Data Min Knowl Disc (2008) 16:197–219
DOI 10.1007/s10618-007-0083-9

Mining functional dependencies from data

Hong Yao · Howard J. Hamilton

Received: 29 June 2006 / Accepted: 16 August 2007 / Published online: 15 September 2007

Abstract In this paper, we propose an efficient rule discovery algorithm, called FD_Mine, for mining functional dependencies from data. By exploiting Armstrong's Axioms for functional dependencies, we identify equivalences among attributes, which can be used to reduce both the size of the dataset and the number of functional dependencies to be checked. We first describe four effective pruning rules that reduce the size of the search space. In particular, the number of functional dependencies to be checked is reduced by skipping the search for FDs that are logically implied by already discovered FDs. Then, we present the FD_Mine algorithm, which incorporates the four pruning rules into the mining process. We prove the correctness of FD_Mine, that is, we show that the pruning does not lead to the loss of useful information. We report the results of a series of experiments. These experiments show that the proposed algorithm is effective on 15 UCI datasets and synthetic data.

Keywords Discovering functional dependencies · Mining functional dependencies · Implication rule · Functional dependencies · Data mining · Knowledge discovery · FD_Mine · Relational databases

Responsible editor: M. J. Zaki.

H. Yao · H. J. Hamilton (✉)
Department of Computer Science, University of Regina, Regina, SK, Canada S4S 0A2
e-mail: hamilton@cs.uregina.ca

H. Yao
e-mail: yao2hong@cs.uregina.ca

1 Introduction

Rule mining is an important task in data mining. Rule mining can be regarded as an algorithmic process that takes data as input and yields rules, such as association rules (Agrawal et al. 1993), implications (Baixeries 2004; Fagin 1977; Ullman 1982), or functional dependencies (Maier 1983; Ullman 1982) as output. The problem addressed in this paper is to design an efficient rule discovery algorithm for mining functional dependencies from a dataset (Mannila and Raiha 1994). A functional dependency (FD) expresses a value constraint between two sets of attributes in a relation $r(U)$, where U is a finite set of discrete variables (attributes) (Maier 1983). For example, in a student database, students' names are functionally dependent on their student IDs. Formally, a *functional dependency* $X \rightarrow Y$, where $X, Y \subseteq U$, is satisfied by $r(U)$, if for all pairs of tuples $t_i, t_j \in r(U)$, we have: if $t_i[X] = t_j[X]$ then $t_i[Y] = t_j[Y]$.

Two types of rules often discovered from databases are implications and functional dependencies. An *implication* describes a relationship between one specific combination of attribute-value pairs, and a *functional dependency* (FD) describes a relationship between all possible combinations of attribute-value pairs. Implications commonly occur in propositional logic and data mining, since they apply to binary data, whereas functional dependencies commonly occur in database theory, since they can be applied to multiple valued data (Baixeries 2007). Let $I = \{I_1, \ldots, I_i, \ldots, I_n\}$ be a set of attributes of a transaction dataset T and let the domain of each attribute I_i be the binary domain $\{0, 1\}$, where 1 indicates the presence of an item in a transaction, and 0 indicates its absence. Informally, an implication says that whenever the set of attributes X is present (that is: related by the binary value 1) in an object, then, the set of attributes Y is also present (related) in that same object (Baixeries 2007). More precisely, an implication is a rule of the form $X = 1 \Rightarrow Y = 1$, where $X, Y \subset I$, $dom(X) = \{0, 1\}, dom(Y) = \{0, 1\}$, and $X \cap Y = \phi$. For simplicity, the "$= 1$" parts of implications are typically not shown. For example, $\{milk, eggs\} \Rightarrow \{bread\}$ is an implication that says that when milk and eggs are purchased, bread must be purchased as well. In other words, if *milk* and *eggs* both have a value of 1, then *bread* must be 1 as well. On the other hand, a functional dependency (FD) is a rule of the form $X \rightarrow Y$, where X and Y are disjoint attributes, and may not have binary domains. For example, *postcode* $\rightarrow$ *areacode* is an FD that says that the value of the *postcode* attribute determines the value of the *areacode* attribute, regardless of how many possible values both attributes have.

In summary, these two types of rules represent two different granularity levels for the semantics of a rule. An implication $X \Rightarrow Y$ represents the semantics that if the value of X in an object is *true*, then the value of Y in the object is *true* too. A FD $X \rightarrow Y$ represents the semantics that if two objects have the same value on attribute X, then they have the same value on attribute Y. A strong relationship between implications and functional dependencies has been shown in Baixeries (2007), Carpineto et al. (1999), Fagin (1977), Ganter and Wille (1999), Sagiv et al. (1981), Ullman (1982). From a syntactical view, both types of rules are equivalent because they have the same axioms, called

Armstrong's axioms (Maier 1983; Ullman 1982). Suppose each implication $X \Rightarrow Y$ is said to correspond to a FD $X \rightarrow Y$. Given a set of implications P and the corresponding set of functional dependencies Q, the implications in the closure of P correspond one-to-one with the FDs in the closure of Q, because they can both be calculated by recursively applying the same ones of Armstrong's axioms. As a result, the methods used for inferring functional dependencies can be adapted to solve the problem of inferring the corresponding implications. Since implication mining plays an essential role in the theory and practice of rule mining and other data mining tasks, the discovery of FDs can also play an important role in data mining.

It is useful to discover functional dependencies from data. For example, from a database of chemical compounds, it is valuable to discover compounds that are functionally dependent on a certain structure attribute (Huhtala et al. 1999). Functional dependencies also have been applied in database management (Flesca et al. 2005; Maier 1983; Ramakrishnan and Gehrke 2002), and data reverse engineering for relational sources (Tan and Zhao 2004). The discovery of functional dependencies from data has been extensively studied (Baixeries 2004; Flach and Savnik 1999; Huhtala et al. 1999, Lopes et al. 2000; Novelli and Cicchetti 2001a, b; Wyss et al. 2001; Yao et al. 2002). In this paper, we want to improve on previous proposed methods for this problem by incorporating more effective pruning rules into the functional dependency mining process.

The proposed pruning rules reduce the size of the search space for mining FDs from data and allows us to advance in the study of faster and more efficient algorithms for mining rules.

In this paper, we describe the FD_Mine algorithm for discovering functional dependencies from data. A preliminary description of the FD_Mine algorithm has been published (Yao et al. 2002). The set of FDs can be divided into two parts: FDs that can be inferred from already discovered FDs using Armstrong's Axioms (Maier 1983), and those that cannot. The FD_Mine algorithm will examine the dataset to find the second type of FDs because the first type of FDs are regarded as redundant. Since the process of checking FDs against a dataset is relatively costly, we attempt to prune redundant FDs whenever possible. More specifically, we first describe four effective pruning rules that reduce the number of functional dependencies to be checked by skipping the search for FDs that are logically implied by the FDs already discovered. Then, we present the FD_Mine algorithm, which incorporates these four pruning rules into the mining process. We prove the correctness of FD_Mine, that is, we show that the pruning does not lead to the loss of useful information. Experimental results show that the proposed algorithms are effective on 15 UCI datasets (UCI Repository of machine learning databases 2005) and synthetic data.

The advantage of FD_Mine is that the number of functional dependencies to be checked against the dataset is reduced due to identifying pruning rules among attributes that are not utilized in previous approaches. For example, if the FDs $X \rightarrow Y$ and $Y \rightarrow X$ are discovered, then no further candidates containing Y need to be considered, since attributes X and Y are equivalent. A *candidate* is a combination of attributes over a dataset. Thus, FD_Mine avoids searching for

functional dependencies that are logically implied by the functional dependencies already discovered. Pruning equivalent attributes is valuable, because the number of candidates increases exponentially with the number of attributes. The other advantage is that all pruning rules can be identified and integrated in a pruning strategy using only nontrivial closure, which improves the performance of the algorithm, because it reduces the overall amount of checking required to discover FDs from data.

The remainder of this paper is organized in six sections. In Sect. 2, related work is described. Background knowledge concerning functional dependencies is presented in Sect. 3. In Sect. 4, the theoretical foundations of functional dependencies are analyzed. Four pruning rules are presented. Section 5 describes algorithms for finding functional dependencies from data by incorporating these pruning rules into the data mining process. We prove the correctness of FD_Mine and show how it relates to previous approaches. In Sect. 6, we report the results of a series of experiments. Finally, in Sect. 7, we summarize the paper.

2 Related work

Given a finite set of discrete variables (attributes) $U = \{v_1, v_2, \ldots, v_m\}$ and a database D on U, any combination X of the variables (attributes) in U can be the left side of a FD $X \rightarrow Y$ such that $X \rightarrow Y$ is satisfied by D. The possible values of X form the search space of the functional dependencies mining problem. For example, the semi-lattice illustrated in Fig. 1, excluding the top level, shows the search space of an exhaustive algorithm for finding FDs for five attributes. Figure 1 shows all possible nonempty combinations of the five attributes A, B, C, D, and E. For these attributes, there are $2^n = 2^5 = 32$ possible subsets of attributes, of which the $2^n - 2 = 30$ nonempty, proper subsets are the candidates. The levels of the semi-lattice are numbered from the bottom to the top. The set $U = \{A, B, C, D, E\}$ at level 5 is not a candidate, because for any FD with the form $U \rightarrow v_i$, we have $v_i = U - U = \phi$. There are $n2^{n-1}$ edges in a full lattice for n attributes (Goodaire and Parmenter 1992). Since the semi-lattice of the total search space of FDs starts from level 1, rather than the empty set, there are $n2^{n-1} - n$ edges in the semi-lattice of the complete search space for FDs.

The size of the search space is exponential to the number of variables in U. One main issue in the discovery of functional dependencies is to prune the search space as much as possible. Existing algorithms can be classified into three categories: the candidate generate-and-test approach (Huhtala et al. 1999; Novelli and Cicchetti 2001a, b), the minimal cover approach (Flach and Savnik 1999; Lopes et al. 2000; Wyss et al. 2001), and the formal concept analysis approach (Baixeries 2004; Lopes et al. 2002). The candidate generate-and-test approach uses levelwise search to explore the search space. It reduces the search space by eliminating candidates using pruning rules. TANE (Huhtala et al. 1999) and FUN (Novelli and Cicchetti 2001a, b) both are levelwise methods. They

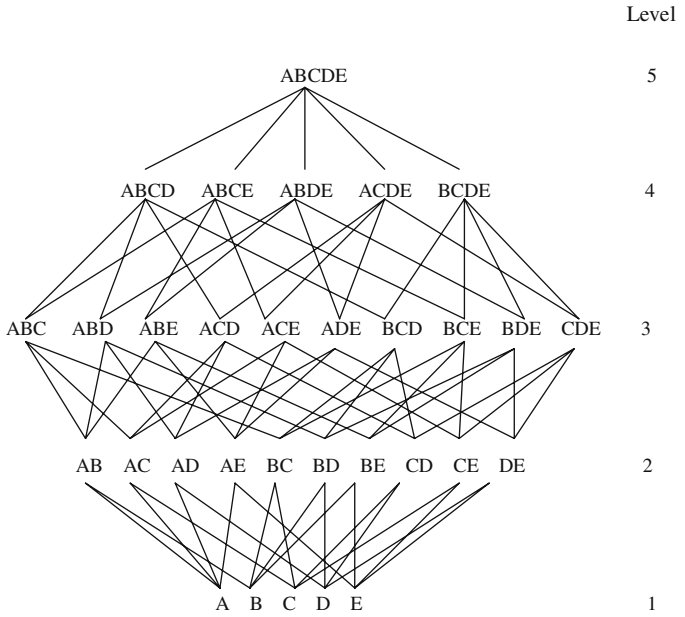

Fig. 1 All possible nonempty combinations of attributes A, B, C, D, and E

begin by testing FDs with small left-hand sides and prune the search space as soon as possible. More specifically, both methods are based on partitioning the set of tuples with respect to their attribute values. Using partitions, TANE and FUN can test the validity of FDs efficiently even for large numbers of tuples. They search the set containment lattice in a levelwise manner. By computing closure of candidates in level k, the FDs in this level are discovered, and results from level k are used to generate candidates in level $k + 1$. The difference between the TANE and FUN algorithms is that they use different pruning rules to eliminate candidates. The minimal cover approach discovers the minimal cover of the set of FDs given a database. FDEP (Flach and Savnik 1999) consists of three algorithms: the bottom-up algorithm, the bi-directional algorithm and the top-down algorithm. Experiments showed the bottom-up method to be the most efficient. The bottom-up method first computes the *negative cover*, which is a cover for all dependencies that are inconsistent with the data set. In particular, the negative cover contains the least general such dependencies. In the second step, the method iterates over these dependencies. For each dependency, the specializations, which are satisfied by the data set, are accumulated. The accumulated result gives the *positive cover*, from which the FDs are obtained. FastFDs (Wyss et al. 2001) and Dep-Miner (Lopes et al. 2000) discover FDs by considering pairs of tuples, i.e. agree sets. First, a stripped partition database is extracted from the initial relation. Then, using such partitions, agree sets are computed and maximal sets are generated. Thus, a minimum FD cover according to these maximal sets is found. FastFDs differs

from Dep-Miner only in that Dep-Miner employs a levelwise search, whereas FastFDs use a first-depth search strategy. The formal concept analysis approach discovers functional dependencies from the view of formal concept analysis. By considering the relationship between relational database theory and formal concept analysis (Demetrovics et al. 1992), the functional dependencies that hold in a database can be extracted by using a predefined formal concept analysis closure operator. In Lopes et al. (2002), the authors conclude that the qualitative comparison between DepMiner (or FastFDs) and TANE (or FUN) is difficult because the approaches widely differ. The drawback of the former is the time-consuming computation of agree sets since it is quadratic with respect to the number of tuples in the relation. The drawback of the latter is their heavy manipulations of attribute sets and the numerous tests which have to be performed. Our FD_Mine approach belongs to generate-and-test approach. FD_Mine differs from TANE or FUN in that more effective pruning rules are identified such that a faster and more efficient algorithm is designed for mining FDs from data.

3 Functional dependencies

Let $U = \{v_1, v_2, \ldots, v_m\}$ be a finite set of discrete variables called *attributes* (Maier 1983). Each attribute v_i has a finite domain, denoted $dom(v_i)$, representing the values that v_i can take on. For a subset $X = \{v_i, \ldots, v_j\}$ of U, we write $dom(X)$ for the Cartesian product of the domains of the individual attributes in X, namely, $dom(X) = dom(v_i) \times \cdots \times dom(v_j)$. A *relation* r (Maier 1983) on U, written $r(U)$, is a finite set of mappings $\{t_1, \ldots, t_n\}$ from U to $dom(U)$ with the restriction that for each mapping $t \in r(U)$, $t[v_i]$ must be in $dom(v_i)$, $1 \leq i \leq m$, where $t[v_i]$ denotes the value obtained by restricting the mapping t to v_i. A relation r on $U = \{v_1, v_2, \ldots, v_m\}$ is depicted in Table 1. Each mapping t is called a *tuple* (Maier 1983) and $t(v_i)$ is called the v_i-value of t. To simplify notation, we may write the singleton set $\{v_i\}$ as the single attribute v_i and the set of attributes $\{v_1, \ldots, v_j\}$ as $v_1 \ldots v_j$. The union $X \cup Y$ of sets of two attributes X and Y is sometimes simply denoted as XY.

Definition 1 (Huhtala et al. 1999) Let $r(U)$ be a relation and $X, Y \subseteq U$. A *functional dependency* (*FD*) is a constraint, denoted $X \rightarrow Y$. The FD $X \rightarrow Y$ is satisfied by $r(U)$ if every two tuples $t_i, t_j \in r(U)$ that have $t_i[X] = t_j[X]$ also have $t_i[Y] = t_j[Y]$. In an FD $X \rightarrow Y$, we refer to X as the *antecedent* and Y as the *consequent*.

Table 1 A relation r on $U = \{v_1, \ldots, v_m\}$

	$\mathbf{v_1}$	$\mathbf{v_2}$	$\cdots$	$\mathbf{v_m}$
	$t_1(v_1)$	$t_1(v_2)$	$\cdots$	$t_1(v_m)$
$r =$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$t_n(v_1)$	$t_n(v_2)$	$\cdots$	$t_n(v_m)$

Example 1 Consider the example relation $r(U)$ shown in Table 2. By Definition 1, the FDs $A \to D, D \to A, AB \to E, BD \to E, BE \to A, BE \to D, CE \to A$, and $CE \to D$ are satisfied by $r(U)$. However, the FD $A \to B$ is not satisfied by $r(U)$, since for tuples $t_1, t_2 \in r(U)$, $t_1(A) = t_2(A)$ but $t_1(B) \neq t_2(B)$.

Armstrong's Axioms (Maier 1983) are the following three inference axioms for FDs defined on sets of attributes X, Y, and Z.

F1. (Reflexivity) If $Y \subseteq X$, then $X \to Y$.
F2. (Augmentation) If $X \to Y$, then $XZ \to YZ$.
F3. (Transitivity) If $X \to Y$ and $Y \to Z$, then $X \to Z$.

Theorem 1 (Maier 1983) *The inference axioms F1, F2, and F3 are sound and complete.*

Soundness indicates that given a set F of FDs satisfying by a relation $r(U)$, any FD inferred from F using F1–F3 is satisfied by $r(U)$ too. Completeness indicates that the three axioms F1–F3 can be applied repeatedly to infer all FDs logically implied by a set F of FDs (Maier 1983). The following two inference axioms can be derived from Armstrong's Axioms and are introduced for convenience (Ramakrishnan and Gehrke 2002).

Union If $X \to Y$ and $X \to Z$, then $X \to YZ$.
Decomposition If $X \to YZ$, then $X \to Y$ and $X \to Z$.

By Decomposition, $X \to YZ$ is sometimes written as $X \to v_i, \forall v_i \in YZ$.

Definition 2 (Ramakrishnan and Gehrke 2002) Let $X, Y \subseteq U$ and F be a set of FDs. The *closure* of X ($X \neq \emptyset$) w.r.t. F, denoted X^+, is defined as $\{Y | X \to Y$ can be deduced from F using Armstrong's Axioms$\}$. The closure of F, denoted F^+, is the set of all FDs that can be deduced from F using Armstrong's Axioms.

Definition 2 indicates that the FD $X \to Y$ holds if and only if $Y \in X^+$. For $X, Y \subseteq U$, we use $(XY)^+$ to denote the closure of $X \cup Y$.

Example 2 Let $F = \{A \to C, BD \to AC\}$. By Definition 2, $A^+ = \{A, C\}$ and $(BD)^+ = \{A, B, C, D\}$.

Table 2 An example relation

TID	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

Now, we introduce partitions of X and XY. These partitions can be used to check whether or not an FD $X \rightarrow Y$ is satisfied by a relation.

Definition 3 Let $X \subseteq U$ and let $t_1, \ldots, t_n$ be all the tuples in a relation $r(U)$. The *partition* over X, denoted $\prod_X$, is a set of the groups such that t_i and t_j, $1 \leq i, j \leq n, i \neq j$, are in the same group if and only if $t_i[X] = t_j[X]$. The number of the groups in a partition is called the *cardinality* of the partition, denoted $|\prod_X|$.

For a single attribute v_i, we use $\prod_{Xv_i}$ to denote the partition of the set of attributes $X \cup \{v_i\}$.

Example 3 In Table 2, by Definition 3, $\prod_A = \{\{t_1, t_2, t_3, t_4, t_7\}, \{t_5, t_6\}\}$, and $\prod_{CE} = \{\{t_1, t_2\}, \{t_3\}, \{t_4, t_7\}, \{t_5\}, \{t_6\}\}$. By Definition 3, $|\prod_A| = 2$ and $|\prod_{CE}| = 5$.

Using the cardinality of the partition, we can check whether or not an FD $X \rightarrow Y$ is satisfied by a relation using Theorem 2, which was suggested by Huhtala et al. (Huhtala et al. 1999).

Theorem 2 (Huhtala et al. 1999) *An FD $X \rightarrow Y$ is satisfied by a relation $r(U)$ if and only if $|\prod_X| = |\prod_{XY}|$.*

For example, in Table 2, FD $CE \rightarrow A$ is satisfied by $r(U)$, since $|\prod_{CE}| = |\prod_{ACE}| = 5$.

4 Theoretical foundation

In this section, we give the theoretical foundation for our approach to finding functional dependencies in a relation. We introduce the concepts of equivalent attributes and nontrivial closure and provide proofs of related lemmas and theorems.

4.1 Equivalent attributes

Definition 4 Let $X, Y \subseteq U$. If $X \rightarrow Y$ and $Y \rightarrow X$, then X and Y are said to be *equivalent sets of attributes*, denoted by the *equivalence* $X \leftrightarrow Y$.

The following theorem shows that we can check whether or not the equivalence $X \leftrightarrow Y$ is satisfied in a relation U by comparing the closures of X and Y.

Theorem 3 *Let $X, Y \subseteq U$. If $Y \subseteq X^+$ and $X \subseteq Y^+$, then $X \leftrightarrow Y$.*

Proof Since $X \rightarrow X^+$ and $Y \subseteq X^+$, then by Decomposition, $X \rightarrow Y$ holds. By a similar argument, $Y \rightarrow X$ holds. As $X \rightarrow Y$ and $Y \rightarrow X$, we have $X \leftrightarrow Y$. □

Example 4 Let $U = \{A, B, C, D\}$ and $F = \{BD \rightarrow AC, CD \rightarrow B\}$. By Definition 2, $(BD)^+ = \{A, B, C, D\}$ and $(CD)^+ = \{A, B, C, D\}$. Since $CD \subseteq (BD)^+$ and $BD \subseteq (CD)^+$, we have $BD \leftrightarrow CD$.

Using equivalence $X \leftrightarrow Y$ and Armstrong's Axioms, Lemmas 1 and 2 are obtained.

Lemma 1 *Let $W, X, Y, Y', Z \subseteq U$ and $Y \subset Y'$. If $X \leftrightarrow Y$ and $XW \rightarrow Z$, then $Y'W \rightarrow Z$.*

Proof By $X \leftrightarrow Y$, we have $Y \rightarrow X$. By Augmentation, $YW \rightarrow XW$. By Transitivity, $YW \rightarrow XW$ and $XW \rightarrow Z$ give $YW \rightarrow Z$. By Augmentation, $Y' - Y$ can be added to both sides of $YW \rightarrow Z$ giving $YW(Y' - Y) \rightarrow Z(Y' - Y)$. By $Y \subset Y'$, $Y'W \rightarrow Z(Y' - Y)$. By Decomposition, $Y'W \rightarrow Z$. □

Lemma 2 *Let $W, X, Y, Z \subseteq U$. If $X \leftrightarrow Y$ and $WZ \rightarrow X$, then $WZ \rightarrow Y$.*

Proof By $X \leftrightarrow Y$, we have $X \rightarrow Y$. By Transitivity, $WZ \rightarrow X$ and $X \rightarrow Y$ give $WZ \rightarrow Y$. □

Definition 5 Let F be a set of FDs and X^+ be the closure of X w.r.t. F. The *nontrivial closure* of X w.r.t. F, denoted X^*, is defined as $X^* = X^+ - \{X\}$.

For $X, Y \subseteq U$, we use $(XY)^*$ to denote the nontrivial closure of the set of attributes $X \cup Y$, similarly to how we use $(XY)^+$ to denote the closure of attributes $X \cup Y$. We have $(XY)^+ = (XY)^* \cup XY$.

Example 5 Recalling Example 2, we have $A^+ = \{A, C\}$ and $(BD)^+ = \{A, B, C, D\}$. By Definition 5, $A^* = \{C\}$ and $(BD)^* = \{A, C\}$.

Theorem 4 *Let $X, Y, Z \subseteq U$ such that $Y^+ \subseteq X$. Then $X - Y^* \rightarrow Z$ if and only if $X \rightarrow Z$.*

Proof We first prove that if $X - Y^* \rightarrow Z$ then $X \rightarrow Z$. By Augmentation, Y^* can be added to both sides of $X - Y^* \rightarrow Z$ giving $X \rightarrow Y^*Z$. By Decomposition, $X \rightarrow Z$. Now we prove that if $X \rightarrow Z$ then $X - Y^* \rightarrow Z$. Since $Y = Y^+ - Y^*$ and $Y^+ \subseteq X$, then $Y \subseteq X - Y^*$. By Reflexivity, $X - Y^* \rightarrow Y$. By Transitivity, $X - Y^* \rightarrow Y$ and $Y \rightarrow Y^*$ give $X - Y^* \rightarrow Y^*$. By Union, $X - Y^* \rightarrow Y^*$ and $X - Y^* \rightarrow X - Y^*$ give $X - Y^* \rightarrow X$. Thus, by Transitivity, $X - Y^* \rightarrow X$ and $X \rightarrow Z$ give $X - Y^* \rightarrow Z$. □

Example 6 Let $U = \{A, B, C, D, E\}$ such that $C \rightarrow A$ is satisfied by $r(U)$. Suppose $X = ACE$, $Y = C$, and $Z = BD$. Since $C \rightarrow A$, $C^* = A$. Then $Y^* = A$ and $Y^+ = C \cup C^* = AC$. Thus, $Y^+ \subseteq X$ and $X - Y^* = CE$. By Theorem 4, if $CE \rightarrow BD$ is satisfied by $r(U)$, then $ACE \rightarrow BD$ is also satisfied by $r(U)$; if $CE \rightarrow BD$ is not satisfied, then $ACE \rightarrow BD$ is not satisfied either.

Theorem 5 *Let $X, Y \subseteq U$. Then $X^* \cup Y^* \subseteq (XY)^+$.*

Proof By Definition 2, $X \rightarrow X^*$ is satisfied by $r(U)$. By Augmentation, Y can be added to both sides of $X \rightarrow X^*$ giving $XY \rightarrow X^*Y$. By Decomposition, $XY \rightarrow X^*$, which indicates that $X^* \subseteq (XY)^+$. It can be similarly shown that $Y^* \subseteq (XY)^+$. Thus, $X^* \cup Y^* \subseteq (XY)^+$. □

Example 7 Let $U = \{v_1, v_2, v_3, v_4, v_5\}$, $v_1^* = \{v_3\}$ and $v_2^* = \{v_4\}$. By Theorem 5, $\{v_3, v_4\} \subseteq (v_1v_2)^+$. That is, $v_1v_2 \rightarrow v_3v_4$. By Decomposition, $v_1v_2 \rightarrow v_3$ and $v_1v_2 \rightarrow v_4$ can also be inferred.

Theorem 5 indicates that when checking all FDs of the form $XY \rightarrow v_i$, where $v_i \in U - XY$, only the FDs of the form $XY \rightarrow v_i$, where $v_i \in U - X^+Y^+$, need to be checked, since $X^* \cup Y^* \subseteq (XY)^+$. As a result, the right side $U - XY$ of FD $XY \rightarrow v_i$, where $v_i \in U - XY$, can be reduced to $U - X^+Y^+$. That is, the attributes X^+Y^+ can be pruned from the right side $U - XY$ of the FD $XY \rightarrow v_i$, where $v_i \in U - XY$.

Lemma 3 *Let $X \subset S \subseteq U$. If $X \rightarrow U - X$, then $S \rightarrow U - S$.*

Proof By Augmentation, $S - X$ can be added to both sides of $X \rightarrow U - X$, giving $X(S - X) \rightarrow (U - X)(S - X)$. As $X \subset S \subseteq U$, we have $S \rightarrow U - X$ and $U - S \subset U - X$. By Decomposition, $S \rightarrow U - S$. □

Lemma 3 indicates that if X functionally determines all attributes in U other than X, then all supersets of X also functionally determine all attributes in U other than X.

If a relation $r(U)$ satisfies the FD $X \rightarrow Y$, but not $X' \rightarrow Y$ for any $X' \subset X$, then $X \rightarrow Y$ is called *left-reduced* . Thus, Lemmas 1, 2, and 3 and Theorems 4 and 5 indicate that if $X \rightarrow Y$ is a left-reduced FD, then all supersets of X also functionally determine Y.

4.2 Pruning rules

A simple approach to finding all functional dependencies that are satisfied by a relation is to propose the set of possible candidates for the antecedent and check them one by one with possible consequents. A *candidate* $X \subset U$ is a nonempty set of attributes being evaluated for the antecedent of functional dependencies.

To find the candidates, we generate all possible candidates at level k from the candidates at level $k - 1$. For instance, the candidate AB at level 2 is generated from the candidates A and B at level 1. Overall, given a candidate X, to find the FDs with X as antecedent that are satisfied by $r(U)$, we will check whether or not X functionally determines each of the remaining attributes in U, i.e., we will check whether or not $X \rightarrow v_i$ is satisfied by $r(U)$ for each $v_i \in U - X$. Using Union, $X \rightarrow Y$ can be inferred if $X \rightarrow v_i$, $\forall v_i \in Y$. For example, let $U = \{A, B, C, D, E\}$ and AE be a candidate. Then, $U - \{A, E\} = \{B, C, D\}$, which indicates that for candidate AE only the FDs $AE \rightarrow B$, $AE \rightarrow C$, and $AE \rightarrow D$ need to be checked. For the semi-lattice shown in Fig. 1, the number of FDs that could be checked is 75, because $n2^{n-1} = 5 \cdot 2^{5-1} - 5 = 75$. However, by designing useful pruning rules, the number of FDs to be checked can be reduced since some FDs can be inferred from discovered FDs without checking them on data.

To check whether or not an FD $X \rightarrow v_i$ is satisfied by a relation $r(U)$, we apply Theorem 2 by comparing the partitions of X and $X \cup v_i$. For example, according to Theorem 2, $AE \rightarrow B$ is satisfied by $r(U)$ if $|\prod_{AE}| = |\prod_{ABE}|$. The FDs that are not checked by FD_Mine are redundant ones that can be inferred from the discovered FDs.

In general, four kinds of FDs can be inferred. First, using equivalence $X \leftrightarrow Y$, any FD $S \rightarrow v_i$, where $Y \subset S$, can be inferred from $X \rightarrow v_i$ by Lemmas 1 and 2. Second, given $Y^+ \subseteq X$, using nontrivial closure Y^*, any FDs $X \rightarrow v_i$, where $v_i \in U - X$, can be inferred from $X - Y^* \rightarrow v_i$ by Theorem 4. Third, given X^+, any FD $S \rightarrow v_i$, where $v_i \in X$ and $X \subset S$, can be inferred from $X \rightarrow v_i$ by Theorem 5. Fourth, given $X \subset S$, any FD $S \rightarrow v_i$, where $v_i \in U - S$, can be inferred from $X \rightarrow v_i$ by Lemma 3. Thus, all these kinds of FDs can be discovered without checking whether or not they are satisfied by $r(U)$.

To simplify our discussion, when considering equivalent attributes X and Y, i.e., $X \leftrightarrow Y$, we assume that the equivalence is written such that the set of attributes X is generated earlier than the set of attributes Y.

Lemmas 1 and 2 indicate that after an equivalence $X \leftrightarrow Y$ has been found, no further sets of attributes containing Y need to be checked.

Example 8 Suppose that we are given the relation $r(U)$ shown in Table 3(a). By examining all entries for attributes A and D, we determine that FDs $A \rightarrow D$ and $D \rightarrow A$ are satisfied by $r(U)$. According to Definition 4, $A \leftrightarrow D$ is satisfied by $r(U)$. By checking attributes A, B, and C in all tuples in $r(U)$, we determine that $AB \rightarrow C$ and $BC \rightarrow A$ are satisfied by $r(U)$. Without Lemmas 1 and 2, we would then need to check all tuples in Table 3(a) again to determine whether or not $BD \rightarrow C$ and $BC \rightarrow D$ are satisfied by $r(U)$. However, with Lemma 1, $BD \rightarrow C$ can be inferred from $A \leftrightarrow D$ and $AB \rightarrow C$. By Lemma 2, $BC \rightarrow D$ can also be inferred from $A \leftrightarrow D$ and $BC \rightarrow A$. Thus, using Lemmas 1 and 2, no further candidate FDs containing D need to be checked. If relation $r(U)$ is only used for discovering functional dependencies, it may make sense to remove attribute D from the relation $r(U)$. The resulting relation, after removing attribute D, is shown in Table 3(b). Even if the attribute is not removed from the relation, it can be ignored in all subsequent processing.

In the context of finding all FDs satisfied by a relation $r(U)$, the following four pruning rules are defined for candidates X and Y, where $X \neq \phi, Y \neq \phi$, and $X, Y \subset U$.

Pruning rule 1 If $X \leftrightarrow Y$ is satisfied by $r(U)$, then candidate Y can be deleted.
Pruning rule 2 If $Y^+ \subseteq X$, then candidate X can be deleted.
Pruning rule 3 Given X^* and Y^*, then when attempting to determine whether or not the set of FDs $XY \rightarrow v_i$, where $v_i \in U - XY$, is satisfied by $r(U)$, only the the set of FDs $XY \rightarrow v_i$, where $v_i \in U - X^+Y^+$, needs to be checked in $r(U)$.
Pruning rule 4 If $\forall v_i \in U - X$, $X \rightarrow v_i$ is satisfied by $r(U)$, then candidate X can be deleted.

Table 3 Effect of removing a redundant attribute D

(a) Original relation

TID	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

(b) Relation without attribute D

TID	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	0

These four pruning rules are used to delete candidates at level $k-1$ before generating candidates at level k. Pruning rule 1 is justified by Lemmas 1 and 2. This rule reduces the size of the search space by eliminating redundant candidates. Pruning rule 2 is justified by Theorem 4. This rule indicates that $X \rightarrow Z$ can be inferred from $X - Y^* \rightarrow Z$. Pruning rule 3 is justified by Theorem 5. This rule indicates that for a candidate X and its superset S, $S \rightarrow X^*$ can be inferred given X^*. Thus, for candidate S, we need to check only the FDs $S \rightarrow v_i, v_i \in U - S - X^*$. Pruning rule 4 is justified by Lemma 3. This rule indicates that for any superset S of X, $S \rightarrow U - S$ can be inferred.

5 The FD_Mine algorithm

In this section, the FD_Mine algorithm for finding FDs in data is described. FD_Mine combines a level-wise search (Mannila and Toivonen 1997) and the four pruning rules given in Sect. 3.2. In a level-wise search, results from level k are used to explore level $k+1$. The FD_Mine algorithm is shown in Fig. 2. First, at level 1, the singleton candidates in U are stored in C_1. At level 2, each candidate v_i of C_1 is used to generate all candidates of the form v_iv_j, where $v_j \in C_1$ and $v_i \neq v_j$, which are stored in C_2. Then all FDs of the form $v_i \rightarrow U - v_i$ are found and stored in F. At level 3, the candidate set C_2 is used to generate the three attribute candidates, which are stored in C_3. All FDs of the form $v_iv_j \rightarrow U - v_iv_j$ are checked and added to F. This procedure is repeated until $C_k = \emptyset$. In addition to storing a set of FDs in F, this algorithm also stores equivalent candidates in E.

In more detail, for each level, the FD_Mine algorithm works as follows. First, in line 6, k is incremented. All candidates at level k are generated in line 7, then

FD Mine$(r(U))$
Input: A relation $r(U)$ over $U = \{v_1, \ldots, v_m\}$
Output: A set F of functional dependencies over $r(U)$.
1. $F = \emptyset$; $E = \emptyset$; $C_1 = U$; $k = 1$;
2. $C_k = CalculatePartition(C_k, r(U))$;
3. $C_k = InitialClosure(C_k)$;
4. **while** $|C_k| > 0$ **do**
5. {
6. $k = k + 1$;
7. $C_k = Apriori\text{-}Gen(C_{k-1})$;
8. $C_k = CalculatePartition(C_k, r(U))$;
9. $C_k = InitialClosure(C_k)$;
10. $F = F \cup ObtainFDs(C_{k-1})$;
11. $E = E \cup ObtainEquivalences(C_{k-1}, F)$;
12. $C_k = Prune(C_{k-1}, C_k, E)$;
13. }
14. **return** (F);

Fig. 2 The FD_Mine algorithm

the partitions of each candidate at level k are calculated in line 8 and their nontrivial closures are initialized to the empty set in line 9. In line 10, for each $X \in C_{k-1}$, all FDs of the form $X \to v_i$ are checked. The equivalent candidates are discovered in line 11. In line 12, candidates may be deleted from C_k by using the four pruning rules given in Sect. 3.2.

The algorithms called by the FD_Mine algorithm are *CalculatePartition*, *InitializeClosure*, *Apriori-Gen*, *ObtainFDs*, *ObtainEquivalences*, and *Prune*. The *CalculatePartition* algorithm accesses the relation to calculate the actual partition of each candidate X in C_k. We refer the reader to Huhtala et al. (1999) for a thorough discussion of this algorithm. The *Apriori-Gen* algorithm generates all possible candidates in C_k from the candidates in C_{k-1}. For example, given $C_2 = \{AB, BC, AC\}$, by *Apriori-Gen*, $C_3 = \{ABC\}$. More details concerning the *Apriori-Gen* algorithm can be found in (Agrawal et al. 1993).

The *InitializeClosure* algorithm, which is shown in Fig. 3, initializes the nontrivial closure of X to the empty set. It is assumed that each X has a *nontrivial closure* field, denoted X^*, for storing the nontrivial closure of X.

The *ObtainFDs* algorithm, which is shown in Fig. 4, checks the FDs of each candidate X in C_k. More precisely, FDs of the form $X \to v_i$, where $v_i \in U - X^+$, are checked by comparing partitions $\prod_X$ and $\prod_{Xv_i}$. Theorem 2 guarantees the correctness of this method. It is assumed that each candidate X has a *partition* field, denoted $\prod_X$, for storing the partition of X.

InitializeClosure(C_k)
1. **for** each candidate X in C_k
2. $X^* = \emptyset$;
3. **return** (C_k);

Fig. 3 The *InitializeClosure* algorithm of FD_Mine

```
ObtainFDs(C_{k-1})
1.   F = ∅;
2.   for each candidate X in C_{k-1}
3.      for each v_i ∈ U − X^+                    // Pruning rule 3
4.         if (| ∏_X | == | ∏_{X v_i} |) then
5.         {
6.            X* = X* ∪ {v_i};
7.            F = F ∪ {X → v_i};      //by Theorem 2
8.         }
9.   return (F);
```

Fig. 4 The *ObtainFDs* algorithm of FD_Mine

```
ObtainEquivalences(C_{k-1}, F)
1.   E = ∅;
2.   for each candidate X in C_{k-1}
3.      for each Y → v_i ∈ F
4.         if (X ⊆ Y^+ and Y ⊆ X^+ ) then
5.            E = E ∪ {X ↔ Y};      //by Theorem 3
6.   return (E);
```

Fig. 5 The *ObtainEquivalences* algorithm of FD_Mine

```
Prune(C_{k-1}, C_k, E)
1.      for each S ∈ C_k
2.         for each X ∈ C_{k-1}
3.            if (X ⊂ S) then
4.            {
5.               if (X ∈ {Z | Y ↔ Z ∈ E}) then
6.               {
7.                  delete S from C_k;        // Pruning rule 1
8.                  break;
9.               }
10.              if (X* ⊂ S) then
11.              {
12.                 delete S from C_k;        // Pruning rule 2
13.                 break;
14.              }
15.              S* = S* ∪ X*;                // Pruning rule 3
16.              if (U == S ∪ S*) then
17.              {
18.                 delete S from C_k;        // Pruning rule 4
19.                 break;
20.              }
21.           }
22.     return (C_k);
```

Fig. 6 The *Prune* algorithm of FD_Mine

The *ObtainEquivalences* algorithm, which is shown in Fig. 5, uses Theorem 3 to obtain equivalent candidates from the discovered FDs.

The *Prune* algorithm, which is shown in Fig. 6, exploits the four pruning rules given in Sect. 3.2 to reduce the size of search space.

We use the relation in Table 2 to provide an example of how FD_Mine works.

Example 9 At level 1, C_1 is set to $C_1 = \{A, B, C, D, E\}$ and the partition of each candidate in C_1 is computed by *CalculatePartition*. For example, the partition of A is $\prod_A = \{\{t_1, t_2, t_3, t_4, t_7\}, \{t_5, t_6\}\}$. At level 2, $C_2 = \{AB, AC, AD, AE, BC, BD,$

$BE, CD, CE, DE\}$ is generated by *Apriori-Gen* and the partition of each candidate in C_2 is computed by *CalculatePartition*. For example, the partition of AB is $\prod_{AB} = \{\{t_1, t_7\}, \{t_2\}, \{t_3\}, \{t_4\}, \{t_5\}, \{t_6\}\}$. Since $|\prod_A| = |\prod_D| = |\prod_{AD}| = 2$, $A^* = D$ and $D^* = A$ are obtained in line 6 of *ObtainFDs*, and FDs $A \rightarrow D$ and $D \rightarrow A$ are discovered in line 7 of *ObtainFDs*. Using *ObtainEquivalences*, the equivalence $A \leftrightarrow D$ is deduced. In the *Prune* algorithm, since $A \leftrightarrow D$ holds, then by pruning rule 1, all candidates that include D, namely, AD, BD, CD, and DE, are removed from C_2, i.e., $C_2 = \{AB, AC, AE, BC, BE, CE\}$. At level 3, $C_3 = \{ABC, ABE, ACE, BCE\}$ is generated by *Apriori-Gen* and the partition of each candidate in C_3 is computed by *CalculatePartition*. Since $|\prod_{AB}| = |\prod_{ABE}| = 6$, $|\prod_{BE}| = |\prod_{ABE}| = 6$, and $|\prod_{CE}| = |\prod_{ACE}| = 6$, *ObtainFDs* adds candidate E to $(AB)^*$ and adds candidate A to $(BE)^*$ and $(CE)^*$. *ObtainFDs* also discovers FDs $AB \rightarrow E$, $BE \rightarrow A$, and $CE \rightarrow A$, and adds them to F. Using *ObtainEquivalences*, the equivalence $AB \leftrightarrow BE$ is deduced from $AB \rightarrow E$ and $BE \rightarrow A$. In the *Prune* algorithm, since $AB \leftrightarrow BE$ is satisfied by $r(U)$, then by pruning rule 1, ABE, and BCE are removed from C_3, i.e., $C_3 = \{ABC, ACE\}$. Since $CE \rightarrow A$ is satisfied by $r(U)$, $(CE)^+ = ACE$. By pruning rule 2, ACE is removed from C_3, i.e., $C_3 = \{ABC\}$. Since $(AB)^* = \{D, E\}$, $(AE)^* = \{D\}$, and $(BE)^* = \emptyset$, then, in line 15 of the *Prune* algorithm, $(ABC)^* = \{D, E\}$, which means $(ABC)^+ = U$. Thus, by pruning rule 4, ABC is removed from C_3 in line 18 of the *Prune* algorithm, i.e., $C_3 = \emptyset$. As no other candidates remain when the *Prune* algorithm finishes, FD_Mine halts.

The portion of the semi-lattice illustrated in Fig. 7 shows the search space for FD_Mine when applied to the relation shown in Table 2. Each node represents a combination of candidates. If an edge is shown between nodes X and $X \cup v_i$, then FD $X \rightarrow v_i$ needs to be checked. A candidate is shown in bold face if its partition needs to be computed by accessing the relation. Hence, the number of edges is the number of FDs to be checked, and the number of candidates in bold face is the number of partitions to be computed by accessing the relation. Fig. 7 shows that FD_Mine checks 32 FDs and computes 19 partitions of candidates, while the semi-lattice shown in Fig. 1 corresponds to checking 75 FDs and computing 31 partitions.

The crucial point about Example 9 is that equivalences are discovered and used as the basis for pruning. First, the discovery at level 2 that FDs $A \rightarrow D$ and $D \rightarrow A$ are satisfied indicates that $A \leftrightarrow D$ is also satisfied. Thus, all candidates involving D at levels 3 and higher do not need to be checked, as reflected in Fig. 7. Second, the discovery at level 3 that FDs $AB \rightarrow E$ and $BE \rightarrow A$ are satisfied indicates that $AB \leftrightarrow BE$ is also satisfied. Hence, all candidates that include BE do not need to be checked at level 4. As shown in Fig. 7, all FDs in the sample relation have already been discovered after checking level 3. In this case, FD_Mine saves a considerable fraction of the computation compared to searching the semi-lattice shown in Fig. 1.

Theorem 6 *Let F be the set of FDs that are obtained by applying FD_Mine to a relation $r(U)$ and F^+ be the closure of F. Then, an FD $X \rightarrow v_i$ is satisfied by $r(U)$ if and only if $X \rightarrow v_i \in F^+$.*

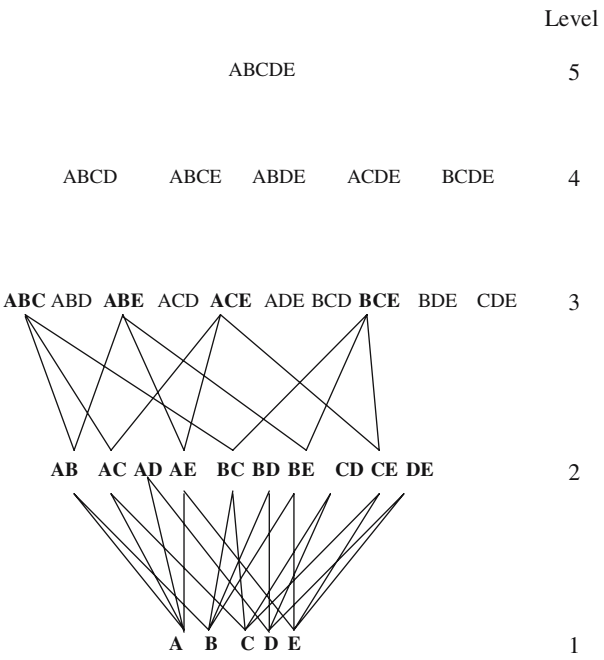

Fig. 7 Portion of semi-lattice accessed by FD_Mine

Proof When FD_Mine is applied to $r(U)$, the possible set of candidates includes all nonempty, proper subsets of U. For any candidate X, we check all possible consequents $v_i \in U - X$. By Union, it is sufficient to check this singleton attribute v_i rather than all possible subsets of $U - X$.

For a candidate X and a possible consequent v_i, we first prove that if $X \rightarrow v_i$ is satisfied by $r(U)$ then $X \rightarrow v_i \in F^+$. Suppose $X \rightarrow v_i$ is satisfied by $r(U)$. In this case, $X \rightarrow v_i$ is either discovered by FD_Mine or not. If $X \rightarrow v_i$ is discovered by FD_Mine, then $X \rightarrow v_i \in F$ and it follows that $X \rightarrow v_i \in F^+$. If $X \rightarrow v_i$ is not discovered by FD_Mine, then according to lines 2 and 3 in *ObtainFDs*, $X \notin C_k$ or $v_i \notin U - X^+$, which could only occur if candidate X is pruned by pruning rules 1, 2, or 4, or v_i is pruned by pruning rule 3. Since Lemmas 1 and 2, Theorem 4, Theorem 5, and Lemma 3 guarantee that pruning rules 1, 2, 3, and 4, respectively, are correct, it follows that $X \rightarrow v_i \in F^+$.

Now we prove that if $X \rightarrow v_i \in F^+$ then $X \rightarrow v_i$ is satisfied by $r(U)$. For any FD $X \rightarrow v_i \in F^+$, either $X \rightarrow v_i \in F$ or $X \rightarrow v_i \notin F$. If $X \rightarrow v_i \in F$, then by Theorem 2, FD $X \rightarrow v_i$ is satisfied by $r(U)$. Otherwise, if FD $X \rightarrow v_i \notin F$, then by Theorem 1, $X \rightarrow v_i$ is satisfied by $r(U)$. □

Theorem 6 indicates that the closure of the set F of FDs discovered by FD_Mine is complete. Therefore, by applying its four pruning rules, FD_Mine prunes only redundant FDs that can be inferred from F using Armstrong's Axioms.

The time complexity of the FD_Mine algorithm depends on the number of attributes m, the number of tuples n, and the degree of correlation among the attributes. The time required varies for different relations, since the number and levels of the equivalent candidates and the FDs that are discovered vary for different relations. The worst case occurs when no FDs are found in the relation, and all combinations of the attributes are tested. The number of combinations of all attributes is $C_1^m + C_2^m + \cdots + C_m^m = 2^m - 1$. Since the combination $U = v_1 \ldots v_m$ does not need to be tested, only $2^m - 2$ candidates will be tested, but this small constant is ignored in the complexity analysis. Thus, the worst case time complexity is $O(n \cdot 2^m)$, assuming that the partition of a candidate can be computed in time $O(n)$, as indicated in Huhtala et al. (1999). Suppose an equivalence $X \leftrightarrow Y$ is discovered at level k, $1 < k < m$. Then all supersets of Y above level k will be eliminated by the pruning rules. Since the number of supersets of Y above level k is $2^{m-k} - 1$, then time complexity will be reduced to $O(n \cdot (2^m - 2^{m-k}))$. If more FDs or equivalent candidates are discovered in the relation, then more of the search space can be pruned by using the discovered information.

6 Experimental results

In this section, experiments on fifteen UCI datasets (UCI Repository of machine learning databases 2005) are summarized, and then a comparison between FD_Mine and TANE (Huhtala et al. 1999) is presented.

6.1 Experimental summary

FD_Mine was applied to fifteen datasets obtained from the UCI Machine Learning Repository (2005). The results are summarized in Table 4, 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, and column 7 is the CPU time in seconds, measured on a SGI R12000 processor.

The results show that in practice the processing time is mainly determined by the number of attributes m, as expected from the $O(n \cdot 2^m)$ theoretical complexity. For example, *Imports-85* dataset contains 26 attributes, the running time is long, even though it only has 205 tuples. The *Chess* dataset has nearly 30,000 tuples, but only 6 attributes, and its running time is much shorter.

6.2 Algorithm comparison

A comparison between FD_Mine and TANE is given. TANE is selected for comparison, because it is the best known previous algorithm for the problem. Work on TANE established the theoretical framework for the problem. The approaches used in TANE is very effective for datasets where no equivalences

Table 4 Experimental results for FD_Mine on fifteen datasets

Dataset Name	# of Attributes	# of Tuples	# of FDs checked	# of FDs found	# of Equivalences	Time (s)
Abalone	8	4,177	594	60	8	1
Balance-scale	5	625	70	1	0	0
Breast-cancer	10	191	5,095	3	0	0
Bridge	13	108	15,397	61	48	2
Cancer-Wisconsin	10	699	4,562	19	0	1
Chess	7	28,056	434	1	0	3
Crx	16	690	47,919	494	235	10
Echocardiogram	13	132	2,676	536	30	0
Glass	10	142	405	27	1	0
Hepatitis	20	155	1,161,108	7,381	4,862	1,327
Imports-85	26	205	2,996,737	3,971	19,888	8,322
Iris	5	150	70	4	0	0
Led	8	50	477	11	1	0
Nursery	9	12,960	2,286	1	0	16
Pendigits	17	7,494	223,143	27,501	671	920

among attributes exist. However, for datasets containing equivalent attributes, TANE may perform unnecessary checking of FDs. For the relation given in Table 2, Fig. 8 shows the portion of the semi-lattice accessed by TANE. The number of edges is the number of FDs to be checked, and the number of highlighted candidates is the number of partitions to be computed by accessing the relation. The portions of semi-lattices shown in Fig. 7 for FD_Mine and Fig. 8 for TANE both have fewer edges than the semi-lattice shown in Fig. 1. In addition, the portion of the semi-lattice accessed by FD_Mine has fewer edges than that accessed by TANE. TANE checks 41 FDs and computes 25 partitions of candidates, which is more than the 32 FDs checked and 19 partitions computed by FD_Mine.

Table 5 further compares the number of FDs checked on this sample relation at each level. TANE checks 21 FDs at level 3, while FD_Mine checks fewer FDs, since candidates AD, BD, CD, ED, ABD, ACD, ADE, BCD, BDE, and CDE are removed after $A \leftrightarrow D$ is found to hold.

To assess performances of FD_MINE, synthetic datasets are generated using three parameters, namely, $|D|$, $|V|$ and cf, where $|D|$ is the number of tuples, $|V|$ is the number of attributes, and cf is a correlation factor and models the different degrees of correlation among attributes. The values of cf range from 0 to 1. The synthetic dataset is created in following way. Suppose $|D| = 100,000$, $|V| = 50$ and $cf = 0.5$, a table with 50 attributes and 100,000 tuples is created. The value of each inserted tuple depends on the correlation factor cf, which controls the number of possible values in a column of the table. More precisely, the range of distinct values for each attribute is $(1 - cf) * |D|$. For example, if $cf = 0.5$ and $|D| = 100,000$, then each value for this attribute is chosen from among $(1 - cf) \cdot |D| = 50,000$ possible values.

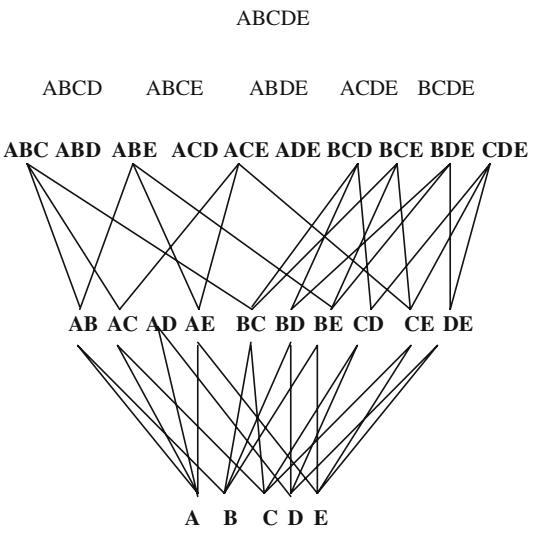

Fig. 8 Portion of the semi-lattice accessed by TANE

Table 5 Number of FDs checked for the sample relation in Table 2

	FD_Mine	TANE
Level 1	0	0
Level 2	20	20
Level 3	12	21
Total	32	41

We studied the effect of different values for the correlation factor *cf* on the processing time and the number of attributes that contributed to the search of FDs for the FD_MINE and TANE algorithms. For each experiment, the number of tuples is fixed to 100,000, the values of correlation factor *cf* varied from 30 to 50%, and the number of attributes ranges from 10 to 60 for each experiment. The experimental results are summarized in Figs. 9 and 10. The experimental results indicate that FD_MINE is more effective than TANE in every case tested since it can eliminate more candidates than TANE using more effective pruning rules. Both figures shows that the gap between execution times of the FD_MINE and TANE is strongly enlarged when the number of attributes increases. It indicates that FD_MINE is more efficient than TANE when the number of attributes is large. For example, in Fig. 9, when *cf* is set to 30%, the number of tuples is set to 100,000, and the number of attributes is set to 60, the execution time of FD_MINE is 101 seconds, but execution time of TANE is 392 s. In Fig. 10, the maximum number of attributes is 50 rather than 60 since TANE ran out of memory when the number of attributes is 60. That is, TANE used up all 12GB of available memory while generating candidates.

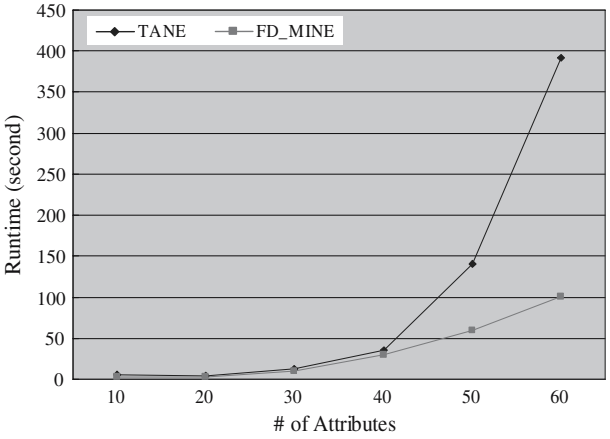

Fig. 9 Execution time for correlated data with 100,000 tuples ($cf = 30\%$)

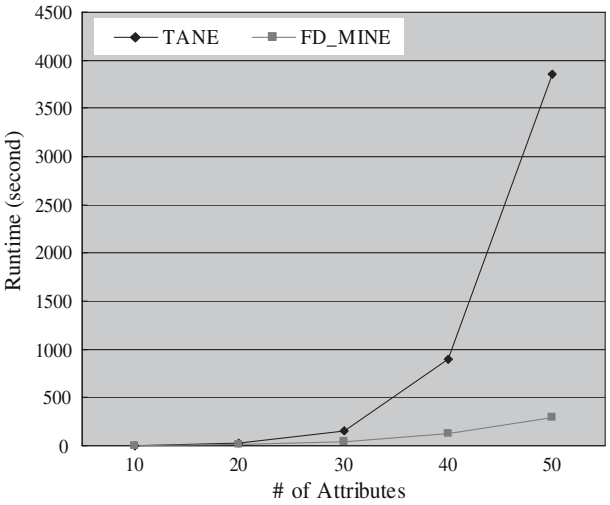

Fig. 10 Execution time for correlated data with 100,000 tuples ($cf = 50\%$)

7 Conclusion

This paper addressed the problem of discovering functional dependencies in a relation. We showed how to simplify the problem by finding equivalences among attributes and calculating the nontrivial closures of candidate sets of attributes. In this paper, we provide three research contributions towards this problem.

First, we identified four effective pruning rules relevant to the search for functional dependencies using equivalence and nontrivial closure. These rules are incorporated into the knowledge discovery process to reduce the number of functional dependencies to be checked on a dataset without leading to the

loss of any useful information. The correctness of pruning rules 1, 2, 3, and 4 is supported by Lemmas 1 and 2, Theorem 4, Theorem 5, and Lemma 3, respectively. Pruning rule 1 is original to our work. Arguably, our formulation of pruning rules 2, 3, and 4 is preferable to previous formulation, because with our approach the conditions under which these three rules apply can be identified using only nontrivial closure. By Theorems 3, pruning rule 1 can also be identified using nontrivial closure. Therefore, by calculating only the nontrivial closures of candidates, our algorithm can efficiently integrate four pruning rules into the *Prune* function, which facilitates the implementation of the the FD_Mine algorithm and also improves the performance of the algorithm.

The second contribution is the FD_Mine algorithm, which finds a set F of functional dependencies that are consistent with a dataset. F^+, the closure of F under Armstrong's Axioms, is equivalent to set of all functional dependencies that are consistent with the dataset, as supported by Theorem 6. We proved the correctness of FD_Mine, that is, the FD_Mine algorithm is guaranteed not to eliminate any valid candidates when it reduces the size of the dataset or the number of FDs to be checked.

The third contribution is our report of the results of a series of experiments on synthetic and real data. The results show that the pruning rules in the FD_Mine algorithm are effective, because they reduce the overall amount of checking required to find FDs from data.

The FD_Mine algorithm could be adapted to find implications in binary datasets. As indicated by Baixeries (2004), the only difference between the processes of judging FDs and judging implications is that the former are judged based on a partition of the set of the tuples, and the latter are judged based on the set of tuples. Since Baixeries (2004) showed that a filter function can be used to find implications as well as functional dependencies, FD_Mine could be adapted to find implications by introducing the same function. More precisely, two changes to the *ObtainFDs* algorithm shown in Fig. 4 would be sufficient. First, a definition of the filter function used in Baixeries (2004) should be added to *ObtainFDs*. Second, since in Line 4 of *ObtainFDs*, a decision is made about whether or not an FD is satisfied by data according to the partition of the set of tuples, this line should be changed to check whether or not an implication is satisfied by data according to the set of tuples. By skipping the unnecessary sets of tuples using this filter function, FD_Mine could thus be adapted to finding implications.

Future work could also attempt to extend the FD_Mine algorithm to discover other data dependencies, such as multivalued dependencies (Maier 1983) and conditional dependencies (Yao et al. 2005). In addition, with its emphasis on equivalences, the FD_Mine approach might be applied to the data cleaning phase of data mining (Kalashnikov and Mehrotra 2006) to reduce the size of a relation without losing any useful information.

References

Agrawal R, Imielinski T, Swami AN (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., pp 207–216

Baixeries J (2004) A formal concept analysis framework to mine functional dependencies. In: Proceeding of the Workshop on Mathematical Methods for Learning, Villa Geno, Italy

Baixeries J (2007). Lattice characterization of Armstrong and symmetric dependencies. Ph.D. Thesis, Universitat Politècnica de Catalunya, Spain, 2007

Carpineto C, Romano G, d'Adamo P (1999) Inferring dependencies from relations: a conceptual clustering approach. Computat Intelligence 15(4):415–441

Demetrovics J, Libkin L, Muchnik IB (1992) Functional dependencies in relational databases: a lattice point of view. Disc Appl Math 40:155–185

Fagin R (1977) Functional dependencies in a relational database and propositional logic. IBM J Res Dev 21(6):534–544

Flach PA, Savnik A (1999) Database dependency discovery: a machine learning approach. AI Commun 12(3):139–160

Flesca S, Furfaro F, Greco S, Zumpano E (2005) Repairing inconsistent XML data with functional dependencies. Encycl Database Technol Appl Idea Group 542–547

Ganter B, Wille R (1999) Formal concept analysis: mathematical foundations. Springer, Berlin/Heidelberg

Goodaire EG, Parmenter MM (1992) Discrete mathematics with graph theory. Prentice Hall, New Jersey

Huhtala Y, Karkkainen J, Porkka P, Toivonen H (1999) TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput J 42(2):100–111

Kalashnikov VD, Mehrotra S (2006) Domain-independent data cleaning via analysis of entity-relationship graph. ACM Trans Database Syst 31(2):716–767

Lopes S, Petit J-M, Lakhal L (2000) Efficient discovery of functional dependencies and Armstrong relations. In: 7th International Conference on Extending Database Technology (EDBT 2000), pp 350–364

Lopes S, Petit J-M, Lakhal L (2002) Functional and approximate dependency mining: database and FCA points of view. Special issue of J Exp Theor Artif Intelligence (JETAI) on Concept Lattices for KDD 14(2–3):93–114

Maier D (1983) The theory of relational databases. Computer Science Press, Rockville, Maryland

Mannila H, Raiha KJ (1994) Algorithms for inferring functional dependencies from relations. Data Knowl Eng 12(1):83–99

Mannila H, Toivonen H (1997) Levelwise search and borders of theories in knowledge discovery. Data Min Knowl Disc 1(3):241–258

Novelli N, Cicchetti R (2001a) FUN: an efficient algorithm for mining functional and embedded dependencies. In: Proceedings of the International Conference on Database Theory, London, UK, pp 189–203

Novelli N, Cicchetti R (2001b) Functional and embedded dependency inference: A data mining point of view. Inform Syst 26(7):477–506

Ramakrishnan R, Gehrke J (2002) Database management systems. McGraw-Hill, New York

Sagiv Y, Delobel C, Parker DS, Fagin R (1981) An equivalence between relational database dependencies and a fragment of propositional logic. J ACM 28(3):435–453

Tan HBK, Zhao Y (2004) Automated elicitation of functional dependencies from source codes of database transactions. Inform Software Technol 46(2):109–117

UCI Repository of machine learning databases (2005) http://www.ics.uci.edu/~mlearn/MLRepository.html

Ullman JD (1982) Principles of database systems. Computer Science Press, Rockville

Wyss C, Giannella C, Robertson EL (2001) FastFDs, a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2001), pp 101–110

Yao H, Hamilton HJ, Butz CJ (2002) FD_Mine: discovering functional dependencies in a database using equivalences. In: Proceedings of the 2nd IEEE International Conference on Data Mining, Maebashi City, Japan, pp 729–732

Yao H, Butz CJ, Hamilton HJ (2005) Causal discovery. In: Maimon O Rokach L (eds) The data mining and knowledge discovery handbook, Springer, pp 945–955