Repairing Entities using Star Constraints in Multirelational Graphs

Peng Lin* Qi Song* Yinghui Wu†‡ Jiaxing Pi§

*Washington State University †Case Western Reserve University

‡Pacific Northwest National Laboratory §Siemens Corporation

{peng.lin, qi.song}@wsu.edu yxw1650@case.edu jiaxing.pi@siemens.com

***Abstract*—This paper studies a class of neighborhood constraints to characterize and repair erroneous entity information in multi-relational graph data. (1) We propose a class of constraints called *star functional dependencies* (StarFDs). Unlike conventional integrity constraints, a StarFD enforces value dependencies conditioned by entities and their relevant neighbors, which are identified by a *star pattern* that incorporates conjunctive regular path queries. StarFDs achieve a balance between expressiveness and complexity: the validation of StarFDs is tractable, and the satisfiability and implication of StarFDs are NP-complete and coNP-complete, respectively. (2) Given a set of StarFDs Σ and a graph G, the *entity repair* problem is to compute a *minimum repair* of G by enforcing Σ with the smallest amount of changes. Although this problem is NP-complete and hard to approximate, we show it is feasible to compute repairs in large graphs. Our approach (a) discriminately detects and resolves errors with *optimal*, *approximable* and *cost-bounded* solutions whenever possible, and (b) incurs a time cost determined by Σ and the size of inconsistencies, for *all* cases. Using real world data, we show that StarFD-based techniques effectively identify and repair errors. We also show that our repairing algorithms benefit other tasks such as fact checking.**

***Index Terms*—data cleaning, knowledge graphs.**

I. Introduction

Real-world graph data is often "dirty" [12], [26], [35]. A major class of errors in the ubiquitous attributed, *multirelational* graphs refer to incorrect attribute values and types pivoted at the entities (nodes). As observed in [34], 23.22% (resp. 25.14%) of 700 sampled triples from diverse classes are caused by incorrect attribute values (resp. wrong types). The need for repairing erroneous entity information is evident in graph search [33], knowledge base completion [25], and provenance [31]. Although integrity constraints such as functional dependencies are extended to capture inconsistencies in labeled graphs [16], the research on repairing erroneous entities in *multirelational* graphs is still in its infancy.

Unlike data cleaning based on integrity constraints [9], repairing erroneous attribute values of entities in a multirelational graph G is more involved. (1) It may require the checking of violations of value constraints among the attributes of the nodes that are "semantically" associated with each other. Such semantic association may not necessarily be explicitly encoded as direct edges (due to *e.g.,* incompleteness), but *paths* summarized by *regular expressions* [3], [6]. (2) Repairing process by updating attribute values requires the detection of new violations via such semantic association.

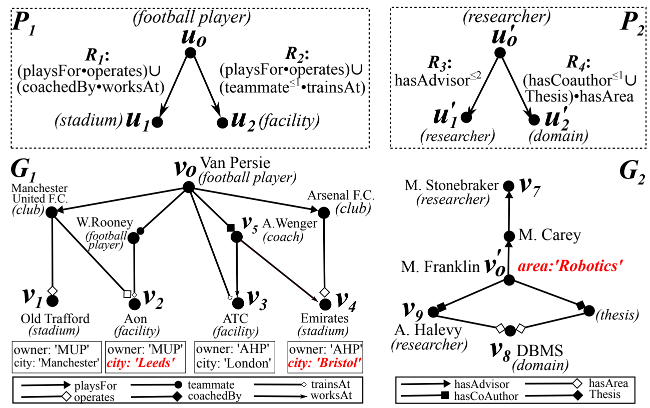


Fig. 1: Capturing erroneous attributes with regular expressions

Consider the following examples.

Example 1: ***[Capturing errors with regular expressions]*** Fig. 1 illustrates a fraction of a knowledge base G_1 about athletes. Each node may carry a type (*e.g.,* `football players`) and a set of attributes (*e.g.,* name) with values (*e.g.,* "Van Persie"). An athlete has associated career information such as the `clubs` they play for (*e.g.,* "Arsenal F.C."), their `coaches` (*e.g.,* "A. Wenger"), `stadiums` (*e.g.,* "Emirates") and training `facilities` (*e.g.,* "ATC"). There are two errors about facts of "Van Persie": the city of the stadium "Emirates" and the city of the training facility "Aon" are marked as 'Bristol' and 'Leeds', respectively.

Such errors can be identified and corrected by the following constraint posed on the neighborhood of football players: "*if a stadium and a facility relevant to the same football player from Premier League are owned by the same company, then they should locate at the same city*." Here the relevant stadiums and facilities of a football player in G_1 are (1) not explicitly encoded by direct neighbors of football players, and (2) may connect to a football player via paths with different labels. Such semantic correlation is identified by a pattern P_1 with two regular expressions below:

- R_1 = (playsFor $\cdot$ operates) $\cup$ (coachedBy $\cdot$ worksAt)
- R_2 = (playsFor $\cdot$ operates) $\cup$ (teammate$^{\leq 1}$ $\cdot$ trainsAt)

Given football player "Van Persie", R_1 specifies stadiums relevant to him as those "*either operated by his club, or those where his coach works at*". Similarly, R_2 identifies his relevant facilities as those *"operated by his club, or those where he or* *at most one of his teammate trains at"*. Given the correct location "ATC" (resp. "Old Trafford"), the city 'Bristol' of stadium "Emirates" (resp. 'Leeds' of facility "Aon") should be corrected as 'London' (resp. 'Manchester').

Another example from academic networks (*e.g.,* Mathematics Genealogy) suggests that a researcher's area can be determined by checking value constraints from his *academic genealogy* (G_2 in Fig. 1). A `researcher` "M. Franklin" has a wrong area 'Robotics'. This can be detected by enforcing the following constraint: *"a researcher should have a consistent area with (1) an area his thesis is in or his close coauthors (within 1 hop) have, and (2) the area is also shared by his close advisors (within 2 hops).* The constraint is specified by pattern P_2 with regular expressions R_3 and R_4, and identifies a correct domain "DBMS" for the entity "M. Franklin". □

These examples suggest the need to condition the integrity and value constraints with semantic associations. Such semantic associations from *e.g.,* "career" or "academic genealogy" can be characterized as multirelational regular paths [3], [6].

While desirable, repairing the errors captured by violations of such constraints is nontrivial.

Example 2: ***[Repairing under constraints with regular expressions]*** The semantic association specifying relevant stadiums and facilities can be expressed by a tree pattern P_1 shown in Fig. 1. One solution is to update the city of "Emirates" from 'Bristol' to 'London', and update the city of entity "Aon" from 'Leeds' to 'Manchester'. Another solution updates the city 'London' of "ATC" to 'Bristol', and the city 'Manchester' of "Old Trafford" to 'Leeds', and the owner company 'AHP' to *e.g.,* 'BCFC. Ltd' for both "ATC" and "Emirates". One may prefer the first repair that makes fewer changes to the attribute values, given that the second repair may modify values that are highly confident to be correct or incur larger editing cost. Moreover, new inconsistencies may be introduced due to the modification of the node attributes. □

The above examples call for constraint models that can incorporate semantic associations captured by regular expressions to detect erroneous entities in multirelational graphs, as well as effective repairing algorithms.

Contribution. This paper studies feasible constraints and algorithms to repair entity information in large graphs.

(1) We propose *star functional dependencies* (StarFDs), a class of neighborhood constraints to detect erroneous attribute values of entities in multirelational graphs (Section II). A StarFD incorporates a *star pattern* that encodes a class of conjunctive *regular path queries* to locate semantically associated neighbors of entities, and enforces value constraints over these entities. StarFDs achieve a balance between expressiveness and computational cost for error detection: the validation problem is tractable for StarFDs. We present an algorithm to detect errors with StarFDs. In addition, we show that (1) the satisfiability problem of StarFDs is NP-complete; and (2) the implication problem is coNP-complete.

(2) We approach *minimum cost repairs* to correct errors (Section III). We introduce a cost model for repairs, and formulate an entity repair problem under StarFDs. Given a graph G and a set of StarFDs Σ, it is to compute a new graph G' that satisfies the StarFDs Σ and incurs a minimum editing cost. Although the validation of StarFDs is tractable, the entity repair under StarFDs is NP-complete, and is hard to approximate. Despite the hardness, we show that entity repair is within reach in practice for large G.

(3) We introduce an entity repairing framework (Section IV and V). The framework partitions the inconsistencies to components that can be independently repaired, and discriminately computes *optimal*, *approximate* and *cost-bounded* solutions for each component respectively whenever possible, by detecting corresponding conditions that ensure the existence of such repairs. All these algorithms incur a time cost determined only by the size of constraints $|\Sigma|$, and a bounded neighborhood of erroneous entities. These ensure practical applications of our repairing algorithms.

(4) We experimentally verify the effectiveness and efficiency of our graph repairing algorithms, using real-world graphs from diverse categories (Section VI). We find that erroneous entities can be efficiently captured and repaired by enforcing StarFDs. For example, for Yago with 4.4 million edges, it takes up to 1.6 seconds to identify erroneous entities, and 4.4 seconds to compute repairs respectively, even for top frequent types such as `Person`. The repairs in turn improve the effectiveness of knowledge base completion [25], [27], where correct neighborhood information plays a critical role.

Related Work. We categorize the related work as follows.

Graph data dependencies. Integrity constraints and data dependencies such as functional dependencies (FDs) have been extended to detect inconsistencies in graphs [16]. These constraints incorporate subgraph isomorphism to identify the fraction of graphs the value constraints that should hold. For example, a graph functional dependency (GFD) φ with subgraph pattern Q enforces value constraints on node attributes identified by Q via subgraph isomorphism [16]. Note that semantic associations that help identify attribute errors may not be easily captured by strict subgraph isomorphism. Moreover, error detection using *e.g.,*GFDs is already coNP-complete [16] (where an error is defined as a subgraph isomorphism), making repairing framework inherently expensive. StarFDs incorporate regular expressions to capture semantic associations for feasible error detection. It permits tractable error detection processes, striking a balance between expressiveness and repairing cost. Repairing algorithms are also not addressed in these work.

Constraint-based Repairing. Computing (optimal) repairs has been studied to satisfy given FDs [7], [20] and its variants [9]. These methods repair relational data by minimally modifying tuples. NADEEF [10] compiles constraints into logic operators and uses MAX-SAT solvers to minimize the editing cost. It repairs RDF tuples by recasting graphs to relational encoding and enforces conventional constraints, instead of capturing and repairing errors characterized by semantic associated entities specified via multirelational paths.

Repairing XML data [17], [29] conforms to XML schema with minimum cost. [17] assigns the nodes in XML subtrees with reliability values (true or false) and updates unreliable content of elements. [29] constructs a conflict hypergraph to encode violations, where each node is a value and each hyperedge is formed by a set of values violating a FD. It then resolves the violations by value modifications. Graph repairing has been recently studied under constraints defined by subgraph isomorphism [8], [15]. GRRs [8] compute graph repairs by enforcing changes that are explicitly encoded by two subgraph patterns. GQRs [15] assumes reliable ground truth, and deduces a certain fix of graphs. Vertex label repairs [28] apply constraints that state which labels are allowed for a node, and focuses on computing a minimum relabeling of nodes to satisfy the label constraints.

Our work differs from prior work in the following. (1) In contrast to schema-level XML repairing [17], [29], we focus on repairing instance-level errors in general multirelational graph data. (2) We develop algorithms that repair erroneous attribute values enforced by StarFDs via regular path queries, beyond updating node labels [28]. (3) StarFDs do not require strong constraints that encode topological change as in [8]. Our repairing framework computes repairing process under minimum editing cost model by enforcing StarFDs. This is also very different from [15] that refers to reliable ground truth. We provide algorithms with guarantees on repairing quality in terms of graph editing cost and constraint satisfiability. These are not addressed by prior work.

II. Neighborhood Constraints

A. Star Constraints: A characterization

Graphs. We consider directed, attributed graphs $G = (V, E, L, f_A)$, where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. Each node $v \in V$ (resp. edge $e \in E$) has a label $L(v)$ (resp. $L(e)$) from a finite alphabet τ. For each node $v \in V$, a function f_A assigns a *tuple* $f_A(v)$ to v, which is a sequence of attribute-value pairs $\{(v.A_1, a_1), \ldots (v.A_n, a_n)\}$, where $(v.A_i, a_i)$ ($i \in [1, n]$) represents that the node attribute $v.A_i$ has a constant value a_i. The *active domain* of G, denoted as $\mathsf{adom}(G)$, is a finite set of values of $v.A$ in G, with v ranging over V and A ranging over all attributes of v.

In practice, the label $L(v)$ (resp. $L(e)$) may encode the type (*e.g.,* `football player` in G_1, Fig. 1) of an entity v (resp. relation name of edge e (*e.g.,* `coachedBy`)); and the function f_A specifies its properties (*e.g.,* v.league = 'EPL'), as seen in property graphs [2], knowledge bases [12] and social networks [21]. We shall also use the following notations. (1) A path ρ in G is a sequence of edges $e_1 = (v_1, v_2)$, $e_2 = (v_2, v_3)$, …, $e_n = (v_n, v_{n+1})$. The length of ρ refers to the number of edges in the sequence. (2) The label of ρ (denoted as $L(\rho)$) is the concatenation of all the edge labels following the sequence, *i.e.,* $L(\rho) = L(e_1) \cdot L(e_2) \cdots L(e_n)$.

We next introduce a class of *star patterns* to characterize semantically associated neighbors of an entity.

Star patterns. A star pattern $P(u_o) = (V_P, E_P, L_P, f_R)$ is a single rooted two-level tree with a set of pattern nodes V_P (resp. pattern edges E_P). (1) V_P consists of a *center* u_o, and a set of leaf nodes $V_P \setminus \{u_o\}$. Each node $u \in V_P$ has a label $L_P(u)$. (2) For each leaf node u_i in V_P and each edge $e_p = (u_o, u_i)$, the function f_R assigns a *regular expression* $f_R(e_p)$ defined by a fragment of regular expressions below:

$$R ::= l | l^{\leq k} | R \cdot R | R \cup R$$

where l is either an edge label from an alphabet τ, or a wildcard '_' that stands for any label in τ. $l^{\leq k}$ denotes the concatenation of no more than k occurrences of label l (k is an integer and $k \geq 1$). $R \cdot R$ (resp. $R \cup R$) denotes the concatenation (resp. disjunction) of regular expressions. We denote the language defined by the expression R as $\mathcal{L}(R)$, *i.e.,* all the strings that can be parsed by R.

Star matches. We use the following notations. (1) A node v in G is a *candidate* of a pattern node u in $P(u_o)$, denoted as $v \sim u$, if $L(v) = L_P(u)$. A pair of nodes (v_o, v) in G is a *candidate* of a pattern edge $e_p = (u_o, u)$, denoted as $(v_o, v) \sim e_p$, if $v_o \sim u_o$ (resp. $v \sim u$), and there exists a path ρ from v_o to v, such that $L(\rho) \in \mathcal{L}(f_R(e_p))$. (2) The *matches* of the center u_o, denoted as $P(u_o, G)$, contains all the candidates v_o of u_o, such that for every edge $e_p \in E_P$, there exists a node v such that $(v_o, v) \sim e_p$.

A *star match* at a match v_o of u_o ($v_o \in P(u_o, G)$), denoted as $P(G, v_o)$, refers to the maximum set $\{(v_o, v) | (v_o, v) \sim e_p, e_p \in E_P\}$. Moreover, given a pattern node u in $P(u_o)$, the *matches* of u at v_o, denoted as $P(u, G, v_o)$, refers to the node set $\{v | (v_o, v) \sim (u_o, u), (v_o, v) \in P(G, v_o)\}$. The *match set* of a star pattern $P(u_o)$ in G, denoted as $P(G)$, refers to the set of all the star matches, *i.e.,* $P(G) = \bigcup_{v_o \in P(u_o, G)} P(G, v_o)$.

Intuitively, star matches identify the matches of the "center" entity u_o along with their semantically associated neighbors. Such association is captured by regular path queries [3], [6].

Example 3: Fig. 1 illustrates a star pattern P_1 (resp. P_2) centered at `football player` (resp. `researcher`). P_1 specifies relevant `stadium` and `facility` of each entity that matches `football player`, via paths that satisfy R_1 and R_2, respectively. The table below illustrates relevant entities specified by P_1.

notation	match set
$P_1(u_o, G_1)$	$\{v_o\}$
$P_1(G_1, v_o)$	$\{(v_o, v_1), (v_o, v_2), (v_o, v_3), (v_o, v_4)\}$
$P_1(u_1, G_1, v_o)$	$\{v_1, v_4\}$
$P_1(u_2, G_1, v_o)$	$\{v_2, v_3\}$
$P_1(G_1)$	$\{(v_o, v_1), (v_o, v_2), (v_o, v_3), (v_o, v_4)\}$

Similarly, the matches of P_2 specify relevant `researchers` (*e.g.,* "A. Halevy" (v_9)) and `domains` (*e.g.,* "DBMS" (v_8)) for a specified `researcher` (*e.g.,* "M. Franklin" (v'_o)). $\square$

Remarks. We do not require the paths that match $e_p \in E_P$ to be simple paths. This is to avoid excluding relevant entities reachable by cycles, which are commonly found in *e.g.,* social communities with mutual relations [24]. We also do not include Kleene stars to exclude "weak" semantic association and irrelevant entities via arbitrarily long paths [36], which may have little contribution to identify erroneous attributes.

We now introduce *star functional dependencies*, incorporating star patterns and value constraints.

Star constraints. A *star functional dependency* (StarFD) is in the form of

$$\varphi = (P(u_o), X \rightarrow Y, \mu)$$

where (1) $P(u_o)$ is a star pattern with a center u_o; (2) μ is a function to assign a unique variable x_u to each node $u \in V_P$; and (3) X and Y are two sets of literals defined over the set of variables assigned by μ. Each literal can be either (a) a constant literal $x_u.A = a$, where a is a constant, or (b) a variable literal $x_u.A = x_{u'}.A'$, where A and A' may refer to a node attribute or specifically the label of u and u', respectively. When A refers to a node label, $x_u.A$ refers to $L(u)$.

We simply denote x_u as u and denote $(P(u_o), X \rightarrow Y, \mu)$ as $(P(u_o), X \rightarrow Y)$ when the context is clear.

Semantics. We first characterize the satisfiability of literals. Given a star match $P(G, v_o)$ and a literal l, we say $P(G, v_o)$ *satisfies* l, denoted as $P(G, v_o) \models l$, if the following holds:

- If l is a constant literal $u.A = c$, then for every match v in $P(u, G, v_o)$, $v.A = c$;
- If l is a variable literal $u.A = u'.A'$, then for every match $v \in P(u, G, v_o)$, there exists a match $v' \in P(u', G, v_o)$, such that $v.A = v'.A'$, or vice versa.

We say $P(G, v_o)$ satisfies X, denoted as $P(G, v_o) \models X$, if (1) $P(G, v_o) \neq \emptyset$, and (2) $P(G, v_o) \models l$ for every literal $l \in X$. $P(G, v_o) \models Y$ is defined similarly.

Given a graph G and a StarFD $\varphi = (P(u_o), X \rightarrow Y)$, we say G *satisfies* φ, denoted as $G \models \varphi$, if for every star match $P(G, v_o)$ centered at a node $v_o \in P(u_o, G)$, if $P(G, v_o) \models X$, then $P(G, v_o) \models Y$. In other words, φ enforces value constraints Y on the attributes of nodes that are semantically associated to a match v_o of u_o and satisfy condition X.

A graph G satisfies a set of StarFDs Σ, denoted as $G \models \Sigma$, if $G \models \varphi$ for every $\varphi \in \Sigma$. It is *consistent w.r.t.* Σ if $G \models \Sigma$.

Example 4: The constraint that uses star pattern P_1 (Fig. 1) to identify location errors can be expressed by a StarFD $\varphi_1 = (P_1(u_o), X_1 \rightarrow Y_1)$, where X_1 contains two literals l_1: u_o.league = 'EPL' and l_2: u_1.owner = u_2.owner, and Y_1 contains a single literal u_1.city = u_2.city. Similarly, a StarFD $\varphi_2 = (P_2(u_o), X_2 \rightarrow Y_2)$ captures errors in research domains in G_2, where X_2 contains a literal l'_1: u'_1.area = u'_2.area, and Y_2 contains a literal l'_2: u'_0.area = u'_2.area.

One can verify the following. (1) As v_o.league = 'EPL', and v_1.owner = v_2.owner (resp. v_3.owner = v_4.owner), $P(G, v_o) \models X_1$. (2) As there does not exist a match v' in $P_1(u_2, G, v_o)$, such that v_1.city = v'.city, $P_1(G_1, v_o) \not\models Y_1$. Thus, $P_1(G_1, v_o) \not\models \varphi$. Similarly, $P_2(G_2, v'_o) \not\models \varphi_2$. □

symbols	notations
G	a graph $G = (V, E, L, f_A)$
$P(u_o)$	a star pattern with a center node u_o
$P(u_o, G)$	the matches of center node u_o of P in graph G
$P(G, v_o)$	a star match at a match v_o of center u_o
$P(u, G, v_o)$	the matches of u in $P(G, v_o)$ at node v_o
φ, Σ	StarFD $\varphi = (P(u_o), X \rightarrow Y)$; Σ is a set of StarFDs
$(P(G, v_o), \varphi)$	a consistent (resp. inconsistent) pair if $P(G, v_o) \models \varphi$ (resp. $P(G, v_o) \not\models \varphi$)
$\mathcal{I}(\varphi, G)$ (resp. $\mathcal{I}(\Sigma, G)$)	inconsistencies under a StarFD φ (resp. Σ)

Table I: Notations

We consider *nontrivial* StarFDs in a normal form that (a) Y contains a single literal, (b) $X \neq \emptyset$, and $Y \notin X$. Our results can be easily extended to lift these assumptions (see [1]).

Inconsistencies. We now characterize errors in terms of violations of StarFDs. Given graph G and a StarFD $\varphi = (P(u_o), X \rightarrow Y)$, an *inconsistency* is a pair $I = (P(G, v_o), \varphi)$, such that $P(G, v_o)$ is a star match at node v_o, $P(G, v_o) \models X$ and $P(G, v_o) \not\models Y$. That is, (1) if Y is a constant literal $u.A = c$, then there exists no match $v \in P(u, G, v_o)$ such that $v.A = c$; or (2) if Y is a variable literal $u.A = u'.A$, then there exists a match $v \in P(u, G, v_o)$ such that no match $v' \in P(u', G, v_o)$ satisfies $v.A = v'.A'$, or vice versa. Otherwise, $(P(G, v_o), \varphi)$ is a *consistent pair*. For example, given StarFD φ_1 and star match $P_1(G_1, v_o)$ at node v_o in G_1 (Example 4), $(P_1(G_1, v_o), \varphi_1)$ is an inconsistent pair. Similarly, $(P_2(G_2, v'_o), \varphi_2)$ is an inconsistent pair.

The *inconsistencies* under φ, denoted as $\mathcal{I}(\varphi, G)$, refer to the set of all the inconsistent pairs $(P(G, v_o), \varphi)$ in G. The inconsistencies under StarFDs Σ are similarly defined as $\mathcal{I}(\Sigma, G) = \bigcup_{\varphi \in \Sigma} \mathcal{I}(\varphi, G)$.

The main notations of this paper are summarized in Table I.

B. Fundamental Problems

We next study three fundamental problems for StarFDs. The *validation* analysis identifies inconsistencies under StarFDs to be repaired. The *satisfiability* analysis helps us decide whether a repair exists under StarFDs. The *implication* analysis reduces redundant constraints that can be already implied.

Validation. Given a set of StarFDs Σ and a graph G, the *validation problem* for StarFDs is to decide whether $G \models \Sigma$. A validation algorithm of StarFDs can be easily extended to a procedure that computes all inconsistencies that violate Σ, which is a first step for computing repairs.

We have good news for StarFDs.

Theorem 1: StarFDs *validation is in* PTIME. □

As a constructive proof of Theorem 1, we present an algorithm errorDetect to compute $\mathcal{I}(\Sigma, G)$. Given Σ and graph G, errorDetect performs two steps. (1) For each StarFD $\varphi = (P(u_o), X \rightarrow Y)$, errorDetect initializes and evaluates a *conjunctive regular path query* $Q(u_o) = \bigwedge_{i=1}^{n} Q_i(u_o)$. For each edge $e_{p_i} = (u_o, u_i) \in E_P$ $(i \in [1, n])$, it initializes a regular path query Q_i that returns all the node pairs $(v_o, v) \sim e_i$

in G. It then invokes a procedure StarMatch to compute the set of star matches $P(G)$. The procedure StarMatch follows regular path query evaluation [33] to construct a query automata and perform consecutive regular reachability tests guided by the automata (see details in [1]). For $P(u_o)$ with n pattern edges, the matches of u_o is computed as $P(u_o, G)$ = $\bigcap_{i=1}^{n} P_i(u_o, G)$. For each star match $P(G, v_o)$, it checks whether $P(G, v_o) \models X$ and $P(G, v_o) \not\models Y$. If so, it adds $(P(G, v_o), \varphi)$ to $\mathcal{I}(\Sigma, G)$.

The algorithm errorDetect correctly computes (at most $\mathsf{card}(\Sigma)|V|$ star matches and inconsistencies, in $O(\mathsf{card}(\Sigma)|V| + |V|(|V| + |E|))$ time. Here $\mathsf{card}(\Sigma)$ refers to the number of StarFDs in Σ. errorDetect validates whether $G \models \Sigma$, by testing if $\mathcal{I}(\Sigma, G) = \emptyset$. Theorem 1 thus follows.

Satisfiability. Given a set Σ of StarFDs, a graph G is a *model* of Σ, if (1) $G \models \Sigma$, and (2) for each StarFD $(P(u_o), X \to Y)$, $P(G) \neq \emptyset$. The satisfiability problem of StarFDs is to decide whether there exists a model of a given set of StarFDs Σ.

Our first result shows that the satisfiability of StarFDs, unlike its counterpart for GFDs (coNP-hard), is still in NP.

Theorem 2: StarFDs *satisfiability is* NP-*complete.* □

Proof sketch: We develop an NP algorithm that guesses a small model G for Σ, and check whether $G \models \Sigma$, in polynomial time. To see the lower bound, we construct a reduction from the satisfiability problem of conditional functional dependencies, which is shown to be NP-hard [13]. We provide the detailed proof in [1]. □

Implication. Given a set of StarFDs Σ and a StarFD φ, the *implication problem* is to decide whether Σ implies φ, denoted as $\Sigma \models \varphi$, *i.e.,* for every graph G, if $G \models \Sigma$, then $G \models \varphi$.

Theorem 3: StarFDs *implication is* coNP-*complete.* □

Proof sketch: We show that deciding $\Sigma \not\models \varphi$ is NP-complete. For the upper bound, we present an NP algorithm that guesses a mapping h from each edge e'_R in a StarFD $\varphi' \in \Sigma$ to an edge e_R in φ such that e'_R and e_R preserve node labels, and preserve equivalent regular languages. For the lower bound, we construct a reduction from the *non-equivalence problem* of two regular expressions without Kleene star, which is NP-complete (cf. [19]). The detailed proof is in [1]. □

Remarks. We consider StarFDs-based error detection as a more efficient option but also compatible with graph functional dependencies (GFDs) [16]. (1) StarFDs capture semantically associated entities with regular path queries. This supports more flexible error identification via indirect connections with heterogeneous edges. (2) Error detection using GFDs is coNP-hard [16], and the inconsistencies defined by subgraph isomorphisms may "overlap" and specify the same erroneous entities for a single GFD. StarFDs identifies at most $\mathsf{card}(\Sigma)|V|$ inconsistencies in polynomial time. The star matches can further be inspected under GFDs and other constraints. We defer StarFDs with general patterns to future work.

III. Entity Repairing

We now formalize entity repairing under StarFDs.

Repairs. Given a set of StarFDs Σ and a graph G such that $G \not\models \Sigma$, a *repair* is a graph $G' = G \oplus O$, such that $G' \models \Sigma$, *i.e.,* $\mathcal{I}(\Sigma, G') = \emptyset$. Here O refers to a set of *single updates* applied to ($\oplus$) G . Each single update (or simply "update") $o \in O$ is a triple $(v.A, a, c)$, where v is a node in G, $(v.A, a) \in f_A(v)$, *i.e.,* a is the value of the node attribute $v.A$ in G, and c is a constant ($c \neq a$) that replaces a.

We characterize repairs with two practical specifications.

Coping with incomplete graphs. The real value of a node attribute $v.A$ may not be already seen in G due to incompleteness [25], [27] or new constant enforced by StarFDs. Following conventional data cleaning that uses "marked nulls" [18], we allow an *update* $o = (v.A, a, c)$ to set value c as *either* (1) a constant $c \in \mathsf{adom}$, where adom is the union of $\mathsf{adom}(G)$ (Section II-A) and the set of constants appeared in the literals from Σ, *or* (2) a variable v_c from an infinite set $\mathcal{V}$, which stands for a constant not in adom, encoding a "missing value".

A repair G' with variables v_c allows the suggestion of (cheap) consistent graphs under Σ; the variables can be later inferred via *e.g.,* graph completion, as suggested by [25].

Partial repairs. A *partial repair* of G *w.r.t.* inconsistencies $\mathcal{I}$, denoted as $G'^{\mathcal{I}}$, is a graph where for each inconsistency $I = (P(G, v_o), \varphi) \in \mathcal{I}$, $P(G'^{\mathcal{I}}, v_o) \models \varphi$. We shall use partial repairs to capture the dynamic process of our repair algorithms. Clearly, a partial repair $G'^{\mathcal{I}}$ is a repair under Σ when $\mathcal{I} = \mathcal{I}(\Sigma, G)$.

We consider updates to attribute and type values only, and defer the study of more complex cases that involve edge manipulation (*e.g.,* edge insertions and deletions) in future work due to their impact to both topology and value constraints.

Minimum Repairs. To measure the quality of repairs, we approach *minimum repairs*, a common method to suggest repairs by minimally modifying the original database [20]. We introduce a cost model for repairs.

Consider a repair $G' = G \oplus O$ under Σ. For each node v in G, let v' be its updated counterpart in G'. Given an attribute A, the *value distance* between v and v' *w.r.t.* A is defined as

$$\mathsf{dist}(v.A, v'.A) = \begin{cases} \mathsf{dist}(a, c, v.A) & c \in \mathsf{adom} \\ 1 & c = v_c, v_c \in \mathcal{V} \end{cases}$$

where $\mathsf{dist}(a, c, v.A)$ is a function that computes a normalized distance between constants a and c by update $o = (v.A, a, c)$.

The function $\mathsf{dist}(a, c, v.A)$ can be Levenshtein distance [11], semantic distance [36] or Euclidean distance [30], measuring distance for strings, class labels or numerical values, respectively. The distance can also be weighted by *e.g.,* confidence of correctness of value a. A higher score indicates a larger cost of a being replaced.

The distance between two nodes v and v' is defined as

$$\mathsf{dist}(v, v') = \sum_{A \in f_A(v)} \mathsf{dist}(v.A, v'.A)$$

The *repair cost* of $G' = G \oplus O$, simply denoted as $c(O)$, is naturally defined as the total editing cost of all node tuples that are updated by O. It is computed as

$$c(O) = \sum_{v \in V} \mathsf{dist}(v, v')$$

Example 5: Consider the inconsistency $(P_1(G_1, v_o), \varphi_1)$ in Example 4. One repair may apply $O_1 = \{o_1, o_2\}$ to G_1, where o_1 = (v_4.city, 'Bristol', 'London') and o_2 = (v_2.city, 'Leeds', 'Manchester'). Another repair aplies $O_2 = \{o_3\}$, where o_3 = (v_o.league, 'EPL', v_c), and v_c is a "marked null" variable. Assume dist('Bristol','London', v_4.city) is 0.2, and dist('Leeds', 'Manchester', v_2.city) is 0.3, then $c(O_1) = 0.5$. The update o_3 has the highest cost 1.0, due to *e.g.,* higher confidence that 'EPL' is correct, or due to a large editing cost. Thus O_1 is preferred due to smaller total repair cost. □

We state the *minimum entity repair* problem as follows:

- Input: a graph G, a finite set of StarFDs Σ.
- Output: a repair $G' = G \oplus O$ under Σ (or equivalently, a set of updates O), such that $c(O) \leq c(O')$ for any other repair $G \oplus O'$ of G obtained by O' under Σ.

The decision version of this problem is to decide whether there exists a repair G' with a cost $c(O) \leq B$, for a cost budget B. Despite that error detection is tractable, computing optimal repairs is nontrivial.

Theorem 4: *Given a graph G and a set of* StarFDs Σ*, the entity repair problem is (1)* NP*-complete for the decision version, and (2)* APX*-hard, even when Σ involves only constant literals or only variable literals.* □

The hardness can be shown by a reduction from the minimum dominating set problem, which is inapproximable for $c \log(n)$ for some constant $c > 0$ and input size n [5]. We present the detailed proof in [1].

Remarks. We do not simply exclude "marked nulls" or updates that violate X literals from possible repairs. Such repairs subsume a condition table defined on repaired entities, following constraint-based repairing [9], [18], [20]. The possible updates can be suggested to users for further refinement. Nevertheless, one can penalize undesired updates with cost functions *e.g.,* setting the cost of "null" updates to 1.0. Our repair framework (Section IV) can be readily extended to produce "not null" or ''enforce Y literals only" repairs [18].

IV. Computing Minimum Repairs

A major challenge of entity repairing is to cope with new inconsistencies during the repairing process. We introduce a feasible repairing framework, denoted as StarRepair. Given a graph G and a set of StarFDs Σ, StarRepair computes a set of updates O to induce a repair $G' = G \oplus O$. It adopts a *dichotomous* approach, to (1) detect and cope with cases that admit *optimal and approximate* repairs, and (2) resolve the rest inconsistencies by *cost-bounded* repairs. For all cases, it incurs a time cost determined by the size of Σ and bounded hop of star matches.

Algorithm StarRepair

Input: Graph G, a set of StarFDs Σ.
Output: A repair G' of G under Σ.

```
1.  set O := ∅; set (I(Σ, G), 𝒢) := errorDetect(G, Σ);
2.  set P_I := partition(I(Σ, G));
3.  for each CC I in P_I do
4.     set U^I := genUpdate(I);
5.     if isIsolated(I, 𝒢) then
6.        if isHyperStar(I, U^I) then
             /* computing optimal repairs */
7.           O := O ∪ optRepair(I, U^I);
          else /* computing approximable repairs */
8.           O := O ∪ apxRepair(I, U^I);
9.  induce non-isolated CCs I from P_I;
    /* compute bounded repairs for remaining CCs */
10. O := O ∪ boundedRepair(I, 𝒢) for each non-isolated I;
11. return G' := G ⊕ O.
```

Fig. 2: Algorithm StarRepair: a dichotomous approach

A. A general framework

We start with an auxiliary structure called *interaction graphs* to encode the dynamic repairing process.

Interaction Graph. We say pairs $(P(G, v_o), \varphi)$ (where φ = $(P(u_o), X \rightarrow Y)$) and $(P'(G, v'_o), \varphi')$ (where φ' = $(P'(u_o), X' \rightarrow Y')$) are *connected at node attribute* $v.A$, if there exists a node v with attribute A in G, such that (a) v is a match of a node u (resp. u') in $P(u_o)$ (resp. $P'(u_o)$), and (b) $u.A$ (resp. $u'.A'$) appears in $X \cup Y$ (resp. $X' \cup Y'$). Otherwise, they are disconnected.

An *interaction graph* $\mathcal{G}$ contains the following: (1) each node in $\mathcal{G}$ is either a consistent pair or an inconsistency $(P(G, v_o), \varphi)$, and (2) there exists an edge between two connected pairs that also carries all node attributes $v.A$ the pairs are connected at. A *connected component* (CC) in $\mathcal{G}$ is a set of inconsistencies $\mathcal{I}$ from $\mathcal{G}$, such that (1) any two inconsistencies in $\mathcal{I}$ are connected via a path of connected inconsistencies in $\mathcal{I}$, and (2) no inconsistency in $\mathcal{I}$ is connected to another inconsistency not in $\mathcal{I}$.

We say a CC $\mathcal{I}$ is *isolated*, if for every inconsistency $(P(G, v_o), \varphi) \in \mathcal{I}$, each consistent pair $(P'(G, v'_o), \varphi')$ in $\mathcal{G}$ connected with $(P(G, v_o), \varphi)$ at any node attribute $v.A$, one of the following cases holds:

Case	consistent pair $(P'(G, v'_o), \varphi')$	place of $v.A$ in $X' \cup Y'$ of φ'
(1)	$P'(G, v'_o) \models X' \wedge P'(G, v'_o) \models Y'$,	$v.A$ appears in X' but not in Y'
(2)	$P'(G, v'_o) \not\models X' \wedge P'(G, v'_o) \models Y'$,	$v.A$ appears in either X' or Y'
(3)	$P'(G, v'_o) \not\models X' \wedge P'(G, v'_o) \not\models Y'$,	$v.A$ appears in Y' but not in X'

The above condition characterizes a set of inconsistencies that do not introduce new inconsistencies when repaired.

Outline. Algorithm StarRepair (Fig. 2) uses a set $\mathcal{I}(\Sigma, G)$ to track the inconsistencies in G under StarFDs Σ, and it computes repairs by processing each CC in $\mathcal{G}$ independently.

(1) It invokes procedure errorDetect to compute $\mathcal{I}(\Sigma, G)$ and construct $\mathcal{G}$ (line 1), and invokes a procedure partition to split $\mathcal{I}(\Sigma, G)$ to a set $\mathcal{P}_{\mathcal{I}}$ of CCs (line 2). It ensures that the repairs for each CC can be *independently* computed (to be discussed).

(2) For each CC $\mathcal{I} \in \mathcal{P}_\mathcal{I}$, it invokes procedure genUpdate to generate a set of atomic updates $\mathcal{U}^\mathcal{I}$ (to be discussed). If $\mathcal{I}$ is isolated (line 5), it computes *approximate* repairs by procedure apxRepair (line 8). A special case that bears *optimal repairs* (line 6; to be discussed) is verified and processed by procedure optRepair (line 7). This repeats until all CCs are processed.

(3) For the remaining CCs that are not isolated, it invokes procedure boundedRepair (line 9-10) to compute a valid repair with repair cost as small as possible. StarRepair then returns G' by applying O (line 11).

We next introduce two procedures partition and genUpdate, followed by their properties that ensure repair quality.

Procedure partition. We partition $\mathcal{I}(\Sigma, G)$ to a set of CCs $\mathcal{P}_\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$. This can be performed by a traversal in $\mathcal{G}$, which has at most $O(\mathsf{card}(\Sigma)|V|)$ nodes (pairs), and can be readily constructed by procedure errorDetect (line 1).

Example 6: Consider graph G_1 in Fig. 1. Let v_5.residence = 'London', v_5.nationality = 'UK', v_5.league = 'EPL', and there is a node v_6 "Strasbourg" with label `city` in 'France' (not shown), where "A. Wenger" (v_5) was born in. Consider two StarFDs below. (1) StarFD $\varphi_3 = (P_3(u_3), X_3 \rightarrow Y_3)$ states that "if a coach u_o from league 'EPL' works at a stadium u_1, then u_o.residence = u_1.city." (2) StarFD $\varphi_4 = (P_4(u_4), X_4 \rightarrow Y_4)$ states that "if a coach u_o was born in a city u_1 in 'France', then u_o.nationality = 'France'." Procedure errorDetect identifies the following inconsistencies $\mathcal{I}(\Sigma, G)$.

CCs	pair	star match
CC_1	$I_1 = (P_1(G_1, v_o), \varphi_1)$	$\{(v_o, v_1), (v_o, v_2), (v_o, v_3), (v_o, v_4)\}$
	$I_2 = (P_3(G_1, v_5), \varphi_3)$	$\{(v_5, v_4)\}$
CC_2	$I_3 = (P_4(G_1, v_5), \varphi_4)$	$\{(v_5, v_6)\}$

$\mathcal{I}(\Sigma, G)$ is then partitioned into CC_1 and CC_2, since I_1 and I_2 are connected at v_3.city, but neither connects to I_3. □

Atomic updates. Given an inconsistency $I = (P(G, v_o), \varphi)$, for a literal $l \in X \cup Y$ of φ, *an atomic update w.r.t.* literal l is a set of single updates, denoted as o^l, such that $G \oplus o^l$ is a partial repair of G *w.r.t.* I, obtained by "enforcing" Y (if l is the literal in Y) or minimally "violating" $l \in X$ for $P(G, v_o)$. We define the set $\mathcal{U}^I = \{o^l : l \in X \cup Y\}$.

Procedure genUpdate **(line 4)**. Given a CC $\mathcal{I}$, genUpdate computes a set of atomic updates $\mathcal{U}^\mathcal{I} = \bigcup_{I \in \mathcal{I}} \mathcal{U}^I$. This is to prepare a "pool" of updates to repair CC $\mathcal{I}$. It computes $\mathcal{U}^\mathcal{I}$ by processing each inconsistency $I = (P(G, v_o), \varphi) \in \mathcal{I}$ and each literal l in $X \cup Y$ of φ with the following cases.

(1) l is a constant literal $u.A = c$. (a) If $l \in Y$, it enforces l by adding $o^l = \{(v.A, a, c) : v \in P(u, G, v_o)$ and $a \neq c\}$ to $\mathcal{U}^I$. (b) Otherwise ($l \in X$), for each $v \in P(u, G, v_o)$, it adds $o^l = \{(v.A, a, v_c)\}$ to $\mathcal{U}^I$, where $v_c \in \mathcal{V}$ is the variable "marked null". Each such o^l leads to violation of X if applied.

(2) l is a variable literal $u.A = u'.A'$. (a) If $l \in Y$, it finds the nodes v in $P(u, G, v_o)$ that has no node in $P(u', G, v_o)$ to satisfy l. For each such node v, it adds $(v.A, a, v'.A')$ to o^l, and finally adds o^l to $\mathcal{U}^I$, where v' ranges over the nodes in $P(u', G, v_o)$. (b) Otherwise ($l \in X$), it finds all the pairs (v, v') such that $v \in P(u, G, v_o)$, $v' \in P(u', G, v_o)$ and $v.A = v'.A'$. It creates violations of l by adding $o^l = \{(v.A, a, v_c)\}$ and $o'^l = \{(v'.A', a', v'_c)\}$ to $\mathcal{U}^I$, where v_c and v'_c are two distinct variables not seen in $\mathcal{U}^I$.

(3) In addition, for each $I = (P(G, v_o), \varphi)$ in $\mathcal{I}$ and each o^l in $\mathcal{U}^I$, genUpdate verifies if o^l is also an atomic update to I', for each inconsistency $I' = (P'(G, v'_o), \varphi')$ in $\mathcal{I}$ that are connected to I in $\mathcal{G}$. If so, it adds o^l to $\mathcal{U}^{I'}$. This captures a case that one atomic update repairs multiple inconsistencies.

Procedure isIsolated **(line 5)**. isIsolated verifies whether a given CC is isolated. For each inconsistency $I \in \mathcal{I}$, it iterates consistent pairs $(P'(G, v'_o), \varphi')$ that are connected to I in $\mathcal{G}$, and for all node attributes $v.A$ they are connected at, it verifies the three cases by the definition of isolated CCs. The above process is in polynomial time in the size of $\mathcal{G}$.

Performance guarantees. We show properties of partition and genUpdate that ensure quality guarantees of repairs.

A partial repair $G^\mathcal{I} = G \oplus O^\mathcal{I}$ is an *α-approximate partial repair* ($\alpha \geq 1$), if $c(O^\mathcal{I}) \leq \alpha \cdot c(O^{\mathcal{I}*})$, where $O^{\mathcal{I}*}$ is the partial repair of $\mathcal{I}$ with minimum cost.

Lemma 1: *If all the* CCs $\mathcal{I}_i$ *processed by* StarRepair *are isolated* CCs, *and* $G^{\mathcal{I}_i} = G \oplus O_i$ *is an α-approximate partial repair* w.r.t. $\mathcal{I}_i$, *then* $G' = G \oplus \bigcup_{i \in [1,n]} O_i$ *is an α-approximate repair of G under Σ.* □

Proof sketch: For each isolated CC, a partial repair can be obtained by applying a set of atomic updates without introducing new inconsistencies, ensured by the conditions that prevent changing any consistent pair to inconsistency via node attributes they connect at. As such, the union of *α-approximate partial repairs* for isolated CCs is a partial repair under Σ that preserves the approximation ratio α. □

Lemma 2: *Given* CC $\mathcal{I}$, *(1) for any inconsistency* $I \in \mathcal{I}$, $G'^I = G \oplus o^l$ *for any atomic update* $o^l \in \mathcal{U}^I$ *generated by* genUpdate *is a partial repair of G* w.r.t. I*; and (2) for any partial repair* $G'^\mathcal{I} = G \oplus O^\mathcal{I}$, *there exists a set of atomic updates* $\mathcal{U}'^\mathcal{I} \subseteq \mathcal{U}^\mathcal{I}$, *such that* $\bigcup_{o^l \in \mathcal{U}'^\mathcal{I}} o^l \subseteq O^\mathcal{I}$. □

We present the proof of Lemma 2 in [1]. It suffices to consider only $\mathcal{U}^\mathcal{I}$ to repair each CC $\mathcal{I}$. Let $|\Sigma| = \sum_{\varphi \in \Sigma} |\varphi|$ be an "encoding" size of Σ, where $|\varphi|$ is the total size of star pattern (including regular expressions) and the size of literals. The size of $\mathcal{U}^\mathcal{I}$ is bounded by $O(|\mathcal{I}||\Sigma||\mathsf{adom}|)$ by genUpdate, and can be generated in $O(|\mathcal{I}||\Sigma||\mathsf{adom}|)$ time. The above analysis ensures partial repairs of isolated CCs can provide repairs of G with quality guarantees (lines 5-8).

B. Approximating Optimal Repairs

We show the optimal partial repairs of $\mathcal{I}$ can be efficiently approximated for isolated $\mathcal{I}$.

Theorem 5: *There exists an $|\Sigma|^2|\mathcal{I}|$-approximation to compute a partial repair for an isolated* CC $\mathcal{I}$ *in* $O(|\mathcal{I}||\Sigma|^2 + |\mathcal{I}|(|\mathcal{I}||\Sigma|^2 + |\mathcal{I}||\Sigma|))$ *time.* □

Procedure apxRepair($\mathcal{I}$,$\mathcal{U}^{\mathcal{I}}$)

1. integer i := 1; $\mathcal{H}_1$:= constructHyper($\mathcal{I}$,$\mathcal{U}^{\mathcal{I}}$);
2. **while** $\mathcal{H}_i := (\mathcal{U}_i^{\mathcal{I}}, \mathcal{E}_i)$ **and** $\mathcal{E}_i \neq \emptyset$ **do**
3. $\quad\gamma_i := \min\{\frac{c_i(o^l)}{\mathsf{deg}_i(o^l)|}\}$ over all $o^l \in \mathcal{U}_i^{\mathcal{I}}$ and $\mathsf{deg}_i(o^l) > 0$;
4. $\quad O_i := \{o^l : o^l \in \mathcal{U}_i^{\mathcal{I}}$ and $c_i(o^l) = \gamma_i \mathsf{deg}_i(o^l)\}$;
5. $\quad O_i$:= resolveConflict(O_i);
6. $\quad\mathcal{U}_{i+1}^{\mathcal{I}}$:= refine($\mathcal{U}_i^{\mathcal{I}}, \mathcal{E}_i, O_i$);
7. $\quad\mathcal{E}_{i+1} := \mathcal{E}_i \setminus \{e : e$ is covered by $O_i\}$;
8. $\quad$**for each** $o^l \in \mathcal{U}_{i+1}^{\mathcal{I}}$ **do**
9. $\qquad c_{i+1}(o^l) := c_i(o^l) - \gamma_i \mathsf{deg}_i(o^l)$;
10. $\quad\mathcal{H}_{i+1} := (\mathcal{U}_{i+1}^{\mathcal{I}}, \mathcal{E}_{i+1})$; $i := i + 1$;
11. Set $\mathcal{U}'^{\mathcal{I}} := \bigcup_1^i O_i$; **return** $O^{\mathcal{I}} := \bigcup_{o^l \in \mathcal{U}'^{\mathcal{I}}} o^l$;

Fig. 3: **Procedure** apxRepair

We next introduce procedure apxRepair, as a constructive proof for Theorem 5. Our main idea is to build a hypergraph to capture the dependencies among the atomic updates over $\mathcal{I}$, and compute repairs by approximating a *minimum weighted constrained vertex cover* of the hypergraph.

Weighted hypergraph. Given an isolated CC $\mathcal{I}$ and a set of atomic updates $\mathcal{U}^{\mathcal{I}}$, apxRepair constructs a hypergraph $\mathcal{H} = (\mathcal{U}^{\mathcal{I}}, \mathcal{E})$, where each node $o^l \in \mathcal{U}^{\mathcal{I}}$ is an atomic update with a weight $c(o^l)$, and each hyperedge in $\mathcal{E}$ is the set $\mathcal{U}^I$ for an inconsistency $I \in \mathcal{I}$. We say a set of atomic updates $\mathcal{U}'^{\mathcal{I}} \subseteq \mathcal{U}^{\mathcal{I}}$ is a *vertex cover*, if $\mathcal{U}'^{\mathcal{I}} \cap \mathcal{U}^I \neq \emptyset$ for each hyperedge $\mathcal{U}^I \in \mathcal{E}$ (*i.e.,* $\mathcal{U}'^{\mathcal{I}}$ is a vertex cover of $\mathcal{H}$).

Forbidden pairs. To ensure that a vertex cover $\mathcal{U}'^{\mathcal{I}}$ corresponds to a valid partial repair, apxRepair introduces a special class of *forbidden edges* $E^{\neg}$, where each forbidden edge $e^{\neg}$ encodes a *forbidden pair* (o^l, o'^l) that are mutually exclusive in a valid repair, *i.e.,* only one of o^l or o'^l can coexist in $\mathcal{U}'^{\mathcal{I}}$ should it encode a partial repair. More specifically, a pair of updates (o^l, o'^l) is a forbidden pair if

(1) There are two single updates $(v.A, a, c) \in o^l$ and $(v.A, a, c') \in o'^l$, and $c \neq c'$; or

(2) There exists an inconsistency $I \in \mathcal{I}$ such that (a) $\{o^l, o'^l\} \subseteq \mathcal{U}^I \cap \mathcal{U}'^{\mathcal{I}}$, and (b) I remains to be an inconsistency in $G \oplus (o^l \cup o'^l)$.

The first case aims to forbid that two updates change a same $v.A$ to different values. The second case prevents unresolved inconsistencies after the two updates are applied.

A set of atomic updates $\mathcal{U}'^{\mathcal{I}}$ is a *constrained vertex cover* if it is a vertex cover of $\mathcal{H}$ and contains no forbidden pair. We present a *sufficient and necessary* condition to characterize partial repairs with $\mathcal{U}'^{\mathcal{I}}$.

Lemma 3: *Given an isolated* CC $\mathcal{I}$*, a graph* $G \oplus O^{\mathcal{I}}$ *is a partial repair if and only if there exists a set of atomic updates* $\mathcal{U}'^{\mathcal{I}}$*, such that* $\bigcup_{o^l \in \mathcal{U}'^{\mathcal{I}}} o^l \subseteq O^{\mathcal{I}}$*, and* $\mathcal{U}'^{\mathcal{I}}$ *is a constrained vertex cover of the hypergraph* $\mathcal{H}$*.* □

We present the detailed proof in [1]. Given Lemma 3, procedure apxRepair (illustrated in Fig. 3) approximates the minimum constrained vertex cover $\mathcal{U}'^{\mathcal{I}}$ of hypergraph $\mathcal{H}$. It (1) extends *layering technique* [32] to $\mathcal{H}$, which decomposes atomic update cost $c(o^l)$ by factorizing it with the number of hyperedges that o^l can "cover", and dynamically selects promising atomic updates over multiple layers (subgraphs) of $\mathcal{H}$, and (2) integrates *conflict resolving* in each layer to enforce the constraints.

Procedure apxRepair **(line 8 of** StarRepair**).** Given an isolated CC $\mathcal{I}$ and the set of atomic updates $\mathcal{U}^{\mathcal{I}}$, apxRepair initializes a hypergraph $\mathcal{H}_1 = (\mathcal{U}_1^{\mathcal{I}}, \mathcal{E}_1)$ (Fig. 3, line 1) by constructHyper (layer 1). It then performs two major steps at each layer i.

Updates selection (lines 3-5). apxRepair computes a set of atomic updates O_i at $\mathcal{H}_i$. For each atomic update o^l, it computes a *degree-weighted* cost $\gamma = \frac{c_i(o^l)}{\mathsf{deg}_i(o^l)}$, where $\mathsf{deg}_i(o^l)$ is the total number of hyperedges $\mathcal{U}^I$ "covered" by o^l, *i.e.,* $o^l \in \mathcal{U}^I$ (line 3). It then sets O_i of layer i as the atomic updates with smallest degree-weighted cost (line 4).

It next refines O_i by resolving forbidden pairs using a procedure resolveConflict (line 5). For each forbidden pair (o^l, o'^l) included in O_i, it removes the one with a larger $c(o^l)$, and removes all the forbidden edges adjacent to o^l. This process repeats until O_i induces no forbidden edge.

Layer construction (lines 6-10). apxRepair then refines $\mathcal{H}_i$ to $\mathcal{H}_{i+1}$ as follows. (1) It removes unpromising updates from $\mathcal{U}_i^{\mathcal{I}}$ by procedure refine($\cdot$) (line 6), which dynamically detects forbidden pairs given the selected updates in O_i, and removes updates *in the following order*: (a) resolve forbidden pairs that have one node in O_i; (b) remove atomic updates in O_i, and (c) remove atomic updates with $\mathsf{deg}(o^l) = 0$. (2) It removes all hyperedges covered by O_i (line 7). Moreover, it updates the degree weighted cost for all the refined updates (lines 8-9). $\mathcal{H}_{i+1}$ is constructed accordingly (line 10).

The above process repeats until all the hyperedges of $\mathcal{H}$ are covered (line 2). The vertex cover is $\mathcal{U}'^{\mathcal{I}} = \bigcup_1^i O_i$, and the set of updates $O^{\mathcal{I}}$ is computed as the union of all selected atomic updates $\bigcup_{o^l \in \mathcal{U}'^{\mathcal{I}}} o^l$ at each layer i and is returned (line 11).

Example 7: Continue with Example 6 and consider I_1 and I_2 of CC_1. Fig. 4 illustrates an initial hypergraph $\mathcal{H}_1$, which contains two hyperedges $\mathcal{U}^{I_1} = \{o_1^l, o_2^l, o_3^l\}$ and $\mathcal{U}^{I_2} = \{o_1^l, o_4^l, o_5^l, o_6^l\}$. Atomic updates are shown as below.

atomic updates $\mathcal{U}^I$	costs
o_1^l = {(v_4.city, 'Bristol', 'London'), (v_1.city, 'Manchester', 'London'), (v_2.city, 'Leeds', 'London')}	$c(o_1^l)$ = 1.6
o_2^l = {(v_4.city, 'Bristol', 'Leeds'), (v_3.city, 'London', 'Manchester')}	$c(o_2^l)$ = 1.2
o_3^l = {(v_0.league, 'EPL', v_c)}	$c(o_3^l)$ = 1.0
o_4^l = {(v_4.city, 'Bristol', 'London')	$c(o_4^l)$ = 0.4
o_5^l = {(v_5.residence, 'London', 'Bristol')}	$c(o_5^l)$ = 0.6
o_6^l = {(v_5.league, 'EPL', v_c)}	$c(o_6^l)$ = 1.0

apxRepair selects o_4^l first, which has the minimum degree-weighted cost $\gamma = 0.4/1$. This leads to forbidden pairs (o_4^l, o_2^l), which changes v_4.city to different values ('Leeds'), and (o_4^l, o_5^l), which leaves I_2 unresolved. Hence, o_2^l and o_5^l are removed by resolveConflict. Procedure refine then refines $\mathcal{H}_1$ as follows. (1) Remove hyperedge $\mathcal{U}^{I_2}$, which is covered by o_4^l; (2) removes zero degree nodes o_6^l; (3) updates costs: $c(o_1^l)$ = 1.6 - 0.4·1.0 = 1.2, and similarly $c(o_3^l)$ = 0.6,; and (4) builds $\mathcal{H}_2 = (\mathcal{U}_2, \mathcal{E}_2)$, where has one hyperedge $\mathcal{U}^{I_1} = \{o_1^l, o_3^l\}$ with updated costs. I_1 is then repaired by selecting o_3^l in $\mathcal{H}_2$. This yields a repair by applying $o_3^l \cup o_4^l$ with total cost 1.4. If o_1^l

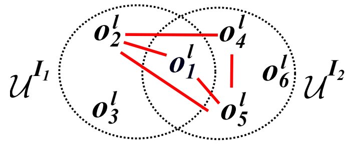


Fig. 4: Approximating optimal repairs (apxRepair, Example 7)

is changed to $\{(v_4$.city, 'Bristol', 'London'), $(v_2$.city, 'Leeds', 'Manchester') with cost $c(o^l_1)$ = 0.6, apxRepair first selects o_1 with degree-weighted cost $0.6/2$ and both $\mathcal{U}^{I_1}$ and $\mathcal{U}^{I_2}$ are covered, and the procedure stops after one iteration. $\square$

Approximation. Algorithm apxRepair correctly computes a constrained vertex cover $\mathcal{U}'^{\mathcal{I}}$ of $\mathcal{H}$ ensured by resolveConflict and refine. Given Lemma 3, $G \oplus O^{\mathcal{I}}$, where $O^{\mathcal{I}} = \bigcup_{o^l \in \mathcal{U}'^{\mathcal{I}}} o^l$ is a partial repair of G *w.r.t.* the isolated CC $\mathcal{I}$.

Let $O^{\mathcal{I}*}$ (obtained by $\mathcal{U}^{\mathcal{I}*}$), be the updates that revise G to the optimal repair *w.r.t.* Σ, $\mathcal{U}'^{\mathcal{I}*}$ be the set of atomic updates induced from the optimal constrained vertex cover for $\mathcal{H}$, and $i = t$ when apxRepair terminates. Define cost $c(\mathcal{U}^I) = \sum_{o^l \in \mathcal{U}^I} c(o^l)$. (1) We show $c(\mathcal{U}'^{\mathcal{I}}) \leq |\mathcal{U}^I| c(\mathcal{U}'^{\mathcal{I}*})$. Note that $c(o^l) = \sum_{i=1}^{t} \gamma \mathsf{deg}_i(o^l)$ if $o^l \in \mathcal{U}'^{\mathcal{I}}$; and $c(o^l) \geq \sum_{i=1}^{t} \gamma_i \mathsf{deg}_i(o^l)$ if $o^l \notin \mathcal{U}'^{\mathcal{I}}$. For each layer i, $\mathcal{U}'^{\mathcal{I}} \cap \mathcal{U}^{\mathcal{I}}_i$ (resp. $\mathcal{U}'^{\mathcal{I}*} \cap \mathcal{U}^{\mathcal{I}}_i$) is a vertex cover of $\mathcal{H}_i$. On the one hand, $c(\mathcal{U}'^{\mathcal{I}}) = \sum_{i=1}^{t} \sum_{o^l \in \mathcal{U}'^{\mathcal{I}} \cap \mathcal{U}^{\mathcal{I}}_i} \gamma_i \mathsf{deg}_i(o^l) \leq \sum_{i=1}^{t} \sum_{o^l \in \mathcal{U}^{\mathcal{I}}_i} \gamma_i \mathsf{deg}_i(o^l) \leq |\mathcal{U}^I| \sum_{i=1}^{t} \gamma_i |\mathcal{E}_i|$. On the other hand, $c(\mathcal{U}'^{\mathcal{I}*}) \geq \sum_{i=1}^{t} \sum_{o^l \in \mathcal{U}'^{\mathcal{I}*} \cap \mathcal{U}^{\mathcal{I}}_i} \gamma_i \mathsf{deg}_i(o^l) \geq \sum_{i=1}^{t} \gamma_i |\mathcal{E}_i|$. Hence, $c(\mathcal{U}'^{\mathcal{I}}) \leq |\mathcal{U}^I| c(\mathcal{U}'^{\mathcal{I}*})$. (2) As $O^{\mathcal{I}} = \bigcup_{o^l \in \mathcal{U}'^{\mathcal{I}}} o^l$, $c(O^{\mathcal{I}*}) \leq c(O^{\mathcal{I}}) \leq c(\mathcal{U}'^{\mathcal{I}})$. For hypergraph $\mathcal{H}$, $c(\mathcal{U}'^{\mathcal{I}*}) \leq c(\mathcal{U}^{\mathcal{I}*}) \leq |\mathcal{I}| c(O^{\mathcal{I}*})$. The second inequality holds because given a single update o, it can be repeatedly applied by at most $|\mathcal{U}^I||\mathcal{I}|$ atomic updates. Putting these together, $c(O^{\mathcal{I}*}) \leq c(O^{\mathcal{I}}) \leq c(\mathcal{U}'^{\mathcal{I}}) \leq |\mathcal{U}^I|^2 |\mathcal{I}| c(O^{\mathcal{I}*})$. As $|\mathcal{U}^I|$ is bounded by $|\Sigma|$, The algorithm is a $|\Sigma|^2|\mathcal{I}|$-approximation.

Complexity. apxRepair takes $O(|\mathcal{I}||\Sigma|^2)$ time to construct $\mathcal{H}_1$. There is at most $|\mathcal{I}|$ iterations. In each iteration, it takes $O(|\mathcal{I}||\Sigma|^2)$ time to select updates, and $O(|\mathcal{I}||\Sigma|)$ time to resolve forbidden pairs. The total time cost is thus in $O(|\mathcal{I}||\Sigma|^2 + |\mathcal{I}|(|\mathcal{I}||\Sigma|^2 + |\mathcal{I}||\Sigma|))$.

We present special cases for which apxRepair achieves better approximation ratio in [1] *e.g.,* $2|\Sigma|^2$.

Tractable Optimal Repairing. We also present a case when computing an optimal repair becomes tractable.

Hyperstar Updates. For an isolated CC $\mathcal{I}$, we say its atomic updates $\mathcal{U}^{\mathcal{I}}$ is a *hyperstar* [23], if for *every* two inconsistencies I and I' in $\mathcal{I}$, $\mathcal{U}^I \cap \mathcal{U}^{I'}$ equals to the same fixed O_c.

Theorem 6: *There exists an algorithm that computes the optimal partial repair in $O(|\mathcal{I}||\Sigma|)$ time for an isolated CC $\mathcal{I}$, when its atomic updates $\mathcal{U}^{\mathcal{I}}$ is a hyperstar.* $\square$

Procedure optRepair **(line 7 of** StarRepair**)**. Given an isolated CC $\mathcal{I}$ and its atomic updates $\mathcal{U}^{\mathcal{I}}$ as a hyperstar, optRepair first computes the center (common subset) O_c of $\mathcal{U}^{\mathcal{I}}$. It then compares two sets of atomic updates, both lead to partial repairs: (1) a singleton $\{o^{l*}\}$, where o^{l*} has the minimum cost in center O_c; and (2) a set of atomic updates $\mathcal{U}^{\mathcal{I}*}$, which selects a least-cost atomic update o^l_I from each set $\mathcal{U}^I \setminus O_c$ over all $I \in \mathcal{I}$. It returns the partial repair with a smaller cost.

Analysis. The optimality guarantee can be shown by contradiction. optRepair takes in total $O(|\mathcal{I}||\Sigma|)$ time. Note that it takes $O(|\mathcal{I}||\Sigma|)$ time to determine whether $\mathcal{U}^{\mathcal{I}}$ is a hyperstar. Theorem 6 thus follows. We present detailed analysis in [1].

These approximable and optimal cases are quite common: our experiments verify that up to 64% (resp. 14%) of detected inconsistencies bear approximable (resp. optimal) repairs over real multirelational graphs (see Exp-2, Section VI).

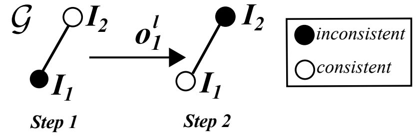


Fig. 5: Heuristic repair (boundedRepair)

V. Cost-Bounded Repairing

We next introduce an algorithm to compute repairs for non-isolated CC. Our idea is to iteratively repair CCs that connect to fewest consistent pairs (thus are less likely to introduce new inconsistencies) as "isolated" ones, and incrementally update interaction graph $\mathcal{G}$ with new inconsistencies.

Algorithm. Procedure boundedRepair (also invoked by algorithm StarRepair, line 10). maintains (a) a set of current consistent pairs C, and (b) a tunable repair budget B (set as $|\mathcal{I}|$ by default), under the intuition that the largest expected cost is $|\mathcal{I}|$ (by *e.g.,* simply repairing with "marked nulls" with cost 1.0). It iteratively performs the following. (1) Induces a maximal set of connected inconsistencies $\mathcal{I}' \subseteq \mathcal{I}$ that has the fewest adjacent consistent pairs in $\mathcal{G}$. (2) Computes an approximate (resp. optimal) partial repair for $\mathcal{I}'$ by invoking genUpdate and apxRepair (resp. optRepair), treating $\mathcal{I}'$ as an "isolated" CC. (3) Invokes a procedure incErrorDetect to incrementally detect new matches and inconsistencies in the consistent pairs that are neighbors of $\mathcal{I}$ in $\mathcal{G}$ (illustrated in Fig. 5), and updates B, $\mathcal{G}$ and $\mathcal{I}$ accordingly. It terminates when B is consumed.

Incremental error detection. The procedure incErrorDetect incrementalizes its counterpart errorDetect (Section II-B) to detect new matches and inconsistencies. An index is constructed by performing random walks in G and extracting sub-expressions that best summarize the paths. incErrorDetect then decomposes regular path queries to sub-expressions and inquires the index. The incremental error detection is quite effective: it improves the efficiency of errorDetect by 3.4 times (Section VI). We present the details in [1].

The algorithm boundedRepair guarantees to terminate with the following invariant for each atomic update o^l: (1) if a $v.A$ appeared in o^l is already repaired to "marked null", it skips o^l; or (2) o^l is applied no more than k times, for a tunable parameter k or B is consumed (see details in [1]).

VI. EXPERIMENTS

Using real-world graphs, we experimentally verify the efficiency and effectiveness of StarFD-based repairing.

Experiment settings. We used the following settings.

Datasets. We use four real-life graphs: (1) Yago[1], a knowledge graph derived from the Web, (2) Yelp[2], a business review graph with nodes as local services (*e.g.,* restaurants, plumbers, *etc*) and edges such as "likes". (3) DBP[3], a knowledge base extracted from Wikipedia, and (4) IMDb[4], a movie database with nodes such as films and actors, and relationships such as "directedBy". The datasets are summarized below.

Dataset	$\|V\|$	$\|E\|$	# node labels	# edge labels	avg. $\|f_A(v)\|$
Yago	2.1M	4.0M	2273	33	3
Yelp	1.5M	1.6M	42	20	5
DBP	2.2M	7.4M	73	584	4
IMDb	5.9M	3.2M	158K	2	3

Error generation. Following the "silver standard" [27] assumption, we consider our datasets as cleaned graphs. Following error generation benchmark [4], we injected errors to each original graph G as follows. (1) We sample $p_1\%$ of the nodes in G, and for each node v, sample $p_2\%$ of its attribute to inject error. The error rate p is computed as the fraction of the polluted node attributes to the total number of distinct node attribute $v.A$ in G. (2) For each sampled node attribute $v.A$, we randomly injected one of the three types of errors [4]: (1) misspells, which randomly select and replace up to 3 characters of the string value of $v.A$; (2) inaccuracy, which selects another value in the active domain $\mathsf{adom}(A)$ of attribute A (values of A in G), and (3) out-of-domain, which assigns a constant not in $\mathsf{adom}(A)$.

StarFD *generation.* We implemented an algorithm StarGen, to generate StarFDs from clean graphs. It selected top-k_1 frequent node labels (e.g. k_1 = 200 in Yago) as $L(u_o)$, and identified their candidates $P(u_o, G)$. Starting with a candidate node, StarGen sampled its neighbors up to a certain hop (e.g. 3 in Yago) to generate top-k_2 frequent paths (e.g. k_2 = 5 in Yago). The top-k_2 frequent paths were converted to regular expressions. Each star pattern was formed by a combination of regular expressions with a center node u_o. For each star pattern, StarGen searched the (equivalent) attribute values or node labels to generate constant and variable literals and aggregated dependencies $X \rightarrow Y$ by the combination of literals. This yields a StarFD for u_o. We discover StarFDs to cover all the polluted attributes (treated as training data), and manually verified each StarFD to ensure its correctness. We defer the discovery of StarFDs as future work.

Metric. Denote the attributes involved in inconsistencies as err, the attributes updated by a repair algorithm as err_r, and the set of correctly repaired attributes as err_t, which contain those attributes reconstructed to the truth values and do not consider "marked nulls". We report the accuracy of the repair algorithm as (1) precision Prec. = $\frac{|\mathsf{err}_t|}{|\mathsf{err}_r|}$, and (2) recall Rec. = $\frac{|\mathsf{err}_t|}{|\mathsf{err}|}$.

Algorithms. We implemented the following algorithms in Java. (1) StarRepair is the algorithm in Fig. 2 with optimized error detection incErrorDetect; (2) to evaluate the effectiveness of optimization, we implemented biBFSRepair, which applied bidirectional search to evaluate regular queries [14] without using incErrorDetect; and (3) SubIsoRepair transforms the StarFDs in Σ to a set of GFDs Σ', and follows StarRepair but uses Σ' as input constraints. For example, the StarFD φ_2 in Example 4 is converted to 6 GFDs by SubIsoRepair. To understand the impact of repairing budget, we also implemented an algorithm StarRepair-$x\%$, a budgeted variant of StarRepair that uses up to $x\%$ ($x > 0$) of the total repair cost as a budget.

All the algorithms measure the cost of an update $o = (v.A, a, c)$ with semantic distance [36], Levenshtein [11], numerical distance [30] normalized by domain range, and constant 1.0, when A refers to a label, a string attribute, a numerical attribute, and the case that c is out of domain ($c \notin \mathsf{adom}(A)$), respectively.

We conducted our experiments on Linux with Intel 2.33GHz CPUs and 256GB memory. Each experiment was run 5 times and the average results were reported.

Exp-1: Efficiency. As shown in Fig. 6(a), it is feasible to repair errors in large graphs under StarFDs. On average, StarRepair outperforms biBFSRepair and SubIsoRepair, by 3.4 times and 7.1 times, respectively. It takes on average 7 seconds for StarRepair to achieve minimum repair. StarRepair also incurs much less cost on error detection compared with SubIsoRepair. For example, StarMatch takes 2.3 (resp. 2.7) seconds to identify errors over Yago (resp. Yelp), and is 10 (resp. 41) times faster than the error detection of SubIsoRepair that performs subgraph enumeration.

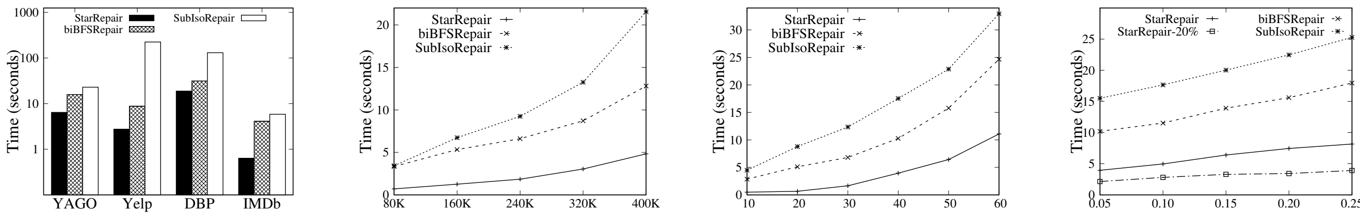
We next evaluate the impact of the following factors with default values summarized below. We use the total number of candidates of center nodes u_o, denoted as $C(u_o, G)$, instead of the graph size, as the time cost of entity repairing is more sensitive to $C(u_o, G)$.

Factor	Yago	Yelp	DBP	IMDb
# of candidates $C(u_o, G)$	320K	80K	350K	210K
# of StarFDs	60	60	30	18
error rate p	0.2	0.2	0.24	0.1
budget rate $x\%$	100%	100%	100%	100%

Varying $C(u_o, G)$. Fig. 6(b) reports the impact of $C(u_o, G)$ over Yago. (1) While all algorithms take longer time as more candidates are provided, they are quite feasible over large graphs. For example, it takes up to 4.82 seconds for StarRepair to repair entities with 400K candidates over Yago. (2) StarRepair is the least sensitive to $C(u_o, G)$ due to optimized error (re-)detection, while SubIsoRepair is the most sensitive due to subgraph isomorphism test and enumeration.

Varying # of StarFDs. Varying the number of StarFDs from 10 to 60 over Yago, Fig. 6(c) shows that all algorithms take longer time with more StarFDs due to more matches and repairs. StarRepair is the least sensitive one due to its subquery optimization.

[1]https://mpi-inf.mpg.de/yago
[2]https://www.kaggle.com/yelp-dataset
[3]https://wiki.dbpedia.org
[4]https://www.imdb.com/interfaces



(a) Overview of efficiency (b) Varying $C(u_o, G)$ (Yago) (c) Varying # of StarFDs (Yago) (d) Varying p and $x\%$ (Yago)

Fig. 6: Efficiency of entity repairing

Varying p and x%. Fixing other parameters as default, we varied the error ratio p from 0.05 to 0.25 over Yago. and tested StarRepair-20% with 20% budget. Fig. 6(d) verifies that all three algorithms take longer time when more attribute values are polluted, due to more inconsistencies to be detected and repaired. StarRepair is the least sensitive to error rate p and is on average 2 and 5 times faster than biBFSRepair and SubIsoRepair, respectively. We observe the error detection cost of all the algorithms takes more fraction in the total time for larger p, while the sub-query optimization of StarRepair reduces cost significantly. For example, the matching time takes on average 20%, 57%, and 71% of the total time for StarRepair, biBFSRepair, and SubIsoRepair, respectively. StarRepair-20% improves StarRepair on average 1.8 times due to the cost-bounded repairing.

The results over other datasets are consistent with our observation. We report more results in [1].

Exp-2: Effectiveness. Using the same settings in Exp-1, we report the effectiveness of StarRepair and SubIsoRepair. We omit the results of biBFSRepair as it has the same accuracy as StarRepair. An overview of accuracy is reported as below.

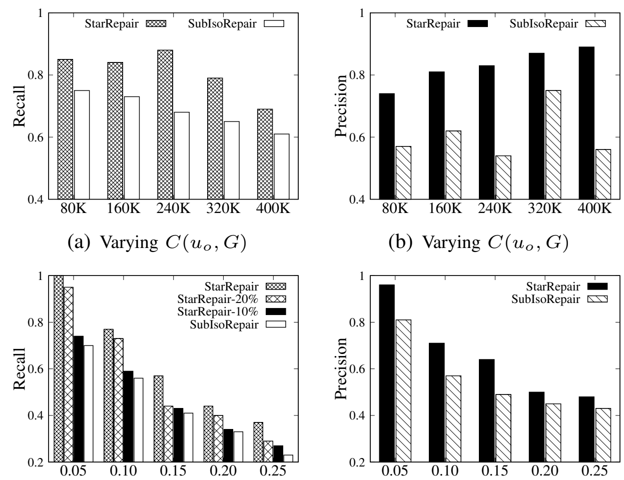
	Yago		Yelp		DBP		IMDb	
Algo.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
StarRepair	0.99	0.82	0.84	0.97	0.92	0.80	0.96	0.98
SubIsoRepair	0.94	0.72	0.82	0.82	0.67	0.65	0.87	0.97

It verifies that StarRepair outperforms SubIsoRepair by gaining 9% more in precision and 14% more in recall on average. We found that SubIsoRepair can have (redundant and overlapped) matches returned by subgraph isomorphism, making "marked nulls" v_c easier to be selected due to smaller degree-weighted cost. StarRepair is quite accurate over all datasets (Prec. = 93% and Rec. = 90% on average).

Varying $C(u_o, G)$. Fig. 7(a) and 7(b) shows the precision (resp. recall) of StarRepair is 82% (resp. 81%) on average and outperforms SubIsoRepair in all number of candidates, which indicates that our method is stable with data size.

Varying p and x%. Fig. 7(c) and 7(d) show the impact of error rate p and budget ratio $x\%$. The result shows both precision and recall decrease with larger p, as more errors are introduced by larger p. We observe the recall of StarRepair-20% (resp. StarRepair-10%) is on average 7% (resp. 15%) lower than StarRepair, because some errors remain unrepaired due to early termination. StarRepair-20% (StarRepair-10%) has precision (not shown) close to StarRepair (within 3%).

We also observe that the repairing quality benefits from more StarFDs (not shown). We report more details in [1].



(a) Varying $C(u_o, G)$ (b) Varying $C(u_o, G)$

(c) Varying p and budget B (d) Varying p and budget B

Fig. 7: Effectiveness of entity repairing (Yago)

Error distribution. We also evaluate the impact of the type of errors to StarRepair. For the three types of errors, we generate one type as major errors (70%) and the other two as minor errors (15%). When the major errors are inaccuracy,

	Yago		Yelp		DBP		IMDb	
Injected Errors	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
misspells	0.98	0.92	0.88	0.97	0.92	0.81	0.96	0.92
inaccuracy	0.94	0.80	0.83	0.92	0.90	0.79	0.91	0.91
out-of-domain	0.99	0.93	0.91	0.98	0.94	0.81	0.94	0.99

StarRepair has on average 6% lower precision and 4% lower recall, compared with misspells and out-of-domain errors. We found that the exact "true" value of inaccurate attributes is relatively more difficult to be recovered exactly by choosing repairs with the smallest editing cost. For misspells and out-of-domain errors, the editing costs are closer to either 0 or 1, respectively. This makes StarRepair be easier to prioritize updates precisely via cost models and guarantees optimality.

We also observe that isolated CCs (for apxRepair and optRepair) are quite common. For example, among all detected inconsistencies, 37%, 78%, and 54% (resp. 5%, 14% and 5%) are isolated CCs (resp. permit optimal repairs) over Yago, Yelp and IMDb, respectively (see details in [1]).

Exp-3: Case Study. Fig. 8 illustrates how StarFDs can be used to repair errors and benefit tasks such as fact checking [27].

(1) A StarFD $\varphi_5 = (P_5(u_o), X_5 \rightarrow Y_5)$ posed on DBP states that *"if a school u_o in U.S.A is located in a city u_5 by itself or through its building, or it has a campus in the suburb*

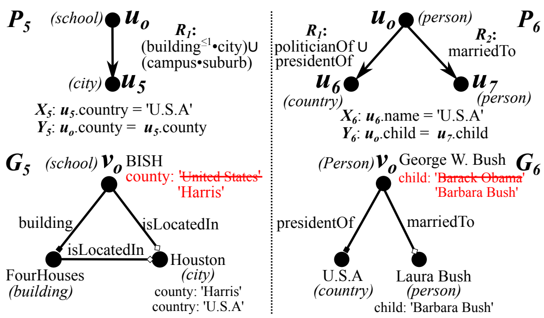


Fig. 8: Case Study

of the city, the school's county should be same as the city's county." φ_5 corrects 14 such errors in DBP. The county of British International School of Houston is wrongly associated to "United States" and is repaired to 'Harris' by φ_5. Such repair further benefits fact checking [22], [27], which predicts the missing links. A fact checking rule [22] states *"if a school u_o owns a campus u_1, which is locatedIn the city u_2, and if u_o.county equals to u_1.county, then there is likely a link <u_o, isLocatedIn, u_2>."* This rule can only be applied when the county of BISH is repaired by φ_5, which in turn identifies a missing link <BISH, isLocatedIn, Houston> in G_5.

(2) A second StarFD $\varphi_6 = (P_6(u_o), X_6 \rightarrow Y_6)$ posed on Yago states that *"If a person u_o is a politician or president of U.S.A. and married to person u_7, then the child of u_7 is also the child of u_o."* This constraint detects and repairs more than 100 errors. We illustrate one such repair in Fig. 8.

VII. Conclusions

We have proposed a class of constraints StarFDs, to identify errors with star-structured regular path patterns. We established the complexity of its fundamental problems *e.g.,* validation and satisfiability. We introduced a dichotomous repairing framework to resolve erroneous attribute values using StarFDs. Our experimental results have verified the effectiveness of StarFD techniques. One topic in future is to investigate StarFDs with general patterns and edge updates. Another topic is to discover and infer StarFDs in large graphs, and to learn high-quality and informative StarFDs with user feedback.

Acknowledgment. This work is supported in part by USDA/NIFA 2018-67007-28797, Siemens, the Department of Energy under DE-IA0000025 for UI-ASSIST project, and NSF under CPS 1932574, RAISE: C-Accel Pilot 1937143, and EPCN 1933279.

References

[1] Full version, https://penglin03.github.io/pdfs/starfds-icde20-full.pdf.
[2] R. Angles and C. Gutierrez. Survey of graph database models. *CSUR*, 2008.
[3] K. Anyanwu and A. Sheth. ρ-queries: enabling querying for semantic associations on the semantic web. In *WWW*, 2003.
[4] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with bart: error generation for evaluating data-cleaning algorithms. *PVLDB*, 2015.
[5] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.
[6] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *TODS*, 2012.
[7] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.
[8] Y. Cheng, L. Chen, Y. Yuan, and G. Wang. Rule-based graph repairing: Semantic and efficient repairing methods. In *ICDE*, 2018.
[9] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *PVLDB*, 2007.
[10] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, 2013.
[11] F. J. Damerau. A technique for computer detection and correction of spelling errors. *CACM*, 1964.
[12] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, 2014.
[13] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 2008.
[14] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.
[15] W. Fan, P. Lu, C. Tian, and J. Zhou. Deducing certain fixes to graphs. In *VLDB*, 2019.
[16] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, 2016.
[17] S. Flesca, F. Furfaro, S. Greco, and E. Zumpano. Repairs and consistent answers for xml data with functional dependencies. In *International XML Database Symposium*, 2003.
[18] P. Guagliardo and L. Libkin. On the codd semantics of sql nulls. *Information Systems*, 2018.
[19] H. B. Hunt III, D. J. Rosenkrantz, and T. G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *JCSS*, 1976.
[20] S. Kolahi and L. V. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.
[21] G. Levchuk, J. Roberts, and J. Freeman. Learning and detecting patterns in multi-attributed network data. In *AAAI Fall Symposium: Social Networks and Social Contagion*, 2012.
[22] P. Lin, Q. Song, J. Shen, and Y. Wu. Discovering graph patterns for fact checking in knowledge graphs. In *DASFAA*, 2018.
[23] Z. Lonc. Decompositions of hypergraphs into hyperstars. *Discrete mathematics*, 1987.
[24] W. E. Moustafa, A. Deshpande, and L. Getoor. Ego-centric graph pattern census. In *ICDE*, 2012.
[25] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 2016.
[26] M. T. Özsu. A survey of rdf data management systems. *Frontiers of Computer Science*, 2016.
[27] H. Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 2017.
[28] S. Song, B. Liu, H. Cheng, J. X. Yu, and L. Chen. Graph repairing under neighborhood constraints. *VLDBJ*, 2017.
[29] Z. Tan and L. Zhang. Improving xml data quality with functional dependencies. In *DASFAA*, 2011.
[30] M. Tang, R. Y. Tahboub, W. G. Aref, M. J. Atallah, Q. M. Malluhi, M. Ouzzani, and Y. N. Silva. Similarity group-by operators for multi-dimensional relational data. *TKDE*, 2016.
[31] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, 2010.
[32] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.
[33] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 2012.
[34] A. Zaveri, D. Kontokostas, M. A. Sherif, L. Bühmann, M. Morsey, S. Auer, and J. Lehmann. User-driven quality evaluation of dbpedia. In *International Conference on Semantic Systems*, 2013.
[35] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, S. Auer, and P. Hitzler. Quality assessment methodologies for linked open data. *SWJ*, 2013.
[36] G. Zhu and C. A. Iglesias. Computing semantic similarity of concepts in knowledge graphs. *TKDE*, 2017.