Information Systems 108 (2022) 102041

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Diversifying repairs of Denial constraint violations

Shuai Li [a,b], Yue Zhang [a,b], Zijing Tan [a,b,*], Shuai Ma [c]

[a] School of Computer Science, Fudan University, China
[b] Shanghai Key Laboratory of Data Science, China
[c] SKLSDE Lab, School of Computer Science and Engineering, Beihang University, China

ARTICLE INFO

Article history:
Received 12 November 2021
Received in revised form 21 February 2022
Accepted 25 March 2022
Available online 31 March 2022
Recommended by Lukasz Golab

Keywords:
Algorithms
Data repairing
Denial constraints

ABSTRACT

Denial constraints (DCs) are expressive enough to subsume many other dependencies, and proven useful in data cleaning for improving data quality. As a complement to the methods of computing a single (nearly) optimum repair of DC violations, in this paper we make the first effort to diversify repairs of DC violations, aiming to generate a set of diversified repairs. (1) We adapt the concept of cardinality-set-minimal repairs to DCs, and relate a cardinality-set-minimal repair to a minimal vertex cover of the conflict hypergraph *w.r.t.* a given set Σ of DCs on a relational instance I. (2) We formalize the problem of diversifying cardinality-set-minimal repairs of DC violations, and address the problem by presenting a set of algorithms and optimizations to generate a set of diversified minimal vertex covers of the conflict hypergraph. (3) Using both real-life and synthetic data, we conduct extensive experiments to verify the effectiveness and efficiency of our methods.

1. Introduction

Real-life data are usually dirty, due to, *e.g.,* errors, conflicts and inconsistencies. This highlights the quest for effective and efficient data quality management [1,2]. Data repairing techniques are at the core of data quality management and extensively studied in the literature. Denial constraints (DCs) [3,4] are general enough to subsume many other data dependencies, *e.g.,* unique column combinations (UCCs), functional dependencies (FDs), conditional functional dependencies (CFDs) [5], and pointwise order dependencies (PODs) [6,7]. Therefore, DCs are well employed in data repairing for improving data quality [4,8–11]. We first give some examples of DCs.

Example 1. Consider a sample relational instance I, as shown in Table 1. It contains 7 tuples and each tuple concerns an employee's information, including name (Name), phone (Phone), pay level (Level), department (DepID), salary (Salary), tax (Tax), city (City) and state (State). There are some data dependencies defined on I.

(1) A UCC states two tuples cannot have the same values in attributes Name and Phone, *i.e.,* the combination of Name and Phone forms a candidate key.

(2) An FD states that two employees that work in the same department (DepID) and have the same salary (Salary) must have the same pay level (Level).

(3) A CFD states that for any tuple, if its value in attribute City is "Las Vegas", then it has "NV" in attribute State.

(4) A POD states that for any two tuples that have the same value in State, the tuple with a larger value in Salary cannot have a smaller value in Tax than the other tuple.

All of the above four constraints can be encoded as DCs, in a universally quantified first order logic formalism (the formal definition of DCs will be reviewed in Section 3).

(1) φ_1: $\forall\ t_\alpha, t_\beta \in R, \neg(t_\alpha[Name] = t_\beta[Name] \wedge t_\alpha[Phone] = t_\beta[Phone])$

(2) φ_2: $\forall t_\alpha, t_\beta \in R, \neg(t_\alpha[DepID] = t_\beta[DepID] \wedge t_\alpha[Salary] = t_\beta[Salary] \wedge t_\alpha[Level] \neq t_\beta[Level])$

(3) φ_3: $\forall t_\alpha \in R, \neg(t_\alpha[City] = \text{"Las Vegas"} \wedge t_\alpha[State] \neq \text{"NV"})$

(4) φ_4: $\forall\ t_\alpha, t_\beta \in R, \neg(t_\alpha[State] = t_\beta[State] \wedge t_\alpha[Salary] > t_\beta[Salary] \wedge t_\alpha[Tax] < t_\beta[Tax])$ □

Due to errors in data, the instance I is *inconsistent w.r.t.* the set $\{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$ of DCs in the sense that it violates some of the DCs. Data repairing techniques can be employed to restore I

* Corresponding author at: School of Computer Science, Fudan University, China.
E-mail addresses: 20210240207@fudan.edu.cn (S. Li), 19210240126@fudan.edu.cn (Y. Zhang), zjtan@fudan.edu.cn (Z. Tan), mashuai@buaa.edu.cn (S. Ma).

https://doi.org/10.1016/j.is.2022.102041
0306-4379/

Table 1
Relation of Employee: I.

TID	Name	Phone	Level	DepID	Salary	Tax	City	State
t_1	W. Donald	303–572	4	A550	30,000	2,300	Middleport	OH
t_2	L. John	225–945	1	A031	8,000	1,200	Shreveport	LA
t_3	R. Scott	219–904	2	A002	50,000	1,000	Shreveport	LA
t_4	E. Jason	215–793	3	A550	30,000	600	Las Vegas	LA
t_5	R. Gregory	262–404	2	A550	11,000	1,100	Maida	ND
t_6	W. Donald	303–572	2	A005	12,000	800	Middleport	OH
t_7	B. Scott	229–336	1	A005	7,000	0	Lima	NY

to a consistent state, *i.e.,* a *repair* of I. In the literature [4,11–16], the most commonly used operation for repairing an instance I is to modify some attribute values in I. Note that there are usually many different ways to resolve inconsistencies.

Example 2. We first show some DC violations in the instance I. (a) The tuple pair $\langle t_1, t_6\rangle$ violates φ_1 because t_1, t_6 have the same values in attributes (Name, Phone). (b) $\langle t_1, t_4\rangle$ violates φ_2, because t_1, t_4 are in the same department and have the same salary, but have different pay levels. (c) Tuple t_4 violates φ_3 by itself. (d) There are two tuple pairs $\langle t_2, t_3\rangle$ and $\langle t_2, t_4\rangle$ that violate φ_4.

We then modify attribute values to resolve the violations. Recall that repairing violations is usually measured by the *repair cost*. The simplest and also mostly adopted cost measure, is computed as the number of changes in the repair [4,11,15]. It can be verified we must modify at least four attribute values to obtain a repair of I. For example, we can get a repair by setting t_1[Name] = "L.Johh", t_2[Salary] = "60,000", t_4[City] = "Shreveport" and t_4[DepID] = "A005". The ways to get a repair are usually not unique. We can get a different repair by setting t_1[Phone] = "218-750", t_2[Tax] = "600", t_4[State] = "NV", t_4[Salary] = "20,000", and another one by setting t_1[DepID] = "A031", t_2[Tax] = "400", t_4[City] = "Shreveport" and t_6[Name] = "B.Scott".

In terms of the number of modified attribute values, we cannot judge which of the three repairs is the best. Intuitively, this motivates the quest for studying an approach to multiple repairs instead of a single one.

From another point of view, the above-mentioned three repairs are *diversified* in the sense they are dissimilar to each other and hence provide good coverage of the different ways to restore the consistency. In contrast, there are repairs that fall short of diversification, *i.e.,* repairs having the same or similar set of modified attribute values. For example, a repair of {t_1[Name] = "L.John", t_2[Tax] = "400", t_4[Salary] = "20,000", t_4[State] = "NV"} is similar to that of {t_1[Name] = "L.John", t_2[Salary] = "60,000", t_4[Salary] = "20,000", t_4[State] = "NV"}, and that of {t_6[Phone] = "101-708", t_2[Tax] = "400", t_4[Salary] = "20,000", t_4[State] = "NV"}. Similar repairs bring limited novel information. This suggests besides repair cost, the idea of *diversification* should be considered when generating a set of repairs. □

Existing works on repairing DC violations, *e.g.,* [4,9–11], mainly focus on the problem of *optimum* repair computation, aiming at computing a single (nearly) optimum repair according to the *repair cost*. As opposed to these works, in this paper we make the first effort to study the problem of generating a set of repairs of DC violation, and in particular, a set of *diversified* repairs. We highlight some scenarios where optimum repair computation is not sufficient but generating multiple (diversified) repairs is desirable.

(1) Optimum repair computation methods aim to find a single repair with the smallest cost among all repairs, but some repairs can indeed have the same smallest cost. Even if the "best" repair exists, no methods guarantee to find it because all methods are heuristics due to the intractability of the optimum repair computation problem [4]. Indeed, our experimental evaluations show that our methods of generating multiple repairs can find repairs better than the repairs from [4,11] in terms of repair cost.

(2) Automatic data repairing is hard to operate in practice, and is usually combined with some manual involvement in an interactive way [17–20]. In an interactive repairing approach, usually several (representative) repairs can help users give feedbacks or comments in a convenient way, since repairs can serve as references to each other, while a single repair is not sufficient.

(3) As remarked in [21,22], by treating each possible repair as a possible world, *uncertain query answering* can be obtained on a dirty dataset following the Monte Carlo approach [23]. Similar idea is recently adopted in [8], which performs probabilistic repair of DC violations on-demand for user queries. Again, a set of repairs rather than a single repairs is required in this scenario.

Contributions. In this paper, we make the first effort to generate a set of diversified repairs of DC violations.

(1) We justify the space of repair generation by adapting cardinality-set-minimal repairs to DCs. For a given set Σ of DCs on a relational instance I, we relate a cardinality-set-minimal repair to a minimal vertex cover of the conflict hypergraph of Σ on I, by providing an algorithm that generates a cardinality-set-minimal repair that modifies exactly all cells (attribute values) from the vertex cover (Section 4).

(2) We formalize the problem of diversifying repairs, and address it by providing a set of algorithms and optimizations to generate a set of diversified minimal vertex covers of the conflict hypergraph (Section 5 and Section 6). The methods to enumerate all minimal vertex covers of a hypergraph, *a.k.a.* hypergraph traversal or hitting set enumeration, are extensively studied [24,25]. However, we are not aware of any methods of diversifying minimal vertex covers. We contend that our methods are valuable in themselves and can find applications in many domains.

(3) We conduct an experimental study to verify our approach, on both real-life and synthetic data (Section 7). The results show that our methods can efficiently generate a set of repairs with good diversification, and can find repairs with smaller repair cost than the repairs produced by former works on optimum repair computation.

2. Related work

As one of the most important aspects of data quality management, data repairing (cleaning) methods are studied for a host of constraints and data quality rules; see [1,2,26] for overviews of this topic. In this section we discuss works close to ours: data repairing of DC violations and techniques for sampling repairs of constraints.

Data repairing with DCs. DCs [3,4] subsume many kinds of constraints, and lend themselves to wide applicability for data repairing. [27,28] study the problem of repairing numerical attributes with linear denial constraints. Repairing an inconsistent instance I *w.r.t.* a set Σ of general DCs is studied in [4], aiming to find a repair I' satisfying Σ with the minimum repair cost. [11] considers the case Σ may be inaccurate, proposes to

repair I by allowing a small variation on Σ and finds a repair that satisfies at least one variant of Σ. [10] unifies qualitative data repairing leveraging DCs with quantitative data repairing leveraging statistical properties, in a framework based on probabilistic inference. [29] resolves the inconsistencies and conflicts in a unified approach, by combining DCs with source reliability. [8] presents a query result relaxation approach that performs probabilistic repairing of DC violations as requested by user queries. [9] introduces the notion of *Delta Rules* that combines DCs, causal rules and database deletion triggers, studies several alternative semantics for repairs, and investigates expressiveness and computational complexity of these different semantics. In addition, there are some centralized or distributed systems built to repair DC violations [30–32].

Different from these works, we study the generation of a set of diversified repairs. We justify the space of repair generation, relate a cardinality-set-minimal repair to a minimal vertex cover of the conflict hypergraph, and present a set of algorithms and optimizations to diversify minimal vertex covers. In particular, the techniques of conflict hypergraph [4] and *repair context over suspect* [4,11] are employed in our methods, for encoding DC violations and for generating repairs from minimal vertex covers.

Sampling repairs of constraint violations. To our best knowledge, we are not aware of any works on generating multiple repairs of DC violations. The methods of random sampling of one repair of FD and CFD violations each time are studied in [21,22]. Leveraging the sampling method in [21], techniques for selecting a set of diversified repairs of FD violations are studied in [33].

This work differs in the following. (1) We study generating multiple repairs of DC violations, which is necessarily much harder than sampling repairs of FD (CFD) violations. The increasing complexity is demonstrated in techniques for computing repairs with DCs [4,11], compared to those with FDs (CFDs) [12–15]. (2) We establish the connection between diversified repairs of DC violations and diversified minimal vertex covers of the conflict hypergraph. We present novel methods to generate diversified minimal vertex covers; the methods are valuable in themselves.

3. Preliminaries

In this section, we review basic notations of DCs and repairs of DC violations.

We denote a relation schema by $R(A_1, \ldots, A_k)$, a relational instance of R by I, and tuples in I by t, s. For a given tuple t from I and an attribute $A \in R$, we denote by $I(t, A)$ the value of attribute A in t from I.

Denial constraints (DCs) [3,4]. Each DC φ is a negation of conjunction of *predicates*, given in a universally quantified first order logic formalism: φ: $\forall t_\alpha, t_\beta \in R, \neg(P_1 \wedge \cdots \wedge P_m)$

Herein, each predicate P_i ($i \in [1, m]$) is in the form of $v_1 \phi v_2$ or $v_1 \phi c$, where v_1, v_2 is of the form $t_x[A]$, $x \in \{\alpha, \beta\}$, A is an attribute of R, c is a constant, and $\phi \in \{<, \leq, >, \geq, =, \neq\}$. A DC is called a *single-tuple* DC if all predicates are in the form of $t_\alpha[A]\phi c$ or $t_\alpha[A]\phi t_\alpha[B]$, *i.e.,* only one tuple is concerned. When φ is imposed on an instance I, $I(t_\alpha, A)$ and $I(t_\beta, A)$ are used to compute the satisfaction of all the predicates. φ is satisfied iff at least one of the predicates is *unsatisfied*. Otherwise, there are DC violations. I is consistent *w.r.t.* φ iff there is no tuple pair $\langle t, s\rangle$ that violates φ (or a single tuple t if φ is a *single-tuple* DC), written as $I \models \varphi$. For a set Σ of DCs, we write $I \models \Sigma$ iff $I \models \varphi$ for every $\varphi \in \Sigma$.

Repairs of DC violations. Given an inconsistent instance I *w.r.t.* a set Σ of DCs, a *repair* of I is another instance I' obtained by modifying attribute values in I such that $I' \models \Sigma$.

Along the same lines as former works [4,11–16], only value modifications are used to repair an inconsistent instance. Hence, given a tuple t and an attribute $A \in R$, we identify the same position in I and its repair I'. We refer to the position as a *cell*, and use $t[A]$ to denote the position. In contrast, I and I' may have different values in the same cell, and we denote by $I(t, A)$ (resp. $I'(t, A)$) the value of attribute A in t from I (resp. I').

4. From minimal vertex cover to repair

In this section, we first adapt the concept of *cardinality-set-minimal* repairs to DCs, to justify the space of repair generation. We then establish the connection between a cardinality-set-minimal repair and a minimal vertex cover of the conflict hypergraph of a given set Σ of DCs on a relational instance I.

4.1. Repair space

It is important to figure out the criterion we adopt in generating repairs of DC violations. In this subsection, we review three notions of repair and justify the cardinality-set-minimal repair considered in this paper.

For a given repair I' of I *w.r.t.* a set Σ of DCs, we denote by $\Delta(I, I')$ the set of cells at which I and I' have different values, *i.e.,* $\Delta(I, I') = \{t[A] \mid I(t, A) \neq I'(t, A)\}$. We denote by $\lambda(I, I')$ the set of pairs of the cell modified in I' and the new value, *i.e.,* $\lambda(I, I') = \{(t[A], I'(t, A)) \mid t[A] \in \Delta(I, I')\}$.

There are different notions of repair in the literature, namely *cardinality-minimal*, *set-minimal*, *cardinality-set-minimal* repairs, respectively. They are considered for sampling repairs of FD (CFD) violations in [21,22].

Cardinality-minimal repair [4,15]. A repair I' of I is cardinality-minimal iff there is no repair I'' of I such that $|\Delta(I, I'')| < |\Delta(I, I')|$.

Set-minimal repair [34]. A repair I' of I is set-minimal iff there is no repair I'' of I such that $\lambda(I, I'') \subset \lambda(I, I')$.

Cardinality-set-minimal repair [21,22]. A repair I' of I is cardinality-set-minimal iff there is no repair I'' of I such that $\Delta(I, I'') \subset \Delta(I, I')$.

We see the following. (1) I' is a cardinality-minimal repair iff it has the minimum number of modified cells among all possible repairs. (2) I' is a set-minimal repair, iff $I' \not\models \Sigma$ if we change the value of any modified cell in I' back to its original value in I. (3) I' is a cardinality-set-minimal repair, iff there does not exist another repair I'' such that the set of modified cells in I'' is a proper subset of the set of modified cells in I'. Note the same cell can have different values in I' and I''. (4) By the definitions, a cardinality-minimal repair is always a cardinality-set-minimal repair, and a cardinality-set-minimal repair is always a set-minimal repair.

Example 3. In Fig. 1, the instance I violates the DC φ, and we show different types of repairs. Specifically, (1) I_1 is a cardinality-minimal repair. It can be verified that the number of modified cells in I_1 is the minimum among all possible repairs. (2) I_2 is a cardinality-set-minimal repair, since we cannot generate a repair by modifying only $t_1[C]$ or $t_2[C]$ (possibly with different values from I_2). (3) I_3 is a set-minimal repair, since changing any subset of $\{t_1[C], t_2[C], t_3[C]\}$ back to their original values does not lead to a repair. (4) I_4 is not a set-minimal repair. This is because it already suffices to repair I by setting $t_3[C] = 2$, *i.e.,* the modification of $t_3[B]$ is unnecessary. □

Different notions of repair lead to different spaces of repair generation. We consider generating cardinality-set-minimal repairs of DC violations, along the same lines as [21,22] for sampling repairs of FD (CFD) violations.

I

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

φ: $\forall t_i, t_j \in I, \neg(t_i[A] = t_j[A] \wedge t_i[B] \neq t_j[B] \wedge t_i[C] < t_j[C])$

I_1

	A	B	C
t_1	1	1	2
t_2	1	2	2
t_3	1	3	2

Cardinality-Minimal
Cardinality-Set-Minimal
Set-Minimal

I_2

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

Not Cardinality-Minimal
Cardinality-Set-Minimal
Set-Minimal

I_3

	A	B	C
t_1	1	1	6
t_2	1	2	6
t_3	1	3	6

Not Cardinality-Minimal
Not Cardinality-Set-Minimal
Set-Minimal

I_4

	A	B	C
t_1	1	1	2
t_2	1	2	2
t_3	1	1	2

Not Cardinality-Minimal
Not Cardinality-Set-Minimal
Not Set-Minimal

Fig. 1. Different types of repairs.

(1) It is known cardinality-minimal repairs *cannot* be efficiently generated. It is NP-hard to generate a single cardinality-minimal repair of FD violations [12,13,15], and this negative result also applies to DCs that subsume FDs. Existing heuristic DC repairing methods, *e.g.,* [4,11], cannot guarantee to compute cardinality-minimal repairs.

(2) The space of set-minimal repairs is larger than that of cardinality-set-minimal repairs. If a set-minimal repair is not a cardinality-set-minimal repair, then there must exist a cardinality-set-minimal repair that only modifies some (not all) cells that are modified in the set-minimal repair. Thus, cardinality-set-minimal repairs are preferable to set-minimal repairs, following the principle of *minimality of changes* [4,11–13,15].

4.2. Method to compute cardinality-set-minimal repair

In this subsection, we present an algorithm to compute cardinality-set-minimal repairs, which is built upon the connection between a cardinality-set-minimal repair and a *minimal vertex cover* of the *conflict hypergraph*.

Conflict hypergraph. The technique of conflict hypergraph is originally introduced for encoding FD violations [15], and further extended to DC violations [4]. A *hypergraph* contains a set of *hyperedges* and each hyperedge can contain an arbitrary number of vertices. In the conflict hypergraph $\mathcal{G} = (V, E)$ of an instance I *w.r.t.* a set Σ of DCs, each hyperedge from E corresponds to a DC violation with all cells involved in the violation as vertices, and a cell belongs to V if it involves in any DC violations.

Example 4 (*Example 2 Continued*)**.** We show the conflict hypergraph in Fig. 2. There are 5 hyperedges, and each hyperedge denotes a DC violation. For example, e_5 denotes t_4[State] and t_4[City] violate φ_3; the value of t_4[City] is "Las Vegas" but the value of t_4[State] is not "NV". □

A repair must modify at least one cell involved in a violation to resolve the violation, and all violations must be eliminated to obtain a repair. In terms of the conflict hypergraph, this implies that all cells modified in a repair form a *vertex cover*, *a.k.a. hitting set* [24,25], of the conflict hypergraph, *i.e.,* the set of modified cells (vertices) intersects with every hyperedge.

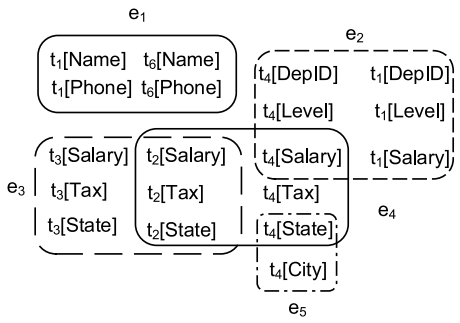

Fig. 2. Conflict hypergraph of Example 2.

Based on this observation, former works on repairing DC violations [4,11] first obtain a *minimum vertex cover* that has the minimum number of modified cells, and then repair cells from the cover. The methods are necessarily heuristics, since it is NP-hard to find a minimum vertex cover [35]. In contrast, as will be seen shortly, we can obtain a cardinality-set-minimal repair based on a *minimal vertex cover*. Recall a vertex cover is minimal if no proper subset of it is a vertex cover, and it is polynomial to find a minimal vertex cover.

Example 5 (*Example 4 Continued*)**.** We can see {t_1[Name], t_3[Salary], t_4[City], t_4[Level], t_4[Tax]} is a minimal vertex cover, while {t_1[Name], t_3[Salary], t_4[Salary], t_4[City]} is a minimum vertex cover. □

In the sequel, we establish the connection between a cardinality-set-minimal repair and a minimal vertex cover, by presenting an algorithm (Algorithm 1) that takes a minimal vertex cover as input and outputs a cardinality-set-minimal repair by modifying exactly all cells from the cover. We first review some DC repairing techniques [4,11] employed in the algorithm.

Suspect set [11]. Given a cell $c = t[A]$ and a φ from the set Σ, the suspect set $susp(c, \varphi)$ is a set of tuple pairs. Each tuple pair in $susp(c, \varphi)$ is of the form $\langle t, s\rangle$ or $\langle s, t\rangle$. Specifically, $\langle t, s\rangle$ (resp. $\langle s, t\rangle$) belongs to $susp(c, \varphi)$ iff (a) φ concerns c; and (b) $\langle t, s\rangle$ (resp. $\langle s, t\rangle$) satisfies all the predicates in φ excluding those concerning c. We denote by $susp(c, \Sigma)$ the set of $susp(c, \varphi)$ for all $\varphi \in \Sigma$.

Example 6 (*Example 2 Continued*)**.** We compute the suspect set for t_4[Salary] *w.r.t.* $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$. (1) For φ_2, four tuple pairs, *i.e.,*$\langle t_1, t_4\rangle$, $\langle t_4, t_1\rangle$, $\langle t_4, t_5\rangle$ and $\langle t_5, t_4\rangle$, satisfy all the predicates of φ_2 excluding the one concerning t_4[Salary]. Thus, $susp(c, \varphi_2) = \{\langle t_1, t_4\rangle, \langle t_4, t_1\rangle, \langle t_4, t_5\rangle, \langle t_5, t_4\rangle\}$. (2) Similarly, we have $susp(c, \varphi_4) = \{\langle t_4, t_2\rangle, \langle t_4, t_3\rangle\}$. (3) φ_1 and φ_3 have no predicates concerning t_4[Salary]. (4) Finally, we have $susp(t_4[\text{Salary}], \Sigma) = \{\varphi_2$: $\{\langle t_1, t_4\rangle, \langle t_4, t_1\rangle, \langle t_4, t_5\rangle, \langle t_5, t_4\rangle\}$, φ_4: $\{\langle t_4, t_2\rangle, \langle t_4, t_3\rangle\}\}$. □

For $c = t[A]$ and $\langle t, s\rangle \in susp(c, \varphi)$, note whether the predicates concerning c are satisfied by $\langle t, s\rangle$ is irrelevant in the computation of $susp(c, \varphi)$. Observe the following. (1) Recall a tuple pair violates φ if all the predicates in φ are satisfied by the pair. If $\langle t, s\rangle$ satisfies the predicates concerning c, then $\langle t, s\rangle$ violates φ. That is, $susp(c, \varphi)$ contains all the violating tuples of φ *w.r.t.* the tuple t. (2) If $\langle t, s\rangle$ does not satisfy the predicates concerning c, then s is considered as a *suspect* [11]. Intuitively, $\langle t, s\rangle$ may form a violating tuple pair of φ after modifying the

Fig. 3. The repair context of cell t_4[Salary].

Algorithm 1: GetRepair

Input: An instance I, a set Σ of DCs, and a minimal vertex cover $\mathcal{C}$ of the conflict hypergraph $\mathcal{G}$
Output: A *cardinality-set-minimal* repair I'

```
I' ← I;
foreach cell c ∈ C do
    repairOneCell(c);
return I';

Function repairOneCell(cell c)
    susp ← compute susp(c, Σ) on I';
    rc ← compute the repair context over susp;
    newvalue ← compute a value for c according to rc;
    modify the value of c to be newvalue in I';
```

value of $t[A]$, since all the predicates irrelevant to c in φ are already *satisfied*.

Repair context over suspect set [11]. The repair context of a cell $c = t[A]$ is the conjunction of expressions that are comparisons between c and another cell or a constant. Specifically, in the repair context (a) each cell is either c, or a cell $s[B]$ where $\langle t, s\rangle$ or $\langle s, t\rangle \in susp(c, \Sigma)$ and $s[B]$ is involved in a predicate concerning c; and (b) each constant is involved in a predicate concerning c. The rationale behind repair context is that if a value for c can be found such that all the expressions in the repair context are satisfied simultaneously, then all violations concerning c are resolved without introducing no new DC violations.

Example 7 (*Example 6 Continued*)**.** According to the suspect set, we have the repair context of t_4[Salary] as shown in Fig. 3. As an example, consider $\langle t_4, t_2\rangle$ from $susp(c, \varphi_4)$. We get the expression t_4[Salary] $\leq t_2$[Salary] by reversing t_4[Salary] $> t_2$[Salary] according to φ_4. Intuitively, by choosing a value for t_4[Salary] that satisfies the expression, we guarantee that $\langle t_4, t_2\rangle$ no longer violates φ_4. As another example, consider $\langle t_4, t_5\rangle$ from $susp(c, \varphi_2)$. We get the expression t_4[Salary] $\neq t_5$[Salary]. The new value of t_4[Salary] is required to satisfy this expression, to avoid introducing new violations. □

Algorithm. GetRepair (Algorithm 1) is given to generate a cardinality-set-minimal repair, by repairing exactly all the cells from a minimal vertex cover of the conflict hypergraph. The key is to modify cells from the cover one by one and avoid introducing new violations. This is achieved by computing a value for each cell c based on the repair context of c. The repair context of c is built according to I'. Hence, the new values assigned to cells before c are considered in computing the repair context of c. The methods to choose a new value for c according to its repair context are studied in [4,11]. Roughly speaking, (a) if possible, a value that satisfies all the expressions in the repair context is chosen. When there are several possible values, the value that minimizes the difference between the original and new values is selected, by using, *e.g.,* string cardinality minimality for string attributes and numerical distance minimality for numerical attributes. (b) As noted in [4,11], a repair context may fail to be satisfied if no value exists that can satisfy all the expressions in the context simultaneously. In this case, a *fresh new value* is used. A fresh new value is a value outside the current active domain and does not satisfy any predicates with it, which guarantees to eliminate any DC violations.

Example 8. For a given minimal vertex cover $\mathcal{C} = \{t_1$[Name], t_3[Salary], t_4[Salary], t_4[City]$\}$ of the conflict hypergraph in Fig. 2, we can modify the value of t_4[Salary] from 30,000 to 8,000 based on the repair context of t_4[Salary] illustrated in Example 7. Similarly, we can modify the values of t_3[Salary], t_1[Name] and t_4[City] to be 8,000, "L.John" and "Maida", respectively. □

Complexity. To repair a cell c from $\mathcal{C}$, (1) it takes $O(|I||\Sigma|)$ to compute the suspect set $susp(c, \Sigma)$, where $|I|$ is the number of tuples in I and $|\Sigma|$ is the number of DCs in Σ; (2) it takes $O(|susp(c, \Sigma)|)$ to compute the repair context, where $|susp(c, \Sigma)|$ is the number of tuple pairs in $susp(c, \Sigma)$. Note that $|susp(c, \Sigma)| \leq 2|I||\Sigma|$; and (3) for simplicity, we omit the time for finding a value of c according to the repair context. To sum up, the worst-case complexity of GetRepair is $O(|\mathcal{C}||I||\Sigma|)$.

Remark. We highlight the differences between GetRepair and the methods of [4,11]. The method of [4] considers repair context without suspect set, and it may introduce new violations in the repair process, as noted in [11]. The method of [11] computes the repair contexts of all the cells from the minimum vertex cover, and tries to find values for all the cells to satisfy the repair contexts simultaneously. If this is not possible, then it assigns fresh new values to some cells until the remaining repair contexts can be simultaneously satisfied. Intuitively, this may favor repair quality but complicate the resolution of repair contexts with a large number of expressions. In contrast, GetRepair deals with cells one by one, by computing repair context for each cell and finding a value for each cell based on its repair context. We adopt this strategy since we aim for a set of repairs rather than a single one. As will be seen in Section 7, the experimental evaluations well demonstrate the benefits of GetRepair.

5. Diversifying repairs of DC violations

In this section, we present algorithms to generate a set of diversified cardinality-set-minimal repairs.

5.1. Diversification objective

As illustrated in Section 1, in practice we often need to generate multiple repairs instead of one repair. The number of cardinality-set-minimal repairs can be huge. Hence, it is important to generate repairs according to some criterion. In this paper, we focus on *diversification*, aiming at generating repairs to provide good coverage of the different ways to repair DC violations.

Given a set of cardinality-set-minimal repairs, we need to quantify their differences. Restrictions in DCs can be imposed by a range. For example, we can use a value no greater than 500 from the active domain as the new value of $t_\alpha[Salary]$, to resolve the DC violation of $\neg(t_\alpha[Salary] > 500)$. It is usually not very instructive to tell the difference between, *e.g.,* 500 and 499. To this end, we use a measure based on the cells modified in repairs to quantify the difference between repairs. The intuition is that the fewer common cells two repairs share, the more different they are. Recall that GetRepair computes a cardinality-set-minimal repair by modifying exactly all cells in a given minimal vertex cover. Leveraging the algorithm, we can relate the diversification of repairs to the diversification of minimal vertex covers. Formally, we measure the difference between two covers (sets) $\mathcal{C}_1, \mathcal{C}_2$ using the well known Jaccard distance:

$$dis(\mathcal{C}_1, \mathcal{C}_2) = 1 - \frac{|\mathcal{C}_1 \cap \mathcal{C}_2|}{|\mathcal{C}_1 \cup \mathcal{C}_2|}$$

Algorithm 2: SubMVC

```
   Input: Hypergraph G = (V, E) and a number k
   Output: a set S of k minimal vertex covers
1  S ← ∅;
2  C_pre ← ∅;
3  mainProcess(∅, E of G);
4  return S;
5
6  Function mainProcess(C, uncov)
7      if |S| = k then
8          return;
9      if uncov = ∅ then
10         if C_pre ≠ ∅ ∧ dis(C, C_pre) < σ then
11             return;
12         C_pre ← C;
13         S ← S ∪ {C};
14         return;
15     edge ← chooseEdgeToCover(uncov);
16     vertices ← vertices of edge;
17     foreach v ∈ vertices do
18         if checkMinimal(C ∪ {v}) = true then
19             mainProcess(C ∪ {v}, uncov \ {edges containing v});
20             if |S| = k then
21                 return;
```

Algorithm 3: RandomMVC

```
   Input: Hypergraph G = (V, E) and a number k
   Output: a set S of k minimal vertex covers
1  S ← ∅;
2  while |S| < k do
3      backSteps ← 0;
4      mainProcess(∅, E of G);
5  return S;
6
7  Function mainProcess(C, uncov)
8      if |S| = k then
9          return;
10     if uncov = ∅ then
11         if C ∉ S then
12             S ← S ∪ {C};
13             backSteps ← a random value no greater than |C| −
                 1;
14         return;
15     edge ← chooseEdgeToCover(uncov);
16     vertices ← messUp(vertices of edge);
17     foreach v ∈ vertices do
18         if backSteps ≠ 0 then
19             break ;
20         if checkMinimal(C ∪ {v}) = true then
21             mainProcess(C ∪ {v}, uncov \ {edges containing v});
22             if |S| = k then
23                 return;
24     if backSteps ≠ 0 then
25         backSteps ← backSteps − 1;
```

We then formalize the diversification objective. There are many different objectives in the literature; see, *e.g.,* [36–38]. We use one of the most commonly used objectives, referred to as *max–min* diversification function:

$$f(\mathcal{S}) = \min_{\mathcal{C}_i,\mathcal{C}_j \in \mathcal{S}} dis(\mathcal{C}_i, \mathcal{C}_j) \tag{1}$$

where $\mathcal{S}$ is a set of k minimal vertex covers, and the aim is to maximize $f(\mathcal{S})$. It is NP-hard to select $\mathcal{S}$ from a given set U such that the function in Eq. (1) is maximized [39]. Note k is a user-defined parameter [36–38]. For diversifying repairs, a relatively small k can be used to generate some diversified repairs for users to inspect, while a relatively large k can be used to approximate uncertain query answers (recall Section 1). It is worth mentioning that our problem is more difficult than common diversification problem, since the set U, *i.e.,* the set of all minimal vertex covers, is not available. Nevertheless, in the sequel we will develop techniques for the problem that work well in practice.

5.2. Algorithms for diversifying minimal vertex covers

Since the difference between repairs is fully determined by the difference between minimal vertex covers they are produced from, in this subsection we develop techniques to diversify minimal vertex covers.

Our first algorithm to generate a set $\mathcal{S}$ of k diversified minimal vertex covers is based on the enumeration of minimal vertex covers. It is inefficient to enumerate all covers and then apply the diversification function for choosing k covers, since the enumeration can have a result set of exponential size in the worst case [24,25] and k is typically much smaller than the size. We aim to enable early termination as soon as k desirable covers are obtained. Intuitively, depth first search (DFS) is preferable to breadth first search (BFS) in quickly generating some covers, but it suffers from the limitation that the successive covers generated in DFS are usually very similar, *i.e.,* with very few different vertices. To this end, we improve the basic DFS to "skip" similar covers by considering the difference between successive covers.

Algorithm. SubMVC (Algorithm 2) takes the conflict hypergraph $\mathcal{G}$ and a number k as inputs, and outputs k diversified minimal vertex covers. It follows a DFS strategy by recursively calling function *mainProcess*. A hyperedge *edge* is *covered* by a cover $\mathcal{C}$ if at least one of its vertex is in $\mathcal{C}$, and $\mathcal{C}$ becomes a vertex cover if all hyperedges from E of $\mathcal{G}$ are covered by $\mathcal{C}$. In *mainProcess*, a hyperedge *edge* not covered yet is chosen, and vertices from *edge* are enumerated to cover it (lines 15–16). When the set *uncov* becomes empty, a cover is found (line 9). The key is to guarantee that only minimal vertex covers are generated. In function *checkMinimal*, it is checked whether adding v into the current cover $\mathcal{C}$ violates the minimality (line 18). Intuitively, v is added into $\mathcal{C}$ only when v covers uncovered hyperedges from *uncov*. However, v may also cover hyperedges that are already covered by vertices in $\mathcal{C}$, such that some vertices from $\mathcal{C}$ become unnecessary. The technique of *critical hyperedges* [40] is used to check the minimality in our implementation. Roughly speaking, it is to check whether all vertices in $\mathcal{C}$ are still *critical* in the sense that each vertex covers hyperedges that are not covered by any other vertices.

SubMVC differs from common DFS enumeration algorithms in two aspects. (1) SubMVC terminates when k covers are obtained, *i.e.,* $|\mathcal{S}| = k$ (lines 7–8). (2) $\mathcal{C}_{pre}$ is used to save the most recent cover added into $\mathcal{S}$. When a new cover $\mathcal{C}$ is found, we add it into $\mathcal{S}$ only if the difference $dis(\mathcal{C}, \mathcal{C}_{pre})$ is no less than a threshold σ (lines 10–14).

Intuitively, SubMVC may fall short of overall performance since it only considers the difference between two successive covers. To this end, we present another algorithm based on randomness.

Algorithm. RandomMVC (Algorithm 3) also follows a DFS strategy to generate k diversified covers, but differs from SubMVC in the following.

(1) After a minimal vertex cover $\mathcal{C}$ is generated, a random value *backSteps* no greater than the size of $\mathcal{C}$ minus 1 is used to reverse the DFS (line 13, lines 18–19 and lines 24–25). Intuitively, the successive minimal vertex covers generated in DFS

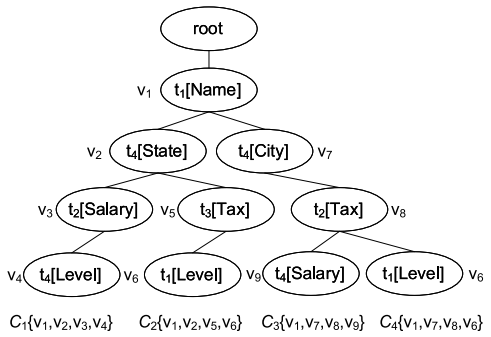

Fig. 4. Example 9 for Algorithm 3.

Algorithm 4: GMM [41]

Input: An integer k and a set U of minimal vertex covers
Output: a set $\mathcal{S}$ of k minimal vertex covers
1 $\mathcal{S} \leftarrow \emptyset$;
2 find $\mathcal{C}$, $\mathcal{C}'$ with the largest $dis(\mathcal{C}, \mathcal{C}')$ from U;
3 $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{C}, \mathcal{C}'\}$;
4 **while** $|\mathcal{S}| \neq k$ **do**
5 $\quad \mathcal{C}'' \leftarrow argmax_{\mathcal{C}'' \in U \setminus \mathcal{S}} min_{\mathcal{C}_i \in \mathcal{S}} dis(\mathcal{C}_i, \mathcal{C}'')$;
6 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{C}''\}$;
7 **return** $\mathcal{S}$;

are similar to each other since they share many common vertices, *i.e.,* the vertices added into the cover following the search path. In order to generate diversified covers, RandomMVC "walks up" *backSteps* + 1 steps instead of a single step of the original DFS. The larger *backSteps* is, the fewer common prefixes (vertices) the two successive covers will share. (2) Before the loop of *mainProcess*, function *messUp* is called to randomly reorder the vertices of *edge* (line 16), to vary the order of dealing with them (line 17).

Example 9 (*Example 4 Continued*)**.** We show in Fig. 4 how RandomMVC generates minimal vertex covers of the conflict hypergraph from Example 4. Let $k = 4$. After the first cover C_1 is generated, suppose we have $backSteps = 1$. Note *backSteps* denotes the additional steps for backtracking. Besides the necessary one step taken by DFS, it additionally goes one step backwards, *i.e.,* from v_3 to v_2, and continues searching from v_2 again. The DFS search and additional backtracking continue until the required four minimal vertex covers are generated. □

Remarks. (1) SubMVC and RandomMVC are necessarily heuristics, due to the high complexity of our problem. It is already NP-hard to select $\mathcal{S}$ from a given set U to maximize diversification, and worse, in our setting the set U of all minimal vertex covers is not available. (2) A small parameter σ in SubMVC helps reduce the running time, possibly at the cost of diversification. We will experimentally study the impact of σ in Section 7. The worst-case complexity of SubMVC is the same as that of the enumeration of minimal vertex covers, since we have to enumerate all minimal vertex covers in the worst case if σ is (too) large. (3) The total number of minimal covers is typically larger than k by several orders of magnitude, and hence, duplicate covers are very unlikely to occur during the process of RandomMVC (line 11 is almost always true). That is, in most cases exactly k minimal covers are computed during RandomMVC. The worst-case complexity for generating one minimal vertex cover with *critical hyperedges* is $O(\|\mathcal{G}\||V|)$ [40], where $\|\mathcal{G}\|$ is the sum of the sizes of hyperedges in $\mathcal{G}$ and $|V|$ is the number of vertices of V.

5.3. Enhanced strategies

In this subsection, we present several strategies that can be combined with the algorithms given in Section 5.2, to trade running time for better diversification.

Strategy one. The first strategy, referred to as BestOfTimes, aims to avoid the bias of random algorithm RandomMVC. Specifically, it runs RandomMVC m times for a set $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$, where $\mathcal{S}_i$ ($i \in [1, m]$) is a set of k minimal vertex covers. It then selects the $\mathcal{S}_i$ with the maximum $f(\mathcal{S}_i)$ (the best diversification).

Strategy two. The second strategy, referred to as Maxmin, first runs RandomMVC for a set U of $m * k$ covers (the same number of covers as BestOfTimes), and then selects k covers from U. For the second step, we employ a well-known 2-approximation algorithm GMM (Algorithm 4), proposed for the max–min facility dispersion problem [41]. Specifically, it first selects two covers $\mathcal{C}$, $\mathcal{C}'$ with the largest $dis(\mathcal{C}, \mathcal{C}')$ from U, and then continues selecting $\mathcal{C}''$ that maximizes the minimal difference until k covers are chosen. In the sequel, we denote by GMM(U, k) the result set $\mathcal{S}$ of k minimal vertex covers found on U.

Strategy three. The third strategy, referred to as IncMaxmin, aims to improve the efficiency of Maxmin. Recall Maxmin computes $m * k$ covers, but there is no guarantee all the computed covers really help improve the diversification. In light of this, after a new cover $\mathcal{C}$ is generated, IncMaxmin checks whether GMM(U, k) is the same as GMM(U', k), where $U' = U \cup \{\mathcal{C}\}$, and decides whether to generate more covers. IncMaxmin continuously monitors the benefit of the new covers in terms of diversification. This enables early termination to improve efficiency.

Different from Maxmin, IncMaxmin calls GMM for every new cover $\mathcal{C}$. We first present an incremental version of GMM, based on a novel data structure.

Maxmin-graph. A maxmin-graph MG$(U, k) = (V, E)$ where V (resp. E) is the set of vertices (resp. edges), is constructed during the process of GMM. Recall that GMM takes as an input a set U of minimal vertex covers, and outputs a set $\mathcal{S}$ of k covers from U. For each cover from $\mathcal{S}$, we add a vertex to V of MG to denote the cover. In the following, we use covers and vertices interchangeably when they are clear from the context.

Specifically, MG is constructed along with $\mathcal{S}$ as follows. (1) When two covers $\mathcal{C}$, $\mathcal{C}'$ with the largest $dis(\mathcal{C}, \mathcal{C}')$ are added into $\mathcal{S}$, we add $\mathcal{C}$, $\mathcal{C}'$ into V, and add an edge $e = (\mathcal{C}, \mathcal{C}')$ into E with a weight $\omega(e) = dis(\mathcal{C}, \mathcal{C}')$. (2) When a cover $\mathcal{C}''$ is added into $\mathcal{S}$, we add $\mathcal{C}''$ into V and an edge $e' = (\mathcal{C}'', \mathcal{C}_1)$ into E, such that $dis(\mathcal{C}_1, \mathcal{C}'') = min_{\mathcal{C} \in \mathcal{S}} dis(\mathcal{C}, \mathcal{C}'')$. We set $\omega(e') = dis(\mathcal{C}_1, \mathcal{C}'')$. This step continues until k covers are added into $\mathcal{S}$.

Vertices in V are numbered according to the order that they are added into V, where v_i denotes the vertex with index i (the order of v_1, v_2 is irrelevant). We denote by e_i the edge (added into E) along with v_{i+1} (added into V), and call e_i the *evidence* of v_{i+1}. Intuitively, e_i justifies the addition of v_{i+1}. In particular, e_1 is the evidence of v_1 and v_2 that are added into V simultaneously.

Example 10. Given $U = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5\}$, $k = 4$ and the distance matrix for covers from U in Fig. 5, we show the construction of maxmin-graph. (1) The first two elements added into $\mathcal{S}$ are $\mathcal{C}_1$ and $\mathcal{C}_5$, for their distance $M_{1,5} = 0.65$. (2) The next element added into $\mathcal{S}$ is $\mathcal{C}_3$. We have $min_{\mathcal{C}_i \in \mathcal{S}} dis(\mathcal{C}_i, \mathcal{C}_3) = 0.25$, while it is 0.05 for $\mathcal{C}_2$ and 0.2 for $\mathcal{C}_4$. (3) Similarly, $\mathcal{C}_4$ is added into $\mathcal{S}$. We have $min_{\mathcal{C}_i \in \mathcal{S}} dis(\mathcal{C}_i, \mathcal{C}_4) = 0.15$, while it is 0.05 for $\mathcal{C}_2$. (4) According to the order, we show indexes of vertices and edges in Fig. 5. □

Algorithm 5: IncGMM

Input: MG(U, k) = (V, E), a new cover $\mathcal{C}$, and GMM(U, k)
Output: A boolean value to indicate whether MG(U, k) and GMM(U, k) need to be updated

1 $m \leftarrow \max_{v_i \in U} dis(v_i, \mathcal{C})$;
2 $pos \leftarrow 0$;
3 **if** $m \leq \omega(e_1)$ **then**
4 $m \leftarrow \min(dis(\mathcal{C}, v_1), dis(\mathcal{C}, v_2))$;
5 $pos \leftarrow 2$;
6 **foreach** $pos < |V|$ **do**
7 **if** $\omega(e_{pos}) < m$ **then**
8 **break** ;
9 $pos \leftarrow pos + 1$;
10 $m \leftarrow \min(m, dis(\mathcal{C}, v_{pos}))$;
11 **if** $pos = |V|$ **then**
12 **if** $|V| = k$ **then**
13 **return** false;
14 **else**
15 complete GMM($U \cup \{\mathcal{C}\}, k$) together with MG($U \cup \{\mathcal{C}\}, k$);
16 **else**
17 remove all vertices v_j ($j > pos$) from V, their related evidences (edges) from E and the covers denoted by them from GMM(U, k);
18 **if** $|V| = 0$ **then**
19 complete GMM($U \cup \{\mathcal{C}\}, k$) together with MG($U \cup \{\mathcal{C}\}, k$);
20 **return** true;

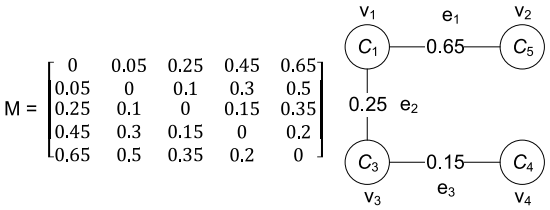

Fig. 5. Example 10 (the construction of maxmin-graph).

Algorithm. IncGMM (Algorithm 5) updates GMM(U, k) in response to a new cover $\mathcal{C}$, by leveraging MG(U, k). It returns a boolean value to indicate whether GMM(U, k) $\neq$ GMM($U \cup \{\mathcal{C}\}, k$).

It first checks whether GMM(U, k) needs to be updated in response to $\mathcal{C}$, by simulating the process of GMM (lines 1–10). It only computes $dis(\mathcal{C}, \mathcal{C}')$ for $\mathcal{C}' \in U$ in this process, by leveraging the *evidences* stored in MG. If $\mathcal{C}$ is found to be "better" than a cover denoted by vertex v_i in MG (but not any vertices v_j such that $j < i$) according to the evidence of v_i, then the verification fails. Intuitively, this implies that the difference between MG(U, k) (resp. GMM(U, k)) and MG($U \cup \{\mathcal{C}\}, k$) (resp. GMM($U \cup \{\mathcal{C}\}, k$)) starts from the position of v_i. We use *pos* to save the largest index of vertex that has been successfully verified. All the vertices v_j (j>pos) and their related evidences (edges) (resp. all the covers denoted by v_j in GMM(U, k)) are removed from MG(U, k) (resp. GMM(U, k)) on line 17. To complete the results of MG($U \cup \{\mathcal{C}\}, k$) and GMM($U \cup \{\mathcal{C}\}, k$), Algorithm GMM and the method to build maxmin-graph are employed in the incremental way: their operations start from the position where the verification fails. To further improve the efficiency, a *lazy evaluation* strategy is adopted. Specifically, the updates are performed only when $\mathcal{C}$ passes all verifications on a *partial* MG (the case of $|V| < k$ on lines 14–15), or when MG is empty (lines 18–19). To complement the strategy, GMM is additionally called for one time if the final GMM(U, k) is incomplete (|GMM(U, k)| $< k$), when IncMaxmin terminates (will be illustrated shortly).

Algorithm 6: Diverse

Input: An instance I of schema R, a set Σ of DCs and a number k
Output: A set X of k diversified cardinality-set-minimal repairs

1 $X \leftarrow \emptyset$;
2 build the conflict hypergraph $\mathcal{G}$ on I *w.r.t.* Σ;
3 generate a set $\mathcal{S}$ of k diversified minimal vertex covers;
4 **foreach** $\mathcal{C} \in \mathcal{S}$ **do**
5 $repair \leftarrow$ GetRepair(I, Σ, $\mathcal{C}$);
6 $X \leftarrow X \cup \{repair\}$;
7 **return** X;

Example 11 (*Example 10 Continued*)**.** Suppose we have a new cover $\mathcal{C}_6$ and $dis(\mathcal{C}_6, \mathcal{C}_i) = 0.05$ for all $\mathcal{C}_i$ from $U = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5\}$. We show the running of IncGMM as follows. (1) $v_1(\mathcal{C}_1)$ and $v_2(\mathcal{C}_5)$ pass the verification because $0.05 < 0.65$. (2) $v_3(\mathcal{C}_3)$ and v_4 ($\mathcal{C}_4$) also pass the verification because $0.05 < 0.25$ and $0.05 < 0.15$ respectively. We have another cover $\mathcal{C}_7$, where $dis(\mathcal{C}_7, \mathcal{C}_1) = dis(\mathcal{C}_7, \mathcal{C}_5) = 0.3$, $dis(\mathcal{C}_7, \mathcal{C}_2) = dis(\mathcal{C}_7, \mathcal{C}_3) = dis(\mathcal{C}_7, \mathcal{C}_4) = dis(\mathcal{C}_7, \mathcal{C}_6) = 0.05$. We see $v_1(\mathcal{C}_1)$ and $v_2(\mathcal{C}_5)$ pass the verification because $0.3 < 0.65$, while $v_3(\mathcal{C}_3)$ fails because $\min_{v_i \in \{v_1, v_2\}} dis(v_i, \mathcal{C}_7) = 0.3$ and $0.3 > 0.25$. Thus, only the first two vertices of MG will be kept. □

Strategy IncMaxmin. We present IncMaxmin. (1) It first employs RandomMVC to generate k minimal vertex covers, add all of them into U, and compute GMM(U, k) and MG(U, k). (2) It then employs RandomMVC to generate more covers. For each generated cover $\mathcal{C}'$, it calls IncGMM to compute GMM($U \cup \{\mathcal{C}'\}, k$) and MG($U \cup \{\mathcal{C}'\}, k$), and sets $U = U \cup \{\mathcal{C}'\}$. (3) It completes GMM($U, k$) if GMM($U, k$) is incomplete and then terminates, when IncGMM returns false p times continuously, *i.e.,* the successive p covers do not help improve the diversification. Herein, p is a parameter, and we set $p = 5$ in our experimental evaluations. We experimentally find IncMaxmin typically generates much fewer covers than Maxmin (Section 7).

5.4. Algorithm for diversifying repairs of DC violations

We put together methods for diversified minimal vertex covers and the method to compute a cardinality-set-minimal repair from a given minimal vertex cover.

Algorithm. Diverse (Algorithm 6) generates k diversified cardinality-set-minimal repairs. After building the conflict hypergraph $\mathcal{G}$, it obtains k diversified minimal vertex covers by employing IncMaxmin (or BestOfTimes, Maxmin) (lines 2–3). It then employs GetRepair to compute a cardinality-set-minimal repair based on every cover (lines 4–6).

6. Optimizations

In this section, we present some optimizations to further improve our approach.

Hypergraph Compression. The conflict hypergraph $\mathcal{G}$ can be very large, leading to an extremely large number of minimal vertex covers. Obviously, not all of the covers are equally important. Intuitively, cells involved in more DC violations are more likely to be erroneous and are hence modified in former DC repairing methods [4,11] for nearly optimum repairs, while cells only involved in very few violations are likely to be correct. In light of this, we develop techniques to compress the hypergraph. This reduces the search space of minimal vertex covers by avoiding covers with vertices concerning very few DC violations and hence

Table 2
Extension of Table 1.

TID	Name	Phone	Level	DepID	Salary	Tax	City	State
t_8	H. Lucy	230–436	3	A550	30000	2500	Los Angeles	CA
t_9	L. David	320–835	3	A550	30000	2500	Los Angeles	CA
t_{10}	T. Mike	333–264	2	A005	7000	0	Lima	NY

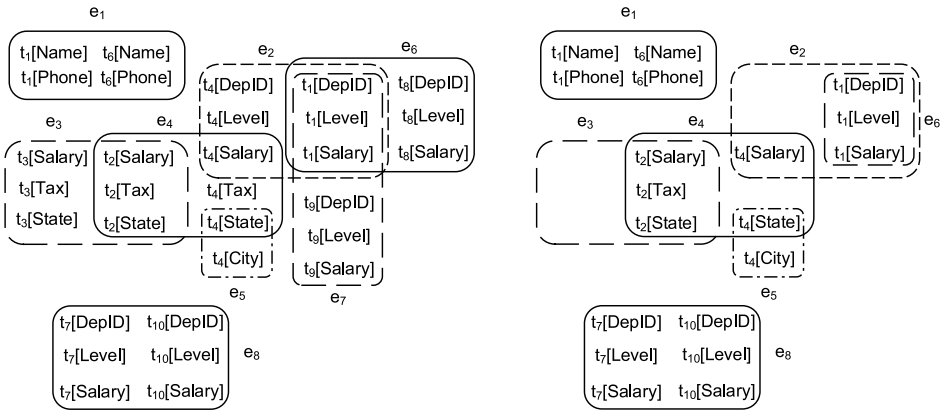

Fig. 6. Conflict hypergraphs of Example 12.

improves the efficiency. The method is necessarily heuristics, since it is beyond reach to only generate minimum vertex covers.

The degree of a vertex in the conflict hypergraph $\mathcal{G}$ denotes the number of hyperedges sharing the vertex, *i.e.,* DC violations concerning the cell. We deal with $\mathcal{G}$ as follows. (1) For every $\varphi \in \Sigma$, identify the subgraph $\mathcal{G}_\varphi$ of $\mathcal{G}$ that contains only hyperedges concerning violations of φ. On each disjoint component of $\mathcal{G}_\varphi$, mark every vertex unless its degree in the component is the smallest among all the vertices; mark all the vertices if they have the same degree in the component. (2) From the unmarked vertices after (1), mark those with the largest degree in $\mathcal{G}$ among all the unmarked ones; do nothing if all the unmarked vertices have the same degree in $\mathcal{G}$. (3) Remove the unmarked vertices from $\mathcal{G}$, and merge hyperedges with the same set of vertices after that. For each hyperedge from $\mathcal{G}$, it can be verified that as least one vertex of the hyperedge is retained. Thus, it still suffices to generate repairs from the compressed hypergraph. Intuitively, we aim to discard the vertices contained in very few hyperedges.

Example 12. To illustrate our method, we additionally add three tuples to Table 1, as shown in Table 2. We show the conflict hypergraph $\mathcal{G}$ in Fig. 6(a) for all the tuples. We compress the hypergraph as follows. (1) There are four hyperedges e_2, e_6, e_7 and e_8 concerning φ_2, which form the subgraph $\mathcal{G}_{\varphi_2}$. Further, $\mathcal{G}_{\varphi_2}$ has two disjoint components, one with e_2, e_6 and e_7, and the other with e_8. In the first component, t_1[DepID], t_1[Level] and t_1[Salary] have a degree of 3, while the other vertices have a degree of 1. Thus, we mark the vertices with a degree of 3. In the second component, we mark all the vertices because they have the same degree. After dealing with all DCs from Σ, we mark t_4[State], t_4[City] and all the vertices from t_1, t_2, t_6, t_7 and t_{10}. (2) We mark t_4[Salary], because its degree in $\mathcal{G}$ is the largest among all the unmarked vertices. (3) We remove all the unmarked vertices, and merge e_6 and e_7. (4) Finally, we show the compressed conflict hypergraph in Fig. 6(b). □

Ordering Cells. Given a minimal vertex cover, GetRepair (Section 4) repairs cells from the cover one by one. It guarantees to generate a cardinality-set-minimal repair, regardless of the order to repair the cells. However, it may assign different values to the same cell c when cells are treated in different orders. The reason is that the repair context of the cell c is affected by the values of the cells modified before c. Along the same lines as optimum repair computations [4,11], we prefer to first repair cells that are more likely to be erroneous. To this end, we heuristically determine an order $\mathcal{O}$ for cells in the conflict hypergraph $\mathcal{G}$. Specifically, in $\mathcal{O}$ vertices are ordered in a descending order of their degrees. To break ties, for vertices v_i, v_j with the same degree, v_i is before v_j if $i < j$, where i, j are random integer *ids* assigned to vertices in $\mathcal{G}$. We improve GetRepair by repairing cells one by one in the order of $\mathcal{O}$.

7. Experiments

In this section, we conduct extensive experiments to verify the effectiveness and efficiency of our repair diversification techniques.

7.1. Experimental settings

Datasets. We use two datasets that are employed to evaluate DC repairing methods in former works [4,8,10,11]. (a) *Hospital* is a real-life dataset from the US Department of Health & Human Services. It has 50K tuples and 15 attributes, and mainly contains categorical data. (b) *Tax* is a synthetic relational table with 50K tuples and 15 attributes, containing both categorical and numerical data. Along the same settings as [4,11], we define 4 DCs on each dataset. The DCs on *Hospital* mainly concern equality comparisons, *i.e.,* "=", while those on *Tax* contain more ordering comparisons, *i.e.,* "<" and ">".

Error introduction. We use BART, an error-generation tool [42], to introduce errors into the datasets. Compared with the ad-hoc strategies adopted in [4,11], it is shown in [42] BART allows the introduction of errors in a controlled way. Specifically, it can guarantee all the introduced errors are *detectable* using the given DCs, *i.e.,* the introduced errors always lead to DC violations. Along the same lines as [42], we define the error rate as the ratio of the number of error cells to the number of tuples in the dataset.

Algorithms. We implement all the algorithms in Java. (1) Our minimal vertex cover diversification strategies BestOfTimes,

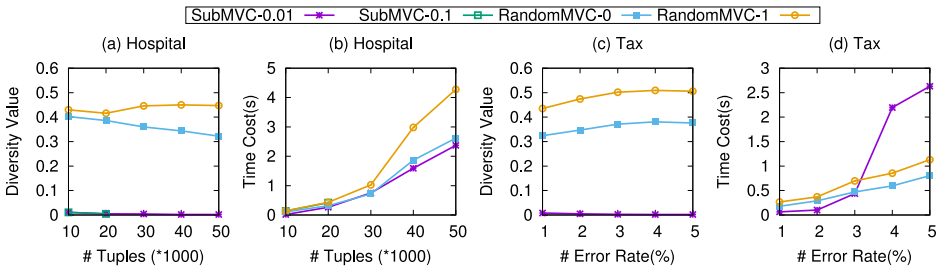

Fig. 7. SubMVC against RandomMVC for diversifying minimal vertex covers.

Maxmin and IncMaxmin, together with the methods SubMVC and RandomMVC. (2) Our repair diversification method Diverse. (3) Some variants of our methods for testing optimization techniques (the details will be illustrated shortly). (4) Holistic [4] and Vfree [11], the state-of-the-art repairing methods leveraging DCs.

All experiments are conducted on a Win11 machine with 3.6 GHz CPU and 16 GB RAM. Gurobi Optimizer 8.0.1 has been used as the external Quadratic Programming (QP) tool to solve the system of inequalities in the repair context when necessary [4,11].

7.2. Experimental results

We report our experimental findings in detail.

7.2.1. Diversifying minimal vertex covers

We first experimentally study the algorithms and enhanced strategies given in Section 5, for diversifying minimal vertex covers.

Exp-1. In this set of experiments, we employ SubMVC and RandomMVC to generate 100 minimal vertex covers, respectively. We report the diversification values (Eq. (1) in Section 5.1) and running times of the methods.

Recall a threshold σ is used to set the lower bound of the difference between two successive covers in SubMVC. A small σ enables SubMVC to generate k covers more efficiently, but may harm the diversification. We consider $\sigma = 0.01$ or 0.1 in this set of experiments. Recall we use a parameter *backSteps* to set the additional backtracking steps in RandomMVC. We consider two cases: RandomMVC-0 uses a random value no greater than the size of the cover minus 1, while RandomMVC-1 forces the DFS to start from the root for each cover.

(1) In Figs. 7a–b, we vary the number of tuples and fix an error rate of 2% on *Hospital*. We see the following. (a) For SubMVC, the diversification values are always low with $\sigma = 0.01$. This is because the successive covers generated in SubMVC are too similar. A larger σ is expected to help overcome the limitation. However, SubMVC-0.1 is too slow to terminate within 30 min when the number of tuples is larger than 20K. Indeed, we find SubMVC-0.1 has to discard tens of thousands of covers before obtaining a cover satisfying the requirement of σ. That is, SubMVC falls short of either scalability or effectiveness. (b) Compared with RandomMVC-0, RandomMVC-1 leads to better diversification at the cost of slightly more time, as expected. Note that RandomMVC-1 is actually very efficient and always terminates within 5 s in this set of experiments. RandomMVC-1 is on average 20% better than RandomMVC-0 in diversification. As the number of tuples increases, the diversification value of RandomMVC-0 decreases, while that of RandomMVC-1 remains stable.

(2) In Figs. 7c–d, we vary the error rate and fix the number of tuples at 20K on dataset *Tax*. We see the following. (a) The performance of SubMVC is still very poor, and SubMVC-0.1 fails to terminate within 30 min for every error rate. (b) The effectiveness of RandomMVC is not affected as the error rate increases. (c) All algorithms cost more time when the error rate increases, since the conflict hypergraph becomes larger. SubMVC-0.01 becomes slower than RandomMVC when the error rate is larger than 3%.

To conclude, we find that RandomMVC-1 is always the best in diversification, and is also very efficient.

Exp-2. In this set of experiments, we combine RandomMVC-1 with the strategies presented in Section 5.3, to experimentally study different strategies. We aim to obtain 100 diversified minimal vertex covers. Specifically, (a) in BestOfTimes, we run RandomMVC 10 times, generate 100 covers each time, and select the best result among the 10 runs. (b) In Maxmin, we generate 1,000 covers with RandomMVC, and employ GMM to choose 100 covers from them. (c) In IncMaxmin, we first generate 100 covers, and continue generating more covers until the termination is judged by IncGMM. For comparison, we also employ RandomMVC to directly generate 100 covers.

(1) In Figs. 8a–b, we vary the number of tuples and fix an error rate of 2% on *Hospital*. We see the following. (a) The strategies really help improve diversification. Compared with the basic RandomMVC, the diversification values on average improve by 3%, 12% and 10% for BestOfTimes, Maxmin and IncMaxmin, respectively. (b) We can obtain a similar diversification result with IncMaxmin as Maxmin (the gap is usually within 2%). (c) Maxmin takes slightly more time than BestOfTimes; they generate the same number of covers, and Maxmin additionally calls GMM to choose covers. (d) The time of IncMaxmin is about 25% of those of BestOfTimes and Maxmin. We find IncMaxmin generates only 200–300 covers (not shown), and the additional cost of calling IncGMM is not significant due to the high efficiency of incremental computations.

(2) We then use *Tax* of 20K tuples, vary error rates and report the results in Figs. 8c–d. The results confirm our observations on *Hospital*. The diversification values on average improve by about 3%, 11% and 10% for BestOfTimes, Maxmin and IncMaxmin, respectively, compared with RandomMVC.

In conclusion, we find that IncMaxmin strikes a better balance between effectiveness and efficiency.

Exp-3. In this set of experiments, we compare IncGMM against GMM in detail. We aim to obtain 200 diversified minimal vertex covers with IncMaxmin. Specifically, leveraging RandomMVC, we first generate 200 covers and then continue generating more covers. For each new cover, we call IncGMM (resp. GMM) to update GMM(U, k) and MG(U, k) (resp. GMM(U, k)) to test the termination condition. The application of IncGMM or GMM affects the efficiency but not the termination condition (the same number of covers are generated in both settings). We report the total time of running IncGMM (resp. GMM).

We vary the number of tuples and fix an error rate of 2% on *Hospital*, as shown in Fig. 9a. IncGMM is about 4 or 5 times faster than GMM. In Fig. 9b, we vary the number of tuples and fix an error rate of 2% on *Tax*. Although the gap between the two methods decreases on *Tax*, IncGMM is still about 3 times faster

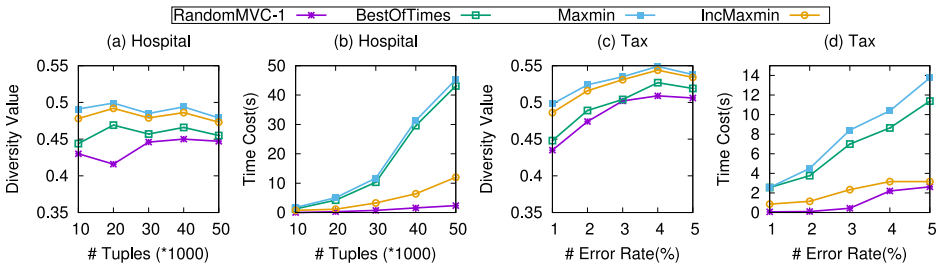

Fig. 8. Comparison among different enhanced strategies.

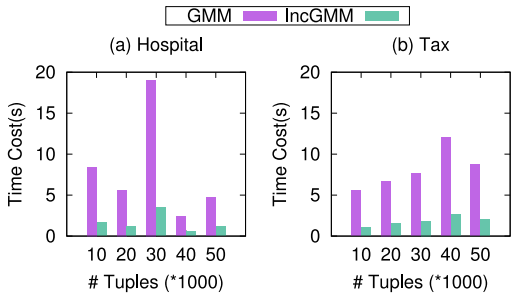

Fig. 9. GMM against IncGMM.

than GMM. IncGMM is much more efficient than GMM due to the incremental computation and lazy evaluation strategy. Besides the fewer number of covers generated, we conclude that IncGMM is critical to the efficiency of IncMaxmin.

Exp-4. In this set of experiments, we evaluate the effectiveness of hypergraph compression (Section 6). In Maxmin, we first generate 200 covers of the original hypergraph (resp. the compressed hypergraph), from which 20 covers are then selected. We use Maxmin here for a fair comparison. It is guaranteed that the same number of covers are generated in Maxmin (but not in IncMaxmin) with or without hypergraph compression. We report the sizes (the numbers of vertices) of the covers, total running time and diversification value.

(1) In Figs. 10a–c, we vary the number of tuples and fix an error rate of 2% on *Hospital*. We see the following. (a) In Fig. 10a, we show the size of every cover obtained on the hypergraph with (resp. without) compression. With hypergraph compression, we obtain covers of small sizes and most covers have similar sizes. In contrast, the sizes of the 20 covers of the original conflict hypergraph can be much larger and vary significantly. As will be seen in Exp-5, the sizes of covers may heavily affect the quality of their corresponding repairs. (b) It takes less time to obtain diversified covers on the compressed hypergraph, as expected. For example, the time reduces by about 20% on *Hospital* with 50K tuples. (c) The diversification value on average slightly decreases by about 8% on the compressed hypergraph, since fewer vertices in the compressed hypergraph imply fewer chances of diversification.

(2) We test dataset *Tax* by varying the error rate and fixing a size of 20K, as shown in Fig. 11. With hypergraph compression, we find the diversification value decreases by about 7%, but the time reduces by half on average.

7.2.2. Repair quality and scalability

Finally, we compare our algorithm Diverse against DC repairing algorithms Holistic [4] and Vfree [11] that compute a single nearly optimum repair. Diverse first employs IncMaxmin to obtain diversified minimal vertex covers on the compressed hypergraph, and then employs GetRepair to repairs cells in the order of $\mathcal{O}$ (Section 6) to generate a set of diversified repairs.

Our aim is to show that in reasonable time, Diverse may well generate repairs that are the same as or better than the repairs produced by Holistic and Vfree in terms of repair quality. This is possible, since (a) DC repairing is NP-hard and heuristic approaches Holistic and Vfree cannot guarantee repair quality; and (b) there may be many repairs that have the same or similar repair quality.

We use the same criteria as [4,11,21] to evaluate repair quality. Specifically, *precision* (P) is the ratio of the number of *corrected changed cells* to the number of all changed cells in repairing methods, *recall* (R) is the ratio of the number of *corrected changed cells* to the number of all error cells, and *F-measure* (F) = $2 \times (P \times R)/(P+R)$. To count the number of *corrected changed cells*, we use 1 (resp. 0.5) if an error cell is updated with the original correct value (resp. is modified but with a value different from the correct one).

Exp-5. In this set of experiments, we use Diverse to obtain a set of 100 diversified repairs and use Holistic and Vfree to compute one repair, respectively. We compare the number of changed (modified) cells, precision, F-measure and running time of different methods. For Diverse, we report the average time for one repair, *i.e.,* the time of Diverse divided by 100.

(1) In Figs. 12a–d, We fix an error rate of 2% and vary the number of tuples on *Hospital*. We see the following. (a) The numbers of changed cells of Vfree and Diverse are significantly smaller than that of Holistic. This is because Holistic may introduce new violations in the repairing process. The 100 repairs of Diverse usually have very similar numbers of changed cells, due to the usage of the compressed hypergraph (recall the sizes of covers shown in Exp-4). (b) The precision and F-measure of Holistic are lower than those of Vfree and Diverse. We find Holistic has low precision and hence low F-measure, because it modifies much more cells than the other methods. Among the diversified repairs of Diverse, there are always some repairs that have better quality than those of Vfree and Holistic. We are very likely to obtain covers of the compressed hypergraph that cover more error cells than the single cover in Vfree (resp. Holistic). (c) The average time of one repair of Diverse is nearly two times (resp. two orders of magnitude) faster than the time of Holistic (resp. Vfree). Holistic may need several rounds to terminate, for handling the new DC violations introduced. Vfree tries to repair all related cells together, but the repair contexts may become very complicated and hence costly to be resolved. In contrast, Diverse repairs cells one by one. This makes the repair context small and the value assignment fast. Moreover, Diverse can share some computations among multiple repairs, *e.g.,* the building of the conflict hypergraph.

(2) In Figs. 12e–h, we vary error rates and fix the number of tuples at 20K on dataset *Hospital*. We see similar results as those shown in (1).

(3) In Figs. 13a–d, we vary the number of tuples and fix an error rate of 2% on *Tax*. We see the following. (a) Holistic incurs more cell modifications than the other methods, although the

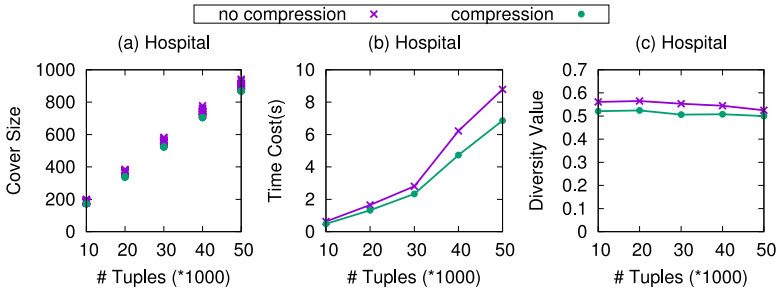

Fig. 10. The effectiveness of hypergraph compression on *Hospital*.

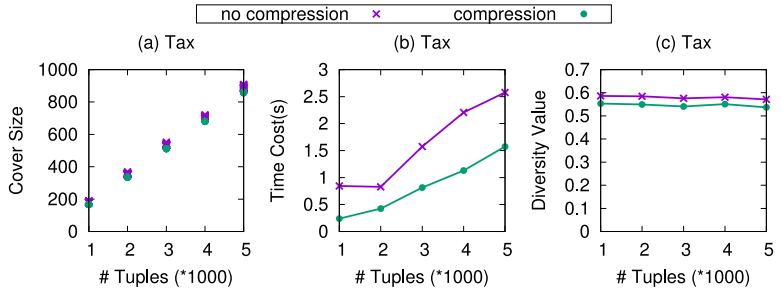

Fig. 11. The effectiveness of hypergraph compression on *Tax*.

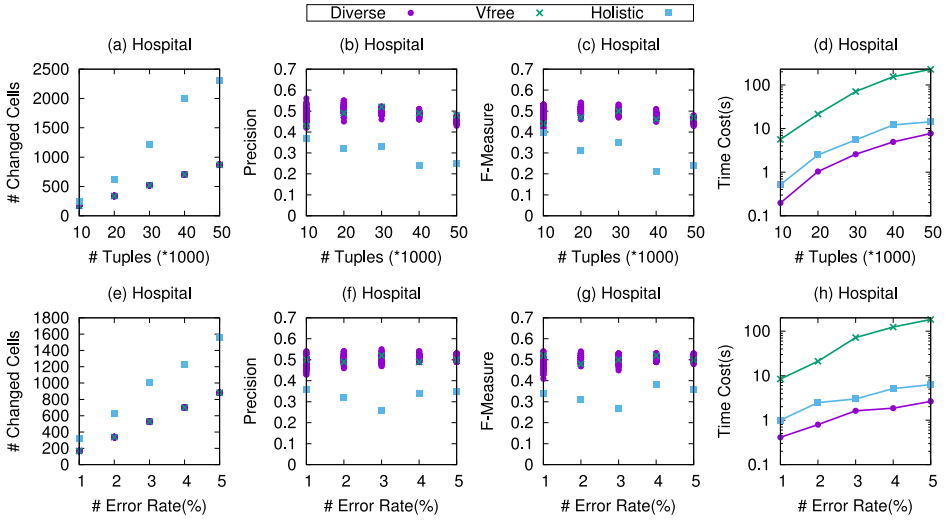

Fig. 12. Repair quality and running time on *Hospital*: Diverse against Holistic and Vfree.

gap decreases compared to that on *Hospital*. (b) Diverse can find repairs with better quality than the other methods. (c) The average time of generating one repair in Diverse is much faster than the times of Holistic and Vfree. This is because value assignments concerning order comparisons are more difficult to be resolved when complicated repair contexts occur in Holistic and Vfree.

(4) In Figs. 13e–h, we vary the error rate and fix the number of tuples at 20K on *Tax*. The results confirm our observations on *Hospital*.

8. Conclusion

We make the first effort to study diversifying repairs of DC violations. We have established the connection between cardinality-set-minimal repairs and minimal vertex covers of the conflict hypergraph, developed a set of algorithms to diversify minimal vertex covers, given optimizations to improve effectiveness and (or) efficiency of our approach, and conducted extensive experiments to verify our methods on both real-life and synthetic data.

There are a host of diversification objectives in the literature, *e.g.*, [36–38], which are suitable to different applications. We intend to adapt more diversification objectives to DC repairs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by National Key R&D Program of China 2018YFB1700403 and National Natural Science Foundation of China 62172102, 61572135, 61925203. All authors approved the version of the manuscript to be published.

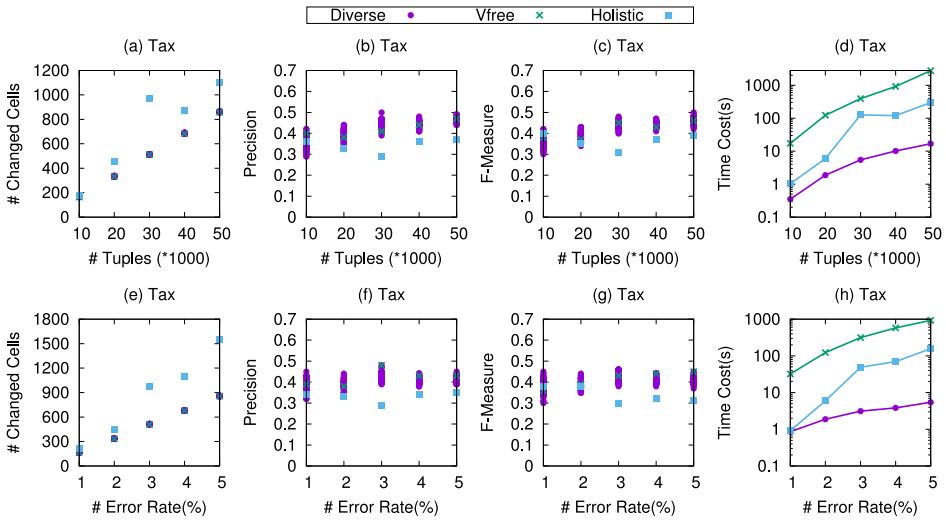

Fig. 13. Repair quality and running time on *Tax*: Diverse against Holistic and Vfree.

References

[1] W. Fan, F. Geerts, Foundations of Data Quality Management, in: Synthesis Lectures on Data Management, Morgan & Claypool Publishers, San Rafael, 2012.
[2] I.F. Ilyas, X. Chu, Data Cleaning, ACM, New York City, 2019.
[3] X. Chu, I.F. Ilyas, P. Papotti, Discovering denial constraints, in: PVLDB, Vol. 6, 2013, pp. 1498–1509.
[4] X. Chu, I.F. Ilyas, P. Papotti, Holistic data cleaning: Putting violations into context, in: ICDE, 2013, pp. 458–469.
[5] W. Fan, F. Geerts, X. Jia, A. Kementsietsidis, Conditional functional dependencies for capturing data inconsistencies, ACM Trans. Database Syst. 33 (2008) 6:1–6:48.
[6] S. Ginsburg, R. Hull, Order dependency in the relational model, Theor. Comput. Sci. 26 (1983) 149–195.
[7] S. Ginsburg, R. Hull, Sort sets in the relational model, J. ACM 33 (1986) 465–488.
[8] S. Giannakopoulou, M. Karpathiotakis, A. Ailamaki, Cleaning denial constraint violations through relaxation, in: SIGMOD, 2020, pp. 805–815.
[9] A. Gilad, D. Deutch, S. Roy, On multiple semantics for declarative database repairs, in: SIGMOD, 2020, pp. 817–831.
[10] T. Rekatsinas, X. Chu, I.F. Ilyas, C. Ré, Holoclean: Holistic data repairs with probabilistic inference, Proc. VLDB Endow. 10 (2017) 1190–1201.
[11] S. Song, H. Zhu, J. Wang, Constraint-variance tolerant data repairing, in: SIGMOD, 2016, pp. 877–892.
[12] P. Bohannon, M. Flaster, W. Fan, R. Rastogi, A cost-based model and effective heuristic for repairing constraints by value modification, in: SIGMOD, 2005, pp. 143–154.
[13] G. Cong, W. Fan, F. Geerts, X. Jia, S. Ma, Improving data quality: Consistency and accuracy, in: VLDB, 2007, pp. 315–326.
[14] S. Hao, N. Tang, G. Li, J. He, N. Ta, J. Feng, A novel cost-based model for data repairing, IEEE Trans. Knowl. Data Eng. 29 (2017) 727–742.
[15] S. Kolahi, L.V.S. Lakshmanan, On approximating optimum repairs for functional dependency violations, in: ICDT, 2009, pp. 53–62.
[16] J. Wang, N. Tang, Dependable data repairing with fixing rules, ACM J. Data Inf. Qual. 8 (2017) 16:1–16:34.
[17] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, N. Tang, Interactive and deterministic data cleaning, in: SIGMOD, 2016, pp. 893–907.
[18] M. Musleh, M. Ouzzani, N. Tang, A. Doan, Coclean: Collaborative data cleaning, in: SIGMOD, 2020, pp. 2757–2760.
[19] S. Thirumuruganathan, L. BertiÉquille, M. Ouzzani, J. Quiané-Ruiz, N. Tang, Uguide: User-guided discovery of fd-detectable errors, in: SIGMOD, 2017, pp. 1385–1397.
[20] M. Yakout, A.K. Elmagarmid, J. Neville, M. Ouzzani, I.F. Ilyas, Guided data repair, Proc. VLDB Endow. 4 (2011) 279–289.
[21] G. Beskales, I.F. Ilyas, L. Golab, Sampling the repairs of functional dependency violations under hard constraints, Proc. VLDB Endow. 3 (2010) 197–207.
[22] G. Beskales, I.F. Ilyas, L. Golab, A. Galiullin, Sampling from repairs of conditional functional dependency violations, VLDB J. 23 (2014) 103–128.
[23] R. Jampani, F. Xu, M. Wu, L.L. Perez, C.M. Jermaine, P.J. Haas, MCDB: A monte carlo approach to managing uncertain data, in: SIGMOD, 2008, pp. 687–700.
[24] Gainer-Dewar A., P. Vera-Licona, The minimal hitting set generation problem: Algorithms and computation, SIAM J. Discret. Math. 31 (2017) 63–100.
[25] L. Lin, Y. Jiang, The computation of hitting sets: Review and new algorithms, Inf. Process. Lett. 86 (2003) 177–184.
[26] I.F. Ilyas, X. Chu, Trends in cleaning relational data: Consistency and deduplication, Found. Trends Databases 5 (2015) 281–393.
[27] L.E. Bertossi, L. Bravo, E. Franconi, A. Lopatenko, The complexity and approximation of fixing numerical attributes in databases under integrity constraints, Inf. Syst. 33 (2008) 407–434.
[28] A. Lopatenko, L. Bravo, Efficient approximation algorithms for repairing inconsistent databases, in: ICDE, 2007, pp. 216–225.
[29] C. Ye, H. Wang, K. Zheng, J. Gao, J. Li, Multi-source data repairing powered by integrity constraints and source reliability, Inf. Sci. 507 (2020) 386–403.
[30] M. Dallachiesa, A. Ebaid, A. Eldawy, A.K. Elmagarmid, I.F. Ilyas, M. Ouzzani, N. Tang, NADEEF: A commodity data cleaning system, in: SIGMOD, 2013, pp. 541–552.
[31] F. Geerts, G. Mecca, P. Papotti, D. Santoro, Cleaning data with llunatic, VLDB J. 29 (2020) 867–892.
[32] Z. Khayyat, I.F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Yin, Bigdansing: A system for big data cleansing, in: SIGMOD, 2015, pp. 1215–1230.
[33] C. He, Z. Tan, Q. Chen, C. Sha, Z. Wang, W. Wang, Repair diversification for functional dependency violations, in: DASFAA 2014, 2014, pp. 468–482.
[34] M. Arenas, L.E. Bertossi, J. Chomicki, Consistent query answers in inconsistent databases, in: PODS, 1999, pp. 68–79.
[35] V.V. Vazirani, Approximation Algorithms, Springer, Heidelberg, 2001.
[36] M. Drosou, E. Pitoura, Search result diversification, SIGMOD Rec. 39 (2010) 41–47.
[37] X. Ge, P.K. Chrysanthis, Prefdiv: Efficient algorithms for effective top-k result diversification, in: EDBT, 2020, pp. 335–346.
[38] S. Gollapudi, A. Sharma, An axiomatic approach for result diversification, in: WWW, 2009, pp. 381–390.
[39] S.S. Ravi, D.J. Rosenkrantz, G.K. Tayi, Approximation algorithms for facility dispersion, in: T.F. Gonzalez (Ed.), Handbook of Approximation Algorithms and Metaheuristics, second ed., in: Volume 2: Contemporary and Emerging Applications, Chapman and Hall/CRC, New York, 2018.
[40] K. Murakami, T. Uno, Efficient algorithms for dualizing large-scale hypergraphs, Discret. Appl. Math. 170 (2014) 83–94.
[41] S.S. Ravi, D.J. Rosenkrantz, G.K. Tayi, Heuristic and special case algorithms for dispersion problems, Oper. Res. 42 (1994) 299–310.
[42] Arocena P.C., B. Glavic, G. Mecca, R.J. Miller, P. Papotti, D. Santoro, Messing up with bart: error generation for evaluating data-cleaning algorithms, in: Proceedings of the VLDB Endowment, 2015, pp. 36–47.