Approximate Denial Constraints

Ester Livshits
Technion
esterliv@cs.technion.ac.il

Alireza Heidari
University of Waterloo
a5heidar@uwaterloo.ca

Ihab F. Ilyas
University of Waterloo
ilyas@uwaterloo.ca

Benny Kimelfeld
Technion
bennyk@cs.technion.ac.il

ABSTRACT

The problem of mining integrity constraints from data has been extensively studied over the past two decades for commonly used types of constraints, including the classic Functional Dependencies (FDs) and the more general Denial Constraints (DCs). In this paper, we investigate the problem of mining from data approximate DCs, that is, DCs that are "almost" satisfied. Approximation allows us to discover more accurate constraints in inconsistent databases and detect rules that are generally correct but may have a few exceptions. It also allows to avoid overfitting and obtain constraints that are more general, more natural, and less contrived. We introduce the algorithm ADCMiner for mining approximate DCs. An important feature of this algorithm is that it does not assume any specific approximation function for DCs, but rather allows for arbitrary approximation functions that satisfy some natural axioms that we define in the paper. We also show how our algorithm can be combined with sampling to return highly accurate results considerably faster.

PVLDB Reference Format:
Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. Approximate Denial Constraints. *PVLDB*, 13(10): 1682-1695, 2020.
DOI: https://doi.org/10.14778/3401960.3401966

1. INTRODUCTION

Integrity constraints are used for stating semantic conditions that the data in the database must comply with. Enforcing the constraints helps to make the database a more accurate model of the real world. Integrity constraints may be obtained by domain experts; however, this is often an expensive task that requires expertise not only in the domain but also in the constraint language. In the past two decades, extensive effort has been invested in exploring the challenge of automatically discovering constraints from the data itself, for different types of constraints, including the classic Functional Dependencies (FDs) [15, 21, 23, 26, 31, 35, 36, 42], the more general Conditional FDs (CFDs) [9, 13, 39], and the more general Denial Constraints (DCs) [4, 11, 37, 38].

In practice, databases nowadays are often inconsistent and violate the integrity constraints that are supposed to hold. In most large enterprises, information is obtained from imprecise and sometimes contradicting sources (e.g., social networks, news feeds, and user behavior data) via imprecise procedures (e.g., natural-language processing and image processing). In such cases, mining constraints that are satisfied by the entire database will be inadequate, as they rely on the assumption that all data values are correct. Hence, in this work, we consider the problem of mining *approximate constraints*, that is, constraints that are "almost" satisfied. Approximate constraints are useful even for accurate datasets, since they avoid overfitting to the current observations, and allow us to detect more general and less contrived rules, as well as rules that are generally correct but may have a few exceptions (which is useful, for example, for the task of detecting outliers).

EXAMPLE 1.1. Consider the database of Table 1 storing information about the yearly income and tax payments of people from different states in the US. We assume that as a general rule, for a given state, a higher yearly income implies higher tax payments. However, the database does not satisfy this constraint (e.g., the tuples t_6 and t_7 jointly violate the constraint, and the same holds for the tuples t_{14} and t_{15}). If we require constraints that are fully satisfied by the entire database, then these violations require us to weaken the rule using exceptions such as "the constraint holds only for two people who have the same name" or "the constraint holds only if none of the people is called Julia and none of them lives in Illinois," which results in very specific and complicated rules. However, we will be able to find the correct constraint if we allow for exceptions, and consider approximate constraints. □

Most of the work to date on approximate constraint discovery has focused on approximate FDs [12, 23, 25] or CFDs [9, 13, 39]. Chu et al. [11] and later Pena et al. [37,38] considered approximate DCs. As the expressive power of (C)FDs is rather restricted, in this work, we consider the problem of mining approximate DCs (ADCs for short) from data. This problem has not received much attention and the currently existing algorithms are AFASTDC [11] and its improved versions BFASTDC [37] and DCFinder [38], that we will discuss in more details in the next section.

A common shortcoming of many existing approaches to approximate constraints (including ADCs) is that the algorithms proposed for this task are often an after-thought of detecting valid exact constraints, and are usually obtained by relaxing some of the parameters of the original algorithm. Hence, existing algorithms miss opportunities to use techniques that are designed specifically for mining approximate constraints. These existing algorithms are often inefficient, since they need to examine "all" combinations of records necessary to validate the discovered DCs. Another drawback of existing algorithms is the fact that the approximation function is hard-wired into the algorithm. However, there are many

Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401966

Table 1: Running example.

	Name	State	Zip	Income	Tax
t_1	Alice	NY	11803	28K	2.4K
t_2	Mark	NY	10102	42K	4.7K
t_3	Bob	NY	13914	93K	11.8K
t_4	Mary	NY	10437	58K	6.7K
t_5	Alice	NY	10437	26K	2.1K
t_6	Julia	WA	98112	27K	1.4K
t_7	Jimmy	WA	98112	24K	1.6K
t_8	Sam	WA	98112	49K	6.8K
t_9	Jeff	WA	98112	56K	7.8K
t_{10}	Gary	WA	98112	50K	7.2K
t_{11}	Ron	WA	98112	58K	8K
t_{12}	Jennifer	WA	98112	61K	8.5K
t_{13}	Adam	WA	98112	20K	1K
t_{14}	Tim	IL	62078	39K	5K
t_{15}	Sarah	IL	98112	54K	5K

possible definitions of approximate constraints, and different works indeed consider different definitions that produce very different results. The most common definition of approximate (C)FDs, for example, is based on the minimal number of tuples that should be removed for the (C)FD to hold [9, 12, 23, 25], while the definition used for approximate DCs is based on the number of tuple pairs violating the DC [11, 37, 38]. It is not clear whether one of the definitions is the "best" one, and it may be the case that different definitions produce better results in different cases.

EXAMPLE 1.2. Consider again the database of Table 1 and the DC φ_1 of Example 1.1: $\forall t, t' \neg(t[\mathsf{State}] = t'[\mathsf{State}] \wedge t[\mathsf{Income}] > t'[\mathsf{Income}] \wedge t[\mathsf{Tax}] \leq t'[\mathsf{Tax}])$. Two out of 210 tuple pairs (i.e., 0.95%) violate φ_1. (Note that $\langle t, t' \rangle$ and $\langle t', t \rangle$ are considered separately.) The minimal number of tuples to remove from the database for φ_1 to hold is two (i.e., one of t_6, t_7 and one of t_{14}, t_{15}); that is, 13.3%. Hence, if we allow, for example, an exception rate of 5%, then φ_1 is an ADC according to the first definition, but not according to the second one.

Now, consider the DC φ_2 stating that the same zip code cannot appear in two different states: $\forall t, t' \neg(t[\mathsf{Zip}] = t'[\mathsf{Zip}] \wedge t[\mathsf{State}] \neq t'[\mathsf{State}])$. Sixteen out of 210 tuple pairs (i.e., 7.62%) violate φ_2 (every pair that includes t_{15} and one of $t_6, \ldots, t_{13}$). The only tuple to remove from the database for φ_2 to be satisfied is t_{15}; thus, it is possible to remove at most 6.67% of the tuples. In this case, if the allowed exception rate is 7%, then φ_2 is an ADC according to the second definition, but not according to the first one. (The difference in the exception rate for these two definitions is very small here, but can be significant in larger databases.) □

The main objective of this work is to gain a deeper understanding of ADCs and introduce a general framework for mining ADCs that takes the semantics (i.e., the approximation function) as input. We introduce the algorithm ADCMiner for mining ADCs from data. The algorithm consists of four main components – a predicate space generator, an evidence set constructor, an enumeration algorithm and a sampler. In summary, our main contributions in this paper are as follows:

- We formally define the problem of ADC mining (Section 4) including the concept of a valid approximation function for ADCs. To the best of our knowledge, we are the first to consider approximate constraint discovery that is not tied to a specific approximation function, but rather to a general family of approximation functions, that captures, but is not limited to, commonly used approximation functions.
- We introduce an algorithm for enumerating ADCs that allows for a general approximation function that is given as an oracle (Section 6). Our algorithm is a general algorithm for enumerating *minimal approximate hitting sets* that can even be used outside the scope of constraint discovery.
- For efficiency, we propose a *sampling* scheme (Section 7), and we address two fundamental problems: *(1)* how to estimate the number of violations of φ in D from a sample; and *(2)* how to use this estimate to deduce the right threshold (or approximation function) to be used when enumerating the ADCs from the sample. While useless for mining exact DCs, sampling allows us to efficiently return highly accurate results (w.r.t. the approximation metric) by leveraging the nature of ADCs and avoiding the space explosion that exact miners suffer from.

We experimentally evaluate our proposal (Section 8) and show that although it subsumes formerly proposed approximation frameworks, we manage to achieve better efficiency. Our experiments also show that we can achieve high precision and recall from a relatively small sample, while reducing the time by as much as 90%.

2. RELATED WORK

We now discuss the relationship between our work and past work on mining DCs from data. Chu et al. [11] have introduced the first algorithms for mining DCs and ADCs from data (FASTDC and AFASTDC, respectively). Their definition of an ADC is based on the fraction of pairs of tuples violating the DC. The algorithm AFASTDC is obtained from FASTDC by modifying the base case of the algorithm; that is, they return a constraint if the fraction of tuple pairs violating it is smaller than some predefined threshold ϵ, rather than when it is zero. Their solution consists of two main parts. First, they generate a certain data structure, namely the evidence set, that we will formally define later on, and then they use the evidence set to generate all the (A)DCs. The first part has a high computational cost, as it requires going over all tuple pairs in the database. Particularly, this algorithm may run for days on a database that consists of one million tuples [11].

Pena et al. [37, 38] significantly improved the running times of this part using bit-level operations, and Position List Indexes that minimize the number of required tuple comparisons. Their focus was on improving the efficiency of the evidence set construction, and they did not modify the second part of the solution (that generates the ADCs) and adopted the definition of ADCs used by Chu et al. [11]. Our work is complementary to that of Pena et al. [37,38] as we focus on other aspects of ADC discovery. In particular, we do not propose a new method to construct the evidence set, but rather use the algorithm of Pena et al. [38] for this purpose.

Another related work is that of Bleifuß et al. [4], who introduced Hydra—an algorithm that significantly improves the running times of DC discovery by incorporating sampling to invalidate candidates. However, their algorithm only works for valid exact DCs and, as stated by the authors, it is not clear whether and how their approach can be generalized to ADCs.

3. PRELIMINARIES

We first present some basic terminology and notation that we use throughout the paper.

By $R(A_1, \ldots, A_k)$ we denote a *relation schema* R with the attributes $A_1, \ldots, A_k$. A *database* D over $R(A_1, \ldots, A_k)$ is a finite set of *tuples* $(c_1, \ldots, c_k)$ where each c_i is a constant. We denote by $t[A_i]$ the value of tuple t in attribute A_i.

A *denial constraint* (DC for short) is an expression of the form $\forall x \neg(\omega(x) \wedge \psi(x))$, where x is a sequence of variables, $\omega(x)$ is a

Table 2: Notation table.

Notation	Meaning
S_φ	The set of predicates in the DC φ
$\mathcal{P}_R$	The predicate space over the relation schema R
$\mathbf{Sat}(t,t')$	The set of predicates satisfied by $\langle t,t'\rangle$
$\mathbf{Evi}(D)$	The evidence set of the database D

conjunction of atomic formulas and $\psi(x)$ is a conjunction of comparisons between two variables in x. Following previous work on the problem of mining DCs [4, 11, 37, 38], we limit ourselves to DCs where $\omega(x)$ is a conjunction of precisely two atomic formulas and the comparison operators are $\mathbb{B} = \{=, \neq, >, <, \geq, \leq\}$.

Let R be a relation schema and let D be a database over R. The *predicate space* $\mathcal{P}_R$ from which DCs can be formed consists of predicates of the form $t[A]\,\rho\,t'[B]$, where A and B are attributes of R, and ρ is a comparison operator from $\mathbb{B}$. Throughout the paper, we will use the following notation for DCs: $\forall t, t' \neg(P_1, \ldots, P_m)$, where each P_i is a predicate from $\mathcal{P}_R$. The *complement* of a predicate $t[A]\,\rho\,t'[B]$ is the predicate $\widehat{P} = t[A]\,\widehat{\rho}\,t'[B]$, where $\widehat{\rho}$ is the complement operator of ρ (e.g., the complement operator of $>$ is $\leq$). For a set $S = \{P_1, \ldots, P_m\}$ of predicates, we denote by $\widehat{S}$ the set $\{\widehat{P_1}, \ldots, \widehat{P_m}\}$ of complement predicates.

For a pair $\langle t, t'\rangle$ of tuples in a database D over R, we denote by $\mathbf{Sat}(t,t')$ the set of all predicates of $\mathcal{P}_R$ satisfied by $\langle t, t'\rangle$. We denote by $\mathbf{Evi}(D)$ the multiset that contains the set $\mathbf{Sat}(t,t')$ for every pair $\langle t, t'\rangle$ of tuples from D. The multiplicity of each element S in $\mathbf{Evi}(D)$ is the number of tuples pairs $\langle t, t'\rangle$ such that $\mathbf{Sat}(t,t') = S$. In practice, we store every element of $\mathbf{Evi}(D)$ once, along with its number of occurrences. We refer to $\mathbf{Evi}(D)$ as the *evidence set* of D [11].

We identify a DC φ with the set S_φ of its predicates. A DC states that its predicates cannot be satisfied all at the same time. That is, a DC φ is satisfied by a tuple pair $\langle t, t'\rangle$ if at least one of the predicates $P \in S_\varphi$ does not hold for $\langle t, t'\rangle$, or, equivalently, $\widehat{P} \in \mathbf{Sat}(t,t')$. A DC φ is *satisfied* by a database D (denoted by $D \models \varphi$) if it is satisfied by all pairs of tuples, and *violated* otherwise. If a DC φ is satisfied by a database D, we say that it is a *valid* DC w.r.t. D.

EXAMPLE 3.1. Table 3 contains a subset of the predicate space $\mathcal{P}_R$ over the relation schema of our running example. We use the operations in $\{<, \leq, >, \geq\}$ only for numeric attributes, and we only allow comparisons among attributes of the same type (i.e., two numeric or string attributes). For example, the predicate $t[\mathsf{Name}] = t[\mathsf{Income}]$ will not appear in $\mathcal{P}_R$. Among the predicates of Table 3, the predicate set $\mathbf{Sat}(t_2, t_5)$ of the tuples t_2 and t_5 of our running example will contain the following predicates: $t[\mathsf{Name}] \neq t'[\mathsf{Name}]$, $t[\mathsf{Income}] > t'[\mathsf{Income}]$, $t[\mathsf{Income}] \geq t'[\mathsf{Income}]$, and $t[\mathsf{Income}] > t'[\mathsf{Tax}]$. The set $\mathbf{Sat}(t_5, t_2)$ will also contain the first two predicates, but it will not contain the other two predicates; instead, $t[\mathsf{Income}] < t'[\mathsf{Income}]$ and $t[\mathsf{Income}] \leq t'[\mathsf{Income}]$ will appear in the set. □

In principle, our solution could be extended to more general DCs. For example, we could relax the limitation on the number of atomic formulas, which will affect mainly the size of $\mathbf{Evi}(D)$ (i.e., if we allow for k atomic formulas, then $\mathbf{Evi}(D)$ will contain a set $\mathbf{Sat}(t_1, \ldots, t_k)$ for each sequence $t_1, \ldots, t_k$ of tuples in D, and each such set will consist of more predicates, as $t_1[A] = t_2[A]$ is different than $t_2[A] = t_3[A]$). We could also consider other types of predicates, such as $t[A]\,\rho\,(k \times t'[B])$, which will increase the size of the predicate space. However, such extensions will have a significant impact on the running times, and the trade-off between more general constraints and lower running times has to be taken into account. When we focus on the DCs considered in this paper, we are already able to discover many constraints that cannot be discovered using FD discovery methods. In our experiments, about 70% of the discovered constraints cannot be expressed as FDs.

Table 3: A sample of the predicate space of our example.

$t[\mathsf{Name}] = t'[\mathsf{Name}]$	$t[\mathsf{Name}] \neq t'[\mathsf{Name}]$
$t[\mathsf{Income}] = t'[\mathsf{Income}]$	$t[\mathsf{Income}] \neq t'[\mathsf{Income}]$
$t[\mathsf{Income}] > t'[\mathsf{Income}]$	$t[\mathsf{Income}] \geq t'[\mathsf{Income}]$
$t[\mathsf{Income}] < t'[\mathsf{Income}]$	$t[\mathsf{Income}] \leq t'[\mathsf{Income}]$
$t[\mathsf{Income}] > t'[\mathsf{Tax}]$	$t[\mathsf{Income}] \geq t'[\mathsf{Tax}]$
$t[\mathsf{Income}] < t'[\mathsf{Tax}]$	$t[\mathsf{Income}] \leq t'[\mathsf{Tax}]$

4. PROBLEM AND SOLUTION OVERVIEW

In this section, we formally define the problem that we study in the paper and give an overview of our solution.

4.1 Problem Definition

We start by defining a *valid approximation function*. Let f be a function that maps a given database D and a set of predicates S_φ (representing a DC φ) into a number in $[0, 1]$. Intuitively, the higher the number is, the more satisfied the DC is by the database. In particular, $f(D, S_\varphi) = 1$ means that φ is a valid DC w.r.t. D. We now define two properties of such a function f, namely, *Monotonicity* and *Indifference to Redundancy*.

DEFINITION 4.1. [Monotonicity] A function f is monotonic if it holds that $f(D, S_\varphi) \leq f(D, S_{\varphi'})$ whenever $S_\varphi \subset S_{\varphi'}$. □

Intuitively, monotonicity ensures that the more predicates a DC contains, the higher its score is. Monotonicity allows us to consider only minimal ADCs (i.e., ADCs that do not strictly contain any ADC), as it assures that whenever φ is an ADC, every φ' such that $S_\varphi \subset S_{\varphi'}$ is also an ADC. Hence, when returning only minimal ADCs φ, we also implicitly provide the user with information on any φ' that can be obtained from φ by adding more predicates. On the other hand, for non-monotonic functions it may be the case that for φ and φ' such that $S_\varphi \subset S'_\varphi$, the DC φ is an ADC, while φ' is not. Thus, returning only φ will result is the loss of valuable information, that is, the fact that φ' is not an ADC. Moreover, monotonocity can be used by mining algorithms for early pruning of branches in the search tree, which improves efficiency, as there is no need to consider supersets of sets corresponding to ADCs.

DEFINITION 4.2. [Indifference to Redundancy] A function f is indifferent to redundancy if we have that $f(D, S_\varphi) = f(D, S'_\varphi)$ whenever $S_\varphi \subset S_{\varphi'}$ and $\{\langle t, t'\rangle \mid t, t' \in D, \{t, t'\} \models \varphi\} = \{\langle t, t'\rangle \mid t, t' \in D, \{t, t'\} \models \varphi'\}$. □

A function f is indifferent to redundancy if adding more predicates to a DC φ without affecting the coverage, does not affect the score; that is, if two DCs φ and φ' such that $S_\varphi \subset S_{\varphi'}$ are satisfied by the exact same tuple pairs, then f gives them the same score. While our algorithm for enumerating minimal ADCs could work for functions that do not satisfy indifference to redundancy, having this property allows us to significantly increase the algorithm efficiency by pruning the search tree early, as we explain in Section 6.

We now define valid approximation functions.

DEFINITION 4.3. [Valid Approximation Function] A function f is a valid approximation function if it satisfies monotonicity and indifference to redundancy. □

In the next section, we will show that this definition is quite general and captures commonly used approximation functions. Next, we give the formal definition of a minimal ADC.

DEFINITION 4.4. [Approximate Denial Constraint] Let D be a database, let f be a valid approximation function, and let $\epsilon \geq 0$. Then, a DC φ is a minimal ADC if:

1. $f(D, S_\varphi) \geq 1 - \epsilon$, *and*
2. no DC φ' s.t. $S_{\varphi'} \subset S_\varphi$ satisfies $f(D, S_{\varphi'}) \geq 1 - \epsilon$. □

The intuition behind using valid approximation functions (i.e., combining the two properties) when considering ADCs is illustrated in the following example.

EXAMPLE 4.5. Consider the following DCs:

$$\varphi = \forall t, t' \neg(t[A] < t'[A] \wedge t[A] \leq t'[A])$$
$$\varphi' = \forall t, t' \neg(t[A] < t'[A])$$

Both DCs are satisfied by the exact same tuple pairs in a given database, since whenever a tuple pair satisfies $t[A] < t'[A]$ it also satisfies $t[A] \leq t'[A]$. Intuitively, the DC φ' is minimal, while φ is not minimal, as there is no benefit in adding the predicate $t[A] \leq t'[A]$ to the DC. For a monotonic function f, it will hold that $f(D, S_{\varphi'}) \leq f(D, S_\varphi)$; however, it may be the case that $f(D, S_\varphi) \geq 1 - \epsilon$, while $f(D, S'_\varphi) < 1 - \epsilon$ (hence, φ is an ADC while φ' is not). The existence of the second property (i.e., indifference to redundancy) resolves this problem since, as aforementioned, the same pairs of tuples satisfy both DCs, which implies that $f(D, S_\varphi) = f(D, S_{\varphi'})$. Hence, we will either return φ' (if $f(D, S_{\varphi'}) \geq 1 - \epsilon$) or none of the DCs. □

Finally, we define the problem that we study in this paper.

PROBLEM 4.6 (ADC MINING PROBLEM). *Given a valid approximation function f, a threshold $\epsilon \geq 0$, and a database D, generate all the nontrivial minimal ADCs for D w.r.t. f and ϵ.*

Since generating all ADCs from the entire database may be prohibitively time consuming for large databases, we also consider the problem of mining ADCs from a sample.

4.2 ADCMiner

Our algorithm, ADCMiner is depicted in Figure 1. The input consists of a database D over a relation schema R, a valid approximation function f, and an approximation threshold $\epsilon \geq 0$. The following are the four main components of the algorithm.

1. A *predicate space generator*, that builds the predicate space $\mathcal{P}_R$ for the given relation schema R. We use the algorithm of Chu et al. [11] for this task. The predicates in $\mathcal{P}_R$ may compare the same attribute in two different tuples (i.e., $t[A]\,\rho\,t'[A]$), two different attributes in the same tuple (i.e., $t[A]\,\rho\,t[B]$), or two different attributes in two tuples (i.e., $t[A]\,\rho\,t'[B]$). We allow comparing two attributes only if they have at least 30% common values as in [11, 38].
2. A *sampler*, which draws a random sample J of tuples from D. We provide a theoretical analysis of mining ADCs from a sample in Section 7 and experimentally evaluate the accuracy of the results obtained from a sample in Section 8.
3. An *evidence set generator*, which builds the evidence set from the sample J. In this paper, we use an existing algorithm for constructing the evidence set [38].
4. An *enumeration algorithm*, which takes as input the predicate space $\mathcal{P}_R$, the evidence set $\mathbf{Evi}(J)$, the approximation function f, and the approximation threshold ϵ and enumerates all the minimal ADCs of J w.r.t. f and ϵ. We discuss this algorithm in Section 6.

Algorithm ADCMiner(R, D, f, ϵ)

1: $\mathcal{P}_R$ = GeneratePSpace(R)
2: J = Sample(D)
3: $\mathbf{Evi}(J)$ = ConstructEvidence($\mathcal{P}_R, J$)
4: ADCEnum($\emptyset, \emptyset, \mathbf{Evi}(J), \mathcal{P}_R, \mathbf{canHit}, f, \epsilon$)

Figure 1: An algorithm for discovering ADCs.

Note that, in principle, it is possible to compare attributes with less than 30% common values; however, relaxing this requirement may also significantly increase the number of unuseful predicates, such as $t[Age] \neq t'[Zip]$. The experiments conducted by Chu et al. [11] have shown that requiring at least 30% common values allows us to identify many of the comparable attributes, while avoiding a significant increase in the number of meaningless predicates.

Regarding the second component, ADCs allow exceptions by definition, and can be seen as DCs obtained from a sample, where the sample consists of the subset of tuples that jointly satisfy the DC. Hence, we are able to obtain good results from a sample, instead of using the whole database. Our experiments show that using a sample of $30\% - 40\%$ of the tuples, we consistently obtain results with a high F_1 score (compared to mining the whole database), while reducing the running time by as much as 90%.

5. APPROXIMATION FUNCTIONS

In this section, we discuss three specific valid approximation functions. Kivinen et al. [24] introduced three definitions of approximate FDs, based on three different measures, which can be easily generalized to DCs. We start by discussing each one of these measures and the corresponding approximation functions.

Let D be a database and let φ be a DC. The first measure proposed by Kivinen et al. [24] (denoted by g_1) is based on the proportion of tuple pairs violating the constraint. Formally, we define the following approximation function based on this measure:

$$f_1(D, S_\varphi) = \left|\{\langle t, t'\rangle \mid t, t' \in D, \{t, t'\} \models \varphi\}\right| / |D|(|D| - 1)$$

Note that as opposed to the definition of g_1, we count the pairs satisfying the DC rather than those violating it. Also, the denominator in our case is $|D|(|D| - 1)$, while it is $|D|^2$ in g_1. Intuitively, $f_1(D, S_\varphi)$ is the probability to select a satisfying tuple pair among all pairs, assuming a uniform distribution of the violations. This measure has been used in [11] and [37, 38] to define ADCs.

The second measure in [24], denoted by g_2, is based on the proportion of "problematic" tuples, that is, tuples that are involved in a violation of the constraint. Here, we define the following approximation function:

$$f_2(D, S_\varphi) = \left|\{t \mid t \in D, \nexists t' \in D, \{t, t'\} \not\models \varphi\}\right| / |D|$$

We have that $g_2(D, \varphi) = 1 - f_2(D, S_\varphi)$. If we consider an inconsistent database D, it may be the case that only one tuple contains errors, but every pair of tuples that includes this tuple violates the DC φ. In this case, it holds that $f_2(D, S_\varphi) = 0$, as all the tuples appear in one violating pair. However, if we just remove this one tuple, the DC will hold. Thus, this measure may be too sensitive, and the last measure (g_3) proposed by Kivinen et al. [24], based on the minimal number of tuples to remove from the database for the constraint to hold, seems to be a better fit in this case. Hence, we introduce the following approximation function.

$$f_3(D, S_\varphi) = \max_{D'}\{|D'| \mid D' \subseteq D, D' \models \varphi\}/|D|$$

That is, the value $f_3(D, S_\varphi)$ (or, equivalently, $1 - g_3(D, \varphi)$) is based on the size of a *cardinality repair* [30] of D (i.e., the largest subinstance of D among all those satisfying the DC). The subinstance D' considered in this function can also be seen as a Most Probable Database [19] in the framework of tuple independent probabilistic databases. The connection between the problems is studied by Livshits et al. [29]. This approximation function has been used in past work on approximate (C)FDs [9, 12, 23, 25].

We now prove that the functions f_1, f_2 and f_3 satisfy both monotonicity and indifference to redundancy.

PROPOSITION 5.1. *The functions f_1, f_2, and f_3 are monotonic.*

PROOF. The denominator does not depend on φ in any of the three functions; hence, monotonicity only depends on the numerator. Clearly, the function f_1 is monotonic, as adding more predicates to φ can only increase the number of tuple pairs that satisfy the DC. For that same reason, the number of tuples $t \in D$ that are not involved in any violation of φ can only increase, and the function f_2 is also monotonic. As for the function f_3, let D' be a subinstance of D such that $D' \models \varphi$ and there is no other subinstance D'' of D that also satisfies this property such that $|D''| > |D'|$. Clearly, for each φ' such that $S_\varphi \subseteq S_{\varphi'}$ it holds that $D' \models \varphi'$ as well. Thus, D' also satisfies the condition in the numerator of f_3 for φ' (although D' is not necessarily maximal in this case), and the value $f_3(D, S_{\varphi'})$ cannot be lower than $f_3(D, S_\varphi)$. □

PROPOSITION 5.2. *The functions f_1, f_2, and f_3 are indifferent to redundancy.*

PROOF. It is rather straightforward that f_1 and f_2 satisfy this property. If the same tuple pairs satisfy both φ and φ', then the function f_1 that counts such pairs assigns the same value to both DCs. This also implies that the same tuples are involved in violations of both DCs, and $f_2(D, S_\varphi) = f_2(D, S_{\varphi'})$. To prove indifference to redundancy for f_3, we will show that every subinstance D' of D satisfies φ if and only it satisfies φ'. This holds since every subinstance D' satisfying one of these DCs does not contain any pair of tuples from D that jointly violate the DC, and since the exact same pairs of tuples from D violate both DCs, it means that it does not contain any tuple pair violating the other DC. □

We also prove the following result regarding the relationships between the functions f_2, f_3 and the function f_1. As will be seen in the next section, throughout the algorithm we always keep track of the sets in $\mathbf{Evi}(D)$ that have an empty intersection with $\widehat{S_\varphi}$; hence, we can compute the function f_1 faster than computing f_2 or f_3. The next proposition allows us to reduce the number of times we are required to compute f_2 or f_3 using the function f_1.

PROPOSITION 5.3. *Let D be a database, φ a DC, and $\epsilon \geq 0$. For $i \in \{2, 3\}$, if $f_i(D, S_\varphi) \geq 1 - \epsilon$ then $f_1(D, S_\varphi) \geq 1 - 2\epsilon$.*

PROOF. The evidence set $\mathbf{Evi}(D)$ contains $2(|D| - 1)$ sets for every tuple $t \in D$ (two sets, $\mathbf{Sat}(t, t')$ and $\mathbf{Sat}(t', t)$, for every tuple $t' \in D$). If $f_2(D, S_\varphi) \geq 1 - \epsilon$, then at most $\epsilon|D|$ tuples appear in a violating pair. Thus, the number of violating pairs is at most $2\epsilon|D|(|D| - 1)$, which is exactly 2ϵ of the tuple pairs. We conclude that $f_1(D, S_\varphi) \geq 1 - 2\epsilon$. As for the function f_3, when we remove a tuple from D, we remove $2(|D| - 1)$ sets from $\mathbf{Evi}(D)$. If $f_3(D, S_\varphi) \geq 1 - \epsilon$, then there is a subinstance D' of D that is obtained by removing at most $\epsilon|D|$ tuples from D such that $D' \models \varphi$. This observation implies that $\mathbf{Evi}(D')$ contains every set in $\mathbf{Evi}(D)$ except for at most $2\epsilon|D|(|D| - 1)$ sets. Since D' satisfies φ, at most $2\epsilon|D|(|D| - 1)$ pairs violate φ, which is at most 2ϵ of the tuple pairs,and again we have that $f_1(D, S_\varphi) \geq 1 - 2\epsilon$. □

Algorithm GreedyF3$(D, S_\varphi, \mathbf{vios}, \epsilon)$

1: $(T, v) =$ SortTuples$(D, S_\varphi, \mathbf{vios})$
2: $u = |S \in \mathbf{Evi}(D) \mid S \cap S_\varphi = \emptyset|$
3: $c = 0$, $R = \emptyset$
4: **while** $c < u$ **do**
5: let t be the first tuple in T
6: $c = c + v(t)$
7: remove t from T and add it to R
8: **return** $(|R|/|D| \leq \epsilon)$

Subroutine SortTuples$(D, S_\varphi, \mathbf{vios})$

1: $v(t) = 0$ for all $t \in D$
2: **for all** $S \in \mathbf{Evi}(D)$ such that $S \cap S_\varphi = \emptyset$ **do**
3: **for all** $t \in \mathbf{vios}[S]$ **do**
4: $v(t) = v(t) + \mathbf{vios}[S][t]$
5: **return** (tuples of D in descending order of $v(t), v(t)$)

Figure 2: A greedy algorithm replacing f_3.

Finally, we discuss the computational complexity of the three functions. The functions f_1 and f_2 can be computed in polynomial time for both FDs and DCs. However, while the function f_3 can be computed in polynomial time for FDs [29], Livshits et al. [28] have shown that this problem is NP-hard even when considering simple DCs over a single relation symbol (e.g., the DC $\forall t, t' \neg(t[B] = t'[A] \wedge t[A] \neq t'[B])$). Hence, we cannot efficiently compute f_3 under conventional complexity assumptions. However, there is a simple reduction from the problem of computing $1 - f_3(D, S_\varphi)$ to the minimum-vertex-cover problem (where the goal is to find a minimal set of vertices that intersects all edges), based on the concept of a *conflict graph*, in which vertices represent tuples and edges represent violations. Since vertex cover is 2-approximable in polynomial time [3], this is also the case for our problem. Thus, to generate ADCs w.r.t. f_3 we could use the 2-approximation algorithm with the threshold 2ϵ. Note that we will return all ADCs, but we may also return some DCs for which it holds that $f_3(D, S_\varphi) \geq 1 - 2\epsilon$ but $f_3(D, S_\varphi) < 1 - \epsilon$.

In practice, the 2-approximation algorithms for minimum vertex cover assume an explicit representation of the graph. In our case, this requires storing, for every set S in $\mathbf{Evi}(D)$, all pairs $\langle t, t' \rangle$ of tuples such that $\mathbf{Sat}(t, t') = S$. As the number of tuple pairs is quadratic in the size of the database, storing this information with reasonable memory usage is infeasible for large databases. Hence, in our experimental evaluation, we implement a greedy algorithm (depicted in Figure 2) instead. The algorithm is inspired by the greedy $O(\log n)$-approximation algorithm for minimum vertex cover, that, in each iteration, selects a vertex that is adjacent to the maximal number of uncovered edges, and then marks each one of these edges as covered. While we do not provide any theoretical guarantees on the result of this algorithm, our experimental evaluation shows that using this algorithm we often obtain more accurate results than the ones obtained using the function f_2.

In the algorithm, we sort the tuples in descending order according to the number of violations they participate in. For that, we use the data structure $\mathbf{vios}$ that stores, for every set $S \in \mathbf{Evi}(D)$ and tuple $t \in D$, the number of tuple pairs $\langle t_1, t_2 \rangle$ such that $\mathbf{Sat}(t_1, t_2) = S$ and either $t_1 = t$ or $t_2 = t$. Then, we select these tuples, one by one, while recording the change to the number

of violations covered by the selected tuples. That is, with every tuple that we select, we add the number of violations it participates in to the number of covered violations c. We stop when the number of covered violations c is at least the number of total violations u. Note that the number of covered violations can be higher than the number of total violations, as if two tuples jointly violate the DC and are both added to the result, we count this violation twice. Finally, we return the DC if the ratio between the number of tuples in the result and the total number of tuples is lower than the threshold.

The most time consuming part of the algorithm is the subroutine SortTuples; hence, the time complexity is $O(|D| \cdot n)$ where n is the number of *distinct* sets in $\mathbf{Evi}(D)$ (recall that $\mathbf{Evi}(D)$ is a multiset), and the space complexity, which depends on the size of $\mathbf{vios}$, is the same. In all of our experiments, the number of distinct sets in $\mathbf{Evi}(D)$ is orders of magnitude smaller than the number of tuple pairs; hence, storing this data structure requires significantly less space than storing data for every pair of tuples.

6. ENUMERATION ALGORITHM

In this section, we introduce an algorithm for enumerating minimal ADCs. Following Chu et al. [11], we reduce our problem to that of enumerating *minimal approximate hitting sets*. The hitting set problem is the following: given a finite set K and a family M of subsets of K, find all subsets of K that intersect every one of the subsets in M. A subset F is a *minimal* hitting set if no proper subset of F is a hitting set. As mentioned in the preliminaries, a pair $\langle t, t' \rangle$ of tuples satisfies a DC φ if $\widehat{P} \in \mathbf{Sat}(t, t')$ for some $P \in S_\varphi$. Hence, it is rather straightforward that φ is a valid DC if $\widehat{S_\varphi}$ is a hitting set of $\mathbf{Evi}(D)$. Note that the other direction does not necessarily hold, as a hitting set may not correspond to a non-trivial DC. For example, the set $\{t[A] = t'[A], t[A] \neq t'[A]\}$ is clearly a hitting set of $\mathbf{Evi}(D)$, but the corresponding DC is trivial. Hence, the reduction is essentially to the hitting set problem with restrictions rather than the general hitting-set problem.

Although the complexity of enumerating minimal hitting sets or, equivalently, hypergraph transversals is still an open problem (after decades of research), many algorithms have been proposed for this task (see [17] for a survey). Yet, to the best of our knowledge, the problem of enumerating minimal *approximate* hitting sets has not received much attention. Here, we refer to a set $F \subseteq K$ that satisfies $f(M, F) \geq 1 - \epsilon$ for a given valid approximation function f and a threshold ϵ as an approximate hitting set. Algorithmic literature typically refers to one of two problems as computing approximate hitting sets: *(1)* enumerating hitting sets, but not necessarily all of them (and not necessarily minimal) [1, 7, 34], and *(2)* computing an approximate hitting set of minimum cardinality [6, 8, 41]. Here, we devise an algorithm for enumerating minimal approximate hitting sets, building upon an algorithm for enumerating minimal hitting sets by Murakami and Uno [33]. In Section 8, we compare the performance of our algorithm to the discovery algorithm used in [11, 37, 38], and show that even though our algorithm is more general, we are able to significantly reduce the running time.

6.1 Enumerating Minimal Hitting Sets

We now introduce the algorithm of Murakami and Uno [33] for enumeraing minimal hitting sets. In the next subsection, we will explain how we adapt the algorithm to the approximation problem.

The algorithm is depicted in Figure 3. The input consists of three data structures, namely $\mathbf{uncov}$, $\mathbf{cand}$ and $\mathbf{crit}$, that are initialized from a given set K of elements and a set M of subsets of K. The algorithm is a recursive algorithm that builds the hitting sets incrementally. It starts with an empty set S, and adds elements to S

Algorithm MMCS($S, \mathbf{crit}, \mathbf{uncov}, \mathbf{cand}$) [33]

```
1: if uncov = ∅ then
2:     output S
3:     return
4: choose a set F from uncov
5: C = cand ∩ F
6: cand = cand \ C
7: for all e ∈ C do
8:     UpdateCritUncov(e, S, crit, uncov)
9:     if crit[u] ≠ ∅ for each u ∈ S then
10:        MMCS(S ∪ {e}, crit, uncov, cand)
11:    cand = cand ∪ {e}
12:    recover the changes to crit and uncov done in 8
13: recover the change to cand done in 6
```

Subroutine UpdateCritUncov($e, S, \mathbf{crit}, \mathbf{uncov}$)

```
1: for all F ∈ uncov do
2:     if e ∈ F then
3:         crit[e] = crit[e] ∪ {F}
4:         uncov = uncov \ {F}
5: for all u ∈ S do
6:     for all F ∈ crit[u] do
7:         if e ∈ F then
8:             crit[u] = crit[u] \ {F}
```

Figure 3: An algorithm for enumerating minimal hitting sets.

until it has a nonempty intersection with every subset of M. The data structure $\mathbf{uncov}$ stores the subsets of M that have an empty intersection with the intermediate S. Since we start with an empty S, initially, $\mathbf{uncov}$ contains every subset of M. The data structure $\mathbf{cand}$ stores the elements of K that can be added to S in the next iterations of the algorithm. Initially, it contains every element of K. Finally, $\mathbf{crit}$ stores, for each element e in the intermediate S, all the subsets in M for which e is critical (i.e., all the subsets that contain e, but do not contain any other element of S). The importance of each one of these data structures will become clear soon.

At each iteration, the algorithm selects a subset F from $\mathbf{uncov}$. The goal is then to add at least one element of F to S, so that the two sets have a nonempty intersection. In line 5 of the algorithm, we store the intersection of F and $\mathbf{cand}$ in C. The set C thus contains all the elements of F that we are allowed to add to S. Then, every element of F is removed from $\mathbf{cand}$. Some of these elements will be added back to $\mathbf{cand}$ later on, while some are permanently removed from this list. The idea is the following. Let $\{e_1, \ldots, e_n\}$ be the set of elements in C. First, we add e_1 to S, and the other elements of C still do not belong to $\mathbf{cand}$; hence, we are able to generate minimal hitting sets that contain e_1, but do not contain any other element of C. Then, we add e_2 to S and we add e_1 to $\mathbf{cand}$ (if some condition holds, as we will explain later). Thus, we are now able to generate minimal hitting sets that contain only e_2, or contain both e_2 and e_1, but do not contain any other element of C. Then, we add e_3 to S and both e_1 and e_2 appear in the list of candidates, and so on. This allows us to avoid generating the same hitting set twice, but it also allows us to prune branches in the search tree early on, as we now explain.

Observe that a set S is a *minimal* hitting set only if *every* element of S is critical to at least one subset. Thus, after adding an element

Algorithm ADCEnum($S, \mathbf{crit}, \mathbf{uncov}, \mathbf{cand}, \mathbf{canHit}, f, \epsilon$)

```
1: if f(D,S) ≥ 1 − ε and IsMinimal(S, f, ε) then
2:     output DC from S
3:     return
4: choose a set F ∈ uncov s.t. canHit[F] = true
5: if such a set F does not exist then
6:     return
7: cand = cand \ F
8: UpdateCanCover(uncov, cand, canHit)
9: if WillCover(S, cand, f, ε) then
10:    ADCEnum(S, crit, uncov, cand, canHit)
11: recover the change to cand done in 7
12: recover the change to canHit done in 8
13: C = cand ∩ F
14: cand = cand \ C
15: for all e ∈ C do
16:    UpdateCritUncov(e, S, crit, uncov)
17:    if crit[u] ≠ ∅ for each u ∈ S then
18:        RemoveRedundantPreds(e, cand)
19:        ADCEnum(S ∪ {e}, crit, uncov, cand, canHit)
20:        cand = cand ∪ {e}
21:    recover the changes to crit and uncov done in 16
22: recover the change to cand done in 14
```

Figure 4: Enumerating minimal ADCs - main.

e of F to S, the UpdateCritUncov subroutine is called. This subroutine updates the data structures in the following way: *(a)* every subset in $\mathbf{uncov}$ that contains e is removed from $\mathbf{uncov}$, as it no longer has an empty intersection with S, *(b)* every subset that has been removed from $\mathbf{uncov}$ is added to the list of subsets for which e is critical, as it does not contain any other element of S, and *(c)* for every element u in S, and for every subset F that belongs to the list of subsets for which u is critical, F is removed from this list if it contains e (as it now contains other elements of S).

The purpose of calling UpdateCritUncov is twofold. First, it updates the data structures after adding a new element to S. Second, it is used to prune branches in the search tree. In line 9 of the algorithm, after the call to the subroutine, the algorithm checks whether for every element of S, the list of subsets for which it is critical is nonempty. Otherwise, as explained above, this branch will never result in a minimal hitting set. Hence, if the test of line 9 fails, we recover all changes to $\mathbf{crit}$ and $\mathbf{uncov}$, and move on to the next element of C in the loop of line 7. In this case, the element e is not added back to $\mathbf{cand}$ due to the observation that if an element is not critical for any subset w.r.t. S, it cannot be critical for any subset w.r.t. any S' such that $S \subseteq S'$. If, on the other hand, the test of line 9 succeeds, we add e back to $\mathbf{cand}$; thus, it could be added to S later on. Murakami and Uno [33] proved the following about the algorithm MMCS: *(a)* it returns only minimal hitting sets, *(b)* it returns every minimal hitting set, and *(c)* it returns each minimal hitting set once. They have shown that the time complexity of the algorithm is $O(\|M\|)$ per iteration, where $\|M\|$ is the sum of sizes of sets in M. Moreover, the space complexity is $O(\|M\|)$.

6.2 Enumerating Approximate Hitting Sets

One may suggest to adapt the algorithm of Figure 3 to generate minimal approximate hitting sets by modifying the base case. Instead of stopping when every subset has a nonempty intersection with S, we will stop when $f(D, S) \geq 1-\epsilon$ for the given function f and threshold ϵ. It is straightforward that this will return only minimal approximate hitting sets, but will it return all of them? The answer to this question is negative. The problem with this approach, which also applies to many other algorithms for enumerating minimal hitting sets [17], is that when we select a new subset at each iteration and try to "hit" it, we define a certain order over the subsets. For example, an easy observation is that we will never return a set that has an empty intersection with the first chosen subset, even if it has a nonempty intersection with any other subset.

Our algorithm ADCEnum for enumerating minimal ADCs is depicted in Figure 4. We modify the algorithm MMCS as follows. First, we change the base case, as aforementioned; that is, we print S only if $f(D, S) \geq 1 - \epsilon$. However, we also have to explicitly check for minimality before printing S. This is due to the fact that while a set S of elements where each $e \in S$ is critical for at least one subset of M is guaranteed to be minimal when considering hitting sets, this is not the case when considering approximate hitting sets, as our S is allowed to have an empty intersection with some subsets of M. Due to the indifference-to-redundancy property, this condition is still necessary when considering approximate hitting sets, since we can remove elements that are not critical for any subset without affecting the set of tuple pairs that have a nonempty intersection with S, and, consequently, without affecting the value of the approximation function. However, this condition is no longer sufficient. Therefore, we check whether S is minimal in the IsMinimal subroutine, depicted in Figure 5. There, we go over all sets S' of elements obtained from S by removing a single element, and for each S' we check whether $f(D, S') \geq 1-\epsilon$. Recall that the approximation functions that we consider are monotonic; hence, if for a subset S' of S it holds that $f(D, S') < 1 - \epsilon$, then we have that $f(D, S'') < 1 - \epsilon$ for any $S'' \subset S'$, and we do not need to go over the subsets of S obtained by removing more than one element.

Next, we choose a subset $F \in \mathbf{uncov}$ and make two recursive calls – one that "hits" the chosen F (i.e., adds an element of F to S) and one that does not. We start with the second one. Observe that our algorithm contains an additional data structure, namely $\mathbf{canHit}$. It is used for the additional recursive call and it contains a single value, true or false, for every subset. Initially, the value is true for all subsets. The idea is the following. Whenever we choose not to hit F, this set remains in $\mathbf{uncov}$. To avoid choosing it again in a future iteration of the algorithm (which may result in an infinite recursion), we update $\mathbf{canHit}[F] =$ false in the UpdateCanBeCovered subroutine depicted in Figure 5. However, F may not be the only subset in $\mathbf{uncov}$ that has an empty intersection with $\mathbf{cand}$ after removing all the elements of F from $\mathbf{cand}$ in line 7. Hence, in this subroutine, we mark every subset that is still in $\mathbf{uncov}$ and does not contain any element of $\mathbf{cand}$. This way, we avoid selecting these subsets in future iterations, which significantly reduces the number of unnecessary recursive calls. Finally, we make the recursive call after checking whether it can result in an approximate hitting set. We check that in the WillCover subroutine that adds all the elements of $\mathbf{cand}$ to S and checks whether the result S' satisfies $f(D, S') \geq 1 - \epsilon$. If this is not the case, then the monotonicity property ensures that this branch will never result in an approximate hitting set (since we cannot increase the value of the approximation function by adding fewer predicates), and we do not make the recursive call.

The second recursive call (where we hit the selected F) is identical to the recursive call of the original algorithm and we do not explain it again here. Note that if we did not assume indifference to redundancy, we could not prune branches based on the $\mathbf{crit}$ data structure (line 17) as done in the original algorithm, since it could be the case that adding predicates that are not critical for any subset

Subroutine IsMinimal(S, f, ϵ)

1: **for all** $e \in S$ **do**
2: **if** $f(D, S \setminus \{e\}) \geq 1 - \epsilon$ **then**
3: **return false**
4: **return true**

Subroutine UpdateCanCover($\mathbf{uncov}, \mathbf{cand}, \mathbf{canHit}$)

1: **for all** $F \in \mathbf{uncov}$ **do**
2: **for all** $e \in \mathbf{cand}$ **do**
3: **if** $e \in F$ **then**
4: continue outer loop
5: $\mathbf{canHit}[F] = \text{false}$

Subroutine WillCover($S, \mathbf{cand}, f, \epsilon$)

1: $S' = S \cup \mathbf{cand}$
2: **if** $f(D, S') \geq 1 - \epsilon$ **then**
3: **return true**
4: **return false**

Figure 5: Enumerating minimal ADCs - subroutines.

actually increases the value of the approximation function (while having no impact on the set of tuple pairs satisfying the DC).

While the algorithm of Figure 4 can be used as a general algorithm for enumerating minimal approximate hitting sets, there are two aspects that are specific to our setting. First, we do not return the hitting set S itself, but the DC obtained from S. Second, before making the recursive call of line 19, and after adding an element u to S, we remove from $\mathbf{cand}$ all the predicates that differ from u only by the operator. This way, we avoid developing branches that will result in trivial DCs, such as $\forall t, t' \neg(t[A] < t'[A] \wedge t[A] \geq t'[A])$, and avoid developing some branches that will fail the minimality condition, such as $\forall t, t' \neg(t[A] < t'[A] \wedge t[A] \leq t'[A])$ (this is again based on the assumption that the approximation function is indifferent to redundancy, and the addition of the predicate $t[A] \leq t'[A]$ cannot affect the value of the approximation function on a set that already contains the predicate $t[A] < t'[A]$).

Finally, to improve the running time of the algorithm, we do not select a random set F in line 4, but rather the set that maximizes the intersection with $\mathbf{cand}$. Murakami and Uno [33] suggested to select the set that minimizes this intersection, as it decreases the number of iterations in the loop of line 15, and, consequently, the number of recursive calls. However, in our case, it increases the number of recursive calls of line 10. Our experiments (on which we report in the extended version of the paper [27]) show that our choice decreases the running times, as the total number of recursive calls decreases compared to Murakami and Uno [33].

6.3 Proof of Correctness

The correctness of ADCEnum is stated in the following theorem. We give a proof sketch here; the full proof is in [27].

THEOREM 6.1. *Let D be a database, f a valid approximation function, and $\epsilon \geq 0$. The following hold for* ADCEnum *w.r.t. D, f and ϵ: (a) it returns only minimal ADCs, (b) it returns all the minimal ADCs, and (c) it returns every minimal ADC once.*

PROOF. *(Sketch)* The proofs of *(a)* and *(c)* are rather straightforward. We prove *(b)* by induction on the depth of the recursion. We prove that ADCEnum($S, \mathbf{crit}, \mathbf{uncov}, \mathbf{cand}, \mathbf{canHit}, f, \epsilon$) returns every minimal ADC φ that satisfies:

1. $S \subseteq S_\varphi$ and $S_\varphi \subseteq (S \cup \mathbf{cand})$,
2. S_φ has an empty intersection with all the sets $F \in \mathbf{Evi}(D)$ for which $\mathbf{canHit}[F] = \text{false}$.

Since at the beginning, $\mathbf{cand}$ contains all the predicates of $\mathcal{P}_R$ and we have that $\mathbf{canHit}[F] = \text{true}$ for each $F \in \mathbf{Evi}(D)$, we will conclude that ADCEnum($\emptyset, \emptyset, \mathbf{Evi}(D), \mathcal{P}_R, \mathbf{canHit}, f, \epsilon$) returns every ADC φ such that $\emptyset \subseteq S_\varphi$ and $S_\varphi \subseteq \mathbf{cand}$ (i.e., all ADCs).

We prove the claim by considering the two recursive calls separately. In particular, we show that we generate, in the recursive call of line 10, all the minimal ADCs φ that satisfy the conditions 1 and 2 such that S_φ has an empty intersection with the set F selected in line 4. Then, we show that we generate every φ satisfying the above conditions such that S_φ has a nonempty intersection with the chosen F, in the recursive call of line 19. In this case, we generate φ in the iteration of the for loop of line 15, where the last predicate of $S_\varphi \cap F$ is selected (and the rest of the predicates in the intersection appear in $\mathbf{cand}$). □

Finally, we discuss the complexity of ADCEnum. There are two components of the algorithm that affect the time complexity compared to the complexity of MMCS—the additional recursive call in line 10, and the computation of the function f that affects the complexity per iteration. Recall that the complexity of MMCS per iteration is $O(\|M\|)$. In our case, we have that $\|M\|$ is bounded by $|\mathcal{P}| \cdot n$, where n is the number of distinct sets in $\mathbf{Evi}(D)$. We compute the function f in the algorithm $|S| + 2$ times, and since $|S|$ is bounded by $|\mathcal{P}|$, we conclude that the time complexity per iteration is $O(|\mathcal{P}| \cdot n + |\mathcal{P}| \cdot t_f)$, where t_f is the time required to compute the function f. The space complexity is not affected compared to MMCS and remains $O(|\mathcal{P}| \cdot n)$.

7. MINING ADCS FROM A SAMPLE

The input to our algorithm is the evidence set and the complexity of building it is quadratic in the size of the database (as we have to go over all tuple pairs), which can be prohibitively expensive for large databases. In this section, we show how to use a sample from the database to produce ADCs with probabilistic guarantees, while avoiding the cost of building the evidence set for the entire database [20]. For simplicity, we limit our discussion to the function f_1 introduced in Section 5. Recall that this function is based on the number of tuple pairs violating the DC in the database.

Let J be a sample uniformly drawn from a database D and let $\epsilon \geq 0$. Let φ be a DC. We address the following problems: *(1)* how to estimate the number of violations of φ in D from J; and *(2)* how to use this estimate to decide on the right threshold (or approximation function) to use when enumerating ADCs from J.

7.1 Estimating the Number of Violations

Since we consider the function f_1 that is based on the number of violations of the DC in the database, we now show how to estimate this number from a sample J uniformly drawn from D. We represent the violations of an ADC φ as a conflict graph $G(V, E)$ [10], where V is the set of vertices corresponding to the tuples of D, and E is the set of edges corresponding to violations of the DC, where an edge (t_1, t_2) exists if the pair $\langle t_1, t_2 \rangle$ violates the DC. Note that this is a directed graph since a pair $\langle t_1, t_2 \rangle$ may violate a DC that is satisfied by $\langle t_2, t_1 \rangle$. Hence, the problem that we consider here is that of estimating the *density* of a graph from a given sample.

To the best of our knowledge, most work on the density of random graphs focus on the generation of samples with density requirements [2,5,16,22,32,40], which seems to be a harder problem.

Hence, the methods proposed there are too robust for our problem, and this is reflected in the high computational complexity of the proposed solutions. In our case, the graph that we obtain is different for every DC, and we need to estimate the density for a different graph in every iteration of the algorithm; hence, using solutions with a high computational cost is infeasible. There is also a line of work that focuses on the related problem of estimating the average degree of a graph, given the degree of some of the vertices [14, 18]; however, a basic requirement in the proposed solutions is to be able to query the actual degree of at least $O(\sqrt{|V|})$ vertices. To obtain this information, we will need to find the conflicts in which each one of the corresponding tuples is involved, which requires going over at least $O(|V| \cdot \sqrt{|V|})$ tuple pairs, and is again prohibitively expensive in our case. Hence, we use a simple method for estimating the graph density from a sample, that has no significant impact on the computational cost of our algorithm.

Let $p = \frac{|E|}{2 \cdot \binom{|V|}{2}}$ (that is, $p = 1 - f_1(D, S_\varphi)$). Let $G_J(V_J, E_J)$ be the conflict graph of J. To estimate p from J, we use the value $\hat{p} = \frac{|E_J|}{2 \cdot \binom{|V_J|}{2}}$. It can be easily shown that $E(\hat{p}) = p$, so it is an unbiased estimator of p. Note that we do not make assumptions about the structure of the conflict graph or about the dependencies between the edges.

We further derive error bounds on our estimator. Using Chebyshev's inequality we obtain the following.

$$\Pr(|\hat{p} - p| > a) \leq \frac{p}{a^2} \cdot \left[\frac{2 \cdot \binom{|V_J|}{2} + \binom{2 \cdot \binom{|V_J|}{2}}{2}}{4 \cdot \binom{|V_J|}{2}^2} - p \right] \quad (1)$$

(We provide all of the intermediate computations of our analysis in the extended version of the paper [27].) The obtained bound is loose since we did not assume anything about the structure of the conflict graph and the dependencies among the violations. Nevertheless, we show that better bounds can be obtained under the assumption that violations (or, equivalently, edges) are introduced randomly and independently.

We first introduce the rationale behind random violations as follows. Assume a random polluter who is a probability distribution over graphs on n labeled vertices, where each directed edge appears independently with probability p. Each violation (edge) independently occurs between two tuples without following any specific pattern. Under this assumption, the number of edges in a sample J produces a binomial distribution.

$$\Pr[E_J = i] = \binom{2 \cdot \binom{|V_J|}{2}}{i} \cdot p^i \cdot (1-p)^{2 \cdot \binom{|V_J|}{2} - i}$$

For simplicity, we assume that the sample size is not too small and p is not too close to 0 or 1; hence, we can approximate the binomial $B(n, p)$ under the mentioned conditions using the normal distribution $N(np, np(1-p))$, and we can define a confidence interval parameterized by a confidence level $1 - 2\alpha$, and $n = 2 \cdot \binom{|V_J|}{2}$. The confidence interval of the normal distribution is given by the following equation:

$$\Pr\left[|p - \hat{p}| \leq z_{1-2\alpha} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right] \geq 1 - 2\alpha \quad (2)$$

where $z_{1-2\alpha}$ is the z-score of the standard normal distribution for confidence level $1 - 2\alpha$.

In the next section, we elaborate on how to use this idea to decide which threshold ϵ_J should be used on the sample, assuming that the desired threshold for the database is ϵ.

7.2 Computing the Sample Threshold

We now focus on the following problem. Given a sample J, a threshold ϵ and an error bound α, find the thresholds that should be used on the sample to obtain accurate ADCs with high probability. Note that the threshold may depend on the DC itself, since different DCs are violated by different tuple pairs, and, consequently, the conflict graphs of different DCs are different. That is, if φ is an ADC on the sample J w.r.t. ϵ_J^φ, then we require that with probability at least $1 - \alpha$, it holds that φ is an ADC on the entire database w.r.t. ϵ. We use Inequality (2) for this task.

Using the symmetry of the normal distribution and some standard mathematical manipulations (given in the extended version of the paper [27]), we obtain the following.

$$\Pr\left[(1-p) \geq (1-\hat{p}) - z_{1-2\alpha} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right] \geq 1 - \alpha$$

Recall that our goal is to find an ϵ_J^φ such that if $1 - \hat{p} \geq 1 - \epsilon_J^\varphi$ then $\Pr(1 - p \geq 1 - \epsilon) > 1 - \alpha$. Thus, all we need to do now is to set:

$$(1-\hat{p}) \geq z_{1-2\alpha} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} + (1-\epsilon) \quad (3)$$

Consequently, if we define $\epsilon_J^\varphi = 1 - z_{1-2\alpha} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} + (1-\epsilon)$, and accept the DC φ if $1 - \hat{p} \geq 1 - \epsilon_J^\varphi$, then with probability at least $1 - \alpha$, this DC is an ADC on the entire database w.r.t. the threshold ϵ. We conclude that we can use Inequality (3) as a criterion for accepting or rejecting an ADC on the sample.

Observe that we can also look at Inequality (3) from a different point of view. Rather than defining a different threshold ϵ_J^φ for every DC, we can define the following approximation function:

$$f_1' = (1-\hat{p}) - z_{1-2\alpha} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

Then, Inequality (3) implies that the DC φ is an ADC on the entire database w.r.t. the threshold ϵ if it is an ADC on the sample w.r.t. the approximation function f_1' and the same ϵ. Note that as the size of the sample increases, the value n increases as well, and the difference between f_1 and f_1' becomes small, as expected.

8. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our ADC discovery algorithm.

8.1 Experimental Setup

We implemented our enumeration algorithm, including the functions f_1 and f_2 in Java 8. As explained in Section 5, the function f_3 is hard to compute for DCs; hence, we implemented the algorithm of Figure 2, and we refer to this algorithm when mentioning the function f_3. We also used the Java implementation of the algorithm AFASTDC by Chu et al. [11] and the Java implementation of the algorithm DCFinder provided by the authors of [38].

All experiments were executed on a machine with an Intel Xeon CPU E5-2603 v3 (1.60GHz, 12 cores) with 64GB of RAM running Ubuntu 14.04.3 LTS. All the experiments were repeated ten times and the average values are reported.

Following previous work on the problem of discovering DCs [4, 11, 38], we evaluate our algorithm on seven real-world datasets (**SP Stock**, **Hospital**, **Food Inspection**, **Airport**, **Adult**, **Flight**, and **NCVoter**), and one synthetic dataset (**Tax**). Table 4 depicts the number of tuples, attributes, and golden DCs (i.e., DCs obtained by human experts) for each one of the datasets.

Table 4: Datasets.

Dataset	#Tuples	#Attributes	#Golden DCs
Tax	1M	15	9
Stock	123K	7	6
Hospital	115K	19	7
Food	200K	17	10
Airport	55K	12	9
Adult	32K	15	3
Flight	582K	20	13
Voter	950K	25	12

8.2 Running Time

We evaluate the running times of our solution on the aforementioned datasets and compare them to the running times of the algorithms AFASTDC [11] and DCFinder [37]. As we do not propose a new technique for constructing the evidence set, we first compare the running times of the enumeration algorithms; that is, we compare ADCEnum with the algorithm SearchMinimalCovers used in [11, 37, 38], that we denote here by SearchMC.

In the experiments, we use the approximation function f_1 (which is the function SearchMC is designed for) with the threshold $\epsilon = 0.1$. Figure 6 depicts the running times of both algorithms. Note that the y axis is in log scale. The results show that our algorithm is two to three times faster than SearchMC on most of the datasets. As an example, it took SearchMC 5750 seconds (96 minutes) to generate all ADCs on the entire Tax dataset, while ADCEnum finished after 2373 seconds (39 minutes); that is, about 2.5 times faster.

In Figure 7, we present the running times of ADCMiner for all three approximation functions. The top and bottom diagrams depict the total running time and the running time of ADCEnum, respectively. Note that the running times of ADCEnum (which is the only part that depends on the choice of the approximation function) are very close for all three functions, and the total running time mostly depends on the evidence-set construction. For the same reason, we see, in Figure 8, that the total running time of our algorithm is not drastically lower than that of DCFinder [38], when considering the entire dataset (however, sampling completely changes the picture).

To construct the evidence set, we used the algorithm of Pena et al. [38], which is the fastest algorithm for that task. However, since their algorithm was not able to process the Tax and NCVoter datasets (using the parameters recommended by the authors) even when dedicating almost the entire memory of our machine to the Java heap, for these datasets we used the algorithm of Chu et al. [11] to construct the evidence set. Hence, we do not report on the running times of DCFinder on Tax and Voter in Figure 8. While for the Adult dataset, building the entire evidence set takes seven minutes, the evidence-set construction requires almost an hour and a half on Stock, more than twenty hours on Flight, and days on Tax and Voter. This highlights the importance of incorporating sampling in our algorithm, as we are able to reduce the running times by as much as 90%, as we explain in the next section.

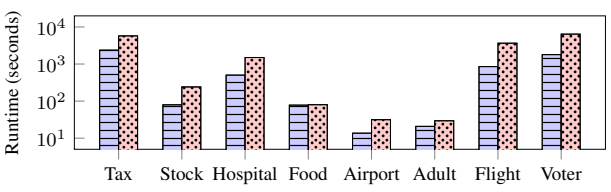

Figure 6: Running times of ADCEnum and SearchMC.

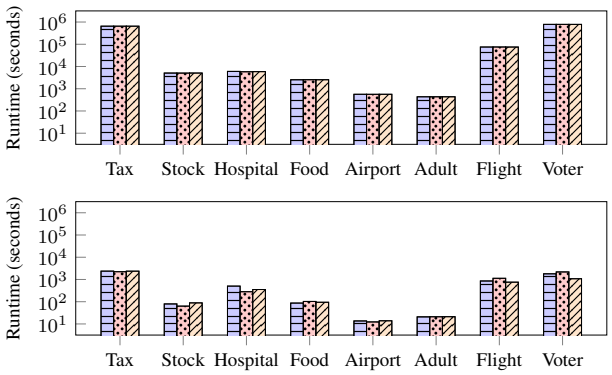

Figure 7: Total running time of ADCMiner (top) and running time of ADCEnum (bottom) for f_1, f_2, and f_3.

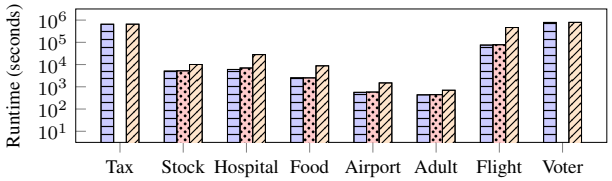

Figure 8: Running times of ADCMiner, DCFinder, and AFASTDC.

8.3 Sampling

We now report on the quality of the ADCs obtained from a sample. In all of our experiments, the sample size is big enough so that the term $z_{1-2\alpha} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$ in the approximation function f_1' defined in Section 7 has practically no impact on the function. Therefore, we use the same approximation function and threshold on both the sample and the entire dataset. In the experiments reported in Figure 9, we use a standard measure of quality, namely the F_1 score (i.e., $2 \cdot \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$). We compare the ADCs obtained from the sample with the ADCs obtained from the entire dataset.

The top charts of Figure 9 show the F_1 score for a fixed threshold $\epsilon = 0.1$ and varying sample sizes, ranging from 1% to 40% of the tuples, for all three approximation functions. Clearly, the larger the sample is, the more accurate the results we obtain. Generally, we see that to obtain an F_1 score of about 0.7 or above we need to see about 40% of the tuples. Note that we obtain a higher F_1-score on larger datasets (for which sampling is particularly important), as for such datasets a relatively small sample allows us to see enough tuples to obtain accurate results. For example, on the Tax and Voter datasets we consistently obtain an F_1-score of at least 0.7 or 0.8 when seeing 30% or 40% of the tuples, respectively.

The bottom charts of Figure 9 depict the F_1 score for a fixed sample size of 40% and varying thresholds, ranging from 0.01 to 0.2, for all three approximation functions. We see that we obtain more accurate results when considering a higher threshold, since a higher ϵ allows for more exceptions, and the DCs obtained using it can be seen as obtained using a smaller sample (as a smaller part of the database satisfies them). Hence, we are able to obtain results with high accuracy when considering a relatively small sample. We conclude that the choice of the right threshold and sample size should be based on the size of the original dataset and the approximation function (as we discuss in more details in the next section).

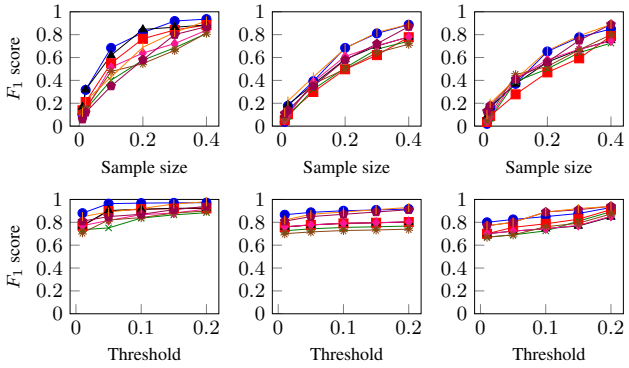

Figure 9: F_1 score for varying sample sizes and fixed $\epsilon = 0.1$ (top), and varying thresholds and fixed sample size 0.4 (bottom), under f_1 (left), f_2 (middle), and f_3 (right). Datasets: Tax (—●—), Stock (—▲—), Hospital (—■—), Food (—×—), Airport (—♦—), Adult (—✳—), Flight (—⬟—), and Voter (—+—).

Next, we show the improvement in running times obtained when using a sample. Figure 10 depicts the running times of ADCMiner for varying sample sizes on all datasets for the function f_1. (As shown in the previous section, the running times for all three functions are very close; hence, they all follow a similar trend.) On Stock, we are able to reduce the running time by more than 60% with a sample consisting of 40% of the tuples—from eighty five to thirty two minutes. For Flight, the running time goes down from almost twenty one hours to seventy minutes—a reduction of almost 95%. For Tax, we cannot use the same algorithm for constructing the evidence set on 100% and 40% of the tuples. Using the evidence-set construction algorithm of Chu et al. [11] we obtain a reduction of more than 94%—from 7.5 days to 10.5 hours. Using the algorithm BFASTDC to construct the evidence set we can obtain a similar reduction (of almost 90%) in the running time [37].

Finally, we validate the theoretical analysis of Section 7. For each dataset, we run our algorithm with the function f_1 on varying sample sizes ranging from 5% to 80% of the tuples. For each such sample, we compute the average value of $\epsilon - \hat{p}$ over the discovered ADCs (recall that $\hat{p}$ is the proportion of violating tuple pairs). Figure 11 depicts the values obtained in this experiment. The actual numbers are very small; hence, the reported numbers are scaled up (i.e., multiplied by a constant 10^x, where x depends on the dataset). We see that as the sample size increases, the value $\epsilon - \hat{p}$ decreases. Moreover, for each dataset, we have that $(\epsilon - \hat{p}) \sim \frac{1}{\sqrt{n}}$ (where $\sim$ denotes asymptotic equivalence and n is defined as in Section 7), which supports our main result of Section 7 (i.e., Inequality (3)).

8.4 Qualitative Analysis

We now compare the three approximation functions discussed in Section 5. For each dataset, we have a set of "golden" DCs obtained by domain experts. We take a sample of $10K$ tuples from each dataset and add noise to the resulting dataset, such that each value has a probability of 0.001 to be modified, and if it is modified, then it has 50% chance of being changed to a new value from the active domain of the corresponding column and 50% chance to being changed to a typo. We also generate another dirty dataset in a similar way, but here, we only allow changing values in 0.001 of the tuples. Hence, in the first dataset, the errors are distributed among the tuples (and the number of modified tuples is usually very close to the number of modified values), while in the second dataset, the errors are concentrated in a small subset of the tuples.

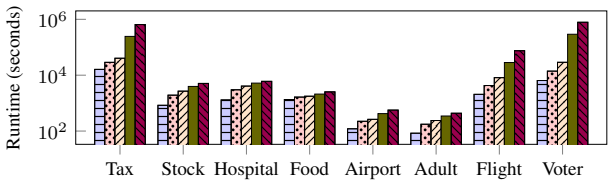

Figure 10: Running times of ADCMiner for varying sample sizes—20% (▯▯), 40% (▯▯), 60% (▯▯), 80% (▮▮), and 100% (▮▮).

Then, we run our algorithm on the two dirty datasets obtained from each one of the original datasets, with varying approximation thresholds ϵ (ranging from 10^{-6} to 10^{-1}). For each ϵ, we compute the G-recall, that is, the number of golden DCs returned divided by the total number of golden DCs. We report the results in Figure 12. We also report the G-recall for $\epsilon = 0$ (i.e., when considering valid DCs) above each diagram (in parentheses). We observe the following phenomena. First, the G-recall for valid DCs is consistently zero, or very close, which highlights the importance of considering approximate DCs. Second, the function f_1 produces results with a higher G-recall on smaller thresholds (i.e., $10^{-5} - 10^{-3}$), while the other two functions have a higher G-recall on the larger thresholds (i.e., $10^{-2} - 10^{-1}$). This is due to the fact that the functions f_2 and f_3 are more sensitive in the sense that a single tuple adds $\frac{1}{n}$ to the value of the functions f_2 and f_3 (where n is the number of tuples), while a pair of tuples adds $\frac{1}{n^2}$ to the value of the function f_1.

Another interesting phenomenon is that we consistently obtain a higher G-recall on the error-concentrated datasets (especially for the functions f_2 and f_3). This is expected, particularly for the function f_3, as when the errors are concentrated in a small subset of the tuples, these tuples will participate in every violation of the DC, and we only need to remove them from the database to satisfy the DC. The function f_3 (or, more accurately, our greedy algorithm) usually behaves better than the function f_2, especially on the error-concentrated datasets, and we are able to obtain a higher G-recall for a larger range of thresholds. As explained in Section 5, this is due to the fact that one erroneous tuple may result in a set of problematic tuples that contains every tuple in the database, while if we just remove this tuple, the DC will be satisfied. For this same reason, while with the function f_2 we constantly obtain the best accuracy using $\epsilon = 10^{-1}$, with the function f_3 we sometimes obtain better results with the smaller threshold $\epsilon = 10^{-2}$.

In the experiments reported in Figure 9, we have used six specific thresholds, with which we do not always obtain the highest possible G-recall. If we conduct a more refined analysis, we find that using the threshold 5×10^{-5} for the function f_1 on the Tax dataset, for example, we are able to obtain a G-recall of 1. Generally, when increasing the threshold, we obtain more general DCs (consisting of fewer predicates) that we do not obtain using smaller thresholds;

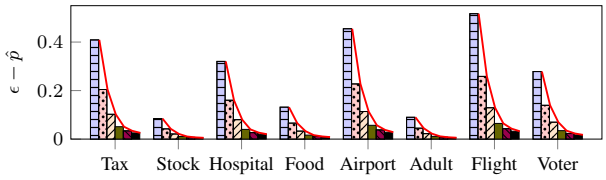

Figure 11: The average difference between ϵ and $\hat{p}$ over the ADCs obtained from varying sample sizes—5% (▯▯), 10% (▯▯), 20% (▯▯), 40% (▮▮), 60% (▮▮), and , 80% (▮▮).

Table 5: Approximate vs Valid DCs. Attributes: St – state, Ph – phone, G – gender, SE – single exemption, MC – measure code, OSt, DSt – origin and destination state, Dtime, ATime, ETime – departure, arrival, and elapsed time, C – county.

Approximate DC	Valid DC
$\forall t, t' \neg(t[\text{St}] = t'[\text{St}] \wedge t[\text{Salary}] > t'[\text{Salary}] \wedge t[\text{Tax}] < t'[\text{Tax}])$	$\forall t, t' \neg(t[\text{St}] = t'[\text{St}] \wedge t[\text{Salary}] > t'[\text{Salary}] \wedge t[\text{Tax}] < t'[\text{Tax}] \wedge t[\text{G}] = t'[\text{G}] \wedge t[\text{SE}] \geq t'[\text{SE}] \wedge t[\text{Ph}] = t'[\text{Ph}])$
$\forall t, t' \neg(t[\text{High}] < t[\text{Low}])$	$\forall t, t' \neg(t[\text{High}] < t[\text{Low}] \wedge t[\text{Open}] < t[\text{High}] \wedge t[\text{Low}] \leq t[\text{Close}])$
$\forall t, t' \neg(t[\text{St}] = t'[\text{St}] \wedge t[\text{MC}] = t'[\text{MC}] \wedge t[\text{StAvg}] \neq t'[\text{StAvg}])$	$\forall t, t' \neg(t[\text{St}] = t'[\text{St}] \wedge t[\text{MC}] = t'[\text{MC}] \wedge t[\text{StAvg}] \neq t'[\text{StAvg}] \wedge t[\text{City}] = t'[\text{City}] \wedge t[\text{Sample}] = t'[\text{Sample}])$
$\forall t, t' \neg(t[\text{Zip}] = t'[\text{Zip}] \wedge t[\text{St}] \neq t'[\text{St}])$	$\forall t, t' \neg(t[\text{Zip}] = t'[\text{Zip}] \wedge t[\text{St}] \neq t'[\text{St}]) \wedge t[\text{Name}] = t'[\text{Name}] \wedge t[\text{FacilityType}] \neq t'[\text{FacilityType}])$
$\forall t, t' \neg(t[\text{OSt}] = t'[\text{OSt}] \wedge t[\text{DSt}] = t'[\text{DSt}] \wedge t[\text{DTime}] \geq t'[\text{DTime}]) \wedge t[\text{ATime}] \leq t'[\text{ATime}]) \wedge t[\text{ETime}] > t'[\text{ETime}])$	$\forall t, t' \neg(t[\text{OSt}] = t'[\text{OSt}] \wedge t[\text{DSt}] = t'[\text{DSt}] \wedge t[\text{DTime}] \geq t'[\text{DTime}]) \wedge t[\text{ATime}] \leq t'[\text{ATime}]) \wedge t[\text{ETime}] > t'[\text{ETime}]) \wedge t[\text{Day}] > t'[\text{Day}])$
$\forall t, t' \neg(t[\text{Age}] < t'[\text{Age}] \wedge t[\text{BirthYear}] < t'[\text{BirthYear}])$	$\forall t, t' \neg(t[\text{Age}] < t'[\text{Age}] \wedge t[\text{BirthYear}] < t'[\text{BirthYear}] \wedge t[\text{C}] \neq t'[\text{C}] \wedge t[\text{Status}] \neq t'[\text{Status}] \wedge t[\text{Reason}] = t'[\text{Reason}])$

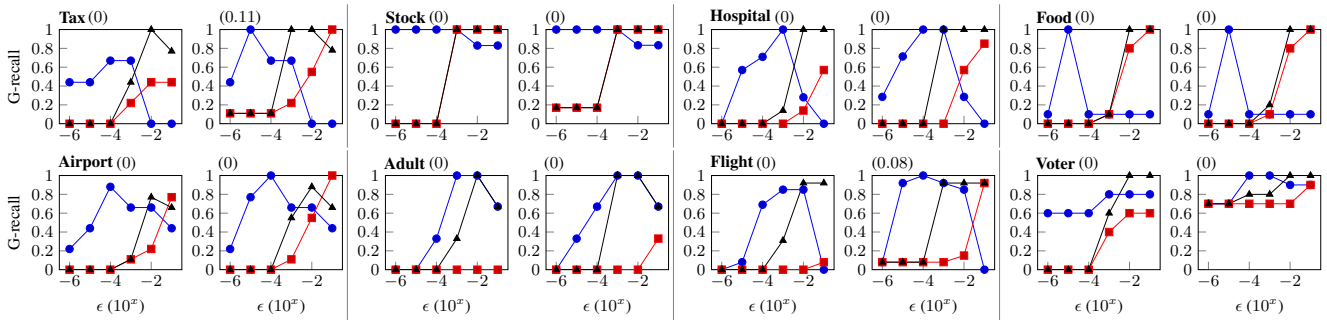

Figure 12: G-recall for varying thresholds under f_1 (—●—), f_2 (—■—) and f_3 (—▲—) for spread (left) and skewed (right) noise.

however, some DCs become "too general", and we may also lose some of the good DCs that we obtained with the smaller threshold. Using the above insights, we can choose a certain threshold (that depends on the approximation function), that will generate good results with high probability. Based on Figure 12, the best thresholds in that sense are 10^{-4}, 10^{-2}, and 10^{-1} for the functions f_1, f_2, and f_3, respectively. Using these thresholds we obtained an average G-recall of 0.71, 0.72, and 0.97, respectively.

Finally, Table 5 presents some of the golden DCs that we were able to obtain with the three functions using the best threshold according to Figure 12, as well as an example of a corresponding valid DC from the same dirty dataset, obtained with $\epsilon = 0$. The DCs were obtained from the Tax, SP Stock, Hospital, Food, Flight, and NCVoter datasets. Many valid DCs are obtained from a single ADC by adding predicates to cover for the errors in the database, which results in longer and fewer general DCs. Hence, we often obtain fewer DCs and shorter DCs when considering ADCs. However, this is not always the case, as sometimes we discover constraints that are ADCs, but cannot be extended to any minimal valid DC.

For example, the DC stating that the same zip code cannot correspond to two states (obtained from the Food dataset) becomes the DC stating that the same zip code cannot correspond to two states if the name and the type of the facility are the same. Clearly, we do not expect to obtain such complicated constraints, which strengthens our motivation for considering ADCs. In fact, while this DC generally holds, there are a few multi-state US zip codes (e.g., the zip code 84536 belongs to both Utah and Arizona). If our original database contained two tuples with the same zip code and different states we could not discover this DC unless considering ADCs. This example shows that ADCs are meaningful even when the database is clean, as they allow us to discover rules that are generally correct, but may have a few exceptions.

9. CONCLUDING REMARKS

We investigated the problem of detecting and enumerating minimal ADCs from data. We introduced a formal definition of an ADC based on a general family of approximation functions that subsumes previous proposals. We devised an algorithm for enumerating minimal ADCs and experimentally evaluated its performance on both real-world and synthetic datasets. Our experimental results showed that constructing the input to our enumeration algorithm requires orders of magnitude more time than enumerating the ADCs for large datasets. We showed that we are able to obtain good results (with high precision and recall) from a sample while avoiding the high computational cost. We also provided a theoretical analysis for the problem of discovering ADCs from a sample.

The computational complexity of the problem of enumerating (A)DCs remains open for future investigation (in terms of combined complexity, where both the schema and database are given as input). In particular, it would be interesting to understand whether this problem is equivalent, harder or easier than the minimal hitting-set problem. The main difference between the two problems is our knowledge about the relationships between the elements (e.g., we know that if a tuple pair does not satisfy a predicate then it satisfies the complement predicate). It is not clear how this additional information affects the complexity.

10. ACKNOWLEDGEMENTS

The work of Ester Livshits and Benny Kimelfeld was supported by the Israel Science Foundation (ISF), grants 1295/15 and 768/19, as well as the Deutsche Forschungsgemeinschaft (DFG) project 412400621 (DIP program). The work of Ester Livshits was also supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau.

11. REFERENCES

[1] R. Abreu and A. J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *SARA*, 2009.

[2] P. Arabie, S. A. Boorman, and P. R. Levitt. Constructing blockmodels: How and why. *Journal of mathematical psychology*, 17(1):21–63, 1978.

[3] R. Bar-Yehuda and S. Even. A linear-time approximation algorithm for the weighted vertex cover problem. *J. Algorithms*, 2(2):198–203, 1981.

[4] T. Bleifuß, S. Kruse, and F. Naumann. Efficient denial constraint discovery with hydra. *PVLDB*, 11(3):311–323, 2017.

[5] M. Boullé. Universal approximation of edge density in large graphs. *arXiv preprint arXiv:1508.01340*, 2015.

[6] N. Bus, N. H. Mustafa, and S. Ray. Practical and efficient algorithms for the geometric hitting set problem. *Discrete Applied Mathematics*, 240:25–32, 2018.

[7] N. Cardoso and R. Abreu. MHS2: A map-reduce heuristic-driven minimal hitting set search algorithm. In *MUSEPAT*, pages 25–36, 2013.

[8] K. Chandrasekaran, R. M. Karp, E. Moreno-Centeno, and S. Vempala. Algorithms for implicit hitting set problems. In *SODA*, pages 614–629, 2011.

[9] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008.

[10] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2):90–121, 2005.

[11] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[12] C. Combi, M. Mantovani, A. Sabaini, P. Sala, F. Amaddeo, U. Moretti, and G. Pozzi. Mining approximate temporal functional dependencies with pure temporal grouping in clinical databases. *Comp. in Bio. and Med.*, 62:306–324, 2015.

[13] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.

[14] U. Feige. On sums of independent random variables with unbounded variance and estimating the average degree in a graph. *SIAM Journal on Computing*, 35(4):964–984, 2006.

[15] P. A. Flach and I. Savnik. Database dependency discovery: A machine learning approach. *AI Commun.*, 12(3):139–160, 1999.

[16] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[17] A. Gainer-Dewar and P. Vera-Licona. The minimal hitting set generation problem: Algorithms and computation. *SIAM J. Discrete Math.*, 31(1):63–100, 2017.

[18] O. Goldreich and D. Ron. On estimating the average degree of a graph. *Electronic Colloquim on Computational Complexity (ECCC)*, 2004.

[19] E. Gribkoff, G. V. den Broeck, and D. Suciu. The most probable database problem. 2014.

[20] A. Heidari, I. F. Ilyas, and T. Rekatsinas. Approximate inference in structured instances with noisy categorical observations. In *UAI*, page 152. AUAI Press, 2019.

[21] A. Heidari, J. McGrath, I. F. Ilyas, and T. Rekatsinas. Holodetect: Few-shot learning for error detection. In *SIGMOD Conference*, pages 829–846. ACM, 2019.

[22] P. W. Holland, K. B. Laskey, and S. Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

[23] 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.

[24] J. Kivinen and H. Mannila. *Approximate dependency inference from relations*, pages 86–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[25] W. Li, Z. Li, Q. Chen, T. Jiang, and Z. Yin. Discovering approximate functional dependencies from distributed big data. In *APWeb*, pages 289–301, 2016.

[26] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - A review. *IEEE Trans. Knowl. Data Eng.*, 24(2):251–264, 2012.

[27] E. Livshits, A. Heidari, I. F. Ilyas, and B. Kimelfeld. Approximate denial constraints. *CoRR*, abs/2005.08540, 2020.

[28] E. Livshits, I. F. Ilyas, B. Kimelfeld, and S. Roy. Principles of progress indicators for database repairing. *CoRR*, abs/1904.06492, 2019.

[29] E. Livshits, B. Kimelfeld, and S. Roy. Computing optimal repairs for functional dependencies. *ACM Trans. Database Syst.*, 45(1):4:1–4:46, 2020.

[30] A. Lopatenko and L. E. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, pages 179–193, 2007.

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

[32] F. Lorrain and H. C. White. Structural equivalence of individuals in social networks. *The Journal of mathematical sociology*, 1(1):49–80, 1971.

[33] K. Murakami and T. Uno. Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics*, 170:83–94, 2014.

[34] L. Nourine, A. Quilliot, and H. Toussaint. Partial enumeration of minimal transversals of a hypergraph. In *CLA*, pages 123–134, 2015.

[35] N. Novelli and R. Cicchetti. FUN: an efficient algorithm for mining functional and embedded dependencies. In *ICDT*, pages 189–203, 2001.

[36] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDB*, 8(10):1082–1093, 2015.

[37] E. H. M. Pena and E. C. de Almeida. BFASTDC: A bitwise algorithm for mining denial constraints. In *DEXA*, pages 53–68, 2018.

[38] E. H. M. Pena, E. C. de Almeida, and F. Naumann. Discovery of approximate (and exact) denial constraints. *PVLDB*, 13(3), 2019.

[39] J. Rammelaere and F. Geerts. Revisiting conditional functional dependency discovery: Splitting the "c" from the "fd". In *ECML/PKDD (2)*, volume 11052 of *Lecture Notes in Computer Science*, pages 552–568. Springer, 2018.

[40] S. E. Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[41] S. A. Vinterbo and A. Öhrn. Minimal approximate hitting sets and rule templates. *Int. J. Approx. Reasoning*,

25(2):123–143, 2000.

[42] 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.