FD_Mine:Discovering Functional Dependencies in a Database Using Equivalences

Hong Yao, Howard J.Hamilton and Cory J. Butz

Technical Report TR 2002-04
August, 2002

ISBN 0-7731-0441-0

FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences

Hong Yao, Howard J.Hamilton, and Cory Butz

Department of Computer Science, University of Regina
Regina, SK, Canada, S4S 0A2
{yao2hong, hamilton, butz}@cs.uregina.ca

Abstract

Functional dependency (FD) traditionally plays an important role in the design of relational databases, and the study of FDs has produced a rich and elegant theory. The discovery of FDs from databases has recently become a significant research problem. In this paper, we propose a new algorithm, called FD_Mine, for the discovery of all minimal FDs from a database. FD_Mine takes advantage of the rich theory of FDs to guide the search for FDs. More specifically, the use of FD theory can reduce both the size of the dataset and the number of FDs to be checked by pruning redundant data and skipping the search for FDs that follow logically from the FDs already discovered. We show that our method is sound, that is, the pruning does not lead to loss of information. Experiments on 15 UCI datasets show that FD_Mine can prune more candidates than previous methods.

Keywords
Data mining, functional dependency, relational database theory

1. Introduction

This paper proposes a new method for finding functional dependencies in data. A *functional dependency* (FD) expresses a value constraint between attributes in a relation [7]. Formally, for a relational schema R with $X \subseteq R$ and $A \in R$, a functional dependency $X \rightarrow A$ can be defined if for all pairs of tuples t_1 and t_2 over R, for all $B \in X$ if $t_1[B] = t_2[B]$ then $t_1[A] = t_2[A]$. If A is not functionally dependent on any proper subset of X, then $X \rightarrow A$ is *minimal* [5]. Henceforth, all FDs mentioned in this paper can be assumed to be minimal.

FDs play an important role in relational theory and relational database design [7]. In early work, the study of FDs focused on the fact that data consistency could be guaranteed by using FDs to reduce the amount of redundant data. FDs were usually obtained from semantic models rather than from data. Current research is based on the fact that FDs may exist in a dataset that are independent of the relational model of the dataset. It is useful to discover these FDs. For example, from a database of chemical compounds, it is valuable to discover compounds that are functionally dependent on a certain structure attribute [5]. In addition, as a kind of data dependency [3, 12], a large dataset can be losslessly decomposed into a set of smaller datasets using the discovered FDs. As a result, the discovery of FDs from database has recently become a popular research problem [2, 4, 5, 6, 8, 11, 12, 13].

Research on FDs was an important aspect of relational database design [7] in the 1980s, and led to achievements such as the Armstrong Axioms, minimal cover, closure and an algorithm for decomposing a schema into third normal form while preserving dependency. The Armstrong axioms can be stated as three rules [7]: *Reflexivity rule*: if $Y \subseteq X$, then $X \rightarrow Y$; *Augmentation rule*: if $X \rightarrow Y$, then $XZ \rightarrow YZ$; *Transitivity rule*: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Y$. These rules can be applied repeatedly to infer all FDs implied by a set of FDs.

Relational database theory forms the underlying basis for the FD_Mine algorithm. Combining domain knowledge and data mining algorithm is better than using either alone. The Apriori algorithm [1] for discovering frequent itemsets can be used to find FDs from a dataset by setting the minimum support to *2/n*, where *n* is the number of instances in the dataset. In this case, the discovered FDs are the association rules with 100% confidence that can be formed from the itemsets. Typically, only a small fraction of the rules have 100% confidence. This constraint can be embedded into algorithm to increase efficiency, as we do in FD_Mine.

The FD_Mine algorithm improves efficiency by pruning redundant candidates. A *candidate* is a combinations of attributes over a dataset. To delete redundant candidates from the database, we use four pruning rules. For example, if the FDs $A \rightarrow B$ and $B \rightarrow A$ are discovered, then no further candidates containing B need be considered, since attributes A and B are equivalent. This pruning is valuable because the number of candidates increases exponentially with the number of attributes.

To prune redundant candidates, relationships among the FDs are analyzed. The set of FDs can be divided into two parts: FDs that can be inferred from the discovered FDs using the theory of relational databases, and those that cannot. The FD_Mine algorithm only examines the database to find the second type of FDs. Relevant aspect of relational database theory are collected here as lemmas, theorems, properties, and pruning rules. For example, if $A \rightarrow B$ and $C \rightarrow D$ are discovered to hold in a database, then $AC \rightarrow BD$ must also hold, so it does not need be checked in the database. By eliminating redundant data and pruning candidates, the FD_Mine algorithm improves mining performance.

The remainder of this paper organized as follows. A statement of the problem and an example of it are given in section 2. In section 3, the relationship among FDs is analyzed, and the formal definitions, lemmas, theorems, and properties are given. Pruning rules and FD_Mine algorithm are presented in section 4. A detailed example is also discussed in this section. Next, the experimental results are shown in section 5. Finally, conclusions are drawn in section 6.

2. Problem Statement

The problem addressed in this paper is to find all functional dependencies among attributes in a database relation. Specifically, we want to improve on previous proposed methods for this problem.

Early methods for discovering of FDs were based on repeatedly sorting and comparing tuples to determine whether or not these tuples meet the FD definition. For example, in Table 2.1, the tuples are first sorted on attribute A, then each pair of tuples that have the same value on attribute A is compared on attribute B, C, D, and E, in turn, to decide whether or not $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, or $A \rightarrow E$ holds. Then the tuples are sorted on attribute B and examined to decide whether or not $B \rightarrow A$, $B \rightarrow C$, $B \rightarrow D$ or $B \rightarrow E$ holds.

This process is repeated for C, D, E, AB, AC, AD, and so on. After the last candidate BCDE has been checked, all FDs will have been discovered. All candidates of five attributes are represented in Figure 2.1.

The disadvantage of this approach is that it does not utilize the discovered FDs as knowledge to obtain new knowledge. If $A \rightarrow B$ has been discovered, a check is still made to determine whether or not $AC \rightarrow B$ holds, by sorting on attributes AC and comparing on attribute B. Instead, $AC \rightarrow B$ can be directly inferred from the previously obtained $A \rightarrow B$ without sorting and comparing tuples again. This approach is inefficient because of this extra sorting and because it needs to examine every value of the candidate attributes to decide whether or not a FD holds. As a result, this approach is highly sensitive to the number of tuples and attributes. It is impracticable for a large dataset.

	A	B	C	D	E
t_1	0	0	0	1	0
t_2	0	1	0	1	0
t_3	0	2	0	1	2
t_4	0	3	1	1	0
t_5	4	1	1	2	4
t_6	4	3	1	2	2
t_7	0	0	1	0	0

Table 2.1 An example dataset.

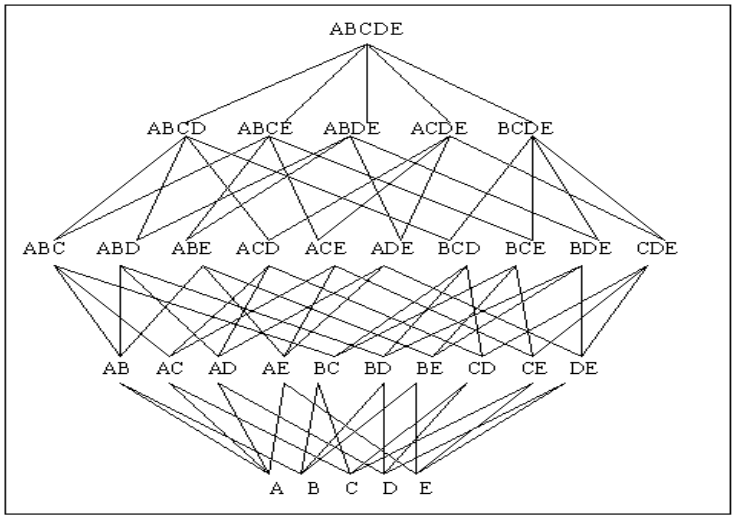

ABCDE
ABCD ABCE ABDE ACDE BCDE
ABC ABD ABE ACD ACE ADE BCD BCE BDE CDE
AB AC AD AE BC BD BE CD CE DE
A B C D E

Figure 2.1 Lattice for 5 attributes.

Recent papers have proposed algorithms that do not sort on any attribute or compare any values. Mannila et al. [8, 9, 10] introduced the concept of a *partition*, which places tuples that have the same values for an attribute into the same group. The problem of determining whether or not a FD holds on a given dataset can be addressed by comparing the number of the groups among partitions for various attributes. For the dataset r, shown in Table 2.1, the partition for attribute A can be denoted as $\Pi_A(r) = \{\{t_1, t_2, t_3, t_4, t_7\}, \{t_5,$

t_6}}. Because the values of tuples t_1, t_2, t_3, t_4, and t_7 on attribute A are all the same, they are assigned to the same group. Similarly, because the values of t_5 and t_6 are the same, they are placed in another group. The partition for the attribute combination AD for Table 2.1 is $\Pi_{AD}(r) = \{\{t_1, t_2, t_3, t_4, t_7\}, \{t_5, t_6\}\}$. The *cardinality of the partition* $|\Pi_A(r)|$, which is the number of groups in partition Π_A, is 2, and $|\Pi_{AD}(r)|$ is 2 too. Because $|\Pi_A(r)|$ is equal to $|\Pi_{AD}(r)|$, A→D can be obtained [5].

Algorithm TANE [5] uses the partition concept to discover FDs. In addition, the set of the candidates are pruned based on the discovered FDs. For instance, if AC→B has been discovered, then ACD→B and ACDE→B can be inferred from AC→B without checking the data, so candidates ACD and ACDE are redundant. According to the dependencies in a dataset, only a portion of the lattice shown in Figure 2.1 may need to be traversed to find all FDs in the relation.

Algorithm FUN [11, 12] uses a procedure that checks for *embedded FDs*, which are FDs that hold on the projection of the dataset. By using embedded FDs, other candidates can be pruned. For example, suppose that A→B holds over ABC. If $|\Pi_{AB}(r)| > |\Pi_{BC}(r)|$, then BC→A does not hold over ABC. On synthetic datasets with correlation rates of 30% to 70%, FUN is faster than TANE. Apparently, FUN was not tested on any UCI datasets.

Our research addresses two related questions. First, can other information from discovered FDs be used to prune more candidates than previous approaches? Secondly, can this pruning be done so that the overall efficiency of the algorithm is improved? We address both these problems by further considering the theoretical properties of FDs, formulating the FD_Mine algorithm to take advantage of these properties, and testing the algorithm on a variety of datasets.

3. Theoretical Analysis of Functional Dependencies

In this section, formal definitions, lemmas, theorems, and properties relevant to FDs are presented. The relationships among FDs are analyzed with particular attention to equivalent candidates and nontrivial closure.

3.1 Equivalent Candidates

Definition 3.1 Let X and Y be candidates over a dataset D, if X→Y and Y→X hold, then X and Y are said to be *equivalent candidates,* denoted as $X \leftrightarrow Y$.

Using Definition 3.1 and the Armstrong axioms [7], Lemma 3.1, and Lemma 3.2 can be obtained.

Lemma 3.1. Let X, Y and Z be candidates over D. If $X \leftrightarrow Y$ and XW→Z hold, then YW→Z holds.

Proof: Since $X \leftrightarrow Y$ holds, then Y→X holds. Since XW→Z holds, then by the Armstrong augmentation and transitivity rules, YW→Z holds. □

Lemma 3.2. Let X, Y and Z be candidates over D. If $X \leftrightarrow Y$ and WZ→X hold, then WZ→Y holds.

Proof: since $X \leftrightarrow Y$ holds, then X→Y holds. Since WZ→X holds, by the Armstrong transitivity rule, WZ→Y holds. □

Using Lemmas 3.1 and 3.2, the number of candidates and possibly the size of the dataset that needs to be checked can be reduced.

Example 3.1. In Table 3.1(a), $A \rightarrow D$, $D \rightarrow A$, $AB \rightarrow C$, $BD \rightarrow C$, $BC \rightarrow A$, and $BC \rightarrow D$ hold. According to Definition 3.1, since $A \rightarrow D$ and $D \rightarrow A$, then $A \leftrightarrow D$ holds. By examining all entries for attribute A, B, and C, we can determine that $AB \rightarrow C$ and $BC \rightarrow A$ hold. Without Lemma 3.1 and 3.2, we need to examine all entries in Table 3.1(a) again to determine whether or not $BD \rightarrow C$ and $BC \rightarrow D$ hold. But with Lemma 3.1, the $BD \rightarrow C$ can be inferred, and with Lemma 3.2, $BC \rightarrow D$ can also be inferred. As a result, attribute D and all its values are redundant to further searching for FDs. No further candidates containing D need to be examined. If the dataset is not needed for other purposes, it may make sense to remove attribute D and all its values from the dataset. Even if the attribute and its data are not removed from the dataset, they can be ignored in all subsequent processing by FD_Mine. The result after removal is shown in Table 3.1(b).

	A	B	C	D
t_1	0	0	0	1
t_2	0	1	0	1
t_3	0	2	0	1
t_4	0	3	1	1
t_5	4	1	1	2
t_6	4	2	1	2
t_7	0	0	0	1

(a) Before

	A	B	C
t_1	0	0	0
t_2	0	1	0
t_3	0	2	0
t_4	0	3	1
t_5	4	1	1
t_6	4	2	1
t_7	0	0	1

(b) After

Table 3.1 Effect of Removing a Redundant Attribute

3.2 Nontrivial Closure

Definition 3.2 Let F be a set of FDs over a dataset D and X be a candidate over D. The *closure of candidate X* with respect to F, denoted Closure(X), is defined as $\{Y \mid X \rightarrow Y$ can be deduced from F by Armstrong's axioms$\}$. The *nontrivial closure of candidate X* with respect to F, denoted Closure'(X), is defined as Closure'(X) = Closure(X) – X.

Example 3.2. Let V, W, X, Y, Z be candidates of a dataset D and $F = \{X \rightarrow Y, X \rightarrow Z, W \rightarrow V\}$ be a set of FDs over D. So Closure(X) is {X, Y, Z}, and Closure'(X) is {Y, Z}.

By using the following theorem, the equivalence $X \leftrightarrow Y$ can be checked by comparing the nontrivial closures of the two candidates X and Y.

Theorem 3.1 Let X and Y be two candidates of dataset D, $Z = X \cap Y$, and let Closure'(X) and Closure'(Y) be the nontrivial closures of candidates X and Y, respectively. If Closure'(X) $\supseteq Y–Z$, and Closure'(Y) $\supseteq X–Z$, then $X \leftrightarrow Y$.

Proof:

Since $X \cap Y=Z$, Closure'(X) $\supseteq$ Y-Z, then $X \rightarrow Y–Z$ holds. Since $Z \subset X$, then $X \rightarrow Z$ also holds, so $X \rightarrow Y$ holds. In the same way, $Y \rightarrow X$ also holds. Therefore $X \leftrightarrow Y$ holds. □

Example 3.3 Let W, X, Y, Z be candidates of dataset D and $F=\{XZ \rightarrow YW, YZ \rightarrow X\}$ be a set of FDs over D. So Closure'(XZ)={W, Y} and Closure'(YZ)={X}. Since $XZ \cap YZ=Z$, $YZ–Z=Y \subset$ Closure'(XZ) and $XZ–X=X \subset$ Closure'(YZ), so $XZ \leftrightarrow YZ$ holds.

Theorem 3.1 can be used to determine whether two candidates are equivalent by comparing their computed nontrivial closures. The nontrivial closures can be used in this manner without first comparing the partitions of the attributes. Whenever an equivalence $X \leftrightarrow Y$ is founded, partition Π_Y can be deleted, resulting in a significant saving of storage space.

In addition, nontrivial closure allows more convenient calculation than closure itself. For example, if $X \rightarrow Y$, and $X \rightarrow Z$ hold, then Closure(X) is {X, Y, Z}. If $XY \rightarrow Z$ holds, then Closure(XY) is also {X, Y, Z}. As a result, Closure(X) is equal to Closure(XY). But X and XY may not be equivalent candidates because Closure'(X) is not equal to Closure'(XY).

3.3 Other properties of Functional Dependencies

Definition 3.3 [5]**.** Let $t_1, t_2, \ldots, t_n$ be all tuples in a dataset D, and X be a candidate over D. The *partition over X*, denoted Π_X, is a set of the groups, such that t_i and t_j are in the same group iff $t_i[X] = t_j[X]$. The number of the groups in the partition is called the *cardinality* of the partition, denoted $|\Pi_X|$.

Example 3.4. In Table 2.1, the partition of attribute A is $\Pi_A = \{\{t_1, t_2, t_3, t_4, t_7\}, \{t_5, t_6\}\}$, and the cardinality of partition Π_A is $|\Pi_A| = 2$.

Suppose the partition over candidate X and the partition over candidate Y are both known. By analyzing the cardinality of the partitions, we can determine whether or not $X \rightarrow Y$ holds. Using the following property.

Property 3.1 [5]**.** Let Π_X and Π_Y be partitions over a dataset. $X \rightarrow Y$ holds if and only if $|\Pi_X|=|\Pi_{XY}|$.

According to the definition of an FD and the Armstrong axioms, the following properties can be inferred.

Property 3.2. Let X and Y be candidates over dataset, and Closure'(X) and Closure'(Y) be the nontrivial closures of candidates X and Y, respectively. If Closure'(XY) is the nontrivial closure of candidate XY, then $\text{Closure'}(X) \cup \text{Closure'}(Y) \subseteq \text{Closure'}(XY)$ holds.

Property 3.2 indicates that if for any X, $X \rightarrow Y$ holds, then $XZ \rightarrow Y$ also holds for any Z. As a result, $XZ \rightarrow Y$ can be inferred without checking the data. It also means a bottom-up algorithm can be used to prune candidates. For example, let V, W, X, Y, Z be attributes of a dataset D. if closure'(X) = {Z} and closure'(Y) ={W}, then $\{W, Z\} \subseteq$ closure'(XY). So $XY \rightarrow W$ and $XY \rightarrow Z$ do not need to be checked. Only $XY \rightarrow V$ needs to be checked.

Property 3.3. Let R be a relational schema and X be an candidate of R over a dataset D. If $X \cup \text{Closure'}(X) = R$, then X is a key.

Property 3.4 [7]**.** Let R be a relational schema and let $XY \subseteq S \subseteq R$. If a functional dependency $X \rightarrow Y$ holds over S, then $X \rightarrow Y$ also holds over R.

Property 3.4 indicates that if a FD holds in XY, a subset of R, it must hold in the whole relational schema R. So $X \rightarrow Y$ only needs to be checked over XY, which reduces the checking scope. This property also indicates that a level-wise search, such as that used in the Apriori algorithm, can be used to discover FDs. At level 1, FDs such as $X \rightarrow Y$ are discovered. At level 2, FDs such as $XY \rightarrow Z$ are discovered. At level k-1, FDs such as $X_1X_2 \ldots X_{k-1} \rightarrow X_k$ are discovered.

The downward closure property for itemsets can be described as follow: if any subset of size (k-1) of a k-itemset is not frequent then the k-itemset is also not frequent. This property can be used to reduce the number of candidate k-itemsets, and it is the basis of the Apriori algorithm. Due to this property, typically only a small fraction the possible k-itemsets are considered as candidate at level k, and of them many are pruned because at least one of their constitent (k-1)-itemsets is not frequent

A similar property exists concerning the FD that need to be check.. In this paper, we say a FD is *checked* if Property 3.1 is used to examine whether or not a FD holds. Otherwise, we say a FD is *not checked*. In the latter case, whether or not a FD holds can be deduced from the already discovered FDs rather than by comparing partitions.

Definition 3.4. Let $X_1, X_2, \ldots, X_k, X_{k+1}$ be (k+1) attributes over a database D. If a $X_1X_2 \ldots X_k \rightarrow X_{k+1}$ is a FD with k attributes on its left hand side, then it is called a *k-level* FD.

Property 3.5 [5]**.** If $X_1X_2 \ldots X_{k-1}X_k \rightarrow X_{k+1}$ is a k-level FD, then it needs to be checked when none of its (k-1)-level subsets $X_{i(1)}X_{i(2)} \ldots X_{i(k-1)} \subset X_1X_2 \ldots X_{k-1}X_k$ satisfies $X_{i(1)}X_{i(2)} \ldots X_{i(k-1)} \rightarrow X_{i(k)}$.

Example 3.5 Let W, X, Y, Z be attributes of a dataset D. If we check XY and obtain Closure'(XY)={Z}, then $XYZ \rightarrow W$ does not be checked. From Closure'(XY)={Z}, we know $XY \rightarrow Z$ holds and $XY \rightarrow W$ does not hold**.** So we can conclude that $XYZ \rightarrow W$ does not hold without checking WXYZ. On the contrary, if $XYZ \rightarrow W$ were to hold, since $XY \rightarrow Z$, then $XY \rightarrow W$ would hold, which contradicts the fact that $XY \rightarrow W$ does not hold.

4. The FD_Mine Algorithm

In this section, four rules are first given for pruning redundant data and candidates. Next, the FD_MINE algorithm is presented. Finally, the algorithm is explained by means of an example.

4.1 Pruning Rules

The well-developed theory of functional dependency provides many ways of deducing one FD from another. As a result, for many databases, the amount of data and the number of candidates that must be checked can be reduced. Based on the lemmas and properties given in the previous section, the following pruning rules are defined. Of these rules, the first reduces the size of the database to be searched, and the remainder reduce the number of candidates.

Before giving pruning rule about equivalent candidates, we henceforth restrict the term equivalent candidate $X \leftrightarrow Y$ to a canonical form such that X is always the candidate that is generated earlier than candidate Y. For example, for a relational schema R={A, B, C, D}, canonical form corresponds to alphabetical order. So $A \leftrightarrow B$ and $BC \rightarrow D$ are in canonical form, but $D \leftrightarrow B$ and $CB \rightarrow D$ are not.

Pruning rule 1. If $X \leftrightarrow Y$, then candidate Y can be deleted.

Rule 1 reduces the search space by eliminating redundant attributes and all their values from database. In addition, if $X \leftrightarrow Y$ holds, then any superset YW of Y does not need to be checked. It also eliminates redundant candidates. For example, because $XY \leftrightarrow WZ$ holds, if $XYT \rightarrow U$ holds, then $WZT \rightarrow U$ also holds.

Pruning rule 2 [5]**.** If X is a key, then any superset XY of X does not need to be checked.

Pruning rule 3. If Closure'(X) and Closure'(Y) are the nontrivial closures of attributes X and Y, respectively, then $XY \rightarrow$ Closure'(X) $\cup$ Closure'(Y) does not need to be checked.

This pruning rule is justified by Property 3.2. It means that if $X \rightarrow Z$ and $Y \rightarrow W$ hold, then $XY \rightarrow WZ$ also holds.

Pruning rule 4 [5]**.** Let $X_1X_2\ldots X_k \rightarrow X_{k+1}$ be a k-level FD. If any subsets $X_{i(1)}X_{i(2)}\ldots X_{i(k-1)}$ of $X_1X_2\ldots X_k$ satisfies $X_{i(1)}X_{i(2)}\ldots X_{i(k-1)} \rightarrow X_{i(k)}$, then $X_1X_2\ldots X_k \rightarrow X_{k+1}$ does not need to be checked.

This pruning rule is justified by Property 3.5. For example, for attributes X, Y, and Z, suppose $XY \rightarrow Z$ holds. If $XY \rightarrow W$ holds, then $XYZ \rightarrow W$ also holds. If $XY \rightarrow W$ does not hold, then $XYZ \rightarrow W$ also does not hold.

The FD_Mine algorithm uses the above four pruning rules to prune the database and the candidates.

4.2 FD_Mine Algorithm

The FD_Mine algorithm combines the Apriori algorithm [1] and the pruning rules given in section 4.1. It is similar to TANE algorithm but TANE only uses 2 of the 4 rules (pruning rule 2 and 4) used by FD_Mine. It uses a level-wise search, where results from level k are used to explore level k+1. First, at level 1, all FDs $X \rightarrow Y$ where X and Y are single attributes are found and stored in FD_SET F_1. The set of candidates that are considered at this level is denoted L_1. F_1 and L_1 are used to generate the candidate $X_i X_j$ of L_2. At level 2, all FDs of the form $X_i X_j \rightarrow Y$ are found and stored in FD_SET F_2, F_1, F_2, L_1, and L_2 are used to generate the candidates of L_3, and so on, until the candidates at level L_{n-1} have been checked or no candidates remain, i.e., $L_k = \phi$ ($k \leq n-1$).

Before introducing the FD_Mine algorithm, the following identifies are introduced.

- *Candidate:* an attribute combination that is checked to see whether it functionally decides other attributes.
- *CANDIDATE_SET*: a set of candidates.
- *Closure'*(*X*): the nontrivial closure of candidate X.
- Π_X: the partition of candidate X.

- $|\Pi_X|$: the cardinality of the partition of candidate X.
- *FD_SET*: the set of discovered functional dependencies, each in the form $X \rightarrow Y$.
- *EQ_SET*: the set of discovered equivalences, each in the form $X \leftrightarrow Y$.
- *KEY_SET*: the set of discovered keys.

```
Algorithm FD_Mine
Purpose: To discover all functional dependencies in a dataset.
Input: Database D and its attributes $X_1, X_2, ... , X_m$
Output: FD_SET, EQ_SET and KEY_SET
{
        1. Initialization Step
             set R = $\{X_1, X_2, ..., X_m\}$, set FD_SET = $\phi$,
             set EQ_SET = $\phi$, set KEY_SET = $\phi$
             set CANDIDATE_SET = $\{X_1, X_2, ..., X_m\}$
             $\forall X_i \in$ CANDIDATE_SET, set Closure'[$X_i$] = $\phi$
        2. Iteration Step
             while CANDIDATE_SET $\neq \phi$ do
             {
                 $\forall X_i \in$ CANDIDATE_SET do
                 {
                     ComputeNonTrivialClosure($X_i$)
                     ObtaintFDandKey
                 }
                 ObtainEQSet(CANDIDATE_SET)
                 PruneCandidates(CANDIDATE_SET)
                 GenerateNextLevelCandidates(CANDIDATE_SET)
             }
        3. Display(FD_SET, EQ_SET, KEY_SET)
}
```

```
Procedure ComputeNonTrivialClosure($X_i$)
{
        for each $Y \subset R - X_i$ – Closure'[$X_i$] do
        if $|\Pi_{Xi}| = |\Pi_{XiY}|$ then
              add Y to Closure'[$X_i$]
}
```

```
Procedure ObtaintFDandKey ($X_i$)
{
        add $X_i \rightarrow$ Closure'[$X_i$] to FD_SET
        if (R = $X_i \cup$ Closure'[$X_i$] ) then
                add $X_i$ to KEY_SET
}
```

```
Procedure  **ObtainEQSet**(CANDIDATE_SET)
{
    for each $X_i \in$ CANDIDATE_SET do
        for $\forall X \rightarrow$ Closure'(X) $\in$ FD_SET do
        {
            set $Z = X \cap X_i$
            if (Closure'(X) $\supseteq X_i - Z$ and Closure'[$X_i$] $\supseteq X - Z$ )  then
                add   $X \leftrightarrow X_i$  to  EQ_SET
        }// end for
}//end procedure
```

```
Procedure **PruneCandidates**(CANDIDATE_SET)
{
    for each $X_i \in$ CANDIDATE_SET do
    {
            if $\exists X \in$ CANDIDATE_SET  such that $X \leftrightarrow X_i \in$ EQ_SET  then
                delete $X_i$  from CANDIDATE_SET   //pruning rule 1
            if  $\exists X_i \in$ KEY_SET then
                delete $X_i$  from CANDIDATE_SET   //pruning rule 2
    }
}
```

```
Procedure **GenerateNextLevelCandidates**(CANDIDATE_SET )
{
        for each $X_i \in$ CANDIDATE_SET do
          for each $X_j \in$ CANDIDATE_SET do
            if ($X_i[1]=X_j[1], \ldots, X_i[k-2] = X_j[k-2]$ and $X_i[k-1] < X_j[k-1]$) then
            {   set $X_{ij} = X_i$ join  $X_j$
                if $\exists X_i \rightarrow X_j[k-1] \in$ FD_SET then delete  $X_{ij}$   //pruning rule 4
                else
                {   compute the partition $\Pi_{Xij}$ of $X_{ij}$
                    if ($R = X_{ij} \cup$ Closure'[$X_{ij}$] ) then add  $X_{ij}$ to KEY_SET
                    else
                    { add $X_{ij}$  to CANDIDATE_SET
                      set Closure'($X_{ij}$) = Closure'($X_i$ ) $\cup$ Closure'($X_j$)   //pruning rule 3
                    }
                }
            } // end if
} // end procedure
```

4.3 Time Complexity Analysis

The time complexity of the algorithm depends on the number of attributes *m*, the number of instances *n*, and the degree of correlation among the attributes. The more FDs or equivalent attributes are present in the dataset, the more search space can be pruned by using learned information as prior knowledge. This pruning reduces the running cost of the algorithm. The time required varies for different datasets, because the number and levels of the FDs that are discovered vary for different datasets. The worst case occurs when no FDs can be found in the dataset, and all combinations of the attributes are tested. The worst case time complexity is $O(n2^m)$. The time required to compute the partition for a candidate is $O(n)$ [5], and the number of combinations of all attributes is $C_1^m +$ $C_2^m + \ldots + C_{m-1}^m = 2^m$.

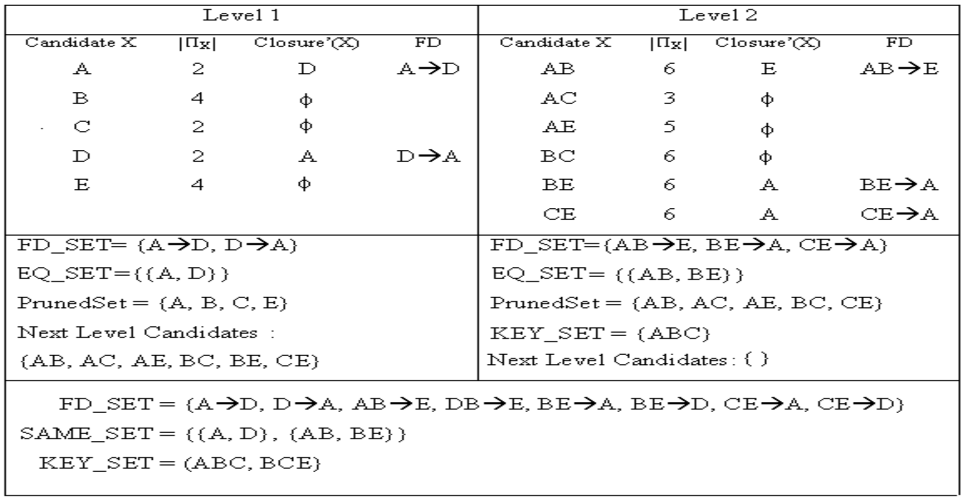
4.4 A Detailed Example

Example: Suppose that FD_Mine is applied to dataset D, as shown in Table 4.1, with R = {A, B, C, D, E}.

	A	B	C	D	E
t_1	0	0	0	2	0
t_2	0	1	0	2	0
t_3	0	2	0	2	2
t_4	0	3	1	2	0
t_5	4	1	1	1	4
t_6	4	3	1	1	2
t_7	0	0	1	2	0

Table 4.1 An example dataset

Table 4.2 summarizes the actions of FD_Mine. In iteration 1, since $|\Pi_A| = |\Pi_{AD}| = 2$, Closure'(A) is set to D, and A→D is deduced. In same way, D→A is discovered, so the equivalence $A \leftrightarrow D$ is obtained. As a result, we only need to combine A, B, C, and E to generate the next level candidates {AB, AC, AE, BC, BE, CE}. At the same time, the nontrivial closure of each generated candidate is computed. For example, the closure'(AB) = closure'(A) $\cup$ closure'(B) = {D} $\cup \phi = \{D\}$. In iteration 2, for candidate AB, only AB→C and AB→E need to be checked, because R–{A, B}–Closure'(AB) = {A, B, C, D, E}–{A, B}–{D}={C, E}. Since $|\Pi_{AB}| = |\Pi_{ABE}| = 6$, then AB→E is obtained. In the same way, at this level, BE→A and CE→A are also discovered, so the equivalence $AB \leftrightarrow BE$ is obtained. As a result, we only need to combine AB, AC, AE, BC, and CE to form the level 3 candidates, which are {ABC, ACE}. Since CE→A, ACE is pruned by pruning rule 4. Since AB→E, then ABC→E. Since $A \leftrightarrow D$, then ABC→D, so ABC is a key, and ABC is also pruned by pruning rule 2. No other candidate remains, so the algorithm halts.



Level 1				Level 2			
Candidate X	$\|\Pi_X\|$	Closure'(X)	FD	Candidate X	$\|\Pi_X\|$	Closure'(X)	FD
A	2	D	A→D	AB	6	E	AB→E
B	4	φ		AC	3	φ	
C	2	φ		AE	5	φ	
D	2	A	D→A	BC	6	φ	
E	4	φ		BE	6	A	BE→A
				CE	6	A	CE→A

FD_SET= {A→D, D→A}
EQ_SET={{A, D}}
PrunedSet = {A, B, C, E}
Next Level Candidates :
{AB, AC, AE, BC, BE, CE}
				FD_SET={AB→E, BE→A, CE→A}
				EQ_SET= {{AB, BE}}
				PrunedSet = {AB, AC, AE, BC, CE}
				KEY_SET = {ABC}
				Next Level Candidates : { }			

FD_SET = {A→D, D→A, AB→E, DB→E, BE→A, BE→D, CE→A, CE→D}
SAME_SET = {{A, D}, {AB, BE}}
KEY_SET = (ABC, BCE}

Table 4.2 Trace of FD_Mine Applied to the Table 4.1 Dataset

5. Experimental Results

In this section, experimental results on fifteen UCI datasets are summarized, and then a comparison between FD_Mine and another algorithm called TANE is presented.

5.1 UCI Dataset Results

FD_Mine was applied to fifteen datasets, obtained from the UCI Machine Learning Repository [14]. The results are summarized in Table 5.1, where column 4 represents the number of FDs that need to be checked on data, column 5 represents the number of discovered FDs, column 6 represents the number of discovered equivalences, column 7 represents the number of discovered keys, and column 8 is the elapsed time in seconds, measured on a SGI R12000 processor.

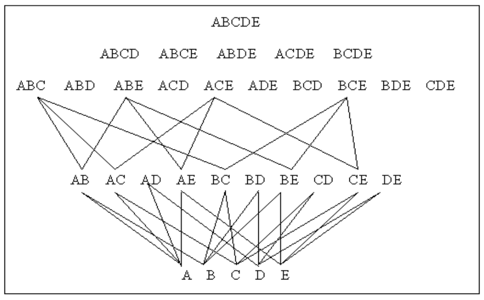
Dataset Name	# of attributes	# of instances	# of checks	# of FDs	# of EQs	# of keys	Time (secs)
Abalone	8	4,177	594	60	8	4	1
Balance-scale	5	625	70	1	0	1	0
Breast-cancer	10	191	5,095	3	0	1	0
Bridge	13	108	15,397	62	48	1	2
Cancer-Wisconsin	10	699	4,562	19	0	1	1
Chess	7	28,056	434	1	0	1	3
Crx	16	690	79,418	1,099	235	12	10
Echocardiogram	13	132	2,676	583	30	3	0
Glass	10	142	405	119	4	1	0
Hepatitis	20	155	1,161,108	8,250	4,862	7	1,327
Imports-85	26	205	2,996,737	4,176	19,888	4	8,322
Iris	5	150	70	4	0	1	0
Led	8	50	477	11	1	1	0
Nursery	9	12,960	2,286	1	0	1	16
Pendigits	17	7,494	223,143	29,934	671	4	920

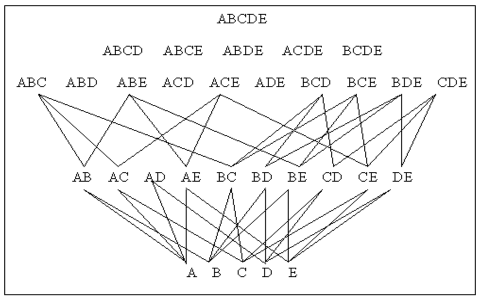
Table 5.1 Experimental Results

The timing results show that the processing time is mainly determined by the number of attributes. For example, for the Imports-85 dataset, the running time is very long, even though it only has 205 instances, because it has 26 attributes. The chess dataset has nearly 30,000 instances, but only 6 attributes, and its running time is much shorter (3 seconds). This results are consistent with the O($n2^m$) complexity of the algorithm. The results also show that for the datasets examined, the possible number of equivalences increases with the number of attributes, because the number of candidates increases exponentially with the number of attributes.

5.2 Algorithm Comparison

In this section, an empirical comparison between FD_Mine and TANE is given. TANE was selected for comparison because it establishes the theoretical framework of the problem and has been tested on an extensive set of UCI datasets [14]. Since FD_Mine reduces the data and candidates by discovering equivalences, it needs to check fewer FDs than TANE. For the dataset given in Table 4.1, Figure 5.1(a) shows the semi-lattice for FD_Mine, and Figure 5.1(b) shows that for TANE. Each node represents a combination of attributes. If an edge is shown between nodes X and XY, then X→Y needs to be checked. Hence, the number of edges is the number of FDs that need to be checked on data by the algorithm. Both semi-lattices shown in Figure 5.1 have fewer edges than the lattice shown in Figure 2.1. In addition, the semi-lattice for FD_Mine has fewer edges than that for TANE. Incidently, for this data, FUN [11, 12] requires the same number of checks as TANE.


ABCDE
ABCD ABCE ABDE ACDE BCDE
ABC ABD ABE ACD ACE ADE BCD BCE BDE CDE
AB AC AD AE BC BD BE CD CE DE
A B C D E

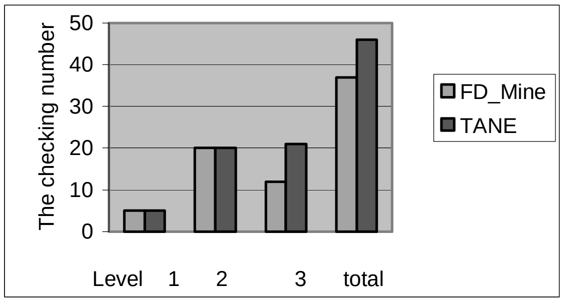

ABCDE
ABCD ABCE ABDE ACDE BCDE
ABC ABD ABE ACD ACE ADE BCD BCE BDE CDE
AB AC AD AE BC BD BE CD CE DE
A B C D E

(a) FD_Mine (b) TANE (or FUN)

Figure 5.1 Semi-lattices for the Data in Table 4.1

Figure 5.2 compares the number checks at each level. FD_Mine generates fewer candidates at the level 3, because of equivalence pruning using pruning rule 1.

	FD_Mine	TANE
Level 1	5	5
Level 2	20	20
Level 3	12	21
Total	37	46



(a) Count (b) Chart

Figure 5.2 Number of FDs checked for Data in Table 4.1

Table 5.2 compares the number of FDs that are checked on data by FD_Mine and TANE for the same 15 UCI datasets used earlier. For example, in the Iris dataset, FD_Mine checks 2,676 FDs on data, but TANE checks 2,766 FDs. Table 5.3 shows more detailed results for the Imports-85 and Hepatitis datasets.

Figure 5.3 presents the data from Table 5.3 in chart form. At levels 1 through 5, both algorithms check approximately the same number of FDs, but at levels 6 through 11, FD_Mine checks fewer FDs than TANE, because it prunes more unnecessary candidates than TANE by using the equivalences and FDs discovered at previous levels.

Dataset Name	# of Attributes	# of Instances	# of FDs Checked	
			FD_MINE	TANE
Abalone	8	4177	594	594
Balance-scale	5	625	70	70
Breast-cancer	10	191	5,095	5,095
Bridge	13	108	15,397	15,626
Cancer-Wisconsin	10	699	4,562	4,562
Chess	7	28,056	434	434
Crx	16	690	79,418	130,605
Echocardiogram	13	132	2,676	2,766
Glass	10	142	405	455
Hepatitis	20	155	1,161,108	1,272,789
Imports-85	26	205	2,996,737	3,564,176
Iris	5	150	70	70
Led	8	50	477	477
Nursery	9	12,960	2,286	2,286
Pendigits	17	7,494	223,143	227,714

Table 5.2 Comparison on UCI Datasets

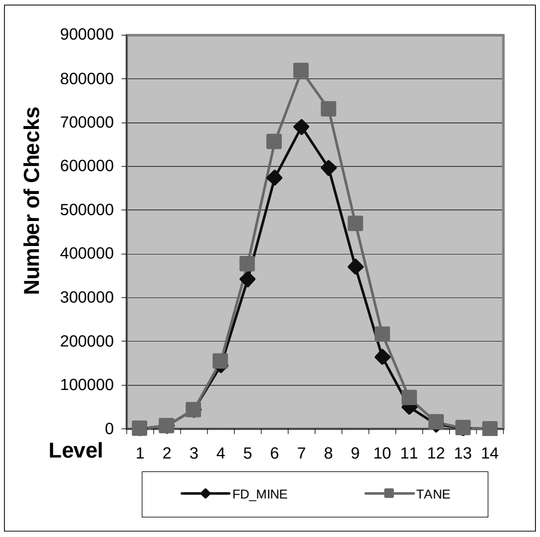
LEVEL	FD_MINE	TANE
1	650	650
2	7,309	7,309
3	44,021	44,021
4	145,647	154,985
5	342,114	377,366
6	573,815	656,106
7	689,825	817,849
8	596,476	731,203
9	370,399	469,283
10	164,187	216,510
11	50,794	71,000
12	10,270	15,703
13	1,176	2,070
14	54	121
Total	2,996,737	3,564,176

(a) Imports-85

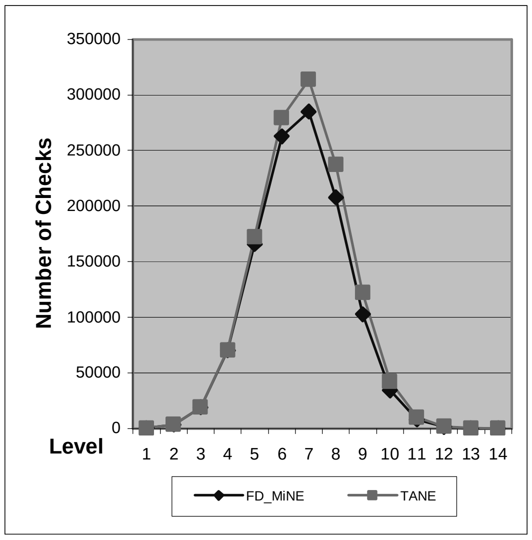
LEVEL	FD_MINE	TANE
1	380	380
2	3,420	3,420
3	19,074	19,074
4	70,099	70,484
5	165,906	172,024
6	262,905	279,461
7	284,865	313,801
8	207,780	237,601
9	102,808	122,133
10	34,332	42,300
11	8,098	10,210
12	1,320	1,735
13	116	160
14	5	6
Total	1,161,108	1,272,789

(b) Hepatitis

Table 5.3 Number of FDs checked on the Import and Hepatitis Datasets



(a) Imports-85



(b) Hepatitis

Figure 5.3 Graphical Comparison on the Import and Hepatitis Datasets

6. Conclusion

This paper addresses the problem of discovering all minimal functional dependencies in a database. We provide three research contributions towards this problem in this paper.

First, we identify several properties of relational databases relevant to the search for functional dependencies. Using these, we proved a theorem that allows equivalences among attributes to be identified based on nontrivial closures is proven. We also listed properties of functional dependencies, equivalences, and nontrivial closures that allow them to be used during the knowledge discovery process.

The second contribution is the FD_Mine algorithm, which is introduced here. This algorithm finds all minimal functional dependencies in a database. Like TANE, FD_Mine is based on partitioning the database and comparing the number of partitions. However, FD_Mine provides additional pruning rules, based on our analysis of the theoretical properties of functional dependencies. These pruning techniques are guaranteed not to eliminate any valid candidates, and whenever they are relevant, they reduce the size of the dataset or the number of checks required.

The third contribution is our report of the results of a series of experiments on synthetic and real data. We ran TANE on a larger set of UCI datasets than it has been applied to in the past, and then we ran FD_Mine on the same datasets. The results show that the pruning rules in the FD_Mine algorithm are valuable because they increase the pruning of candidates and reduce the overall amount of checking required to find the same FDs. Among algorithms for this problem, TANE is based on partitions, FUN is based on embedded FD, and FD_Mine emphasizes partitions and equivalences. With its emphasis on equivalences, the FD_Mine approach can also identify redundant data. It might be applied during the data cleaning phase to reduce the size of the database without losing any useful information.

Future work should attempt to discover other data dependencies, such as multivalued dependencies [7] and conditional dependencies. In addition, if too many equivalences are found, especially at higher levels, the FD-Mine algorithm might be slowed down. This potential slowing would occur if the algorithm were to use more time to check the discovered equivalences than would be needed to check for functional dependencies directly. To address this potential problem, a technique for balancing the number of equivalences in comparison to the level might be introduced.

References

1. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. A., and Verkamo, I., Fast Discovery of Association Rules. Fayyad, U. M., Gregory P. S., Smyth, P., Uthurusamy, R. (eds.): In *Advances in Knowledge Discovery and Data Mining*, AAAI/MIT Press, 1996, 307-328.
2. Collopy, E., and Levene, M., Evolving Example Relations to Satisfy Functional Dependencies. *Issues and Applications of Database Technology (IADT 1998*), Berlin, Germany, 1998, 440-447.
3. Dalkilic, M. M., and Robertson, E.L., Information Dependencies. *Symposium on the Principles of Database Systems (PODS 2000)*, Dallas, USA, 2000, 245-253.
4. Flach, P. A., and Savnik, I., Database Dependency Discovery: A Machine Learning Approach. *AI Communications*, 12(3):139-160 (1999).
5. Huhtala, Y., Kärkkäinen, J., Porkka, P., and Toivonen, H., TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computing Journal*, 42(2):100-111 (1999).
6. Lopes, S., Petit, J. M., and Lakhal, L., Efficient Discovery of Functional Dependencies and Armstrong Relations. *International Conference on Extending Database Technology (EDBT 2000)*, Konstanz, Germany, 2000, 350-364.
7. Maier, D., *The Theory of Relational Databases*, Computer Science Press, 1983.
8. Mannila, H., and Räihä, K.J., Algorithms for Inferring Functional Dependencies from Relations. *Data and Knowledge Engineering*, 12(1):83-99 (1994).
9. Mannila, H., and Toivonen, H., Levelwise Search and Borders of Theories in Knowledge Discovery. *Data Mining and Knowledge Discovery*, 1(3):241-258 (1997).
10. Mannila, H., Theoretical Frameworks for Data Mining. *SIGKDD Explorations*, 1(2):30-32 (2000).
11. Novelli, N., and Cicchetti, R., FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. *International Conference on Database Theory (ICDT 2001)*, London, UK, 2001, 189-203.
12. Novelli, N., and Cicchetti, R., Functional and Embedded Dependency Inference: A Data Mining Point of View. *Information Systems*, 26(7):477-506 (2001).
13. Wyss, C., Giannella, C., and Edward, E.L., FastFDs, A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances - Extended Abstract. *Data Warehousing and Knowledge Discovery, Third International Conference (DaWaK 2001)*, Munich, Germany, 2001, 101-110.
14. UCI Machine Learning Repository, http://www1.ics.uci.edu/~mlearn/MLRepository.html