Discovery of Approximate Differential Dependencies

Jixue Liu a  Selasi Kwashie a  Jiuyong Li a  
Feiyue Ye b  Millist Vincent a  

a University of South Australia, 
b Jiangsu University of Technology 

{jixue.liu, selasi.kwashie, jiuyong.li}@unisa.edu.au  
millist.vincent@unisa.edu.au,  yfy@jstu.edu.cn

Abstract

Differential dependencies (DDs) capture the relationships between data columns of relations. They are more general than functional dependencies (FDs) and the difference is that DDs are defined on the distances between values of two tuples, not directly on the values. Because of this difference, the algorithms for discovering FDs from data find only special DDs, not all DDs and therefore are not applicable to DD discovery. In this paper, we propose an algorithm to discover DDs from data following the way of fixing the left hand side of a candidate DD to determine the right hand side. We also show some properties of DDs and conduct a comprehensive analysis on how sampling affects the DDs discovered from data.

keywords: Differential dependencies, functional dependencies, dependency discovery, knowledge discovery, lattice, partition

1 Introduction

Data quality has been a core concern in data management. Many types of constraints, like logic constraints, keys, referential constrains, and functional dependencies, have been designed to control data quality in databases. Differential dependency [16] is a new type of such dependencies. It also generalizes metric functional dependencies defined in [7].

Differential dependencies (DDs) are defined to constrain the closeness of the values of dependent attributes with respect to the closeness of the values of determinant attributes. More specifically, the differential dependency \( X \langle w \rangle \rightarrow A \langle w_1 \rangle \) requires that when the distance of any two tuples on the attribute set \( X \) is within the range \( w \), the distance of the same two tuples on the attribute \( A \) are within the range \( w_1 \). For example, \( Age \langle [0, 0] \rangle \rightarrow Sal \langle [0, 1] \rangle \) is a DD that requires that if the \( Age \) difference of two tuples is 0, their \( Sal \) difference should
be no more than 1. This DD is satisfied by the data in Table 1. \( Age([0, 0]) \) is called the left hand side (lhs) and \( Sal([0, 1]) \) the right hand side (rhs).

Table 1: A table satisfying \( Age([0, 0]) \rightarrow Sal([0, 1]) \) but violating \( Age([0, 0]) \rightarrow Sal([0, 0]) \)

| tid | Age | Edu | Sex | Sal |
|-----|-----|-----|-----|-----|
| t₁  | 20  | 3   | 0   | 3   |
| t₂  | 20  | 3   | 1   | 3   |
| t₃  | 20  | 4   | 0   | 4   |
| t₄  | 25  | 5   | 1   | 5   |

\( Edu \) means education completed: 3=Bachelors 4=Masters 5=PhD

Differential dependencies are defined on closeness of tuples and the closeness is relative. They do not rely on absolute values. In other words, to satisfy \( Age([0, 0]) \), the \( Age \) values of two tuples can be 20 and 20, and can also be 50 and 50; in both cases, the difference is 0.

Like functional dependencies and matching dependencies [5], differential dependencies can be used in many applications. They can be used to warrant data quality like what function dependencies do. For example, a DD can be defined to require that if two post codes of two locations are close, their addresses must be in the same city or the distance between their addresses must be within a certain range.

Differential dependencies can also be used to detect data quality issues in data cleaning. If a discovered DD shows that two attributes that should take close values have taken values that are far apart, their data has quality problems. This gives guide to data cleaning processes to quickly identify problems.

Differential dependencies holding on data may describe patterns that are new to domain experts. They enrich the knowledge in the application areas [2]. In addition, the knowledge represented by the DDs can be used in query processing and data management as it is done in inductional databases [13].

Among many of these applications, especially in knowledge discovery and data cleaning, the discovery of DDs from existing data is a critical task. Because DDs are more general than other types of dependencies like FDs and match dependencies etc, some of the patterns that are interesting and can be described in DDs cannot be described in previous dependencies. This raises two problems. One is that the algorithms used to find previous dependencies will not find DD specific patterns. The other is that if we want to know these DD specific patterns, new algorithms much be developed. This motivates the work in this paper and developing an effective and efficient algorithm is the objective of this paper.

The most relevant work on DD discovery to our work is the reduction algorithm proposed in [16]. It uses a fix-rhs-reduce-lhs approach and reduces the search space containing subsumption-ordered nodes to find DDs for each rhs attribute and every of its interval. This approach involves storing an exponentially sized search space, which has a severe performance bottleneck and makes
the applicability of the algorithm very much limited to relations with very small number of attributes. At the same time, the distance intervals of attributes are assumed to start from 0 in its experiments. This leads to some possible DDs not to be found. The detailed analysis of these is given in the section of Related Work of this paper. In contrast, our algorithm follows a fix-left-find-right approach: it checks all possible lhs candidates and determines the rhs and it finds all DDs.

In this paper, we show that DDs have a property that given a lhs for a DD, a rhs can always be found if its interval is large enough. Thus, there is a possibility that the rhs interval is affected by outliers in the data which reduces the usefulness of discovered dependencies. We propose to use approximate satisfaction so that outlier data points can be identified and ignored.

We propose an algorithm to discover differential dependencies and the algorithm is a partition based approach. Pruning rules are used to reduce the search space. At the same time, we use two parameters, support to reduce the complexity of the computation, and interestingness to find only 'good' DDs. The interestingness parameter is design to avoid DDs with trivially large rhs intervals.

We conduct a comprehensive analysis on how sampling, while it reduces the size of the computation, introduces errors in the discovered DDs and possible ways to filter out the errors. Sampling causes ‘wanted’ DDs to be missed, ‘unwanted’ DDs to be discovered, and trivial DDs to appear like non-trivial DDs.

Our experiments show that (1) our approach is effective in identifying problems in data when DDs of full satisfaction and of approximate satisfaction are compared; (2) our algorithm is efficient in space and time consumption; (3) our algorithm discovered outliers in the Adult data. (4) Errors of sampling can be significant, but some of the errors can be filtered.

The rest of the paper is organized in the following way. In Section 2 preliminary definitions are given. Section 3 presents some properties of DD satisfaction. Section 4 presents the discovery algorithm and the pruning techniques. Section 5 details how sampling is done. Section 6 presents the experiment results. Section 7 discusses the works in the literature related to this work. The final section concludes the paper and discusses future work.

2 Differential dependencies and properties

We use $R$ to denote a relation schema and $r$ a relation instance of $R$. $\text{dom}(A)$ denotes the domain of attribute $A$. For a set of attributes $X = \{A_1, \cdots, A_k\}$ and a single attribute $B$, $XB$ means $\{A_1, \cdots, A_k, B\}$. $|X|$ returns the number of elements in $X$.

Let $d_A(v_1, v_2)$ be a function calculating the distance of the two values, $v_1$ and $v_2$, of attribute $A$. It is assumed that $d_A(v_1, v_2) = d_A(v_2, v_1) \geq 0$. The distance function can be defined in editing distances of text values, differences of numeric values, or in other ways. A differential function (DF) of attribute $A$ with
regard to a distance interval \( w = [d_1, d_2] \) is a boolean function and is defined by 
\[ A(w) = d_1 \leq d_A(v_1, v_2) \leq d_2. \] The two functions \( \text{left}(w) \) and \( \text{right}(w) \) return 
\( d_1 \) and \( d_2 \) respectively. If \( d_1 = d_2 \), the notation of the interval is simplified to 
\( w = [d_1] \).

Let \( w_a = [d_1, d_2] \) and \( w_b = [d_3, d_4] \) be two intervals of an attribute \( A \). The \textbf{order} \( w_a \leq w_b \) holds if \( d_2 \leq d_3 \). \( w_a \) and \( w_b \) are \textbf{adjacent}, denoted by \( w_a \leq w_b \), if \( d_2 = d_3 \). The \textbf{combination} of two adjacent intervals \( w_a \) and \( w_b \), denoted by 
\( w_a + w_b \), is \([d_1, d_4]\).

A differential function of a set of attributes \( X = \{A_1, \cdots, A_k\} \) with regard to a cube \( w = w_1 \times \cdots \times w_k = [d_{11}, d_{12}] \times \cdots \times [d_{k1}, d_{k2}] \) is defined by \( X(w) = A_1(w_1) \cdots A_k(w_k) = \bigwedge_{i=1}^k A_i(w_i) \). Two cubes \( w_a \) and \( w_b \) on the same set of attributes are \textbf{adjacent} if their intervals on one of the attributes are adjacent, and their intervals on all other attributes are the same. The \textbf{combination} of two adjacent cubes \( w_a \) and \( w_b \), denoted by \( w_a + w_b \), is the smallest cube containing both \( w_a \) and \( w_b \). In the rest of this paper, cubes are also referred to as intervals.

Two differential functions \( X(w_a) \) and \( Y(w_b) \) are \textbf{joinable} if common attributes have the same interval in both functions, i.e., for each \( A \in (X \cap Y) \) we have \( A(w_a) \in X(w_a) \) and \( A(w_b) \in Y(w_b) \). The \textbf{join} of two joinable \( X(w_a) \) and \( Y(w_b) \), denoted by \( X(w_a)Y(w_b) \), is the cube containing all DFs of \( X(w_a) \) and \( Y(w_b) \), i.e., \( A_1(w_1) \cdots A_n(w_n) \) where \( n = |X \cup Y| \) and \( \forall A_i(w_i)[i \in [1, \cdots, n]](A_i(w_i) \in X(w_a) \text{ or } A_i(w_i) \in Y(w_b)) \).

A differential function \( X(w_x) \) \textbf{subsumes} a differential function \( Y(w_y) \), denoted by \( X(w_x) \succeq Y(w_y) \), if \( \forall A_j(w_{xj}) \in X(w_x) \ (\exists A_j(w_{yj}) \in Y(w_y))(w_{yj} \subseteq w_{xj}) \). That is, the subsuming function (the left hand side) has less dimensions and larger intervals.

\textbf{Definition 2.1} (Differential dependency (DD)). \cite{16} A DD is a formula \( f = X(w_x) \rightarrow Y(w_y) \) where \( X(w_x) \) and \( Y(w_y) \) are differential functions. A relation \( r \) satisfies \( f \) if and only if for any two tuples \( t_1 \) and \( t_2 \) in \( r \), if \( X(w_x) \) returns true, \( Y(w_y) \) returns true. \( X(w_x) \) is the lhs and \( Y(w_y) \) is the rhs.

\textbf{Definition 2.2} (Partition). Given a set of attributes \( X \) and an interval \( w \) of \( X \), the \textbf{tuple pair partition} (partition, for short) for DF \( X(w) \) is a set of tuple pairs satisfying \( X(w) \):

\[
F(X(w)) = \{(t_p, t_q) \mid (t_p, t_q) \in r \land r \land t_p \neq t_q \land \forall A_i(w_i) \in X(w) \} \]

\[
\text{left}(w_i) \leq d_A, (t_p[A_i], t_q[A_i]) \leq \text{right}(w_i)) \}
\]

Given a sequence of adjacent intervals \( w_1, \cdots, w_k, w_i \leq w_{i+1} \) \((i = 1, \cdots, k-1)\) of the single attribute \( B \), the \textbf{partition for attribute} \( B \) is defined to be the sequence of labeled partitions for \( B \)'s intervals in order:

\[
\mathcal{F}(B) = [w_1 : F(B\langle w_1 \rangle), \cdots, w_k : F(B\langle w_k \rangle)]
\]
Tuple pair partitions for DFs are similar to the partitions for relations. More details can be found in \cite{3}.

For performance reasons, tuple pairs in a partition are ascending ordered. This order is defined to be \((t_p, t_q) < (t_u, t_v)\) if \(t_p < t_u\) or \((t_p = t_u\) and \(t_q < t_v\)). In this way, operations on partitions can be done in linear time of their sizes.

**Example 2.1.** If we assume that \(d_A(v_1, v_2) = abs(v_1 - v_2)\) for any attribute \(A\) in Table \ref{5} the table satisfies \(Age([0]) \rightarrow Sal([0, 1])\) but violates \(Age([0]) \rightarrow Sal([0])\). To see these, we calculate partitions and have

\[
\begin{align*}
F(Age([0])) &= \{(t_1, t_2), (t_1, t_3), (t_2, t_3)\} \\
F(Age([5])) &= \{(t_1, t_4), (t_2, t_4), (t_3, t_4)\} \\
F(Age) &= \{0 : \{(t_1, t_2), (t_1, t_3), (t_2, t_3)\}, 5 : \{(t_1, t_4), (t_2, t_4), (t_3, t_4)\}\} \\
F(Sal([0])) &= \{(t_1, t_2)\} \\
F(Sal([1])) &= \{(t_1, t_3), (t_2, t_3), (t_3, t_4)\} \\
F(Sal([2])) &= \{(t_1, t_4), (t_2, t_4)\} \\
F(Sal([0, 1])) &= \{(t_1, t_2), (t_1, t_3), (t_2, t_3), (t_3, t_4)\}
\end{align*}
\]

where, as examples, \(F(Age([0]))\) is the set of all tuple pairs whose \(Age\) distances are 0, and \(F(Sal([1]))\) the set of all tuple pairs whose \(Sal\) distances are 1. We notice that all the tuple pairs in \(F(Age([0]))\) are in \(F(Sal([0, 1]))\). So \(Age([0]) \rightarrow Sal([0, 1])\). Tested in the same way, \(F(Age([0])) \not\subseteq F(Sal([0]))\), so \(Age([0]) \nrightarrow Sal([0])\). □

Because the distance \(d\) of two tuples on an attribute \(A\) is a specific number, \(d\) falls in only one of the two non-overlapping intervals \(w_1\) and \(w_2\) of \(A\), but not in both at the same time. Thus, we have the following properties for tuple pair partitions.

**Lemma 2.1** (Properties of partitions).

1. \(F(X(w_a)) \cap F(X(w_b)) = \emptyset\) if \(w_a\) and \(w_b\) do not overlap.
2. \(F(X(w_a + w_b)) = F(X(w_a)) \cup F(X(w_b))\) if \(w_a\) and \(w_b\) are adjacent.
3. \(F(X(w_a)Y(w_2)) = F(X(w_a)) \cap F(Y(w_2))\) if \(X(w_a)\) and \(Y(w_2)\) are joinable.

Item (3) of the lemma tells how the partition for two joining DFs should be calculated. This will be used in our algorithm later on.

In this paper, we are interested in DDs with single attributes on the rhs like in the case of functional dependency discovery \cite{6,9}. The reason is that if we know \(X(w) \rightarrow B(w_b)\) and \(X(w) \rightarrow C(w_c)\), we can derive \(X(w) \rightarrow B(w_b)C(w_c)\) based on the inference rules proposed in \cite{16}.

With tuple pair partitions, the satisfaction of DD follows the lemma below.

**Lemma 2.2.**

1. \(F(X(w)) \subseteq F(B(w'))\) if and only if \(X(w) \rightarrow B(w')\).
2. \(F(X(w)B(w')) = F(X(w))\) if and only if \(X(w) \rightarrow B(w')\).
3. \(|F(X(w)B(w'))| = |F(X(w))|\) if and only if \(X(w) \rightarrow B(w')\).

The lemma is correct. (a) follows the DD definition. (b) is correct because of Lemma 2.1(3). (c) is correct because of (b).
**Definition 2.3** (Minimal DD). Given a set \( \Sigma \) of DDs, a DD is minimal if it has a single attribute DF on the rhs and is not implied by other DDs in \( \Sigma \).

All implication axioms of DDs are given in [16]. The ones that are used in this paper will be given shortly and they will not include the transitivity rule. In other words, our implication is defined in a restricted way, which is a common shortage in all level-wise functional dependency (FD) discovery algorithms [6, 12, 20].

Minimal DDs are very different from minimal FDs because of attribute intervals. In FDs, if \( A \rightarrow B \), then \( AC \rightarrow B \) is not minimal. However in DDs, if \( A \langle w_1 \rangle \rightarrow B \langle w_2 \rangle \) holds, the DD \( A \langle w_1 \rangle C \langle w_3 \rangle \rightarrow B \langle w_2b \rangle \) may still be minimal because \( w_2b \) can be smaller from \( w_2 \). When DDs and FDs are compared, intervals for DDs work like extra attributes for FDs.

The rules below will be used to detect non-minimal DDs. We are interested in discovering minimal DDs and non-minimal ones will be ignored.

**Lemma 2.3.** If \( X \langle w_1 \rangle \rightarrow B \langle w \rangle \) holds, then the following DDs are not minimal:

(a) \( X \langle w_1 \rangle \rightarrow B \langle w \rangle \) if \( X \langle w_x \rangle \geq X \langle w_1 \rangle \). (smaller lhs)

(b) \( X \langle w_x \rangle Y \langle w_y \rangle \rightarrow B \langle w \rangle \). (extra DF \( Y \langle w_y \rangle \) making lhs smaller)

(c) \( X \langle w_x \rangle \rightarrow B \langle w_2 \rangle \) where \( B \langle w_2 \rangle \geq B \langle w \rangle \). (larger rhs)

(d) If \( B \langle w_2 \rangle \geq B \langle w \rangle \), then DD \( X \langle w_x \rangle B \langle w_2 \rangle Y \langle w_y \rangle \rightarrow C \langle w_c \rangle \) is implied by \( X \langle w_x \rangle Y \langle w_y \rangle \rightarrow C \langle w_c \rangle \). (lhs reducible)

These rules can be easily proved using Lemma 2.2. They lay the foundation for minimality check in the algorithm proposed later.

**Lemma 2.4.** If \( X \langle w_x \rangle A \langle w_1 \rangle \rightarrow B \langle w \rangle \) and \( X \langle w_x \rangle A \langle w_2 \rangle \rightarrow B \langle w \rangle \) hold and \( w_1 \) and \( w_2 \) are adjacent, then \( X \langle w_x \rangle A \langle w_1 + w_2 \rangle \rightarrow B \langle w \rangle \).

The lemma is correct because the two DDs have the same rhs and because of Lemma 2.2(a). It will be used to combine DDs so that non-minimal DDs described by Lemma 2.3(a) can be removed.

### 3 DD satisfaction

In this section, we present results on properties of DD satisfaction and define approximate satisfaction. The results will guide us to find satisfied DDs more efficiently.

Consider DD \( X \langle w_x \rangle \rightarrow B \langle w \rangle \) where \( B \not\in X \). We would like to know what \( w \) should be if the DD is satisfied. Assume that two points \( p_1 \) and \( p_2 \) represent two tuples in relation \( r \) and they are \( w_x \)-apart from each other along \( X \). Let their \( B \)-distance be defined by \( \text{abs}(p_1[B] - p_2[B]) \) where \( p[B] \) means the \( B \) coordinate of \( p \). The maximal (minimal resp.) \( B \)-distance for \( X \langle w_x \rangle \) is the maximal (minimal) \( B \)-distance of all the point pairs that are \( w_x \)-apart along \( X \) in the space \( XB \).

Figure 1 shows some data points represented by small circles and a \( B \)-distance \( w_b \) for the interval \( w_x \). This distance is also the maximal \( B \)-distance
for \( w_x \). The minimal \( B \)-distance is 0 which occurs between \( p_1 \) and \( p_3 \) and some other pairs. Motivated by the example, we have the following observation.

**Observation 3.1.** The DD \( X\langle w_x \rangle \rightarrow B\langle w \rangle \) is satisfied by a relation \( r \) if and only if \( \text{left}(w) \) is the minimal \( B \)-distance and \( \text{right}(w) \) the maximal \( B \)-distance for all tuple pairs that are \( w_x \)-apart on \( X \).

The observation can also be formally proved, which is straightforward and left out.

In many cases, especially when the number of tuples of the relation is large, \( \text{left}(w) \) would be 0. The case \( \text{left}(w) \neq 0 \) happens only if there are no two points that are \( w_x \)-part on a plane parallel to \( X \). One example of such a case is that the distinct points form vertical lines perpendicular to the plane (like the 4 points on the left in Figure 1) and the lines are more than \( w_x \)-apart. Obviously the \( \text{left}(w) \neq 0 \) case is rare.

An implication of this observation is that for any DD \( X\langle w_x \rangle \rightarrow B\langle w \rangle \), if \( w \) is wide enough, the DD is always satisfied. Let \( \text{maxd}(B) \) be the maximal distance of attribute \( B \) of all tuple pairs in the relation \( r \). We have the following observation.

**Observation 3.2.** The DD \( X\langle w_x \rangle \rightarrow B\langle [0, \text{maxd}(B)] \rangle \) is always satisfied without respect to what \( X\langle w_x \rangle \) is.

The observation indicates that for any DF \( X\langle w_x \rangle \) and an attribute \( B \), we can always find \( [0, \text{maxd}(B)] \) such that \( X\langle w_x \rangle \rightarrow B\langle [0, \text{maxd}(B)] \rangle \) is satisfied. However such a DD is not useful because the rhs interval does not indicate any specific closeness. Thus, in addition to the search for valid \( X\langle w_x \rangle \) and pruning the search, which will be presented in the next section, another problem of the discovery is to find the tightest \( w \) for \( X\langle w_x \rangle \) and \( B \). This is done by following the partition definitions and Observation 3.1.

We calculate DF partition \( F(X\langle w_x \rangle) \) and the attribute partition \( F(B) \) which contains ordered partitions of DFs of the single attribute \( B \). We then search \( F(B) \) from right to left for the first DF partition \( F(B\langle w_j \rangle) \) that overlaps with \( F(X\langle w_x \rangle) \). The \( \text{right}(w_j) \) is what \( \text{right}(w) \) should be. In the same way, we
search $F(B)$ from left to right for the first DF partition $F(B\langle w_i \rangle)$ that overlaps with $F(X\langle w_x \rangle)$. The left($w_i$) is what left($w$) should be. Formally,

$$left(w) = \min\{left(w_i) \mid F(B\langle w_i \rangle) \in F(B) \land F(B\langle w_i \rangle) \cap F(X\langle w_x \rangle) \neq \phi\}$$

$$right(w) = \max\{right(w_j) \mid F(B\langle w_j \rangle) \in F(B) \land F(B\langle w_j \rangle) \cap F(X\langle w_x \rangle) \neq \phi\}$$

(3)

Because the DF partitions of $F(B)$ are ordered by $w_i$, calculating the set intersections from the two ends of $F(B)$ is efficient.

We now show the motivation for approximate satisfaction of DDs. Given two DDs $f_1 = X\langle w_x \rangle \rightarrow B\langle [0,d_1] \rangle$ and $f_2 = X\langle w_x \rangle \rightarrow B\langle [0,d_2] \rangle$ with the same lhs, if $d_2$ is much smaller than $d_1$, then, $f_2$ is much more interesting than $f_1$ as the tuples satisfying the lhs are closer on $B$. Closeness is the soul of DDs. Unfortunately, isolated points like $p_2$ in Figure 1 have significant negative impact on the usefulness of DDs. As shown in the figure, the $w_b$ must cover the distance of the pair $p_1$ and $p_2$ for the DD to be satisfied. In contrast, if $p_2$ did not exist, the $w_b$ would be much smaller and covers just the dense patch of pairs. For this reason, we define approximate DDs.

**Definition 3.1** (Approximate differential dependency satisfaction). Given a large number $\epsilon$ in $[0,1]$ called the approximate satisfaction threshold, the DD $X\langle w_x \rangle \rightarrow B\langle w \rangle$ is approximately satisfied ($\epsilon$-satisfied) if, for all tuple pairs satisfying $X\langle w_x \rangle$, at least $\epsilon$ fraction of them with lowest distance satisfy $B\langle w \rangle$.

2

The definition can be easily adapted if the majority of tuple pairs have large distances and a few outliers have very small distances. The $\epsilon$-satisfaction is different from the confidence of associations because the order is involved in the definition.

**Definition 3.2** (Interestingness). Given DD $\sigma = X\langle w_x \rangle \rightarrow B\langle w \rangle$ and the relation $r$, the support to $\sigma$, denoted by supp($\sigma$), is the ratio of the number of all tuple pairs satisfying $X\langle w_x \rangle B\langle w \rangle$ over the number of all tuple pairs of $r$. Let $\text{maxd}(B)$ be the maximal distance of attribute $B$ in $r$. The interestingness of $\sigma$ is defined to be

$$intr(\sigma) = \frac{\text{supp}(\sigma)}{\text{maxd}(B)}$$

This definition gives each discovered DD a number. The larger the number is, the more interesting the DD is. A DD discovered with higher support has higher interestingness. A DD discovered having a narrower rhs interval also has higher interestingness. Importantly, the definition implies that the DDs with fewer attributes on the lhs are more interesting because it is often the case that
the DDs with less attributes on the lhs are satisfied by more tuple pairs and therefore have more support than DDs with more attributes on the lhs.

Interestingness can have a tie in two cases: the case where the support is small and the rhs interval is narrow and the case where the support is large and the rhs interval is wide. Both cases are not interesting to the discovery and such a tie does not matter.

3.1 Outliers

Finally in this section, we show the connection between outliers, fully satisfied DDs and approximately satisfied DDs. Let \( f_1 = X(w) \rightarrow B(w_1) \) be a fully satisfied DD from a relation \( r \) and \( f_2 = X(w) \rightarrow B(w_2) \) be an \( \epsilon \)-satisfied DD. Obviously \( w_2 \leq w_1 \). The ratio

\[
rr = \frac{w_1 - w_2}{w_1}
\]

indicates the amount of interval width reduced by \( \epsilon \). If \( rr \) is smaller than \( \epsilon \), the chance for the existence of outliers for \( X(w) \rightarrow B(w_1) \) is very high. This becomes an effective mechanism for outlier detection. By applying this mechanism to data cleaning, possible outlier tuples can be identified, which makes it possible for the verification and cleaning work to be applied to the right point.

4 DD discovery

In this section, we present our DD discovery algorithm. With DDs, for any \( X(w_x) \) and \( B \), an interval \( w \) can always be found so that \( X(w_x) \rightarrow B(w) \) is satisfied. This property of DDs determines that the discovery of DDs is very different from the discovery of functional dependencies.

Our way of discovering DDs contains the processes of generating lhs, calculating the rhs interval \( w \), checking minimality and pruning.

To generate lhs of DDs, we assume that each attribute \( B \), with the maximal distance value \( maxd(B) \), has a user-selected interesting distance range \([0, ur(B)]\) \((ur(B) \leq maxd(B))\), and a sequence of distance intervals \( w_1, \ldots, w_{k-1}, w_k \) where \( w_i \) \((i = 1, \ldots, k - 1)\) and \( w_{i+1} \) are adjacent, \( right(w_{k-1}) = ur(B) \), and \( w_k = [ur(B), maxd(B)] \). That is, \( w_1, \ldots, w_{k-1} \) cover the whole user-selected interesting range and \( w_k \) includes all the remaining distances. For example, if the maximum distance interval of attribute \( Age \) is 55, and the interesting distance range of \( Age \) is \([0, 40]\), then the sequence of intervals of \( Age \) can be \([0, 5], [6, 10], \ldots, [36, 40], [41, 55]\). These intervals are called base intervals.

The sizes of the base intervals can be small or large. Smaller intervals will lead to more accurate DDs to be discovered. However, they also mean a larger \( k \), the number of intervals, which has exponential impact on the complexity of computation. Fortunately in most applications, only small distances are interesting. For example, an \( Age \) distance over 40 means that a young employee...
and a retiring employee are compared. Such a comparison would not be very useful in many cases. In the same way, in a data quality application in which typos are to be detected, an editing distance over 10 between two words may not be helpful. This is because such a 'large' distance should not be considered for typos. In the case of applications where large distances are more interesting, large interval sizes can be used to cover more small distances while small interval sizes can be used to cover large distances. We leave the user-selected interesting range and the interval sizes to be decided by domain experts.

4.1 Generating lhs

The process of generating lhs produces nodes for a lattice. A node is a tuple \((v,F(v),dds(v))\) where \(v = X\langle w\rangle\) is a differential function, \(F(v) = F(X\langle w\rangle)\) is the tuple pair partition for \(v\), and \(dds(v) = \{Y_1\langle w_{y1}\rangle B_1\langle w_1\rangle, \ldots, Y_k\langle w_{yk}\rangle B_k\langle w_k\rangle\}\) is the set of differential functions for both sides of the satisfied DDs \(Y_i\langle w_{yi}\rangle \rightarrow B_i\langle w_i\rangle\) such that \(w_i\) is a base interval of \(B_i\) and \(Y_i\langle w_{yi}\rangle \subseteq v\). The \(\subseteq\) symbol indicates that \(dds(v)\) will be carried from level to level. \(F(v)\) is used to test partition containment for satisfaction, \(dds(v)\) is used to detect reducible DDs.

Level-1 nodes are constructed by single attribute differential functions. Each attribute \(B\) and its base intervals \(w_1, \ldots, w_k\) form \(k\) differential functions \(B\langle w_1\rangle, \ldots, B\langle w_k\rangle\). The differential functions of all attributes form the first level nodes. Let \(v\) be such a node. We set \(dds(v) = \phi\) and calculate \(F(v)\) as defined in Equation (1).

A Level-\(i\) \((i \geq 2)\) node is joined from two Level-\((i - 1)\) nodes if they share \((i - 2)\) preceding single attribute differential functions and their tailing single attribute differential functions are of different attributes. For example, the node \(A\langle w_1\rangle B\langle w_2\rangle C\langle w_3\rangle D\langle w_4\rangle\) at Level-4 is joined from \(A\langle w_1\rangle B\langle w_2\rangle C\langle w_3\rangle\) and \(A\langle w_1\rangle B\langle w_2\rangle D\langle w_4\rangle\) at Level-3. For opposite examples, the join of any pair of the three Level-3 nodes \(A\langle w_1\rangle B\langle w_2\rangle C\langle w_3\rangle\), \(A\langle w_1\rangle B\langle w_2\rangle C\langle w_3\rangle\), and \(A\langle w_1\rangle C\langle w_2\rangle D\langle w_3\rangle\) will not produce a Level-4 node. A special case of this principle is that a Level-2 node is combined from two Level-1 nodes if they have different attributes. Figure 4.1 shows the lattice with three attributes \(A, B,\) and \(C\). Each of the fist two has two intervals and \(C\) has one interval.

![Figure 2: An example lattice](attachment:image.png)
Assume that the relation has $m$ attributes and attribute $A_i$ has $k_i$ intervals. Let $x = k_1 + \cdots + k_m$. The number of nodes in the lattice is $C(x, 1) + C(x, 2) + \cdots + C(x, m)$ where $C(x, k)$ is the combination of $k$ elements out of $x$ total elements.

We note that the lattice generated here is different from the Apriori lattice \[1\] used in FD discovery \[9\]. In the lattice for FD discovery, each edge represents a candidate FD, all nodes at a level has the same number of in-coming edges and the same number of out-going edges, and the last level has only one node. In our lattice for DD discovery, edges represent how nodes are generated, and they do NOT represent candidate DDs. The nodes at the same level have two in-coming edges showing the derivation but may have different number of out-going edges. The last level has multiple nodes in general. The candidate DDs are then derived from each of the nodes.

### 4.2 Calculating rhs interval $w$

The components of a node are derived from the joining nodes of the previous level. Let node $v = X\langle w_x \rangle$ be joined from the two nodes $v_1 = Y\langle w_y \rangle$ and $v_2 = Z\langle w_z \rangle$ at the previous level. Then $F(v) = F(v_1) \cap F(v_2)$, and $dds(v) = dds(v_1) \cup dds(v_2)$.

The candidate DDs of a node $v = X\langle w_x \rangle$ are generated using $v$ as the lhs and the attributes that are not in $v$ as the rhs. The intervals of the rhs attributes are represented by $w$ which are to be decided. Let $v = A\langle w_a \rangle B\langle w_b \rangle$ and $R = \{A, B, C, D\}$. Then the candidate DDs are $v \rightarrow C\langle w_c \rangle$, $v \rightarrow D\langle w_d \rangle$.

Deciding the rhs interval $w$ of a candidate DD follows Equation (3). If $\text{right}(w) > \text{ur}(B)$ where $\text{ur}(B)$ is upper limit of the user-selected interesting range of $B$, the DD is trivial. Otherwise, it is not. A non-trivial DD $f = X\langle w_x \rangle \rightarrow B\langle w_b \rangle$ is added to $\Sigma$, the set of all discovered DDs, if it is not implied by other DDs that are already in $\Sigma$. If $w$ equals to a specific base interval $w_i$, then $X\langle w_x \rangle$ is added to $dds(v)$, which will be used to detect reducible non-minimal DDs.

When approximate satisfaction is used with threshold $\epsilon$, Equation (3) is modified to the following to decide the interval $w$. Note that both formulae takes minimal.

\[
\text{left}(w) = \min\{\text{left}(w_i) \mid F(B\langle w_i \rangle) \in \mathcal{F}(B) \land c_1 = \text{True}\}
\]

\[
\text{right}(w) = \min\{\text{right}(w_i) \mid F(B\langle w_i \rangle) \in \mathcal{F}(B) \land c_1 = \text{true} \land c_2 = \text{true}\}
\]

where

\[
c_1 = F(B\langle w_i \rangle) \cap F(X\langle w_x \rangle) \neq \phi
\]

\[
c_2 = \frac{\sum_{j=1}^{k} |F(B\langle w_j \rangle) \cap F(X\langle w_x \rangle)|}{|F(X\langle w_x \rangle)|} < (1 - \epsilon)
\]

These equations are implemented in the following function where $F(B\langle w_i \rangle)$ is the $i$-th element of $\mathcal{F}(B)$.
Function 4.1. **FindRhs**($F(X\langle w_x \rangle), F(B),$ $\epsilon$) as interval:
1: $cnt = 0$;
2: for $i = k, \ldots, 1$ do
3:     $F = F(X\langle w_x \rangle) \cap F(B\langle w_i \rangle)$, $cnt + |F|
4:     if $\frac{|F(X\langle w_x \rangle)|}{|F(\langle w_i \rangle)|} > (1 - \epsilon)$: $r = right(w_i)$, break
5: end for
6: for $i = 1, \ldots, k$ do
7:     $F = F(X\langle w_x \rangle) \cap F(B\langle w_i \rangle)$, $cnt = |F|
8:     if $cnt > 0$: $l = left(w_i)$, break
9: end for
10: return $[l, r]$

### 4.3 Pruning

The lattice generation and $w$ calculation guarantee that all the single attribute differential functions on the lhs are with only base intervals and all rhs intervals are minimal. We discuss the rules of Lemma 2.3. The condition of rule (a) will never be met because $w_x$ is a base interval and the smaller interval $w_1$ does not exist. The condition $B\langle w_2 \rangle \geq B\langle w \rangle$ of rule (c) will never be met because for the same lhs in $X\langle w_x \rangle \rightarrow B\langle w \rangle$ and $X\langle w_x \rangle \rightarrow B\langle w_2 \rangle$, we always find $B\langle w \rangle$, not $B\langle w_2 \rangle$. The condition $B\langle w_2 \rangle \geq B\langle w \rangle$ of rule (d) needs to be specialized to $B\langle w_2 \rangle = B\langle w \rangle$ because the intervals on the lhs can only be base intervals. Thus in minimality check, we only need to consider rules (b) and (d) with the condition of (d) changed to $B\langle w_2 \rangle = B\langle w \rangle$.

Following the lattice generation and the discussion above, we have the following lemma.

**Lemma 4.1.** If $X\langle w_x \rangle \rightarrow B\langle w \rangle$ where $w$ is a base interval, all nodes containing $X\langle w_x \rangle B\langle w \rangle$ can be pruned.

**Proof:** Consider DD $f_1 = X\langle w_x \rangle B\langle w \rangle Y\langle w_y \rangle \rightarrow C\langle w_c \rangle$. By Lemma 2.3(d), $f_1$ can be reduced to $f_2 = X\langle w_x \rangle Y\langle w_y \rangle \rightarrow C\langle w_c \rangle$. $f_2$ involves less attributes and it must have been checked at a previous level. Consequently nodes like $X\langle w_x \rangle B\langle w \rangle Y\langle w_y \rangle$ can be pruned from the lattice. $\square$

We note that we can use $X\langle w_x \rangle \rightarrow B\langle w \rangle$ to prune nodes containing $X\langle w_x \rangle B\langle w \rangle$, but we cannot avoid computing $w_2$ for $X\langle w_x \rangle A\langle w_a \rangle \rightarrow B\langle w_2 \rangle$. The reason is that $w_2$ may be narrower than $w$ and if this is the case, $X\langle w_x \rangle A\langle w_a \rangle \rightarrow B\langle w_2 \rangle$ is a true minimal DD. This is different from the case in FD discovery [6]. In FD discovery, if $X \rightarrow B$, the check of $X A \rightarrow B$ is not necessary. Furthermore, if every attribute $B$ in the node $X$ is determined non-trivially by a subset of $X$, the node can be pruned. The two FD discovery pruning rules do not apply to DD discovery.

We also use the support to a node $X\langle w_x \rangle$ to prune the nodes from the lattice. The support is the ratio of tuple pairs satisfying $X\langle w_x \rangle$ over all tuple pairs and is calculated by $supp(X\langle w_x \rangle) = \frac{|F(X\langle w_x \rangle)|}{|\mathbb{R}|}$. If the support is lower
than a threshold \(\delta\), no DD will be calculated for the node. The reason for using a support threshold is that we do not want DDs satisfied by only a few tuple pairs to be discovered. Such DDs do not reflect typical relationships.

The following lemma is used in many association rule mining works.

**Lemma 4.2.** Given a support threshold \(\delta\) and a node \(X\langle w \rangle\), if \(\text{supp}(X\langle w \rangle) < \delta\), \(\text{supp}(X\langle w \rangle \cap Y\langle w_y \rangle) < \delta\) for any \(Y\langle w_y \rangle\). All nodes containing \(X\langle w \rangle\) can be pruned.

The lemma is correct because \(F(X\langle w \rangle \cap Y\langle w_y \rangle) = F(X\langle w \rangle) \cap F(Y\langle w_y \rangle)\), \(|F(X\langle w \rangle \cap Y\langle w_y \rangle)| < |F(X\langle w \rangle)|\), so \(\text{supp}(X\langle w \rangle \cap Y\langle w_y \rangle) < \text{supp}(X\langle w \rangle)\).

---

**Algorithm 1 MinDD**

**Inp:** relation \(r\) on schema \(R\), \(W_A = [w_1, \ldots, w_k]\) for every attribute \(A \in R\), satisfaction threshold \(\epsilon\) and support threshold \(\delta\)

**Outp:** the set of minimal DDs \(\Sigma\)

1. Calculate \(F(A\langle w \rangle)\) for all \(A \in R\) and all \(w \in W_A\); Calculate \(F(A)\) for all \(A \in R\).
2. Create level-1 nodes: for each \(A \in R\) and for each interval \(w \in W_A\), \{let \(v = A\langle w \rangle\); if \(|F(A\langle w \rangle)| > 0\); add node \((v, F(v) = F(A\langle w \rangle), \text{dds}(v) = \emptyset)\) to \(L_1\) \}
3. for level \(i = 1, \ldots, |R|\) do
   4. for node \(v \in L_i\) do
      5. if \(\exists Y\langle w_y \rangle \in \text{dds}(v)\langle Y\langle w_y \rangle \in v\): \(v\) is reducible, remove \(v\) from \(L_i\), continue
      6. for attribute \(B \in R\) and \(B \notin \text{attr}(v)\) do
         7. let candidate DD be \(f = v \rightarrow B\langle w \rangle\)
         8. // calculate rhs:
         9. \(w = \text{FindRhs}(F(v), F(B), \epsilon)\) [Function 4.1]
         10. if \(w \not\subseteq [0, wR(B)];\) continue;
         11. if \(\text{ChkImply}(f, \Sigma) = \text{false}\); add \(f\) to \(\Sigma\)
         12. if \(w\) is a base interval, add \(vB\langle w \rangle\) to \(\text{dds}(v)\).
   13. end for attribute \(B\)
   14. end for node \(v\)
   15. // build nodes for next level
   16. let \(v_1 = X\langle w_x \rangle A\langle w_a \rangle\) and \(v_2 = X\langle w_x \rangle C\langle w_c \rangle\) \((A \neq C)\) be nodes in \(L_i\)
   17. \(v = X\langle w_x \rangle A\langle w_a \rangle C\langle w_c \rangle\), \(F(v) = F(v_1) \cap F(v_2)\), \(\text{dds}(v) = \text{dds}(v_1) \cup \text{dds}(v_2)\)
   18. if \(|F(v)| \geq \delta\) (support pruning): add \((v, F(v), \text{dds}(v))\) to \(L_{i+1}\)
   19. end for level \(i\)
20. return \(\Sigma\)

---

### 4.4 The Algorithm

Our algorithm is called *MinDD* and is given in Algorithm 1. It builds the lattice in the breadth-first manner and for each node to be added to the lattice,
it calculate DDs for the nodes. Line 1 of the algorithm calculates tuple pair partitions for differential functions and for single attributes. Line 2 builds the first level nodes. Line 5 uses $dds(v)$ to check reducibility and to prune nodes based on Lemma 4.1 from the current level. Lines 7-12 derive candidate DDs and determines their rhs intervals. In Line 10, the algorithm uses the user-selected interesting range of the rhs attribute $B$ to filter the found DD: if it is not in the interesting range, it is ignored. Line 11 uses the function $ChkImply()$ to check to see if the DD is implied by other DDs in $\Sigma$. This function will be described in detail later. Line 12 maintains the differential functions of satisfied DDs of the node. Lines 15-17 builds nodes for the next level and Line 18 prunes the node if it does not have enough support.

The complexity of this algorithm is analyzed as follows. The lattice has $2^{\Sigma_w}$ nodes where $\Sigma_w$ is the sum of the number of intervals of all attributes. The partition of a node is the product of the partitions of the two participating parent nodes. This takes $(|r|^2)^2$ operations where $|r|^2$ is the number of tuples of the distance relation of $r$. Thus, the complexity of the algorithm is $O((|r|^2)^2 \Sigma_w)$.

4.5 Detection of implication of DDs

We design a tree structure, called a DD-tree, to store all DDs having the same rhs differential function, and use the tree to check whether a DD is implied. The root node of the tree is the rhs $B\langle w_B \rangle$ of the DDs. Other nodes are single attribute differential functions in the lhs of these DDs. Child nodes of a node are sorted by their attributes and their intervals. A path $B\langle w_B \rangle/A_1\langle w_1 \rangle/A_2\langle w_2 \rangle/\cdots/A_j\langle w_j \rangle$ represents the DD $A_1\langle w_1 \rangle A_2\langle w_2 \rangle \cdots A_j\langle w_j \rangle \rightarrow B\langle w_B \rangle$.

Definition 4.1 (path prefix). A path $B\langle w_B \rangle/A_1\langle w_1 \rangle/A_2\langle w_2 \rangle/\cdots/A_j\langle w_j \rangle$ is a prefix of path $B\langle w_B \rangle/A_1\langle \tilde{w}_1 \rangle/A_2\langle \tilde{w}_2 \rangle/\cdots/A_k\langle \tilde{w}_k \rangle$ if $j < k$ and for each $i = 1 \cdots j$ ($A_i\langle w_i \rangle \succeq A_i\langle \tilde{w}_i \rangle$).

Lemma 4.3. A DD $f_2$ is implied by DD $f_1$ if the path of $f_1$ is a prefix of the path of $f_2$.

This lemma is correct because of Lemma 2.3.

Definition 4.2 (non-redundant). A DD tree is non-redundant if no node has two or more child nodes of the same differential functions.

Assume that a hash table $h(B\langle w_B \rangle, tr)$ is created to store the rhs differential functions like $B\langle w_B \rangle$ and their trees. With the hash table, the following function checks whether the given DD is implied by previously found DDs in the way specified by Lemma 2.3 where the $Combine()$ function will be introduced later.

Function 4.2. $ChkImply(DD f)$ as bool:
1. assume $f = A_1\langle w_{A_1} \rangle \cdots A_j\langle w_{A_j} \rangle \rightarrow B\langle w_B \rangle$
2. let $h = h(B\langle w_B \rangle, tr)$ be the hash table.
3: retrieve tr using $B\langle w_B \rangle$ from $h$.
4: convert f to path $p = B\langle w_B \rangle/A_1\langle w_{A_1} \rangle/\cdots/A_j\langle w_{A_j} \rangle$
5: if tr contains path $q$ s.t. $q$ is a prefix of $p$ then
6: return false
7: else
8: add $A_1\langle w_{A_1} \rangle/\cdots/A_j\langle w_{A_j} \rangle$ to the root of tr,
   tr is now redundant, Combine(tr,p).
9: return true
10: end if

The Combine() function which is defined in Procedure 4.1 implements Lemma 2.4. Two child nodes $v_1 = A\langle w_1 \rangle$ and $v_2 = A\langle w_2 \rangle$ of a node is combinable if $v_1$ and $v_2$ have identical child trees. The child trees of node $v$ is denoted by children($v$). The combination extends the interval of $v_1$ to $w_1 + w_2$ and deletes $v_2$.

Procedure 4.1.

Combine(dd-tree tr, path $p = B\langle w_B \rangle/A_1\langle w_{A_1} \rangle/\cdots/A_j\langle w_{A_j} \rangle$):
1: locate the last node $v_1 = A_j\langle w_{A_j} \rangle$ of $p$ in tr.
2: for $v_1 = A_1\langle w_{A_1} \rangle, \cdots, A_j\langle w_{A_j} \rangle$ do
3: let $x$ be parent of $v_1$
4: find another child node $v_2 = A\langle w_2 \rangle$ of $x$
5: if $v_2 == null$ or children($v_1$)! = children($v_2$) break
6: replace $A\langle w_A \rangle$ by $A\langle w_1 + w_2 \rangle$; delete $v_2$.
7: end for

As an example, given DDs
\[
f_1 = A\langle w_1 \rangle C\langle w_2 \rangle D\langle w_3 \rangle \rightarrow B\langle w \rangle,
\]
\[
f_2 = A\langle w_1 \rangle C\langle w_3 \rangle D\langle w_3 \rangle \rightarrow B\langle w \rangle,
\]
\[
f_3 = A\langle w_1 \rangle C\langle w_3 \rangle D\langle w_4 \rangle \rightarrow B\langle w \rangle
\]
checked by ChkImply() in order, the tree $tr$ with root $B\langle w \rangle$ have the following two child paths $A\langle w_1 \rangle/C\langle w_2 + w_3 \rangle/D\langle w_3 \rangle$ and $A\langle w_1 \rangle/C\langle w_3 \rangle/D\langle w_4 \rangle$.

We use the following lemma to show that DDs with implication relationship will not be put into different DD-trees.

Lemma 4.4. If DDs $f_1$ and $f_2$ are discovered by Algorithm 4.1 and $f_2$ is implied by $f_1$, both DDs will be directed to the same DD-tree.

Proof: If $f_1 = X\langle w_x \rangle \rightarrow B\langle w \rangle$, based on previous discussion (Lemma 2.3) and Section 4.3, DD implication happens in the following cases. (1) $X\langle w_x \rangle B\langle w \rangle A\langle w_a \rangle \rightarrow C\langle w_c \rangle$ is reducible. (2) $f_2 = X\langle w_x \rangle A\langle w_a \rangle \rightarrow B\langle w \rangle$ is implied.

Case (1) does not happen because the algorithm uses dds($v$) to filter such DDs. Case (2) is the only way DD implication happens. Because $f_2$ and $f_1$ have the same rhs, they are put into the same DD-tree. □

The lemma implies that the implication detection is complete and no implied DDs will be output by the algorithm.
5 Sampling and errors

The discovery algorithm has the complexity factored by $|r|^4$ (see subsection 4.3) for relation $r$. This indicates that the discovery from large data sets is not possible. For this reason, we use sampling when data sets get large.

The sample size has to be determined before sampling can be conducted. We note that sampling for dependency discovery is different from sampling for many statistical studies and the difference is the number of results that can be observed from a sample. For example, in the study of the percentage of heads to appear in coin toss, each toss gets a result (true or false). Thus if in a sample a coin is tossed for 380 times, 380 results will be observed. For such problems in statistics, well-developed formulas for determining the sample size are available.

In dependency discovery, however, the satisfaction must be calculated based on the whole data set, not on individual tuples. In other words, if a sample contains 380 tuples drawn from the original data set, only one result (true or false) about the satisfaction of a DD can be observed. Because of this difference, sample sizes in dependency discovery must be carefully studied.

Consider a data set $r$, its distance relation $\overline{r}$, and a candidate DD $f$. Assume that $\overline{r}$ has $N$ total tuples and $M_f$ of them support $f$. The sample must be taken without replacement because each tuple in $\overline{r}$ is used only once in testing the satisfaction of $f$. When a sample $\overline{r}_s$ of $N_s$ tuples are drawn randomly from $\overline{r}$ with no replacement, the probability for $\overline{r}_s$ to have exact $k_f$ supporting tuples follows the hypergeometric distribution

$$P(N,M_f,N_s,k_f) = \binom{M_f}{k_f} \binom{N-M_f}{N_s-k_f} \div \binom{N}{N_s}$$

where the notation $\binom{x}{y}$ is the number of $y$-combinations from $x$ elements.

The sampling process causes change to: (1) the support to the DD $f$ and (2) the rhs interval of $f$. We firstly analyze the change to the support.

Given a support threshold $\theta$, two types of erroneous DDs, namely missed-wanted DDs and found-unwanted DDs, may happen because of sampling. A wanted DD is one with a support equal to or higher than the threshold $\theta$ in $\overline{r}$ (before sampling) and an unwanted DD is one with a support lower than the threshold $\theta$ in $\overline{r}$. A DD $f$ is missed-wanted (missed for short) if the number of its supporting tuples in $\overline{r}$ is more than the threshold ($M_f \geq \theta N$), but the number of its supporting tuples in the sample $\overline{r}_s$ is less than the threshold ($< \theta N_s$). An example of such a DD is $f_2$ in Table 2. The probability for a wanted $f$ to be missed is

$$P(N,M_f,N_s,k_f < \theta N_s) = \sum_{k_f=0}^{\theta N_s-1} P(N,M_f,N_s,k_f)$$

We like this probability to be small.

From this equation, we can derive the sample size $N_s$ if we set the probability to a specific value. An explicit formula for getting the sample size $N_s$ seems not
Table 2: Missed DDs and unwanted DDs. (b) is a sample from (a). The set of DDs found from (a) is listed in the middle and the set of DDs from (b) are listed below (b).

(a): \( \bar{r} \)

| A | B |
|---|---|
| 0 | 1 |
| 0 | 1 |
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |

DDs on \( \bar{r} \): (\( \theta = 1/3 \))

\[ f_1 = A(0) \rightarrow B(1) \]
\[ f_2 = A(2) \rightarrow B(3) \]

(b): \( \bar{r}_s \) (sample rate=1/3)

| A | B |
|---|---|
| 0 | 1 |
| 1 | 2 |

DDs on \( \bar{r}_s \): (same \( \theta \))

\[ f_1 = A(0) \rightarrow B(1) \]
\[ f_3 = A(1) \rightarrow B(2) \]

\( f_2 \) missed. \( f_3 \) is unwanted.

easy to get. However, it is easy to program a calculator to determine \( N_s \) given all other parts of the formula.

Assume that \( \bar{r} \) has \( N = 10000 \) tuples, \( M_f = 10 \) of them support \( f \), and \( \theta = 0.0005 \). If we set the chance for \( f \) to be missed to \( P(N, M_f, N_s, k_f < \theta N_s) = 0.05 \), the sample size \( N_s \) needs to be 3941. If we choose the smallest threshold: \( \theta = 0.0001 \) for the specific \( N \), for the same \( M_f \) and same \( P \), the sample size must be 2588. The smaller threshold leads to smaller sample size.

If we reduce the number of supporting tuples \( M_f = 1 \) and keep \( \theta = 0.0001 \) and \( P(N, M_f, N_s, k_f < \theta N_s) = 0.05 \), the sample size must be 9501. Obviously the proportion of supporting tuples and the threshold control the sample size. The smaller the number of supporting tuples or the larger the threshold, the larger the sample size has to be.

A **found-unwanted** (unwanted for short) DD \( f \) is opposite to a missed wanted DD. \( f \) is found-unwanted if it has \( M_f < \theta N \) number of supporting tuples in \( \bar{r} \) but has \( \theta N_s \) or more supporting tuples in the sample \( \bar{r}_s \). An example of a found-unwanted DD is \( f_3 \) in Table 2. The probability for this to happen is

\[
P(N, M_f, N_s, k_f \geq \theta N_s) = 1 - \sum_{k_f=0}^{\theta N_s-1} P(N, M_f, N_s, k_f) \]  \( (6) \)

We like this probability to be small. By comparing Formulas (5) and (6), we find that the two are complementary. This implies that irrespective of the sample size (small or large), the chance of one of the errors is high as \( M_f \) in both cases is close and small. Thus, the analysis of support change caused by sampling does not help with the determination of the sample size.

The above of analysis is on the possibility for one DD to become an error. The total number of erroneous DDs discovered from a sample relates the total number of DDs from the distance relation, their supports, and their probabilities. Let \( \Sigma_a \) (resp. \( \Sigma_b \)) be the set of all wanted DDs (resp. unwanted DDs) on the distance relation \( \bar{r} \), \( f_i \) a DD in \( \Sigma_a \) or \( \Sigma_b \) with \( M_{f_i} \) supporting tuples in \( \bar{r} \) and \( k_{f_i} \) supporting tuples in the sample, and \( \theta \) the support threshold. The number \( E_m \) of DDs missed and the number \( E_{uw} \) of unwanted DDs found from
the sample are given in Formula (7) below.

\[
E_m = \sum_{f_i \in \Sigma_a} P(N, M_{f_i}, N_s, k_{f_i} \geq \theta N_s, M_{f_i} < \theta N), \quad M_{f_i} < \theta N
\]

\[
E_{uw} = \sum_{f_i \in \Sigma_b} P(N, M_{f_i}, N_s, k_{f_i} < \theta N_s, M_{f_i} \geq \theta N), \quad M_{f_i} \geq \theta N
\]

(7)

Note that the \(\Sigma_b\) contains DDs with low support and such DDs are much more in number than the wanted DDs in \(\Sigma_a\). Consequently \(E_{uw} \gg E_m\). This has been confirmed by our experiments.

A further type of error caused by sampling is the reduction of rhs intervals. This can be demonstrated by the data in Table 3. Part(a) is the distance relation. From the relation, \(f_1\) and \(f_2\) are discovered and \(f_2\) is trivial. Parts(b) and (c) are two samples from (a) with the same sample rate of 1/3. The DDs found from the samples are listed below them. The trivial DD \(f_2\) becomes non-trivial \(f_3\) in (b) and non-trivial \(f_4\) in (c).

### Table 3: Reduction of rhs intervals

| (a): \(\bar{r}\) |  |
|---|---|
| \(A\) | \(B\) |
| 0 | 2 |
| 0 | 2 |
| 0 | 2 |
| 1 | 2 |
| 1 | 1 |
| 1 | 0 |

DDs on \(\bar{r}\):
- \(f_1 = A(0) \rightarrow B(2)\)
- \(f_2 = A(1) \rightarrow B(0, 2)\)
- \(f_2\) is trivial

| (b): sample1 | (c): sample2 |
|---|---|
| \(A\) | \(B\) |
| 0 | 2 |
| 1 | 0 |
| 1 | 1 |

DDs on sample1:
- \(f_1 = A(0) \rightarrow B(2)\)
- \(f_3 = A(1) \rightarrow B(0)\)

DDs on sample2:
- \(f_1 = A(0) \rightarrow B(2)\)
- \(f_4 = A(1) \rightarrow B(1)\)

From the example, we can use \(f_3\) and \(f_4\) to guess \(f_2\) by combining the intervals of the DDs. After the combination, we get \(\hat{f}_2 = A(1) \rightarrow B(0, 1)\). This is a way to guessing the true interval. We define the following operation.

**Definition 5.1 (DD combination).** Given two intervals \(w_1\) and \(w_2\), the combination of \(w_1\) and \(w_2\), denoted by \(w_1 \sqcup w_2\), is the minimal interval enclosing both \(w_1\) and \(w_2\).

Given sibling DDs \(f_a = X(w) \rightarrow A(w_a)\), \(f_b = X(w) \rightarrow A(w_b)\), \(\ldots\), \(f_k = X(w) \rightarrow A(w_k)\), the combination of the DDs, denoted by \(f_a \sqcup f_b \sqcup \cdots \sqcup f_k\), is defined to be \(X(w) \rightarrow A(w_a \sqcup w_b \sqcup \cdots \sqcup w_k)\). \(\square\)

Following the combination operation, as the number of samples increases, the interval of \(\hat{f}_2\) becomes closer to that of \(f_2\). We now analyze how the number of samples affect the errors in guessing the right interval.

A tuple is allowed to be included in a sample only once and the chance for this to happen is the sample rate \(\rho = \frac{N_s}{N}\) where \(N_s\) is the sample size which is the same for all samples. For each sample, a tuple is either in or out. A tuple may be included in many samples. Thus, for a tuple to be included in \(ks\) of \(ns\)
samples follows the binomial distribution \( B(ks) = \binom{ns}{ks} \varrho^{ks} (1-\varrho)^{ns-ks} \). The probability for a tuple to be included in one or more samples is \( B(ks \geq 1) = 1 - \binom{ns}{0} \varrho^0 (1-\varrho)^{ns-0} = 1 - (1-\varrho)^{ns} \). From this we have

\[
n_x = \frac{\ln(1-B)}{\ln(1-\varrho)}
\]

where \( B \) is short handed notation for \( B(ks \geq 1) \). If \( \varrho = 10\% \) and \( B = 90\% \), \( ns \) needs to be 22. If we reduce the sample rate to \( \varrho = 1\% \), \( ns \) must be 230.

This formula indicates that when the computation is possible, a large sample rate should be used to reduce the number of samples needed.

When the DDs from multiple samples are combined, the probabilities of missed and unwanted DDs will change. The chance for a wanted DD \( f \) to be missed from a sample is \( P_m = P(N,M_f,N_s,k_f < \theta N_s) \) by Formula (5). The chance of missing \( f \) in all the \( ns \) samples is \( P_m^{ns} \). It is easy to see that \( P_m > P_m^{ns} \) if \( ns > 1 \). The more samples are used, the less possible that \( f \) is missed.

Similarly, by Formula (6), the chance of finding an unwanted DD \( f \) in all samples is \( P_{uw} = 1-P_m^{ns} \). As the number \( ns \) of samples increases, the probability \( P_{uw} \) for \( f \) to be discovered gets larger.

After DDs are discovered from samples and are combined, we use three filters to remove the ones that are possibly erroneous. These filters are (1) the support, (2) the ratio of the number of the samples from which a DD is found over the total number of samples, and (3) the ratio of the average interval width in samples over the combined interval width of all samples. Our experiments show that the second filter can best minimize the errors and next one is the third filter and the support does not work well. The details will be shown in the experiments section.

The above sampling analysis is based on the distance table \( \bar{r} \). Computing all tuples of \( \bar{r} \) from the raw data \( r \) is too expensive and we need to avoid this. We recall the way in which \( \bar{r} \) is computed. Assume that each tuple of \( r \) has an index starting from 0, the tuples are ordered by the index, and \( n = |r| \). The first tuple of \( r \) is paired with all its following tuples to generate the first \((n-1)\) tuples of \( \bar{r} \). Then the second tuple of \( r \) is paired with its following tuples of \( r \) to compute the next \((n-2)\) tuples of \( \bar{r} \). After the \( x \)-th tuple of \( r \) is paired with its following tuples, the total number \( i \) of tuples generated is equivalent to the area of a quadrilateral with a pair of parallel edges and two neighboring right angles. Thus \( i = [(n-1) + (n-x-1)] \times x/2 \). To determine the \( x \)-th and \( y \)-th
tuples of \( r \) for the \( i \)-th tuple of \( r \), we use

\[
x = \text{int}\left(\frac{(2n-1) - \sqrt{(2n-1)^2 - 8i}}{2}\right)
\]

\[
d_i = i - \frac{(2n-1-x) * x}{2}
\]

if \((d_i > 0)\) : \( y = x + d_i \);

else : \{ \( x = x - 1 \); \( y = (n-1) \); \}

(8)

where \( \text{int}(\) \) is the truncate function, \( d_i \geq 0 \), the indexes \( x \) and \( y \) start from 0, and the index \( i \) starts from 1. Using the formula, only the distances of tuple pairs that are drawn to be in the sample need to be computed.

6 Experiments

Our experiments are done on a laptop computer with Intel i5-2520M CPU@2.5GHz, 8GB of main memory, and Windows 7 OS. The programming language used in the implementation is Java with JDK 1.7.

We use the experiments to demonstrate the following points.

(1) How our algorithm performs to the change to the number of tuples and the number of attributes in data.

(2) How the parameters, support, satisfaction threshold \( \epsilon \) and the interestingness, affect the results and efficiency performance.

(3) How sampling affects the DDs found in comparison to non-sampling cases.

6.1 Data sets and distance functions

We use three data sets, the DBLP, the US Census data, and the US Airline data. (1) The DBLP data is the publication reference data in XML obtained from [http://dblp.uni-trier.de/xml/]. This data is then transformed to the relational format having 6 columns namely the type (journal or conference), number of authors, author names, title, journal/conference name, and year. The volume of data is huge. We downloaded a portion of the data containing 40,000 tuples. (2) The Adult data is the US Census data obtained from [http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/] which has 15 columns and 32,000 tuples. (3) The US Airline data is about the airline on-time running statistics. The data was obtained from [http://www.transtats.bts.gov]. After the columns containing large amount of null values and repeating values are removed, the data has 20 columns covering dates, airlines and flights, departure and destination airports, and on-time running information. The data has about 500,000 tuples.

The distance functions \( d_X(t_1, t_2) \) for the attributes of the data sets are important to the discovery. Different differential functions lead to different computation sizes, different DDs discovered, and different amount of time used in the discovery. Table 4 lists the distance functions used for the attributes of
Table 4: Number of intervals of data sets

| Attr    | \(d_A(t_1, t_2)\) | nIntv |
|---------|---------------------|-------|
| Type    | wordDif            | 2     |
| Title   | wordDif            | 31    |
| NAuth   | numeDif            | 11    |
| Authors | wordDif            | 23    |
| Forum   | wordDif            | 18    |
| Year    | numeric            | 43    |

| Attr    | \(d_A(t_1, t_2)\) | nIntv |
|---------|---------------------|-------|
| Age     | numeDif/5           | 14    |
| WorkClass | hierDist       | 6     |
| Wedge   | numeDif/1000       | 101   |
| Education | hierDist      | 15    |
| EducNumb | numeDif          | 15    |
| Marital | wordDif            | 3     |
| Occupa | wordDif            | 2     |
| ...    | ...                | ...   |

| Attr    | \(d_A(t_1, t_2)\) | nIntv |
|---------|---------------------|-------|
| FlightNumb | Bool           | 31    |
| DepTime | numeDif            | 12    |
| DepDelay | numeDif/15       | 47    |
| Distance | numeDif/100    | 32    |
| ArrTime | numeDif            | 12    |
| ...    | ...                | ...   |

the data sets, and the number of intervals (nIntv) for the attributes. In the table, ‘wordDif’ means that the distance is the number of different words in the two tuples for the attribute, e.g., the distance between ‘Hello World’ and ‘Hello Helen’ is 2. ‘numeDif/5’ means that the distance is the difference of numeric values of two tuples for the attribute divided by 5. ‘hierDis’ indicates that the distance is the number of edges of the shortest path between the two nodes in the taxonomy hierarchy of the attribute.

If the taxonomy for WorkClass is WorkClass\(\langle\text{neverWorked}\rangle\)(\langle\text{worked}\rangle\langle\text{withPay}\rangle\langle\text{withoutPay}\rangle) where the notation \(A(B)\) represents that \(B\) is a child of \(A\), the distance between \text{neverWorked} and \text{withPay} is 3. ‘Bool’ indicates that the distance is 0 (same) or 1 (different).

We understand that there are many possible ways to define the distance functions. The domain expertise and computation capabilities need to be taken into consideration when the functions are designed.

### 6.2 Time performances

We implemented our algorithm called the **Lattice algorithm** and also the algorithm proposed in [16] called the **Split algorithm** (we give it such a name because it splits the search space). We note that the Lattice and the Split algorithms find different DDs as shown in detail in the related work section.

The comparison of the time performances of the two algorithms is not very meaningful. The reason for including the Split algorithm in the experiments is to show our respect to existing work.

Figure 3 shows the experiment results. Figure 3(a) and (b) are about the
time (in seconds) used for DD discovery versus the data size in the number of tuple pairs of the data divided by 1000. Part (a) shows the time used by the two algorithms on the DBLP data. The line ‘Split-dblp’ for the Split algorithm is very sensitive to data size (and even more to the number of attributes, as shown in (d)), and used more time and ran out of memory very soon. Because of this, it was not tested against other data sets. Part (b) is the time performance of our Lattice algorithm on DBLP, Adult, and USair data. Because the low number of attributes of the DBLP data, the line ‘Lat-Dblp’ is almost flat. For the other two data sets, the lines are close to straight.

We notice that the data size change causes another complication. As the data size increases, the number of attribute intervals also increases (Figure 3(c)). The increased number of intervals has impact on the time performance. This means that the time-size performance in (a) and (b) also includes the effect caused by some increase of the number of intervals. The number of intervals gets stable when the size reaches about ‘30’.

The relationship between the time and the number of attributes is shown in Figure 3(d). The data size was fixed to ‘80’ in the experiments for the chart. The x-axis is labeled with ‘nIntv’ which means the sum of the number of intervals of all attributes. The reason for showing the number of intervals instead of the number of attributes is that the former has the actual time performance impact. An attribute with 40 intervals has much more impact on the performance than 5 attributes each with only 2 intervals. In the chart, when the number of intervals
gets to more than 100, the performance gets worse very quickly for the Lattice algorithm. We note that the ‘Lat-dblp’ line is shorter because the data set has less total number of intervals for 6 attributes than other data sets. At the same time, when the total number of intervals exceeds 70, ‘Split-dblp’ ran out of memory.

6.3 The effect of the parameters

The implementation uses three parameters, namely the number of interesting intervals, the support, and the satisfaction threshold as input to experiment cases for our Lattice algorithm.

Setting a threshold for the number of interesting intervals for an attribute means to reduce the total number of intervals in the calculation. The effect of this is already shown in Figure 3(c).

As in data mining, the effect of support has significant impact on performance. Setting a minimal support threshold can change many non-computable cases to computable. We did an analysis using our Lattice algorithm and the result is shown in Figure 4(a) and (b) where ‘nDDs’ means the number of DDs discovered from data. From Figure 3(c), we know that the total number of intervals for the USair data is much larger. This leads to much more DDs found from the data too.

Setting an approximate satisfaction threshold (less than but close to 100%) has two effects. One is that the number of non-trivial DDs in the output is increased. The reason is that some DDs are trivial in the case of full satisfaction and become non-trivial in the case of approximate satisfaction. The other effect is that the time used to find approximately satisfied DDs is less than that for fully satisfied DDs. The reason behind this is that more discovered DDs cause more pruning. These two effects are shown in Figure 4(d) and (c) respectively. The axis ‘sat-thres’ means satisfaction threshold and when it is 1, the satisfaction is full.

6.3.1 Data quality problems

We run experiments on the adult data with two values for the approximate satisfaction threshold: $\epsilon = 1$ (full satisfaction) and $\epsilon = 0.95$ (approximate satisfaction) while all other parameters, including the support, are fixed. We then compare the results. For each fully satisfied DD $f_1 = X \langle w \rangle \rightarrow A \langle w_1 \rangle$ and its approximately satisfied partner DD $f_2 = X \langle w \rangle \rightarrow A \langle w_2 \rangle$, we calculate the ratio $\text{ratio} = \frac{w_2}{w_1}$ which indicates the amount left after the shrink. This ratio is then used to rank all fully satisfied DDs. DDs with low ratios mean that by excluding a very small portion of tuple pairs (5%), the rhs intervals of the DDs become much smaller.

Table 5 gives some examples of the DDs. The first column of the table is the support of the fully satisfied DDs, the second column is the ratio, and the third column is $f_2$. We choose to explain the first two DDs. The first DD in the table says that when satisfaction threshold is reduced from 1 to .95, the interval of
Table 5: Interval shrink when approximate threshold is set to .95.

| support | ratio | approximate DD          |
|---------|-------|--------------------------|
| 26.52   | 50.00 | $hrsWk(0) \rightarrow eduNum(0,6)$ |
| 100.00  | 33.33 | $capLoss(0) \rightarrow capGain(0)$ |
| 18.52   | 75.00 | $relat(0) \rightarrow marital(0,2)$ |
| 9.79    | 42.85 | $occup(0) \rightarrow eduNum(0,5)$ |

$eduNum$ is reduced by 50%. That is, 5% of tuple pairs spreads their $eduNum$ values in $[6,12]$. We see that these 5% of tuples are outliers. After excluding the outliers, most people who work for the same number of hours have an education difference no more than 6. This rule is supported by 26% of total tuple pairs.

The second one says that for 95% of people pairs, if they make the same amount of capital loss, they make the amount of gain. Only 5% of people (outliers) are different. This rule is fully supported by all pairs.

6.4 Errors of sampling

Sampling enables the discovery problems that are impossible to compute to become computable and plays an important role in data mining and data analysis. The aim of experiments here is to compare the DDs discovered from the whole data with the DDs found from the samples to analyze the errors caused by
Given a relation \( r \), a sample size \( N_s \), and the number of samples \( n_s \), following Formula (8) repeatedly, we draw \( n_s \) samples, \( s_1, \ldots, s_{n_s} \), from \( r \). Let \( D^r_{\theta} \) denote the set of DDs found from the distance relation \( \bar{r} \) directly with the support threshold \( \theta \), and \( D^s_{\theta} \) the set of DDs discovered from the sample \( s_i \). Let \( D^\theta \) denote the set of DDs combined from the DDs discovered from all the samples following Definition 5.1.

The number of missed DDs is \( |D^r_{\theta} - D^s_{\theta}| \). The relative error rate of missed DDs is

\[
err_m = \frac{|D^r_{\theta} - D^s_{\theta}|}{|D^r_{\theta}|} \quad (9)
\]

Similarly, the number of unwanted DDs is \( |D^s_{\theta} - D^{\bar{r}}_{\theta}| \) and the error rate is

\[
err_{uw} = \frac{|D^s_{\theta} - D^{\bar{r}}_{\theta}|}{|D^{\bar{r}}_{\theta}|} \quad (10)
\]

We now present the experiments we conducted to analyze some properties of sampling with the DBLP data.

The first experiment analyzes how the data sizes and sample rates affect the number of DDs discovered and the results are shown in Figure 5. Part (a) of the figure shows that as data size increases, with the support threshold \( \theta = 0.5\% \), the number of DDs discovered from the unsampled data, \( D^r_{\theta} \), decreases. Part (b) shows that as the sample rate increases, the number of DDs discovered from the samples, \( D^s_{\theta} \), also decreases. We note that if zero support threshold is used, the number of DDs discovered from unsampled data and the samples will increases because the number of possible intervals will increase.

The second experiment is about errors of sampling rate and the results are shown in Figure 6. It shows how the error rates (Formulas (9, 10)) change with the sample rate. It shows that the error rate of missed wanted DDs is much lower than the found unwanted error rate. As we explained before, the number of unwanted DDs is much more than the number of missed DDs. The figure also shows that as the sample rate increases, the overall error rates (both missed and
unwanted) reduce too. Readers may feel confused as we said in Formulas (5 and 6) that the errors of missed wanted and found unwanted are complementary. We note that complementary formulas are about the error possibilities of individual DDs. What we present in this diagram is about the number of all DDs. We believe that the down trend of errors with the increasing sample rate is caused by the fact that the number of DDs decreases as the sample rate gets larger and that the errors in DDs are reduced even faster. This is the evidence for our conclusion that large sample size should be used if the computation is practical.

A further experiment is on the relationship between errors and the number of samples and this is shown in Figure 7. This experiment is done with fixed sample rate, fixed support and fixed data size. Parts (a) and (b) show that as the number of samples increases, the intervals of attributes are recovered better and the errors of missed wanted DDs get smaller. However, Part (c) shows that the increasing number of samples causes the errors of found unwanted DDs to grow.

Another experiment analyzes the relationship between the errors and the support and the result is shown in Figure 8. The experiment is done with the support threshold of 0.5%. The errors in terms of missed wanted and found unwanted DDs are derived and then these erroneous DDs are grouped by these support. Part (a) shows that although any unwanted DDs with the support between 0 and 0.5% can be found as errors, most of errors are from the DDs whose support is close to the threshold. In the same way, among the missed wanted DDs, the number with close to threshold support is more than the number with higher support.

The final experiment investigates whether there is a possible way to filter some of the error DDs from the discovery. For a set of discovered DDs $D_i^s$ from 10 samples, we use three types of filters. The first type is the count of the files from which a DD is found. If a DD is found from one of the samples but not from others, this DD may not be significant. The result of this way of filtering is shown in Figure 9(a). The vertical axis is the relative error rate of the number of missed and the number of unwanted. The horizontal axis labeled by ‘filecnt’ the the file count used in filtering. When the file count gets to 6 or more (out of 10 samples), the total error rate reduces to the minimal.
Figure 7: The errors for different number of samples

Figure 8: The relationship between errors and support
Figure 9: Filtering insignificant DDs

The second way of filtering uses an interval ratio and the result is in Figure 9(b). Suppose that for DD \( X(w) \rightarrow A(w') \), we find \( w' = [1, 2] \) from the first sample and \( w' = [3, 4] \) from the second sample. Then the average width of \( w' \) for each sample is 2 and the combined interval is 4. The interval ratio then is 2/4. If the ratio is small over many samples, each sample only get a small portion of a large interval and the DD is less significant. Figure 9(b) indicates that this way of filtering achieves the minimal error rate when when the ratio is 0.6. A larger ratio will cause the loss of more missed DDs and this is undesirable.

The third way of filtering is the most direct way which uses support of the discovered DDs. A filtering threshold (like 0.007) is used to filter away the DDs if their average support over all samples is less than the filtering threshold. The result in Figure 9(b) shows that this way of filtering is not as good as the other two because if a larger filtering threshold is used, too few DDs remain.

7 Related work

On the discovery of functional dependencies (FDs), many algorithms have been developed for this purpose. The major ones are TANE [6], FD-Mine [20], FUN [12], hash-based [11], Dep-Miner [11], and FastFDs [19] etc.. FDs hold at the schema level and the algorithms for discovering FDs do not apply to DDs.

In recent years, a few new types of dependencies, XML functional dependencies, conditional functional dependencies, matching dependencies, and differential dependencies, have been proposed for all sorts of purposes. Some discovery algorithms are also proposed for these types of dependencies: [22, 18] for XML
Figure 10: Difference

functional dependencies, [4] [21] [8] [18] for conditional functional dependencies, [15] [17] for matching dependencies, and [16] for differential dependencies.

One work that relates to ours is the algorithm discovering matching dependencies in [17]. We note that matching dependencies are special cases of differential dependencies, which is what our work aims to discover. The algorithm in [17] transforms the raw data into a distance data set and then discovers a matching dependency for a given \( X \rightarrow Y \) based on the relative frequencies of distance tuples. It searches for the thresholds (interval boundaries in our terms) for \( X \). The assumption with a given candidate restricts the general problem with the search space size \( 2^{|R|} \cdot d \) to a very specific problem with the search space size \( d^{|X|} \) where \( d \) is the average number of possible intervals among all attributes of schema \( R \). Our work does not assume any given \( X \rightarrow Y \) and therefore is much more general.

The algorithm in [16] on DD discovery is the most related work to what we do in this paper. Its core reduction algorithm works by fixing the rhs attribute and its interval of a candidate DD, and then splitting a given search space of the left hand side to find lhs while pruning the subspaces that are not possible to contain any lhs. The algorithm assumes “that differential functions in \( \Phi(X) \) has already been arranged by subsumption order”. We see that the size of the search space is large. If only the intervals containing 0 distance are considered, and each attribute has \( m \) such intervals, there are \( m^{|X|} \) number of points in the search space. This space size becomes larger when the intervals are defined in other ways.

In addition to the method and the representation differences, the two methods find different DDs. For the distance data shown in Figure 7 where \( E \) stands for \( Edu \), \( A \) \( Age \), and \( S \) \( Sal \). A cell labeled with \( S(0,1) \) means that the distances of the tuple pairs falling in the cell are between 0 and 1. With this data, our method finds the following DDs.

\[
\begin{align*}
\text{Age}(0) \rightarrow \text{Edu}(0) & \rightarrow \text{Sal}(0) \\
\text{Age}(1) \rightarrow \text{Edu}(1) & \rightarrow \text{Sal}(0) \\
\text{Age}(2) \rightarrow \text{Edu}(0) & \rightarrow \text{Sal}(0) \\
\text{Age}(0) \rightarrow \text{Edu}(1) & \rightarrow \text{Sal}(0,1) \\
\text{Age}(1) \rightarrow \text{Edu}(0) & \rightarrow \text{Sal}(0,1) \\
\text{Age}(2) \rightarrow \text{Edu}(1) & \rightarrow \text{Sal}(1,3)
\end{align*}
\]

The Split method finds the DDs of
Both sets of DDs are correct with regard to the DD definition and the respective methods. The Split method finds less DDs but our method identifies dense groups better.

The Split method may be used with our base intervals. In that case, if $Sal$ has $m$ base intervals, the number of rhs for $Sal$ to be considered is the factorial of $m$ (i.e., $m!$). In contrast to the case where the intervals starting with 0 are used, only $m$ rhs intervals need to be considered. As the performance for starting-with-0 intervals is already very poor, adding the complexity makes it simply not computable.

8 Conclusion

This paper proposes an algorithm for discovering differential dependencies from data. The algorithm works based on traversing of a lattice level by level and uses a number of way to prune implied DDs to reduce the computation size. The lattice is different from the lattice used in association rule and functional dependency discovery in that edges in the lattice do not represent candidate differential dependencies.

We also conducted a comprehensive analysis on the errors caused by sampling in the discovery computation and propose ways to filter out possible errors.

The proposed algorithm still have high complexity. The future work of this paper includes the investigation of further pruning methods so that uninteresting DDs can be avoided. Work is also needed to investigate how DDs can be used to repair low quality data values and to identify schema mapping and related relations.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. *VLDB*, pages 487–499, 1994.

[2] Radim Belohlavek and Vilem Vychodil. Data tables with similarity relations: functional dependencies, complete rules and non-redundant bases. *DASFAA*, pages 544–658, 2006.

[3] Stavros S. Cosmadakis, Paris C. Kanellakis, and Nicolas Spyratos. Partition semantics for relations. *PODS*, pages 261–275, 1985.

[4] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *TKDE*, 2010.
[5] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. PVLDB, pages 407–418, 2009.

[6] Yka Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. Computer Journal, 42(2):100–111, 1999.

[7] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. ICDE, pages 1275–1278, 2009.

[8] Jiuyong Li, Jixue Liu, Hannu Toivonen, and Jianming Yong. Effective pruning for the discovery of conditional functional dependencies. The Computer Journal, 2012. http://comjnl.oxfordjournals.org/content/early/2012/06/23/comjnl.bxs082.short.

[9] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Dependency discovery from data - a review. TKDE, 24(2):251–246, 2012.

[10] Jixue Liu, Feiyue Ye, Jiuyong Li, and Junhu Wang. On discovery of functional dependencies from data. DKE, 2013.

[11] Stephane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. LNCS 1777 - 7th International Conference on Extending Database Technology (EDBT): Advances in Database Technology, 1777:350–364, 2000.

[12] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. ICDT, pages 189–203, 2001.

[13] Luc De Raedt. A perspective on inductive databases. SIGKDD Explorations, 4(2):69–77, 2002.

[14] Sheldon M. Ross. Introduction to Probability and Statistics for Engineers and Scientists (Fourth Edition). Elsevier Inc., 2009.

[15] Shaoxu Song and Lei Chen. Discovering matching dependencies. CIKM, pages 1421–1424, 2009.

[16] Shaoxu Song and Lei Chen. Differential dependencies: Reasoning and discovery. TODS, 2011.

[17] Shaoxu Song and Lei Chen. Discovering matching dependencies. arXiv:0903.3317v2 [cs.DB], 2013. http://arxiv.org/abs/0903.3317.

[18] Loan T.H Vo, Jinli Cao, and Wenny Rahayu. Discovering conditional functional dependencies in xml data. ADC, 2011.

[19] Catharine Wyss, Chris Giannella, and Edward Robertson. Fastfds: a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. DaWaK, pages 101–110, 2001.
[20] Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. *Journal of Data Mining and Knowledge Discovery*, 16(2):197–219, 2008.

[21] Peter Z. Yeh and Colin A. Puri. Discovering conditional functional dependencies to detect data inconsistencies. *VLDB Conference*, 2010.

[22] Cong YU and H. V. Jagadish. Xml schema refinement through redundancy detection and normalization. *The VLDB Journal*, 17(2):203–223, 2008.