the lattice and the non-trivial dependencies: we view an edge between sets X and $X \cup \{A\}$ as representing the non-trivial dependency $X \to A$.

The efficiency of the levelwise algorithm is based on reducing the computation on each level by using results from previous levels. In the rest of this section, we describe how to use previous levels to solve efficiently the tasks of computing partitions and testing minimality.

Computing partitions We do not need to compute the partitions from scratch for every set of attributes we consider. Instead, when we work our way through the lattice, we can compute a partition as a product of two earlier partitions: the *product* of two partitions π' and π'', denoted $\pi' \cdot \pi''$, is the least refined partition π that refines both π' and π''. We have the following result.

Lemma 3 *For all $X, Y \subseteq R$, $\pi_X \cdot \pi_Y = \pi_{X \cup Y}$.*

We compute the partitions $\pi_{\{A\}}$, for each $A \in R$, directly from the database. Partitions π_X, for $|X| \geq 2$, are computed as a product of partitions with respect to two subsets of X. Any two different subsets of size $|X| - 1$ will do, which is convenient for the levelwise algorithm since we only need partitions from the previous level.

Testing minimality When the algorithm is processing a set X, it will test dependencies of the form $X \setminus \{A\} \to A$, where $A \in X$. This allows validity testing based on Lemma 2, since both π_X and $\pi_{X \setminus \{A\}}$ have already been computed. To test the minimality of $X \setminus \{A\} \to A$, we need to know whether $Y \setminus \{A\} \to A$ holds for some proper subset Y of X. This information is stored in the set $\mathcal{C}(X \setminus \{A\})$ of right-hand side candidates of $X \setminus \{A\}$ for all A.

More exactly, if $A \in \mathcal{C}(X)$ for a given set X, then A has not been found to depend on any proper subset of X, i.e., either $A \in X$ and $X \setminus \{A\} \to A$ does not hold, or $A \in R \setminus X$. Formally, the collection $\mathcal{C}(X)$ of *rhs candidates* of a set $X \subseteq R$ is

$$\mathcal{C}(X) = \{A \in X \mid X \setminus \{A\} \to A \text{ does not hold}\} \cup R \setminus X.$$

To find minimal dependencies, it suffices to test dependencies $X \setminus \{A\} \to A$, where $A \in X$ and $A \in \mathcal{C}(X \setminus \{B\})$ for all $B \in X$.

Example 3 *Assume that the algorithm is considering the set $X = \{A, B, C\}$, and that $\{C\} \to A$ is a valid dependency. Since $\{C\} \to A$ holds, we have that $A \notin \mathcal{C}(\{A, C\}) = \mathcal{C}(X \setminus \{B\})$, and $\{B, C\} \to A$ is thus not minimal.* □

4 Pruning

Pruning the search space means reducing the number of dependencies we have to consider. For example, if we find that $X \rightarrow A$ holds, then $Y \rightarrow A$ is not minimal for any proper superset Y of X. Thus, we can automatically discard $Y \rightarrow A$ from consideration.

The levelwise algorithm has a powerful mechanism for pruning the search space. When the algorithm is processing the level of the lattice that contains a set X, we can with one stroke cut off all supersets of X from the lattice simply by deleting X [12]. If some property of X tells us that no superset of X is interesting to us, we just delete X. In our case, empty rhs candidate set is such a property. That is, if $\mathcal{C}(X) = \emptyset$, then $\mathcal{C}(Y) = \emptyset$ for all supersets Y of X, and no dependency of the form $Y \setminus \{A\} \rightarrow A$ can be minimal.

We can further improve the pruning based on the properties of dependencies stated in the following lemma.

Lemma 4 *Let $B \in X$ and let $X \setminus \{B\} \rightarrow B$ be a valid dependency. (1) If $X \rightarrow A$ holds, then $X \setminus \{B\} \rightarrow A$ holds. (2) If X is a superkey, then $X \setminus \{B\}$ is a superkey.*

Rhs candidate pruning The first part of the lemma allows us to remove additional attributes from the rhs candidate sets and, consequently, make pruning by empty rhs candidate set more effective. The resulting collection $\mathcal{C}^+(X)$ of *rhs$^+$ candidates* of a set $X \subseteq R$ is

$$\mathcal{C}^+(X) = \{A \in R \mid \text{for all } B \in X, \quad X \setminus \{A, B\} \rightarrow B \text{ does not hold}\}.$$

The following lemma shows that we can use the rhs$^+$ candidates to test the minimality of a dependency just as we would use the rhs candidates.

Lemma 5 *Let $A \in X$ and let $X \setminus \{A\} \rightarrow A$ be a valid dependency. The dependency $X \setminus \{A\} \rightarrow A$ is minimal if and only if, for all $B \in X$, we have $A \in \mathcal{C}^+(X \setminus \{B\})$.*

Key pruning When a key is found during the search of dependencies, additional pruning methods can be applied. Recall that an attribute set X is a *superkey* if no two rows agree on X, and a *key* if it is a superkey and no proper subset of it is a superkey. Normally, a dependency $X \rightarrow A$ is tested when $X \cup \{A\}$ is processed because we need $\pi_{X \cup \{A\}}$ for validity testing. However, if X is a superkey then $X \rightarrow A$ is always valid and we do not need $X \cup \{A\}$.

Now, consider a superkey X that is not a key. Obviously, a dependency $X \rightarrow A$ is not minimal for any $A \notin X$. Furthermore, if $A \in X$ and $X \setminus \{A\} \rightarrow A$ holds, then, by the second part of Lemma 4, $X \setminus \{A\}$ is a superkey and we do not need π_X for testing the validity of $X \setminus \{A\} \rightarrow A$. In other words, we have no use for X or π_X in finding minimal dependencies. Hence, we can delete all keys and cut off all of their supersets, i.e., the superkeys that are not keys.

5 Algorithms

To find all valid minimal non-trivial dependencies, we search the set containment lattice in a levelwise manner. A *level* L_ℓ is the collection of attribute sets of size ℓ such that the sets in L_ℓ can potentially be used to construct dependencies based on the considerations of the previous sections. We start with $L_1 = \{\{A\} \mid A \in R\}$, and compute L_2 from L_1, L_3 from L_2, and so on, according to the information we obtain during the algorithm.

Algorithm TANE: levelwise search of dependencies.

```
L_0 := {∅}
C+(∅) := R
L_1 := {{A} | A ∈ R}
ℓ := 1
while L_ℓ ≠ ∅
   COMPUTE-DEPENDENCIES(L_ℓ)
   PRUNE(L_ℓ)
   L_{ℓ+1} := GENERATE-NEXT-LEVEL(L_ℓ)
   ℓ := ℓ + 1
```

Generating levels The procedure GENERATE-NEXT-LEVEL computes the level $L_{\ell+1}$ from L_ℓ. The level $L_{\ell+1}$ will contain only those attribute sets of size $\ell + 1$ which have all their subsets of size ℓ in L_ℓ. The pruning methods guarantee that no dependencies are lost. The specification of GENERATE-NEXT-LEVEL is

$$L_{\ell+1} = \{X \mid |X| = \ell + 1 \text{ and for all } Y \text{ with } Y \subset X \text{ and } |Y| = \ell \text{ we have } Y \in L_\ell\}.$$

GENERATE-NEXT-LEVEL also computes the partition for each new attribute set generated. Algorithms are given in [4].

Procedure COMPUTE-DEPENDENCIES(L_ℓ)

```
1  for each X ∈ L_ℓ do
2     C+(X) := ⋂_{A∈X} C+(X \ {A})
3  for each X ∈ L_ℓ do
4     for each A ∈ X ∩ C+(X) do
5        if X \ {A} → A is valid then
6           output X \ {A} → A
7           remove A from C+(X)
8           remove all B in R \ X from C+(X)
```

By Lemma 5, steps 2, 4 and 5 guarantee that the procedure outputs exactly the minimal dependencies of the form $X \setminus \{A\} \rightarrow A$, where $X \in L_\ell$ and $A \in X$. The validity testing on line 5 is based on Lemma 2.

COMPUTE-DEPENDENCIES(L_ℓ) also computes the sets $\mathcal{C}^+(X)$ for all $X \in L_\ell$. The following lemma shows that this is done correctly.

Lemma 6 *For all $Y \in L_{\ell-1}$, let $\mathcal{C}^+(Y)$ be correctly computed. After executing the procedure* COMPUTE-DEPENDENCIES(L_ℓ), *$\mathcal{C}^+(X)$ is correctly computed for all $X \in L_\ell$.*

Line 8 implements the difference between $\mathcal{C}^+(X)$ and $\mathcal{C}(X)$. If that line was removed, the algorithm would work correctly, but pruning might be less effective.

Procedure PRUNE(L_ℓ)

```
for each X ∈ L_ℓ do
  if C+(X) = ∅ do
    delete X from L_ℓ
  if X is a (super)key do
    for each A ∈ C+(X) \ X do
      if A ∈ ⋂_{B∈X} C+(X ∪ {A} \ {B}) then
        output X → A
    delete X from L_ℓ
```

Procedure PRUNE implements the two pruning rules described in Section 4. By the first rule, X is deleted if $\mathcal{C}^+(X) = \emptyset$. By the second rule, X is deleted if X is a key. In the latter case, the algorithm may also output some dependencies. In [4], we show that the pruning does not cause the algorithm to miss any dependencies.

Approximate dependencies Algorithm TANE can be modified so that it computes all minimal approximate dependencies $X \to A$ with $g_3(X \to A) \leq \varepsilon$, for a given threshold value ε. The key modification is to change the validity test on line 5 of procedure COMPUTE-DEPENDENCIES to

5' **if** $g_3(X \setminus \{A\} \to A) \leq \varepsilon$ **then**

In addition, line 8 of COMPUTE-DEPENDENCIES has to be removed or changed to

8' **if** $X \setminus \{A\} \to A$ holds exactly **then**
9' remove all B in $R \setminus X$ from $\mathcal{C}^+(X)$

Optimizations In [4], we give two methods that reduce the time and space requirement of the partition computations. The first one replaces partitions with a more compact representation, and the second one is a method to quickly bound the g_3 error.

6 Analysis

Worst case analysis The time and space complexities of the TANE algorithm depend on the number of sets in the levels L_ℓ, called the size of a level. Let s_{max} be the size of the largest level, and s the sum of the sizes of the levels. In the worst case, $s = O(2^{|R|})$ and $s_{max} = O(2^{|R|}/\sqrt{|R|})$. Another factor is the number of keys, denoted by k. In the worst case, $k = O(s_{max}) = O(2^{|R|}/\sqrt{|R|})$.

In summary, the algorithm has time complexity $O(s(|r| + |R|^2) + k|R|^3)$ and space complexity $O(s_{max}(|r| + |R|))$. The following theorem gives upper bounds for the time and space complexities in terms of the size of the input.

Theorem 1 *Algorithm* TANE *has time complexity $O\big((|r| + |R|^{2.5})2^{|R|}\big)$ and space complexity $O\big((|r| + |R|)2^{|R|}/\sqrt{|R|}\big)$.*

Approximate validity testing needs $O(|r|)$ time in contrast to the $O(1)$ time of exact validity testing. Thus, the time complexity of finding approximate dependencies with TANE is $O(v|r| + s|R|^2 + k|R|^3)$, where v is the number of validity tests done. In the worst case, $v = s|R|/2 = O(|R|2^{|R|})$, and thus the time in terms of the size of the input is $O\big((|r||R| + |R|^{2.5})2^{|R|}\big)$.

Practical analysis Due to the structure of the dependency set and pruning, s and s_{max} can be significantly smaller than the worst case analysis shows. The number k of keys is almost always much smaller than s_{max}.

We have implemented the attribute sets as bit vectors of $O(1)$ words and the random access with hashing. This means, in practice, that set operations and random access take constant time. To reduce the main memory requirement of the algorithm, the partitions can be stored on disk.

The properties of the algorithm after the above modifications are summarized below.

- CPU time: $O(s(|r| + |R|) + k|R|^2)$
- disk accesses: $O(s)$ accesses of size $O(|r|)$
- main memory requirement: $O(|r||R| + s_{max})$
- disk space requirement: $O(s_{max}|r|)$

In the approximate dependency version of TANE, validity testing takes more time and partitions are needed more often. Because of the latter, we only consider the case where partitions are not stored on disk. The approximate dependency algorithm works in $O(v|r| + s|R| + k|R|^2)$ time and $O(s_{max}|r|)$ space. However, because there are more approximately valid dependencies, pruning can be much more effective reducing s, s_{max}, and v.

Comparison to other algorithms One of the main advantages of the new algorithm is the linear dependency on the number of rows in the relation (for a fixed set of dependencies). To our knowledge, the only previously published practical algorithm achieving this is by Schlimmer [18, 19], who uses decision trees for validity tests. The decision tree approach is roughly equivalent to computing each partition from partitions with respect to singletons. It is slower by a factor $O(|R|)$ than using partitions the way we do. All other

Table 1: Performance of the algorithms on real life databases.

Database				Time (s)		
Name	$\|r\|$	$\|R\|$	N	TANE	TANE/MEM	FDEP
Lymphography	148	19	2730	68.2	24.0	88.0
Hepatitis	155	20	8250	29.6	14.1	663
Wisconsin breast cancer	699	11	46	0.76	0.25	15.0
Wisconsin breast cancer $\times$ 64	44736	11	46	80.5	23.0	17521
Wisconsin breast cancer $\times$ 128	89472	11	46	173	247	*
Wisconsin breast cancer $\times$ 512	357888	11	46	884	*	*
Adult	48842	15	85	1451	*	*
Chess	28056	7	1	3.63	2.03	6685

algorithms that we know of have $\Omega(|r|^2)$ or $\Omega(|r|\log|r|)$ dependency on the number of rows. Many of these could actually be implemented to run in linear time as well, if sorting is not realized by using comparisons but by using other mechanisms instead, e.g., hashing. However, we are not aware of such implementations.

Schlimmer also used the levelwise search strategy, as did Bell and Brockhausen [1]. Both use less effective pruning criteria than we do, i.e., their algorithms may end up computing larger part of the lattice.

There are also algorithms that search the lattice in a more depth-first like manner [11, 9]. Such a search allows criteria for the pruning of the search space that are different from the breath-first search of the levelwise algorithm. A comparison of the effectiveness of pruning in the two approaches is difficult. However, validity and minimality testing, and the mechanisms of pruning are less efficient in the depth-first algorithms.

Still another approach is to first compute all maximal invalid dependencies by a pairwise comparison of all rows, and then compute the minimal valid dependencies from the maximal invalid dependencies [7, 2, 9, 17]. The first part of such algorithms requires $\Omega(|r|^2)$ time with respect to the number of rows but is polynomial both in the number of rows and the number of attributes, while the second part requires exponential time in the number of attributes but has no dependency on the number of rows. The algorithm by Savnik and Flach [17] implements the second part with a depth-first search. During the search, the maximal invalid dependencies are used both for testing validity of dependencies and for pruning the search space. In Section 7, we present results of an experimental comparison between our algorithm and the algorithm of Savnik and Flach.

7 Performance

We have implemented the TANE algorithm described in this paper and experimented with it to find out how it performs in practice. We have two implementations of the algorithm. The first, scalable version, denoted simply as TANE, keeps most of the partitions on disk as described in Section 6. The other version, TANE/MEM, works completely in main memory. Both are available via the WWW page at http://www.cs.helsinki.fi/research/fdk/datamining/tane/.

To provide perspective, we performed the same experiments with the FDEP program of Savnik and Flach. The FDEP implementation is based on the algorithm described in [17] and is available at [16].

All algorithms, including FDEP, are written in C and were compiled with GNU C compiler with full optimizations. All experiments were run on the same 233 MHz Pentium PC with 64 MB of memory running Linux operating system. The times below are real times elapsed in the experiments as reported by Unix `time` command. We report "wall clock" times rather than CPU times in order to make the cost of I/O processing better visible and to give a fair account of the cost of swapping of TANE/MEM with large databases.

We ran the algorithms on a number of real life databases. The databases and their descriptions are available on the UCI Machine Learning Repository [13]. The number of rows, columns, and minimal dependencies found (N) in each database are shown in Table 1. The datasets labeled "Wisconsin breast cancer $\times$ n" are concatenations of n copies of the Wisconsin breast cancer data. The set of dependencies is the same in all of them. To avoid duplicate rows, all values in each copy were appended with a unique string specific to that copy.

The top three rows of Table 1 show the performance of the algorithms on three small databases. Our algorithms perform competitively in all cases. The Lymphography and Hepatitis databases are apparently very similar. However, our algorithms are much faster on Hepatitis than on Lymphography while FDEP is an order of magnitude faster on

Table 2: Performance of TANE/MEM on approximate dependency discovery.

Database	$\varepsilon = 0.0$		$\varepsilon = 0.01$		$\varepsilon = 0.05$		$\varepsilon = 0.25$		$\varepsilon = 0.5$	
	N	Time (s)	N	Time (s)	N	Time (s)	N	Time (s)	N	Time (s)
Lymphography	2730	89.1	3388	22.2	7031	4.89	578	0.32	21	0.01
Hepatitis	8250	16.6	9666	14.6	6617	9.27	350	0.06	160	0.01
W. breast cancer	46	0.28	113	0.27	126	0.23	181	0.12	18	0.02
W. breast cancer $\times$ 64	46	25.5	113	26.7	126	20.3	181	12.6	18	3.89
Chess	1	1.99	1	2.55	1	3.10	2	4.0	17	3.59

Table 3: Previously reported performance results and new results (running times in seconds). Numbers taken from other articles are marked with a "dag" (†); the source is given at the top of the column.

Database					Bell	Bitton	FDEP	Schlimmer	TANE
Name	$\|r\|$	$\|R\|$	$\|X\|$	N	et al [1]	et al [2]	[17]	[19]	
Lymphography*	150	19	7	641	> 33 h†	-	540 s†	-	-
Lymphography	148	19	19	2730	-	-	88 s	-	68.2 s
Rel1	7	7	7	8	-	0.02 s†	-	-	-
Rel6	236	60	60	56	-	994 s†	-	-	-
W. breast cancer	699	11	4	35	259 s†	-	15 s	4440 s†	0.34 s
W. breast cancer	699	11	11	46	533 s†	-	15 s	-	0.76 s
W. breast cancer $\times$ 128	89472	11	11	46	-	-	*	-	173 s
Books	9931	9	9	25	17040 s†	-	-	-	-

Lymphography than on Hepatitis. This is a good demonstration of how different approaches to pruning the search space have different effects.

The bottom part of Table 1 reports the performance of TANE on five large databases. For TANE/MEM and FDEP, some experiments are marked with (*) as infeasible; for TANE/MEM because of the lack of main memory, and for FDEP if it did not finish within 5 hours. TANE, on the other hand, found the dependencies in minutes and was never in danger of running out of memory.

Table 2 shows performance results for TANE/MEM in the approximate dependency discovery task, for different thresholds ε. Results for the Hepatitis, Wisconsin breast cancer, and Chess data sets are also presented graphically in Figure 3: N_ε/N_0 stands for the number of approximate dependencies found relative to the case for functional dependencies; similarly, $\text{Time}_\varepsilon/\text{Time}_0$ denotes the relative discovery time.

Overall, approximate dependencies are found efficiently. The number of dependencies found varies differently for each data set. Within a reasonable range $0 \leq \varepsilon \leq 0.1$, the time either increases slightly (Chess data set), decreases slightly (Wisconsin breast cancer), or drops significantly (Hepatitis). The drop is even stronger with the Lymphography data set (shown only in the table). Approximate dependencies could not be discovered in the Adult data set with TANE/MEM due to the lack of main memory.

To find out how the number of rows affects the algorithms, we ran a series of experiments with increasing number of rows. The relations were formed by concatenating multiple copies of the Wisconsin breast cancer data as described earlier. The results are illustrated in Figure 4. FDEP performs almost quadratically in the number of rows while our algorithms are very near linear. The sharp turn in the curve of TANE/MEM is caused by the algorithm running out of main memory and starting to use swap space. With the largest relation (357888 rows, 512 times Wisconsin breast cancer), TANE used about 22 MB of main memory and about 235 MB of temporary disk space.

The current implementations of our algorithms have not been optimized for memory and disk space consumption. With some form of data compression, the feasible range of our algorithms can be extended further. Even in their current form our algorithms can handle much larger databases than FDEP. Previously reported results are even worse [2, 17, 19, 1], see Table 3.

The table contains results published in previous articles,

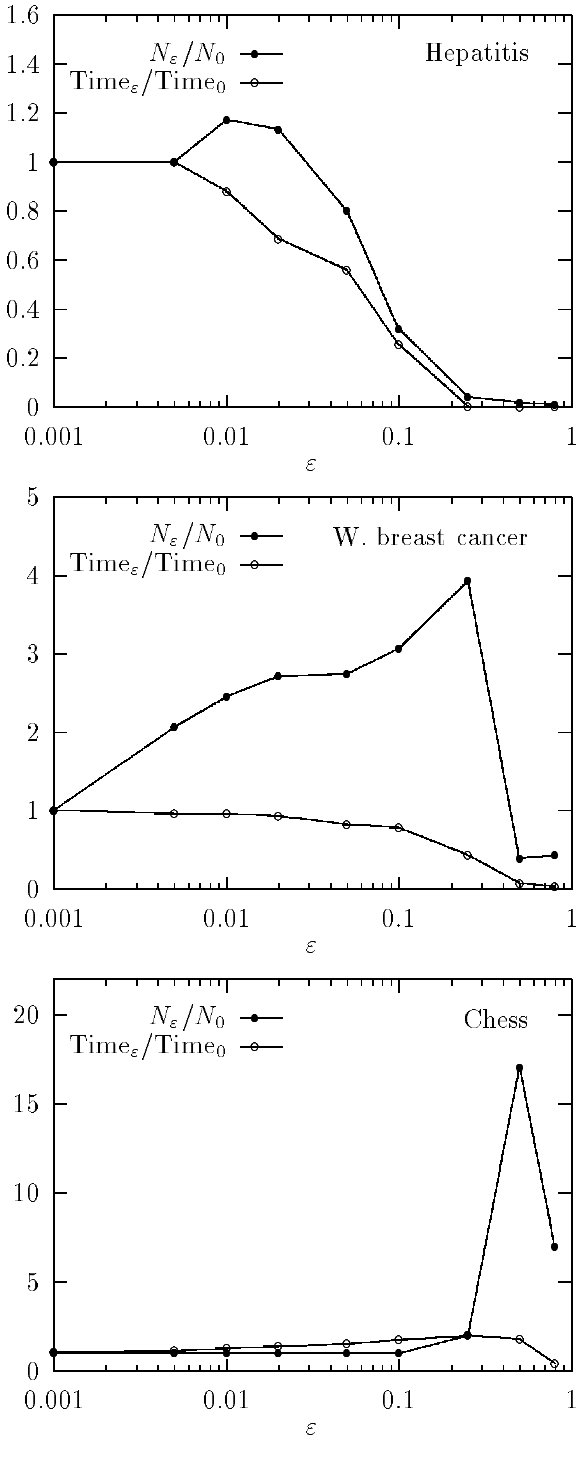

Figure 3: Performance of TANE/MEM for approximate dependencies in the Hepatitis (top), Wisconsin breast cancer (middle), and Chess (bottom) data sets.

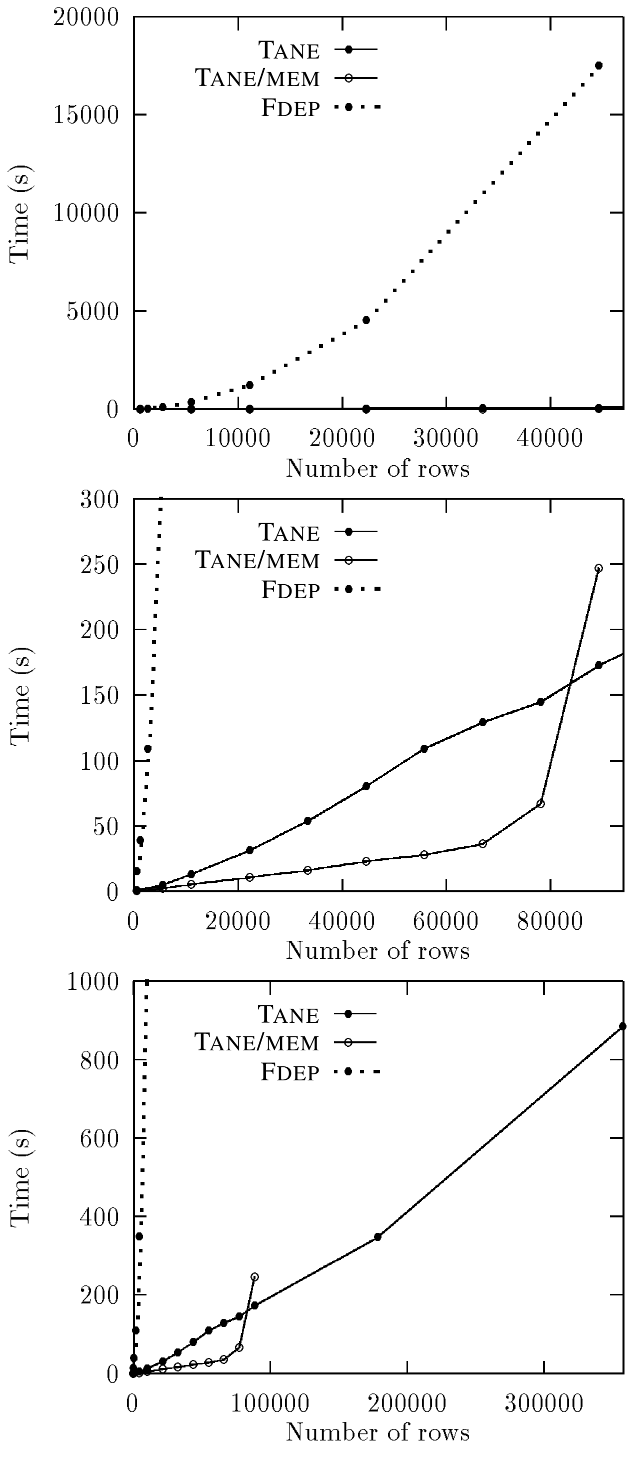

Figure 4: Performance of the algorithms when the number of rows increases. The three graphs show the same data on different scales.

all marked with (†), and results we obtained using TANE and the publicly available version of FDEP. Many of the databases used in previous articles are not publicly available, so results are missing altogether; these are marked with (-). Since the tests have been run in different environments direct comparisons are not possible. The results are, however, trend-setting.

Few notes are in order. First, $|X|$ denotes an upper limit for the number of attributes in the left-hand side of a dependency. Limiting the maximum size makes the task easier. N stands for the size of the results, i.e., the number of dependencies output. The outputs are, however, different: some algorithms only output a (minimal) cover of the dependencies that hold.

Second, the Lymphography data set marked with (*), used by Bell and Brockhausen [1] as well as by Savnik and Flach [17], is different from the one available from the UCI repository.

Third, it should be noted that Bell and Brockhausen [1] are the only ones to report results obtained on top of a commercial rdbms, whereas all others use flat files and specialized access methods.

For an overview, consider the Wisconsin breast cancer data set with the left-hand side limit $|X| = 4$. Although small and restricted, it is the only case for which there are results for four algorithms. TANE discovers dependencies in 0.34 seconds, FDEP in 15 s (larger by a factor $c = 44$), Bell and Brockhausen [1] in 259 s ($c = 760$), and Shclimmer [19] in 4440 s ($c = 130000$).

8 Concluding remarks

We have given a new algorithm for the discovery of functional and approximate dependencies from relations. The approach is based on considering partitions of the relation, and deriving valid dependencies from the partitions. The algorithm searches for dependencies in a breadth-first or levelwise manner. We showed how the search space can be pruned effectively, and how the partitions and dependencies can be computed efficiently. Experimental results and comparisons demonstrate that the algorithm is fast in practice, and that its scale-up properties are superior to previous methods. The method works well with relations of up to hundreds of thousands of rows.

The method is at its best when the dependencies are relatively small. When the size of the (minimal) dependencies is roughly one half of the number of attributes, the number of dependencies is exponential in the number of attributes, and the situation is more or less equally bad for any algorithm. When the dependencies are larger than that, the levelwise method that starts the search from small dependencies obviously is further from the optimum. The levelwise search can, in principle, be altered to start from the large dependencies. Then, however, the partitions could not be computed as efficiently.

There are also other interesting data mining applications for partitions. Association rules between attribute–value pairs can be computed with a small modification of the present algorithm. An equivalence class corresponds then to a particular value combination of the attribute set. By comparing equivalence classes instead of full partitions, we can find association rules. A possible future research direction is to use the unified view that partitions provide to functional dependencies and association rules, independently observed also in [3], to find an apt generalization of both and to develop an algorithm for discovering such rules.

Acknowledgements

The Adult, Hepatitis, Lymphography, and Wisconsin breast cancer data have been obtained from the UCI Machine Learning Repository [13]. The Lymphography domain originates from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data. The FDEP program was obtained from I. Savnik's FDEP Home Page [16]. Finally, we thank Heikki Mannila and Jean-François Boulicaut for helpful comments.

References

[1] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. Tech. Rep. LS-8 Report-14, University of Dortmund, Apr. 1995.

[2] D. Bitton, J. Millman, and S. Torgersen. A feasibility and performance study of dependency inference. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 635–641. IEEE Computer Society Press, 1989.

[3] M. M. Dalkilic, D. V. Gucht, and E. L. Robertson. CE: the classifier-estimator framework for data mining. In *Proceedings of the 7th IFIP 2.6 Working Conference on Database Semantics (DS-7)*, Leysin, Switzerland, Oct. 1997. Chapman and Hall.

[4] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions (extended version). Technical Report C-1997-79, Department of Computer Science, University of Helsinki, Nov. 1997.

[5] J. Kivinen and H. Mannila. Approximate dependency inference from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

[6] S. Kramer and B. Pfahringer. Efficient search of strong partial determinations. In E. Simoudis, J. Han,

and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 371–378, Portland, OR, Aug. 1996. AAAI Press.

[7] H. Mannila and K.-J. Räihä. Design by example: An application of Armstrong relations. *Journal of Computer and System Sciences*, 33(2):126–141, Oct. 1986.

[8] H. Mannila and K.-J. Räihä. Dependency inference. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases (VLDB'87)*, pages 155–158, Los Altos, CA, 1987. Morgan Kaufmann.

[9] H. Mannila and K.-J. Räihä. *The Design of Relational Databases*. Addison-Wesley, Menlo Park, California, 1992.

[10] H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40:237–243, 1992.

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

[12] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

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

[14] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 218–227, New Orleans, Louisiana, 1996. IEEE Computer Society.

[15] B. Pfahringer and S. Kramer. Compression-based evaluation of partial determinations. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 234–239, Montréal, Canada, Aug. 1995. AAAI Press.

[16] I. Savnik. FDEP home page [http://martin.ijs.si/savnik/fdep.html], June 1996.

[17] I. Savnik and P. Flach. Bottom-up induction of functional dependencies from relations. In G. Piatetsky-Shapiro, editor, *Knowledge Discovery in Databases, Papers from the 1993 AAAI Workshop (KDD'93)*, pages 174–185. AAAI, 1993.

[18] J. C. Schlimmer. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In G. Piatetsky-Shapiro, editor, *Proceedings of the Tenth International Conference on Machine Learning*, pages 284–290. Morgan Kaufmann, 1993.

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

[20] Z. Tari, O. Bukhres, J. Stokes, and S. Hammoudi. The reengineering of relational databases based on key and data correlations. In S. Spaccapietra and F. Maryanski, editors, *Searching for Semantics: Data Mining, Reverse Engineering, etc.* Chapman and Hall, 1998. To appear.

[21] G. E. Weddell. Reasoning about functional dependencies generalized for semantic data models. *ACM Transactions on Database Systems*, 17(1):32–64, Mar. 1992.