Automatic Discovery of Functional Dependencies and Conditional Functional Dependencies: A Comparative Study

Nabiha Asghar, Amira Ghenai
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{nasghar,aghenai}@uwaterloo.ca

1 Introduction

Over the last twenty years, several algorithms have been proposed for automatic rule/constraint discovery from data, for the purpose of data cleaning. These algorithms look for constraints such as functional dependencies (FDs), conditional FDs (CFDs), inclusion dependencies (INDs), conditional INDs (CINDs), association rules, integrity constraints (ICs) and denial constraints (DCs), among others. While some of these techniques are direct generalizations and extensions of others, many differ greatly from the rest in approach, characteristics and general flavour. Many of these algorithms have not been tested and compared against each other, therefore their core differences and relative strengths and weaknesses are hard to comprehend.

Broadly speaking, the first goal of this project is to analyse this body of constraint-discovery algorithms and build a taxonomy of sorts, which classifies these algorithms based on their similarities and differences, and tests their relative strengths and weaknesses using common test datasets. For now, we limit the scope of this project to the following algorithms for discovery of FDs and CFDs. In particular, we consider the following seven algorithms: FUN [1], Fdep [2] FASTFD [3], Dep-Miner [4], TANE [5], CTANE [6] and CTANE-2 [7].

The second goal is to find/create datasets that can test these algorithms with different parameters and edge cases to highlight their core characteristics, strengths and weaknesses, as well as analyse their performance and accuracy relative to each other. We look for expected as well as anomalous behaviour by these algorithms in this experimental study, that can shed some light on potential algorithmic improvements or point us in the direction of creating hybrid techniques in future.

2 Preliminaries

In this section, we briefly review the definitions of FDs and CFDs with examples.

	ID	Name	Bday	Wage	Supvsr
	(I)	(N)	(B)	(W)	(S)
t_1	e_1	Bob	d_1	1	e_5
t_2	e_2	John	d_1	1	e_1
t_3	e_3	John	d_1	2	e_1
t_4	e_4	Peter	d_3	2	e_2

Table 1: An Example Instance

2.1 Functional Dependencies

Consider a relational schema R with attributes $attr(R)$.

A functional dependency (FD) φ, defined as $X \rightarrow Y$, means that X functionally determines Y where $X, Y \subseteq attr(R)$. The FD φ is satisfied by a database instance r on R if, for any two tuples $t_1, t_2 \in r$, $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$. In this case, we say that r satisfies φ, denoted by $r \models \varphi$. X is called the left-hand side (LHS) or the determinant and Y is called the right-hand side (RHS) or the dependent. [8]

In other words, if there exist two tuples t_1 and t_2 in an instance r that have the same value for attributes X and different values for attribute Y, then there must be some errors present in t_1 or t_2. This is one of the main reasons why FDs are extensively used for data cleaning, as we will see in detail later.

An FD φ is *minimal* if removing an attribute from its LHS makes it invalid. Additionally, an FD is said to be *trivial* if the RHS is a subset of the LHS. It can be shown through Armstrong's rules that multiple attributes in the RHS of FDs can be decomposed into multiples FDs with one attribute on the RHS [8]. Therefore, in this report, we are only interested on FDs with one attribute on the RHS. Thus, given a database instance r of schema R, the FD discovery problem is to find all valid, minimal and nontrivial FDs with one attribute in the RHS that hold on r.

Example 1. Consider Table 1. We see that Name uniquely determines the attributes Birthday and Supervisor. Therefore two possible FDs in this table are $N \rightarrow B$ and $N \rightarrow S$, because we cannot find a pair of tuples t_i and t_j such that $t_i[N] = t_j[N] \wedge t_i[B] \neq t_j[B]$. In contrast, the FD $B \rightarrow S$

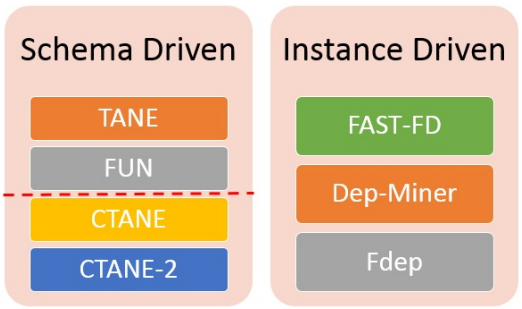


Figure 1: FD and CFD Discovery: Schema Driven vs. Instance Driven Algorithms

is violated by t_1 and t_2 because $t_1[B] = t_2[B] = d_1$, but $t_1[S] \neq t_2[S]$.

2.2 Conditional Functional Dependencies

A conditional functional dependency (CFD) ϕ over a relation schema R is a pair $(X \rightarrow Y, t_p)$ where $X, Y \subseteq attr(R)$. $X \rightarrow Y$ is a standard FD, referred to as the FD *embedded* in ϕ, and t_p is a *pattern tuple* with attributes in X and Y, where for each $B \in X \cup Y$, $t_p[B]$ is either a constant in $domain(B)$ or an unnamed variable '$-$' that draws values from $domain(B)$, and is called a wildcard. We separate the X and Y attributes in t_p with '$||$'. Clearly, standard FDs are a special case of CFDs, because an FD $X \rightarrow Y$ can be expressed as a CFD $(X \rightarrow Y, t_p)$ where $t_p[B] = -$ for all $B \in X \cup Y$. The notion of nontrivial and minimal CFDs carries over from the corresponding notion for FDs.

Example 2. Consider Table 1 again. Here, $\phi_1 = (N \rightarrow B, (John \mid\mid d_1))$ and $\phi_2 = (B \rightarrow W, (d_1 \mid\mid -))$ are both CFDs. We say that $support(\phi_1) = 2$ and $support(\phi_2) = 3$, because a total of 2 and 3 tuples satisfy CFDs ϕ_1 and ϕ_2 respectively.

3 Literature Review: FD and CFD Discovery Algorithms

In this section, we present a brief overview and the core insights for each rule discovery algorithm considered in this study.

FD and CFD discovery approaches can be divided into schema-driven and instance-driven approaches. In this work, we present different techniques from both categories: TANE [5], FUN [1], CTANE [6] and CTANE-2 [7] as examples of schema-driven techniques and Fdep [2], FASTFD [3] and Dep-Miner [4] as examples for instance-driven technique. We will see in results later that FUN, TANE, CTANE and CTANE-2 are sensitive to the size of the schema, while FASTFD, Fdep and Dep-Miner are sensitive to the size of the instance.

3.1 Schema-Driven Algorithms

3.1.1 FUN

FUN [1] is an example of a schema-driven algorithm that uses a level-wise approach to explore the attribute set lattice of an input relation, where at each level k of the lattice, the possible candidates are the *free sets*. The introduced concept of free sets is defined as follows:

Definition 1 *Free Set*
Let $X \subseteq R$ be a set of attributes and r be an instance over a relation R. X is a free set in r if and only if $\nexists X' \subset X, |X'_r| = |X_r|$ where $|X'_r|$ stands for the cardinality of the projection of r over X.

To compute the FDs supported by the instance r of relation R, two more concepts are needed: attribute closure X^+ and quasi-attribute closure $X^\Diamond$. The closure of set X is calculated as follows:

$$X^+ = X + \{A \mid A \in (R - X) \wedge |R[X]| = |r[XA]|\} \quad (1)$$

This essentially means that X^+ contains attribute A on a node at the next level if $X \rightarrow A$. Second, the quasi-closure of X is $X^\Diamond = X + (X - A_1)^+ + ... + (X - A_k)^+$, where $X = A_1 ... A_k$. In fact, $X^\Diamond$ contains the attributes on all the parent nodes of X and all the dependent nodes of the parent nodes. Both values are computed given a set of free set and a level in the lattice, and those values are computed because the FDs are constructed using members of $Fr(r)$ and the two closures:

$$FD = \{X \rightarrow A \mid X \in Fr(r) \wedge A \in (X^+ - X^\Diamond)\} \quad (2)$$

The method used by FUN algorithm to prune the free-sets $Fr(r)$ is to prune non-free-sets X. Later, the algorithm computes the closure of the parent free-set nodes Y of X with the cardinality of the free sets and without accessing the partitions [8].

Algorithm 1 shows how FUN uses attribute lattice traversal level-wise technique to compute FDs. For example, at level 1, the algorithm computes the cardinality of all single attributes and the quasi-closure is set to itself. At level 2, we combine two attributes and recalculate the corresponding 2-attribute cardinality at level 1. If the cardinality of a 2-attribute combination X is the same as the cardinality of its parent P at the previous level, and X is a non-free set and does not participate in future node generation, then we compute the closure of every attribute set P at the previous level. Next, the quasi-closure of every attribute set X at the current level is calculated. Then, the algorithm moves to node generation at level 3.

3.1.2 TANE

TANE [5] is a schema-driven algorithm that automatically discovers exact and approximate FDs in a given dataset, through an exhaustive breadth-first search technique on a

Algorithm 1 FUN

Require: Relation Instance r, Relation R
Ensure: All non-trivial, minimal FDs Σ
1: $L_1 \leftarrow \{\{A\} \mid A \in R\}$
2: $l \leftarrow 1$
3: **while** $L_l \neq \emptyset$ **do**
4: ComputeClosure(L_{l-1}, L_l)
5: ComputeQuasiClosure(L_l, L_{l-1})
6: DisplayFD(L_{l-1})
7: PurePrune(L_l, L_{l-1})
8: $L_l \leftarrow L_l + 1$
9: **end while**

lattice of database attributes, by employing new and efficient pruning methods to trim the search space. The attribute lattice (see Figure 2) is a levelwise structure that contains all the singleton sets of attributes on the first level, all the sets of pairs of attributes on the second level, all the triples on the third level, and so on. An attribute-set X at level i is linked to the attribute-set Y at level $(i + 1)$ if and only if $X \subset Y$; in particular, Y contains exactly one attribute not in X.

TANE starts its search for minimal, nontrivial FDs at the first level of the lattice, and works its way up the levels one by one to larger attribute sets. At its core, for each set X at level i, the algorithm checks whether the FDs $X\backslash\{A\} \rightarrow A$, where $A \in X$, hold on the data or not. This guarantees that only nontrivial FDs are considered. In addition, to ensure that only minimal FDs are checked, TANE keeps track of the possible RHS candidates $\mathcal{C}^+(X)$ for each X. Efficiently checking whether an FD holds on the data is one of the most important and crucial contributions by TANE. This is done by dividing the data tuples into equivalence classes with respect to any attribute-set X. Thus, a partition π_X, containing the equivalence classes with respect to X, of the data tuples is created for every attribute-set X. This gives a nice characterization of when an FD holds: *An FD $X\backslash\{A\} \rightarrow A$ holds if and only if the number of equivalence classes with respect to $X\backslash\{A\}$ is equal to the number of equivalence classes with respect to X*. Once all the items at level i are checked, TANE moves on to level $i + 1$.

Ordinarily, checking every item at every level of the lattice is an exponential process. However, the main insight regarding TANE is that this levelwise, small-to-large search strategy proceeds in a very systematic and pre-determined way, which provides two benefits.

1. It allows information from previous levels to be used in the next levels. For example, the partitions of the data tuple do not need to be created from scratch for every subset of the attributes. TANE computes only the partitions of singleton attributes directly from the database. Partitions π_X for $|X| \geq 2$ are computed as a product of partitions with respect to any rwo different subsets of X of size $|X| - 1$; this is conveniently done because only partitions from the previous levels are needed. Another example of information propagation and re-use across levels is that TANE efficiently keeps track of the sets $\mathcal{C}^+(X)$ for each set X, by computing $\mathcal{C}^+(X)$ as the intersection of $\mathcal{C}^+(X\backslash\{A\})$ for all $A \in X$, which have already been computed in previous steps.
2. Once an FD $X\backslash\{A\} \rightarrow A$ is known to hold on the data, all the supersets of $X\backslash\{A\}$ are removed from all the higher levels of the lattice. In addition, A and all $B \in R\backslash X$ are removed from $\mathcal{C}^+(X)$ (because they would only give rise to redundant FDs). These pruning tricks substantially trim the otherwise exponentially large search space of the attribute sets and the RHS candidates for each attribute set.

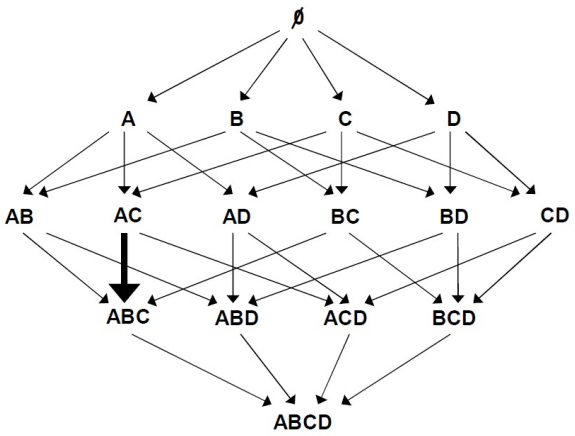


Figure 2: Attribute Search Lattice

A high-level pseudocde for TANE is shown in Algorithm 2.

Algorithm 2 TANE

Require: Relation r over schema R
Ensure: All non-trivial minimal FDs
1: $L_0 \leftarrow \{\emptyset\}$
2: $L_1 \leftarrow \{\{A\} \mid A \in R\}$
3: $\mathcal{C}^+\{\emptyset\} \leftarrow R$
4: $l \leftarrow 1$
5: **while** $L_l \neq \emptyset$ **do**
6: ComputeDependencies(L_l)
7: Prune(L_l)
8: DisplayFDs(L_l)
9: $L_{l+1} \leftarrow$ GenerateNextLevel(L_l)
10: $l \leftarrow l + 1$
11: **end while**

3.1.3 CTANE

The CTANE [6] algorithm, published almost 10 years after TANE, is a straightforward extension of TANE, and automatically discovers variable and constant CFDs in the data. The idea is to leverage the advantages of TANE's levelwise

search strategy and pruning techniques and extend them to accomodate CFDs (which are a generalization of FDs).

Since a CFD is a combination of an FD and a pattern tuple for that FD, the elements of the search lattice for CTANE are now of the form (X, t_p), where X is an attribute set and t_p is a pattern tuple over X that may contain constants or unnamed variables '$-$' to signify wildcard values. The first level contains elements of the form (A, α) where A is a singleton attribute and $\alpha \in domain(A) \cup \{-\}$. An element (X, t_p) at level i is linked to the element (Y, s_p) at level $(i + 1)$ if $X \subset Y$, Y contains exactly one attribute not in X, and s_p agrees with t_p on the X attributes. For each element (X, t_p) at level i, CTANE checks whether the CFDs $X\backslash\{A\} \rightarrow A, (t_p[X\backslash\{A\}] \mid\mid t_p[A])$, where $A \in X$, hold on the data or not. As before, the set $\mathcal{C}^+(X, t_p)$ is also maintained for every (X, t_p) to ensure that the considered CFDs are minimal.

As before, the idea of partitions and equivalence classes is used to efficiently check whether a CFD holds or not. More concretely, *a CFD* $X\backslash\{A\} \rightarrow A, (t_p[X\backslash\{A\}] \mid\mid c_A)$ *holds if and only if the number of equivalence classes with respect to the element* $(X\backslash\{A\}, t_p[X\backslash\{A\}])$ *is equal to the number of equivalence classes with respect to the element* (X, t_p). The idea of information propagation and re-use across levels is carried forward from TANE to CTANE, as expected. The partitions for elements at higher levels of the lattice can be easily computed from smaller already-computed partitions. Similarly, the set $\mathcal{C}^+(X, t_p)$ for any (X, t_p) can be computed as the intersection of the already-computed sets $\mathcal{C}^+(X\backslash\{B\}, t_p[X\backslash\{B\}])$, where $B \in X$.

Finally, the lattice-pruning strategies are similar to those in TANE. Once a CFD $X\backslash\{A\} \rightarrow A, (t_p[X\backslash\{A\}] \mid\mid c_A)$ is found to be true for the data, the set $\mathcal{C}^+(X, t_p)$ is pruned and the element (X, t_p) and its supersets are removed from the lattice.

CTANE allows k-frequent CFDs to be mined from the data, where k is a user-defined parameter called the *support threshold*. The insight here is that 1-frequent CFDs may not be useful in practice, because they capture erroneous and dirty tuples as well. Thus, it is more useful to mine for CFDs satisfied by at least k tuples. This also helps with pruning the search space considerably, because the number of CFDs that hold with a high support is significantly smaller. Choosing the right value of k is an NP- hard problem and it represents a trade-off between efficiency and completeness of results. Varying the value of k allows us to analyze the run-time and memory performance of CTANE on different datasets, as we demonstate in the experiments later.

CTANE's pseudocode is exactly the same as the TANE pseudocode, shown in Algorithm 2, except that the first level L_1 and $\mathcal{C}^+\{\emptyset\}$ are initialized as $L_1 = \{(A, -) \mid A \in attr(R)\} \cup \{(A, a_1) \mid a_1 \in \pi_A(r), A \in attr(R)\}$ and $\mathcal{C}^+\{\emptyset\} = L_1$.

3.1.4 CTANE-2

CTANE-2 [7] is another lattice-based, schema-driven algorithm for automatic CFD discovery, which was published around the same time as CTANE. Though the authors do not give this algorithm any name, we name it CTANE-2 due to its similarity with TANE and CTANE. The goal of CTANE-2 is the same as CTANE's: discovering minimal, nontrivial CFDs from the data that satisfy a minimum support threshold k. CTANE-2 allows the threshold to be of types other than support; conviction, confidence, interest and χ^2-Test are some of the options mentioned in the paper. This flexibility is missing in CTANE.

The underlying insight of CTANE-2 is remarkably similar to CTANE (and TANE). The attribute lattice used here is of the type used in TANE and shown in Figure 2. It is traversed in a levelwise, breadth-first manner, where the singleton attributes and edges at level 1 are considered first, followed by all the attribute pairs and edges at level 2, and so on. Based on an edge in the lattice between sets X (at level i) and $Y = X \cup A$ (at level $i+1$), CTANE-2 checks all the CFDs $[Q, P] \rightarrow A$, where $X = P \cup Q$, P is the set of variable attributes and Q is a set of conditional attributes. For example, in Figure 2, for edge $(X, Y) = (AC, ABC)$, CTANE-2 checks the following CFDs: $(Q = A, P = C \rightarrow B), (Q = C, P = A \rightarrow B), (Q = A, C, P = \phi \rightarrow B)$ and $(Q = \phi, P = A, C \rightarrow B)$, where Q if non-empty, takes on all possible values of the attribute set it equals. For example, in the first candidate CFD, Q takes on all the values of A occuring in the database. Note that the fourth candidate CFD is essentially an FD, and the third candidate CFD has a constant left-hand-side.

CTANE-2's method of checking whether a CFD holds on the data is also based on equivalence classes and partitions of attribute sets, but the actual validity test is slightly different from that in TANE and CTANE. Here, the notion of *subsumed classes* is used. Concretely, for attribute sets X and Y, an equivalence class x_i in π_X is subsumed by an equivalence class y_i in π_Y if $x_i \subseteq y_i$. The symbol $\Omega_X \subseteq \pi_X$ is used to denote the set of all the subsumed classes in π_X. The set Ω_X is an important tool, which is also used to ensure minimality of CFDs; this is explained later. The validity test, then, is as follows: *The candidate* $[Q, P] \rightarrow A$ *is a valid CFD if and only if there exists an equivalence class* q_i *in the partition* π_Q*, such that* q_i *contains values only from* $\Omega_{P \cup Q}$. Intuitively, this means that the CFD validity test is based on identifying values in $X = P \cup Q$ that map to the same $Y = P \cup Q \cup A$ value. In other words, we are interested in the equivalence classes of X that do not split into two or more classes in Y. This is just a different way of phrasing CTANE's CFD validity test and the notion of *refinement* of classes.

Note that the candidate dependencies considered by CTANE-2 are never trivial, because the edges of the lattice never correspond to trivial dependencies. The pruning strategies employed by CTANE-2 are also similar to what

we have seen before. For candidates of the form $X \rightarrow A$ that hold as FDs, all supersets of X are removed from the lattice. For candidates of the form $[Q, P] \rightarrow A$ that hold as CFDs, all the classes $x_i \in \Omega_X$ are pruned from being considered in subsequent evaluations of this CFD where the conditions are supersets of Q. This also ensures the minimality of the discovered rules and is congruent to the idea of maintaining sets of the form $\mathcal{C}^+(X)$ in TANE and CTANE. Finally, increasing the support threshold k (or the confidence/conviction/interest threshold) also helps trim the search space by quickly pruning unlikely candidates at an early stage of traversal.

A high-level pseudocode for CTANE-2 is shown in Algorithm 3.

Algorithm 3 CTANE-2

Require: Relation r over schema R, current level k
Ensure: All non-trivial minimal CFDs
1: Initialize CandidateCFDList $CL = \{\}$, GlobalCandidateList $G = \{\}$
2: **for** $X \in$ level k **do**
3: consider marked edge (X, Y)
4: **if** $|\pi_X| = |\pi_Y|$ **then**
5: (X, Y) is an FD. Unmark edge (X, Y).
6: Remove supersets of (X, Y) from G.
7: **else**
8: $O_X =$ subsumed classes of X, V_X=$(X - O_X)$
9: $(O_X, CL) =$ FindCFDs(O_X, X, Y, k)
10: **if** $O_X \neq \emptyset$ **then**
11: $G(X', Y', Q, P) = (V_X, O_X)$.
12: **end if**
13: **end if**
14: **if** $G = \emptyset$ **then**
15: break, since no candidates remaining
16: **end if**
17: **end for**
18: **return** CL

3.2 Instance Driven Algorithms

3.2.1 Dep-Miner

Dep-Miner [4] is an example of an instance-driven FD discovery technique that first computes the agree sets of tuples using stripped partitioning technique. Later, a levelwise algorithm is used for computing the LHS (minimal set cover) of the minimal non-trivial functional dependencies.

Let R be a relation schema and r an instance of R. For two tuples $t_1, t_2 \in r$, the agree set of t_1 and t_2 is defined as:

$$A(t_1, t_2) = \{B \in R \mid t_1[B] = t_2[B].\} \tag{3}$$

The agree sets of instance r are:

$$\Re_r = \{A(t_1, t_2) \mid t_1, t_1 \in r, t_1 \neq t_2.\} \tag{4}$$

Then, a maximal set is an attribute set X which, for some attribute A, is the largest possible set not determining A. We denote by $max(dep(r), A)$ the set of maximal sets for A w.r.t. $dep(r)$:

$$max(dep(r), A) = \{X \subseteq R \mid r \not\models X \rightarrow A, \forall Y \subseteq R, X \rightarrow Y, r \models Y \rightarrow A\} \tag{5}$$

$$MAX(dep(r)) = \cup_{A \in R} max(dep(r), A) \tag{6}$$

To compute the FDs from $MAX(dep(r))$, Dep-Miner uses the notion of hypergraph. A collection H of subsets of R is a *simple hypergraph* if $\forall X \in H, X \neq 0$, and $(X, Y \in H$ and $X \subseteq Y \Rightarrow X = Y)$. Minimal transversals of simple hypergraph are related to LHS of functional dependencies. Algorithm 4 shows a general overview of Dep-Miner.

Algorithm 4 Dep-Miner

Require: Relation Instance r, Relation R
Ensure: All non-trivial minimal FDs Σ
1: **for all** A in R **do**
2: calculate $\Re_r^A$
3: calculate CMAX-SET
4: **end for**
5: Find LHS from CMAX-SET using levelwise technique
6: Every output X $\rightarrow$ A is $\in \Sigma$

3.2.2 FASTFD

FASTFD [3] is an instance-based FD discovery algorithm that starts by computing the difference set of tuples, and then adopts a depth-first search technique to find minimal covers for the difference sets.

Let R be a relation schema and r an instance of R. For two tuples, $t_1, t_2 \in r$, the difference set of t_1 and t_2 is defined as:

$$D(t_1, t_2) = \{A \in R \mid t_1[A] \neq t_2[A].\} \tag{7}$$

The difference sets of instance r are:

$$D_r = \{D(t_1, t_2) \mid t_1, t_2 \in r, D(t_1, t_2) \neq \emptyset.\} \tag{8}$$

Given attribute $A \in R$, the difference sets of r modulo A are:

$$D_r^A = \{D - \{A\} \mid D \in D_r \text{ and } A \in D.\} \tag{9}$$

An FD $\varphi : X \rightarrow A$ is a minimal functional dependency over r if and only if X is a minimal cover of $D_r^A(r)$. In other words, φ is a valid FD if and only if X covers D_r^A i.e. X intersects with every element in D_r^A. Therefore, instead of computing FDs, the problem is now transformed into finding all minimal set covers of D_r^A for every attribute $A \in R$. Algorithm 5 shows how to compute FDs from $D_r^A(r)$.

3.2.3 Fdep

Fdep [2] is an instance-based FD discovery algorithm that is categorized under the dependency induction algorithms. Fdep introduces the concept of dependency discovery as an *induction task* where the tuples in a relation represent

Algorithm 5 FASTFD

Require: Relation Instance r, Relation R
Ensure: All non-trivial, minimal FDs Σ
1: **for all** $A \in R$ **do**
2: calculate $D_r^A(r)$
3: **end for**
4: **for all** $A \in R$ **do**
5: Find minimal set cover of $D_r^A(r)$ using depth-first search
6: Every output $X \rightarrow A$ is $\in \Sigma$
7: **end for**

instances of that relation, and dependencies represent hypotheses about the relation. Induction algorithms can be classified as bottom-up, top-down or bi-directional. [2] showed that *bottom-up* search method within this framework is superior. Therefore, we are going to give a brief overview of this type only as this is the technique used in the experimental section.

Fdep relies on the concept of *negative cover*, which is a cover of all FDs violated by the relation. Negative cover is calculated using agree-sets of tuples of the relation (agree-set concept has been defined previously). Agree sets can be calculated using partitions, like TANE and Dep-Miner.

The main idea behind using agree sets is that if $ag(t_1, t_2)$, then $A \in (R - X)(t_1[A] \neq t_2[A])$, which means the FD $X \rightarrow A$ is violated. This is the main idea behind using the negative cover for discovering functional dependencies.

The next concept used is the *max-set* of attribute A, which denotes the maximum list of agree-sets that do not include the attribute A. The max-set of all attributes is called the negative closure and it represents all FDs violated by a relation.

The max-sets are then used to derive FDs supported by instance r. The FDs with the RHS of A, denoted by $FD(A)$, are formulated in two steps: [8]

$$FD_1(A) = \{X \rightarrow A \mid X \in (R-A) \wedge \exists Y \in max(A)(X \subseteq Y)\} \quad (10)$$

$$FD(A) = \{f \mid f \in FD_1(A) \wedge \nexists g \in FD_1(A)(lhs(g) \in lhs(f))\} \quad (11)$$

The first step says that for any $Y \in max(A)$, the FD $Y \rightarrow A$ is violated if there is a tuple V such that YV is not $max(A)$, then the FD $YV \rightarrow A$ is satisfied. As a result, $X \rightarrow A$ as A is not a subset of Y. The second step says that $FD(A)$ has only minimal FDs.

4 Source Code, Datasets and Implementation Details

The Java source code for the five FD algorithms (TANE, FUN, FASTFD, Fdep and Dep-Miner) was obtained from the Metanome Project[1]. The source code for CTANE and CTANE-2, on the other hand, is not available publicly. Upon contacting the authors of CTANE, we were told that its source code is proprietary knowledge owned by Bell Labs and cannot be shared. Implementing CTANE proved to be an arduous and time-consuming task, because the paper does not mention all the algorithmic details needed for a reader to implement them on his/her own. Significant time was spent understanding, debugging and testing[2] our code in Python. We also set out to implement CTANE-2 ourselves, but its papers mentions minimal details about how the candidate lists are initialized and maintained. They also fail to mention how the subsumed sets are built and updated efficiently, and how the CFD validity check is efficiently implemented. We did not have time to experiment with these ourselves so, in the interest of time, we contacted the authors and acquired a copy of the source code for CTANE-2, implemented in Perl. Ideally, all implementations should have been in the same language, so that run-times could be compared. Nevertheless, most of the CFD-related experiments presented here give results that can be interpreted in a language-independent manner.

We used thirteen standard datasets for our experiments, all available for free download at the UCI Machine Learning Repository[3] unless stated otherwise:

1. Iris, containing 5 attributes and 150 tuples ($|R| = 5$ and $|r| = 150$).
2. Balance-scale ($|R| = 5$ and $|r| = 650$).
3. Chess ($|R| = 7$ and $|r| = 28056$).
4. Abalone ($|R| = 9$ and $|r| = 4177$).
5. Nursery ($|R| = 9$ and $|r| = 12960$).
6. Breast Cancer Wisconsin ($|R| = 11$ and $|r| = 699$).
7. Bridges ($|R| = 13$ and $|r| = 108$).
8. Echocardiogram ($|R| = 13$ and $|r| = 132$).
9. Adult ($|R| = 14$ and $|r| = 48842$).
10. Letter ($|R| = 16$ and $|r| = 32561$).
11. Ncvoter[4] ($|R| = 19$ and $|r| = 1000$).
12. Automobile ($|R| = 26$ and $|r| = 205$).
13. Horse ($|R| = 27$ and $|r| = 368$).

All the experiments were done on an Intel Core i5 CPU with 4 cores (1.6 GHz each) and 8 GB RAM.

5 Experiments with FD Discovery Algorithms

Table 2 shows the performance of FastFD, TANE, FUN, Dep-Miner and Fdep on all the thirteen datasets. It is important to mention here that the version of TANE used for this experiment is TANE/MEM, which works completely in the main memory and there is no disk usage.

[1] https://hpi.de/cn/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html.

[2] We actually had to go back and implement TANE in Python ourselves in order to understand exactly how it works, so that we could in turn figure out how CTANE works!

[3] http://archive.ics.uci.edu/ml/

[4] available at ftp://alt.ncsbe.gov.gov/data/

Name	|R|	|r|	|F|	FASTFD	TANE	Dep-Miner	Fdep	FUN
iris	5	150	4	0.15	0.53	0.21	0.25	0.15
balance-scale	5	625	1	0.63	0.29	0.21	0.11	0.22
chess	7	28056	1	28.06	1.22	132.21	98.30	0.84
abalone	9	4177	137	4.18	0.63	1.89	3.03	0.29
nursery	9	12960	1	12.96	1.72	92.55	33.53	1.26
breast-cancer	11	699	46	0.70	0.95	0.68	0.21	0.35
bridges	13	108	142	0.11	0.66	0.26	0.11	0.44
echocardiogram	13	132	538	0.13	0.52	0.21	0.09	0.14
adult	14	48842	78	48.84	•	2948.31	432.47	•
letter	16	32561	61	32.56	•	686.94	181.45	•
ncvoter	19	1000	758	1.00	3.07	7.36	0.72	3.86
automobile	26	205	4176	0.21	•	11713.52	0.33	7926.68
horse	27	368	128726	0.37	•	○	7.65	•

Table 2: Run-time Results on 13 Datasets. [○] indicates that Dep-Miner took longer than 4 hours and was aborted. [•] indicates that the program ran out of main memory.

Quantitative Analysis of Table 2: First, we consider the running times of every algorithm in Table 2 separately and analyze the performance of the algorithms as $|r|$ and $|R|$ increase for different datasets:

- FASTFD tends to work well with datasets where the number of tuples is relatively small, such as Bridges and Echocardiogram. However, its performance degrades as the number of tuples becomes high; this is obvious when running the algorithm on Chess, Letter and Adult datasets, where the execution times are the highest. This is mainly because higher number of tuples increases the difference set computation in FASTFD. From these results, we can conclude that FASTFD is sensitive to the size of the instance, rather than the size of the schema, because for datasets with high $|R|$ and low $|r|$ (Automobile and Horse), the performance was good. Indeed, FASTFD is an *instance-driven* algorithm.

- TANE shows good performance where $|R|$ is low. However, as the number of attributes starts to grow, TANE experiences serious difficulties due to excessive memory consumption and runs out of memory. For Adult, Letter, Automobile and Horse datasets, TANE runs out of memory and cannot find the minimal FDs using only the main memory. This is because TANE adopts a level-wise candidate generation and pruning strategy which is dependent on the number of attributes in the relation. To sum up, TANE is a *schema-driven* algorithm because it is sensitive to the size of the schema.

- Dep-Miner shows better performance with datasets where $|r|$ is low, e.g. Breast-cancer-Wisconsin and Bridges. When the number of tuples is high, the algorithm takes more time to compute the minimal FDs. Examples of such relations are Chess, where the number of tuples is very high, as well as Adult and Automobile. Dep-Miner takes a longer time with higher number of tuples due to the computation of the agree sets. From these results, we can see that Dep-Miner is an *instance-driven* algorithm . Section 5.2 gives a detailed comparison between the two discovered instance-driven algorithms.

- Fdep has shown to be very efficient for datasets where the number of tuples is less than 10000. However, as the number of tuples increase, the performance of the algorithm decreases and this is evident for relations like Adult, Letter and Chess. This is mainly due to the negative cover computation (agree-set) as the number of tuples $|r^2|$ increase. Fdep is sensitive to the size of the instance and for this reason it is an *instance-driven* algorithm.

- FUN shows good performance for almost all the datasets except for the ones where the number for attributes is growing. In Adult, Letter and Horse datasets, FUN fails to compute the minimal FDs as it runs out of memory and the main reason behind that is that FUN adopts an attribute lattice traversal technique that is exponential in the number of attributes. As this algorithm is sensitive to the size of the schema, it is considered to be a *schema-driven* algorithm. Section 5.1 compares TANE and FUN in detail.

5.1 Schema Driven Algorithms (TANE vs. FUN)

We chose to compare two of the schema-driven algorithms in terms of performance: TANE and FUN. We will mainly compare them in terms of the wall clock time and the correlation factor.

5.1.1 Wall Clock Time

This set of experiments involved experiments with TANE and FUN on nine datasets. The results are shown in Figure 3.

Quantitative Analysis of Figure 3: Figure 3 shows that, for most of the datasets, FUN had better performance than TANE. The main reason for FUN's higher efficiency is that TANE suggests different pruning strategies which may be

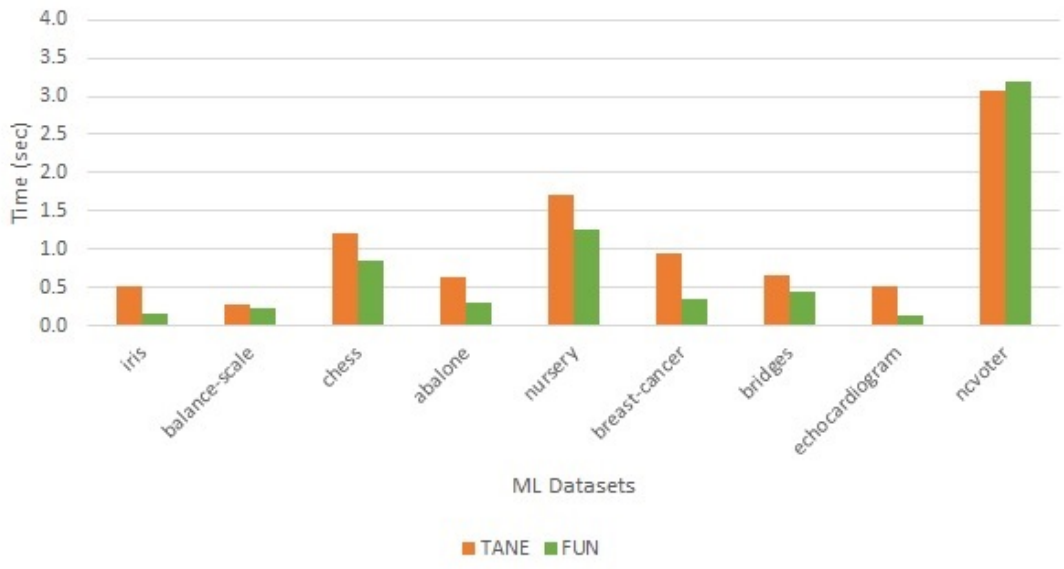


Figure 3: Wall clock time of TANE and FUN algorithms on Nine Datasets

costly to compute. The gap between execution time of TANE and FUN is caused, on one hand, by the source set (C+) computation and, on the other hand, by the exploration of the search space, more reduced in FUN than in TANE. In other words, TANE tends to discard irrelevant candidates, some of which capture non-minimal FDs, whereas FUN examines all candidates but only captures possible minimal FDs. Any candidate provided with a set C+ different from R is a non free set. It is deleted by FUN whereas TANE eliminates a candidate only if its associated C+ is empty. Thus the search space explored by FUN is smaller than that of TANE. The new characterization that we propose is simpler and sound. It is based on three concepts. FUN eliminates the candidates of FD discovery much faster, compared to TANE.

5.1.2 Correlation Factor

In this section, we present experimental results obtained with randomly-generated integer relation instances and with fixed attributes $|R| = 10$. We used a synthetic dataset to control the rate of identical values by introducing the parameter correlation factor (CF), which controls the number of identical values in a column of the table. For example, if CF has a value of 50% for an attribute and the number of tuples is 1000, then each value for this attribute is chosen from 500 possible values. Tests were done on various datasets, classified into three groups: datasets without constraints, datasets with CF = 50% and datasets with CF = 90%.

Quantitative Analysis of Table 3: The complexity of FUN stems from the cost of computing free sets and the cost of computing FDs. As CF increases, the length of the stripped partitions in the free set computation also increases, as the cardinality computation becomes more complex. Thus, the free set calculation becomes more time-consuming as the CF progressively increases. This is the primary reason for the difference between the execution times of FUN and TANE.

	CF=0.0		CF=0.5		CF=0.9	
|r|	FUN	TANE	FUN	TANE	FUN	TANE
50000	5.04	3.34	5.08	4.29	4.50	3.60
100000	12.97	5.30	13.77	6.89	6.89	4.13
150000	25.16	10.28	22.14	10.82	11.74	5.81
200000	36.11	12.65	34.31	14.20	14.99	8.43
250000	50.78	19.70	23.64	10.48	21.87	11.20
300000	64.63	22.93	28.57	12.62	26.43	15.19

Table 3: Execution times in seconds for correlated data ($|R| = 10$)

5.2 Instance Driven Algorithms (FASTFD vs. Dep-Miner)

We chose to compare two of the instance-driven algorithms in terms of performance: FASTFD and Dep-Miner. Table 4 shows experiments run on the 13 UCI datasets.

Quantitative Analysis of Table 4: The performance of FASTFD and DepMiner is very similar when the dataset has a small number of attributes. The difference between the wall clock time of the algorithms is more obvious when the number of attributes gets large. For example, in the Automobile dataset with 26 attributes, the difference between the wall clock time of the two algorithms is remarkable. In datasets where the number of attributes is large, FastFD is more efficient than Dep-Miner and the main reason relies

Name	|R|	|r|	**FastFD**	**Dep_Miner**
iris	5	150	0.10	0.21
balance-scale	5	625	0.65	0.21
chess	7	28056	133.19	132.21
abalone	9	4177	2.19	1.89
nursery	9	12960	97.27	92.55
breast-cancer	11	699	0.69	0.68
bridges	13	108	0.26	0.26
echocardiogram	13	132	0.23	0.21
adult	14	48842	3165.40	2948.31
letter	16	32561	678.80	686.94
ncvoter	19	1000	1.42	7.36
automobile	26	205	2.72	11713.52
horse	27	368	338.81	○

Table 4: Execution times in seconds for 13 datasets. [○] indicates Dep-Miner took longer than 4 hours and was aborted.

on the difference they both compute the minimal difference cover. Dep-Miner uses lattice traversal method which is exponential in the number of attributes and very expensive, while FASTFD uses depth-first search technique to compute the minimal set cover in order to output the minimal FDs. Additionally, use of the lattice traversal method by Dep-Miner makes the space usage/memory usage of this algorithm worse than FASTFD; FASTFD is never in danger of running out of memory.

6 Experiments with CFD Discovery Algorithms

We now explain our experiments with the CFD discovery algorithms. Since we study only two CFD algorithms, both of which are schema-driven, we present the results for each dataset one by one. The experiments compare the running time, memory usage and the size of output of the two algorithms, and makes deductions about their inherent characteristics based on the results.

6.1 Experiments with Iris Dataset

Figures 4, 5 and 6 show the results of the experiments done on the Iris dataset using CTANE and CTANE-2. The results for TANE are mentioned in the text.

Figure 4 shows the running time (wall-clock time) of CTANE and CTANE-2 when the support threshold and number of attributes are varied. For CTANE (Figure 4(a)), the running time explodes exponentially for support = 10 when $|R|$ is increased, but rapidly falls down to under 1 second at support =15. For CTANE-2 (Figure 4(b)), we again see an exponential blowup in running time at support = 10 for increasing values of $|R|$. However, the running time decreases more slowly when the support is increased, and goes down to around 1 minute when the support is 40. Note here that for (a) and (b), the scale of y-axis is different. This is because CTANE is implemented in Python and CTANE-2 is implemented in Perl, and Python, in general, is much slower than Perl. However, the trends still show an exponential running time for both when the support is low and the number of attributes is high. *The key insights here are the exponential blow up in running time when support = 10, and the difference in the way the running time falls when the support is increased. This is rapid for CTANE, but gradual for CTANE-2.* In contrast, TANE (not shown in the Figure) always takes 0.36 seconds to complete, regardless of the value of $|R|$.

Figure 5 shows how the RAM usage (more specifically called the maximum resident set size) varies for CTANE and CTANE-2 when the support threshold and the value of $|R|$ are varied. We see trends here that are similar to the running time trends. The RAM usage blows up exponentially when $|R|$ is increased at support = 10, for both algorithms. Furthermore, for both algorithms, it falls under 200 MB at support = 15 for all values of $|R|$, and for support ≥ 40, it falls under 100 MB. *The key insights here are the exponential blow up in memory usage at support = 10, and the fact that the y-axis scales are very different for (a) and (b), which signifies that CTANE uses exponentially more memory than CTANE-2 when support = 10.* In contrast, TANE always uses 40 MB of RAM, regardless of the value of $|R|$.

Finally, Figure 6 shows the number of CFDs output by the two algorithms when the support and the number of attributes are varied. Here, the two algorithms behave very differently (note that the y-axis scales are the same for both algorithms, so the results are directly comparable). At support = 10, the number of CFDs output by the two algorithms is almost equal. However, for $10 <$ support < 60, CTANE consistently outputs more CFDs than CTANE-2. For support ≥ 60, the number of CFDs is equal for the two algorithms for each value of $|R|$; this is because both algorithms output FDs only, and these numbers are in fact the same as those output by TANE. *The key insights here are the blowup in the number of CFDs output by both algorithms at support $\leq$ 10, and the fact that CTANE consistently outputs more CFDs than CTANE-2 for* $10 <$ *support* < 60.

Overall Interpretation: The explosion in running time and memory usage at support = 10 is understandable for both algorithms, because the number of CFDs at support = 10 increases exponentially when $|R|$ is increased. CTANE takes more time than CTANE-2 in general because Python is slower than Perl. The two most interesting findings are as follows.

1. CTANE uses exponentially more memory than CTANE-2 when support = 10, even though the number of CFDs output at this support are almost equal. We investigated the reasons for this by looking at the intermediate results of the two algorithms, and discovered that CTANE checks (and rejects) far more candidates than CTANE-2. This means that CTANE-2's pruning strategies are superior to CTANE's.
2. CTANE outputs more CFDs than CTANE-2 for $10 <$

support < 60. To understand this discrepancy, we looked at the actual CFDs output by both the algorithms. For example, for $|R| = 5$, the CFDs output by CTANE and CTANE-2 are as follows.

CFDs output by CTANE for support = 10 and for support = 15:

1) $A, B, C \rightarrow E, (-, -, - \,||\, -)$
2) $A, B, C \rightarrow E, (-, -, - \,||\,$ Iris-virginica$)$
3) $A, B, C \rightarrow E, (-, -, - \,||\,$ Iris-setosa$)$
4) $A, B, C \rightarrow E, (-, -, - \,||\,$ Iris-versicolor$)$
5) $A, B, D \rightarrow E, (-, -, - \,||\, -)$
6) $A, B, D \rightarrow E, (-, -, - \,||\,$ Iris-virginica$)$
7) $A, B, D \rightarrow E, (-, -, - \,||\,$ Iris-setosa$)$
8) $A, B, D \rightarrow E, (-, -, - \,||\,$ Iris-versicolor$)$
9) $A, C, D \rightarrow E, (-, -, - \,||\, -)$
10) $A, C, D \rightarrow E, (-, -, - \,||\,$ Iris-virginica$)$
11) $A, C, D \rightarrow E, (-, -, - \,||\,$ Iris-setosa$)$
12) $A, C, D \rightarrow E, (-, -, - \,||\,$ Iris-versicolor$)$
13) $B, C, D \rightarrow E, (-, -, - \,||\, -)$
14) $B, C, D \rightarrow E, (-, -, - \,||\,$ Iris-virginica$)$
15) $B, C, D \rightarrow E, (-, -, - \,||\,$ Iris-setosa$)$
16) $B, C, D \rightarrow E, (-, -, - \,||\,$ Iris-versicolor$)$

CFDs output by CTANE-2 for support = 10:

1) $A, B, C \rightarrow E, (-, -, - \,||\, -)$
2) $A, B, D \rightarrow E, (-, -, - \,||\, -)$
3) $A, C, D \rightarrow E, (-, -, - \,||\, -)$
4) $B, C, D \rightarrow E, (-, -, - \,||\, -)$
5) $D \rightarrow E, (0.2 \,||\, -)$
6) $B, C, D \rightarrow A, (2.8, -, - \,||\, -)$
7) $C \rightarrow E, (1.5 \,||\, -)$
8) $D \rightarrow E, (1.3 \,||\, -)$
9) $A, B, D \rightarrow C, (-, 3.1, - \,||\, -)$
10) $A, C, D \rightarrow B, (-, -, 1.8 \,||\, -)$
11) $B, C, D \rightarrow A, (3.1, -, - \,||\, -)$
12) $C \rightarrow E, (1.4 \,||\, -)$
13) $A, B, D \rightarrow C, (-, 2.9, - \,||\, -)$
14) $A, B, E \rightarrow C, (-, 2.9, - \,||\, -)$

CFDs output by CTANE-2 for support = 15:

1) $A, B, C \rightarrow E, (-, -, - \,||\, -)$
2) $A, B, D \rightarrow E, (-, -, - \,||\, -)$
3) $A, C, D \rightarrow E, (-, -, - \,||\, -)$
4) $B, C, D \rightarrow E, (-, -, - \,||\, -)$
5) $D \rightarrow E, (0.2 \,||\, -)$

We see that both algorithms detect some CFDs that go undetected by the other. CTANE CFDs # 2, 3 and 4, for example, are simply special cases of CFD # 1 (which is an FD). Such special cases of FDs are never output by CTANE-2, and this is why CTANE-2 is more efficient in terms of run-tme as well as memory. Furthermore, CTANE-2 focuses on detecting CFDs whose right-hand-sides are wild-cards, and constants only appear in the left-hand-sides. Overall, it looks like CTANE-2's output is much more useful than that of CTANE.

6.2 Experiments with Balance-scale Dataset

Figures 7, 8 and 9 show the results of the experiments done on the Balance-scale dataset, which has a slightly higher number of tuples than Iris.

As before, CTANE's running time shoots up exponentially for support < 200 when we increase $|R|$ from 5 to 11 (Figure 7). The same is not true for CTANE-2, who's running time increases exponentially when $|R|$ is increased, but in a way that is less dependent on the support. In other words, CTANE-2 seems to have consistent efficiency across different values of support, and this is not true for CTANE. Exactly the same trend is observed with the memory usage results (Figure 8). Finally, the number of CFDs output by CTANE-2 are not affected by support either (although they increase when $|R|$ is increased, and this is expected). CTANE, on the other hand, outputs many more CFDs and matches CTANE-2's output only when support ≥ 300; these are infact all the FDs that are output by TANE too (Figure 9). In light of the number of CFDs output by these two algorithms, it makes sense that the running time and memory usage by CTANE-2 remains more or less unaffected by support, but that CTANE shows exponential increase in running time and memory usage as support is decreased. Therefore, it is safe to say that the major factor that makes these two algorithms behave differently is the number and kind of CFDs they output. A deeper analysis of the CFDs output by them again shows that the each algorithm detects some CFDs that are missed by the other, but that many of CTANE's CFDs are special cases of FDs and this causes the exponential blow up. This does not happen with CTANE-2.

It is worth mentioning here that TANE's running time and memory usage always remains roughly constant at under 1 second and under 42 MB respectively.

6.3 Experiments with Abalone Dataset

To confirm our previous findings and insights about CTANE and CTANE-2, we run similar experiments on one more dataset: Abalone, which contains almost 6.5 times as many tuples as the Balance-scale dataset. The results are shown in Figures 10, 11 and 12.

The trends of running time, memory usage and number of CFDs discovered are almost exactly the same as those seen for the Balance-scale dataset. The only difference here is that CTANE-2 shows exponential increase in running time and memory usage behaviour around the support = 100 mark. The behaviour of CTANE is also exactly what we expect by now, and as always, the number of CFDs output by CTANE-2 is much lesser than those output by CTANE for supprt < 1000.

These results confirm our hypothesis that CTANE outputs many more CFDs than CTANE-2, many of which may not be very useful. On the other hand, CTANE-2 mostly outputs CFDs that are NOT special cases of already found FDs, and focuses on CFDs that have constants only on the left-hand-side. CTANE-2's pruning strategies also seem superior than CTANE's, which is evident in the memory usage comparisons we have seen. For running time, even though the results of the two algorithms are not directly

comparable (due to the fact that they are implemented in different languages), we can still say that we see an exponential increase in the run-time of CTANE when support or $|R|$ are increased. This is true for CTANE-2 as well, but its exponent seems to be much lower than that of CTANE.

7 Conclusion and Future Work

We have presented an experimental and comparitive study of seven algorithms for automatic discovery of FDs and CFDs. Our experiments test these algorithms under different input parameters in order to identify their key characteristics, their main algorithmic and performance-related differences, relative strengths and relative weaknesses.

There is much to be done in future, in order to turn this comparative study into a consolidated study of all the rule and/or constraint discovery algorithms out there. First and foremost, we intend to experiment wih more FD and CFD discovery algorithms (e.g. FD_Mine [9], DFD [10], CFD tableaux generation [11], FASTCFD [6], CFDMiner [6] and CORDS [12]) in order to make this study more comprehensive. Second, we plan to do more variable and extensive experiments, such as testing the effect of correlation factor on CFD discovery algorithms, and the memory usage of FD discovery algorithms. Third, we intend to include other types of constraint into this study, such as denial constraints, inclusion dependencies and conditional inclusion dependencies, and hope to extend this study by experimenting with algorithms such as FASTDC [13], SPIDER [14] and BINDER [15]. Fourth, we intend to make sure that all algorithms are implemented in the same language, so that results are directly comparable. Fifth, we intend to use servers with more powerful cores and bigger RAM (at least 64 GB) so that we do not run out of memory and we can run experiments with much bigger and realistic datasets. Finally, we intend to include recent, realistic and bigger datasets with tuples in millions, so that all these algorithms can be stress-tested for efficiency and usability.

References

[1] N. Novelli and R. Cicchetti, "Fun: An efficient algorithm for mining functional and embedded dependencies," in *Database TheoryICDT 2001*, pp. 189–203, Springer, 2001.

[2] P. A. Flach and I. Savnik, "Database dependency discovery: a machine learning approach," *AI communications*, vol. 12, no. 3, pp. 139–160, 1999.

[3] C. Wyss, C. Giannella, and E. Robertson, "Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract," in *Data Warehousing and Knowledge Discovery*, pp. 101–110, Springer, 2001.

[4] S. Lopes, J.-M. Petit, and L. Lakhal, "Efficient discovery of functional dependencies and armstrong relations," in *EDBT*, vol. 1777, pp. 350–364, Springer, 2000.

[5] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, "Tane: An efficient algorithm for discovering functional and approximate dependencies," *The computer journal*, vol. 42, no. 2, pp. 100–111, 1999.

[6] W. Fan, F. Geerts, J. Li, and M. Xiong, "Discovering conditional functional dependencies," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 683–698, 2011.

[7] F. Chiang and R. J. Miller, "Discovering data quality rules," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1166–1177, 2008.

[8] J. Liu, F. Ye, J. Li, and J. Wang, "On discovery of functional dependencies from data," *Data & Knowledge Engineering*, vol. 86, pp. 146–159, 2013.

[9] H. Yao, H. J. Hamilton, and C. J. Butz, "Fd_mine: discovering functional dependencies in a database using equivalences," in *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pp. 729–732, IEEE, 2002.

[10] Z. Abedjan, P. Schulze, and F. Naumann, "Dfd: Efficient functional dependency discovery," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 949–958, ACM, 2014.

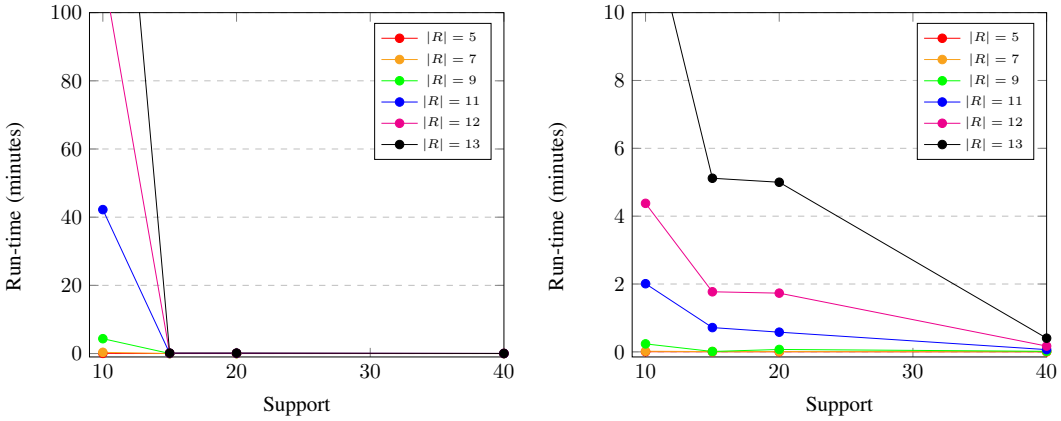
[11] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu, "On generating near-optimal tableaux for conditional functional dependencies," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 376–390, 2008.

[12] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga, "Cords: automatic discovery of correlations and soft functional dependencies," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 647–658, ACM, 2004.

[13] X. Chu, I. F. Ilyas, and P. Papotti, "Discovering denial constraints," *Proceedings of the VLDB Endowment*, vol. 6, no. 13, pp. 1498–1509, 2013.

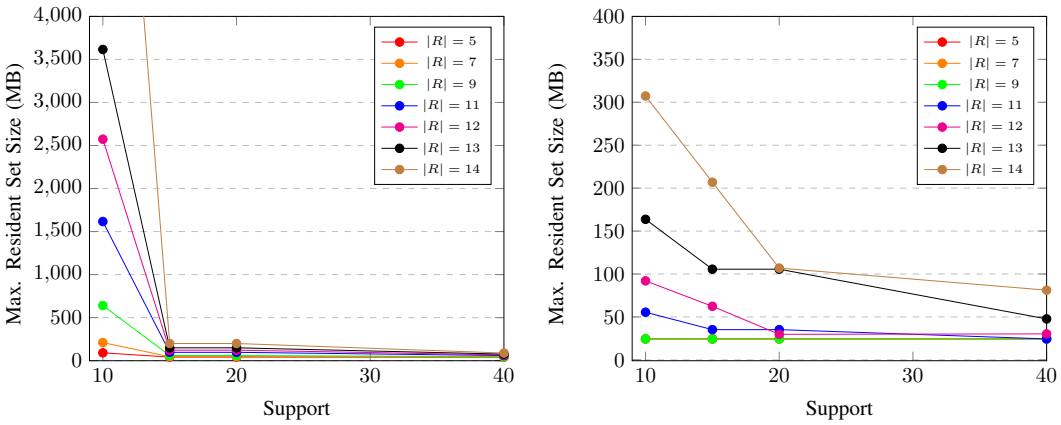
[14] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz, "Efficiently detecting inclusion dependencies," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 1448–1450, IEEE, 2007.

[15] T. Papenbrock, S. Kruse, J.-A. Quiané-Ruiz, and F. Naumann, "Divide & conquer-based inclusion dependency discovery," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 774–785, 2015.



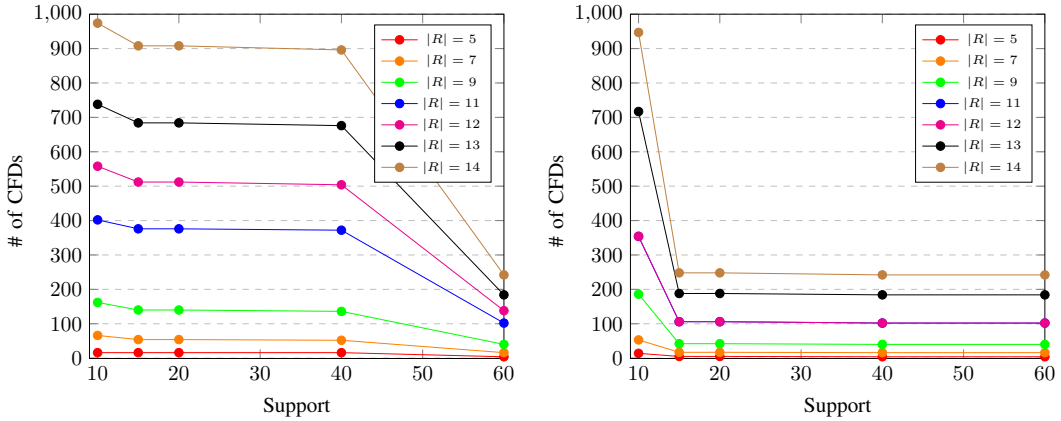
(a) CTANE (b) CTANE-2

Figure 4: Run-time Experiments. Dataset: Iris. # of tuples = 150



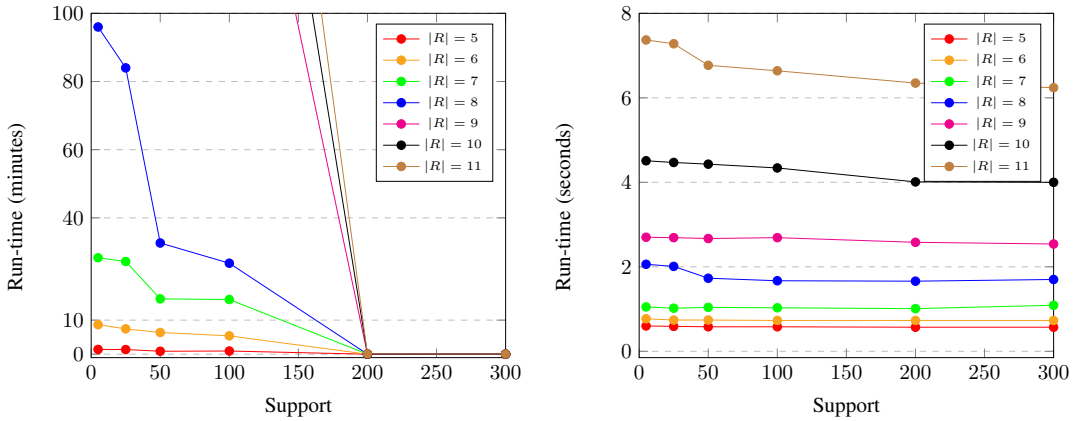
(a) CTANE (b) CTANE-2

Figure 5: Memory Experiments. Dataset: Iris. # of tuples = 150



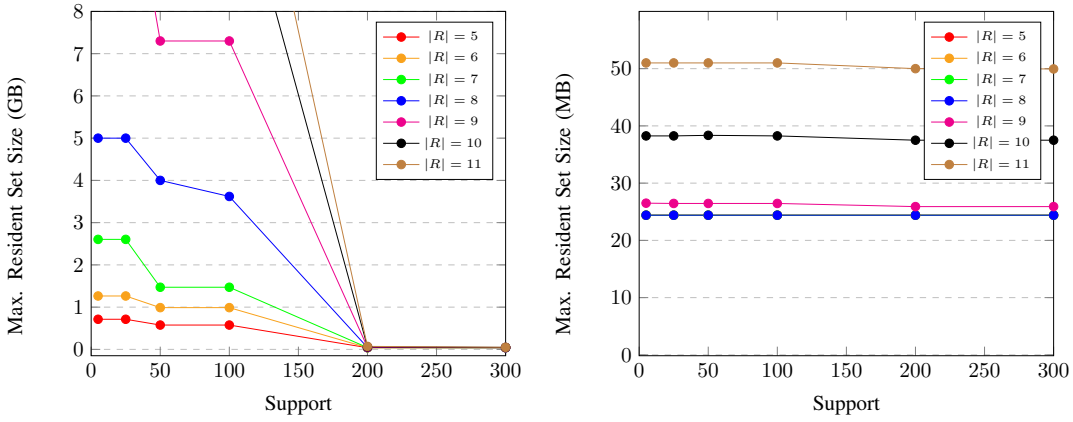
(a) CTANE (b) CTANE-2

Figure 6: Experiments with # of CFDs. Dataset: Iris. # of tuples = 150



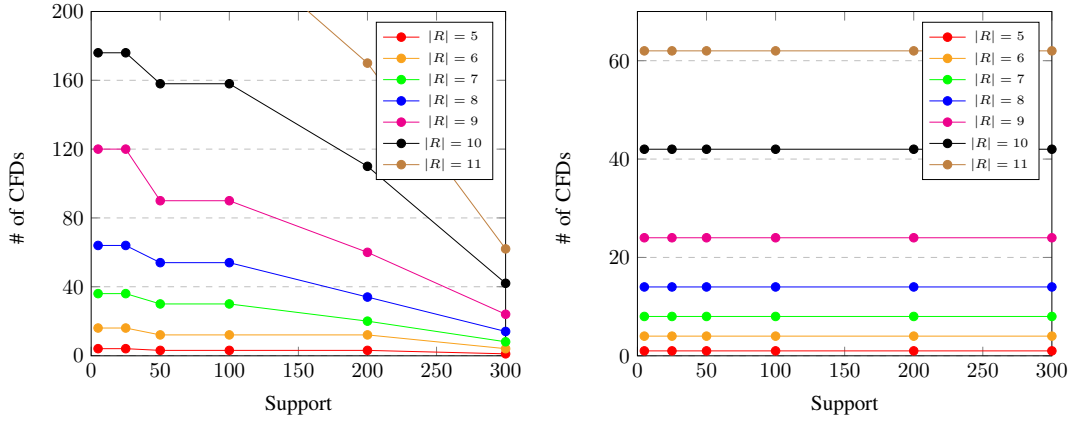
(a) CTANE (b) CTANE-2

Figure 7: Run-time Experiments. Dataset: Balance-Scale. # of tuples = 650



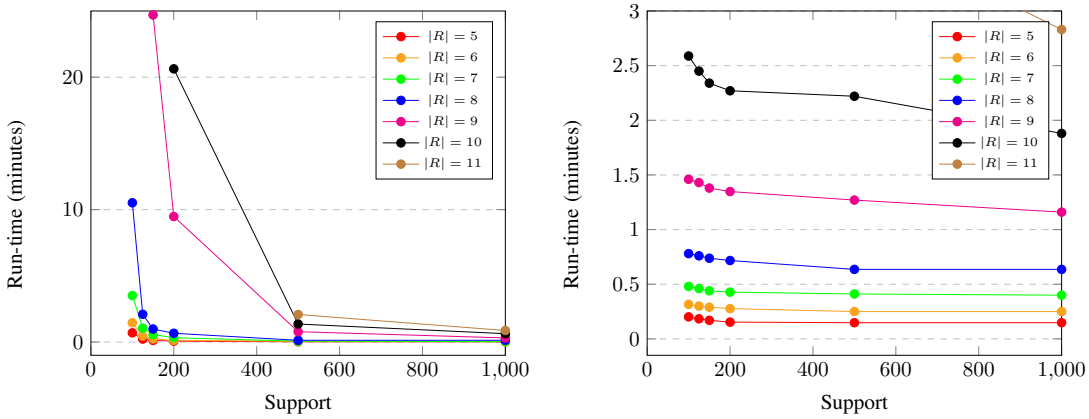
(a) CTANE (b) CTANE-2

Figure 8: Memory Experiments. Dataset: Balance-Scale. # of tuples = 650



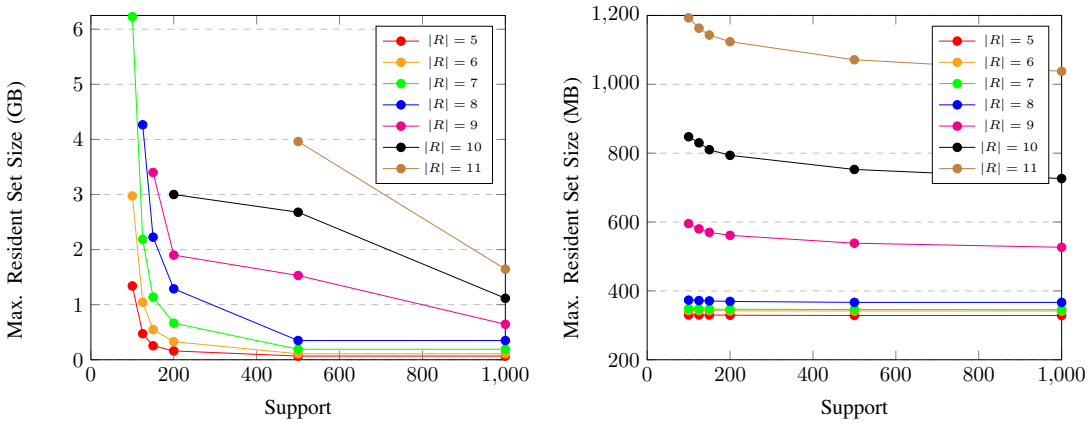
(a) CTANE (b) CTANE-2

Figure 9: Experiments with # of CFDs. Dataset: Balance-Scale. # of tuples = 650



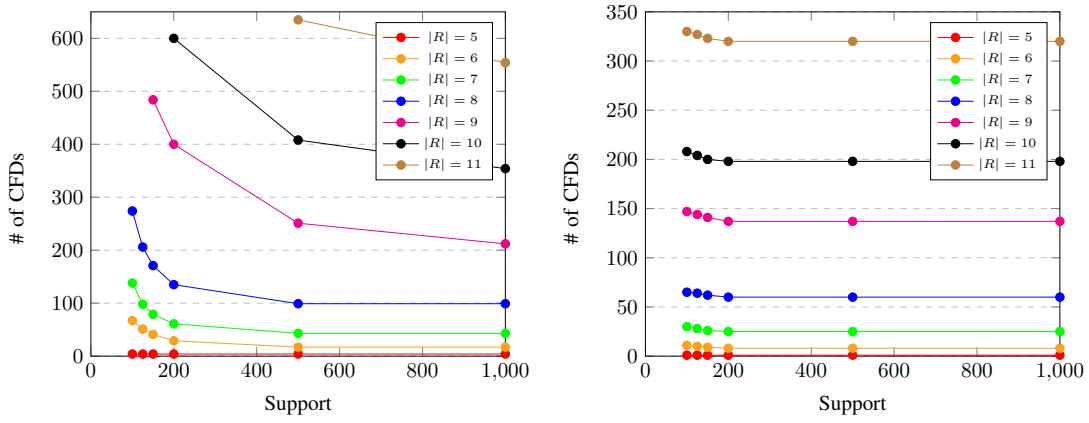
(a) CTANE (b) CTANE-2

Figure 10: Run-time Experiments. Dataset: Abalone. # of tuples = 4177



(a) CTANE (b) CTANE-2

Figure 11: Memory Experiments. Dataset: Abalone. # of tuples = 4177



(a) CTANE (b) CTANE-2

Figure 12: Experiments with # of CFDs. Dataset: Abalone. # of tuples = 4177