On the Relative Trust between Inconsistent Data and Inaccurate Constraints

George Beskales [1] Ihab F. Ilyas [1] Lukasz Golab [2] Artur Galiullin [2]

[1] *Qatar Computing Research Institute*
{gbeskales,ikaldas@}qf.org.qa

[2] *University of Waterloo*
{lgolab,agaliullin}@uwaterloo.ca

***Abstract*—Functional dependencies (FDs) specify the intended data semantics while violations of FDs indicate deviation from these semantics. In this paper, we study a data cleaning problem in which the FDs may not be completely correct, e.g., due to data evolution or incomplete knowledge of the data semantics. We argue that the notion of relative trust is a crucial aspect of this problem: if the FDs are outdated, we should modify them to fit the data, but if we suspect that there are problems with the data, we should modify the data to fit the FDs. In practice, it is usually unclear how much to trust the data versus the FDs. To address this problem, we propose an algorithm for generating non-redundant solutions (i.e., simultaneous modifications of the data and the FDs) corresponding to various levels of relative trust. This can help users determine the best way to modify their data and/or FDs to achieve consistency.**

I. Introduction

Poor data quality is a serious and costly problem, often addressed by specifying the intended semantics using constraints such as Functional Dependencies (FDs), and modifying or discarding inconsistent data to satisfy the provided constraints. For example, many techniques exist for editing the data in a non-redundant way so that a supplied set of FDs is satisfied [3], [5], [11]. However, in practice, it is often unclear whether the data are incorrect or whether the intended semantics are incorrect (or both). It is difficult to get the semantics right, especially in complex data-intensive applications, and the semantics (and/or the schema) may change over time. Thus, practical data cleaning approaches must consider errors in the data as well as errors in the specified constraints, as illustrated by the following example.

Example 1: Figure 1 depicts a relation that holds employee information. Data are collected from various sources (e.g., Payroll records, HR) and thus might contain inconsistencies due to, for instance, duplicate records and human errors. Suppose that we initially assert the FD `Surname, GivenName` $\rightarrow$ `Income`. That is, whenever two tuples agree on attributes `Surname` and `GivenName`, they must agree on `Income`. This FD probably holds for Western names, in which surname and given name uniquely identify a person in most cases, but not for Chinese names (e.g., tuples t_6 and t_9 probably refer to different people). Thus, we could change the FD to `Surname, GivenName, BirthDate` $\rightarrow$ `Income` and resolve the remaining inconsistencies by modifying the data, i.e., setting the Income of t_5 (or t_3) to be equal to that of t_3 (resp. t_5).

	GivenName	Surname	BirthDate	Gender	Phone	Income
t_1	Jack	White	5 Jan 1980	Male	923-234-4532	60k
t_2	Sam	McCarthy	19 Jul 1945	Male	989-321-4232	92k
t_3	Danielle	Blake	9 Dec 1970	Female	817-213-1211	120k
t_4	Matthew	Webb	23 Aug 1985	Male	246-481-0992	87k
t_5	Danielle	Blake	9 Dec 1970	Female	817-988-9211	100k
t_6	Hong	Li	27 Oct 1972	Female	591-977-1244	90k
t_7	Jian	Zhang	14 Apr 1990	Male	912-143-4981	55k
t_8	Ning	Wu	3 Nov 1982	Male	313-134-9241	90k
t_9	Hong	Li	8 Mar 1979	Female	498-214-5822	84k
t_{10}	Ning	Wu	8 Nov 1982	Male	323-456-3452	95k

Fig. 1. An Example of a database instance of persons

In [6], an algorithm was proposed to generate a single repair of both the data and the FDs, which has the lowest cost according to a unified cost model of modifying a tuple and modifying an FD (i.e., adding an extra column to its left-hand-side). However, in practice, data and FDs are not always equally "trustworthy". For example, FDs that were automatically discovered from legacy data may be less reliable than those manually specified by a domain expert. Also, the reliability of data depends on many factors such as data sources and extraction methods. Returning to Example 1, trusting the FD more than the data may suggest changing the `Income` of tuples t_5, t_6 and t_{10} to be equal to the income of t_3, t_9 and t_8, respectively, while keeping the FD unchanged. Trusting the data more the than the FD could lead to modifying the FD to be `Surname, GivenName, Birthdate, Phone` $\rightarrow$ `Income`, while keeping the data unchanged.

In this paper, we argue that the notion of *relative trust* between data and FDs must be taken into account when resolving inconsistencies. We propose an algorithm that generates multiple suggestions for how to modify the data and/or the FDs in a minimal and non-redundant way, corresponding to various levels of relative trust. These suggested repairs can help users and domain experts determine the best way to resolve inconsistencies.

Returning to Example 1, it is not clear whether we should modify the data alone, or add `Birthdate` (or also `Phone`) to the left-hand-side of the FD and resolve any remaining violations by modifying the data. Without complete knowledge of the data semantics, these possibilities may not be obvious by manually inspecting the data and the FDs. Computing various alternative fixes corresponding to different relative trust levels can give users a better idea of what could be wrong with their data and their constraints, and how to resolve problems.

Implementing the proposed relative-trust-aware approach requires solving the following technical challenges:

- **Minimality of changes**. As in previous work on data repair [3], [5], [7], [11], the suggested modifications to the data and the FDs should be non-redundant and (approximately) minimal. However, it is not obvious how to define non-redundancy and minimality when both the data and the FDs can be modified, especially if we want to produce multiple suggestions, not just a single one that is globally minimal according to a unified cost model. Furthermore, finding a data repair with the fewest possible changes is already NP-hard even if the FDs cannot be modified.
- **Specifying Relative Trust**. In previous work on simultaneously repairing data and FDs [6], the level of relative trust was fixed and implicitly encoded in the unified cost model. Since we want to produce multiple suggested repairs corresponding to various levels of relative trust, we need to define a semantically meaningful metric for relative trust.

In this paper, we address a data cleaning problem in which we are given a set of FDs and a data set that does not comply with the specified FDs, and we return multiple non-redundant suggestions (corresponding to different levels of relative trust) for how to modify the data and/or the FDs in order to achieve consistency. We make the following technical contributions:

- We define a space of minimal FD and data repairs based on dominance with respect to the amount of data changes and the amount of changes to the FDs. We propose a simple definition of relative trust, in which a parameter τ specifies the maximum number of allowed data changes; the smaller the τ, the greater the trust in the data.
- We give an efficient algorithm for finding minimal modifications to a set of FDs such that no more than τ data modifications will be required to satisfy the modified FDs. The algorithm prunes the space of possible FD modifications using A* search combined with a novel heuristic that estimates the distance to an optimal set of FD modifications. Intuitively, this algorithm computes a minimal repair of the FDs for the relative trust level specified by τ. We then give an algorithm that lists the required data modifications. Since computing the fewest data changes required to satisfy a set of FDs is NP-hard, we resort to approximation. The suggested repairs generated by our algorithm are *provably* close to our minimality criteria on the data changes. The approximation factor only depends on the number of FDs and the number of attributes.
- Using the above algorithm as a subroutine, we give a technique for generating multiple suggested repairs corresponding to a range of relative trust values. We optimize this technique by reusing repairs using higher values of τ to obtain repairs for smaller τ.

	A	B	C	D
t_1	1	1	1	1
t_2	v_1^A	2	1	3
t_3	2	3	1	1
t_4	v_2^A	3	v_1^C	3

Possible instantiation

	A	B	C	D
t_1	1	1	1	1
t_2	3	2	1	3
t_3	2	3	1	1
t_4	4	3	2	3

Fig. 2. An example of a V-instance and a possible instantiation

Finally, we perform various experiments that justify the need to incorporate relative trust in the data cleaning process and we show order-of-magnitude performance improvements of the proposed algorithms over straightforward approaches.

The remainder of the paper is organized as follows. Section II gives the notation and definitions used in the paper. In Section III, we introduce the concepts of minimal repairs and relative trust. Section IV introduces our approach to finding a nearly-minimal repair for a given relative trust value, followed by a detailed discussion of modifying the FDs in Section V and modifying the data in Section VI. Section VII presents the algorithm for efficiently generating multiple suggested repairs. Section VIII presents our experimental results, Section IX discusses related work, and Section X concludes the paper.

II. Preliminaries

Let R be a relation schema consisting of m attributes, denoted $\{A_1, \ldots, A_m\}$. Let $|R|$ be the number of attributes in R. $Dom(A)$ denotes the domain of an attribute $A \in R$. We assume that attribute domains are unbounded. An instance I of R is a set of tuples, each of which belongs to the domain $Dom(A_1) \times \cdots \times Dom(A_m)$. We refer to an attribute $A \in R$ of a tuple $t \in I$ as a *cell*, denoted $t[A]$.

For two attribute sets $X, Y \subseteq R$, a functional dependency (FD) $X \rightarrow Y$ holds on an instance I, denoted $I \models X \rightarrow Y$, iff for every two tuples t_1, t_2 in I, $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$. Let Σ be the set of FDs defined over R. We denote by $|\Sigma|$ the number of FDs in Σ. We say that I satisfies Σ, written $I \models \Sigma$, iff the tuples in I do not violate any FD in Σ. We assume that Σ is minimal [1], and each FD is of the form $X \rightarrow A$, where $X \subset R$ and $A \in R$.

We use the notion of V-instances, which was first introduced in [11], to concisely represent multiple data instances. In V-instances, cells can be set to variables that may be instantiated in a specific way.

Definition 1: **V-instance**. Given a set of variables $\{v_1^A, v_2^A, \ldots\}$ for each attribute $A \in R$, a V-instance I of R is an instance of R where each cell $t[A]$ in I can be assigned to either a constant in $Dom(A)$, or a variable v_i^A.

A V-instance I represents multiple (ground) instances of R that can be obtained by assigning each variable v_i^A to any value from $Dom(A)$ that is not among the values of A already occurring in I, and such that no two distinct variables v_i^A and v_j^A can have equal values. For example, Figure 2 shows a V-instance that contains variables v_1^A, v_2^A and v_1^C. One instantiation of this V-instance is also shown in Figure 2 by assigning v_1^A, v_2^A, and v_1^C to 3,4 and 2, respectively. For brevity, we refer to a V-instance as an instance in the remainder of the paper.

We say that a vector $X = (x_1, \ldots, x_k)$ dominates another vector $Y = (y_1, \ldots, y_k)$, written $X \prec Y$, iff for $i \in \{1, \ldots, k\}$, $x_i \leq y_i$, and at least one element x_j in X is strictly less than the corresponding element y_j in Y.

III. Spaces of Possible Repairs

In this section, we define a space of minimal repairs of data and FDs (Section III-A), and we present our notion of relative trust (Section III-B).

A. Minimal Repairs of Data and FDs

We consider data repairs that change cells in I rather than deleting tuples from I. We denote by $\mathcal{S}(I)$ all possible repairs of I. All instances in $\mathcal{S}(I)$ have the same number of tuples as I. Because we aim at modifying a given set of FDs, rather than discovering a new set of FDs from scratch, we restrict the allowed FD modifications to those that relax (i.e., weaken) the supplied FDs. We do not consider adding new constraints. That is, Σ' is a possible modification of Σ iff $I \models \Sigma$ implies $I \models \Sigma'$, for any data instance I. Given a set of FDs Σ, we denote by $\mathcal{S}(\Sigma)$ the set of all possible modifications of Σ resulting from relaxing the FDs in Σ in all possible ways. We define the universe of possible repairs as follows.

Definition 2: **Universe of Data and FDs Repairs.** Given a data instance I and a set of FDs Σ, the universe of repairs of data and FDs, denoted $\mathbf{U}$, is the set of all possible pairs (Σ', I') such that $\Sigma' \in \mathcal{S}(\Sigma)$, $I' \in \mathcal{S}(I)$, and $I' \models \Sigma'$.

We focus on a subset of repairs in $\mathbf{U}$ that are Pareto-optimal with respect to two distance functions: $dist_c(\Sigma, \Sigma')$ that measures the distance between two sets of FDs, and $dist_d(I, I')$ that measures the distance between two database instances. We refer to such repairs as *minimal repairs*, defined as follows.

Definition 3: **Minimal Repair.** Given an instance I and a set of FDs Σ, a repair $(\Sigma', I') \in \mathbf{U}$ is minimal iff $\nexists (\Sigma'', I'') \in \mathbf{U}$ such that $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), dist_d(I, I'))$.

We deliberately avoid aggregating changes to data and changes to FDs into one metric in order to enable using various metrics for measuring both types of changes, which might be incomparable. For example, one metric for measuring changes in Σ is the number of modified FDs in Σ, while changes in I could be measured by the number of changed cells. Also, this approach specifies a wide spectrum of Pareto-optimal repairs that ranges from completely trusting I (and only changing Σ) to completely trusting Σ (and only changing I).

For a repair I' of I, we denote by $\Delta_d(I, I')$ the cells that have different values in I and I'. We use the cardinality of $\Delta_d(I, I')$ to measure the distance between I and I', which has been widely used in previous data cleaning techniques (e.g., [5], [7], [11]). That is, $dist_d(I, I') = |\Delta_d(I, I')|$.

Recall that we restrict the modifications to Σ to those that relax the constraints in Σ. Thus, an FD F' is a possible modification of an FD F iff $I \models F \Rightarrow I \models F'$, for any instance I. We use a simple relaxation mechanism: we only allow appending zero or more attributes to the left-hand-side (LHS) of an FD. Formally, an FD $X \rightarrow A \in \Sigma$ can be modified by appending a set of attributes $Y \subseteq (R \setminus XA)$ to the LHS, resulting in an FD $XY \rightarrow A$. We disallow adding A to the LHS to prevent producing trivial FDs.

Note that different FDs in Σ might be modified to the same FD. For example, both $A \rightarrow B$ and $C \rightarrow B$ can be modified to $AC \rightarrow B$. Therefore, the number of FDs in any $\Sigma' \in \mathcal{S}(\Sigma)$ is less than or equal to the number of FDs in Σ. We maintain a mapping between each FD in Σ and its corresponding repair in Σ'. Without loss of generality, we assume hereafter that $|\Sigma'| = |\Sigma|$ by allowing duplicate FDs in Σ'.

We define the distance between two sets of FDs as follows. For $\Sigma = \{X_1 \rightarrow A_1, \ldots, X_z \rightarrow A_z\}$ and $\Sigma' = \{Y_1X_1 \rightarrow A_1, \ldots, Y_zX_z \rightarrow A_z\}$, the term $\Delta_c(\Sigma, \Sigma')$ denotes a vector $(Y_1, \ldots, Y_z)$, which consists of LHS extensions to the FDs. To measure the distance between Σ and Σ', we use the function $\sum_{Y \in \Delta_c(\Sigma, \Sigma')} w(Y)$, where $w(Y)$ is a weighting function that determines the relative penalty of adding a set of attributes Y. The weighting function $w(.)$ is intuitively non-negative and monotone (i.e., for any two attribute sets X and Y, $X \subseteq Y$ implies that $w(X) \leq w(Y)$). A simple example of $w(Y)$ is the number of attributes in Y. However, this does not distinguish between attributes that have different characteristics. Other features of appended attributes can be used for obtaining other definitions of $w(.)$. For example, consider two attributes A and B that could be appended to the LHS of an FD, where A is a key (i.e., $A \rightarrow R$), while B is not. Intuitively, appending A should be more expensive that appending B because the new FD in the former case is trivially satisfied. In general, the more informative a set of attributes is, the more expensive it is when being appended to the LHS of an FD. The information captured by a set of attributes Y can be measured using various metrics, such as the number of distinct values of Y in I, and the entropy of Y. Another definition of $w(Y)$ could rely on the change in the description length for modeling I using FDs due to appending Y (refer to [6], [12]).

In general, $w(Y)$ depends on a given data instance to evaluate the weight of Y. Therefore, changing the cells in I during the repair generating algorithm might affect the weights of attributes. We make a simplifying assumption that $w(Y)$ depends only on the initial instance I. This is based on an observation that the number of violations in I with respect to Σ is typically much smaller than the size of I, and thus repairing data does not significantly change the characteristics of attributes such as entropy and the number of distinct values.

B. Relative Trust in Data vs. FDs

We defined a space of minimal repairs that covers a wide spectrum, ranging from repairs that only alter the data, while keeping the FDs unchanged, to repairs that only alter the FDs, while keeping the data unchanged. The idea behind relative trust is to limit the maximum number of cell changes that can be performed while obtaining I' to a threshold τ, and to obtain a set of FDs Σ' that is the closest to Σ and is satisfied by I'. The obtained repair (Σ', I') is called a τ-constrained repair, formally defined as follows.

Definition 4: τ**-constrained Repair** Given an instance I, a set of FDs Σ, and a threshold τ, a τ-constrained repair (Σ', I') is a repair in $\mathbf{U}$ such that $dist_d(I, I') \leq \tau$, and no other repair $(\Sigma'', I'') \in \mathbf{U}$ has $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$.

In other words, a τ-constrained repair is a repair in $\mathbf{U}$ whose distance to I is less than or equal to τ, and which has the minimum distance to Σ across all repairs in $\mathbf{U}$ with distance to I also less than or equal to τ. We break ties by distance to I (i.e., if two repairs have an equal distance to Σ and have distances to I less than or equal to τ, we choose the one closer to I).

Possible values of τ range from 0 to the minimum number of cells changes that must be applied to I in order to satisfy Σ, denoted $\delta_{opt}(\Sigma, I)$. We can also specify the threshold on the number of allowed cell changes as a percentage of $\delta_{opt}(\Sigma, I)$, denoted τ_r (i.e., $\tau_r = \tau / \delta_{opt}(\Sigma, I)$).

The mapping between minimal repairs and τ-constrained repairs is as follows. (1) Each τ-constrained repair is a minimal repair; (2) All minimal repairs can be found by varying the relative trust τ in the range $[0, \delta_{opt}(\Sigma, I)]$, and obtaining the corresponding τ-constrained repairs. Specifically, each minimal repair (Σ', I') is equal to a τ-constrained repair, where τ is in the range defined as follows. Let (Σ'', I'') be the minimal repair with the smallest $dist_d(I, I'')$ that is strictly greater than $dist_d(I, I')$. If such a repair does not exist, let (Σ'', I'') be $(\emptyset, \emptyset)$. The range of τ is defined as follows.

$$\tau \in \begin{cases} [dist_d(I, I'), dist_d(I, I'')) & \text{if } (\Sigma'', I'') \neq (\emptyset, \emptyset) \\ [dist_d(I, I'), \infty) & \text{if } (\Sigma'', I'') = (\emptyset, \emptyset) \end{cases} \tag{1}$$

If $(\Sigma'', I'') = (\emptyset, \emptyset)$, the range $[dist_d(I, I'), \infty)$ corresponds to a unique minimal repair where $dist_d(I, I')$ is equal to $\delta_{opt}(\Sigma, I)$ and $\Sigma' = \Sigma$. We prove these two points in the following theorem (proof is in [4]).

Theorem 1: Each τ-constrained repair is a minimal repair. Each minimal repair (Σ', I') corresponds to a τ-constrained repair, where τ belongs to the range defined in Equation 1.

IV. Computing a Single Repair for a Given Relative Trust Level

There is a strong interplay between modifying the data and the FDs. Obtaining a data instance that is closest to I while satisfying a set of FDs Σ' highly depends on Σ'. Also, obtaining a set of FDs Σ' that is closest to Σ, such that Σ' holds in a given data instance I', highly depends on I'. This interplay represents the main challenge in simultaneously modifying the data and the FDs.

For example, consider a simple approach that alternates between editing the data and modifying the FDs until we reach consistency. This may not give a minimal repair (e.g., we might make a data change in one step that turns out to be redundant after we change one of the FDs in a subsequent step). Furthermore, we may have to make more than τ cell changes because it is difficult to predict the amount of necessary data changes while modifying the FDs.

Our solution to generating a minimal repair for a given level of relative trust consists of two steps. In the first step, we modify the FDs to obtain a set Σ' that is as close as possible to Σ, while guaranteeing that there exists a data repair I' satisfying Σ' with a distance to I less than or equal to τ. In the second step, we materialize the data instance I' by modifying I with respect to Σ' in a minimal way. We describe this approach in Algorithm 1.

Algorithm 1 `Repair_Data_FDs(`Σ`,`I`,`τ`)`

1: obtain Σ' from $\mathcal{S}(\Sigma)$ such that $\delta_{opt}(\Sigma', I) \leq \tau$, and no other $\Sigma'' \in \mathcal{S}(\Sigma)$ with $\delta_{opt}(\Sigma'', I) \leq \tau$ has $dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$. (ties are broken using $\delta_{opt}(\Sigma', I)$)
2: **if** $\Sigma' \neq \emptyset$ **then**
3:     obtain I' that satisfies Σ' while performing at most $\delta_{opt}(\Sigma', I)$ cell changes, and return (Σ', I').
4: **else**
5:     Return $(\emptyset, \emptyset)$
6: **end if**

Finding Σ' in the first step requires computing the minimum number of cell changes in I to satisfy Σ' (i.e., $\delta_{opt}(\Sigma', I)$). Note that computing $\delta_{opt}(\Sigma', I)$ does not require materializing an instance I' that satisfies Σ' and has the fewest changes. Instead, we collect enough statistics about FD violations to compute $\delta_{opt}(\Sigma', I)$. We will discuss this step in more detail in Section V. Obtaining a modified instance I' in line 3 will be discussed in Section VI.

The following theorem establishes the link between the repairs generated by Algorithm 1 and Definition 4. The proof is in [4].

Theorem 2: Repairs generated by Algorithm 1 are τ-constrained repairs.

A key step in Algorithm 1 is computing $\delta_{opt}(\Sigma', I)$ – the minimum number of cells in I that have to be changed in order to satisfy Σ'. Unfortunately, computing the exact minimum number of cell changes when Σ' contains more than one FD is NP-hard [11]. We propose an approximate solution based on upper-bounding the minimum number of necessary cell changes. Assume that there exists a P-approximate upper bound on $\delta_{opt}(\Sigma', I)$, denoted $\delta_P(\Sigma', I)$ (details are in Section VI). That is, $\delta_{opt}(\Sigma', I) \leq \delta_P(\Sigma', I) \leq P \cdot \delta_{opt}(\Sigma', I)$, for some constant P. By using $\delta_P(\Sigma', I)$ in place of $\delta_{opt}(\Sigma', I)$ in Algorithm 1, we can satisfy the criteria in Definition 4 in a P-approximate way. Specifically, the repair generated by Algorithm 1 becomes a P-approximate τ-constrained repair, which is defined as follows (the proof is similar to Theorem 2).

Definition 5: P**-approximate** τ**-constrained Repair** Given an instance I, a set of FDs Σ, and a threshold τ, a P-approximate τ-constrained repair (Σ', I') is a repair in $\mathbf{U}$ such that $dist_d(I, I') \leq \tau$, and no other repair $(\Sigma'', I'') \in \mathbf{U}$ has $(dist_c(\Sigma, \Sigma''), P \cdot dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$.

In the remainder of this paper, we present an implementation of line 1 (Section V) and line 3 (Section VI) of Algorithm 1. Our implementation is P-approximate, as defined above, with

	A	B	C	D
t_1	1	1	1	1
t_2	1	2	1	3
t_3	2	2	1	1
t_4	2	3	4	3

$\Sigma = \{ A \rightarrow B, C \rightarrow D \}$

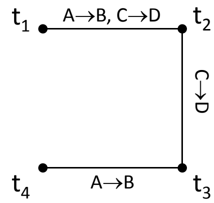

Fig. 3. An example of a conflict graph

$P = 2 \cdot \min\{|R| - 1, |\Sigma|\}$, where $|R|$ denotes the number of attributes in R, and $|\Sigma|$ denotes the number of FDs in Σ.

V. Minimally Modifying the FDs

In this section, we compute a modified set of FDs Σ' that is part of a P-approximate τ-constrained repair (line 1 of Algorithm 1). That is, we need to obtain $\Sigma' \in \mathcal{S}(\Sigma)$ such that $\delta_P(\Sigma', I) \leq \tau$, and no other FD set $\Sigma'' \in \mathcal{S}(\Sigma)$ with $\delta_P(\Sigma'', I) \leq \tau$ has $dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$.

First, we need to introduce the notion of a conflict graph of I with respect to Σ, which was previously used in [2]:

Definition 6: **Conflict Graph.** A conflict graph of an instance I and a set of FDs Σ is an undirected graph whose set of vertices is the set of tuples in I, and whose set of edges consists of all edges (t_i, t_j) such that t_i and t_j violate at least one FD in Σ.

Figure 3 shows an instance I, a set of FDs Σ, and the corresponding conflict graph. The label of each edge represents the FDs that are violated by the edge vertices.

In Section VI, we present an algorithm for obtaining an instance repair I' that satisfies a set of FDs $\Sigma' \in \mathcal{S}(\Sigma)$. The number of cell changes performed by our algorithm is linked to the conflict graph of Σ' and I as follows. Let $C_{2opt}(\Sigma', I)$ be a 2-approximate minimum vertex cover of the conflict graph of Σ' and I, which we can obtain in PTIME using a greedy algorithm [8]. The number of cell changes performed by our algorithm is at most $\alpha \cdot |C_{2opt}(\Sigma', I)|$, where $\alpha = \min\{|R| - 1, |\Sigma|\}$. Moreover, we prove that the number of changed cells is 2α-approximately minimal. Therefore, we define $\delta_P(\Sigma', I)$ as $\alpha \cdot |C_{2opt}(\Sigma', I)|$, which represents a 2α-approximate upper bound of $\delta_{opt}(\Sigma', I)$ that can be computed in PTIME. Based on the definition of $\delta_P(\Sigma', I)$, our goal in this section can be rewritten as follows: obtain $\Sigma' \in \mathcal{S}(\Sigma)$ such that $C_{2opt}(\Sigma', I) \leq \frac{\tau}{\alpha}$, and no other FD set $\Sigma'' \in \mathcal{S}(\Sigma)$ with $C_{2opt}(\Sigma'', I) \leq \frac{\tau}{\alpha}$ has $dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$.

Figure 4 depicts several possible modifications of Σ from Figure 3, along with $dist_c(\Sigma, \Sigma')$ (assuming that the weighting function $w(Y)$ is equal to $|Y|$), the corresponding conflict graph, $C_{2opt}(\Sigma', I)$, and $\delta_P(\Sigma', I)$. For $\tau = 2$, the modifications of Σ that are part of P-approximate τ-constrained repairs are $\{CA \rightarrow B, C \rightarrow D\}$ and $\{DA \rightarrow B, C \rightarrow D\}$.

A. Searching the Space of FD Modifications

We model the possible FD modifications $\mathcal{S}(\Sigma)$ as a state space, where for each $\Sigma' \in \mathcal{S}(\Sigma)$, there exists a state representing $\Delta_c(\Sigma, \Sigma')$ (i.e., the vector of attribute sets appended to LHSs of FDs to obtain Σ'). Additionally, we call $\Delta_c(\Sigma, \Sigma')$ a *goal state* iff $\delta_P(\Sigma', I) \leq \tau$, for a given threshold value τ (or equivalently, $C_{2opt}(\Sigma', I) \leq \frac{\tau}{\alpha}$). The cost of a state $\Delta_c(\Sigma, \Sigma')$ is equal to $dist_c(\Sigma, \Sigma')$. We assume that the weighting function $w(.)$ is monotone and non-negative. Our goal is to locate the cheapest goal state for a given value of τ, which amounts to finding an FD set Σ' that is part of a P-approximate τ-constrained repair.

Σ'	$dist_c(\Sigma, \Sigma')$	Conflict Graph Edges	$C_{2opt}(\Sigma', I)$	$\delta_P(\Sigma', I)$
A→B, C→D	0	(t_1,t_2), (t_2,t_3), (t_3,t_4)	t_2, t_3	4
CA→B, C→D	1	(t_1,t_2), (t_2,t_3)	t_2	2
DA→B, C→D	1	(t_1,t_2), (t_2,t_3)	t_2	2
A→B, **A**C→D	1	(t_1,t_2), (t_3,t_4)	t_1, t_3	4
A→B, **B**C→D	1	(t_1,t_2), (t_2,t_3), (t_3,t_4)	t_2, t_3	4
CA→B, **A**C→D	2	(t_1,t_2)	t_1	2
...		...	...	...

Fig. 4. An example of multiple FD repairs

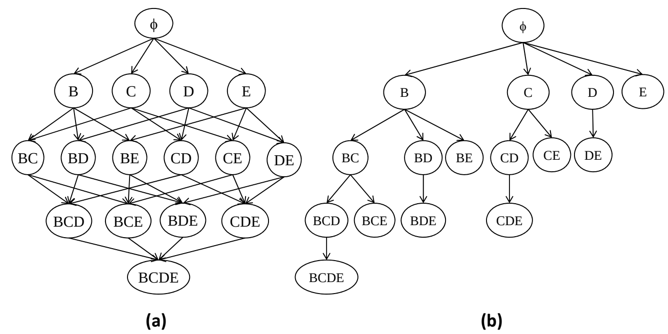

Fig. 5. The state search space for $R = \{A, B, C, D, E, F\}$ and $\Sigma = \{A \rightarrow F\}$: (a) a graph search space (b) a tree search space

The monotonicity of the weighting function w (and hence the monotonicity of the overall cost function) allows for pruning a large part of the state space. We say that a state $(Y'_1, \ldots, Y'_z)$ *extends* another state $(Y_1, \ldots, Y_z)$, where $z = |\Sigma|$, iff for all $i \in \{1, \ldots, z\}$, $Y_i \subseteq Y'_i$. Clearly, if $(Y_1, \ldots, Y_z)$ is a goal state, we can prune all states extending $(Y_1, \ldots, Y_z)$.

In Figure 5(a), we show all the states for $R = \{A, B, C, D, E, F\}$ and $\Sigma = \{A \rightarrow F\}$. Each arrow in Figure 5(a) indicates that the destination state extends the source state by adding exactly one attribute. We can find the cheapest goal state by traversing the graph in Figure 5(a). For example, we can use a level-wise breadth-first search strategy [14], which iterates over states with the same number of attributes, and, for each such set of states, we determine whether any state is a goal state. If one or more goal states are found at the current level, we return the cheapest goal state and terminate the search.

We can optimize the search by adopting best-first traversal of the states graph [14]. That is, we maintain a list of states to be visited next, called the *open list*, which initially contains the state $(\emptyset, \ldots, \emptyset)$, and a list of states that have been visited, called the *closed list*. In each iteration, we pick the cheapest state S from the open list, and test whether S is a goal state. If S is a goal state, we return it and terminate the search. Otherwise, we add S to the closed list, and we insert into the

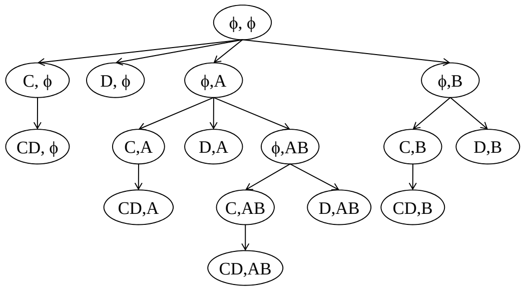

Fig. 6. A space for $R = \{A, B, C, D\}$ and $\Sigma = \{A \rightarrow B, C \rightarrow D\}$

open list all the states that extend S by exactly one attribute and are not in the closed list.

We can avoid using a closed list that keeps track of visited states, and hence reduce the running time, by ensuring that each state can only be reached from the initial state $(\emptyset, \ldots, \emptyset)$ using a unique path. In other words, we need to reduce the graph in Figure 5(a) to a tree (e.g., Figure 5(b)). To achieve this, we assign each state, except $(\emptyset, \ldots, \emptyset)$, to a single parent. Assume that attributes in R are totally ordered (e.g., lexicographically). For Σ with a single FD, the parent of a state Y is another state $Y \setminus \{A\}$ where A is the greatest attribute in Y. Figure 5(b) shows the search tree that is equivalent to the search graph in Figure 5(a). In general, when Σ contains multiple FDs, the parent of a state $(Y_1, \ldots, Y_z)$ is determined as follows. Let A be the greatest attribute in $\bigcup_{i=1}^{z} Y_i$, and j be the index of the last element in the vector $(Y_1, \ldots, Y_z)$ that contains A. The parent of the state $(Y_1, \ldots, Y_z)$ is another state $(Y_1, \ldots, Y_{j-1}, Y_j \setminus \{A\}, Y_{j+1}, \ldots, Y_z)$. Figure 6 depicts an example search space for the two FDs shown in Figure 3.

B. A-based Search Algorithm*

One problem with best-first tree traversal is that it might visit cheap states that will only lead to expensive goal states or no goal states at all. The A* search algorithm [14] avoids this by estimating the cost of the cheapest goal state reachable (i.e., descending) from each state S in the open list, denoted $\underline{gc}(S)$, and visiting the state with the smallest $\underline{gc}(S)$ first. In order to ensure soundness of the algorithm (i.e., returning the cheapest goal state), we must not overestimate the cost of the cheapest goal state reachable from a state S [14].

Algorithm 2 describes the search procedure. The goal of lines 1 and 12-16, along with the sub-procedure `getDescGoalStates`, is computing $\underline{gc}(S)$. The reminder of Algorithm 2 follows the A* search algorithm: it initializes an open list, which is implemented as a priority queue called PQ, by inserting the root state $(\emptyset, \ldots, \emptyset)$. In each iteration, the algorithm removes the state with the smallest value of $\underline{gc}(S)$ from PQ and checks whether it is a goal state. If so, the algorithm returns the corresponding FD set. Otherwise, the algorithm inserts the children of the removed state into PQ, after computing $\underline{gc}(.)$ for each inserted state.

The two technical challenges of computing $\underline{gc}(S)$ are the tightness of the bound $\underline{gc}(S)$ (i.e., being close to the actual cost of the cheapest goal state descending from S), and the computational complexity. In the following, we describe how we address these challenges.

Algorithm 2 `Modify_FDs` (Σ, I, τ)

1: construct the conflict graph G of Σ and I, and obtain the set of all difference sets in G, denoted $\mathcal{D}$
2: $PQ \leftarrow \{(\emptyset, \ldots, \emptyset)\}$
3: **while** PQ is not empty **do**
4:   pick the state S_h with the smallest value of $\underline{gc}(.)$ from PQ
5:   let Σ_h be the FD set corresponding to S_h
6:   Compute $C_{2opt}(\Sigma_h, I)$
7:   **if** $|C_{2opt}(\Sigma_h, I)| \cdot \min\{|R| - 1, |\Sigma|\} \leq \tau$ **then**
8:     **return** Σ_h
9:   **end if**
10:   remove S_h from PQ
11:   **for** each state S_i that is a child of S_h **do**
12:     let Σ_i be the FD set corresponding to S_i
13:     let $\mathcal{D}_s$ be the subset of difference sets in $\mathcal{D}$ that violate Σ_i
14:     let G_0 be an empty graph
15:     $minStates \leftarrow$ `getDescGoalStates`$(S_i, S_i, G_0, \mathcal{D}_s, \tau)$
16:     set $\underline{gc}(S_i)$ to the minimum cost across all states in $minStates$, or ∞ if $minStates$ is empty
17:     **if** $\underline{gc}(S_i)$ is not ∞ **then**
18:       insert S_i into PQ
19:     **end if**
20:   **end for**
21: **end while**
22: **return** $\emptyset$

Given a conflict graph G of I and Σ, each edge represents two tuples in I that violate Σ. For any edge (t_i, t_j) in G, we refer to the attributes that have different values in t_i and t_j as the *difference set* of (t_i, t_j). Difference sets have been introduced in the context of FD discovery (e.g., [13], [15]). For example, the difference sets for (t_1, t_2), (t_2, t_3), and (t_3, t_4) in Figure 3 are BD, AD, and BCD, respectively. We denote by $\mathcal{D}$ the set of all difference sets for edges in G (line 1 in Algorithm 2). The key idea that allows efficient computation of $\underline{gc}(S)$ is that all edges (i.e., violations) in G with the same difference set can be completely resolved by adding one attribute from the difference set to the LHS of each violated FD in Σ. For example, edges corresponding to difference set BD in Figure 3 violate both $A \rightarrow B$ and $C \rightarrow D$, and to fix these violations, we need to add D to the LHS of the first FD, and B to the LHS of the second FD. Similarly, fixing violations corresponding to difference set BCD can be done by adding C or D to the first FD (second FD is not violated). Therefore, we partition the edges of the conflict graph G based on their difference sets. In order to compute $\underline{gc}(S)$, each group of edges corresponding to one difference set is considered atomically, rather than individually.

Let $\mathcal{D}_s$ be a *subset* of difference sets that are still violated at the current state S_i (line 13). Given a set of difference sets $\mathcal{D}_s$, the recursive procedure `getDescGoalStates`$(S, S_c, G_c, \mathcal{D}_c, \tau)$ (Algorithm 3) finds all minimal goal states descending from S that resolve $\mathcal{D}_c$, taking into consideration the maximum number of allowed cell changes τ. Therefore, $\underline{gc}(S)$ can be assigned to the cheapest state returned by the procedure getDescGoalStates. Note that we use a subset of difference sets that are still violated ($\mathcal{D}_s$), instead of using all violated difference sets, in order to efficiently compute $\underline{\text{gc}}(S)$. The computed value of $\underline{\text{gc}}(S)$ is clearly a lower bound on the cost of actual cheapest goal state descending from the current state S. To provide tight lower bounds, $\mathcal{D}_s$ is selected such that difference sets corresponding to large numbers of edges are favored. Additionally, we heuristically ensure that the difference sets in $\mathcal{D}_s$ have a small overlap.

We now describe Algorithm 3. It recursively selects a difference set d from the set of non-resolved difference sets $\mathcal{D}_c$. For each difference set d, we consider two alternatives: (1) excluding d from being resolved, if threshold τ permits, and (2) resolving d by extending the current state S_c. In the latter case, we consider all possible children of S_c to resolve d. Once S_c is extended to S'_c, we remove from $\mathcal{D}_c$ all the sets that are now resolved, resulting in $\mathcal{D}'_c$. Due to the monotonicity of the cost function, we can prune all the non-minimal states from the found set of states. That is, if state S_1 extends another state S_2 and both are goal states, we remove S_1.

Algorithm 3 getDescGoalStates$(S, S_c, G_c, \mathcal{D}_c, \tau)$

Require: S : the state for which we compute $\underline{\text{gc}}(.)$
Require: S_c : the current state to be extended (equals S at the first entry)
Require: G_c : the current conflict graph for non-resolved difference sets (is empty at the first entry)
Require: $\mathcal{D}_c$: the remaining difference sets to be resolved
if $\mathcal{D}_c$ is empty **then**
  return $\{S_c\}$
end if
$States \leftarrow \emptyset$
select a difference set d from $\mathcal{D}_c$
let G'_c be the graph whose edges are the union of edges corresponding to d and edges of G_c
compute a 2-approximate minimum vertex cover of G'_c, denoted C_{2opt}
if $|C_{2opt}| \cdot \min\{|R|-1, |\Sigma|\} < \tau$ **then**
  $\mathcal{D}'_c \leftarrow \mathcal{D}_c \setminus \{d\}$
  $States \leftarrow States \cup$ getDescGoalStates$(S, S_c, G'_c, \mathcal{D}'_c, \tau)$
end if
for each possible state S'_c that extends S_c, is descendant of S, and resolves violations corresponding to d **do**
  let $\mathcal{D}'_c$ be all difference sets in $\mathcal{D}_c$ that are still violating Σ'_c that is corresponding to S'_c
  $States \leftarrow States \cup$ getDescGoalStates$(S, S'_c, G_c, \mathcal{D}'_c, \tau)$
end for
remove any non-minimal states from $States$
return $States$

In the following lemma, we prove that the computed value of $\underline{\text{gc}}(S)$ is a lower bound on the cost of the cheapest goal descending from state S. The proof is in [4].

Lemma 1: For any state S, $\underline{\text{gc}}(S)$ is less than or equal to the cost of the cheapest goal state descendant of S.

Based on Lemma 1, and the correctness of the A* search algorithm [14], we conclude that the FD set generated by Algorithm 2 is part of a P-approximate τ-constrained repair.

We now discuss the complexity of Algorithms 2 and 3. Finding all difference sets in line 1 in Algorithms 2 costs $O(|\Sigma| \cdot n + |\Sigma| \cdot |E| + |R| \cdot |E|)$, where n denotes the number of tuples in I, and E denotes the number of edges in the conflict graph of I and Σ. Difference sets are obtained by building the conflict graph of I and Σ, which costs $O(|\Sigma| \cdot n + |\Sigma| \cdot |E|)$ (more details are in Section VI), and then computing the difference set for all edges, which costs $O(|R| \cdot |E|)$. In the worst case, Algorithm 2, which is based on A* search, will visit a number of states that is exponential in the depth of the cheapest goal state [14], which is less than $|\Sigma| \cdot (|R|-2)$. However, the number of states visited by an A* search algorithm is the minimum across all algorithms that traverse the same search tree and use the same heuristic for computing $\underline{\text{gc}}(S)$. Also, we show in our experiments that the actual number of visited states is much smaller than the best-first search algorithm (Section VIII).

The worst-case complexity of Algorithm 3 that finds $\underline{\text{gc}}(S)$ is $O(|E| \cdot |R|^{|\Sigma| \cdot |\mathcal{D}_c|})$, where $|\mathcal{D}_c|$ is the number of difference sets passed to the algorithm. This is due to recursively inspecting each difference set in $\mathcal{D}_c$ and, if not already resolved by the current state S_c, appending one more attribute from the difference set to the LHS of each FD. At each step, approximate vertex graph cover might need to be computed, which can be performed in $O(|E|)$.

VI. Near-Optimal Data Modification

In this section, we derive a P-approximation of $\delta_{opt}(\Sigma', I)$, denoted $\delta_P(\Sigma', I)$, where $P = 2 \cdot \min\{|R|-1, |\Sigma|\}$. We also give an algorithm that makes at most $\delta_P(\Sigma', I)$ cell changes in order to resolve all the inconsistencies with respect to the modified set of FDs computed in the previous section.

There are several data cleaning algorithms that obtain a data repair for a fixed set of FDs, such as [5], [7], [11]. Most approaches do not provide any bounds on the number of cells that are changed during the repairing process. In [11], the proposed algorithm provides an upper bound on the number of cell changes and it is proved to be near-minimum. The approximation factor depends on the set of FDs Σ, which is assumed to be fixed. Unfortunately, we need to deal with multiple FD sets, and the approximation factor described in [11] can grow arbitrarily while modifying the initial FD set. That is, the approximation factors for two possible repairs Σ', Σ'' in $\mathcal{S}(\Sigma)$ can be different. In this section, we provide a method to compute $\delta_P(\Sigma', I)$ such that the approximation factor is equal to $2 \cdot \min\{|R|-1, |\Sigma|\}$, which depends only on the number of attributes in R and the number of FDs in Σ.

The output of our algorithm is a V-instance, which was first introduced in [11] to concisely represent multiple data instances (refer to Section II for more details). In the remainder of this paper, we refer to a V-instance as simply an instance.

The algorithm we propose in this section is a variant of the data cleaning algorithm proposed in [3]. The main difference is that we clean the data tuple-by-tuple instead of cell-by-cell. That is, we first identify a set of clean tuples that satisfy Σ' such that the cardinality of the set is approximately maximum. We convert this problem to the problem of finding the minimum vertex cover, and we use a greedy algorithm with an approximation factor of 2. Then, we iteratively modify violating tuples as follows. For each violating tuple t, we iterate over the attributes of t in a random order, and we modify each attribute, if necessary, to ensure that the attributes processed so far are clean.

Given a set of FDs Σ', the procedure `Repair_Data` in Algorithm 4 generates an instance I' that satisfies Σ'. Initially, the algorithm constructs the conflict graph of I and Σ'. Then, the algorithm obtains a 2-approximate minimum vertex cover of the obtained conflict graph, denoted $C_{2opt}(\Sigma', I)$, using a greedy approach described in [8] (for brevity, we refer to $C_{2opt}(\Sigma', I)$ as C_{2opt} in this section). The clean instance I' is initially set to I. The algorithm repeatedly removes a tuple t from C_{2opt}, and it changes attributes of t to ensure that, for every tuple $t' \in I' \setminus C_{2opt}$, t and t' do not violate Σ' (lines 5-15). This is achieved by repeatedly picking an attribute of t at random, and adding it to a set denoted $Fixed\_Attrs$ (line 9). After inserting an attribute A, we determine whether we can find an assignment to the attributes outside $Fixed\_Attrs$ such (t, t') are not violating Σ', for all $t' \in I' \setminus C_{2opt}$. We use Algorithm 5 to find a valid assignment, if any, or to indicate that no valid assignment exists. Note that when $Fixed\_Attrs$ contains only one attribute (line 6), it is guaranteed that a valid assignment exists (line 7). If a valid assignment is found, we keep $t[A]$ unchanged. Otherwise, we change $t[A]$ to the value of attribute A of the valid assignment found in the previous iteration (line 11). The algorithm proceeds until all tuples have been removed from C_{2opt}. We return I' upon termination.

Algorithm 4 `Repair_Data`(Σ', I)

1: let G be the conflict graph of I and Σ'
2: obtain a 2-approximate minimum vertex cover of G, denoted C_{2opt}
3: $I' \leftarrow I$
4: **while** C_{2opt} is not empty **do**
5:   randomly pick a tuple t from C_{2opt}
6:   $Fixed\_Attrs \leftarrow \{A\}$, where A is a randomly picked attribute from R
7:   $t_c \leftarrow$ `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt})$
8:   **while** $|Fixed\_Attrs| < |R|$ **do**
9:     randomly pick an attribute A from $R \setminus Fixed\_Atts$ and insert it into $Fixed\_Attrs$
10:     **if** `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt}) = \emptyset$ **then**
11:       $t[A] \leftarrow t_c[A]$
12:     **else**
13:       $t_c \leftarrow$ `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt})$
14:     **end if**
15:   **end while**
16:   remove t from C_{2opt}
17: **end while**
18: **return** I'

Algorithm 5 searches for an assignment to attributes of a tuple t that are not in $Fixed\_Attrs$ such that every pair (t, t') satisfies Σ' for all $t' \in I' \setminus C_{2opt}$. An initial assignment t_c is created by setting attributes that are in $Fixed\_Attrs$ to be equal to t, and setting attributes that are not in $Fixed\_Attrs$ to new variables. The algorithm repeatedly selects a tuple $t' \in I' \setminus C_{2opt}$ such that (t, t') violates an FD $X \rightarrow A \in \Sigma'$. If attribute A belongs to $Fixed\_Attrs$, the algorithm returns $\emptyset$, indicating that no valid assignment is available. Otherwise, the algorithm sets $t[A]$ to be equal to $t'[A]$, and adds A to $Fixed\_Attrs$. When no other violations could be found, the algorithm returns the assignment t_c.

Algorithm 5 `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt})$

construct a tuple t_c such that $t_c[A] = t[A]$ if $A \in Fixed\_Attrs$, and $t_c[A] = v_i^A$ if $A \notin Fixed\_Attrs$, where v_i^A is a new variable
while $\exists t' \in I' \setminus C_{2opt}$ such that for some FD $X \rightarrow A \in \Sigma'$, $t_c[X] = t'[X] \wedge t_c[A] \neq t'[A]$ **do**
  if $A \in Fixed\_Attrs$ **then**
    return $\emptyset$
  else
    $t_c[A] \leftarrow t'[A]$
    add A to $Fixed\_Attrs$
  end if
end while
return t_c

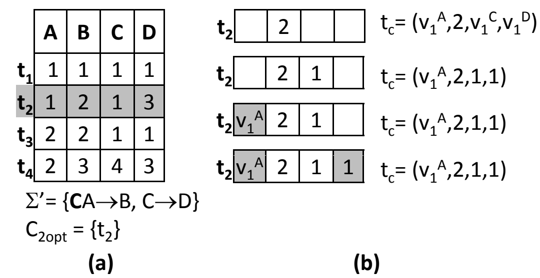

Fig. 7. An example of repairing data: (a) initial value of I', Σ' and C_{2opt} (b) steps of fixing the tuple t_2

In Figure 7, we show an example of generating a data repair for $\Sigma' = \{CA \rightarrow B, C \rightarrow D\}$, given the instance I shown in Figure 7(a). After adding the first attribute B to $Fixed\_Attrs$, the current valid assignment, denoted t_c, is equal to $(v_1^A, 2, v_1^C, v_1^D)$. When inserting C to $Fixed\_Attrs$, there is no need to change the value of C because we can find a valid assignment to the remaining attributes, which is $(v_1^A, 2, 1, 1)$. After inserting A to $Fixed\_Attrs$, no valid assignment is found, and thus we set $t[A]$ to the value of attribute A of the previous valid assignment t_c. Similarly, we set $t[D]$ to $t_c[D]$ after inserting D into $Fixed\_Attrs$. The resulting instance satisfies Σ'. The following lemma proves the soundness and completeness of Algorithm 5. The proof is in the full version of this paper [4].

Lemma 2: Algorithm 5 is both sound (i.e., the obtained assignments are valid) and complete (it will return an assignment if a valid assignment exists).

The following theorem proves the P-optimality of Algorithm 4. The proof is in [4].

Theorem 3: For a given instance I and a set of FDs $\Sigma' \in \mathcal{S}(\Sigma)$, Algorithm `Repair_Data`$(\Sigma', I)$ obtains an instance $I' \models \Sigma'$ such that the number of changed cells in I' is at most $|C_{2opt}(\Sigma', I)| \cdot \min\{|R|-1, |\Sigma|\}$, and it is $2 \cdot \min\{|R|-1, |\Sigma|\}$-

approximate minimum.

We now describe the worst-case complexity of Algorithms 4 and 5. Algorithm 5 has a complexity of $O(|R|+|\Sigma'|)$ because constructing t_c in line 1 costs $O(|R|)$, and the loop in lines 2-9 iterates at most $|\Sigma'|$ times. The reason is that, for each FD $X \rightarrow A \in \Sigma'$, there is at most one tuple in $I' \setminus C_{2opt}$ satisfying the condition in line 2 (otherwise, tuples in $I' \setminus C_{2opt}$ would be violating $X \rightarrow A$).

Constructing the conflict graph in line 1 in Algorithm 4 takes $O(|\Sigma'| \cdot n + |\Sigma'| \cdot |E|)$, where $|\Sigma'|$ is the number of FDs in Σ', n is the number of tuples in I and E is the set of edges in the resulting conflict graph. This step is performed by partitioning tuples in I based on LHS attributes of each FD in Σ' using a hash function, and constructing sub-partitions within each partition based on right-hand-side attributes of each FD. Edges of the conflict graph are generated by emitting pairs of tuples that belong to the same partition and different sub-partitions. The approximate vertex cover is computed in $O(|E|)$ [8]. The loop in lines 4-17 iterates a number of times equal to the size of the vertex cover, which is $O(n)$. Each iteration costs $O(|R| \cdot (|R|+|\Sigma'|))$. To sum up, the complexity of finding a clean instance I' is $O(|\Sigma'| \cdot |E| + |R|^2 \cdot n + |R| \cdot |\Sigma'| \cdot n)$. Assuming that $|R|$ and $|\Sigma'|$ are much smaller than n, the complexity is reduced to $O(|E| + n)$.

VII. Computing Multiple Repairs

So far, we discussed how to modify the data and the FDs for a given value of the relative trust parameter τ. One way to obtain a small sample of possible repairs is to execute Algorithm 1 multiple times while randomly varying the value of τ. This approach can be easily parallelized, but it is inefficient for two reasons. First, multiple values of τ could result in the same repair, and some executions of the algorithm would be redundant. Second, different invocations of Algorithm 2 are expected to visit the same states, which represents a waste of computational resources. To overcome these drawbacks, we develop an algorithm (Algorithm 6) that generates all FD-repairs corresponding to a range of τ values. We can use Algorithm 4 to find the corresponding data modification for each modified FD set.

Algorithm 6 generates repairs corresponding to the threshold range $\tau \in [\tau_l, \tau_u]$. Initially, $\tau = \tau_u$. The search algorithm proceeds by visiting states in order of $\underline{gc}(.)$, and expanding PQ by inserting new states. Once a goal state is found, the corresponding FD repair Σ_h is added to the set of possible repairs. The set Σ_h corresponds to the parameter range $[\delta_P(\Sigma_h, I), \tau]$. Therefore, we set the new value of τ to $\delta_P(\Sigma_h, I) - 1$ in order to discover a new repair. Because the value of $\underline{gc}(.)$ depends on the value of τ, we recompute $\underline{gc}(.)$ for all states in PQ. Note that states that have been previously removed from PQ because they were not goal states (line 13) cannot be goal states with respect to the new value of τ. The reason is that if a state is not a goal state for $\tau = x$, it cannot be a goal state for $\tau < x$ (refer to line 8). The algorithm terminates when PQ is empty or when $\tau < \tau_l$.

Algorithm 6 `Find_Repairs_FDs` $(\Sigma, I, \tau_l, \tau_u)$

1: $PQ \leftarrow \{(\emptyset, \ldots, \emptyset)\}$
2: $\tau \leftarrow \tau_u$
3: $FD\_Repairs \leftarrow \emptyset$
4: **while** PQ is not empty and $\tau \geq \tau_l$ **do**
5: Pick the state S_h with the smallest value of $\underline{gc}(.)$ from PQ
6: Let Σ_h be the FD set corresponding to S_h
7: Compute $C_{2opt}(\Sigma_h, I)$
8: **if** $|C_{2opt}(\Sigma_h, I)| \cdot \min\{|R|-1, |\Sigma|\} \leq \tau$ **then**
9: Add Σ_h to $FD\_Repairs$
10: $\tau \leftarrow |C_{2opt}(\Sigma_h, I)| \cdot \min\{|R|-1, |\Sigma|\} - 1$
11: For each state $S_i \in PQ$, recompute $\underline{gc}(S_i)$ using the new value of τ
12: **end if**
13: Remove S_h from PQ
14: **for** each state S_i that is a child of S_h **do**
15: Compute $\underline{gc}(S_i)$ (similar to Algorithm 2)
16: Insert S_i into PQ
17: **end for**
18: **end while**
19: **return** $FD\_Repairs$

VIII. Experimental Evaluation

In this section, we study the relationship between repair quality and relative trust, and we compare our approach to the technique introduced in [6]. Also, we show the efficiency of our repair generating algorithms.

A. Setup

Experiments were conducted on a SunFire X4100 server with a Quad-Core 2.2GHz processor, and 8GB of RAM. All computations were executed in memory. Repairing algorithms are executed as single-threaded processes, and we limit memory usage to 1.5GB. We use a real data set, namely the Census-Income data set[1], which is part of the UC Irvine Machine Learning Repository. Census-Income consists of 300k tuples and 40 attributes (we only use 34 attributes in our experiments). To perform experiments on smaller data sizes, we randomly pick a sample of tuples.

We tested two variants of Algorithm `Repair_Data_FDs`: A^*`-Repair` which uses the A*-based search algorithm described in Section V-B, and `Best-First-Repair` which uses a best-first search to obtain FD repairs, as we described in Section V. Both variants use Algorithm 4 to obtain the corresponding data repair. We use the number of distinct values to measure the weights of sets of attributes appended to LHS's of FDs (i.e., $w(Y) = F_{count(Y)}\Pi_Y(I)$). In our experiments, we adjust the relative threshold τ_r, rather than the absolute threshold τ. We also implemented the repairing algorithm introduced in [6], which uses a unified cost model to quantify the goodness of each data-FD repair and obtains a repair with the (heuristically) minimum cost.

In order to assess the quality of the generated repairs, we first use an FD discovery algorithm to find all the minimal FDs with a relatively small number of attributes in the LHS (fewer than 6). In each experiment, we randomly select a number of

[1] http://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD)

FDs from the discovered list of FDs. We denote by I_c and Σ_c the clean database instance and the FDs, respectively. The data instance I_c is perturbed by changing the value of some cells such that each cell change results in a violation of an FD. Specifically, we inject two types of violations as follows.

- **Right-hand-side violation.** We first search for two tuples t_i, t_j that agree on XA for some FD $X \rightarrow A \in \Sigma$. Then, we modify $t_i[A]$ to be different from $t_j[A]$.
- **Left-hand-side violation.** We search for two tuples t_i, t_j such that for some FD $X \rightarrow A$, $t_i[X \setminus \{B\}] = t_j[X \setminus \{B\}]$, $t_i[B] \neq t_j[B]$ and $t_i[A] \neq t_j[A]$, where $B \in X$. We introduce a violation by setting $t_i[B]$ to $t_j[B]$.

We refer to the resulting instance as I_d. In our approach, we concentrate on one method of fixing FDs, which is appending one or more attributes to LHS's of FDs. Therefore, we perform FDs perturbation by randomly removing a number of attributes from their LHS's. The perturbed set of FDs is denoted Σ_d. The cleaning algorithm is applied to (Σ_d, I_d), and the resulting repair is denoted (Σ_r, I_r). The parameters that control the perturbation of data and FDs are (1) Data Error Rate, which is the fraction of cells that are modified, and (2) FD Error Rate, which is the fraction of LHS attributes that were removed. We use the following metrics to measure the quality of the modified data and FDs.

- **Data precision:** the ratio of the number of correctly modified cells to the total number of cells modified by the repair algorithm. A modification of a cell $t[A]$ is considered correct if the values of $t[A]$ in I_c and I_d are different, and either $t[A]$ in I_r is equal to $t[A]$ in I_c, or $t[A]$ is a variable in I_r.
- **Data recall:** the ratio of the number of correctly modified cells to the total number of erroneous cells (i.e., cells with different values in I_d and I_c).
- **FD precision:** the ratio of the number of attributes correctly appended to LHS's of FDs in Σ_d to the total number of appended attributes.
- **FD recall:** the ratio of the number of attributes correctly appended to LHS's of FDs in Σ_d to the total number of attributes removed from Σ_c while constructing Σ_d.

In order to measure the overall quality of a repair (Σ_r, I_r), we compute the harmonic averages of precision and recall for both data and FDs (also called F-scores). Then, we compute the average F-score for data and FDs, which we refer to as the *combined F-score*.

B. Impact of Relative Trust on Repair Quality

In this experiment, we measure the combined F-score at various error rates. We use 5000 tuples from the Census-Income data set to represent the clean instance I_c, and we use an FD with 6 LHS attributes to represent Σ_c. Figure 8 shows the combined F-score for various data sets, for multiple values of τ_r. When only FDs perturbation is performed, the peak quality occurs at $\tau_r = 0\%$ (i.e., when no changes to data are allowed). At FD error rate of 50%, the peak quality occurs at $\tau_r = 17\%$. At 30% FD error rate and 5% data error rate,

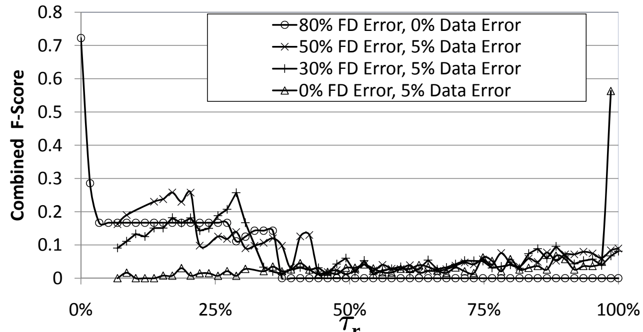

Fig. 8. Repair quality at multiple error rates

	FD Error	Data Error	FD Precision	FD Recall	Data Precision	Data Recall	Combined F-Score
Uniform-Cost Repairing	80%	0%	1	0	0	1	0
	50%	5%	1	0	0.09	0.71	0.08
	30%	5%	1	0	0.04	0.70	0.04
	0%	5%	1	1	0.03	0.69	0.53
Relative-Trust Repairing	80%	0%	0.5	0.4	1	1	0.72
	50%	5%	0.33	1	0.06	0.01	0.26
	30%	5%	0.03	0.01	0.03	0.01	0.26
	0%	5%	0.13	0.13	0.13	0.13	0.56

Fig. 9. The maximum quality achievable by our algorithm and the algorithm in [6]

the peak quality occurs at higher value ($\tau_r = 28.9\%$). Finally, when only data perturbation is performed, the peak quality occurs at $\tau = 100\%$ (i.e., the algorithm can freely change the data, while obtaining the cheapest FD repair, which is the original FD).

In Figure 9, we compare the quality of repairs generated by our algorithm, denoted `Relative-Trust Repairing`, to the quality of repairs generated by the algorithm from [6], denoted `Uniform-Cost Repairing`. For both algorithms, we tested multiple parameter settings and we reported the quality metrics of the repair with the highest combined F-score. For example, for FD error of 50% and data error of 5% our algorithm achieved the maximum combined F-score of 0.26 at $\tau = 17\%$. For all data sets, the algorithm from [6] did not modify the FD using any parameter settings, resulting in FD precision of 1 and recall of 0 for the first three data sets, and recall of 1 for the fourth data set. Because our algorithm is aware of the different levels of relative trust, we achieve higher quality scores when choosing the appropriate value of τ. This is clear in the first data set with FD error of 80% and data error of 0%. Insisting on modifying the data, not the FD, resulted in FD recall of 0, and data precision of 0. On the other hand, when setting τ to 0%, our algorithm kept the data unmodified, resulting in perfect data precision and recall, and changed the FD, resulting in FD precision of 0.5 and FD recall of 0.4.

In general, the precision and recall for data repairs is relatively low due to the high uncertainty about the right cells to modify. For example, given an FD $A \rightarrow B$, and two violating tuples t_1 and t_2, we have four cells that can be changed in order to repair the violation: $t_1[A]$, $t_1[B]$, $t_2[A]$, and $t_2[B]$. This can be reduced by considering additional information such as the user trust in various attributes and tuples (e.g., [5], [6], [11]). Using this information is not considered in our work.

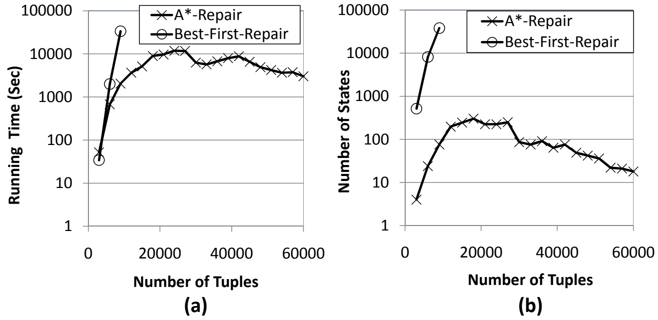

Fig. 10. Performance at various instance sizes

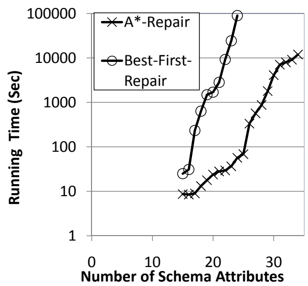

Fig. 11. Salability with number of attributes

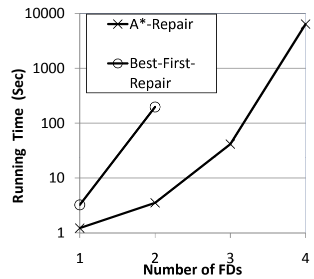

Fig. 12. Salability with number of FDs

C. Performance Results

1) Scalability with the Number of Tuples: In this experiment, we show the scalability of our algorithms with respect to the number of tuples. We use two FDs, and we set τ_r to 1%. Figures 10(a) and 10(b) show the running time, and the number of visited states, respectively, against the number of tuples. When increasing the number of tuples in the range $[0, 20000]$, the number of unique difference sets increases, while the average frequency of difference sets remains relatively small, compared to τ. It follows that the computed lower bounds $\underline{\text{gc}}(S)$ are very loose because most difference sets considered by Algorithm 3 can be left unresolved (i.e., the condition in line 8 is true). Thus, the search algorithm needs to visit more states, as we show in Figure 10(b).

When the number of tuples increases beyond 20000, we notice in Figure 10 that the running time, as well as the number of visited states, decreases. The reason is that, in the state searching algorithm, the number of distinct difference sets stabilizes after reaching a certain number of tuples, and the frequencies of individual difference sets start increasing. It follows that most difference sets can no longer remain unresolved, and tighter lower bounds $\underline{\text{gc}}(S)$ are reported, which leads to decreasing the number of visited states (Figure 10(b)).

Algorithm `Best-First-Repair` does not depend on cost estimation, and thus, the execution time rapidly grows with the number of tuples in the entire range $[0, 60000]$.

2) Scalability with the Number of Attributes: Figure 11 depicts the scalability of our approach with respect to the number of attributes. In this experiment, we used two FDs and 24000 tuples, and we set τ_r to 1%. We changed the number of attributes by excluding some number of attributes from the input relation. The running time increases with the number of attributes mainly because the size of state space increases exponentially with the number of attributes.

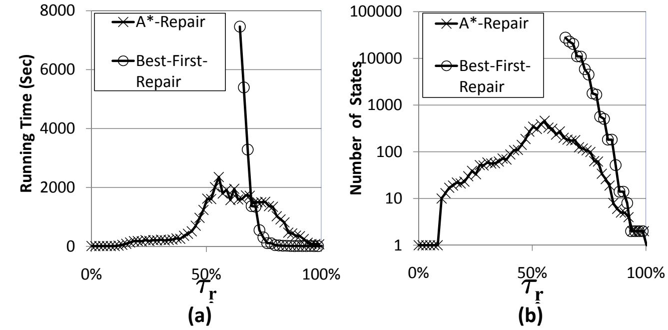

Fig. 13. Effect of τ on (a) running time (b) visited states

3) Scalability with the Number of FDs: Figure 12 depicts the scalability of our approach with the number of FDs. In this experiment, we used 10000 tuples, and we set τ_r to 1%. We use a single FD, and we replicate this FD multiple times to simulate larger sizes of Σ. We avoid having different FDs in Σ since different FDs require varying amount of time to modify, which makes comparing different FD sets with various number of FDs difficult. Note that in our algorithm, each FD in Σ is considered separately from other FDs (even if multiple FDs happened to be exactly the same). The size of state space grows exponentially with the number of FDs. Thus, the searching algorithm visits more states, which increases the overall running time for both approaches: A^*`-Repair` and `Best-First-Repair`. Note that the algorithm `Best-First-Repair` did not terminate in 24 hours when the number of FDs is greater than 2.

4) Effect of the Relative Trust Parameter τ: Figures 13(a) and 13(b) show the running time and the number of visited states, respectively, for various values of τ_r. In this experiment, we fix the number of tuples to be 5000, and we use Σ_d with one FD. The number of appended attributes ranges from 9 at $\tau_r = 10\%$ to 1 at $\tau_r = 99\%$. No repair could be found for τ_r less than 10%. We notice that at small values of τ, Algorithm A^*`-Repair` is orders of magnitude faster than Algorithm `Best-First-Repair`. This is due to the effectiveness (i.e., tightness) of the cost estimation implemented in Algorithm A^*`-Repair`. The lack of such estimation causes Algorithm `Best-First-Repair` to visit many more states.

As the value of τ_r increases up to 55%, Algorithm A^*`-Repair` becomes slower. The reason is that larger values of τ_r decreases the tightness of computed bounds $\underline{\text{gc}}(S)$. As τ_r increases beyond 55%, we notice an improvement in the running time as we only need to add a few attributes to reach a goal state.

5) Generating Multiple Repairs: In this experiment, we assess the efficiency of two approaches that generate possible repairs for a given range of τ_r. In the first approach, denoted `Range-Repair`, we execute Algorithm 6, and we invoke the data repair algorithm (Algorithm 4) for each obtained FD repair. In the second approach, denoted `Sampling-Repair`, we invoke the algorithm A^*`-Repair` at a sample of possible

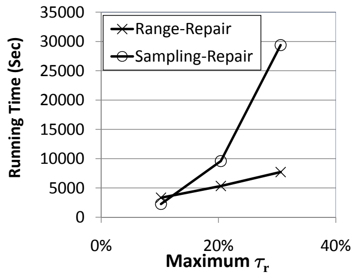

Fig. 14. Performance under uncertain relative trust

values of τ_r. In this experiment, we used 5000 tuples, and one FD. We set the minimum value of τ_r to 0, and we varied the upper bound of τ in the range $[10\%, 30\%]$, which is represented by the X-axis in Figure 14. For the sampling approach, we started by $\tau_r = 0\%$, and we increased τ_r in steps of 1.7% (which is equal to 13 in this experiment) until we reach the maximum value of τ_r. Figure 14 shows the running time for both approaches. We observe that `Range-Repair` outperforms the sampling approach, especially at wide ranges of τ_r. For example, for the range $[0, 30\%]$, `Range-Repair` is 3.8 times faster than `Sampling-Repair`.

IX. Related Work

The closest work to ours is [6], which proposed a technique to obtain a single repair, (Σ', I'), of the FDs and the data, respectively, for a given input (Σ, I). A unified cost model was proposed to measure the distance between a repair (Σ', I') and the inputs (Σ, I). An approximate algorithm was presented that obtains a repair with the minimum cost. There are many differences between our work and [6] including: (1) we incorporate the notion of relative trust in the data cleaning process and produce multiple suggested repairs corresponding to various levels of relative trust; (2) [6] does not give any minimality guarantees for the generated repairs; (3) the algorithm proposed in [6] searches a constrained repair space by only considering adding single attributes to LHS's of FDs in Σ, while we explore a larger repair space that considers appending any subset of R to the LHS of each FD.

The idea of modifying a supplied set of FDs to better fit the data was also discussed in [9]. The goal of that work was to generate a small set of Conditional Functional Dependencies (CFDs) by modifying the embedded FD. Modifying the data and relative trust were not discussed.

The problem of cleaning the data in order to satisfy a fixed set of FDs has been studied in, e.g., [3], [5], [7], [11]. In our context, these solutions may be classified as having a fixed threshold τ_r of 100%. Part of our work is inspired by the algorithm proposed in [3] in the sense that we incrementally modify the data until there are no inconsistencies left. However, we modify individual tuples instead of attribute values. Also, unlike the approach in [3], we provide an upper bound on the number of changes.

Another related problem is discovering which FDs hold (approximately or exactly) on a fixed database instance (e.g., [10], [12], [13], [15]). There are two important differences between these approaches and our work: (1) instead of discovering the FDs from scratch, we start with a set of provided FDs which have a certain level of trust, and we aim for a minimal modification of the provided FDs that yields at most τ violations; (2) in previous work, there are only "local" guarantees on the goodness (i.e., the number of violating tuples) of each FD, whereas in this paper we must make "global" guarantees that the whole set of FDs cannot be violated by more than τ tuples. Thus, existing techniques for FD discovery are not applicable to our problem.

X. Conclusions

In this paper, we studied a data quality problem in which we are given a data set that does not satisfy the specified integrity constraints (namely, Functional Dependencies), and we are uncertain whether the data or the FDs are incorrect. We proposed a solution that computes various suggestions for how to modify the data and/or the FDs (in a nearly-minimal way) in order to achieve consistency. These suggestions cover various points on the spectrum of relative trust between the data and the FDs, and can help users determine the best way to resolve inconsistencies in their data. We believe that our relative trust framework is relevant and applicable to many other types of constraints, such as conditional FDs (CFDs), inclusion dependencies and denial constraints. In future work, we plan to develop cleaning algorithms within our framework for these constraints.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Scalar aggregation in fd-inconsistent databases. In *ICDT*, pages 39–53, 2001.

[3] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010.

[4] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. Technical report, 2012. http://arxiv.org/abs/1207.5226.

[5] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

[6] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, pages 446–457, 2011.

[7] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[9] L. Golab, H. J. Karloff, F. Korn, and D. Srivastava. Data auditor: Exploring data quality and semantics using pattern tableaux. *PVLDB*, 3(2):1641–1644, 2010.

[10] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[11] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009.

[12] S. Kramer and B. Pfahringer. Efficient search for strong partial determinations. In *KDD*, pages 371–374, 1996.

[13] S. Lopes, J.-M. Petit, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT*, pages 350–364, 2000.

[14] J. Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984.

[15] C. M. Wyss, C. Giannella, and E. L. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In *DaWaK*, pages 101–110, 2001.