Efficient Approximation Algorithms for Repairing Inconsistent Databases

Andrei Lopatenko
Google &
Free University of Bozen–Bolzano
lopatenko@google.com

Loreto Bravo
Carleton University &
University of Edinburgh
lbravo@scs.carleton.ca

Abstract

We consider the problem of repairing a database that is inconsistent wrt a set of integrity constraints by updating numerical values. In particular, we concentrate on denial integrity constraints with numerical built-in predicates. So far research in this context has concentrated in computational complexity analysis. In this paper we focus on efficient approximation algorithms to obtain a database repair and we present an algorithm that runs in $O(n \log n)$ wrt the size of the database. Our experimental evaluations show that even for large databases an approximate repair of the database can be computed efficiently despite the fact that the exact problem is computationally intractable. Finally, we show that our results can also be applied to database repairs obtained by a minimal number of tuple deletions.

1. Introduction

When merging different database instances in a data exchange setting, the resulting database may be inconsistent wrt a set of integrity constraints. Two techniques have been used to deal with inconsistent databases: data cleaning [19, 17] and consistent query answering (CQA) [3]. While data cleaning results in a new consistent database, CQA keeps the inconsistent database as is and considers the ICs at query time to return only the so called "consistent answers". In [4] techniques from CQA were modified to perform data cleaning, i.e. for finding a unique repair of the database. In this paper we aim at providing optimized algorithms for those modified techniques.

A repair of a database can be obtained by deletions and insertions of whole tuples as well as by updating attributes, generally under the assumption that the repair should be "as close as possible" to the inconsistent database. This closeness can be interpreted in many different ways such as minimal number of changes, minimal set of changes under set inclusion and minimal numerical distance (refer to [3] for a survey and references of notions considered in CQA).

Example 1.1 Consider a database that classifies different types of papers and their environmental "friendliness". Each tuple has the ID of the paper, if it is environmentally friendly (EF), the percentage of Recycled Content (PRC) and if the bleaching process was Chlorine Free (CF). EF and CF can take values 1 or 0 and PCR is a positive value smaller than 100. A paper is environmentally friendly, i.e. $EF = 1$, only if the percentage of recycled content is higher or equal to 50% and the bleaching process is chlorine free. The following database is inconsistent wrt this constraint. We add a column in order to label the different tuples.

D:

Paper	*ID*	*EF*	*PRC*	*CF*	
	B1	1	40	0	t_1
	C2	1	20	1	t_2
	E3	1	70	1	t_3

Tuples t_1 and t_2 do not satisfy the constraint. A repair obtained by deleting a minimal number of tuples, would leave only t_3 in the database. An alternative approach is to modify the values of some attributes in a minimal way in order to restore consistency. For example, we can solve the inconsistency of the second tuple by changing the value of EF from 1 to 0. Less information is lost when we update attributes. □

Attribute-update repairs, i.e repairs obtained by updating numerical values that minimize the numerical distance from the original database have been considered before in [21, 22, 4, 6]. This approach is of particular interest for database applications, like census, demographic, financial, and experimental data, that contain quantitative data usually associated to nominal or

1-4244-0803-2/07/$20.00 ©2007 IEEE.

qualitative data, e.g. number of children associated to a household identification code (or address); or measurements associated to a sample identification code. This databases may contain semantic constraints. For example, a census form for a particular household may be considered incorrect if the number of children exceeds 20; or if the age of a parent is less than 10. These restrictions can be expressed with denial integrity constraints, that prevent some attributes from taking certain combinations of values [11]. Inconsistencies in numerical data can be resolved by changing individual attribute values, while keeping values in the keys, e.g. without changing the household code, the number of children is decreased considering the admissible values.

The optimization problem of finding the minimum distance from an attribute-update repair wrt *IC* to a given input instance is *MAXSNP*-hard [4]. By transforming this problem into a minimum weight set cover problem, a repair can be calculated by using approximation algorithms such as the greedy [10] and layer algorithms [13, Chapter 3].

Since the running time of the greedy algorithm is $O(n^3)$, it is not efficient enough for real databases where the database can have one million or more tuples. In this paper, we introduce a variation of the greedy algorithm that runs in $O(n^2 \log n)$ in the general case and in $O(n \log n)$ under the assumption that a tuple can be involved in a number of inconsistencies that is bounded by a constant. This assumption is valid in most practical cases where a tuple is generally not involved in many inconsistencies. We experiment with – and compare the performance of – the greedy and layer algorithm and their modified versions. We show that the modified greedy algorithm runs faster than the greedy algorithm, and that in its turn, the greedy algorithm runs faster than the layer and modified layer algorithm. We also show that the approximations obtained with the greedy algorithms are better than the ones of the layer algorithms.

Finally, we also consider repairs obtained by a minimal number of deletions, i.e. tuple-deletion repairs, and we show that they can also be obtained as attribute-update repairs. This implies that we can also use the modified-greedy algorithm to compute tuple-deletion repairs.

The paper is structured as follows. Section 2 introduces definitions. Sections 3 describes the transformation of the problem of obtaining attribute-update repairs to the MWSCP, introduces a modified version of the MWSC greedy algorithm and gives a running time analysis of it. Section 4 shows some experimental results that compare the performance of the greedy and layer algorithm and their modified versions. Section 5 shows how to transform the tuple-deletion repair problem into the attribute-update repair problem. Finally Section 6 presents some conclusions.

2. Preliminaries

Consider a relational schema $\Sigma = (\mathcal{U}, \mathcal{R} \cup \mathcal{B}, \mathcal{A})$, with domain $\mathcal{U}$ that includes $\mathbb{Z}$,[1] $\mathcal{R}$ a set of database predicates, $\mathcal{B}$ a set of built-in predicates, and $\mathcal{A}$ a set of attributes. For $R \in \mathcal{R}$, $\mathcal{A}_R$ is a subset of $\mathcal{A}$ associated to R. A database instance is a finite collection D of *database tuples*, i.e. of ground atoms $P(\bar{c})$, with $P \in \mathcal{R}$ and $\bar{c}$ a tuple of constants in $\mathcal{U}$ such that $|\bar{c}| = |\mathcal{A}_R|$. There is a set $\mathcal{F} \subseteq \mathcal{A}$ of all the *flexible* attributes, those that take values in $\mathbb{Z}$ and are allowed to be modified. Attributes outside $\mathcal{F}$ are called *hard*. $\mathcal{F}$ need not contain all the numerical attributes, that is we may also have hard numerical attributes.

We assume that each relations $R \in \mathcal{R}$ has a primary key K_R, $K_R \subseteq \mathcal{A}_R$. $\mathcal{K}$ is the set of key constraints. We assume that $\mathcal{K}$ is satisfied by the initial instance D, denoted $D \models \mathcal{K}$. It also holds $\mathcal{F} \cap K_R = \emptyset$, i.e. values in key attributes cannot be updated. In addition, there may be a separate set of *flexible* ICs *IC* that may be violated, and it is the job of a repair to restore consistency wrt them (while still satisfying $\mathcal{K}$).

A *linear denial constraint* [14] has the form $\forall \bar{x} \neg(A_1 \wedge \ldots \wedge A_m)$, where the A_i are database atoms (i.e. with predicate in $\mathcal{R}$), or built-in atoms of the form $x\theta c$, where x is a variable, c is a constant and $\theta \in \{=, \neq, <, >, \leq, \geq\}$, $x = y$ or $x \neq y$. We will usually replace $\wedge$ by commas in denials. We denote by $\mathcal{A}_{\mathcal{B}}(ic)$ the attributes in built-ins in a constraint ic. A set of linear denials *IC* is *local* if: (a) Attributes participating in equality atoms or joins are all hard attributes; (b) For each $ic \in IC$, $(\mathcal{A}_{\mathcal{B}}(ic) \cap \mathcal{F}) \neq \emptyset$; (c) No attribute A appears in *IC* both in comparisons of the form $A < c_1$ and $A > c_2$.[2] A linear denial constraint ic is said to be *local* if the set $\{ic\}$ is *local*. Local constraints have the property that by doing local fixes, no new inconsistencies will be generated, and there will always exist a repair [4]. Locality is a sufficient, but not necessary condition for existence of repairs.

For a tuple $\bar{k}$ of values for the key K_R of relation R in an instance D, $\bar{t}(\bar{k}, R, D)$ denotes the unique tuple $\bar{t}$ in relation R in instance D whose key value is $\bar{k}$. To each attribute $A \in \mathcal{F}$ a fixed numerical weight α_A is assigned.

[1] With simple denial constraints, numbers can be restricted to, e.g. $\mathbb{N}$ or $\{0, 1\}$.

[2] To check condition (c), $x \leq c$, $x \geq c$, $x \neq c$ have to be expressed using $<, >$, e.g. $x \leq c$ by $x < c + 1$.

Definition 2.1 [4] For instances D and D' over schema Σ with the same set $val(K_R)$ of tuples of key values for each relation $R \in \mathcal{R}$ and a distance function *Dist*, the Δ-*distance* between the databases is

$$\Delta(D, D') = \sum_{\substack{R \in \mathcal{R}, A \in \mathcal{F} \\ \bar{k} \in val(K_R)}} \alpha_A Dist(\pi_A(\bar{t}(\bar{k}, R, D)), \pi_A(\bar{t}(\bar{k}, R, D')))$$

where π_A is the projection on attribute A. □

The distance function *Dist* should increase monotonically over the absolute values of the differences between values and all the results that follow are valid for any such distance. For example, it can be the "city distance" L_1 (the sum of absolute differences) or the "euclidian distance" L_2 (the sum of the square of differences). The type of distance to use will depend on the type of data stored in the databases. For simplicity in the examples we will assume L_1-distance. The coefficients α_A can be chosen in many ways depending on the relevance of the attribute, the actual distribution of the data, or a compensation of different scales of measurement.

Definition 2.2 [4] For an instance D, a set of flexible attributes $\mathcal{F}$, a set of key dependencies $\mathcal{K}$, such that $D \models \mathcal{K}$, and a set of flexible ICs *IC*: A *repair candidate* for D wrt *IC* is an instance D' such that: (a) D' has the same schema and domain as D; (b) for every $R \in \mathcal{R}$, $A \in \mathcal{A}_R \setminus \mathcal{F}$ and $\bar{k} \in val(K_R)$, $\pi_A(\bar{t}(\bar{k}, R, D)) = \pi_A(\bar{t}(\bar{k}, R, D'))$; (c) $D' \models \mathcal{K}$; and (d) $D' \models IC$. A *repair* for D is a repair candidate D' that minimizes the distance $\Delta(D, D')$ over all the instances that satisfy (a)–(d). Finally, $Rep^{At}(D, IC)$ is the set of repairs of D wrt *IC*. □

Intuitively, D' is a repair if it has the same schema as D, does not modify the values in the hard attributes of D, satisfies the constraints and is as close as possible to D.

Example 2.3 (example 1.1 continued) $\mathcal{R} = \{Paper\}$, $\mathcal{A} = \{ID, EF, PRC, CF\}$, $K_{Paper} = \{ID\}$, $\mathcal{F} = \{EF, PRC, CF\}$, with weights $\bar{\alpha} = (1, \frac{1}{20}, \frac{1}{2})$, resp. The constraints over environmentally friendly papers can be expressed as $ic_1 : \forall xyzw\neg(Paper(x, y, z, w) \wedge y > 0 \wedge z < 50)$ and $ic_2 : \forall xyzw\neg(Paper(x, y, z, w) \wedge y > 0 \wedge w < 1)$. The first two tuples of D do not satisfy the constraints. Since we want to repair the database by making the smallest possible modifications, we consider as possible repairs for t_1: $t_1^1 = Paper(B1, \mathbf{0}, 40, 0)$[3] or $t_1^2 = Paper(B1, 1, \mathbf{50}, \mathbf{1})$, and for t_2 : $t_2^1 = Paper(C2, \mathbf{0}, 20, 1)$ or $t_2^2 = Paper(C2, 1, \mathbf{50}, 1)$. Then, there are four natural candidate repairs $D_1 = \{t_1^1, t_2^1, t_3\}$, $D_2 = \{t_1^2, t_2^1, t_3\}$, $D_3 = \{t_1^1, t_2^2, t_3\}$ and $D_4 = \{t_1^2, t_2^2, t_3\}$. The respective distances are: $\Delta(D, D_1) = 1 + 1 = 2$, $\Delta(D, D_2) = \frac{1}{20} \times 10 + \frac{1}{2} \times 1 + 1 = 2$, $\Delta(D, D_3) = 1 + (\frac{1}{20} \times 30) = 3$ and $\Delta(D, D_4) = (\frac{1}{20} \times 10 + \frac{1}{2} \times 1) + (\frac{1}{20} \times 30) = 2.5$. Since D_1 and D_2 minimize the distance, they are the only repairs for D wrt the constraint: Therefore, the two repairs of the database are:

D_1:

Paper	*ID*	*EF*	*PRC*	*CF*	
	B1	0	40	0	t_1^1
	C2	0	20	1	t_2^1
	E3	1	70	1	t_3

D_2:

Paper	*ID*	*EF*	*PRC*	*CF*	
	B1	1	50	1	t_1^2
	C2	0	20	1	t_2^1
	E3	1	70	1	t_3

□

Definition 2.4 [4] A set I of database tuples from D is a *violation set* for $ic \in IC$ if $I \not\models ic$, and for every $I' \subsetneq I$, $I' \models ic$. Let $\mathcal{I}(D, ic) = \{I | I$ is a violation set for $ic\}$ and $\mathcal{I}(D, ic, t) = \{I | I \in \mathcal{I}(D, ic)$ and $t \in I\}$. The *degree of inconsistency of a tuple* t is $Deg(t, IC) = |\{I | I \in \mathcal{I}(D, ic, t), ic \in IC\}|$ and the *degree of inconsistency of a database* D is $Deg(D, IC) = Max\{Deg(t, IC) | t \in D\}$. □

A violation set I for ic is a minimal set of tuples that simultaneously participate in the violation of ic. The degree of inconsistency of a tuple is the number of violation sets that contain it.

Example 2.5 (example 2.3 continued) Let us add to our database a table *Pub* that stores the *ID* of a publication (e.g. magazine) the *ID* of the paper used (*PID*) and its number of pages (*Pag*). The primary key for this table is *ID* and the only flexible attribute is *Pag* with $\alpha_{Pag} = \frac{1}{10}$.

Pub	*ID*	*PID*	*Pag*	
	235	B1	45	p_1
	112	B1	30	p_2
	100	E3	80	p_3

The publisher has a requirement that publications of more than 40 pages should use paper that contains at least 70% of recycled paper, i.e. $ic_3 : \forall xyzuvw\neg(Pub(x, y, z) \wedge Paper(y, u, v, w) \wedge z > 40 \wedge v < 70)$. Then the violation sets for each integrity constraints are: $\mathcal{I}(D, ic_1) = \{\{t_1\}, \{t_2\}\}$, $\mathcal{I}(D, ic_2) = \{\{t_1\}\}$ and $\mathcal{I}(D, ic_3) = \{\{t_1, p_1\}\}$. Note that $\{ic_1, ic_2, ic_3\}$ is a set of local constraints. □

[3] The attributes that are changed wrt the original tuple are in bold.

Definition 2.6 Given an instance D and ICs IC, a *local fix* for $t \in D$, is a tuple t' with: (a) the same values for the hard attributes as t; (b) $S(t,t') := \{(I, ic) \mid ic \in IC,\ I \in \mathcal{I}(D, ic, t)$ and $((I \setminus \{t\}) \cup \{t'\}) \models ic\} \neq \emptyset$; and (c) there is no tuple t'' that simultaneously satisfies (a), $S(t,t'') = S(t,t')$, modifies the same attributes as t' and $\Delta(\{t\},\{t''\}) \leq \Delta(\{t\},\{t'\})$, where Δ denotes the distance function. A *mono-local fix* is a local fix that modifies only one attribute. □

$S(t,t')$ contains the violation sets that include t and are solved changing t by t'. A local fix t' solves at least one inconsistency and minimizes the distance to t. In [4] it was proven that if D' is a repair D wrt IC then for every relation R and every key $\bar{k}$ in it, $\bar{t}(\bar{k}, R, D')$ is equal to $\bar{t}(\bar{k}, R, D)$ or it is a local fix of $\bar{t}(\bar{k}, R, D)$ wrt to IC. Here, instead of using local fixes, we will show we can concentrate in mono-local fixes. This can be done since any local fix can be seen as a combination of mono-local fixes.

Proposition 2.7 Given a tuple t, a local integrity constraint ic and a flexible attribute A such that $A \in \mathcal{A}_{\mathcal{B}}(ic)$, there is a unique mono-local fix t' that modifies attribute A. □

Definition 2.8 shows how to obtain the mono-local fix of tuple t that modifies attribute A and solves an inconsistency wrt ic.

Definition 2.8 Given a local constraint ic, a tuple t and an attribute $A \in \mathcal{A}_{\mathcal{B}}(ic)$, the tuple $MLF(t, ic, A)$ is obtain by: (1) replacing every $\leq$ and $\geq$ in ic by $<$ and $>$, (2) If ic contains (a) $A < c_1, \ldots, A < c_n$ then $MLF(t, ic, A)$ is a tuple obtained from t by replacing attribute A by $Min\{c_1, \ldots, c_n\}$. (b) $A > c_1, \ldots, A > c_n$ then $MLF(t, ic, A)$ is a tuple obtained from t by replacing attribute A by $Max\{c_1, \ldots, c_n\}$. □

Proposition 2.9 For a database D, a local constraint ic, a tuple t and an attribute $A \in \mathcal{A}_{\mathcal{B}}(ic)$. If $\mathcal{I}(D, ic, t) \neq \emptyset$, then $MLF(t, ic, A)$ is the unique mono-local fix of t wrt ic for attribute A. □

Example 2.10 (example 2.3 continued) All the possible local fixes for t_1 are $t_1^1 = MLF(t_1, IC_1, EF) = MLF(t_1,\ IC_2,\ EF) = Paper(B1, \mathbf{0}, 40, 0)$, $t_1^2 = Paper(B1,\ 1, \mathbf{50}, \mathbf{1})$, $t_1^3 = MLF(t_1, IC_1, PRC) = Paper(B1, 1, \mathbf{50}, 0)$ and $t_1^4 = MLF(t_1, IC_2, CF) = Paper(B1, 1, 40, \mathbf{1})$. t_1^2 is a local fix but not a mono-local fix since it modifies two attributes. The violation sets solved with each mono-local fix are: $S(t_1, t_1^1) = S(t_1, t_1^2) = \{(\{t_1\}, ic_1),\ (\{t_1\}, ic_2)\}$, $S(t_1, t_1^3) = \{(\{t_1\}, ic_1)\}$ and $S(t_1, t_1^4) = \{(\{t_1\}, ic_2)\}$. □

3. Attribute-Update Repairs

For a fixed set of linear denials IC, the optimization problem of finding the minimum distance from a repair wrt IC to a given input instance is *MAXSNP*-hard and therefore cannot be uniformly approximated within arbitrarily small constant factors (unless $P = NP$) [4]. By restricting to a set of local denial constraints the problem is still *MAXSNP*-hard but can we can transformed into an instance of the *Minimum Weighted Set Cover Optimization Problem* (*MWSCP*). Given a collection $\mathcal{S}$ of subsets of a set U and a positive weight $w(S_i)$ for each $S_i \in \mathcal{S}$, a set cover $\mathcal{C}$ is a subset of $\mathcal{S}$ such that every element in U belongs to at least one member of $\mathcal{C}$. The weight of the cover is the sum of the weights of the sets in it. The *MWSCP* consists in finding a set cover with a minimum weight.

Definition 3.1 For a database D and a set IC of local denials, the instance $(U, \mathcal{S}, w)$ for the *MWSCP*, denoted $(U, \mathcal{S}, w)^{(D,IC)}$, where U is the underlying set, $\mathcal{S}$ is the set collection, and w is the weight function, is given by: (a) $U := \mathcal{I}(D, IC)$. (b) $\mathcal{S}$ contains the $S(t,t')$, where t' is a mono-local fix for a tuple $t \in D$. (c) $w(S(t,t')) := \Delta(\{t\},\{t'\})$. □

This transformation is a slight modification of the one in [4] where t' could also be local fix that modified more than one attribute. The modification can be done since a local fix can always be constructed as a combination of mono-local fixes [5, Proposition A.3]. In both cases if we find a minimum weight cover $\mathcal{C}$, we could try to construct the repair by replacing each inconsistent tuple $t \in D$ by the mono-local fix t' with $S(t,t') \in \mathcal{C}$. If there is more than one mono-local fix for a tuple t then we need to combine the updates in a unique local fix. This can be done easily since in an optimal solution of the MWSCP, if there is more than one mono-local fixes they will modify different attributes (by Proposition 2.7) and therefore we can construct a unique local fix by combining the updates of the different attributes.

Definition 3.2 [4] Let $\mathcal{C}$ be a cover for instance $(U, \mathcal{S}, w)^{(D,IC)}$ of the *MWSCP*. (a) $\mathcal{C}^\star$ is obtained from $\mathcal{C}$ as follows: For each tuple t with mono-local fixes $t_1, \ldots, t_n$, $n > 1$, such that $S(t, t_i) \in \mathcal{C}$, replace in $\mathcal{C}$ all the $S(t, t_i)$ by a single $S(t, t^\star)$, where $t^\star$ is such that $S(t, t^\star) = \bigcup_{i=1}^{n} S(t, t_i)$. (b) $D(\mathcal{C})$ is the database instance obtained from D by replacing t by t' if $S(t,t') \in \mathcal{C}^\star$. □

Tuple $t^\star$ always exists and is a local fix of t obtained from the mono-local fixes $t_1, \ldots, t_n$ by combining the updates associated to each of them.

Example 3.3 (example 2.5 continued) We illustrate the reduction from *DFOP* to *MWSCP*. To construct the *MWSCP* instance we need the mono-local fixes. The mono-local fixes of t_1 are $t_1^1 = Paper(B1, \mathbf{0}, 40, 0)$, $t_1^2 = Paper(B1, 1, \mathbf{50}, 0)$, $t_1^3 = Paper(B1, 1, 40, \mathbf{1})$ and $t_1^4 = Paper(B1, 1, \mathbf{70}, 0)$ with $S(t_1, t_1^1) = \{(\{t_1\}, ic_1), (\{t_1\}, ic_2)\}$, $S(t_1, t_1^2) = \{(\{t_1\}, ic_1)\}$, $S(t_1, t_1^3) = \{(\{t_1\}, ic_2)\}$ and $S(t_1, t_1^4) = \{(\{t_1\}, ic_1), (\{t_1, p_1\}, ic_3)\}$. The mono-local fixed of t_2 are $t_2^1 = Paper(C2, \mathbf{0}, 20, 1)$ and $t_2^2 = Paper(C2, 1, \mathbf{50}, 1)$ with $S(t_2, t_2^1) = \{(\{t_2\}, ic_1)\}$ and $S(t_2, t_2^2) = \{(\{t_2\}, ic_1)\}$. The mono-local fixes of p_1 are $p_1^1 = Pub(235, B1, \mathbf{40})$ with $S(p_1, p_1^1) = \{(\{t_1, p_1\}, ic_3)\}$. The *MWSCP* instance is shown in the table below, where the elements are rows and the sets (e.g. $S_1 = S(t_1, t_1^1)$), columns. An entry 1 means that the set contains the corresponding element; and a 0, otherwise.

Set	S_1	S_2	S_3	S_4	S_5	S_6	S_7
Local Fix	t_1^1	t_1^2	t_1^3	t_1^4	t_2^1	t_2^2	p_1^1
Weight	1	0.5	0.5	1.5	1	1.5	1
$(\{t_1\}, ic_1)$	1	1	0	1	0	0	0
$(\{t_1\}, ic_2)$	1	0	1	0	0	0	0
$(\{t_2\}, ic_1)$	0	0	0	0	1	1	0
$(\{t_1, p_1\}, ic_3)$	0	0	0	1	0	0	1

There are three minimal covers, each with weight 3: $\mathcal{C}_1 = \{S_1, S_5, S_7\}$ $\mathcal{C}_2 = \{S_2, S_3, S_5, S_7\}$ and $\mathcal{C}_3 = \{S_3, S_4, S_5\}$. Since all the mono-local fixes in $\mathcal{C}_1$ correspond to different tuples, the repair associated to it can be obtained by directly replacing each tuple by its fix: $D(\mathcal{C}_1)$=$\{t_1^1, t_2^1, t_3, p_1^1, p_2, p_3\}$. On the other hand, $\mathcal{C}_2$ has two mono-local fixes for t_1. In this case we need to combine t_1^2 and t_1^3 into $t_1^5 = (B1, 1, \mathbf{50}, \mathbf{1})$ producing the repair $D(\mathcal{C}_2)$=$\{t_1^5, t_2^1, t_3, p_1^1, p_2, p_3\}$. Finally, for $\mathcal{C}_3$ we need to combine t_1^3 and t_1^4 into $t_1^6 = (B1, 1, \mathbf{70}, \mathbf{1})$ producing the repair $D(\mathcal{C}_3)$=$\{t_1^6, t_2^1, t_3, p_1, p_2, p_3\}$. □

The MWSCP can be approximated in the general case within a logarithmic factor using the greedy algorithm (see Algorithm 1) [16, 10]. The greedy algorithm consists in choosing at each stage the set which contains the largest number of uncovered elements until all the elements are covered [10].

Example 3.4 (example 3.3 continued) In the first iteration we have $w_{ef}(S_1) = 0.5$, $w_{ef}(S_2) = 0.5$, $w_{ef}(S_3) = 0.5$, $w_{ef}(S_4) = 0.75$, $w_{ef}(S_5) = 1$, $w_{ef}(S_6) = 1.5$ and $w_{ef}(S_7) = 1$. We can choose S_1, S_2, S_3 or S_4. If we choose S_1 we get $\mathcal{C} = \{S_1\}$. Now the effective weights are updated to: $w_{ef}(S_1)$ = undefined, $w_{ef}(S_2)$ = undefined, $w_{ef}(S_3)$ = undefined, $w_{ef}(S_4) = 1.5$, $w_{ef}(S_5) = 1$, $w_{ef}(S_6) = 1.5$ and $w_{ef}(S_7) = 1$. By choosing S_5 we get $\mathcal{C} = \{S_1, S_5\}$. Now the only effective weights that are not undefined are $w_{ef}(S_4) = 1.5$ and $w_{ef}(S_7) = 1$. Therefore we choose S_7 and now $\mathcal{C} = \{S_1, S_5, S_7\}$. All the elements are covered so the process stops and $\{S_1, S_5, S_7\}$ is an approximate cover. In this case $\mathcal{C}$ is an optimal cover. □

Algorithm 1: $GreedySC(U, \mathcal{S}, w)$: Set Cover Greedy Approximation Algorithm

Input: An instance $(U, \mathcal{S}, w)$ of the MWSCP constructed using Definition 3.1
Output: Set cover $\mathcal{C}$
$\mathcal{C} \leftarrow \emptyset$, $E \leftarrow \emptyset$, $i \leftarrow 0$;
while While there are non-covered elements $(U \setminus E \neq \emptyset)$ **do**
  foreach $s \in \mathcal{S}$ **do**
    $w_{ef}(s) \leftarrow \frac{w(s)}{|s|}$;
  $M \leftarrow$ element in $\mathcal{S}$ with smallest w_{ef};
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{M\}$;
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{M\}$;
  $E \leftarrow E \cup S$;
  foreach $s \in \mathcal{S}$ **do**
    $s \leftarrow s \setminus M$
return $\mathcal{C}$;

A cover that is not optimal might contain two mono-local fixes for the same tuple t and attribute A but for different ICs. In this case in order to construct the repair from the cover, we use the mono-local fix with higher weight, since that one will subsume the other one (since the ICs are local).

Proposition 3.5 Given a database D and a set of local linear denial ICs *IC*, the running time of the greedy algorithm for the instance $(U, \mathcal{S}, w)^{(D, IC)}$ of the MWSCP is $O(n^3)$ for n the size of the D. If $Deg(D, IC)$ is bounded by a constant, the running time is $O(n^2)$. □

Proof: The cost of updating the weights, choosing the element with smallest weight and updating set E in iteration i is $O(|\mathcal{S}_i|)$ where S_i is the size of $\mathcal{S}$ in iteration i. Since one set is deleted in each iteration, $O(|\mathcal{S}_i|))$ is $O((|\mathcal{S}|-i))$. The cost of deleting the covered elements from the sets in $\mathcal{S}_i$ is $O(|\mathcal{S}_i||M|)$, i.e. $O(|(\mathcal{S} - i)||M|)$. Then, the cost of each iteration is $O((|\mathcal{S}| - i)(1+|M|))$. In the worst case we will have $|\mathcal{S}|$ iteration, i.e. all the sets in $\mathcal{S}$ will belong to the cover. Therefore the running time is $\sum_{i=1}^{|\mathcal{S}|}((|\mathcal{S}| - i)(1 + |M|)) = (1 + |M|)(|\mathcal{S}|^2 - |\mathcal{S}|(|\mathcal{S}| + 1)/2)$ which is $O((1+|M|)|\mathcal{S}|)$. $\mathcal{S}$ is the set of all possible mono-local fixes for a set of inconsistent tuple, therefore $|\mathcal{S}|$ is $O(n|\mathcal{F}|)$ for n the number of tuples in the database. On the other hand, in the general case M is $O(n)$ since a tuple my be involved in all the inconsistencies of the database. In this case, this would give us a running time of $O(n^3)$. If

the degree of inconsistency of the database is bounded, then M is also bounded and therefore the algorithm runs in $O(n^2)$. □

It is common to find in practice inconsistent databases where the degree of inconsistency is bounded by a small number. For example, in a census databases, where a tuple corresponds to a person and his/her data, each tuple can only be involved in inconsistencies with the tuples representing other members of his/her household [11]. Since the number of members in household can be bounded, by 10 for example, the degree of inconsistency is bounded. Therefore, in most practical cases, we expect the greedy algorithm to run in $O(n^2)$.

Even though the algorithm runs in quadratic time when the degree of inconsistency is bounded, it is still too expensive for large databases (more than one million tuples). This is why we want to modify the greedy algorithm to obtain a more efficient algorithm. The expensive task in the algorithm corresponds to searching for the set with more uncovered elements, i.e. the one with minimum effective weight. We can improve the running time by storing $\mathcal{S}$ of the MWSCP in a priority queue P. Each object in the priority queue will contain an inconsistent tuple t, a mono-local fix of it, its weight, and a set of pointers to the elements of the U, i.e. the violation sets. These violation sets will be stored in an array A. Therefore, the modification of the greedy algorithm consists on modifying the data structure that stores the sets and elements, in such a way that we can efficiently search for the set with minimal effective weight.

First we will describe how to construct the MWSC problem from the database D and the set of local constraints IC. Algorithm 2 constructs array A containing the violation sets. The violations sets are obtained by rewriting each integrity constraint as a SQL view that is empty if it is being satisfied.

Example 3.6 (example 2.3 continued) The violations sets of ic_1 can be retrieved from D by posing the SQL query: `SELECT X Y Z W FROM Paper WHERE Y>0 AND Z <50` □

Algorithm 3 constructs the priority queue P by computing for each inconsistent tuple t all the mono-local fixes $t' = MLF(t, ic, A)$ and creating an array order by the weight associated to it. This ordered array can be seen as an array implementation of a heap priority queue. At this stage, the set $S(t, t')$ of every object in P is empty. Algorithm 4 will later add to each $S(t, t')$ references to the violations sets stored in array A. The algorithm uses an auxiliary hash table that contains links to the elements in P.

Algorithm 2: $ViolationSet(D, IC)$: Construct Array of Violation Sets

Input: Database D and set of local ICs IC
Output: Array A containing the violation sets
Array A;
foreach $ic \in IC$ **do**
  Q ← SQL query to obtain violation sets of ic;
  S ← execute_query(Q,D);
  foreach $s \in S$ **do**
    Add (S, ic) to A;
return A;

Algorithm 3: $LocalFix\_PQ(D, IC)$: Construct priority queue with mono-local fixes

Input: Database D and set of local ICs IC
Output: A priority queue containing mono-local fixes, ordered by their weight
Array P, L; int i ← 0, j ← 0;
foreach $ic \in IC$ **do**
  foreach Relation $R \in ic$ **do**
    foreach $A \in (\mathcal{A}_R \cap \mathcal{A}_{\mathcal{B}}(ic) \cap \mathcal{F})$ **do**
      Q ← SQL query that retrieves $(t \mid$ t is in relation R, $t \in v$, $v \in \mathcal{I}(D, ic))$ ordered by $\Delta(t, MLF(t, ic, A))$;
      Array T ← execute_query(Q,D);
      L[j] ← i; j++; int k ← 0;
      while $k < T.size()$ **do**
        $t \leftarrow T[k]$;
        $t' \leftarrow (MLF(T[k], ic, A))$;
        $P[i + k] \leftarrow (t, t', \Delta(t, t'), \emptyset)$ with $\Delta(t, t')$ as priority
        k++;
      $i \leftarrow i + k$;
execute in-place ordered merge on P using L as an array of pointers on ordered subarrays;
priority queue Q ← P;
return Q;

Algorithm 4: $LinkPQArray(P, A)$: links elements in P with elements in A

Input: Priority queue P with elements $(t, t', w, \emptyset)$ and array A with elements (S, ic)
Output: Modifies P to contain $(t, t', w, S(t, t'))$
Hash table H; int $i = 0$;
foreach $(t, t', w, S) \in P$ **do**
  Add link to (t, t', w, S) into H using t as key;
while $i < A.size()$ **do**
  foreach tuple $u \in A[i].S$ **do**
    foreach $(u, u', w, S) \in H$ **do**
      if $(A[i].S \setminus u \cup u')$ satisfies $A[i].ic$ **then**
        Add link to $A[i]$ in S;
        Add link to (u, u', w, S) in $A[i]$;

Using the priority queue P and the array structure A of the MWSC problem, the modified-greedy algorithm can now be run as described in Algorithm 5. Now, the minimal element of the priority queue is added to the cover in each iteration. Every time a set is added to the set cover, the weights in the priority queue are updated. Since the elements in A have also links to the elements in P that contain them, this update can be done efficiently. Secondly, we need to restore the heap structure. This can be done by performing up-heap for every updated element.

Algorithm 5: $ModifiedGreedySC(A, P)$: modified Set Cover Greedy Approximation Algorithm

Input: An array A containing violation sets and a priority queue P containing $(t, t', w, S(t, t'))$, t is a tuple, t' its fix, w the weight of the fix and $S(t, t')$ the set of violation sets solved by t'

Output: Subset of elements in P such that the union of the $S(t, t')$ covers all the elements in A

Set C;
while There are non-covered elements in A **do**
  $(t, t', w, S(t, t')) \leftarrow$ minimal element in P;
  Add $(t, t', w, S(t, t'))$ to C;
  Mark in A elements in $S(t, t')$ as covered;
  Update weights of elements in P which contain elements in $S(t, t')$;
  Update P to preserve heap structure;
return C;

Algorithm 6 describes the complete algorithm including the construction of the MWSC problem and the execution of the modified-greedy algorithm.

Algorithm 6: Approximate repair

Input: Database D and set of local ICs IC
Output: Approximation of database repair
Create priority queue $P \leftarrow LocalFix\_PQ(D, IC)$;
Create array $A \leftarrow ViolationSet(D, IC)$;
$LinkPQArray(P, A)$;
Set $C \leftarrow ModifiedGreedySC(A, P)$;
return repair D' constructed using set cover C;

Proposition 3.7 Given a database D and a set of local ICs IC the running time of the modified-greedy algorithm for the instance $(U, \mathcal{S}, w)^{(D,IC)}$ of the MWSCP is $O(n^2 \log n)$ for n the size of the D. If $Deg(D, IC)$ is bounded the running time is $O(n \log n)$. □

Proof: First we will analyze the cost of each iteration. Obtaining the minimal element in P can be done in $O(|\log \mathcal{S}|)$. Marking the elements in A as covered is done in constant time for each element in $S(t, t')$ and therefore all the marking can be done in $O(|S(t, t')|)$. Since in $(U, \mathcal{S}, w)^{(D,IC)}$ each element in U belongs to a bounded number of sets in $\mathcal{S}$, the cost of changing the weights of the modified sets takes $O(|S(t, t')|)$. Finally the heap structure can be updated by performing up-heap for each updated element, and therefore it runs is $O(|\log(\mathcal{S})||S(t, t')|)$. Then, the total cost of one iteration of the algorithm is $O(|\log \mathcal{S}| + 2|S(t, t')| + |\log(\mathcal{S})||S(t, t')|)$. There can be at most $|\log \mathcal{S}|$ iteration and therefore the total cost is $O(|\mathcal{S}|(|\log \mathcal{S}| + 2|S(t, t')| + |\log(\mathcal{S})||S(t, t')|))$. We know that $|\mathcal{S}|$ is $O(n)$ and that if the degree of inconsistency of the database is not bounded that $|S(t, t')|$ is $O(n)$ and in consequence that the running time is $O(n^2 \log n)$. On the other hand, if the degree of inconsistency of database is bounded, the running time is $O(n \log n)$. □

The modified greedy algorithm computes the same approximation as the greedy algorithm and therefore approximates within a logarithmic factor. In our particular case we can instead use the layer algorithm to approximate within a constant factor since the number of occurrences of an element of U in elements of $\mathcal{S}$ is bounded by a constant [13, Chapter 3]. The layer algorithm consists on adding to the cover, in each iteration, the sets with weight zero until all the elements are covered. The weight of each set s is set at iteration i to $w_i(s) = (w_{i-1}(s) - c|s|)$ for $c = min\{w_{ef}(s) \mid s \in \mathcal{S}_i\}$ and $S_i = S_{i-1} \setminus C_i$ [20]. The new data structure used for the modified version of the greedy algorithm can also be used for the layer approximation algorithm.

4. Experiments

We would like to study the performance of the greedy and layer algorithms and their modified versions introduced in this article, i.e. the ones using a priority queue to store the sets in $\mathcal{S}$. First we compare which one provides, in practice, a better approximation of the optimal weight of the set cover, i.e. the distance between the original database and the repair. Second, we compare the running time of the two algorithms and their modified versions.

The architecture of the system is shown in Figure 1. The *configuration file* contains information about the schema of the database, the integrity constraints, the flexible/non-flexible attributes, database repair mode (update, insert into a new database, dump into text file). The *configuration parser* parses the configuration file and stores configuration information in main memory. The *database connectivity* component is respon-

sible for the low-level database connectivity (JDBC, native database connectivity). The *mapping component* is responsible for mapping the database into the instance of MWSCP and then to construct the repair from the approximate cover obtained as a result of the algorithm. In order to create the MWSCP instance, the mapping component reads the configuration file, then loads data from database through the database connectivity element into main memory. When the instance is created, the mapping component notifies the *MWSCP solver component*, which solves the $MWSCP$. The solution is sent back to the mapping component that reads configuration file to choose the repair export mode (database update, database insert, database dump into text file) and exports the solution. In this article we have concentrated in the optimization of the MWSCP solver and the mapping component.

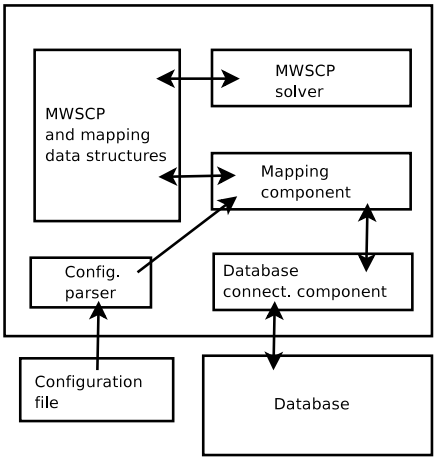

Figure 1. Architecture of the Repair Program

The database repair system with this architecture was implemented in C++ (GNU compiler) and tested on 2.4GHz, Intel, 1GB RAM, Ubuntu Linux computer. The data were stored in an Oracle 10G database system. We considered the following data schema used in [4, 5]: $\mathcal{R} = \{Client, Buy\}$, $\mathcal{A}_{Client} = \{ID, A, C\}$, $\mathcal{A}_{Buy} = \{ID, I, P\}$, $\mathcal{K}_{Client} = \{ID\}$, $\mathcal{K}_{Buy} = \{ID, I\}$, $\mathcal{F} = \{A, P, C\}$ and $IC = \{\forall ID, P, A, C \neg(Buy(ID, I, P),\ Client(ID, A, C), A < 18,\ P > 25),\ \forall ID, A, C \neg(Client(\ ID, A, C), A < 18,\ C > 50)\}$. We generated 3 random databases for different sizes with around 30% of tuples involved in inconsistencies and calculated approximation of the repairs of them wrt *IC*. The results provided for each size correspond to the average of the three results.

The first step consisted in comparing the algorithm to determine which gave a better approximation of the database repair problem. The results are shown in Figure 2. Theoretically we would expect the layer algorithm to give a better approximation than the greedy, but the experiments show that in practice the latter produces better approximation of the optimal weight of the MWSCP. We do not need to run this experiments for the modified version of the algorithms since they provide the same approximation of the total weight.

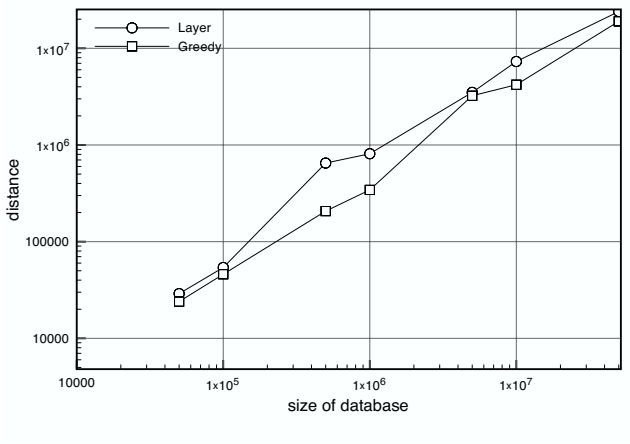

Figure 2. Distance Approximation

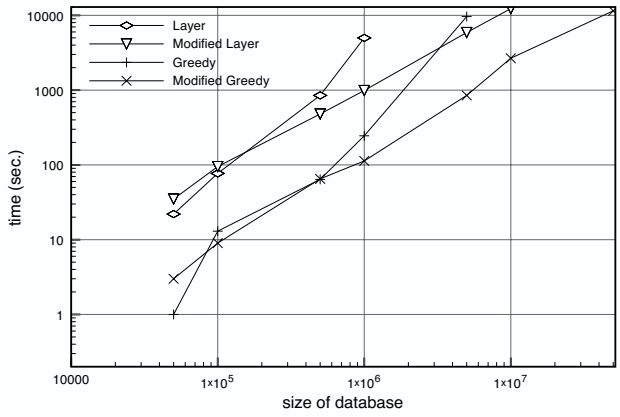

Figure 3. Running Time

Next, we compared the running time of the MWSCP approximation algorithms. Therefore, we only considered the time of the MWSCP solver component. The results of the experiments are shown in Figure 3. Clearly, the use of a priority queue to store the set of the MWSCP instance increases the performance for large databases for both, the layer and greedy algorithm. More over, the modified greedy algorithm shows to be the most efficient for large databases.

The experiments show that the modified-greedy algorithm can be used efficiently to calculate the repairs of database wrt a set of local integrity constraints, even

for large databases. Not only that, but the experiments show that in practice, the modified-greedy algorithms runs faster and gives better approximations than the layer algorithm.

5. Cardinality Repairs

So far we have considered repairs that can be obtained by updating values of tuples in the database. Alternatively we could consider repairing inconsistencies by deleting tuples. The "minimality" of the repairing process can be captured by either minimizing the number of tuples (cardinality semantics) to be deleted or by deleting only sets of tuples that are minimal under set inclusion (set semantics). In the context of CQA, the set semantics for tuple deletions and also for tuple deletions and insertions [1, 9, 2, 3, 8, 12] and the cardinality semantics [3, 7, 15] have been studied.

In practical cases, the cardinality semantics seems a more natural approach. Consider a database where one tuple contradicts, for example, a thousand tuples: the set semantics would have two repairs – to delete one tuple or to delete thousand tuples, while for the cardinality semantics, there is a unique repair that deletes only one tuple. There are also some applications that require cardinality semantics [11]. The complexity of query answering of ground atomic queries for cardinality semantics is $P^{NP(log)}$ hard and checking if a database is a repair is NP-hard [15]. Obtaining a database repair under this semantics is computationally untractable and therefore, there is a need for approximation algorithms.

In this section we show that it is possible to reduce the problem of finding repairs under the cardinality semantics to the problem of finding repairs obtained by minimal attribute updates (attribute repairs). Furthermore, the set of ICs obtained in the transformation will be local so we can use the approximation algorithms described in Section 3.

In order to transform the problem, we can add an extra attribute δ to each of the relations and assign a 1 to it in each tuple. These δ attributes will be the only flexible attributes in the database. A tuple with $\delta = 1$ is a tuple in the database, whereas one with $\delta = 0$ is a deleted tuple. Therefore, the deletions are defined by replacing the value of attribute δ by a zero. The following definition formalizes the transformation.

Definition 5.1 Given a database D and a set of linear denial ICs IC. Let $D^{\#}$ be the database obtained from D by adding one extra attribute δ_R to every $R \in \mathcal{R}$ and by filling δ_R by ones in each tuple. Let $K_{R^{\#}} = \mathcal{A}_R \setminus \delta_R$ for each $R \in \mathcal{R}$, $\mathcal{F} = \{\delta_R | R \in \mathcal{R}\}$ and $\alpha_A = 1$ for all $A \in \mathcal{F}$. Let $IC^{\#}$ contain $\forall \bar{x} \neg(A_1 \wedge \ldots \wedge A_m \bigwedge_{R_i \in ic} \delta_{R_i} > 0)$ for each $ic \in IC$ of form $\forall \bar{x} \neg(A_1 \wedge \ldots \wedge A_m)$. □

Note that the set of constraints $IC^{\#}$ will also be local, since the only flexible attributes are the δ and they are always compared with $>$.

Definition 5.2 Let $D \downarrow \delta$ be the database obtained from D by first deleting from every relation $R \in \mathcal{R}$ the tuples with $\delta_R = 0$ and then deleting attribute δ_R from all R in $\mathcal{R}$. □

Let $Rep^{\#}(D, IC)$ be the set of repairs obtained by repairing D wrt IC using a minimum number of tuple deletions.

Proposition 5.3 Given a database D and a set of extended linear denial constraints IC, $D' \in Rep^{\#}(D, IC)$ iff there is a $D'' \in Rep^{At}(D^{\#}, IC^{\#})$ such that $D'' \downarrow \delta = D'$. □

Note that we do not require IC to contain primary keys for the relations in D nor we required them to be local. These restrictions are needed for the attribute-update repairs. By construction, $IC^{\#}$ is a set of local constraints and therefore we can always apply the approximation algorithm introduced in Section 3.

Example 5.4 Consider $\mathcal{R} = \{P, T\}$, $\mathcal{A}_P = \{A, B\}$ and $\mathcal{A}_T = \{C, D\}$. A database $D = \{P(1, b), P(1, c), P(2, e), T(e, 4)\}$ is inconsistent wrt $ic_1 = \forall xyz \neg(P(x, y) \wedge P(x, z) \wedge y \neq z)$ and $ic_2 = \forall xyz \neg(P(x, y) \wedge T(y, z) \wedge z < 5)$. If we want to calculates the repairs obtained by tuple deletion we can transform the problem into the following attribute repair problem: $\mathcal{A}_{P^{\#}} = \{A, B, \delta_P\}$, $\mathcal{A}_{T^{\#}} = \{C, D, \delta_T\}$, $K_{R^{\#}} = \{A, B\}$, $K_{T^{\#}} = \{C, D\}$, $\mathcal{F} = \{\delta_P, \delta_T\}$, $\bar{\alpha} = \{1, 1\}$, $D^{\#} = \{P(1, b, 1), P(1, c, 1), P(2, e, 1), T(e, 4, 1)\}$ and $IC^{\#} = \{\forall xyz\delta_1\delta_2 \neg(P(x, y, \delta_1) \wedge P(x, z, \delta_2) \wedge y \neq z \wedge \delta_1 > 0 \wedge \delta_2 > 0), \forall xyz\delta_1\delta_2 \neg(P(x, y, \delta_1) \wedge T(y, z, \delta_2) \wedge z < 5 \wedge \delta_1 > 0 \wedge \delta_2 > 0)$. Tuples $P(1, b, 1)$ and $P(1, c, 1)$ are inconsistent wrt ic_1. The inconsistency can be solved by updating δ_P to 0 in the former or latter tuple, which is equivalent to deleting the first or second tuple respectively. The inconsistencies wrt ic_2 are dealt with similarly and the following are the attribute-update repairs: $D_1^{\#} = \{P(1, b, 0), P(1, c, 1), P(2, e, 0), T(e, 4, 1)\}$, $D_2^{\#} = \{P(1, b, 1), P(1, c, 0), P(2, e, 0), T(e, 4, 1)\}$, $D_3^{\#} = \{P(1, b, 0), P(1, c, 1), P(2, e, 1), T(e, 4, 0)\}$ and $D_4^{\#} = \{P(1, b, 1), P(1, c, 0), P(2, e, 1), T(e, 4, 0)\}$. The tuple-deletion repairs of D wrt IC are $D_1 = D_1^{\#} \downarrow \delta = \{P(1, c), T(e, 4)\}$, $D_2 = D_2^{\#} \downarrow \delta = \{P(1, b), T(e, 4)\}$, $D_3 = D_3^{\#} \downarrow \delta = \{P(1, c), P(2, e)\}$ and $D_4 = D_4^{\#} \downarrow \delta = \{P(1, b), P(2, e)\}$. □

6. Conclusion

In this paper we have provided an optimized algorithm to compute attribute-update repairs for an inconsistent database wrt a set of local linear denial constraints. The algorithm works by first transforming the repair problem into the MWSCP and then using an efficient modification of the greedy algorithm to solve the MWSCP. Our experiments suggest that the algorithm runs efficiently even for large databases.

Since the greedy and layer algorithm approximates within a logarithmic and constant factor respectively, we would expect to get better approximations from the layered algorithm. However, the experiments show that the greedy algorithm gives better results for our problem specification. Furthermore, the modified greedy algorithm runs faster than the non-modified greedy algorithm, which in turn runs faster than the layer algorithm.

Finally, by transforming the tuple-deletion repair problem into an attribute-update repair problem we are able to use the modified greedy algorithm. In this case the database does not need to have a set of primary key constraint and there is no requirement of locality over the set of linear denial ICs.

The results presented in this paper could be used to compute repairs under other repair semantics by doing small modifications. For example we can giving different values to the attributes α_{δ_R} so that the distance $\Delta(D, D')$ is modified to give different priority to deletions from different tables. For example, assigning $\alpha_T = 1$ and $\alpha_R = .5$ will have as a consequence that we prefer deleting tuples from table R than from T. Other possible modification is to combine tuple deletions with tuples updates by using as flexible attributes not only δ_R but other attributes. If in example 5.4 we set $\mathcal{F} = \{\delta_P, \delta, T, D\}$, we can repair inconsistencies wrt ic_2 by either deleting tuples or updating attribute D.

References

[1] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *Proc. of PODS'99*, pages 68–79. ACM Press, 1999.

[2] M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *Theoretical Computer Science*, 296(3):405–434, 2003.

[3] L. Bertossi and J. Chomicki. *Logics for Emerging Applications of Databases*, chapter Query Answering in Inconsistent Databases, pages 43–83. Springer, 2003.

[4] L. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. Complexity and Approximation of Fixing Numerical Attributes in Databases under Integrity Constraints. In *Proc. of DBPL'05*, Springer LNCS 3774, pages 262–278, 2005.

[5] L. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. Complexity and Approximation of Fixing Numerical Attributes in Databases under Integrity Constraints. Journal submission, 2006.

[6] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A Cost-based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *Proc. of SIGMOD Conference*, pages 143–154. ACM, 2005.

[7] F. Buccafurri, N. Leone, and P. Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[8] A. Calì, D. Lembo, and R. Rosati. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *Proc. of PODS'03*, pages 260–271. ACM Press, 2003.

[9] J. Chomicki and J. Marcinkowski. Minimal-change Integrity Maintenance Using Tuple Deletions. *Information and Computation*, 197(1-2):90–121, 2005.

[10] V. Chvatal A Greedy Heuristic for the Set Covering Problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[11] E. Franconi, A. Laureti-Palma, N. Leone, S. Perri, and F. Scarcello. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *Proc. of LPAR'01*, Springer LNCS 2250, pages 561–578, 2001.

[12] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. In *Proc. of ICDT'05*, Springer LNCS 3363, pages 337–351, 2005.

[13] D. Hochbaum (ed.) *Approximation Algorithms for NP-Hard Problems*. PWS, 1997.

[14] G. Kuper, L. Libkin and J. Paredaens.(eds.) *Constraint Databases*.Springer, 2000.

[15] A. Lopatenko and L. Bertossi. Complexity of Consistent Query Answering in Databases under Cardinality-based and Incremental Repair Semantics. To appear in *Proc. ICDT'07*, Springer, 2007.

[16] C. Lund and M. Yannakakis On the Hardness of Approximating Minimization Problems. *Journal of the Association for Computing Machinery*, 45(5):960–981, 1994.

[17] H. Muller and J.C. Freytag. Problems, Methods and Challenges in Comprehensive Data Cleansing. Technical report, Humboldt-Universitat zu Berlin, Institut fr Informatik, 2003.

[18] Ch. Papadimitriou *Computational Complexity*. Addison–Wesley, 1994.

[19] E. Rahm and H. H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[20] V. Vazirani Approximation Algorithms. Springer, 2001.

[21] J. Wijsen. Condensed Representation of Database Repairs for Consistent Query Answering. In *Proc. of ICDT'03*, Springer LNCS 2572, pages 378–393, 2003.

[22] J. Wijsen. Database Repairing using Updates. *ACM Transactions on Database Systems*, 30(3):722–768, 2005.